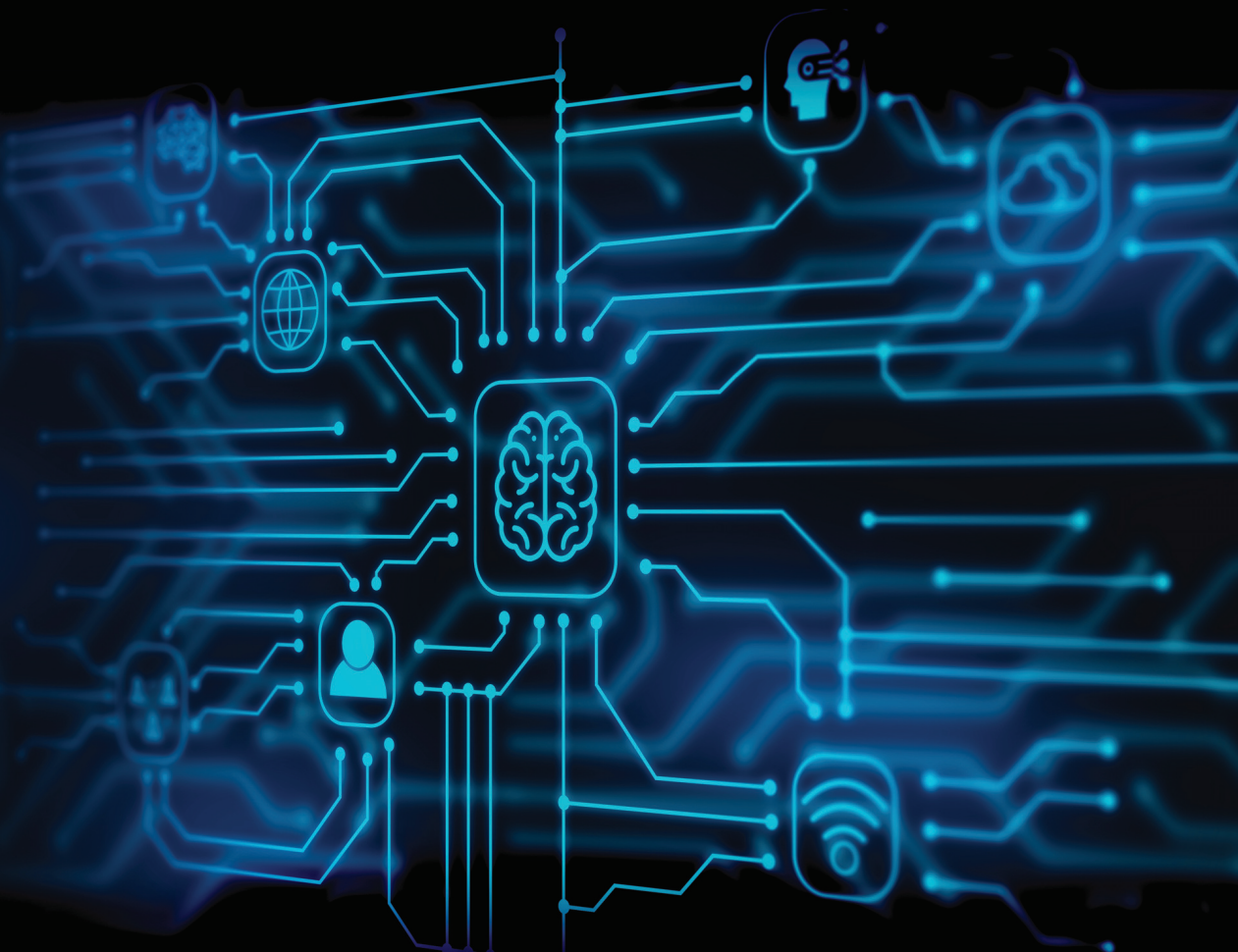


SECOND EDITION

<packt>

Hands-On Artificial Intelligence for IoT

Expert machine learning and
deep learning techniques for
developing smarter IoT systems



Dr. Amita Kapoor

Hands-On Artificial Intelligence for IoT

Second Edition

Expert machine learning and deep learning techniques for
developing smarter IoT systems

Dr. Amita Kapoor



Hands-On Artificial Intelligence for IoT

Second Edition

Copyright © 2025 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Sunith Shetty

Relationship Lead: Sanajana Gupta

Project Manager: Hemangi Lotlikar

Content Engineer: Ayushi Bulani

Technical Editor: Seemanjay Ameriya and Sweety Pagaria

Copy Editor: Safis Editing

Proofreader: Ayushi Bulani

Indexer: Hemangini Bari

Production Designer: Ponraj Dhandapani

Growth Lead: Bhavesh Amin

DevRel Marketing Coordinator: Anjitha Mural

First published: January 2019

Second edition: May 2025

Production reference: 1240425

Published by Packt Publishing Ltd.

Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK

ISBN 978-1-83546-718-3

www.packtpub.com

*For those who taught me the joy of discovery, those who continue to learn with me, and every reader
who turns these pages—your passion for knowledge fuels mine.*

– Amita Kapoor

Contributors

About the author

Dr. Amita Kapoor is an accomplished author, educator, and AI expert with over 25 years of academic and industry experience. Co-founder of NePeur and Retured, she collaborates internationally—including with the IMF—on Digital Transformation and AI applications. Her bestselling books include *Deep Learning with TensorFlow and Keras* and *Hands-On Artificial Intelligence for IoT*, First Edition. As a global educator, Amita teaches at the University of Oxford and leads international workshops. She also served as a board member for Neuromatch (2022–2024), fostering accessible AI education worldwide.

I thank my partner Narotam Singh for his unwavering support in all my endeavors. Gratitude to the Packt team for their guidance and professionalism. Special thanks to the readers of the first edition whose insightful reviews and valuable feedback greatly shaped this book. Finally, heartfelt appreciation to the technical reviewers for their thoroughness and invaluable suggestions..

About the reviewers

Sougata Pal is a passionate technology specialist playing the role of an Enterprise Architect in software architecture design and application scalability management, team building, and management. With overall 15+ years of experience, have worked with different startups to large-scale enterprises to develop their business application infrastructure which will enhance the reach to the customers. Having contributed to different open source projects on GitHub to empower the open source community. For the last couple of years, playing around with Federated Learning and CyberSecurity algorithms, to enhance the performance of CyberSecurity processes by introducing concepts of Federated Learning.

Hector Duran Lopez Velarde received a B.Che.E. from UPAEP in 1997 and an M.Sc. in Automation and Artificial Intelligence from Tecnologico de Monterrey (ITESM), Mexico, in 2000. He has worked as a controls and automation engineer for companies such as Honeywell and General Electric. He has also participated in several research projects as a technical lead. His experience in software development, industrial automation, and artificial intelligence has led him to develop complete industrial IoT solutions. He is currently working at Deloitte as a Senior Consultant in Conversational AI and Large Language Models.

Huge thanks to my wife, Yaz, and to my children, Ivana and Hector, for all their support and love.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/ds>



Table of Contents

Preface

xv

Part 1: Foundations and Basic Integrations of IoT and AI

1

Principles and Foundations of IoT and AI 3

What is IoT 101?	4	Cross-industry standard process for data mining	14
IoT reference model	6	AI platforms and IoT platforms	15
IoT platforms	7	Tools Used in this book	16
IoT verticals	8	TensorFlow	16
Big Data and IoT	9	Keras 3.0	17
Edge IoT – enhancing data efficiency and responsiveness	11	Datasets	18
Infusion of AI – Data Science in IoT	12	Summary	20

2

Data Access and Distributed Processing for IoT 21

Technical requirements	21	Data in the XLSX Format	30
Data in the TXT Format	22	Using OpenPyXL for XLSX files	30
Using TXT files in Python	22	Using pandas with XLSX files	31
CSV Format	24	Working with the JSON Format	32
Working with CSV files with the csv module	24	Using JSON files with the JSON module	32
Working with CSV files with the pandas module	28	JSON files with the pandas module	33
Working with CSV files using NumPy	29		

Handling Data in the HDF5 Format	34	NoSQL data	41
Using HDF5 with PyTables	34	HDFS for Large-Scale Data Storage and Retrieval	42
Using HDF5 with pandas	35	Using hdfs3 with HDFS	42
Using HDF5 with h5py	36	Using PyArrow's filesystem interface for HDFS	43
Object Store for Raw Data	37	Edge Data Storage and Processing	43
Working with Structured SQL Data	38	Summary	44
The SQLite database engine	39		
The MySQL database engine	40		

3

Machine Learning for IoT	45		
Technical requirements	45	Naive Bayes	64
Understanding ML and its Role in IoT	46	Gaussian naive Bayes for wine quality	65
Learning Paradigms	47	Decision Trees	67
Supervised learning	48	Decision trees in scikit-learn	69
Unsupervised learning	48	Decision trees in action	70
Prediction using Linear Regression	49	Ensemble Learning	72
Regression using deep learning	50	Voting classifier	73
Electrical power output prediction using regression	50	Bagging and pasting	74
Logistic Regression for Classification	54	Improving Your Model – Tips and Tricks	76
Cross-entropy loss function	56	Feature scaling to resolve uneven data scales	76
Classifying wine using logistic regressor	56	Overfitting	76
Classification using Support Vector Machines	59	No Free Lunch theorem	78
Maximum margin hyperplane	60	Hyperparameter tuning and grid search	78
Kernel trick	62	Summary	79
Classifying wine using SVM	62		

Part 2: Advanced AI Techniques and Their Application in IoT

4

Deep Learning for IoT	83
DL 101	84
DL – Why Now?	85
Exploring Artificial Neurons	86
Sigmoid activation function	88
Hyperbolic tangent activation function	88
Rectified linear unit activation function	89
Softmax activation function	90
Leaky ReLU activation function	90
Exponential Linear Unit activation function	91
Learning in Artificial Neural Networks	93
Multilayered Perceptrons for Regression and Classification	94
MLP as universal function approximators	97
Empirical proof of MLP's versatility – modeling non-linear functions	98
Understanding CNNs	100
Different layers of CNN	101
Milestones in CNN architectures – from LeNet to MediaPipe	105
ResNet50 for image classification	107
Recurrent Neural Networks	109
Long short-term memory	111
Gated recurrent unit	112
Exploring Autoencoders	113
Variational autoencoders	115
Introduction to OpenVINO and TinyML	116
Using Keras 3 for Hyperparameter Tuning	116
Summary	120

5

Optimization Techniques in IoT	121
Technical requirements	121
Optimization	122
Deterministic and Analytic Methods	124
Gradient descent method	124
The Newton-Raphson method	126
Natural Optimization Methods	127
Simulated annealing	127
Particle swarm optimization	128
Genetic algorithms	128
Introduction to Genetic Algorithms	129
The genetic algorithm	131
Pros and cons	133
Frameworks for EAs	135
PyGAD – GA Python library	135

DEAP – distributed EAs in Python	135	EAs and Recent Developments	144
Implementing GAs	136	Model architecture using EAs	145
Guess the word	136	Summary	148
Optimizing neural network weights with GAs	140		

6

Reinforcement Learning for IoT **151**

Technical requirements	151	Q-network	166
Introduction to RL	152	Taxi drop-off using Q-network	167
RL terminology	153	DQN to play an Atari game	172
Deep RL	155	Double DQN	178
Some successful applications of RL	155	Dueling DQN	178
Simulated Environments	157	Policy Gradients	180
Introducing Gymnasium	157	Why policy gradients?	181
Q-learning	161	The actor-critic algorithm	182
Taxi drop-off using Q-learning	163	Summary	183

7

Generative Models for IoT **185**

Introduction to Generative Models	185	KerasNLP	207
Understanding VAEs	187	Sentiment analysis using pre-trained Tiny Bert	208
Anomaly detection using VAEs	189	Fine-tuning a model in KerasNLP	209
GANs	194	Summary	210
Variants of GANs	196		
Applications of GANs	198		

Part 3: Implementing Intelligent IoT Solutions in Diverse Domains

8

Distributed Learning Using Keras 213

Technical requirements	213	The Keras Distribution API	217
What is Distributed Training?	214	Key features of the Keras 3 distribution API	217
Why distributed learning is crucial for IoT	215	Distributed Training in Action	224
Keras 3 and Distributed Learning	215	Optimizing large models with ModelParallel and LayoutMap	228
Multi-backend support in Keras 3	215	Summary	231
Distributed training strategies in Keras 3	216		

9

AI Cloud Platforms for IoT 233

Technical requirements	234	Google AutoML	250
Introduction to Cloud Computing for IoT	234	Amazon SageMaker	250
Cloud Computing Basics	235	Amazon Bedrock	251
Types of cloud services	235	Security and Privacy in IoT Cloud Platforms	252
Cloud deployment models	237	Security features of major cloud providers	253
AI Services in Cloud Platforms	239	Emerging Trends and the Future of AI Cloud Platforms for IoT	255
ML APIs and tools	240	Edge AI and hybrid cloud for IoT	256
Other useful Amazon services for IoT and AI	244	Quantum computing in IoT	256
Using the Azure Speech SDK in Python to capture speech commands	245	Summary	257
Pretrained models in the cloud	247		
Training and Deploying AI Models for IoT on Cloud Platforms	248		
Google Vertex AI	249		

10

Deep Learning for Time-Series Data from IoT		259	
Technical requirements	260	Review of RNNs and LSTMs for Time-Series Analysis	268
Introduction to Time-Series Data	260	Recap of RNN and LSTM concepts	268
What is time-series data?	260	Adjustments and considerations for IoT data	269
How is time-series data different?	260	LSTM for CO level prediction	270
The Importance of Time-Series Analysis in IoT	261	Pretrained Models for Time-Series Forecasting	276
Real-time monitoring	262	Meta's Prophet	276
Predictive analytics	262	Other pretrained models	283
Challenges in handling time-series data in IoT	262	Summary	284
Working with Large Time-Series Data Using SparkML	263	References	284

11

Leveraging AI for Visual Data from IoT		285	
Technical requirements	285	Deep learning architectures	289
A Brief Introduction to Visual Data in IoT Ecosystems	286	Recurrent and temporal convolutional networks for video analysis	301
Handling Challenges in Visual Data in IoT	286	Attention Mechanisms Capturing the Spotlight in Visual AI	303
Hardware constraints	287	ViTs – a paradigm shift in visual AI	303
Mitigating data privacy and ethical issues	288	Spatial-temporal attention – unlocking the power of video analysis	305
Examples of datasets for bias detection and fairness enforcement	288	Summary	310
Leveraging AI Techniques for Visual Data in IoT	289		

12

AI for Text, Audio and Speech Data from IoT 311

Technical requirements	311	Working with Audio Data in IoT	331
Introduction to Text, Audio, and Speech Data	312	Audio processing and feature extraction	331
Fundamentals of Text Data Processing	313	Audio command recognition	331
Data collection and preprocessing	313	Speech recognition and transcription	332
Encoding text	317	Voice assistants and IoT integration	332
Example implementation	318	Using audio commands to control an IoT device	333
NLU pipeline implementation using a transformer	324	Summary	336

Part 4: Applying AI and IoT in Real-World Scenarios

13

AI for Personal and Home IoT 339

Technical Requirements	339	Privacy Concerns in Personal IoT	361
Wearable and Fitness Devices	340	Case study 1: The smart camera breach (Ring cameras)	361
AI integration in wearables for health insights	340	Case study 2: The smart thermostat data leak	362
Edge computing for real-time health monitoring	342	Case study 3: Strava military base exposure	362
Analyzing wearable data with Python	343	Case Study 4: IoT data breach (ORVIBO Smart Home)	362
Smartwatch Health Monitoring in Action	355	Getting Started with Your Own Smart IoT Projects	363
Smart Home Devices	356	Summary	364
AI-driven home automation and intelligence	357		
Case study: an integrated smart home implementation	359		

14

AI for IIoT 367

Introduction to AI-Powered IIoT	367	Electrical Load Forecasting in Industry	389
Predictive Maintenance using AI	369	STLF using LSTM	389
Preventive maintenance real-world examples	371	IoT in Agriculture – Precision Farming with Edge AI	398
Predictive maintenance: hands-on examples	372	Summary	405

15

AI for Smart Cities IoT 407

Technical requirements	407	AI-Powered Urban Analytics	413
Why Do We Need Smart Cities?	407	AI-driven traffic optimization	413
AI Applications in Smart Cities	409	Crime hotspots prediction	419
Smart traffic management	410	Adapting IoT for Smart Cities and the Necessary Steps	423
Smart parking	410	Summary	426
Smart waste management	411	References	426
Smart policing	412		

Index 429

Other Books You May Enjoy 446

Preface

Dive into the revolutionary world of IoT and AI with *Hands-On Artificial Intelligence for IoT*, your comprehensive guide to integrating intelligent algorithms into IoT devices. This book demystifies the complex interplay between IoT and AI, providing practical insights and expert techniques to enhance and optimize your IoT systems. Whether you're an engineer, developer, or tech enthusiast, this book is designed to empower you with the skills and knowledge needed to transform IoT devices into smart, efficient, and AI-driven systems.

In this book, you'll embark on a journey through the realms of AI and IoT, exploring how these two transformative technologies can work together to create smarter and more efficient systems. From fundamental concepts to advanced applications, the book covers a wide range of topics, ensuring a holistic understanding of how AI can be integrated into IoT devices. You'll learn about machine learning algorithms, deep learning techniques, and practical applications that can be implemented in real-world IoT scenarios.

Discover the power of data-driven insights as you delve into practical machine learning and deep learning techniques, tailored specifically for IoT systems. The book provides hands-on examples, case studies, and real-world projects, enabling you to apply your learning immediately and effectively. You'll gain invaluable experience in optimizing IoT devices, enhancing their performance, and integrating AI algorithms to make intelligent decisions.

Hands-On Artificial Intelligence for IoT is not just a book; it's a journey towards mastering the art of AI-driven IoT. With expert guidance, practical examples, and a clear, concise approach, this book is your key to unlocking the full potential of IoT through the power of artificial intelligence. Transform your devices, enhance their capabilities, and step into the future of technology with confidence and expertise.

Who this book is for

This book is tailored for IoT developers, engineers, and tech enthusiasts, particularly those with a background in Python, looking to integrate artificial intelligence and machine learning into IoT systems. It provides a comprehensive guide to enhancing and optimizing IoT devices through practical AI and ML techniques. You will embark on a hands-on journey, acquiring skills to transform ordinary IoT systems into intelligent, efficient solutions. This book is ideal for Python developers eager to apply their knowledge in new, innovative ways and anyone with a foundational understanding of IoT concepts ready to elevate their skills.

What this book covers

Chapter 1, Principles and Foundations of IoT and AI, introduces the core concepts of IoT, AI, data science, and edge computing. You will learn how AI and data science enhance IoT systems and explore a hands-on example using IoT data, setting the stage for intelligent, efficient applications.

Chapter 2, Data Access and Distributed Processing for IoT, is where you will learn how to access and process data from various sources such as CSV, JSON, SQL/NoSQL databases, and streaming platforms. It introduces edge data storage and processing, equipping you to handle distributed IoT data effectively.

Chapter 3, Machine Learning for IoT, explores supervised and unsupervised machine learning techniques—such as regression, decision trees, and clustering—and their application to IoT data. You will also get hands-on with frameworks such as Scikit-Learn and TinyML to build intelligent IoT systems.

Chapter 4, Deep Learning for IoT, dives into deep learning techniques such as MLP, CNN, and RNN, exploring their practical applications in IoT. The chapter features hands-on examples using Keras 3, including image classification with ResNet50, experiments with activation functions, and hyperparameter tuning—empowering you to effectively apply deep learning to real-world IoT data scenarios.

Chapter 5, Optimization Techniques in IoT, introduces a range of optimization techniques to enhance the performance and efficiency of AI models in IoT applications. You will explore evolutionary methods such as genetic algorithms, along with particle swarm optimization, simulated annealing, and other strategies. Practical implementation is emphasized through the use of Python libraries such as DEAP and PyGAD, enabling hands-on learning for solving real-world IoT challenges.

Chapter 6, Reinforcement Learning for IoT, introduces core reinforcement learning concepts such as Q-learning, policy gradients, and Deep Q-Networks. You will learn how to apply these ideas in IoT scenarios through hands-on implementation using Keras. With practical examples such as training agents to make decisions based on rewards, the chapter builds a strong foundation for applying reinforcement learning to dynamic, data-driven IoT environments.

Chapter 7, Generative Adversarial Networks for IoT, introduces the concepts of generative and adversarial learning, including architectures such as GANs, DCGANs, and CycleGANs. You will gain hands-on experience training **Conditional GANs (CGANs)** on datasets such as MNIST, and implementing **Variational Autoencoders (VAEs)** for anomaly detection—an essential capability for intelligent IoT systems. By the end, you'll understand how to generate synthetic data and detect unusual patterns in IoT environments.

Chapter 8, Distributed AI for IoT, explores distributed learning techniques to scale AI for large and complex IoT applications. You will learn about the principles of data parallelism and model parallelism and implement them using Keras 3. Through practical examples, you'll understand how to efficiently train AI models across multiple devices or processors—crucial for deploying scalable, high-performance IoT solutions.

Chapter 9, AI Cloud Platforms for IoT, introduces major cloud platforms and their AI services tailored for IoT applications. You will gain hands-on experience using Amazon Rekognition (via Boto3) for object detection and the Azure Speech SDK for recognizing spoken commands. By leveraging cloud-based AI capabilities, you'll learn how to extend the intelligence of IoT systems without heavy local computation.

Chapter 10, Deep Learning for Time Series Data from IoT, focuses on time series analysis for IoT, using air quality data as a real-world example. You will build and compare models using SparkML regressors, LSTM neural networks, and Meta's Prophet library for forecasting. The chapter also contrasts the capabilities of Prophet and Intel's Chronos toolkit, helping you understand which tools best suit different IoT time series scenarios.

Chapter 11, AI for Video and Image Data from IoT, dives into processing visual data from IoT devices using a range of powerful AI techniques. You will implement face blurring with Gaussian filters, image classification using MobileNet, pose detection with MediaPipe, and object detection and localization using YOLO. The chapter also introduces Vision Transformers for advanced feature extraction, enabling robust, real-time analysis of image and video data in smart IoT systems.

Chapter 12, AI for Text, Audio, and Speech Data from IoT, explores how to process and analyze text and speech data in IoT systems. You will learn core NLP techniques such as tokenization, normalization, stop word removal, and lemmatization, along with the concept of embeddings. You will also work with pre-trained **Automatic Speech Recognition (ASR)** models to transcribe spoken commands and fine-tune language models to identify user intent—key for building voice-enabled and context-aware IoT applications.

Chapter 13, AI for Personal and Home IoT, explores the application of AI in personal and home IoT settings. You will learn how to analyze data from fitness trackers and build models to predict future step counts based on historical patterns. The chapter also discusses real-world examples where wearable devices have supported health monitoring and highlights various smart home devices, showcasing how IoT and AI can work together to create safer, more personalized living environments.

Chapter 14, AI for Industrial IoT, focuses on real-world industrial IoT applications powered by AI. You will build models for short-term electric load forecasting and machine failure prediction and use LSTM networks to estimate the **Remaining Useful Life (RUL)** of equipment. The chapter also includes case studies of industries successfully leveraging AI and IoT, highlighting how these technologies have improved efficiency, reduced downtime, and boosted overall productivity.

Chapter 15, AI for Smart Cities IoT, explores how AI and IoT are transforming urban living through real-world case studies on smart parking, traffic management, street lighting, and waste management. You will also implement reinforcement learning to optimize traffic light timings using synthetic data and build predictive models to identify potential crime hotspots. These examples illustrate how intelligent systems can enhance safety, sustainability, and efficiency in modern cities.

To get the most out of this book

Before you begin, a working knowledge of Python is essential, as the book focuses on applying AI techniques through hands-on coding. All code examples are provided in Python and can be run easily on Google Colab, requiring minimal setup. Familiarity with basic machine learning concepts will be helpful but is not mandatory. Step-by-step instructions are included to guide you through practical implementation.

Software/hardware covered in the book	Operating system requirements
Keras 3	Windows, macOS, or Linux
pandas	
NumPy	
scikit-learn	
Matplotlib	
Seaborn	

Besides the dependencies and hardware/software requirements listed, some chapters may require additional libraries specific to the use case being discussed. These dependencies are mentioned clearly at the beginning of each relevant chapter, along with step-by-step installation instructions.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book’s GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

The only true way to learn is by doing—run the code examples provided in the book, experiment by changing parameters, and observe the outcomes. Hands-on practice will not only deepen your understanding but also build the confidence to apply these concepts to real-world problems. Let curiosity be your guide as you explore and innovate.

Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Hands-On-Artificial-Intelligence-for-IoT-Second-Edition>. If there’s an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`, Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example, “Let’s generate the forecast for the next 365 days. Since we don’t have the future values for `PT08.S1(CO)`, `NMHC(GT)`, and `NOx(GT)`, we will assume these values remain constant at their mean.”

A block of code is set as follows:

```
future = model.make_future_dataframe(periods=365)

future['PT08.S1(CO)'] = df['PT08.S1(CO)'].mean()
future['NMHC(GT)'] = df['NMHC(GT)'].mean()
future['NOx(GT)'] = df['NOx(GT)'].mean()
forecast = model.predict(future)
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in **bold**:

```
y_pred = model.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print(f"Root Mean Squared Error (RMSE), {rmse}")

Output>>>
59/59 1s 10ms/step
Root Mean Squared Error (RMSE), 0.05522588172402996
exten => i,1,VoiceMail(s0)
```

Any command-line input or output is written as follows:

```
pip install pyspark
```

Bold, Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example, “From the graph, we can see that CO levels peak on **Wednesday** and **Friday**, and dip on **Saturday** and **Sunday**.”

Tips or important notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback, If you have questions about any aspect of this book, email us at customer care@packtpub.com and mention the book title in the subject of your message.

Errata, Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy, If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author, If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share Your Thoughts

Once you've read *Hands-On Artificial Intelligence for IoT*, we'd love to hear your thoughts! Please click [here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/978-1-83546-718-3>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1: Foundations and Basic Integrations of IoT and AI

This part introduces the fundamental building blocks of IoT and AI integration. We will learn how to access, process, and model IoT data using machine learning and edge computing. With a strong focus on foundational concepts, this section builds the essential skills needed to design intelligent IoT systems from the ground up.

This part has the following chapters:

- *Chapter 1, Principles and Foundations of IoT and AI*
- *Chapter 2, Data Access and Distributed Processing for IoT*
- *Chapter 3, Machine Learning for IoT*

Principles and Foundations of IoT and AI

Congratulations on purchasing this book; it suggests that you're keenly interested in keeping yourself updated with the recent advancements in technology. This book addresses the major trends shaping today's business landscape, including the **Internet of Things (IoT)**, enhanced by **Large Language Models (LLMs)**, big data, **Artificial Intelligence (AI)**, and the emerging field of Generative AI. The exponential growth in the number of devices connected to the internet, coupled with the surge in data volume generated by these devices and the integration of LLMs in IoT, necessitates the adoption of advanced analytical, predictive, and generative techniques from AI, **deep learning (DL)**, and Generative AI. This book provides a comprehensive overview of the various analytical, predictive, and generative models and methods available in the field of AI, specifically tailored for the vast and complex data landscape created by IoT.

This chapter will briefly introduce you to these trends and expand on how they're interdependent. The data generated by IoT devices is uploaded to the cloud; hence, you'll also be introduced to the various IoT cloud platforms and the data services they offer.

This chapter will cover the following topics:

- What is IoT 101?
- Big Data and IoT
- Infusion of AI – Data Science in IoT
- Tools Used

What is IoT 101?

The term *IoT* was coined by Kevin Ashton in 1999. At that time, most of the data fed to computers was generated by humans; he proposed that the best way would be for computers to take data directly, without any intervention from humans. So, he proposed that things such as **Radio frequency Identification (RFID)** and sensors, which gather data, should be connected to a network and feed directly to a computer.

Tip

You can read the complete article where Ashton talks about what he means by IoT here: <http://www.itrco.jp/libraries/RFIDjournal-That%20Internet%20of%20Things%20Thing.pdf>.

Today, IoT (also called the *internet of everything* and sometimes, the *fog network*) refers to a wide range of things, such as sensors, actuators, and smartphones connected to the internet. These things can be anything – a person with a wearable device (or even a mobile phone), an RFID-tagged animal, or even our day-to-day devices such as a refrigerator, washing machine, or a coffee machine. These things can be physical things – that is, things that exist in the physical world and can be sensed, actuated, and connected – or of the information world (a virtual thing) – that is, things that aren’t tangibly present but exist as information (data) and can be stored, processed, and accessed. These things necessarily have the ability to communicate directly with the internet; optionally, they might have the potentiality of sensing, actuation, data capture, data storage, and data processing.

The **International Telecommunication Unit (ITU)**, a United Nations agency, defines IoT as follows:

“[A] global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies.”

You can learn more at <https://www.itu.int/en/ITU-T/gsi/iot/Pages/default.aspx>.

The wide expanse of **Information and Communications Technology (ICT)** already provides us with communication at any time or any place; the IoT added a new dimension of **ANY THING** communication:

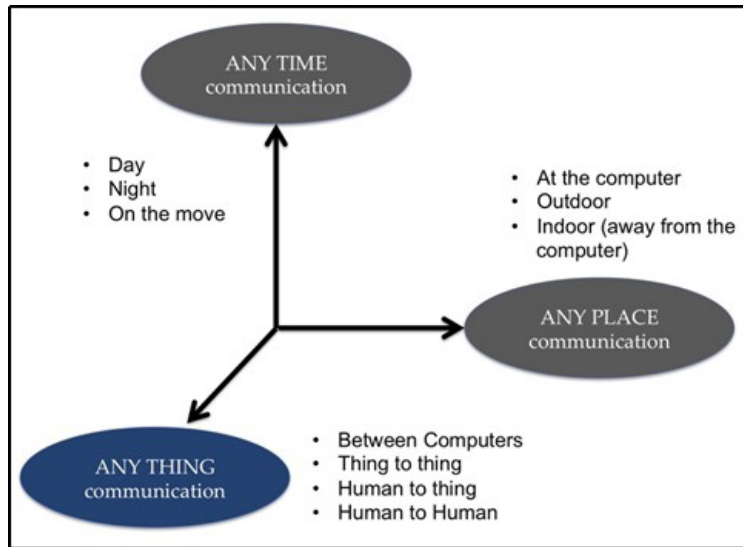


Figure 1.1: A new dimension introduced in IoT (adapted from the b-ITU-T.Y.2060 report)

It's predicted that IoT as a technology will have a far-reaching impact on people and the society we live in. To give you a glimpse of its far-reaching effects, consider the following scenarios:

- You, like me, live in a high-rise building and are very fond of plants. With lots of effort and care, you've made a small indoor garden of your own using potted plants. Your boss asks you to go for a week-long trip, and you're worried your plants won't survive for a week without water. The IoT solution is to add soil moisture sensors to your plants, connect them to the internet, and add actuators to remotely switch on or off the water supply and artificial sunlight. Now, you can be anywhere in the world, but your plants won't die, and you can check an individual plant's soil moisture condition and water it as needed.
- You had a very tiring day at the office; you just want to go home and have someone make you coffee, prepare your bed, and heat up water for a bath, but sadly, you're home alone. Not anymore – IoT can help. Your IoT-enabled home assistant can prepare the right flavor coffee from the coffee machine, order your smart water heater to switch on and maintain the water temperature exactly the way you want, and ask your smart air conditioner to switch on and cool the room.

The choices are limited only by your imagination. The two preceding scenarios correspond to consumer IoT – the IoT with a focus on consumer-oriented applications. There also exists a large scope of **Industry IoT (IIoT)**, where manufacturers and industries optimize processes and implement remote monitoring capabilities to increase productivity and efficiency. In this book, you'll find hands-on experience with both IoT applications.

IoT reference model

Just like the OSI reference model for the internet, IoT architecture is defined through six layers – four horizontal layers and two vertical layers. The two vertical layers are management and security, and they're spread over all four horizontal layers, as seen in the following diagram:

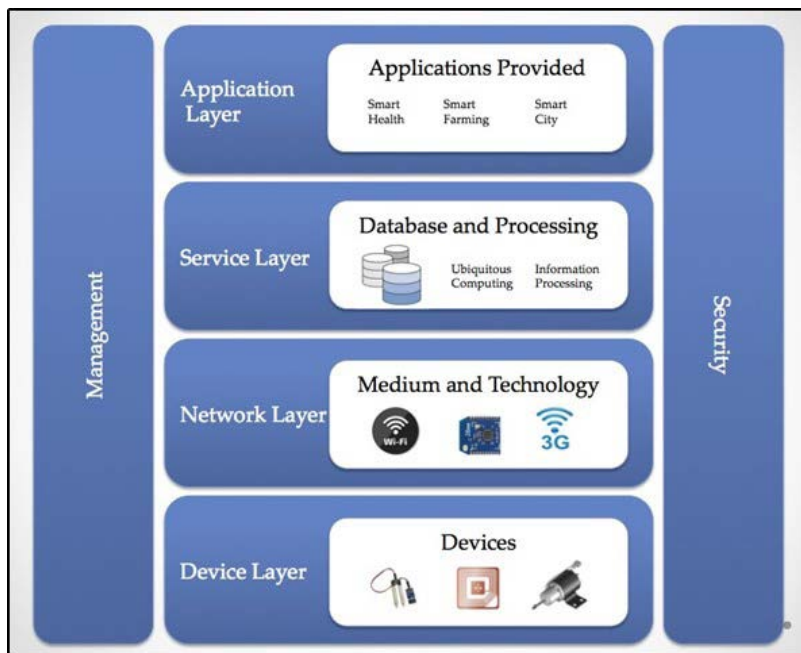


Figure 1.2: The IoT layers

Each layer has a unique function, like so:

- **The device layer:** At the bottom of the stack, we have the device layer, also called the perception layer. This layer contains the physical things needed to sense or control the physical world and acquire data (that is, by perceiving the physical world). Existing hardware, such as sensors, RFID, and actuators, constitutes the perception layer.

- **The network layer:** This layer provides the networking support and transfer of data over either wired or wireless network. The layer securely transmits the information from the devices in the device layer to the information processing system. Both transmission Medium and Technology are part of the networking layer. Examples include 3G, **Universal Mobile Telecommunications System(UMTS)**, ZigBee, Bluetooth, Wi-Fi, and so on.
- **The service layer:** This layer is responsible for service management. It receives information from the network layer, stores it in the database, processes that information, and can make an automatic decision based on the results.
- **The application layer:** This layer manages the applications dependent upon the information processed in the service layer. There's a wide range of applications that can be implemented by IoT – smart cities, smart farming, and smart homes, to name a few.
- **The management layer:** This vertical layer spans across the device, network, service, and application layers to ensure the overall effective operation and governance of the IoT system. It encompasses functions such as device management, including provisioning and authentication, system administration, and maintenance tasks to keep the network operational, efficient, and up to date.
- **The security layer:** The security layer also cuts across all horizontal layers, providing essential protections for the IoT ecosystem. This includes safeguarding data integrity and privacy from the device level up to the application layer, implementing encryption, conducting regular security assessments, and ensuring compliance with relevant standards and regulations to prevent unauthorized access and cyber threats.

IoT platforms

Information from the network layer is often managed with the help of IoT platforms. Many companies today provide IoT platform services, where they help not only with data but also enable seamless integration with different hardware. Since they function as a mediator between the hardware and application layer, IoT platforms are also referred to as IoT middleware and are part of the service layer in the IoT reference stack. IoT platforms give you the ability to connect and communicate with things from anywhere in the world. In *Chapter 9*, we'll briefly cover some popular IoT platforms, such as Microsoft Azure IoT Hub, AWS IoT Core, and Cisco Industrial IoT Solutions.

You can select which IoT platform is best for you based on the following criteria:

- **Scalability:** The addition and deletion of new devices to the existing IoT network should be possible.
- **Ease of use:** The system should work perfectly and deliver all its specifications with minimum intervention.

- **Third-party integration:** Heterogeneous devices and protocols should be able to inter-network with each other.
- **Deployment options:** It should be workable on a broad variety of hardware devices and software platforms.
- **Data security:** The security of data and devices is ensured.

IoT verticals

A vertical market is a market in which vendors offer goods and services specific to an industry, trade, profession, or other groups of customers with specialized needs. IoT enables the possibility of many such verticals, and some of the top IoT verticals are as follows:

- **Smart buildings:** Buildings with IoT technologies can help not only reduce the consumption of resources but also improve the satisfaction of the humans living or working in them. The buildings have smart sensors that can not only monitor resource consumption but also proactively detect residents' needs. Data is collected via these smart devices and sensors to remotely monitor a building, energy, security, landscaping, HVAC, lighting, and so on. The data is then used to predict actions, which can be automated according to events; hence, efficiency can be optimized, saving time, resources, and cost.
- **Smart agriculture:** IoT can enable local and commercial farming to be more environmentally friendly, cost-effective, and production-efficient. Sensors placed through the farm can help to automate the process of irrigation. It's predicted that smart agricultural practices will enable a manifold increase in productivity and, hence, food resources.
- **Smart city:** A smart city can be a city with smart parking, a smart mass transit system, and so on. A smart city can address traffic, public safety, energy management, and so on for both its government and citizens. By using advanced IoT technologies, it can optimize the usage of the city infrastructure and quality of life for its citizens.
- **Connected healthcare:** IoT enables critical business and patient monitoring decisions to be made remotely and in real time. Individuals carry medical sensors to monitor body parameters, such as heartbeat, body temperature, and glucose levels. Wearable sensors, such as accelerometers and gyroscopes, can be used to monitor a person's daily activity.
- **Smart retail:** IoT technology revolutionizes inventory management, customer interaction, digital signage, and smart checkout processes. It enables a more personalized and enriched shopping experience by leveraging real-time data to tailor services and streamline operations.

- **Supply chain/logistics:** IoT enhances supply chain transparency and efficiency through sophisticated asset tracking, comprehensive fleet management, and meticulous cold chain monitoring. This suite of technologies enables real-time visibility and control, optimizing the flow of goods from origin to delivery.

We'll be covering some of them as a case study in this book. The content of this book is focused on information processing and the applications being implemented on IoT, so we'll not go into further detail about the devices, architecture, and protocols involved in IoT reference stacks.

Note

You can refer to the following references to learn more about the IoT architecture and different protocols:

Da Xu, Li, Wu He, and Shancang Li. Internet of things in industries: A survey. *IEEE Transactions on industrial informatics* 10.4 (2014): 2233-2243.

Khan, Rafiullah, et al. Future internet: The internet of things architecture, Possible Applications and Key Challenges. *Frontiers of Information Technology (FIT)*, 2012 10th International Conference on. IEEE, 2012.

This website provides an overview of the protocols involved in IoT: <https://www.postscapes.com/internet-of-things-protocols/>.

Now that we've walked through what IoT is, how it can impact our daily lives, and where it is valuable, in the next section, let's check out one of the biggest boosts to the way IoT works in the modern world – big data.

Big Data and IoT

IoT has connected things never previously connected to the internet, such as car engines, resulting in the generation of a large amount of continuous data streams. In the first edition of this book, an IHS analysis forecasted an upsurge in IoT devices to 75.44 billion by 2025. However, recent revisions account for the COVID-19 pandemic's impact and other market dynamics, adjusting this number to approximately thirty billion. Factors such as the COVID-19 disruption, slower industrial IoT growth, semiconductor shortages, and refined IoT definitions contribute to this recalibration. Despite the downward adjustment, forecasts unanimously indicate significant IoT expansion, suggesting a vibrant trajectory for the technology moving forward:

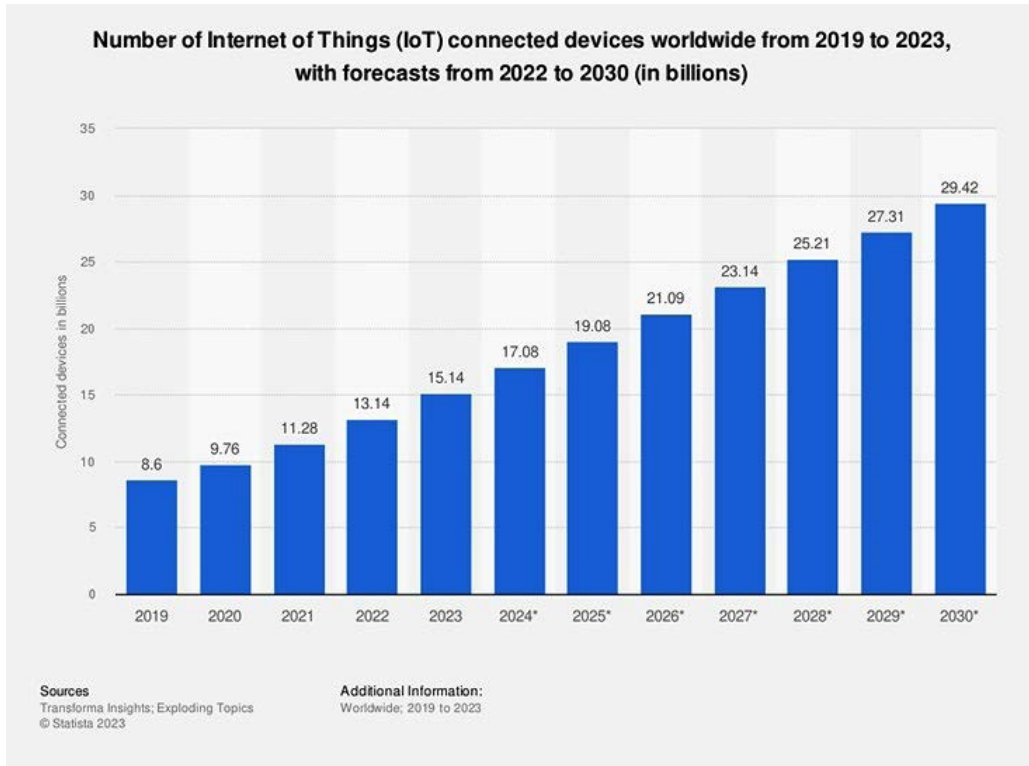


Figure 1.3: Prediction about the growth of IoT devices by 2025

Despite the recalibration of IoT device estimates, the pervasive influence of IoT technology continues to burgeon. The reduction in sensor cost, efficient power consumption techniques, a large range of connectivity (infrared, NFC, Bluetooth, Wi-Fi, etc.), and the availability of cloud platforms that support IoT deployment and development are the major reasons for this pervasion of IoT in our homes, personal lives, and industry. This has also motivated companies to think about providing new services and developing new business models. Some examples include the following:

- **FedEx SenseAware:** SenseAware by FedEx is a cutting-edge solution that offers unparalleled real-time insights for critical shipments. Tailored for customers with specialized needs, such as those requiring the transportation of high-value, technologically sophisticated, or temperature-sensitive items, SenseAware provides comprehensive monitoring. Customers benefit from detailed tracking capabilities that extend to humidity, precise location, and even exposure to light, ensuring the integrity and security of each shipment with exceptional accuracy and control.
- **Amazon Go stores:** These use IoT sensors and machine vision technologies to create a checkout-free shopping experience, where customers simply take items off the shelves and are automatically charged as they leave the store.

The amount of data generated in the process is both voluminous and complex, necessitating big data. A big data approach and IoT are almost made for each other; the two work in conjunction.

Edge IoT – enhancing data efficiency and responsiveness

Amid this vast data influx, **Edge IoT** emerges as a pivotal element, tasked with refining the efficiency and responsiveness of data processing. IoT is a relentless generator of continuous data streams that detail the status of connected devices, encompassing variables such as temperature, pollution levels, geolocation, and proximity. Characteristically time-series and inherently autocorrelated, this data is dynamic, presenting challenges that call for advanced analysis techniques.

Before the journey of this data to the cloud for further processing, a transformative step occurs at the edge, which is where Edge IoT comes to the forefront. By handling data transformation at the edge, we gain several advantages:

- **Immediate analysis:** By processing data where it's generated, Edge IoT provides swift insights, essential for real-time decision-making.
- **Reduced latency:** Local data processing minimizes the delay in transmitting data to and from the cloud, crucial for applications that require quick responses.
- **Bandwidth efficiency:** Processing data at the edge reduces the volume of data that needs to be sent to the cloud, easing network traffic and associated costs.

At the edge, data is not merely passed along; it is enhanced and refined. Edge IoT involves several key processes:

- Temporal or spatial analysis
- Summarizing data at the edge
- The aggregation of data
- Correlating data in multiple IoT streams
- Cleaning data
- Filling in the missing values
- Normalizing data
- Transforming it into different formats acceptable to the cloud

You might think that while we have highlighted several processes and concepts, some have only been briefly touched upon. This is intentional. Our goal is to establish a solid foundation and familiarize you with the fundamental ideas in IoT, AI, Data Science, and edge computing.

We will explore many of these processes more thoroughly in the coming chapters.

At the edge, **complex event processing (CEP)** is used to combine data from multiple sources and infer events or patterns.

Data is analyzed using **stream analytics** – for example, applying analytical tools to the stream of data but developing the insights and rules used externally in an offline mode. The model is built offline and then applied to the stream of data generated. The data can be handled in different ways:

- **Atomic:** A single instance of data at a time is used.
- **Micro batching:** A group of data per batch.
- **Windowing:** Data within a timeframe per batch.

Stream analytics can be combined with CEP to combine events over a time frame and correlate patterns to detect special patterns (for example, an anomaly or failure).

By situating these data-handling capabilities at the edge, IoT ecosystems can respond more nimbly to changing conditions and make smarter use of a network's bandwidth and cloud resources. Edge IoT is not just a component of big data and IoT convergence; it is also an active agent in advancing the intelligence and autonomy of the IoT infrastructure. Let us next explore the most important component of adding AI to IoT – data.

Infusion of AI – Data Science in IoT

A very popular phrase among data scientists and machine learning engineers is “*AI is the new electricity*,” as said by Professor Andrew Ng in NIPS 2017. We can expand this as follows – if AI is the new electricity, data is the new coal, and IoT the new coalmine.

IoT generates an enormous amount of data; presently, ninety percent of the data generated isn't even captured, and out of the 10% that is captured, most is time-dependent and loses its value within milliseconds. Manually monitoring this data continuously is both cumbersome and expensive. This necessitates a way to intelligently analyze and gain insight from this data; the tools and models of AI provide us with a way to do exactly this with minimum human intervention. The major focus of this book will be on understanding the various AI models and techniques that can be applied to IoT data. We'll be leveraging a suite of sophisticated algorithms from the realms of **machine learning (ML)**, **DL**, and the emergent field of Generative AI, which includes transformative LLMs. The accompanying diagram illustrates the relationship between AI, ML, DL, and now, Generative AI:

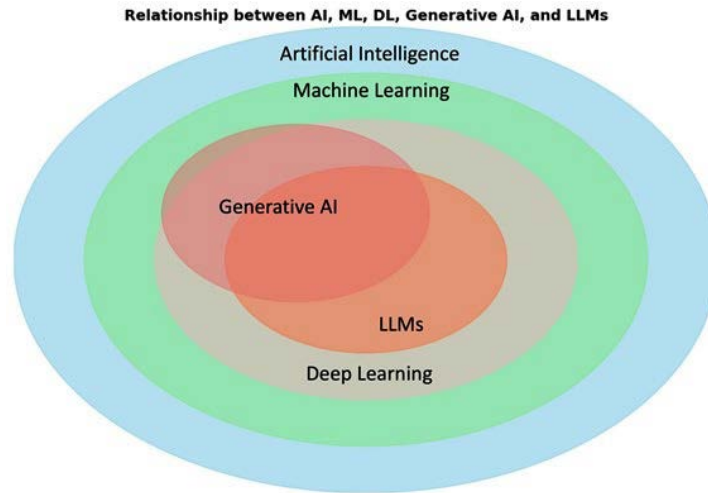


Figure 1.4: Relationship between AI, ML, DL, Generative AI, and LLMs

- **AI:** The overarching discipline that encompasses any technique that enables machines to mimic human behavior.
- **ML:** A subset of AI that includes statistical methods, enabling machines to improve tasks with experience.
- **DL:** A subset of ML that uses neural networks with three or more layers, which are adept at processing unstructured data such as images and text.
- **Generative AI:** An advanced frontier within AI focusing on the generation of content, which includes LLMs that can produce human-like text, create images, or simulate environments.

Generative AI, particularly through LLMs, is revolutionizing the field by not only analyzing but also generating new data and patterns that can enhance decision-making, automate creative processes, and personalize interactions. These models learn from vast datasets and can produce outputs that were previously unthinkable, from drafting articles to composing code. As we integrate these powerful models into our systems, we're opening doors to unprecedented applications and efficiencies.

By closely monitoring the behavior of a multitude of interconnected devices, IoT, empowered by big data analytics and AI, endeavors to extract meaningful insights from data to refine and optimize underlying processes. This ambitious goal introduces several challenges:

- Storing real-time generated events.
- Running analytical queries over stored events.
- Performing analytics using AI/ML/DL techniques over data to gain insights and make predictions.

Each of these challenges represents a critical juncture in the value chain of IoT data analytics, which, when effectively managed, unlocks the full potential of IoT and AI synergy.

Cross-industry standard process for data mining

For IoT problems, the most used **data management (DM)** methodology is the **cross-industry standard process for data mining (CRISP-DM)**, proposed by Chapman et al. It's a process model that states the tasks that need to be carried out to successfully complete DM. It's a vendor-independent methodology divided into six different phases:

1. **Business understanding:** Identify the IoT objectives; determine how sensor data and connectivity can address business challenges or create new opportunities.
2. **Data understanding:** Collect and explore IoT data to detect patterns, anomalies, and system performance benchmarks.
3. **Data preparation:** Clean, integrate, and transform IoT data into a structured format suitable for analysis.
4. **Modeling:** Apply AI/ML/DL algorithms to IoT data to build predictive models, iterating as needed for accuracy.
5. **Evaluation:** Assess the models against IoT business goals for validity and effectiveness in a real-world context.
6. **Deployment:** Integrate the models into the IoT ecosystem to make data-driven decisions and automate processes.

The following diagram shows the different stages:

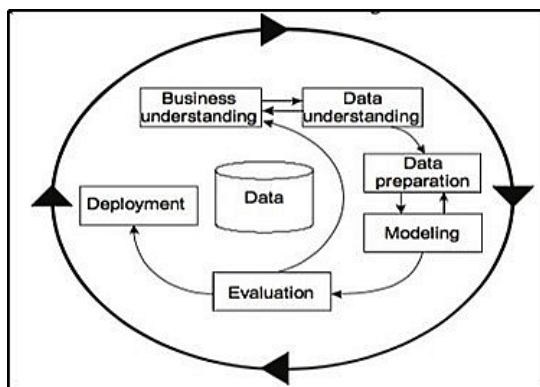


Figure 1.5: The different stages in CRISP-DM

These phases are not strictly linear but, rather, iterative; as new data comes in and the business environment changes, you may need to revisit earlier stages, with data science and AI playing important

roles in *steps 2–5*. For example, if a model's performance degrades over time, you may need to go back to the data understanding or preparation phases to adjust for changes in sensor calibration or to incorporate additional data sources.

Note

The details about CRISP-DM and all its phases can be read in the following:

Marbán, Óscar, Gonzalo Mariscal, and Javier Segovia. A data mining & knowledge discovery process model. Data Mining and Knowledge Discovery in Real Life Applications. InTech, 2009.

AI platforms and IoT platforms

A large number of cloud platforms with both AI and IoT capabilities are available today. These platforms provide the capability to integrate sensors and devices and perform analytics on the cloud. There exist more than thirty cloud platforms in the global market, each targeting different IoT verticals and services. The following screenshot lists the various services that AI/IoT platforms support:



Figure 1.6: Services that different AI/IoT platforms support

Let's briefly talk about some popular cloud platforms. In *Chapter 12*, we'll learn how to use the most popular ones. The following is a list of some of the popular cloud platforms:

- **IBM Watson IoT Platform:** Hosted by IBM, the platform provides device management; it uses the **Message Queuing Telemetry Transport (MQTT)** protocol to connect with IoT devices and applications. It provides real-time scalable connectivity. The data can be stored for a period and accessed in real time. IBM Watson also provides Bluemix **Platform-as-a-Service (PaaS)** for analytics and visualizations. We can write code to build and manage applications that interact with data and connected devices. It supports Python along with C#, Java, and Node.js.
- **Microsoft IoT-Azure IoT suite:** This provides a collection of preconfigured solutions built on Azure PaaS. It enables reliable and secure bidirectional communication between IoT devices and the cloud. The preconfigured solutions include data visualization, remote monitoring, and configuring rules and alarms over live IoT telemetry. It also provides Azure Stream Analytics to process data in real time. Azure Stream Analytics allows us to use Visual Studio. It supports Python, Node.js, C, and Arduino, depending on the IoT devices.
- **AWS IoT:** AWS IoT allows IoT devices to communicate via MQTT, **HTTP**, and WebSockets. It provides secure, bidirectional communication between IoT devices and the cloud. It also has a rules engine that can be used to integrate data with other AWS services and transform it.

Rules can be defined that trigger the execution of user code in Java, Python, or Node.js. AWS Lambda allows us to use our own custom-trained models. With basics under our belt, let's move on to the actual tools we will use in this book to build AI-powered solutions for different IoT-based data and tasks.

Tools Used in this book

For the implementation of IoT-based services, we need to follow a bottom-up approach. For each IoT vertical, we need to find the analytics and the data and, finally, implement it in code.

Due to its availability in almost all AI and IoT platforms, Python will be used for coding in this book. Along with Python, some helpful libraries such as NumPy, pandas, SciPy, TensorFlow, and Keras (Keras 3.0) will be used to perform AI/ML analytics on data. For visualization, we will use Matplotlib and Seaborn.

TensorFlow

TensorFlow is an open source software library developed by the Google Brain team; it has functions and APIs for implementing deep neural networks. It works with Python, C++, Java, R, and Go. It can be used to work on multiple platforms – CPU, GPU, mobile, and even distributed platforms. TensorFlow allows model deployment and ease of use in production. It provides a unified and simplified API, with Keras integrated as its official high-level API, to build and train models with ease.

TensorFlow 2.0 promotes eager execution by default, improving the library's usability and making it more Pythonic. This means that operations are evaluated immediately and return their values without requiring a session and graph. TensorFlow 2.0 still retains its capability to operate across various platforms, including CPUs, GPUs, and mobile devices, allowing for distributed training and inference.

In TensorFlow 2.0, models are created and trained more intuitively, often with fewer lines of code compared to other frameworks. It also provides advanced optimizer functions, automatic differentiation capabilities, and a flexible suite of tools for complex neural network construction and training.

Let's see the code to perform matrix multiplication in TensorFlow. The whole code can be accessed from the GitHub repository (<https://github.com/PacktPublishing/Hands-On-Artificial-Intelligence-for-IoT-Second-Edition/tree/main/Chapter01>) filename, `matrix_multiplication.ipynb`:

We'll begin by importing the TensorFlow module:

```
import tensorflow as tf
import numpy as np
```

Next, we define two random matrices, `mat1` and `mat2`, that we need to multiply:

```
# A random matrix of size [3,5]
mat1 = np.random.rand(3,5)
# A random matrix of size [5,2]
mat2 = np.random.rand(5,2)
```

Next, we define the operation to be performed using the matrix multiplication method, `tf.matmul`:

```
result = tf.matmul(mat1,mat2)
```

The operations are executed as they are defined (eager execution), and the result can be obtained by simply calling `.numpy()` on the resulting tensor:

```
print(result.numpy())
```

The following is the result. Remember that since we are using random input, without setting a seed, you may get a different result from this matrix multiplication:

```
[[2.18607697 1.69920084]
 [2.33175192 1.97293846]
 [1.18090558 0.96962395]]
```

Keras 3.0

Keras 3.0 is an advanced iteration of the original Keras library, tailored for the construction and training of deep learning models. It inherits the simplicity and ease of use that Keras is known for, while

introducing new features that extend its utility. Keras 3.0 is part of the broader Keras ecosystem, which is a high-level neural network API, written in Python and capable of running on top of TensorFlow, JAX, or PyTorch. Its major features include the following:

- **Modular and composable:** Keras 3 is designed to be modular with a minimalistic interface. This design makes it a flexible and extensible framework that can be adapted to various ML tasks.
- **Backend-agnostic:** Like Keras, Keras 3 is designed to be backend-agnostic. It abstracts and encapsulates the backend-specific implementations while providing a consistent high-level API.
- **Friendly API:** Keras 3 maintains a user-friendly API that is easy to understand and work with. This contributes to faster development cycles and a reduction in the learning curve for new users.
- **Community and ecosystem:** Being part of the Keras ecosystem, Keras 3 benefits from a large community and a growing ecosystem of tools, libraries, and extensions.

While Keras 3 is not tailored specifically for IoT or edge computing, its features, such as backend flexibility, ease of deployment, and a modular design, may offer certain advantages in an IoT/edge setup. Exploring the combination of Keras 3 with efficient backends and hardware accelerators could unveil optimizations and performance improvements beneficial to IoT and edge deployments.

Datasets

In the upcoming chapters, we'll be learning about different DL models and ML methods. They all work on data; while a large number of datasets are available to demonstrate how these models work, in this book, we'll use datasets available freely through wireless sensors and other IoT devices. The following are some of the datasets used in this book and their sources.

The combined cycle power plant dataset

This dataset contains 9,568 data points collected from a **combined cycle power plant (CCPP)** over the course of six years (2006–2011). CCPP uses two turbines to generate power, the gas turbine and the steam turbine. There are three main components of the CCPP plant – a gas turbine, heat recovery system, and steam turbine. The dataset, available at UCI ML (<http://archive.ics.uci.edu/ml/datasets/combined+cycle+power+plant>), was collected by Pinar Tufekci from Namik Kemal University and Heysem Kaya from Bogazici University. The data consists of four features that determine the average ambient variables. The averages are taken from various sensors located around the plant that record ambient variables per second. The aim is to predict the net hourly electrical energy output. The data is available in both the `.xls` and `.ods` formats.

The features in the dataset are as follows:

- The **Ambient Temperature (AT)** is in the range of 1.81°C and 37.11°C. The **Ambient Pressure (AP)** is in the range of 992.89 to 1033.30 millibars. **Relative Humidity (RH)** is in the range of 25.56% to 100.16%.
- Exhaust **Vacuum (V)** is in the range of 25.36 to 81.56 cm Hg.
- The net hourly electrical energy output (PE) is in the range 420.26 to 495.76 MW.

Note

Further details about the data and the problem can be read in the following:

Pınar Tüfekci, Prediction of full load electrical power output of a baseload operated combined cycle power plant using machine learning methods, International Journal of Electrical Power & Energy Systems, Volume 60, September 2014, Pages 126–140, ISSN 0142-0615.

Heysem Kaya, Pınar Tüfekci, Sadık Fikret Gürgen: Local and Global Learning Methods for Predicting Power of a Combined Gas & Steam Turbine, Proceedings of the International Conference on Emerging Trends in Computer and Electronics Engineering ICETCEE 2012, pp. 13–18 (Mar. 2012, Dubai).

Wine quality dataset

Wineries around the world have to undergo wine certifications and quality assessments to safeguard human health. The wine certification is performed with the help of physicochemical analysis and sensory tests. With the advancement of technology, the physicochemical analysis can be performed routinely via in-vitro equipment.

We use this dataset for classification examples in this book. The dataset can be downloaded from the UCI-ML repository (<https://archive.ics.uci.edu/ml/datasets/Wine+Quality>). The wine quality dataset contains the results of physicochemical tests on different samples of red and white wine. Each sample was further rated by an expert wine taster for quality on a scale of zero to ten.

The dataset contains, in total, four thousand eight hundred and ninety eight instances, and it has a total of twelve attributes. The twelve attributes are as follows:

- Fixed acidity
- Volatile acidity
- Citric acid
- Residual sugar
- Chlorides
- Free sulfur dioxide

- Total sulfur dioxide
- Density
- pH
- Sulfates
- Alcohol quality

The dataset is available in the CSV format.

Note

Details about the dataset can be read from this paper by Cortez, Paulo, et al:

Modeling wine preferences by data mining from physicochemical properties. Decision Support Systems 47.4 (2009): 547–553 (<https://www.sciencedirect.com/science/article/abs/pii/S0167923609001377>).

Summary

In this chapter, we learned about IoT, big data, and AI. We introduced the common terminologies used in IoT. We learned about the IoT architecture for data management and data analysis. The enormous data generated by IoT devices necessitates special ways to handle it.

We learned about how data science and AI can help in both the analytics and prediction generated by the many IoT devices. Various IoT platforms were briefly explored in this chapter, as were some popular IoT verticals. We also learned about special DL libraries – TensorFlow and Keras 3.0. Finally, some of the datasets we'll be using throughout the book were introduced.

The next chapter will cover how to access the datasets available in various formats.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/ds>



Data Access and Distributed Processing for IoT

Data is everywhere – images, speech, text, weather information, the speed of your car, your last EMI, and changing stock prices. With the integration of **Internet of Things (IoT)** systems, the amount of data produced has increased many-fold. An example is sensor readings, which could be taken for room temperature, soil alkalinity, and so on. This data is stored and made available in various formats. In this chapter, we will learn how to read, save, and process data in some of the most popular formats. Specifically, you will do the following:

- Data in the TXT Format
- Data in the XLSX Format
- CSV Format
- Working with the JSON
- Handling Data in the HDF5 Format
- Object Store for Raw Data
- Working with Structured SQL Data
- HDFS for Large-Scale Data Storage and Retrieval
- Edge Data Storage and Processing

Technical requirements

To run the codes of this chapter, you will require the following:

- Python $\geq$ 3.9
- Pandas $\geq$ 1.5
- NumPy $\geq$ 1.23

- Matplotlib >= 3.7
- OpenPyXL >= 3.1
- SQLite
- MySQL
- HDF5 >= 1.10
- PyTables
- MongoDB
- PyArrow >= 10.0

The code is available at the following GitHub URL: <https://github.com/PacktPublishing/Hands-On-Artificial-Intelligence-for-IoT-Second-Edition/tree/main/Chapter02>.

Data in the TXT Format

One of the simplest and most common formats for storing data is the TXT format; many IoT sensors log sensor readings with different timestamps in the simple `.txt` file format. Python provides built-in functions for creating, reading, and writing into TXT files.

We can access TXT files in Python itself without using any module; the data, in this case, is of the string type, and you will need to transform it to other types to use it. Alternatively, we can use NumPy or pandas.

Using TXT files in Python

Python has built-in functions that read and write into TXT files. The complete functionality is provided using four sets of functions – `open()`, `read()`, `write()`, and `close()`. As the names suggest, they are used to open a file, read from a file, write into a file, and finally, close it, respectively. If you are dealing with string data (text), this is the best choice. In this section, we will use a Shakespeare play in TXT form, making use of the books available by Project Gutenberg (<https://www.gutenberg.org/ebooks/author/65>) ; specifically, we will download the book *The Taming of the Shrew* in the TXT format.

If you are using Colab or the Jupyter Notebook, you can download the play directly in the working directory using the following command::

```
!wget https://www.gutenberg.org/ebooks/1508.txt.utf-8
```

We will define the following variables to access the data:

```
data_file = '1508.txt.utf-8'
```

The first step here is to open the file:

```
f = open(data_file)
```

Next, we read the whole file; we can use the `read` function, which will read the whole file as one single string:

```
contents = f.read()
```

This reads the whole file (consisting of four million five hundred eighty-three thousand seven hundred ninety-eight characters) into the `contents` variable. Let's explore the contents of the `contents` variable; the following command will print the first one thousand characters:

```
print(contents[11000:12000])
```

The preceding code will print the output as follows:

```
ove pictures? We will fetch thee straight
Adonis painted by a running brook,
And Cytherea all in sedges hid,
Which seem to move and wanton with her breath
Even as the waving sedges play with wind.

LORD.
We'll show thee Io as she was a maid
And how she was beguiled and surpris'd,
As lively painted as the deed was done.

THIRD SERVANT.
Or Daphne roaming through a thorny wood,
Scratching her legs, that one shall swear she bleeds
And at that sight shall sad Apollo weep,
So workmanly the blood and tears are drawn.

LORD.
Thou art a lord, and nothing but a lord:
Thou hast a lady far more beautiful
Than any woman in this waning age.

FIRST SERVANT.
And, till the tears that she hath shed for thee
Like envious floods o'er-run her lovely face,
She was the fairest creature in the world;
And yet she is inferior to none.
```

```

SLY.
Am I a lord? and have I such a lady?
Or do I dream? Or have I dream'd till now?
I do not sleep: I see, I hear, I speak;
I smell sweet savours, and I feel soft things:
Upon m

```

3

If the TXT files contain numeric data, it is better to use NumPy; if data is mixed, pandas is the best choice.

CSV Format

The **comma-separated value (CSV)** files are the most popular formats for storing tabular data generated by IoT systems. In a .csv file, the values of the records are stored in plain-text rows, with each row containing the values of the fields separated by a separator. The separator is a comma by default but can be configured to be any other character. In this section, we will learn how to use data from CSV files with Python's csv, numpy, and pandas modules. We will use the household_power_consumption data file. The file can be downloaded from the following GitHub link: https://github.com/ahanse/machlearning/blob/master/household_power_consumption.csv, which we will do now:

```
!wget https://raw.githubusercontent.com/ahanse/machlearning/master/
household_power_consumption.csv
```

To access the data files, we will define the following variables:

```
data_file = 'household_power_consumption.csv'
```

Generally, to quickly read data from CSV files, we use the Python CSV module; however, if the data needs to be interpreted as a mixture of date and numeric data fields, it's better to use the pandas package. If the data is only numeric, NumPy is the most appropriate package.

Working with CSV files with the csv module

In Python, the csv module provides classes and methods to read and write CSV files. The csv.reader method creates a reader object from which rows can be read iteratively.

Each time a row is read from the file, the reader object returns a list of fields. For example, the following code demonstrates reading a data file and printing rows:

```

import csv
import os
with open(os.path.join(data_folder, data_file), newline='') as csvfile:
    csvreader = csv.reader(csvfile)
    for row in csvreader:
        print(row)

```

The rows are printed as a list of field values; here is a sample of the output:

```
['Date', 'Time', 'Global_active_power', 'Global_reactive_power',
'Voltage', 'Global_intensity', 'Sub_metering_1', 'Sub_metering_2',
'Sub_metering_3']
['16/12/2006', '17:24:00', '4.216', '0.418', '234.840', '18.400',
'0.000', '1.000', '17.000']
['16/12/2006', '17:25:00', '5.360', '0.436', '233.630', '23.000',
'0.000', '1.000', '16.000']
['16/12/2006', '17:26:00', '5.374', '0.498', '233.290', '23.000',
'0.000', '2.000', '17.000']
['16/12/2006', '17:27:00', '5.388', '0.502', '233.740', '23.000',
'0.000', '1.000', '17.000']
['16/12/2006', '17:28:00', '3.666', '0.528', '235.680', '15.800',
'0.000', '1.000', '17.000']
['16/12/2006', '17:29:00', '3.520', '0.522', '235.020', '15.000',
'0.000', '2.000', '17.000']
['16/12/2006', '17:30:00', '3.702', '0.520', '235.090', '15.800',
'0.000', '1.000', '17.000']
['16/12/2006', '17:31:00', '3.700', '0.520', '235.220', '15.800',
'0.000', '1.000', '17.000']
['16/12/2006', '17:32:00', '3.668', '0.510', '233.990', '15.800',
'0.000', '1.000', '17.000']
['16/12/2006', '17:33:00', '3.662', '0.510', '233.860', '15.800',
'0.000', '2.000', '16.000']
['16/12/2006', '17:34:00', '4.448', '0.498', '232.860', '19.600',
'0.000', '1.000', '17.000']
```

The `csv.writer` method returns an object that can be used to write rows to a file. As an example, the following code writes the first ten rows of the file to a temporary file and then prints it:

```
with open(data_file, newline='') as csvfile, \
    open('temp.csv', 'w', newline='') as tempfile:
    csvreader = csv.reader(csvfile)
    csvwriter = csv.writer(tempfile)
    for row, i in zip(csvreader, range(10)):
        csvwriter.writerow(row)
with open('temp.csv', newline='') as tempfile:
    csvreader = csv.reader(tempfile)
    for row in csvreader:
        print(row)
```

In .csv files, there are two important attributes – the `delimiter` field and the `quoting` field characters. You can set them while creating reader and writer objects. The `delimiter` is the character that separates or delimits fields (columns) in a CSV file. It acts as a boundary signifier, telling the program where one field ends and the next begins. By default, the `delimiter` field is comma (,)

and the other delimiters are specified with the `delimiter` argument to the reader or writer functions. For example, the following code saves the file with `|` as the delimiter:

```
with open(data_file, newline='') as csvfile, \
    open('temp.csv', 'w', newline='') as tempfile:
    csvreader = csv.reader(csvfile)
    csvwriter = csv.writer(tempfile, delimiter='|')
    for row, i in zip(csvreader, range(10)):
        csvwriter.writerow(row)

# Print the contents of temp.csv
with open('temp.csv', 'r') as tempfile:
    contents = tempfile.read()
    print(contents)
```

Let's now read the entire contents of `temp.csv` and print them; this will show all the delimiters, as they are in the file:

```
Date|Time|Global_active_power|Global_reactive_power|Voltage|Global_
intensity|Sub_metering_1|Sub_metering_2|Sub_metering_3
16/12/2006|17:24:00|4.216|0.418|234.840|18.400|0.000|1.000|17.000
16/12/2006|17:25:00|5.360|0.436|233.630|23.000|0.000|1.000|16.000
16/12/2006|17:26:00|5.374|0.498|233.290|23.000|0.000|2.000|17.000
16/12/2006|17:27:00|5.388|0.502|233.740|23.000|0.000|1.000|17.000
16/12/2006|17:28:00|3.666|0.528|235.680|15.800|0.000|1.000|17.000
16/12/2006|17:29:00|3.520|0.522|235.020|15.000|0.000|2.000|17.000
16/12/2006|17:30:00|3.702|0.520|235.090|15.800|0.000|1.000|17.000
16/12/2006|17:31:00|3.700|0.520|235.220|15.800|0.000|1.000|17.000
16/12/2006|17:32:00|3.668|0.510|233.990|15.800|0.000|1.000|17.000
```

Next, let's discuss `quotechar` and quoting settings, something that makes working with CSV files much easier in Python's CSV module. Imagine that you're working with a CSV file where some data includes commas, such as addresses or sentences. Now, CSV files usually use commas to separate different pieces of data. So, how do you tell the computer that a comma in an address is part of the address and not a separator for a new piece of data? That's where `quotechar` comes in handy! `quotechar` is like a guardian. It wraps around your data like a protective shield. Usually, this shield is a double quotation mark (`"`). So, if your address is 123, Baker Street, it becomes `"123, Baker Street"` in the CSV file. This tells the computer, *"Hey, everything inside these quotes is one piece of data; don't split it up!"* Then, there is `quoting`, which you can think of as the rule maker, deciding where to use this protective shield. The following is the list of existing quoting rules:

- `csv.QUOTE_ALL`: This rule is like wrapping everything in bubble wrap. Every single piece of data, regardless of what it is, gets quotes.

- `csv.QUOTE_MINIMAL`: This is more selective. It's like only putting fragile items in bubble wrap. Here, only data with special characters such as commas get quotes.
- `csv.QUOTE_NONNUMERIC`: Imagine you're packing a box and you decide to wrap only non-metal items. Similarly, this rule wraps only non-numeric data in quotes.
- `csv.QUOTE_NONE`: This is like saying, “*No bubble wrap needed!*”, and it doesn't put anything in quotes, even if there are commas or special characters.

Using these settings helps you make sure your CSV files are easy to understand and work with, no matter what kind of data they contain. It's all about making sure your data stays the way you want it – neatly organized!

Now, let's put this into practice by saving a CSV file with all fields quoted, using a custom quote character.

We want to read a file and write the first ten rows to a new file, `temp.csv`. But this time, we'll use the pipe (`|`) as the delimiter, quote all fields, and use an asterisk (`*`) as the quote character. Here's how we do it:

```
with open(data_file, newline='') as csvfile, \
    open('temp.csv', 'w', newline='') as tempfile:
    csvreader = csv.reader(csvfile)
    csvwriter = csv.writer(tempfile, delimiter='|', quotechar='*',
                           quoting=csv.QUOTE_ALL)
    for row, _ in zip(csvreader, range(10)):
        csvwriter.writerow(row)
```

This code results in `temp.csv` being saved with each field wrapped in asterisks. For instance, a row in the file looks like this:

```
*0007-01-01*|*00:00:00*|*2.58*|*0.136*|*241.97*|*10.6*|*0*|*0*|*0*
```

Remember to read the file with the same arguments; otherwise, the `*` quote character will be treated as part of the field values and printed as follows:

```
['*0007-01-01*', '*00:00:00*', '*2.58*', '*0.136*', '*241.97*',
 '*10.6*', '*0*', '*0*', '*0*']
```

Using the correct arguments with the reader object prints the following:

```
['0007-01-01', '00:00:00', '2.58', '0.136', '241.97', '10.6', '0',
 '0', '0']
```

Now, let's see how we can read CSV files with pandas, another popular Python library.

Working with CSV files with the pandas module

Before working with pandas, you will need to import it:

```
import pandas as pd
```

In pandas, the `read_csv()` function returns a `DataFrame` after reading the CSV file:

```
df = pd.read_csv(data_file)
df.head()
```

The `DataFrame` head is shown here:

	Date	Time	Global_active_power	Global_reactive_power	Voltage	Global_intensity	Sub_metering_1	Sub_metering_2	Sub_metering_3
0	16/12/2006	17:24:00	4.216	0.418	234.84	18.4	0.0	1.0	17.0
1	16/12/2006	17:25:00	5.360	0.436	233.63	23.0	0.0	1.0	16.0
2	16/12/2006	17:26:00	5.374	0.498	233.29	23.0	0.0	2.0	17.0
3	16/12/2006	17:27:00	5.388	0.502	233.74	23.0	0.0	1.0	17.0
4	16/12/2006	17:28:00	3.666	0.528	235.68	15.8	0.0	1.0	17.0

Figure 2.1: The `DataFrame` output

You can see in the preceding output that pandas automatically interpreted the date and time columns as their respective data types. The pandas `DataFrame` can be saved to a CSV file with the `to_csv()` function:

```
df.to_csv('temp1.csv', index=False)
```

When working with CSV files, pandas offers a suite of arguments to customize how you read and write data. Let's dive into some of these arguments and explore how they're used:

- **header:** This argument lets you specify which row to use as the header (i.e., the column names) of your `DataFrame`. If your CSV file doesn't have a header row, you can set this to `None`. It's a great way to ensure that your data is labeled correctly from the get-go.
- **sep:** Short for **separator**, this defines the character that delineates or separates fields in each row. The default is a comma (,), making it perfect for standard CSV files. But if your data uses a different separator, such as a tab or semicolon, `sep` is where you can specify that.
- **names:** Need to define or redefine column names? This argument is your friend. It allows you to set custom column names for each column in your `DataFrame`, offering flexibility if your CSV file lacks headers or if you want to rename them.
- **usecols:** This powerful feature lets you specify which columns to extract from the CSV file. By listing the columns you need, pandas will only read those, skipping the rest. It's particularly useful for working with large datasets when you only need a subset of the data.
- **dtype:** Data types are crucial for analysis, and `dtype` lets you define the types for each column in your `DataFrame`. Whether you're working with integers, floats, or strings, setting `dtype` ensures that your data is interpreted and processed correctly.

Pandas offers these and many other arguments to give you control and flexibility when dealing with CSV files.

Now, let's see how to read data from CSV files with the NumPy module.

Working with CSV files using NumPy

The NumPy module provides two functions to read values from CSV files – `np.loadtxt()` and `np.genfromtxt()`. Just like pandas, you need to import NumPy first:

```
import numpy as np
```

Let's try `np.loadtxt()` on the `temp1.csv` file we created in the preceding section:

```
arr = np.loadtxt('temp1.csv', skiprows=1,
                 usecols=(2,3), delimiter=',')
print(arr)
```

This code reads columns three and four from the file that we created earlier, saving them in an array, as follows:

```
[4.216  0.418]
[5.36   0.436]
[5.374  0.498]
[5.388  0.502]
[3.666  0.528]
[3.52   0.522]
[3.702  0.52 ]
[3.7    0.52 ]
[3.668  0.51 ]
[3.662  0.51 ]]
```

While `np.loadtxt()` is great for reading CSV files, it hits a roadblock when it comes across missing data. It simply doesn't know what to do with missing values. That's where `np.genfromtxt()` steps in; this function is specifically designed to handle such situations with ease. It not only reads CSV files but also gracefully manages missing data. It has arguments that you can use to define what to do with those missing parts. You can tell it to treat specific symbols (such as `?`) as missing values and even instruct it on what value to fill in those gaps, such as `np.nan`. Here's how you can rewrite the preceding code using `np.genfromtxt()` to read data, gracefully handling any missing values:

```
arr = np.genfromtxt(
    'temp1.csv', skip_header=1, usecols=(2,3),
    delimiter=',', missing_values='?', filling_values=np.nan
)
```


NumPy arrays produced as a result of applying AI to IoT data can be saved with `np.savetxt()`. For example, the array we loaded previously can be saved as follows:

```
np.savetxt('temp2.csv', arr, delimiter=',')
```

The `np.savetxt()` function also accepts various other useful arguments, such as the format for saved fields and headers. Check the NumPy documentation for more details on this function.

CSV is the most popular data format on IoT platforms and devices. In this section, we learned how to read CSV data using three different packages in Python. Let's learn about XLSX, another popular format, in the next section.

Data in the XLSX Format

Excel, a component of the Microsoft Office package, is one of the popular formats in which data is stored and visualized. Since 2010, Office has supported the `.xlsx` format. We can read XLSX files using the OpenPyXL and pandas functions.

Using OpenPyXL for XLSX files

OpenPyXL is a Python library for reading and writing Excel files. It is an open source project. A new workbook is created using the following command:

```
wb = Workbook()
```

We can access the currently active sheet by using the following command:

```
ws = wb.active()
```

To change the sheet name, use the `title` command:

```
ws.title = "Demo Name"
```

A single row can be added to the sheet using the `append` method:

```
ws.append()
```

A new sheet can be created using the `create_sheet()` method. An individual cell in the active sheet can be created using the column and row values:

```
# Assigns the cell corresponding to # column A and row 10 a value of 5
ws['A10'] = 5
#or
ws.cell(column=1, row=10, value=5)
```

A workbook can be saved using the save method. To load an existing workbook, we can use the load_workbook method. The names of the different sheets in an Excel workbook can be accessed using get_sheet_names().

The following code creates an Excel workbook with three sheets and saves it; later, it loads the sheet and accesses a cell. The code can be accessed from GitHub at `OpenPyXl_example.ipynb`:

```
# Creating and writing into xlsx file
from openpyxl import Workbook
from openpyxl.utils import get_column_letter
wb = Workbook()
dest_filename = 'empty_book.xlsx'
ws1 = wb.active
ws1.title = "range names"
for row in range(1, 40):
    ws1.append(range(0,100,5))
ws2 = wb.create_sheet(title="Pi")
ws2['F5'] = 2 * 3.14
ws2.cell(column=1, row=5, value= 3.14)
ws3 = wb.create_sheet(title="Data")
for row in range(1, 20):
    for col in range(1, 15):
        _ = ws3.cell(
            column=col, row=row,
            value="{0}".format(get_column_letter(col))
        )
print(ws3['A10'].value)
wb.save(filename = dest_filename)

# Reading from xlsx file
from openpyxl import load_workbook
wb = load_workbook(filename = 'empty_book.xlsx')
sheet_ranges = wb['range names']
print(wb.sheetnames)
print(sheet_ranges['D18'].value)
```

In this section, we used OpenPyXL to read data from an Excel file; the next section will show you how you can read Excel directly using the versatile pandas.

Using pandas with XLSX files

We can load existing .xlsx files with the help of pandas. The read_excel method is used to read Excel files as a DataFrame. This method uses an argument, sheet_name, which is used to specify the sheet we want to load. The sheet name can be specified either as a string or a number starting from 0. The to_excel method can be used to write into an Excel file.

The following code reads an Excel file, manipulates it, and saves it. The code can be accessed from GitHub at `Pandas_excel_example.ipynb`:

```
import pandas as pd
df = pd.read_excel("empty_book.xlsx", sheet_name=0)
df.describe()
result = df * 2
result.describe()
result.to_excel("empty_book_modified.xlsx")
```

In this section, we utilized the `openpyxl` and `pandas` libraries to read and process data available in the XLSX format. Next, we will proceed to work with data in the JSON format.

Working with the JSON Format

JavaScript Object Notation (JSON) is another popular data format in IoT systems. In this section, we will learn how to read JSON data with Python's JSON, NumPy, and pandas packages.

For this section, we will use the `zips.json` file, which contains US ZIP codes with city codes, geolocation details, and state codes. The file can be downloaded at this link: <https://raw.githubusercontent.com/qmmr/mongodb/master/zips.json>. The file has JSON objects recorded in the following format:

```
{ "_id" : "01001", "city" : "AGAWAM", "loc" : [-72.622739,42.070206 ],
  "pop" : 15338, "state" : "MA" }
```

Using JSON files with the JSON module

To load and decode JSON data, use the `json.load()` or `json.loads()` functions. As an example, the following code reads the first 10 lines from the `zips.json` file and prints them nicely:

```
import os
import json
from pprint import pprint

with open(data_file) as json_file:
    for line,i in zip(json_file,range(10)):
        json_data = json.loads(line)
        pprint(json_data)
```

The objects are printed as follows:

```
{'_id': '01001',
 'city': 'AGAWAM',
 'loc': [-72.622739, 42.070206],
 'pop': 15338,
 'state': 'MA'}
```

The `json.loads()` function takes string objects as input while the `json.load()` function takes file objects as input. Both functions decode the JSON object and load it in the `json_data` file as a Python dictionary object.

The `json.dumps()` function takes an object and produces a JSON string, and the `json.dump()` function takes an object and writes the JSON string to the file. Thus, both these functions do the opposite of the `json.loads()` and `json.load()` functions.

JSON files with the pandas module

JSON strings or files can be read with the `pandas.read_json()` function, which returns a DataFrame or series object. For example, the following code reads the `zips.json` file:

```
import pandas as pd
df = pd.read_json(data_file, lines=True)
df.head()
```

We set `lines=True` because each line contains a separate object in the JSON format. Without this argument being set to `True`, pandas will raise `ValueError`. The DataFrame is printed as follows:

	city	loc	pop	state	_id
0	ACMAR	[-86.51557, 33.584132]	6055	AL	35004
1	ADAMSVILLE	[-86.959727, 33.588437]	10616	AL	35005
2	ADGER	[-87.167455, 33.434277]	3205	AL	35006
3	KEYSTONE	[-86.812861, 33.236868]	14218	AL	35007
4	NEW SITE	[-85.951086, 32.941445]	19942	AL	35010

Figure 2.2: The pandas DataFrame

To save the pandas DataFrame or series object to a JSON file or string, use the `DataFrame.to_json()` function.

While CSV and JSON remain the most popular data formats for IoT data, due to its large size, it is often necessary to distribute data. There are two popular distributed mechanisms for data storage and access – HDF5 and HDFS. Let's first learn about the HDF5 format.

Handling Data in the HDF5 Format

Hierarchical Data Format (HDF) is a specification put together by the HDF Group, a consortium of academic and industry organizations (<https://www.hdfgroup.org/>). In HDF5 files, data is organized into groups and datasets. A group is a collection of groups or datasets. A dataset is a multidimensional homogeneous array.

In Python, **PyTables** and **h5py** are two major libraries for handling HDF5 files. Both these libraries require HDF5 to be installed. To download and install the latest version of HDF5, visit <https://www.hdfgroup.org/downloads/hdf5/>.

Using HDF5 with PyTables

Let's first create an HDF5 file from the numeric data we have in the `temp.csv` file by following these steps:

1. Get the numeric data:

```
import numpy as np
arr = np.loadtxt('temp.csv', skiprows=1,
                 usecols=(2,3), delimiter=',')
```

2. Open the HDF5 file:

```
import tables
h5filename = 'pytable_demo.hdf5'
with tables.open_file(h5filename,mode='w') as h5file:
```

3. Get the root node:

```
root = h5file.root
```

4. Create a group with `create_group()` or a dataset with `create_array()`, and repeat this until all the data is stored:

```
h5file.create_array(root, 'global_power', arr)
```

5. Close the file:

```
h5file.close()
```

6. Let's read the file and print the dataset to make sure that it is properly written:

```
with tables.open_file(h5filename,mode='r') as h5file:
    root = h5file.root
```

```
for node in h5file.root:
    ds = node.read()
    print(type(ds), ds.shape)
    print(ds)
```

We get the NumPy array back:

figure 2.3: The NumPy array in the table

Using HDF5 with pandas

We can also read and write HDF5 files with pandas. To read HDF5 files with pandas, they must first be created with it. For example, let's use pandas to create an HDF5 file containing global power values:

import pandas as pd

```
import numpy as np
arr = np.loadtxt('temp.csv', skiprows=1, usecols=(2,3), delimiter=',')
import pandas as pd
store = pd.HDFStore('hdfstore_demo.hdf5')
print(store)
store['global_power'] = pd.DataFrame(arr)
store.close()
```

Now, let's read the HDF5 file that we created and print the array back:

```
import pandas as pd
store = pd.HDFStore('hdfstore_demo.hdf5')
print(store)
print(store['global_power'])
store.close()
```

The values of the `DataFrame` can be read in three different ways:

```
store['global_power']
store.get('global_power')
store.global_power
```

`pandas` also provides the high-level `read_hdf()` function and the `to_hdf()` `DataFrame` method to read and write HDF5 files.

More documentation on HDF5 in `pandas` is available at the following link: https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html.

Using HDF5 with `h5py`

The `h5py` module is the most popular way to handle HDF5 files in Python. A new or existing HDF5 file can be opened with the `h5py.File()` function. After the file is open, its groups can simply be accessed by subscripting the file object as if it were a dictionary object. For example, the following code opens an HDF5 file with `h5py` and then prints the array stored in the `/global_power` group:

```
import h5py
hdf5file = h5py.File('pytable_demo.hdf5')
ds=hdf5file['/global_power']
print(ds)
for i in range(len(ds)):
    print(arr[i])
hdf5file.close()
```

The `arr` variable prints an HDF5 dataset type:

```
<HDF5 dataset "global_power": shape (10, 2), type "<f8"> [4.216 0.418]
[5.36  0.436]
[5.374 0.498]
[5.388 0.502]
[3.666 0.528]
[3.52  0.522]
[3.702 0.52 ]
[3.7   0.52]
[3.668 0.51 ]
[3.662 0.51 ]
```

For a new `hdf5file`, datasets and groups can be created by using the `hdf5file.create_dataset()` function, which returns the dataset object, and the `hdf5file.create_group()` function, which returns the folder object. The `hdf5file` file object is also a folder object,

representing /, the root folder. Dataset objects support array-style slicing and dicing to set or read values from them. For example, the following code creates an HDF5 file and stores one dataset:

```
import numpy as np
arr = np.loadtxt('temp.csv', skiprows=1, usecols=(2,3), delimiter=',')
import h5py
hdf5file = h5py.File('h5py_demo.hdf5', 'w')
dataset1 = hdf5file.create_dataset('global_power', data=arr)
hdf5file.close()
```

h5py provides an `attrs` proxy object with a dictionary-like interface to store and retrieve metadata about a file, folders, and datasets. For example, the following code sets and then prints the dataset and file attribute:

```
dataset1.attrs['owner']='City Corp.'
print(dataset1.attrs['owner'])
hdf5file.attrs['security_level']='public'
print(hdf5file.attrs['security_level'])
```

For more information about the h5py library, refer to the documentation at the following link: <http://docs.h5py.org/en/latest/index.html>.

So far, we have learned about different data formats. Often, large data is stored commercially in databases; therefore, we will explore how to access both SQL and NoSQL databases next.

Object Store for Raw Data

Raw data, often voluminous and in its most unprocessed form, presents unique challenges in storage and accessibility. **Object storage** addresses these challenges head-on, offering a scalable and cost-effective solution for storing data in a distributed environment. Unlike traditional filesystems, object storage manages data as distinct units, or “**objects**,” each accompanied by rich metadata, enabling more efficient organization and retrieval.

In this section, we will explore how object storage operates, its advantages over conventional storage methods, and its role in managing the deluge of raw data generated in today’s data-driven world. We’ll cover popular object storage solutions such as Amazon S3, Google Cloud Storage, and Azure Blob Storage, alongside open source alternatives.

Object storage represents a paradigm shift in data storage technology, primarily designed for handling large volumes of unstructured data. Unlike traditional filesystems or block storage that manage data in a hierarchy of files or as blocks within volumes, object storage treats data as distinct units, called “**objects**”.

Object storage uses a flat namespace, which means all objects are stored in a single, massive pool without a hierarchical structure. This approach significantly simplifies data management at scale. Each object in object storage contains not just the data itself but also a rich set of metadata and a globally

unique identifier. The metadata can include details such as the date of creation, access permissions, and content type, enabling more sophisticated data management and retrieval strategies. Interaction with object storage is typically done through *RESTful APIs*, using standard HTTP methods (GET, PUT, and DELETE). This makes object storage highly compatible and accessible over the internet.

Object storage can *easily scale* to store petabytes of data or more, with minimal performance degradation. Its flat structure is inherently more scalable than hierarchical filesystems. Data is often replicated across multiple physical locations, providing protection against data loss and high data durability. Due to its simplicity and scalability, object storage tends to be more *cost-effective*, especially for large datasets. It is typically billed based on usage, which can result in lower costs compared to provisioning blocks or volumes. With its *API-driven access*, object storage can be accessed from anywhere over the internet. This makes it ideal for cloud-based applications and organizations with distributed data access needs. The use of *rich metadata* allows for more flexible and powerful ways to manage and retrieve data, supporting a variety of applications, from big data analytics to cloud-native applications. Many object storage systems offer *versioning and immutability features*, which are crucial for data compliance and protection against accidental deletions or modifications.

To summarize, object storage provides a highly scalable, durable, and flexible solution for storing vast amounts of unstructured data, making it a cornerstone of modern data storage strategies, particularly in cloud environments.

There are several object storage solutions, each offering unique features and capabilities. In the following table, we compare some of the most popular object storage solutions – Amazon S3, Google Cloud Storage, Azure Blob Storage, and open source alternatives such as MinIO and Ceph.

Object storage service	Notable features
Amazon S3	Extensive integrations
Google Cloud Storage	Data analytics integration
Azure Blob Storage	Azure ecosystem integration
MinIO	S3 compatibility and open source
Digital Ocean	Offers an S3-compatible service called Flexify.io
Ceph	Self-healing capabilities and open source

Table 2.1: Object storage services and their features

In this section, we explored the concept of object storage, examining various online services that offer object stores and discussing their benefits. In the following section, we will shift our focus to SQL data.

Working with Structured SQL Data

Most databases are organized using relational models. A relational database consists of one or more related tables of information, and the relationship between information in different tables is described using keys. Conventionally, these databases are managed using the **Database Management System**

(DBMS), software that interacts with end users, different applications, and the database itself to capture and analyze data. Commercially available DBMSes use **Structured Query Language (SQL)** to access and manipulate databases. We can also use Python to access relational databases. In this section, we will explore SQLite and MySQL, two very popular database engines that work with Python.

The SQLite database engine

According to the **SQLite** home page (<https://sqlite.org/index.html>), SQLite is a self-contained, high-reliability, embedded, full-featured, public-domain SQL database engine.

SQLite is optimized for use in embedded applications. It is simple to use and quite fast. We need to use the `sqlite3` Python module to integrate SQLite with Python. The `sqlite3` module is bundled with Python 3, so there is no need to install it.

We will use synthetically created data on soccer players for demonstrative purposes. You can download the data from the GitHub page of this book. We assume that you already have a SQL server installed and started:

1. The first step after importing `sqlite3` is to create a connection to the database using the `connect` method:

```
import sqlite3
import pandas as pd
connection = sqlite3.connect('soccer_database.sqlite')
print("Database opened successfully")
```

2. The synthetic soccer database consists of four tables. We can use `read_sql` to read the database table or SQL query into the DataFrame. This prints a list of all the tables in the database:

```
tables = pd.read_sql(
    "SELECT * FROM sqlite_master WHERE type='table';",
    connection
)
print(tables)
```

3. Let's read data from the team table:

```
teams = pd.read_sql("SELECT * FROM teams;", connection)
teams.head()
```

4. We can use SQL queries on tables. In the following example, we select players whose height is greater than or equal to 180, and whose weight is greater than or equal to 90:

```
selected_players = pd.read_sql_query(
    "SELECT*FROM players WHERE height>=180 AND weight>=90 ",
    connection
)
print(selected_players)
```

5. Finally, do not forget to close the connection using the `close` method:

```
connection.close()
```

If you made any changes in the database, you will need to use the `commit()` method.

The MySQL database engine

Although we can use SQLite for large databases, **MySQL** is generally preferred. In addition to being scalable for large databases, MySQL is also useful where data security is paramount. Before using MySQL, you will need to install the Python MySQL connector. There are many possible Python MySQL connectors, such as MySQLdb, PyMySQL, and MySQL; we will use `mysql-connector-python`.

In all three, after making a connection using the `connect` method, we define the `cursor` element and use the `execute` method to run different SQL queries.

1. To install MySQL, we use the following command:

```
pip install mysql-connector-python
```

2. Now that the Python MySQL connector is installed, we can start a connection with the SQL server. Replace the host, user, and password configurations with your SQL server configuration:

```
import mysql.connector
connection = mysql.connector.connect(
    host = "127.0.0.1",      # your host
    user = "root",          # username
    password = "*****"    # password
)
```

3. Let's check existing databases in the server and list them. To do this, we use the `cursor` method:

```
mycursor = connection.cursor()
mycursor.execute("SHOW DATABASES")
for x in mycursor:
    print(x)
```

4. We can access one of the existing databases. Let's list the tables in one of the databases:

```
connection = mysql.connector.connect(
    host="127.0.0.1",      # your host
    user="root",          # username
    password="*****",    # replace with your password
    database = 'mysql'
)
mycursor = connection.cursor()
mycursor.execute("SHOW TABLES")
```

```
for x in mycursor:
    print(x)
```

In this section, we've learned how to interact with MySQL databases using Python. We've seen how to establish a connection to the MySQL server, list the databases, and retrieve the tables within a database. Understanding these operations is essential for managing and querying data efficiently in RDBMS. As we move forward, we will pivot our attention to NoSQL data.

NoSQL data

The **Not Only Structured Query Language (NoSQL)** database is not a relational database; instead, data can be stored in the key-value, JSON, document, columnar, or graph formats.

They are frequently used in big data and real-time applications. We will learn here how to access NoSQL data using MongoDB, and we assume that you have the MongoDB server configured properly and on:

1. We will need to establish a connection with the Mongo daemon using the **MongoClient** object. The following code establishes the connection to the default host, `localhost`, and port (27017), and it also gives us access to the database:

```
from pymongo import MongoClient
client = MongoClient()
db = client.test
```

2. In this example, we will try to load the cancer dataset available in scikit-learn to the Mongo database. So, we will first get the breast cancer dataset and convert it into a pandas DataFrame:

```
from sklearn.datasets import load_breast_cancer
import pandas as pd
cancer = load_breast_cancer()
data = pd.DataFrame(cancer.data, columns=[cancer.feature_names])
data.head()
```

3. Next, we convert this into the JSON format, use the `json.loads()` function to decode it, and insert the decoded data into the open database:

```
import json
data_in_json = data.to_json(orient='split')
rows = json.loads(data_in_json)
db.cancer_data.insert(rows)
```

4. This will create a collection named `cancer_data` that contains the data. We can query the document we just created, using the `cursor` object:

```
cursor = db['cancer_data'].find({})
df = pd.DataFrame(list(cursor))
print(df)
```

When it comes to distributed data on the IoT, **Hadoop Distributed File System (HDFS)** is another popular method for providing distributed data storage and access in IoT systems. In the next section, we will study how to access and store data in HDFS.

HDFS for Large-Scale Data Storage and Retrieval

HDFS is a popular storage and access method for storing and retrieving data files for IoT solutions. The HDFS format can hold large amounts of data in a reliable and scalable manner. Its design is based on the Google File System (<https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf>). HDFS splits individual files into fixed-size blocks that are stored on machines across a cluster. To ensure reliability, it replicates the file blocks and distributes them across the cluster; by default, the replication factor is 3. HDFS has two main architecture components:

- The first, **NodeName**, stores the metadata for the entire filesystem, such as filenames, their permissions, and the location of each block of each file.
- The second, **DataNode** (one or more), is where file blocks are stored. It performs **Remote Procedure Calls (RPCs)** using **protobufs**.

Note

gRPC is a high-performance, open source, and efficient framework for building RPCs between services. It's designed for efficient and scalable communication between applications. Learn more about gRPC at <https://grpc.io/docs/what-is-grpc/introduction/>.

There are many options to programmatically access HDFS in Python, such as `snakebite`, `pyarrow`, `hdfs3`, `pywebhdfs`, and `hdfsccli`. In this section, we will primarily focus on libraries that facilitate interaction with HDFS. While we will consider libraries such as `hdfs3` that provide native RPC client interfaces for direct communication with HDFS, we will also include `PyArrow`, known for its high-level interface and compatibility with modern Python versions, particularly Python 3. Both libraries offer unique approaches to working with HDFS, catering to different needs and scenarios in Python-based applications.

Using `hdfs3` with HDFS

`hdfs3` is a lightweight Python wrapper around the C/C++ `libhdfs3` library. It allows us to use HDFS natively from Python. To start, we first need to connect with the HDFS `NameNode`; this is done using the `HDFFileSystem` class:

```
from hdfs3 import HDFFileSystem
hdfs = HDFFileSystem(host = 'localhost', port=8020)
```

This automatically establishes a connection with NameNode. Now, we can access a directory listing using the following:

```
print(hdfs.ls('/tmp'))
```

This will list all the files and directories in the `tmp` folder. You can use functions such as `mkdir` to make a directory and `cp` to copy a file from one location to another. To write into a file, we open it first using the `open` method and use `write`:

```
with hdfs.open('/tmp/file1.txt', 'wb') as f:
    f.write(b'You are Awesome!')
```

Data can be read from the file:

```
with hdfs.open('/tmp/file1.txt') as f:
    print(f.read())
```

You can learn more about `hdfs3` from its documentation: <https://media.readthedocs.org/pdf/hdfs3/latest/hdfs3.pdf>.

Using PyArrow's filesystem interface for HDFS

PyArrow has a C++-based interface for HDFS. By default, it uses `libhdfs`, a JNI-based interface, for the Java Hadoop client. Alternatively, we can also use `libhdfs3`, a C++ library for HDFS. We connect to NameNode using `hdfs.connect`:

```
import pyarrow as pa
hdfs = pa.hdfs.connect(host='hostname', port=8020, driver='libhdfs')
```

If we change the driver to `libhdfs3`, we will use the C++ library for HDFS from Pivotal Labs. Once the connection to the NameNode is made, the filesystem is accessed using the same methods as for `hdfs3`.

HDFS is preferred when data is extremely large. It allows us to read and write the data in chunks, which is helpful for accessing and processing streaming data.

Edge Data Storage and Processing

In an era where data generation is ubiquitously distributed, edge data storage and processing becomes a necessity. Edge computing brings data storage and computational power closer to the location where it is needed, aiming to reduce latency and bandwidth use.

Edge data storage and processing involves capturing, storing, and processing data near the source of data generation, rather than relying on a centralized data-processing warehouse. This “edge” can be anything from a mobile or IoT device to a local edge server. Instead of sending vast amounts of raw data

back to a centralized or cloud-based server, edge computing devices process data locally or in nearby edge servers. This approach allows for real-time data processing, crucial for time-sensitive applications.

IoT devices generate a significant amount of data. Edge computing enables these devices to process data in real time, leading to faster and more efficient IoT systems. In autonomous vehicles, edge computing allows the immediate processing of sensor data in vehicles, critical for safety and navigation decisions. And in smart city applications, edge computing helps to manage everything from traffic control to energy management in real time.

Summary

This chapter dealt with many different data formats and, in the process, many different datasets. We started with the simplest TXT data and accessed the Shakespeare play data. We learned how to read data from CSV files using the `csv`, `NumPy`, and `pandas` modules. We moved on to the JSON format; we used Python's JSON and `pandas` modules to access JSON data. From data formats, we progressed to accessing databases and covered both SQL and NoSQL databases. Then, we learned how to work with HDFS in Python. In our pursuit of understanding modern data storage solutions, we turned our attention to object storage, a crucial component in managing vast amounts of unstructured data. We examined popular object storage solutions such as Amazon S3, Google Cloud Storage, and Azure Blob Storage, along with open source alternatives such as MinIO and Ceph. Finally, we ventured into the cutting-edge territory of edge data storage and processing.

Accessing data is the first step. In the next chapter, we will learn about machine learning tools that will help us to design, model, and make informed predictions on data.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/ds>



Machine Learning for IoT

Building on the foundational IoT and data-access concepts covered in the previous chapters, this chapter now focuses on how to analyze and derive insights from IoT data using standard ML algorithms. By the end of the chapter, you will understand the role of ML in the IoT pipeline, learn both supervised and unsupervised paradigms, and gain hands-on experience implementing popular ML techniques with Keras.

After reading this chapter, you will know about the following:

- Understanding ML and its Role in IoT
- Learning Paradigms
- Prediction using Linear Regression
- Logistic Regression for Classification
- Classification using Support Vector Machines
- Naive Bayes
- Decision Trees
- Ensemble Learning
- Improving your Model – Tips and Tricks

Technical requirements

To run the code in this chapter, you will need the following Python (>3.8) modules:

- pandas 1.5
- NumPy 1.25
- scikit-learn 1.2.x
- Matplotlib 3.x
- Keras 3.x

The codes can be accessed on GitHub at <https://github.com/PacktPublishing/Hands-On-Artificial-Intelligence-for-IoT-Second-Edition/tree/main/Chapter03>.

Understanding ML and its Role in IoT

In the ever-evolving landscape of technology, ML – a pivotal subset of artificial intelligence – continues to redefine the boundaries of automated data analysis and decision-making. ML empowers computer programs to learn and improve from experience autonomously, without explicit programming. In an era overwhelmed by the deluge of big data, the sheer volume and velocity of information generated outstrips human capacity for manual analysis. This reality underscores the indispensability of ML in deciphering and leveraging the vast data streams of our times.

A recent report by IoT Analytics, *State of IoT - Spring 2023*, highlights the burgeoning expansion of IoT connections. In 2022 alone, global IoT connections surged by eighteen percent to 14.3 billion active endpoints, with a forecasted growth of sixteen percent in 2023, potentially reaching 16.7 billion active endpoints. This exponential growth trajectory is echoed in the Ericsson Mobility Report (you can find the latest report here: [These figures not only signify the rapid expansion of IoT but also reflect the symbiotic relationship between IoT and ML. As IoT platforms generate colossal data streams, ML emerges as the quintessential tool for extracting meaningful insights and enabling intelligent decision-making in real time.](https://iotbusinessnews.com/2024/09/04/26399-state-of-iot-2024-number-of-connected-iot-devices-growing-13-to-18-8-billion-globally/#:~:text=According%20to%20IoT%20Analytics'%20171,of%2015%25%20over%202022), which anticipates global connected devices to reach a staggering 29 billion by 2030 – a near doubling from the 15.1 billion recorded in 2020. Similarly, Statista projects the proliferation of over eight billion IoT devices across various industries by 2030.</p></div><div data-bbox=)

The key drivers of this growth encompass the following:

- **Technological advancements:** The advent of 5G and subsequent generations promises enhanced connectivity, facilitating more efficient and reliable data transmission. Innovations in low-power sensor technology are revolutionizing data collection capabilities. An increasing affordability curve for devices and connectivity solutions is catalyzing widespread adoption.
- **Sector-specific demands:** Smart homes and appliances, wearable devices for health monitoring, industrial automation, predictive maintenance, connected transportation, and smart city initiatives are at the forefront of this expansion.

Concurrently, specific research areas have gained prominence, addressing the nuanced challenges and opportunities within the IoT and ML realms:

- **Security and privacy:** With the rising complexity and scale of IoT networks, safeguarding user data and mitigating vulnerabilities is paramount. The establishment of robust authentication and authorization protocols is crucial.

- **Interoperability and standards:** Ensuring seamless communication between diverse devices and platforms is critical. This involves the development of universal standards for data exchange and communication.
- **AI and ML integration:** Enhancing data analysis capabilities and advancing predictive maintenance and automation are key focus areas, leveraging the strengths of AI and ML in processing vast and complex data sets.
- **Sustainable IoT:** Emphasizing energy efficiency and environmental responsibility, this area focuses on optimizing energy consumption and advocating for renewable energy sources in IoT infrastructures.

As we ride the crest of this technological wave, it's vital to understand that we are not merely observers but active participants in shaping the future of ML and IoT. This book aims to demystify these technologies, offering insights into their current states, potential trajectories, and impacts on various sectors. Prepare to embark on an enlightening journey into the heart of ML and IoT, technologies that are not only redefining our present but also charting the course of our future.

Note

In case you are interested in reading more about the proliferation of IoT, here are some references:

1. Ericsson Mobility Report: <https://www.ericsson.com/en/reports-and-papers/mobility-report>
2. State of IoT Spring 2023 (the report is behind a paywall): <https://iot-analytics.com/product/state-of-iot-spring-2023/>

Learning Paradigms

Machine learning algorithms can be grouped in various ways based on how they handle data, uncertainty, or the learning process. Understanding these paradigms is crucial because it helps you select the most suitable approach for your IoT problem. For instance, if your data is labeled and you have a clear prediction goal, you might choose supervised methods; if you want to discover hidden patterns in unlabeled data, unsupervised methods could be more appropriate. ML algorithms can be classified based on the method they use, as follows:

- **Probabilistic versus non-probabilistic:** **Probabilistic algorithms** are those that incorporate uncertainty by using probability distributions to make predictions or decisions, such as **Bayesian networks**, where predictions are based on the likelihood and probability of certain outcomes. On the other hand, **non-probabilistic algorithms** do not rely on probability for predictions, instead using deterministic approaches, such as **support vector machines (SVMs)**, which classify data without estimating probability distributions.

- **Modeling versus optimization:** **Modeling algorithms** focus on constructing a mathematical representation of the data to understand or predict phenomena, such as **decision trees** that model decision paths. **Optimization algorithms** are designed to find the most efficient solution or optimal parameters, such as the **gradient descent** algorithms that are used in neural network training to minimize error.
- **Supervised versus unsupervised:** **Supervised learning algorithms** learn from labeled data, mapping inputs to known outputs, exemplified by **linear regression** for predicting continuous values. Conversely, **unsupervised learning algorithms** discover patterns or groupings in unlabeled data, such as **K-means clustering** that groups data based on similarities without prior labeling.

In this book, we classify our ML algorithms as supervised versus unsupervised. The distinction between these two depends on how the model learns and the type of data that's provided to the model to learn. Let's look at both these algorithms in more depth in the following headings.

Supervised learning

Let's say I give you a series and ask you to predict the next element:

```
(1, 4, 9, 16, 25, ...)
```

You guessed right: the next number will be 36, followed by 49, and so on. This is supervised learning, also called learning by example; you weren't told that the series represents the square of positive integers—you were able to guess it from the five examples provided.

In a similar manner, in supervised learning, the machine learns from example. It's provided with training data consisting of a set of pairs (X, Y) where X is the input (it can be a single number or an input value with a large number of features) and Y is the expected output for the given input. Once trained on the example data, the model should be able to reach an accurate conclusion when presented with new data.

Supervised learning is used to predict, given a set of inputs, either a real-valued output (regression) or a discrete label (classification). We'll explore both regression and classification algorithms in the coming sections.

Unsupervised learning

Let's say you were given eight circular blocks of different radii and colors, and you are asked to arrange or group them in an order. What would you do?

Some may arrange them in increasing or decreasing order of radii, some may group them according to color. There are so many ways, and for each of us, the way we choose will be dependent on what internal representation of the data we had while grouping. This is unsupervised learning, and most human learning is in this category.

In unsupervised learning, the model is just given the data (X) but isn't told anything about it; the model learns the underlying patterns and relationships in the data by itself. Unsupervised learning is normally used for clustering and dimensionality reduction.

Now that we have learned the basic learning paradigms, let us move ahead and see how they can be used in different scenarios. We will start with prediction using linear regression – a supervised learning methodology.

Prediction using Linear Regression

Aaron, a friend of mine, is a little sloppy with money and is never able to estimate how much his monthly credit card bill will be. Can we do something to help him? Well, yes, linear regression can help us to predict a monthly credit card bill if we have sufficient data. Thanks to the digital economy, all of his monetary transactions for the last five years are available online. We extracted his monthly expenditure on groceries, stationery, and travel, as well as his monthly income. Linear regression helped not only in predicting his monthly credit card bill but also by giving insight into which factor was most responsible for his spending.

This was just one example; linear regression can be used in many similar tasks. In this section, we'll learn how we can perform linear regression on our data.

Linear regression is a supervised learning task. It's one of the most basic, simple, and extensively used ML techniques for prediction. The goal of regression is to find a function $F(x, W)$, for a given input-output pair (x, y) , so that $y = F(x, W)$. In the (x, y) pair, x is the independent variable and y is the dependent variable. Both are continuous variables. It helps us to find the relationship between the dependent variable y and the independent variable(s) x .

The input x can be a single input variable or many input variables. When $F(x, W)$ maps a single input variable x , it's called **simple linear regression**; for multiple input variables, it's called **multiple linear regression**.

The function $F(x, W)$ is approximated using the following expression:

$$y_i \approx F(x_i, W) = W_0 + \sum_{j=1}^d x_{ij} W_j$$

In this expression, d is the dimensions of x (number of independent variables), and W is the weight associated with each component of x . To find the function $F(x, W)$, we need to determine the weights. The natural choice is to find the weights that reduce the squared error; hence, our objective function is as follows:

$$\mathcal{L} = \sum_{i=1}^N (y_i - F(x_i, W))^2$$

In the preceding function, N is the total number of the input-output pair presented. To find the weights, we differentiate the objective function with respect to weight and equate it to 0. In matrix notation, we can write the solution for the column vector $W = (W_0, W_1, W_2, \dots, W_N)^T$ as follows:

$$\nabla_w \mathcal{L} = 0$$

On differentiating and simplifying, we get the following:

$$W = (X^T X)^{-1} X^T Y$$

X is the input vector of size $[N, d]$ and Y the output vector of size $[N, 1]$. The weights can be found if $(X^T X)^{-1}$ exists, that's if all of the rows and columns of X are linearly independent. To ensure this, the number of input-output samples (N) should be much greater than the number of input features (d).

Regression using deep learning

After exploring the fundamentals of traditional linear regression, we now shift our focus to a more advanced modeling approach using deep learning. Although both methods share the common goal of predicting a continuous target variable, the deep learning approach introduces additional complexity and flexibility. In deep learning, regression tasks are approached through the construction of neural networks, architectures capable of capturing intricate relationships in data. These networks consist of multiple layers of neurons, each layer capable of learning different aspects of the data through a process of weighted inputs and activation functions. This structure allows deep learning models to approximate complex, non-linear functions, extending far beyond the capabilities of traditional linear regression models.

The strength of deep learning in regression lies in its ability to model complex, non-linear relationships that traditional methods cannot capture effectively. This makes it particularly useful for large datasets with numerous input features or intricate interactions between variables, or when the data does not meet the assumptions of linear regression.

Electrical power output prediction using regression

Now that you understand the basics of linear regression, let's use it to predict the electrical power output of a combined cycle power plant. We described this dataset in *Chapter 1, Principles and Foundations of AI and IoT*; here, we'll use Keras3 and its automatic gradient to find the solution. The dataset can be downloaded from the UCI ML archive (<http://archive.ics.uci.edu/ml/datasets/combined+cycle+power+plant>).

Let's understand the execution of code in the following steps:

1. We import Keras. We will use it to build our regressor model:

```
import keras as K
from keras.layers import Dense, Input
print(K.__version__)
```

2. We will also require some helpful functions from NumPy, pandas, Matplotlib, and scikit-learn:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import train_test_split
```

3. Next comes data. We will load the data file using pandas. Since I am using Colab, the path of my data file is `'/content/data/CCPP/Folds5x2_pp.xlsx'`. In your system, change the path to your data file:

```
file_path = '/content/data/CCPP/Folds5x2_pp.xlsx'
df = pd.read_excel(file_path, sheet_name='Sheet1')
df.describe()
```

This results in the following output:



	AT	V	AP	RH	PE
count	9568.000000	9568.000000	9568.000000	9568.000000	9568.000000
mean	19.651231	54.305804	1013.259078	73.308978	454.365009
std	7.452473	12.707893	5.938784	14.600269	17.066995
min	1.810000	25.360000	992.890000	25.560000	420.260000
25%	13.510000	41.740000	1009.100000	63.327500	439.750000
50%	20.345000	52.080000	1012.940000	74.975000	451.550000
75%	25.720000	66.540000	1017.260000	84.830000	468.430000
max	37.110000	81.560000	1033.300000	100.160000	495.760000

Figure 3.1: pandas DataFrame

All the data variables under consideration are of continuous value. The variables include **ambient temperature (AT)**, **exhaust vacuum (V)**, **ambient pressure (AP)**, and **relative humidity (RH)**, which collectively constitute the input features for our regression model. The model aims to predict the hourly energy output (**PE**), serving as the target variable for our regression analysis.

4. Let us separate the data into input features X (**independent variables**) and output feature Y (**dependent variable**):

```
X, Y = df[['AT', 'V', 'AP', 'RH']], df['PE']
```

- Next, we scale the dataset using scikit-learn's `MinMaxScaler`. It normalizes the data by scaling each feature to a given range, typically between 0 and 1:

```
scaler = MinMaxScaler()
X_new = scaler.fit_transform(X)
target_scaler = MinMaxScaler()
Y_new = target_scaler.fit_transform(Y.values.reshape(-1,1))
```

- The next step in the data preprocessing pipeline is splitting the data into training a dataset and a test dataset. This allows us to use one set to teach the model (*training dataset*) and the other to check how well it learned (*test dataset*), ensuring it can make accurate predictions on new, unseen data:

```
X_train, X_test, Y_train, y_test = train_test_split(X_new, Y_
new, test_size=0.4, random_state=333)
```

- Now that our data is ready, let us build our model. We are using Keras3 to build a regression model. Since it is a simple model, a single hidden layer can do the trick. To make the task easier, we will use Keras Sequential API:

```
regressor = K.Sequential()
regressor.add(Input(shape=(X_train.shape[1],)))
regressor.add(Dense(3))
regressor.add(Dense(1))
```

In the first line, we initiate the Keras Sequential API. We then add an Input layer, a Dense hidden layer with three units, and finally an output layer with only one unit.

Note

If we use only one dense layer, we will have a linear regressor.

- Now that the model architecture has been built, the next step is to choose the optimizer and loss function we want to optimize. Since it is a regression problem, using the **mean square error (MSE)** as the **objective function** is the natural choice, as discussed in the preceding section. It is important to note that despite a complex architecture, Keras will do **automatic differentiation** of the objective function (MSE) and update all the weights. In the code snippet that follows, we do this using the Keras `compile` method. We select stochastic gradient descent as our optimizer:

```
regressor.compile(optimizer='sgd', loss='mse')
```

- Next, we train the model using the `fit` method:

```
history = regressor.fit(X_train, Y_train, batch_size=32,
epochs=50)
```

10. Ideally, as the model learns, the objective function should decrease, so let us the learning of our model:

```
loss = history.history['loss']
epochs = range(1, len(loss) + 1)
plt.figure(figsize=(10, 6))
plt.plot(epochs, loss, 'b-', label='Training Loss')
plt.title('Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

Here is the plot, showing a fine convergence.

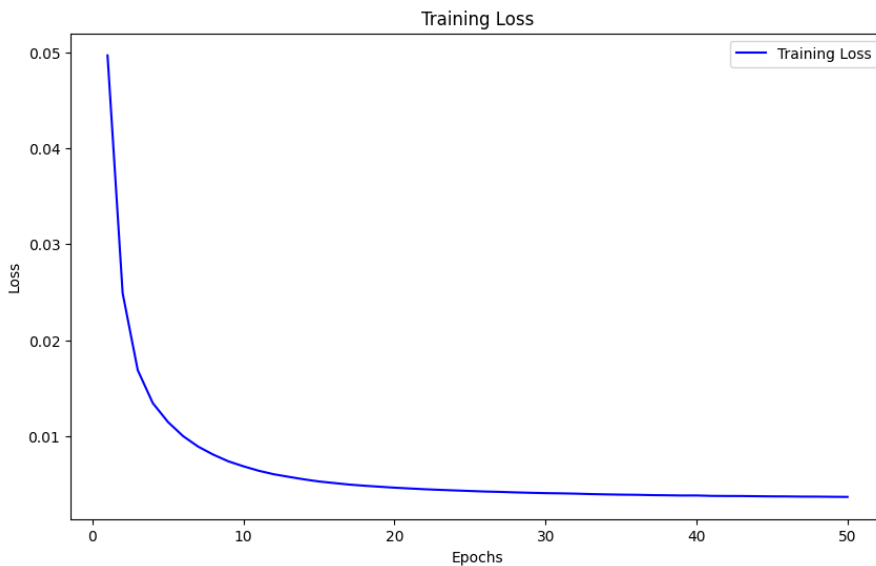


Figure 3.2: Training loss plot

11. Using the predict method, we can now make predictions for unseen data:

```
y_pred = regressor.predict(X_test)
```

12. Here, you can compare the output of the regressor model with the ground truth:

```
x_range = range(len(y_test))
plt.figure(figsize=(12, 6))
plt.plot(x_range, y_test, 'b-', label='Actual Values')
```



```
plt.plot(x_range, y_pred, 'r-', label='Predicted Values')
plt.title('Comparison of Actual vs Predicted Values')
plt.xlabel('Data Point Index')
plt.ylabel('Values')
plt.legend()

plt.show()
```

Here is the plot. You can see that the ground truth and model prediction are similar.

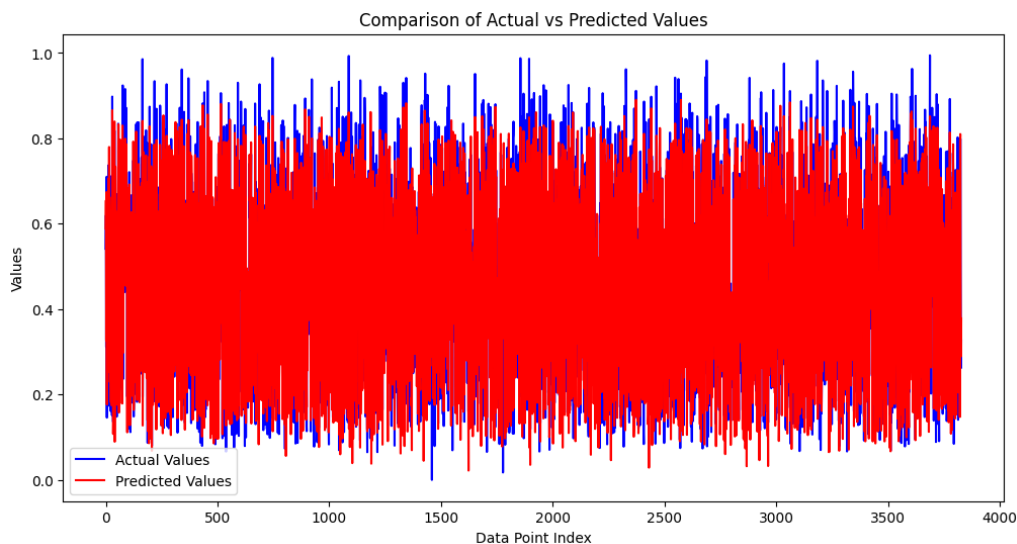


Figure 3.3: Actual and predicted values

On the test dataset, we achieved an R2 value of 0.92 and an MSE of 0.003. These values indicate that our model is quite good at predicting the electrical output power. Next, we will see how we can use a similar approach classification task.

Logistic Regression for Classification

In the previous section, we learned how to predict. There's another common task in ML: the task of classification. Distinguishing between dogs and cats, filtering out spam from legitimate emails, or identifying various objects within a room or scene – these tasks all exemplify classification challenges.

Logistic regression is an old classification technique. It provides the probability of an event taking place, given an input value. The events are represented as categorical dependent variables, and the probability of a particular dependent variable being 1 is given using the logit function:

$$y_{\text{pred}} = P(y = 1|X = x) = \frac{1}{1 + \exp(-(b + W^T X))}$$

Before going into the details of how we can use logistic regression for classification, let's examine the logit function (also called the Sigmoid function because of its S-shaped curve). The following diagram shows the logit function. Its derivative varies with respect to the input X , the Sigmoid function (solid line), and its derivative (dashed line):

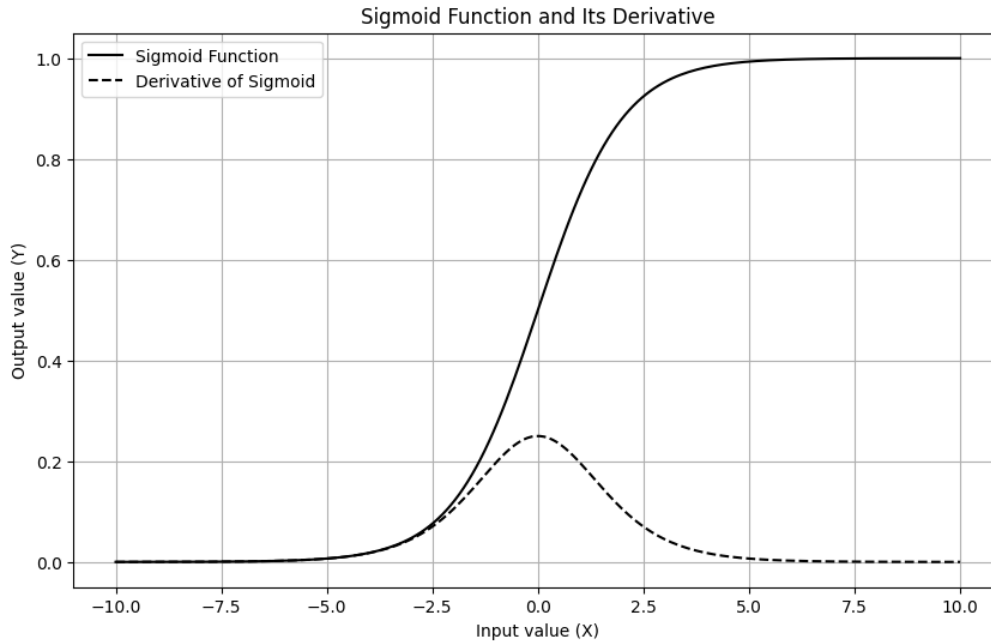


Figure 3.4: Sigmoid function and its derivative

A few important things to note from this diagram include the fact that the value of sigmoid (and hence $\hat{Y}_{pred}$) lies between $(0, 1)$.

Furthermore, the derivative of the sigmoid is highest when $W^T X + b = 0$ and the highest value of the derivative is just 0.25 (the Sigmoid at the same place has a value of 0.5).

The slope by which the sigmoid varies depends on the weights, and the position where we'll have the peak of the derivative depends on the bias.

I would suggest that you play around with the `sigmoid_function.ipynb` program available at this book's GitHub repository, to get a feel of how the Sigmoid function changes as the weight and bias change.

Cross-entropy loss function

Logistic regression aims to find weights W and bias b , so that each input vector, X_i , in the input feature space is classified correctly to its class, Y_i . In other words, Y_i and y_pred_i should have a similar distribution for the given X_i . We first consider a binary classification problem; in this case, the data point Y_i can have a value of 1 or 0. Since logistic regression is a supervised learning algorithm, we give as input the training data pair (X_i, Y_i) and let P be the probability that $P(y=1|X=X_i)$. Then, for p training data points, the total average loss is defined as follows:

$$\text{loss} = \frac{1}{P} \sum_{i=1}^P Y_i \log(y_{pred_i}) + (1 - Y_i) \log(1 - y_{pred_i})$$

Hence, for every data pair, for $Y_i = 1$, the first term will contribute to the loss term, with the contribution changing from infinity to 0 as y_pred_i varies from 0 to 1. Similarly, for $Y_i = 0$, the second term will contribute to the loss term, with the contribution changing from infinity to zero as y_pred_i varies from 1 to 0.

For multiclass classification, the loss term is generalized to the following:

$$\text{loss} = \sum_{i=1}^P \sum_{j=1}^K Y_{ij} \log(y\_pred_{ij})$$

In the preceding, K is the number of classes. An important thing to note is that, while the output Y_i and y_pred_i were single values for binary classification, for multiclass problems, both Y_i and y_pred_i are now vectors of K dimensions, with one component for each category.

Classifying wine using logistic regressor

Let's now use what we've learned to classify wine quality. I can hear you thinking: what, wine quality? No way! Let's see how our logistic regressor fares compared to professional wine tasters. We'll be using the wine quality dataset (<https://archive.ics.uci.edu/dataset/186/wine+quality>) ; details about the dataset are given in *Chapter 1*. The full code is in the file named `Wine_quality_using_logistic_regressor.ipynb` in the GitHub repository. Let's understand the code step by step:

1. The first step is loading all of the modules:

```
import keras as K
from keras.layers import Dense, Input
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import (
    accuracy_score, classification_report)
from sklearn.model_selection import train_test_split
```

- We read the data; in the present code, we are analyzing only the red wine, so we read data from the `winequality-red.csv` file. The file contains the data values separated not by commas, but instead by semicolons, so we need to specify the separator argument:

```
file_path = '/content/data/winequality-red.csv'
df = pd.read_csv(file_path, sep=';')
df.head()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5

Figure 3.5: pandas DataFrame

- We separate input features and target quality from the data file. In the file, the target (wine quality) is given on a scale from zero to ten. Here, for simplicity, we divide it into three classes. If the initial quality is less than or equal to four, we sort it into the third class (signifying *bad*); between four and six (including six), we consider it *OK* (second class); and above six, we consider it *good* (first class). We also normalize the input features and split the data into training and test datasets:

```
X, Y = df[columns[0:-1]], df[columns[-1]]
scaler = MinMaxScaler()
X_new = scaler.fit_transform(X)
Y = Y.copy()
Y.loc[(Y<=4)] = 3
Y.loc[(Y<=6) & (Y>4)] = 2
Y.loc[(Y>6)] = 1
X_train, X_test, Y_train, y_test = train_test_split(
    X_new, Y, test_size=0.4, random_state=333)
```

- We will make use of scikit-learn's `LogisticRegressor` here. The `LogisticRegression` class from scikit-learn has many arguments; you can learn about them from its documentation (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html). Here, we are just setting the `random_state` argument so that the results are reproducible:

```
classifier = LogisticRegression(random_state=0)
```

- The next step is simple; we train the classifier on the training dataset:

```
classifier.fit(X_train, Y_train)
```

6. We can now make predictions on the test dataset:

```
y_pred = classifier.predict(X_test)
```

7. Let us see its performance on the test data:

```
acc = accuracy_score(y_test, y_pred)
print(f"Accuracy: {acc*100}%")

cr = classification_report(y_test, y_pred)
print("Classification Report:\n", cr)
```

The learned model gives us an accuracy of approximately 82.5% on the test dataset. That's pretty impressive!

8. We can visualize the features and how they affect the wine quality. The code that follows does that:

```
coefficients = classifier.coef_
feature_names = columns[0:-1]
n_features = coefficients.shape[1]

# Plotting
fig, ax = plt.subplots(figsize=(12, 6))
for i in range(coefficients.shape[0]):
    # Plot each class's coefficients
    ax.bar(
        np.arange(n_features) + i * 0.2, coefficients[i],
        width=0.2, label=f'Class {i+1}'
    )

ax.set_xticks(np.arange(n_features) + 0.2)
ax.set_xticklabels(feature_names)

plt.xticks(rotation=45)
plt.xlabel('Features')
plt.ylabel('Coefficient Value')
plt.title('Feature Importance in Wine Quality')
plt.legend()
plt.show()
```

Here is the feature importance plot:

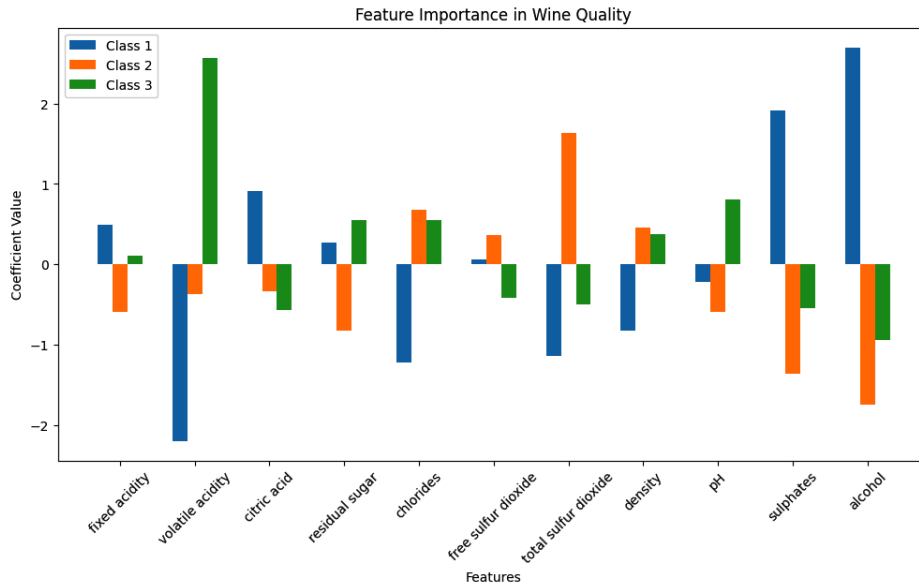


Figure 3.6: Feature importance in wine quality

The interpretation of feature importance necessitates a careful and informed approach. As we can see, for each feature, there are three bars of different colors, each corresponding to one of the three quality classes. This shows how the influence of each feature varies when predicting different quality levels. A feature that is positively associated with one class might be negatively associated with another. Each bar's height and direction (positive or negative) indicate the strength and direction of the relationship between the feature and the likelihood of a wine being in the associated quality class, compared to the other classes. For feature importance derived from model coefficients to be interpretable, it's essential to normalize the input features. This ensures that all features have the same scale of measurement, allowing for a fair and accurate comparison of their respective influences on the model's predictions. Let us now move on to another popular ML algorithm: support vector machines.

Classification using Support Vector Machines

Using **Support Vector Machines (SVMs)** is arguably the most used ML technique for classification. The main idea behind SVMs is that we find an optimal hyperplane with a maximum margin separating the two classes. If the data is linearly separable, the process of finding the hyperplane is straightforward, but if it isn't linearly separable, then the kernel trick is used to make the data linearly separable in some transformed high-dimensional feature space.

SVM is considered a non-parametric supervised learning algorithm. The main idea of SVM is to find a maximal margin separator: a separating hyperplane that is farthest from the training samples presented.

Consider the following diagram. The red dots represent class 1, for which the output should be 1, and the blue dots represent class 2, for which the output should be -1. There can be many lines that can separate the red dots from the blue ones. The diagram demonstrates three such lines: A, B, and C, respectively. Which of the three lines do you think will be the best choice? Intuitively, the best choice is line B, because it's farthest from the examples of both classes and hence ensures the least error in classification:

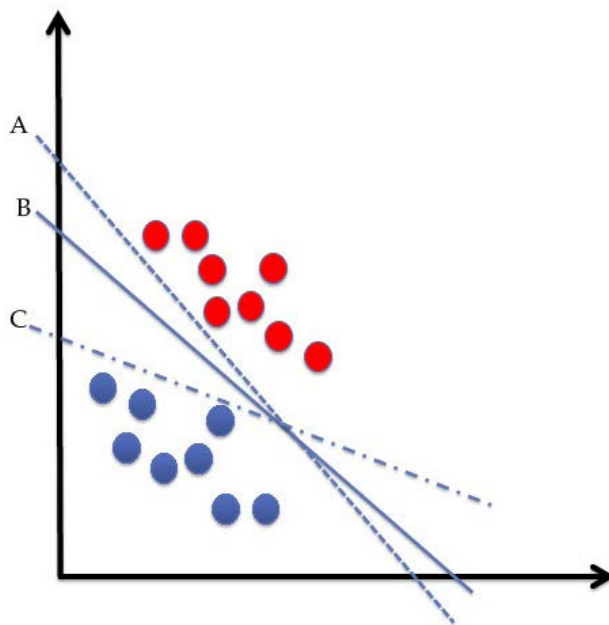


Figure 3.7: Binary classification and separating lines

In the following section, we'll learn the basic math behind finding the maximal-separator hyperplane. Though the math here is mostly basic, if you don't like math, you can simply skip to the implementation section where we use SVM to classify wine again!

Maximum margin hyperplane

From our knowledge of linear algebra, we know that the equation of a plane is given by the following:

$$W_0 + W^T X = 0$$

In SVM, this plane should separate the positive classes ($y = 1$) from the negative classes

($y = -1$). There's an additional constraint: the distance (margin) of this hyperplane from the closest positive and negative training vectors (X_{pos} and X_{neg} , respectively) should be maximum. Hence, the plane is called the maximum margin separator.

Note

The X_{pos} and X_{neg} vectors are called support vectors, and they play an important role in defining the SVM model.

Mathematically, this means that the following is true:

$$W_0 + W^T X_{\text{pos}} = 1$$

This is also true:

$$W_0 + W^T X_{\text{neg}} = -1$$

From these two equations, we get the following:

$$W^T (X_{\text{pos}} + X_{\text{neg}}) = 2$$

Dividing by the weight vector length into both sides, we get the following:

$$\frac{W^T (X_{\text{pos}} + X_{\text{neg}})}{\|W\|} = \frac{2}{\|W\|}$$

So, we need to find a separator such that the margin between positive and negative support vectors is maximum, that is: $2/\|W\|$ is maximum, while at the same time all the points are classified correctly, such as the following:

$$y_i (W^T X_i + b) > 1$$

Using a little math, which we won't go into in this book, the preceding condition can be represented as finding an optimal solution to the following:

$$\operatorname{argmax}_b \sum_j \alpha_j = \frac{1}{2} \sum_{j,k} \alpha_j \alpha_k y_j y_k (X_j X_k)$$

This is subject to the following constraints:

$$\begin{aligned} \alpha_j &\geq 0 \\ \sum_j \alpha_j y_j &= 0 \end{aligned}$$

From the values of alpha, we can get weights W from α , the vector of coefficients, using the following equation:

$$W = \sum_j \alpha_j X_j$$

This is a standard quadratic programming optimization problem. Most ML libraries have built-in functions to solve it, so you need not worry about how to do so.

Note

For those of you who are interested in knowing more about SVMs and the math behind them, the book *The Nature of Statistical Learning Theory* by Vladimir Vapnik, published by Springer Science+Business Media in 2013, is an excellent reference.

Kernel trick

The previous method works fine when the input feature space is linearly separable.

What should we do when it isn't? One simple way is to transform the data (X) into a higher-dimensional space where it's linearly separable and find a maximal margin hyperplane in that high-dimensional space. Let's see how; our hyperplane in terms of α is as follows:

$$W_0 + \sum_i \alpha_i X^T X^{(i)}$$

Let φ be the transform. Then we can replace X with $\varphi(X)$ and hence, its dot product $X^T X^{(i)}$ with a function $K(X^T, X^{(i)}) = \varphi(X^T) \varphi(X^{(i)})$, called **kernel**. So, we now just preprocess the data by applying the transform φ and then find a linear separator in the transformed space as before.

The most used kernel function is the **Gaussian kernel**, also called the **radial basis function**. It is defined as follows:

$$K(X^T, X^{(i)}) = \exp(-\gamma \|X^i - X^j\|^2)$$

Classifying wine using SVM

We'll use the `svm.SVC` function provided by the scikit library for the task. The following code can be found in the `Wine_quality_using_SVM.ipynb`.

Note

The SVC classifier of scikit is a support vector classifier. It can also handle multiclass support using a one-versus-one scheme. Some of the optional parameters of the method are as follows:

- **C**: It's a parameter specifying the penalty term (default value is 1.0).
- **kernel**: It specifies the kernel to be used (the default is `rbf`). The possible choices are linear, poly, rbf, Sigmoid, precomputed, or callable.
- **gamma**: It specifies the kernel coefficient for rbf, poly, and Sigmoid, as well as the default value (the default is `auto`).
- **random_state**: It sets the seed of the pseudo-random number generator to use when shuffling the data.

Follow the given steps to create our SVM model:

1. Let's load all of the modules we'll need for the code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler, LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.svm import SVC
import seaborn as sns
```

2. We read the data file, preprocess it, and separate it into test and training datasets. This time, for simplicity, we're dividing into two classes (good and bad):

```
file_path = '/content/data/winequality-red.csv'
df = pd.read_csv(file_path, sep=';')
bins = (0,5.5,10)
df['quality'] = pd.to_numeric(df['quality'], errors='coerce')
categories = pd.cut(
    df['quality'], bins, labels = ['bad','good'])
df['quality'] = categories
X = df.drop(['quality'], axis = 1)
scaler = MinMaxScaler()
X_new = scaler.fit_transform(X)
y = df['quality']
labelencoder_y = LabelEncoder()
y = labelencoder_y.fit_transform(y)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size = 0.2, random_state = 333)
```

3. Now we use the SVC classifier and train it on our training dataset with the fit method:

```
classifier = SVC(kernel = 'rbf', random_state = 45)
classifier.fit(X_train, y_train)
```

4. Let's now predict the output for the test dataset:

```
y_pred = classifier.predict(X_test)
```

5. The model gave an accuracy of 67.5% and the confusion matrix is as follows:

```
print(f"Accuracy is {accuracy_score(y_test, y_pred)}".format)
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt='2.0f')
```

Here you can see the confusion matrix as a heatmap:

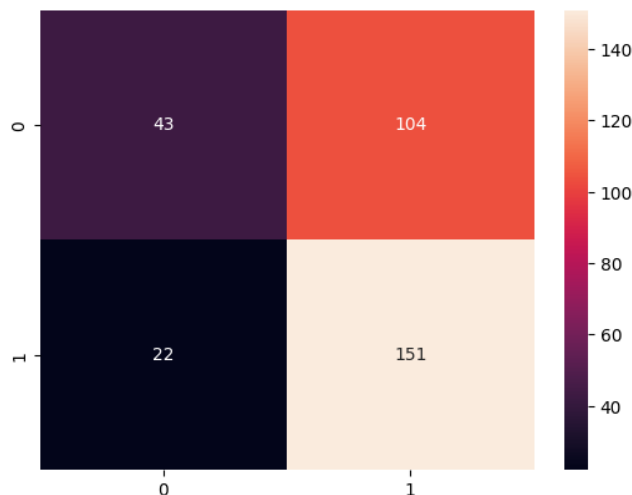


Figure 3.8: Confusion matrix heatmap

The preceding code uses the binary classification; we can change the code to work for more than two classes as well. For example, in the second step, we can replace the code with the following:

```
bins = (0,3.5,5.5,10)
categories = pd.cut(df['quality'], bins, labels = ['bad','ok','good'])
df['quality'] = categories
```

Try it out and see whether adding more categories improves the classifier or degrades its performance. Additionally, you can also work around it by changing the decision boundaries for the categories.

Naive Bayes

Naive Bayes is one of the simplest and fastest ML algorithms. This too belongs to the class of supervised learning algorithms. It's based on the **Bayes probability theorem**. One important assumption that we make in the case of the naive Bayes classifier is that all of the features of the input vector are independent and **identically distributed (iid)**. The goal is to learn a conditional probability model for each class C_k in the training dataset:

$$p(C_k | x_1, x_2, \dots, x_n) = p(C_k | X)$$

Under the iid assumption, and using the Bayes theorem, this can be expressed in terms of the joint probability distribution $p(C_k | X)$:

$$p(C_k | X) \sim p(C_k, X) \sim p(C_k) \prod_{i=1}^n p(x_i | C_k)$$

We pick the class that maximizes this **Maximum A Posteriori (MAP)** term:

$$\underset{k \in \{1, \dots, K\}}{\operatorname{argmaxp}} \left(C_k \right) \prod_{i=1}^n p(x_i | C_k)$$

There can be different naive Bayes algorithms, depending upon the distribution of $p(x_i | C_k)$. The common choices are Gaussian in the case of real-valued data, Bernoulli for binary data, and MultiNomial when the data contains the frequency of a certain event (such as document classification).

Let's now see whether we can classify the wine using naive Bayes. For the sake of simplicity and efficiency, we'll use the built-in scikit naive Bayes distributions. Since the features values, we have in our data are continuous-valued—we'll assume that it has a Gaussian distribution, and we'll use `GaussianNB` from scikit-learn.

Gaussian naive Bayes for wine quality

The scikit-learn naive Bayes module supports three naive Bayes distributions. We can choose either of them depending on our input feature data type. The three naive Bayes available in scikit-learn are as follows:

- `GaussianNB`
- `MultinomialNB`
- `BernoulliNB`

The wine data, as we have already seen, is a continuous data type. Hence, it will be good if we use Gaussian distribution for $p(x_i | C_k)$ —that is, the `GaussianNB` module. Therefore, we'll add `from sklearn.naive_bayes import GaussianNB` in the import cell of the notebook. You can read more details about the `GaussianNB` module from the scikit-learn link:

https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html

The first two steps will remain the same as in the SVM case. Now, instead of declaring an SVM classifier, we'll declare a `GaussianNB` classifier, and we'll use its `fit` method to learn the training examples. The result from the learned model is obtained using the `predict` method. So, follow these steps:

1. Import the necessary modules. Note that we're now importing `GaussianNB` from the `scikit` library:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler, LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.naive_bayes import GaussianNB
import seaborn as sns
```

2. Read the data file and preprocess it:

```

file_path = '/content/data/winequality-red.csv'
df = pd.read_csv(file_path, sep=';')

bins = (0,5.5,10)
df['quality'] = pd.to_numeric(df['quality'], errors='coerce')
categories = pd.cut(
    df['quality'], bins, labels = ['bad','good'])
df['quality'] = categories
X = df.drop(['quality'], axis = 1)
scaler = MinMaxScaler()
X_new = scaler.fit_transform(X)
y = df['quality']
labelencoder_y = LabelEncoder()
y = labelencoder_y.fit_transform(y)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size = 0.2, random_state = 333)

```

3. Now we declare a Gaussian naive Bayes, train it on the training dataset, and use the trained model to predict the wine quality on the test dataset:

```

classifier = GaussianNB()
classifier.fit(X_train, y_train) #Predicting the Test Set
y_pred = classifier.predict(X_test)

```

That's all, folks; our model is ready and kicking. The accuracy of this model is 74.68% for the binary classification case. In the following screenshot, you can see the heatmap of the confusion matrix:

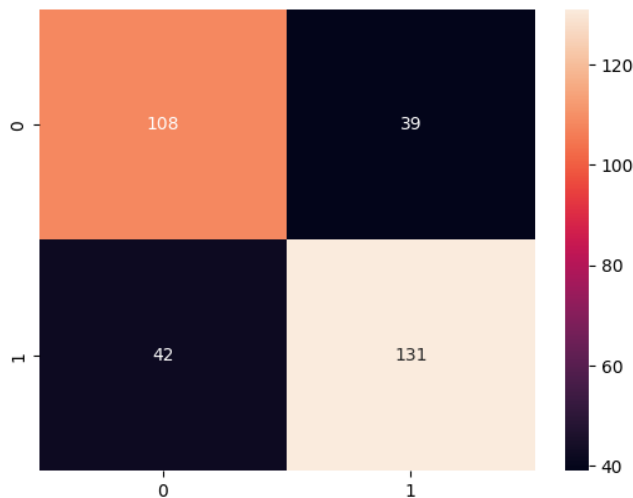


Figure 3.9: Confusion matrix heatmap

Before you conclude that naive Bayes is best, let's be aware of some of its pitfalls:

- Naive Bayes makes the prediction based on the frequency-based probability; therefore, it's strongly dependent on the data we use for training
- Another issue is that we made the iid assumption about input feature space; this isn't always true

Having explored naive Bayes, let's turn our attention to decision trees and learn how to implement them using scikit-learn.

Decision Trees

In this section, you'll learn about another ML algorithm that's very popular and fast—decision trees. In decision trees, we build a tree-like structure of decisions; we start with the root, choose a feature, split into branches, and continue until we reach the leaves, which represent the predicted class or value. The algorithm of decision trees involves two main steps:

1. Decide which features to choose and what conditions to use for splitting
2. Know when to stop

Let's understand it with an example. Consider a sample of forty students. We have three variables: gender (boy or girl; discrete), class (XI or XII; discrete), and height (five to six feet; continuous). Eighteen students prefer to go to the library in their spare time and the rest prefer to play. We can build a decision tree to predict who will be going to the library and who will be going to the playground in their leisure time. To build the decision tree, we'll need to separate the students who go to the library or the playground based on the highly significant input variable among the three input variables. The following diagram gives the split based on each input variable:

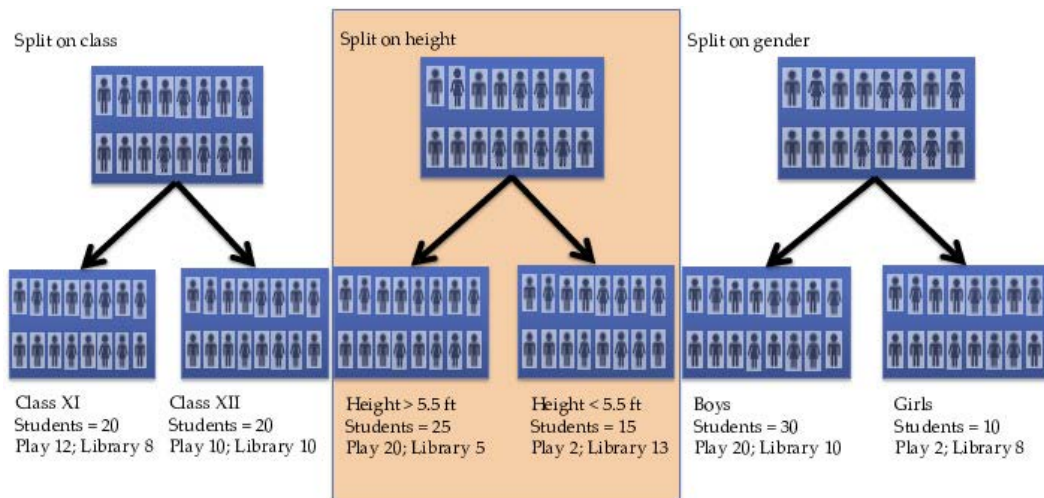


Figure 3.10: Decision tree split based on each input variable

We consider all of the features and choose the one that gives us the maximum information. In the previous example, we can see that a split over the *height* feature generates the most homogeneous groups, with the **Height > 5.5 ft** group containing 80% of students who play and 20% who go to the library in their leisure time. We can also observe that the **Height < 5.5 ft** group contains 13% of students who play and 86% who go to the library in their spare time. Hence, we'll make our first split on the height feature. We'll continue the split in this manner and finally reach the decision (leaf node) telling us whether the student will play or go to the library in their spare time. The following diagram shows the decision tree structure; the black circle is the root node, the blue circles are the decision nodes, and the green circles are the leaf nodes:

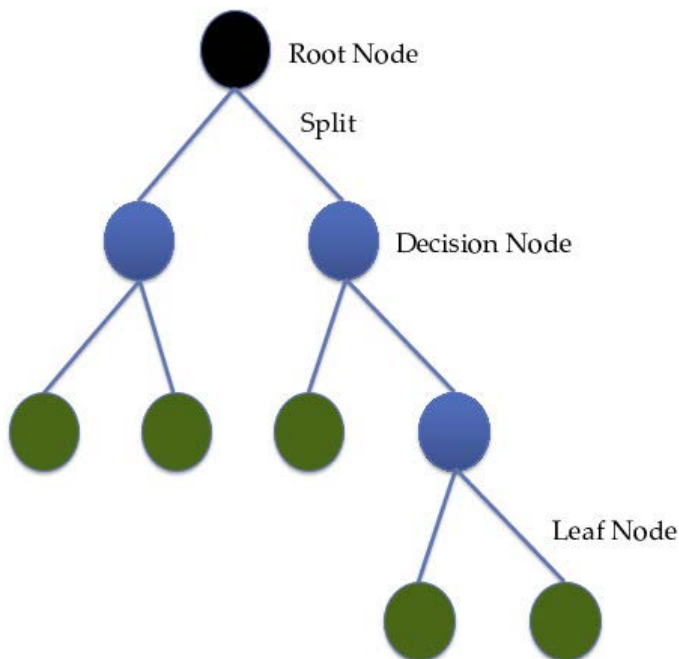


Figure 3.11: Decision tree nodes

The decision trees belong to the family of greedy algorithms. To find the most homogeneous split, we define our cost function so that it tries to maximize the same class input values in a particular group. For regression, we generally use the mse cost function:

$$\mathcal{L}_{reg} = \sum (y - y_{pred})^2$$

Here, y and y_{pred} represent the given and predicted output values for the input values (i); we find the split that minimizes this loss.

For classification, we use either the **gini impurity** or cross-entropy as the loss function:

$$\mathcal{L}_{gini} = \sum c_k(1 - c_k)$$

$$\mathcal{L}_{CE} = -\sum c_k \log(c_k)$$

In the preceding formula, c_k defines the proportion of the same class input values present in a particular group.

Note

Some of the good resources to learn more about decision trees are as follows:

- L. Breiman, J. Friedman, R. Olshen, and C. Stone. Classification and Regression Trees. Wadsworth, Belmont, CA, 1984
- J.R. Quinlan. C4. 5: programs for ML. Morgan Kaufmann, 1993
- T. Hastie, R. Tibshirani and J. Friedman. Elements of Statistical Learning, Springer, 2009

Decision trees in scikit-learn

The scikit-learn library provides `DecisionTreeRegressor` and `DecisionTreeClassifier` to implement regression and classification. Both can be imported from `sklearn.tree`. `DecisionTreeRegressor` is defined as follows:

```
class sklearn.tree.DecisionTreeRegressor (
    criterion='mse', splitter='best', max_depth=None,
    min_samples_split=2, min_samples_leaf=1,
    min_weight_fraction_leaf=0.0, max_features=None,
    random_state=None, max_leaf_nodes=None,
    min_impurity_decrease=0.0,
    min_impurity_split=None, presort=False
)
```

The different arguments are as follows:

- `criterion`: It defines which loss function to use to determine the split. The default value is the **MSE**. The library supports the use of **Friedman MSE** and **mean absolute error (MAE)** as loss functions.
- `splitter`: We use this to decide whether to use the greedy strategy and go for the best split (default) or whether we can use random splitter to choose the best random split.
- `max_depth`: It defines the maximum depth of the tree.

- `min_samples_split`: It defines the minimum number of samples required to split an internal node. It can be integer or float (in this case, it defines the percentage of minimum samples needed for the split).

`DecisionTreeClassifier` is defined as follows:

```
class sklearn.tree.DecisionTreeClassifier(
    criterion='gini', splitter='best', max_depth=None,
    min_samples_split=2, min_samples_leaf=1,
    min_weight_fraction_leaf=0.0, max_features=None,
    random_state=None, max_leaf_nodes=None,
    min_impurity_decrease=0.0, min_impurity_split=None,
    class_weight=None, presort=False
)
```

The different arguments are as follows:

- `criterion`: It tells us which loss function to use to determine the split. The default value for the classifier is the gini. The library supports the use of entropy as loss functions.
- `splitter`: We use this to decide how to choose the split (the default value is the best split), or we can use random splitter to choose the best random split.
- `max_depth`: It defines the maximum depth of the tree. When the input feature space is large, we use this to restrict the maximum depth and take care to avoid overfitting.
- `min_samples_split`: It defines the minimum number of samples required to split an internal node. It can be an integer or float (in this case, it tells us the percentage of minimum samples needed for the split).

Note

We've only listed the commonly used preceding arguments; details regarding the remaining parameters of the two can be found on the scikit-learn website at <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html#sklearn-tree-decisiontreeregressor> and <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn-tree-decisiontreeclassifier>.

Decision trees in action

We'll use a decision tree regressor to predict electrical power output first. The dataset and its description have already been introduced in *Chapter 1*. The code is available at the GitHub repository in the file named `ElectricalPowerOutputPredictionUsingDecisionTrees.ipynb`:

1. As in the previous code examples, let us start by importing the necessary modules:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
```

2. Let us read the file we downloaded for the hands-on regression exercise:

```
file_path = '/content/data/CCPP/Folds5x2_pp.xlsx'
df = pd.read_excel(file_path, sheet_name='Sheet1')
```

3. We perform the same preprocessing steps:

```
X, Y = df[['AT', 'V', 'AP', 'RH']], df['PE']
scaler = MinMaxScaler()
X_new = scaler.fit_transform(X)
target_scaler = MinMaxScaler()
Y_new = target_scaler.fit_transform(Y.values.reshape(-1,1))
X_train, X_test, Y_train, y_test = train_test_split(
    X_new, Y_new, test_size=0.4, random_state=333)
```

4. Now, we will use scikit-learn's `DecisionTreeRegressor` class to build the model. We will train it using the train split with the help of the `fit` method:

```
model = DecisionTreeRegressor(max_depth=3)
model.fit(X_train, Y_train)
```

5. Next, we make the prediction on the test data set and evaluate the performance of our decision tree regressor:

```
y_pred = model.predict(X_test)
# Calculate R-squared value
r2 = r2_score(y_test, y_pred)
print("R-squared value:", r2)

# Calculate Mean Squared Error
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

The model gives an R-square value of 0.90 and an mse of 0.0047, which is very similar to the neural network regressor we had built earlier.

Note

Experiment with the `DecisionTreeClassifier` for the wine quality classification task. Using default parameters, you will get about 70% accuracy. We can see that, for small data sizes, we can use both decision trees and naive Bayes with almost equal success. Decision trees suffer from overfitting, which can be taken care of by restricting the maximum depth or setting a minimum number of training inputs. They, like naive Bayes, are unstable—a little variation in the data can result in a completely different tree. This can be resolved by making use of bagging and boosting techniques. Last, but not least, since it's a greedy algorithm, there's no guarantee that it will return a globally optimal solution.

Having explored various ML models for regression and classification tasks, we now proceed to the next stage: integrating multiple models to enhance overall performance.

Ensemble Learning

In our daily lives, when we have to make a decision, we take guidance not from one person but from many individuals whose wisdom we trust. The same can be applied in ML; instead of depending upon one single model, we can use a group of models (**ensemble**) to make a prediction or classification decision. This form of learning is called ensemble learning.

Conventionally, ensemble learning is used as the last step in many ML projects. It works best when the models are as independent of one another as possible. The following diagram gives a graphical representation of ensemble learning:

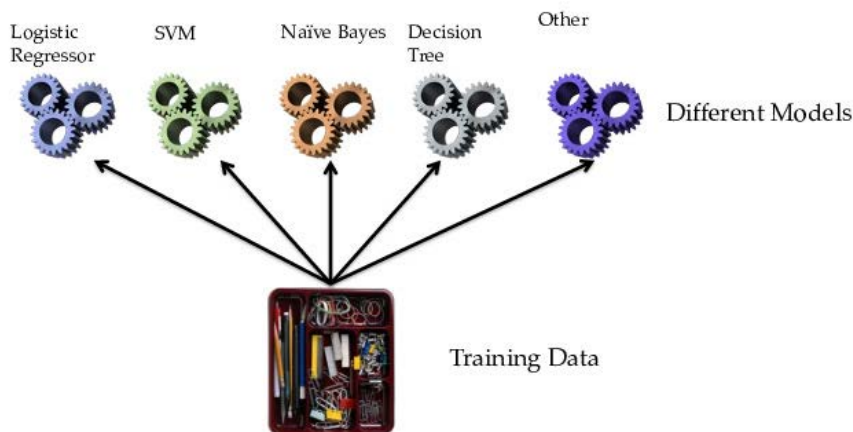


Figure 3.12: Ensemble learning

The training of different models can take place either sequentially or in parallel. There are various ways to implement ensemble learning: voting, bagging and pasting, as well as random forest. Let's learn about each of these techniques and learn how we can implement them.

Voting classifier

The voting classifier follows the majority; it aggregates the prediction of all the classifiers and chooses the class with the maximum number of votes. For example, in the following screenshot, the voting classifier will predict the input instance to belong to **class 1**:

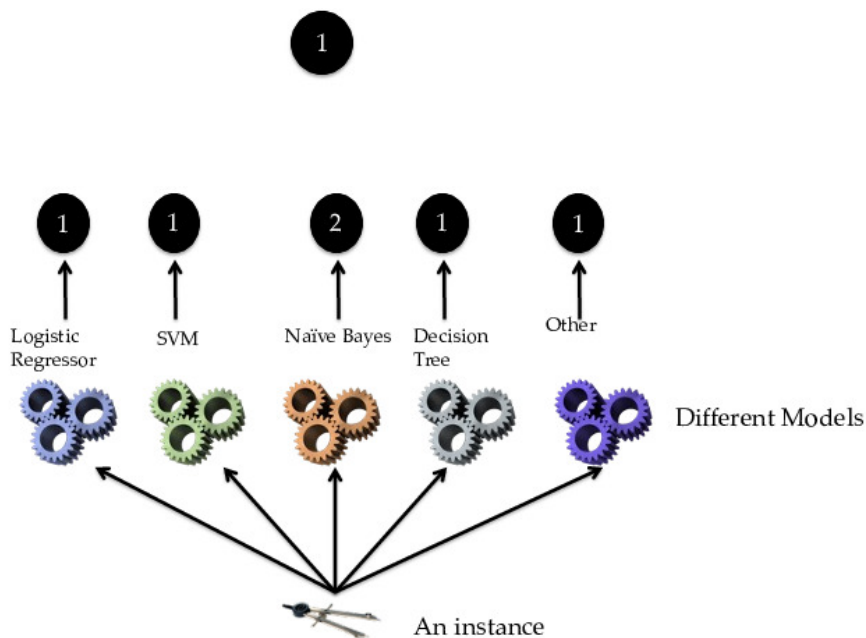


Figure 3.13: Voting classifier

scikit-learn has the `VotingClassifier` class to implement this. Using ensemble learning on wine quality classification, we can reach an accuracy score of 74%, higher than any of the models considered alone. The complete code is in the `Wine_quality_using_Ensemble_learning.ipynb` file. Here, for brevity, I will only mention the changes in the code:

1. The import will now include all the classifiers we will using, including `VotingClassifier`. For this code, I am using `DecisionTreeClassifier`, `GaussianNB`, and `SVC`:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler, LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
```

```
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import VotingClassifier

import seaborn as sns
```

2. As before, you will read the data, preprocess it, and split it. We need to define the three base classifiers and the voting classifier. We will train a voting classifier that will internally manage the operations of the three base classifiers:

```
clf1 = SVC(random_state=22)
clf2 = DecisionTreeClassifier(random_state=23)
clf3 = GaussianNB()
ecf = VotingClassifier(
    estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)],
    voting='hard'
)
ecf = ecf.fit(X_train, y_train)
```

3. Next, we evaluate it on the test dataset using similar code. Our VotingClassifier, the ensemble of GaussianNB, Decision Tree, and SVC gives about 76.56% accuracy, better than each individual model.

Bagging and pasting

In the voting approach, we apply various algorithms to the same dataset for training. Alternatively, ensemble learning can be achieved by training different instances of the same algorithm on separate subsets of the training data. The training subset is sampled randomly. The sampling can be done with replacement (bagging) or without replacement (pasting):

- **Bagging:** Here, additional data for training is generated from the original dataset using combinations with repetitions. This helps in decreasing the variance of different models.
- **Pasting:** Since pasting is done without replacement, each subset of the training data can be used once at most. It's more suitable if the original dataset is large.

The scikit library has a method for performing bagging and pasting; from `sklearn.ensemble`, we can import `BaggingClassifier` and use it. The following code estimates five hundred decision tree classifiers, each with one thousand training samples using bagging (for pasting, keep `bootstrap=False`):

```
from sklearn.ensemble import BaggingClassifier
bag_classifier = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=1000, bootstrap=True, n_jobs=-1
)
```

```
bag_classifier.fit(X_train, y_train)
y_pred = bag_classifier.predict(X_test)
```

This results in an accuracy of about 81% for wine quality classification. The confusion matrix heatmap follows:

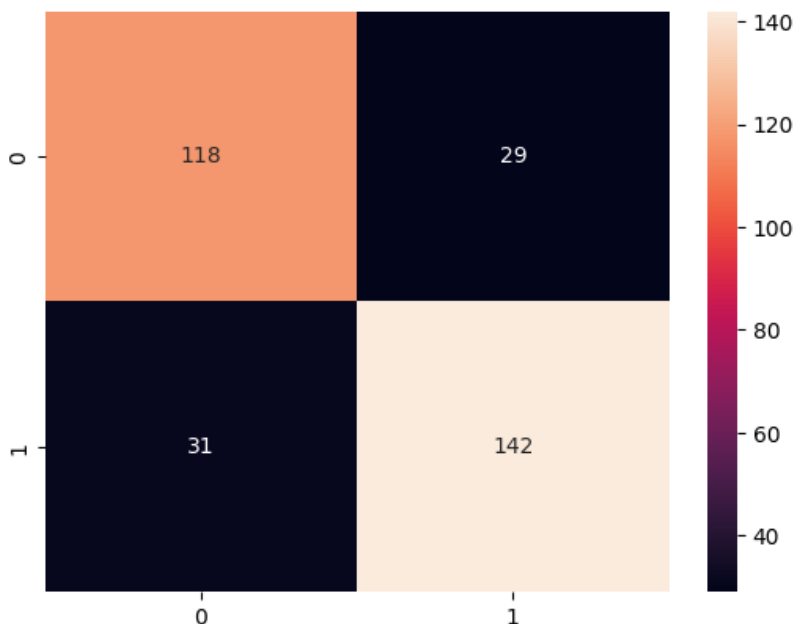


Figure 3.14: Confusion matrix using the bagging classifier with 1,000 decision tree classifiers

The 'n_jobs' parameter in the `BaggingClassifier` specifies the number of CPU cores to employ for parallel processing. Setting this parameter to -1 instructs the classifier to utilize all available CPU cores for enhanced computation efficiency.

Note

An ensemble of only decision trees is called a random forest. Thus, what we've implemented previously is a random forest. We can directly implement a random forest in scikit using the `RandomForestClassifier` class. The advantage of using the class is that it introduces extra randomness while building the tree. While splitting, it searches for the best feature to split among a random subset of features.

Improving Your Model – Tips and Tricks

In this chapter, we've learned a large number of ML algorithms, each with its own pros and cons. In this section, we'll look into some common problems and ways to resolve them.

Feature scaling to resolve uneven data scales

The data that's collected normally doesn't have the same scale; for example, one feature may be in the range of 10–100 and another one may be only distributed in the two to five range. This uneven data scale can have an adverse effect on learning. To resolve this, we use the method of feature scaling (normalization). The choice of normalization has been found to drastically affect the performance of certain algorithms. Two common normalization methods (also called **standardization** in some books) are as follows:

- **Z-score normalization:** In z-score normalization, each individual feature is scaled so that it has the properties of a standard normal distribution, that is, a mean of 0 and variance of 1. If μ is the mean and σ the variance, we can compute z-score normalization by making the following linear transformation on each feature:

$$x_{new} = \frac{x_{old} - \mu}{\sigma}$$

- **Min-max normalization:** Min-max normalization rescales the input features so that they lie in the range between 0 and 1. It results in reducing the standard deviation in the data and hence suppresses the effect of outliers. To achieve min-max normalization, we find the maximum and minimum value of the feature (x_{max} and x_{min} , respectively) and perform the following linear transformation:

$$x_{new} = \frac{x_{old} - x_{min}}{x_{max} - x_{min}}$$

We can use the `StandardScaler` or `MinMaxScaler` scikit library methods to normalize the data. In all of the examples in this chapter, we've used `MinMaxScaler`; you can try changing it to `StandardScaler` and observing whether the performance changes. In the next chapter, we'll also learn how to perform these normalizations in **Keras 3**.

Overfitting

Sometimes, the model tries to overfit the training dataset; in doing so, it loses its ability to generalize and hence performs badly on the validation dataset. This, in turn, will affect its performance on unseen data values. There are two standard ways to take care of overfitting: regularization and cross-validation.

Regularization

Regularization adds a term in the loss function to ensure that the cost increases as the model increases the number of features. Hence, we force the model to stay simpler. If $\mathcal{L}(X, Y)$ was the loss function earlier, we can replace it with the following:

$$\mathcal{L}_{new} = \mathcal{L}_{old}(X, Y) + \lambda N(W)$$

In the preceding formula, N can be $L1$ norm, $L2$ norm, or a combination of the two, and λ is the regularization coefficient. Regularization helps in reducing the model variance, without losing any important properties of the data distribution:

- **Lasso regularization:** In this case, the N is $L1$ norm. It uses the modulus of weight as the penalty term N :

$$N(W) = \sum_{j=1}^p |W_j|$$

- **Ridge regularization:** In this case, the N is $L2$ norm, given by the following:

$$N(W) = \sum_{j=1}^p W_j^2$$

Cross-validation

Using cross-validation can also help in reducing the problem of overfitting. In k -fold cross-validation, data is divided into k -subsets, called folds. Then it trains and evaluates the model k -times; each time, it picks one of the folds for validation and the rest for training the model. We can perform the cross-validation when there is less data, and when the training time is small. Scikit provides a `cross_val_score` method to implement the k -folds. To apply this method, we first specify the model (in this case, `classifier`) that we wish to evaluate. The following code snippet carries out cross-validation using 10 folds on our chosen model. This process involves dividing the training data (`X_train` and `y_train`) into **ten subsets** (or *folds*), then iteratively training the model on **nine of these subsets** and evaluating it on the remaining subset. This cycle is repeated until each fold has been used once as the test set.

The output of `cross_val_score` is an array of accuracy scores, one for each fold, representing the model's performance on that particular subset of data. By calculating the mean and standard deviation of these scores, we obtain an average accuracy and a measure of variance, respectively:

```
from sklearn.model_selection import cross_val_score
accuracies = cross_val_score(
    estimator = classifier, X = X_train, y = y_train, cv = 10)
print(f"Accuracy Mean {accuracies.mean()} Accuracy Variance
{accuracies.std()}")
```

The result of this is an average mean and variance value. Ideally, a robust model should exhibit high average accuracy (suggesting strong predictive performance) and low variance (indicating consistency in its performance across different subsets of the training data).

No Free Lunch theorem

With so many models, one always wonders which one to use. Wolpert, in his famous paper *The Lack of A Priori Distinctions Between Learning* (<https://direct.mit.edu/neco/article-abstract/8/7/1341/6016/The-Lack-of-A-Priori-Distinctions-Between-Learning>), explored this issue and showed that if we make no prior assumption about the input data, then there's no reason to prefer one model over any other. This is known as the **No Free Lunch theorem**.

This means that no single model can be pre-emptively deemed superior for all problems. This implies that the efficacy of a model can only be determined through empirical testing. In practical scenarios, however, it's infeasible to assess every available model due to sheer volume and complexity. Therefore, we typically rely on informed assumptions about the data to narrow down our choices and focus on evaluating a select few models that are most pertinent to our specific problem.

Hyperparameter tuning and grid search

Different models have different **hyperparameters**. For example, in linear regressor, the learning rate was a hyperparameter; if we're using regularization, then the regularizing parameter λ is a hyperparameter. What should be their value? While there's a rule of thumb for some hyperparameters, most of the time we either make a guess or use grid search to perform a sequential search for the best hyperparameters. In the following code snippet, we present the code to perform a hyperparameter search in the case of SVM using the scikit library. In the coming chapters, we'll see how we can use Keras 3 to perform hyperparameter tuning:

```
from sklearn.model_selection import GridSearchCV
classifier = SVC()
parameters = [
    {'C': [1, 10], 'kernel': ['linear']},
    {
        'C': [1, 10], 'kernel': ['rbf'],
        'gamma': [0.1, 0.2, 0.3, 0.4, 0.5,
                  0.6, 0.7, 0.8, 0.9]
    }
]
grid_search = GridSearchCV(
    estimator = classifier,
    param_grid = parameters,
    scoring = 'accuracy', cv = 10,
)
grid_search.fit(X_train, y_train)
best_accuracy = grid_search.best_score_
best_parameters = grid_search.best_params_
best_accuracy
```

GridSearchCV will provide us with the hyperparameters that produce the best results for the SVM classifier.

Summary

The goal of this chapter was to provide you with an intuitive understanding of different standard ML algorithms so that you can make an informed choice. We covered the popular ML algorithms used for classification and regression. We also learned how supervised and unsupervised learning are different from each other. Linear regression, logistic regression, SVM, naive Bayes, and decision trees were introduced, along with the fundamental principles involved in each. We used regression methods to predict the electrical power production of a thermal station and classification methods to classify wine as good or bad. Lastly, we covered the common problems with different ML algorithms and explored some tips and tricks to solve them.

In the upcoming chapter, we will delve into deep learning algorithms and witness their capabilities first hand.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/ds>



Part 2: Advanced AI Techniques and Their Application in IoT

In this part, we go deeper into advanced AI methods, including deep learning, optimization, reinforcement learning, and generative models. Each technique is applied to real-world IoT scenarios, giving you practical insights into how cutting-edge algorithms can solve complex challenges in intelligent IoT systems.

This part has the following chapters:

- *Chapter 4, Deep Learning for IoT*
- *Chapter 5, Optimization Techniques in IoT*
- *Chapter 6, Reinforcement Learning for IoT*
- *Chapter 7, Generative Adversarial Networks for IoT*

Deep Learning for IoT

In the last chapter, we learned about different **machine learning (ML)** algorithms. The focus of this chapter is on **neural networks** based on **multiple-layered models**, also known as **deep learning (DL)** models. These models have become a buzzword in recent years, attracting significant attention and investment in the field of artificial intelligence-based start-ups. One of the remarkable achievements of DL includes attaining above-human-level accuracy in tasks such as object detection and games such as Chess and Go. The recent triumphs of **Large Language Models (LLMs)**, based on the transformer architecture – a deep neural network – also underscore the power and versatility of DL.

In this chapter, we will delve into different DL models and explore their applications, particularly in processing data generated by the **Internet of Things (IoT)**. We'll start with an overview of the DL journey and then focus on four popular models – the **multilayer perceptron (MLP)**, the **convolutional neural network (CNN)**, the **recurrent neural network (RNN)**, and **autoencoders**. Additionally, we will introduce **TinyML** and **OpenVino**, providing insights into their role and integration in the DL ecosystem. This comprehensive exploration will equip you with a solid understanding of how to leverage DL in various real-world scenarios. Specifically, you will learn about the following:

- DL 101
- DL – Why Now?
- Exploring Artificial Neurons
- Learning in Artificial Neural Networks
- Multilayered Perceptrons for Regression and Classification
- Understanding CNNs
- Recurrent Neural Networks
- Exploring Autoencoders
- Introduction to OpenVINO and TinyML
- Using Keras 3 for Hyperparameter Tuning

DL 101

The human mind has always intrigued philosophers, scientists, and engineers alike. The desire to imitate and replicate the intelligence of the human brain by man has been written about over many years; Galatea by Pygmalion of Cyprus in Greek mythology, Golem in Jewish folklore, and Maya Sita in Hindu mythology are just a few examples. Robots with AI have been a favorite of (science) fiction writers since time immemorial.

AI, as we know it today, was conceived in parallel with the idea of computers. The seminal 1943 paper *A Logical Calculus of the Ideas Immanent in Nervous Activity* by McCulloch and Pitts proposed the first neural network model – the threshold devices that could perform logical operations such as AND, OR, and AND-NOT. In his pioneering 1950 work *Computing Machinery and Intelligence*, Alan Turing proposed a **Turing test** – a test to identify whether a machine has intelligence or not. Rosenblatt, in 1957, laid the foundation for networks that could learn from experience in his report *The Perceptron – a perceiving and recognizing automaton*. These ideas were far ahead of their time; while the concepts looked theoretically possible, computational resources at that time severely limited the performance you could get from these models that could do logic and learn.

Note

While these papers may seem old and irrelevant, they are very much worth reading and give great insight into the vision these initial thinkers had. The following are links to these papers for those of you who are interested:

A Logical Calculus of the Ideas Immanent in Nervous Activity by McCulloch and Pitts: <https://link.springer.com/article/10.1007%2FBF02478259>

Computing Machinery and Intelligence by Alan Turing: <http://phil415.pbworks.com/f/TuringComputing.pdf>

The Perceptron – a perceiving and recognizing automaton by Rosenblatt: <https://websites.umass.edu/brain-wars/1957-the-birth-of-cognitive-science/the-perceptron-a-perceiving-and-recognizing-automaton/>

Another interesting paper is one by Wang and Raj from Carnegie Melon University, *On the Origin of Deep Learning*; the 72-page paper covers in detail the history of DL, starting from the McCulloch–Pitts model to the latest attention models: <https://arxiv.org/pdf/1702.07800.pdf>

Two AI winters and a few successes later (with the breakthrough in 2012, when Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton's AlexNet entry in the annual ImageNet challenge achieved an error rate of 16%), we stand today at a place where DL has outperformed most of the existing AI techniques. The following screenshot from Google Trends shows that, roughly around 2014, DL became popular and has been growing since then, with another bump in 2022 with the release of ChatGPT by OpenAI:

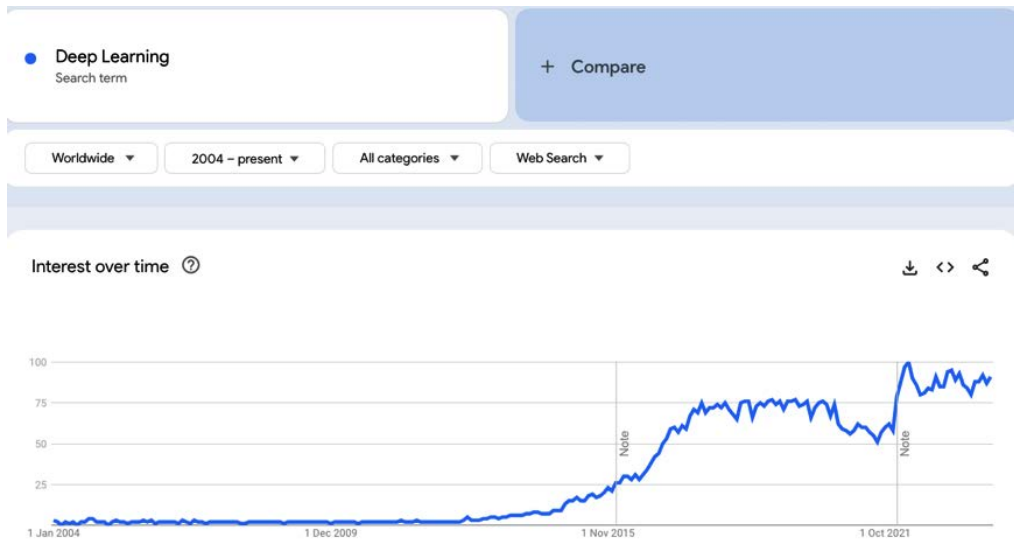


Figure 4.1: DL in Google Trends from 2004 to January 2024

Let's see the reasons behind this growing trend and analyze whether it's just hype or whether there's more to it.

DL – Why Now?

Most of the core concepts in the field of DL were already in place by the 1980s and 1990s; therefore, the question arises as to why we suddenly see an increase in the applications of DL to solve different problems, from image classification and image inpainting to self-driving cars and speech generation. The major reason is twofold, outlined as follows:

- **Availability of large high-quality datasets:** The internet resulted in the generation of an enormous amount of datasets in terms of images, video, text, and audio. While most of it's unlabeled, by the effort of many leading researchers (for example, Fei Fei Li creating the ImageNet dataset), we finally have access to large, labeled datasets. If DL is a furnace lighting your imagination, data is the fuel burning it. The greater the amount and variety of data, the better the performance of a model.
- **Availability of parallel computing using graphical processing units:** In DL models, there are mainly two mathematical matrix operations that play a crucial role – namely, matrix multiplication and matrix addition. The possibility of parallelizing these processes for all the neurons in a layer with the help of **graphical processing units (GPUs)** made it possible to train the DL models in a reasonable time.

Once the interest in DL grew, people came up with further improvements, such as better optimizers for the gradient descent (the necessary algorithm used to calculate weight and bias update in DL models) – for example, **Adam** and **RMSprop**; new regularization techniques, such as **dropout** and **batch normalization**, that not only help in overfitting but can also reduce the training time; and, last but not the least, the availability of DL libraries such as TensorFlow, Torch, and Keras, which made it easier to define and train complex architectures.

According to Andrew Ng, founder of `deeplearning.ai`, despite plenty of hype and frantic investment, we won't see another AI winter because improvements in the computing devices “*will keep the performance advances and breakthroughs coming for the foreseeable future.*” Andrew Ng said this at EmTech Digital in 2016, and true to his prediction, we have seen advancements in the processing hardware, with Google's **Tensor Processing Units (TPUs)**, Intel Movidius, and NVIDIA's latest GPUs. Moreover, there are cloud computing GPUs that are available today at as low as 0.40 cents per hour, making it affordable for all.

Note

You can read the complete *AI Winter Isn't Coming* article, published in MIT Technology Review, at <https://www.technologyreview.com/2016/12/07/155592/ai-winter-isnt-coming/>. Here, Andrew Ng answers different queries regarding the future of AI.

For effective DL, GPU processing power is crucial due to its ability to handle parallel computations, essential for the matrix operations in DL algorithms. While cloud computing services are popular, you should also consider the long-term benefits of local GPU setups. Beyond Google Colaboratory and Kaggle, platforms such as AWS, Microsoft Azure, and IBM Cloud offer robust GPU services. It's important to weigh the free tiers' limitations against the more consistent, but costlier, paid services. These platforms generally support major DL libraries such as TensorFlow and PyTorch, making them versatile for various applications. Practical applications where GPUs shine include image processing and training complex neural networks. Lastly, keeping an eye on emerging hardware trends, such as TPUs, is vital, as they may offer more optimized solutions for DL tasks in the future.

As we conclude our discussion on the factors responsible for the emergence of DL, let's pivot to its core component – artificial neurons.

Exploring Artificial Neurons

The fundamental component of all DL models is an artificial neuron. The artificial neuron is inspired by the working of biological neurons. It consists of some inputs connected via weights (also called **synaptic connections**); the weighted sum of all the inputs goes through a processing function (called the **activation function**) and generates a non-linear output.

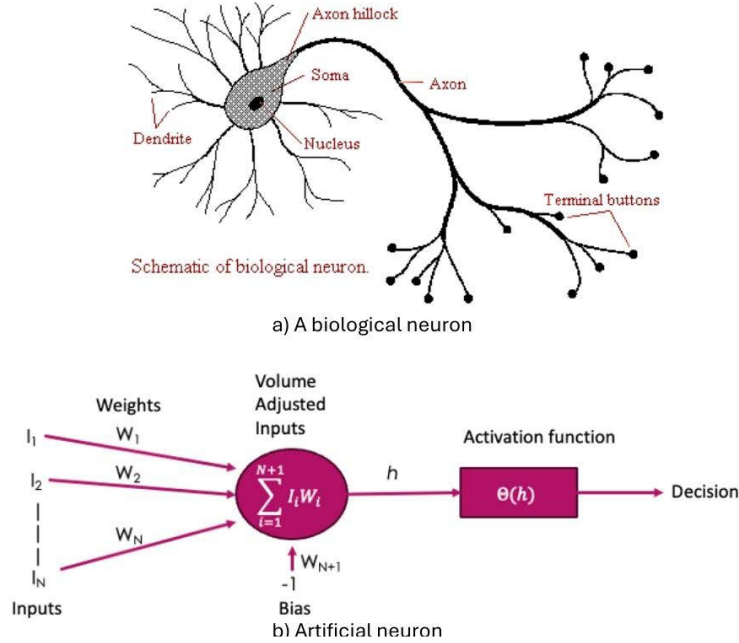


Figure 4.2: Building blocks of intelligence – biological neuron and artificial neuron. Image sources: https://commons.wikimedia.org/wiki/File:Biological_neuron.gif; <https://medium.com/tipz-ai/demystifying-neural-networks-from-neurons-to-deep-learning-f18c97f0ee9a>

If X is the i^{th} input to the artificial neuron (j) connected via the synaptic connection w_{ij} , then the net input to the neuron, commonly called the activity of the neuron, can be defined as the weighted sum of all the inputs and can be expressed mathematically as the following:

$$h_j = \sum_{i=1}^N X_i W_{ij} - \theta_j$$

In the preceding equation, N is the total number of inputs to the j^{th} neuron, and θ_j is the threshold of the j^{th} neuron; the output of the neuron is then given by the following:

$$y_j = g(h_j)$$

In the preceding equation, g is the activation function. Activation functions play a pivotal role in DL, as they determine how a network processes inputs and learns. There's a wide array of activation functions to choose from, each with distinct characteristics and applications in different DL models. Next, we will explore various activation functions, providing both their mathematical formulations and graphical representations. This will not only help in understanding their mechanisms but also in selecting the most appropriate one for a specific DL task.

Sigmoid activation function

The sigmoid activation function is mathematically expressed as follows:

$$g(h) = \frac{1}{1 + e^{-h}}$$

It maps inputs to a range between 0 and 1 and is ideal for binary classification. Its “S”-shaped curve facilitates smooth output transitions and continuous backpropagation through its non-zero derivative. However, it is less favored in modern neural networks, due to the vanishing gradient problem and its non-zero-centered output.

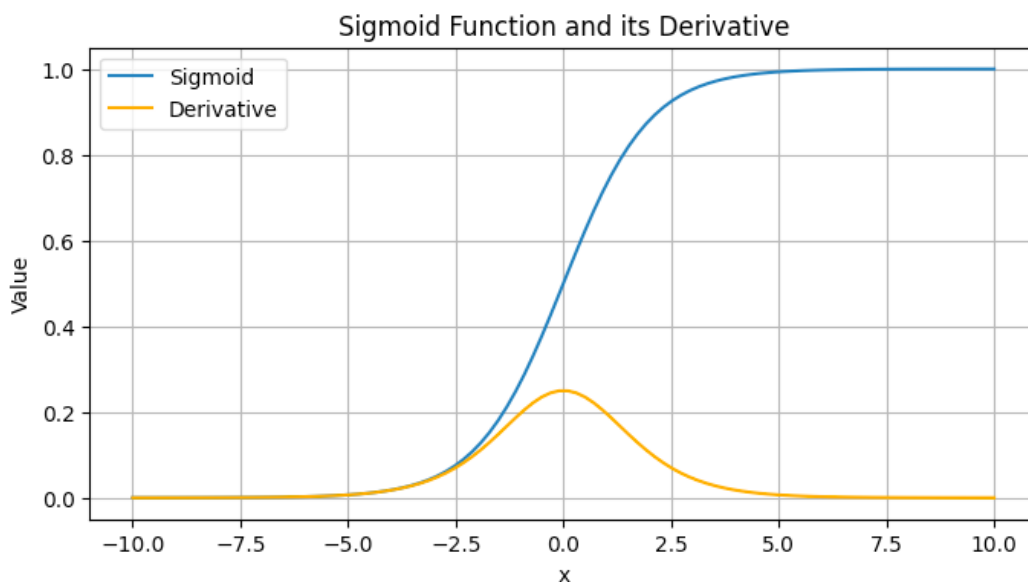


Figure 4.3: The sigmoid activation function and its derivative

Hyperbolic tangent activation function

The hyperbolic **tangent (tanh)** activation function is a widely used activation function in DL, defined mathematically as follows:

$$g(h) = \tanh(h) = \frac{e^h - e^{-h}}{e^h + e^{-h}}$$

It outputs values in the range of -1 to 1, making it zero-centered, which helps to mitigate the issue of “vanishing gradients” to some extent compared to the sigmoid function. This zero-centered nature also leads to faster convergence during training. The tanh function shares a similar shape to the sigmoid function but stretches across the y -axis, offering steeper gradients for large inputs, which can result in more decisive outputs.

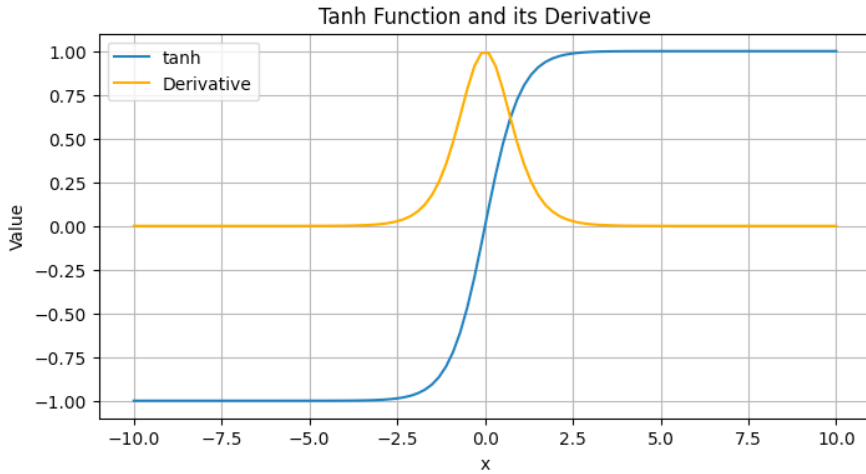


Figure 4.4: Tan-hyperbolic activation function and its derivative

Rectified linear unit activation function

The **Rectified Linear Unit (ReLU)** activation function is a cornerstone in modern neural networks, defined as $ReLU(x) = \max(0, x)$. Its simplicity and efficiency lie in passing positive inputs directly, while negative inputs are set to zero. This design leads to faster computational times and alleviates the vanishing gradient problem, as it provides a constant gradient for positive inputs. Widely used in hidden layers of deep neural networks, ReLU has been instrumental in training large and complex models efficiently. However, its limitation is the “dying ReLU” problem, where neurons can become inactive and only output zero if they fail to activate across several data points. Despite this, ReLU’s advantages in sparsity and computational efficiency make it a default choice for many DL applications.

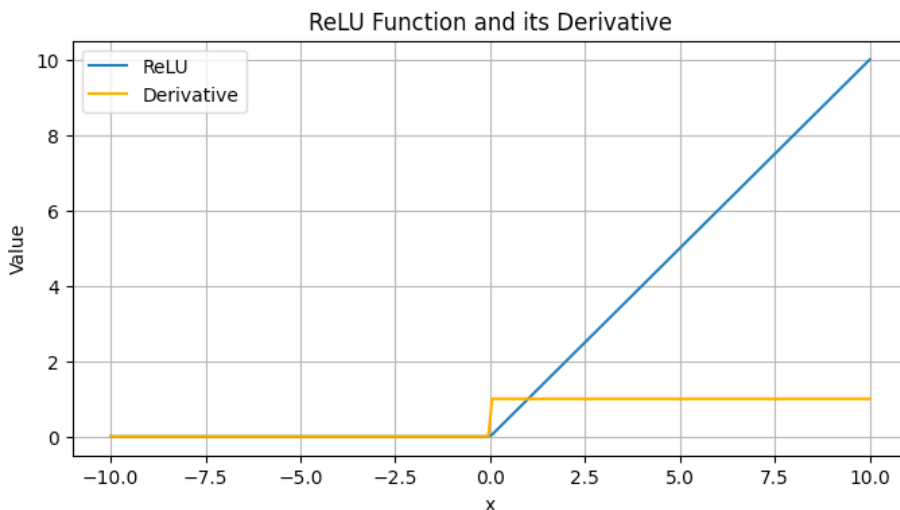


Figure 4.5: The ReLU activation function and its derivative

Softmax activation function

The softmax activation function is a vital tool in DL, particularly for multi-class classification problems. Defined mathematically as follows:

$$g(h_i) = \frac{e^{h_i}}{\sum_{i=1}^N e^{h_i}}$$

For j in the range of the input vector, it transforms a vector of values into a probability distribution. Each output of the softmax function is in the range (0,1), and the sum of all outputs is 1, making it ideal for interpreting the outputs as probabilities. Unlike other activation functions such as ReLU or sigmoid, softmax is typically applied in the output layer of a neural network model, especially when handling multiple classes. Its ability to handle multiple categories simultaneously and output a probability distribution makes it a go-to choice for tasks such as image classification, natural language processing, and any scenario where the prediction of multiple potential categories is essential.

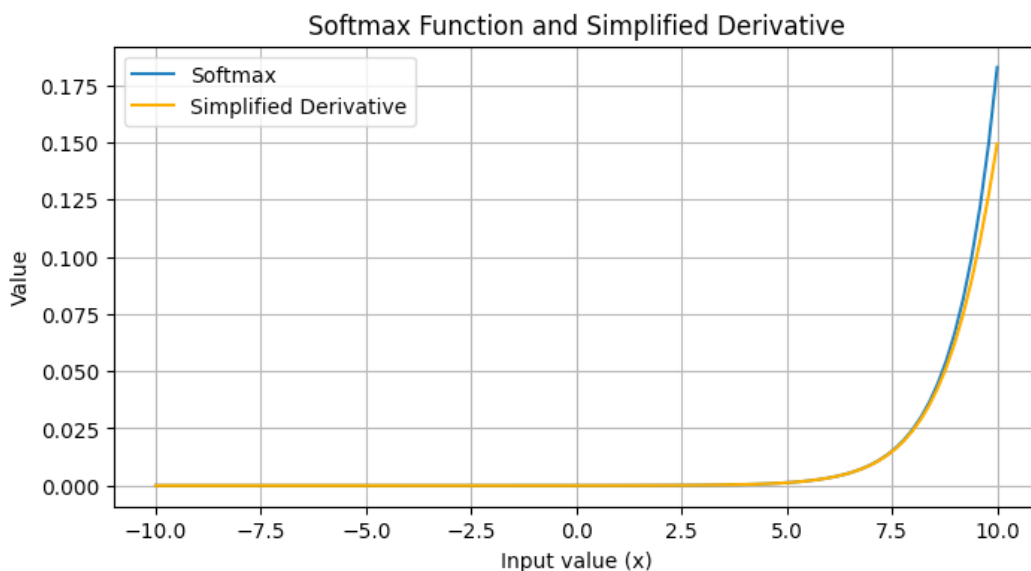


Figure 4.6: The softmax activation function and its simplified derivative (for one variable)

Leaky ReLU activation function

The **Leaky Rectified Linear Unit (Leaky ReLU)** activation function is an advanced variant of the classic ReLU activation function, designed to address its primary limitation, the “dying ReLU” problem. It is defined as:

$$g(h) = \begin{cases} h & \text{for } h \geq 0 \\ \alpha h & \text{for } h < 0 \end{cases}$$

where α is a small, non-zero slope (typically around 0.01); it allows a small, non-zero gradient when the unit is not active. This slight gradient ensures that neurons never die out completely, as they still propagate a small amount of the gradient during backpropagation even when they're inactive. Leaky ReLU maintains most of the benefits of ReLU, such as computational efficiency and sparsity, while mitigating the risk of dead neurons. It's particularly useful in deep networks where the dying ReLU problem is more pronounced. The choice of α is crucial – if it's too small, it approximates ReLU; if it's too large, it may lead to undesirable model behaviors.

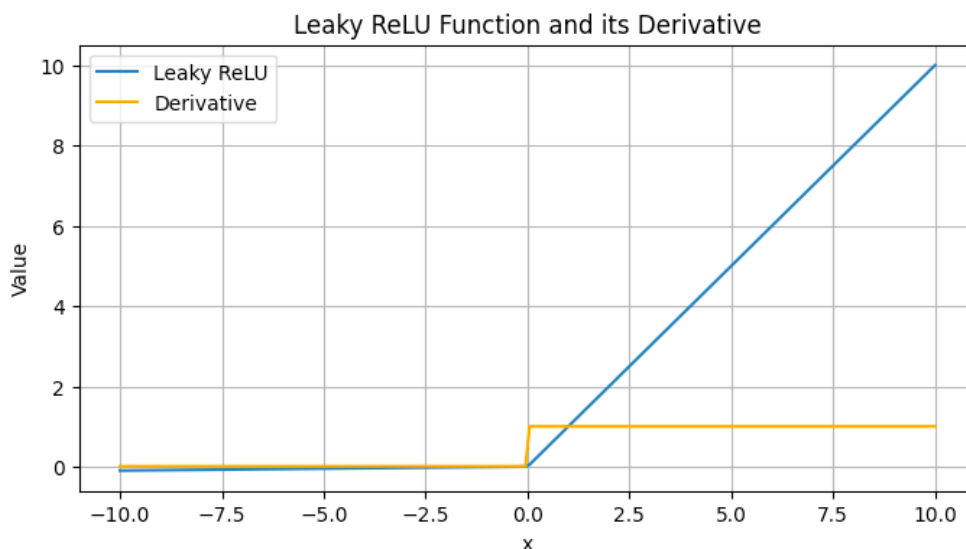


Figure 4.7: Leaky ReLU and its derivative

Exponential Linear Unit activation function

The **Exponential Linear Unit (ELU)** activation function is a variant of the ReLU activation function, designed to enhance the performance of deep neural networks. It is defined as:

$$g(h) = \begin{cases} h & \text{for } h > 0 \\ \alpha(e^h - 1) & \text{for } h \leq 0 \end{cases}$$

where α is a hyperparameter; ELU combines the benefits of ReLU and its ability to create non-linear decision boundaries. For positive inputs, ELU behaves like the ReLU function. For negative inputs, it provides a smooth, exponential curve that approaches $-\alpha$, ensuring a smoother gradient transition than ReLU. This characteristic helps to reduce the vanishing gradient problem, leading to faster learning and improved performance of deep neural networks, especially in scenarios where the input distribution is not centered around zero. The ELU activation function is particularly useful when the model needs to learn representations from negative input values without completely zeroing them out, as in the case of ReLU.

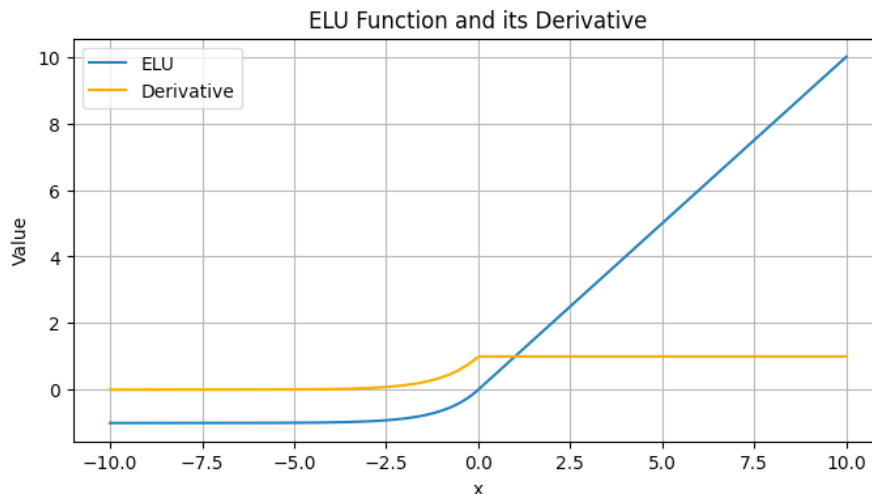


Figure 4.8: The ELU activation function and its derivative

In this section, we learned about the basic structure of an artificial neuron and various options for the activation function. Next, let's build a single neuron.

The essence of an artificial neuron

As we delve into the world of DL and artificial neurons, it's fascinating to observe the parallels between these advanced concepts and more traditional statistical models, such as linear and logistic regression. At its core, a single artificial neuron can be seen as a close cousin to these regression models.

When an artificial neuron operates without an activation function, or with a linear activation function, it performs a computation akin to linear regression. The neuron takes multiple inputs, applies weights (similar to regression coefficients), and produces an output that is a linear combination of those inputs. This mirrors the process of predicting a continuous dependent variable from independent variables in linear regression.

Conversely, when we introduce a non-linear activation function such as the sigmoid, our humble artificial neuron starts performing a task similar to logistic regression. It still computes a weighted sum of inputs but then transforms this sum into a probability-like output, ranging between 0 and 1. This is essentially what happens in logistic regression when we estimate the probability of a binary outcome.

So, whether you're looking at a neuron in a neural network or a regression model in statistics, you're observing different facets of the same fundamental principle – using weighted inputs to predict an outcome. It's this elegant simplicity that forms the stepping stone to the complex architectures that drive today's DL miracles.

Learning in Artificial Neural Networks

Can we use this single neuron and make it learn? The answer is yes; the process of learning involves adapting the weights such that a predefined **loss function** (L) reduces. If we update the weights in the direction opposite to the gradient of the loss function with respect to weights, it will ensure that the loss function decreases with each update. This algorithm is called the **gradient descent** algorithm and is at the heart of all DL models. Mathematically, if L is the loss function and η the learning rate, then the weight w_{ij} is updated and represented as follows:

$$w_{ij}^{new} = w_{ij}^{old} - \eta \frac{\partial L}{\partial w_{ij}}$$

To make our artificial neuron learn, we need first to decide on the following parameters:

- **Learning rate parameter:** The learning rate parameter determines how fast we descend the gradient. Conventionally, it lies between 0 and 1. If the learning rate is too high, the network may either oscillate around the correct solution or completely diverge from the solution. Conversely, when the learning rate is too low, it will take a long time to converge to the solution finally.
- **Activation function:** The activation function decides how the output of the neuron varies with its activity. Since the weight update equation involves a derivative of the loss function, which in turn will depend on the derivative of the activation function, we prefer a continuous-differentiable function as the activation function for the neuron. Initially, sigmoid and hyperbolic tangent were used, but they suffered from slow convergence and vanishing gradients (the gradient becoming zero, and hence, no learning, while the solution hasn't been reached). In recent years, ReLU and its variants such as leaky ReLU and ELU have been preferred, since they offer fast convergence and, at the same time, help in overcoming the vanishing gradient problem. In ReLU, we sometimes have a problem with dead neurons – that is, some neurons never fire because their activity is always less than zero, and hence, they never learn. Both leaky ReLU and ELU overcome the problem of dead neurons by ensuring a non-zero neuron output, even when the activity is negative. The lists of the commonly used activation functions, and their mathematical and graphical representations, were explained before this section. (You can play around with the `activation_functions.ipynb` code.)
- **Loss function:** The loss function is the parameter that our network tries to minimize, so choosing the right loss function is crucial for learning. As you will delve deep into DL, you will find many cleverly defined loss functions. You will see how, by properly defining loss functions, we can make our DL model create new images, visualize dreams, give a caption to an image, and much more. Conventionally, depending on the type of task regression or classification, people use a **mean square error** (MSE) or **categorical-cross entropy** loss function. We covered both in *Chapter 3*, and there are many other options in Keras; you can check the available loss functions in the Keras documentation: <https://keras.io/api/losses/>.

Note that if you were to implement this artificial neuron without using Keras (to be specific, without using any of the DL libraries mentioned earlier), then you will need to calculate the gradient yourself – for example, you will write a function or code that will first compute the gradient of the loss function, and then you will have to write code to update all of the weights and biases.

For a single neuron with the MSE loss function, calculating derivative is still straightforward, but as the complexity of the network increases, calculating the gradient for the specific loss function, implementing it in code, and then, finally, updating weights and biases can become a very cumbersome act.

Now, let's fast-forward to Keras 3, which has further refined the art of **automatic differentiation**, making it even more effortless for developers to train neural networks. Automatic differentiation remains a cornerstone of Keras, allowing you to build execution graphs and define loss functions while it handles the intricate calculations of gradients.

Keras 3 introduces updates that streamline this process further. It is designed to work seamlessly with TensorFlow, JAX, and PyTorch backends, ensuring that you can integrate Keras models into training loops of these frameworks without friction. This flexibility means you can leverage the strengths of each backend while enjoying the user-friendly Keras interface.

For example, you can use TensorFlow's `tf.GradientTape` or JAX's `jax.grad` for **automatic differentiation** with Keras models. Additionally, Keras 3 supports a range of optimizers, including the well-known SGD and Adam, and introduces learning rate schedules such as `ExponentialDecay` to modulate learning rates over time, making your model's training phase more efficient.

With Keras 3, the process of updating weights using gradients calculated through automatic differentiation is not just simpler but also more powerful. You can now enjoy the same ease of defining and training models with `fit()`, while also having the option to drop down to lower-level, backend-specific operations if needed.

After delving into the intricacies of learning in neural networks, we'll next explore the practical application of **Multilayer Perceptrons (MLPs)** in both regression and classification tasks. We will also take a deeper look at the backpropagation algorithm.

Multilayered Perceptrons for Regression and Classification

In our journey through the neural network landscape, we've seen how a single artificial neuron functions – taking in inputs, applying weights, and giving us an output. This is great for simple problems with clear, linear boundaries. But what about the messier, more complex problems where a straight line just doesn't cut it?

Note

The network with a single layer of neurons is called a simple perceptron. The perceptron model was given by Rosenblatt in 1958 (<https://homepages.math.uic.edu/~lreyzin/papers/rosenblatt58.pdf>). The paper created lots of ripples in the scientific community, and lots of research was initiated in the field. It was first implemented in hardware for the task of image recognition. Although perceptron seemed very promising initially, the book *Perceptrons* by Marvin Minsky and Seymour Papert proved that simple perceptron can solve only linearly separable problems (<https://dramitakapoor.com/the-perceptron-a-probabilistic-model-for-information-storage-and-organization-in-the-brain-f-rosenblatt/>).

We can use multiple layers of single neurons – in other words, use MLPs. Just as in real life, we solve a complex problem by breaking it into small problems; each neuron in the first layer of the MLP breaks the problem into small linearly separable ones. Since the information flows here in one direction from the input layer to the output layer via hidden layers, this network is also called a feedforward network. In the following diagram, we can see how the XOR problem is solved, using two neurons in the first layer and a single neuron in the output layer. The network breaks the non-linearly separable problem into three linearly separable problems.

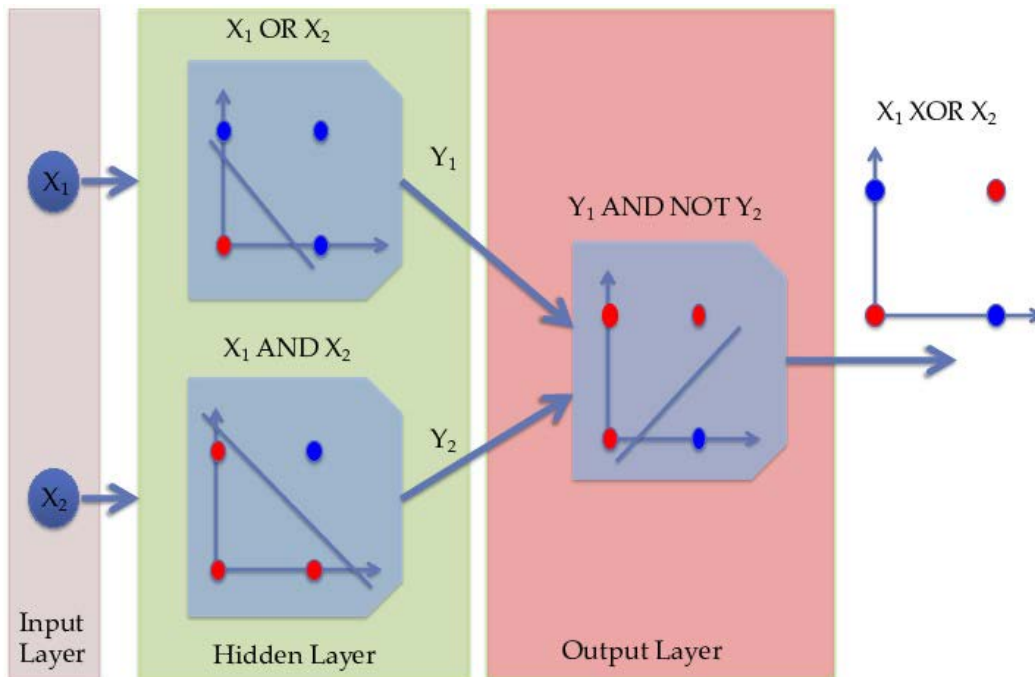


Figure 4.9: Breaking non-linear XOR into three linearly solvable problems

The previous diagram illustrates XOR being solved using MLP, with one hidden layer with neurons and one neuron in the output layer. Red points represent zero and blue points represent one. We can see that the hidden neurons separate the problem into two linearly separable problems (AND and OR); the output neuron then implements another linearly separable logic, the AND-NOT logic. By combining them, we are able to solve the XOR, which is not linearly separable.

The hidden neurons transform the problem into a form that the output layer can use. The idea of multiple layers of neurons was initially given by McCulloch and Pitts, but while Rosenblatt provided the learning algorithm for simple perceptrons, he had no way of training MLPs. The main challenge with MLPs was determining how to adjust the hidden neurons' weights. While we could easily measure the error at the output neurons against the expected results, and thus update their weights using gradient descent, the hidden neurons presented a problem. Unlike output neurons, their desired outputs aren't directly observable, making it unclear how to calculate their errors and adjust their weights accordingly. Hence, in the absence of any learning algorithm, MLPs were never explored much. This changed in 1982 when Hinton proposed the backpropagation algorithm; here is the link to his Nature paper that introduced the backpropagation algorithm: <https://www.nature.com/articles/323533a0>. The backpropagation algorithm can be used to calculate the error and, hence, the weight updates for the hidden neurons. They employed a neat and straightforward mathematical trick of differentiation using the chain rule, solving the problem of passing the errors at the output layer back to the hidden neurons and, in turn, breathing life back into neural networks. Today, the backpropagation algorithm is at the heart of almost all DL models.

Here's a step-by-step breakdown of the backpropagation algorithm:

1. **Forward pass:** Input data is passed through the network, layer by layer, until a prediction is made.
2. **Compute loss:** Calculate the loss by comparing the prediction to the true output.
3. **Backward pass:** Start from the output layer and move back through the network. For each neuron, do the following:
 - I. Calculate the gradient of the loss function with respect to each weight.
 - II. Update the weights in the opposite direction of the gradient to reduce the loss.
4. **Repeat:** Do this for many iterations or until the network's predictions are satisfactory.

The following is the pseudocode for the backpropagation algorithm:

```
initialize network weights (often small random values)
for each example in our training dataset:
    perform a forward pass through the network
    calculate the loss (how far off our predictions are)
    perform a backward pass through the network (backpropagation)
        calculate gradients (using the chain rule)
```

```
update weights (to hopefully reduce the loss)
check if the overall loss is low enough to stop
if not, repeat the process with the next batch of examples
```

For those of you interested in the mathematics behind how exactly the gradients are calculated and how weights are updated, you can find a detailed explanation in the *Appendix*. There, we'll dive into the derivatives, the chain rule, and the magic that makes learning possible for neural networks.

MLP as universal function approximators

Having explored the architecture of MLPs and the transformative role of the backpropagation algorithm, we will now venture into an intriguing aspect that elevates MLPs beyond mere computational models to powerful tools of universal capability. This brings us to the **universal approximation theorem**, a theoretical foundation that underpins the remarkable versatility of MLPs.

In 1989, Hornik et al. and George Cybenko independently proved the universal approximation theorem. The theorem, in its simplest form, states that a large enough feedforward multilayered perceptron, under mild assumptions on an activation function, with a single hidden layer can approximate any Borel measurable function, with any degree of accuracy we desire.

Simply put, it means that the neural network is a universal approximator, and we can approximate any function, as follows:

- We can do so using a single hidden layer feedforward network.
- We can do so provided the network is large enough (that is, we add more hidden neurons if needed).
- Cybenko proved the theorem for the sigmoid activation function at the hidden layer and the linear activation function at the output layer. Later, Hornik et al showed that it's actually the property of MLPs and can be proved for other activation functions too.

The theorem guarantees that MLP can solve any problem but does not give any measure of how large a network should be. Also, it does not guarantee learning and convergence.

You can refer to the papers using the following links:

- Hornik et al: <https://www.sciencedirect.com/science/article/abs/pii/0893608089900208>
- Cybenko: <https://pdfs.semanticscholar.org/05ce/b32839c26c8d2cb38d5529cf7720a68c3fab.pdf>

Empirical proof of MLP's versatility – modeling non-linear functions

Having established the theoretical underpinnings of MLPs as universal approximators, it's time to put this concept into action. We will now demonstrate the practical application of this principle using a hands-on example in Python with Keras. The following code snippet illustrates how an MLP can approximate a complex, non-linear function. By generating synthetic data based on a cubic function with added noise, we'll train an MLP to fit this data.

Let's dive into the code and witness the MLP in action, as it learns to mimic the behavior of a complex function:

1. We start by importing necessary libraries – `numpy` for numerical operations, Keras for building and training the MLP, and Matplotlib for plotting:

```
import numpy as np
from keras import layers
from keras import Sequential
import matplotlib.pyplot as plt
```

2. We create a synthetic dataset by defining a complex function (a cubic function with added noise). This function represents a non-linear relationship between x and y :

```
def generate_data():
    x = np.linspace(-2, 2, 400)
    y = x**3 - x**2 + 2
    noise = np.random.normal(0, 0.1, x.shape)
    y += noise
    return x, y
```

3. Before training the model, let's visualize the original function with the noise:

```
x, y = generate_data()
plt.scatter(x, y)
plt.title("Original Function with Noise")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

This generates the following output:

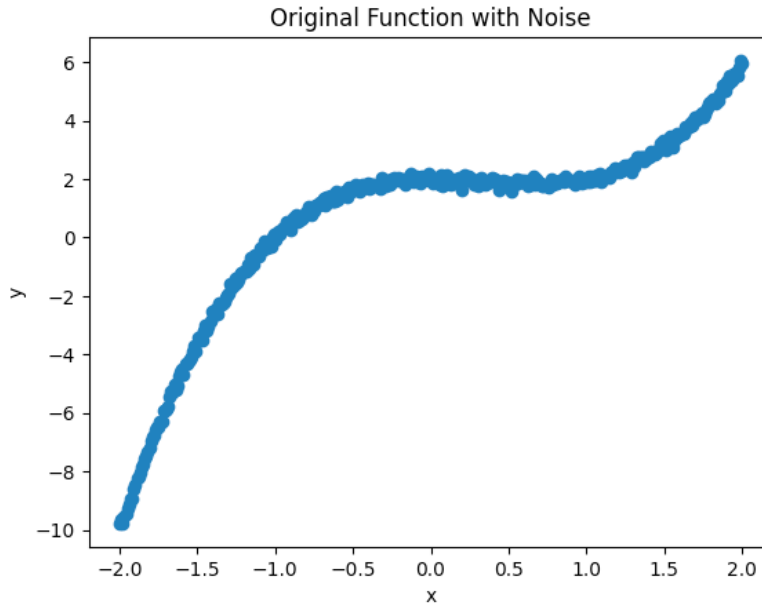


Figure 4.10: The synthetic function plot with noise added

- Next, we build a simple MLP with two hidden layers, each with 64 neurons and ReLU activation. The model is designed to predict a single output value for each input:

```
model = Sequential([
    layers.Dense(64, activation='relu', input_shape=(1,)),
    layers.Dense(64, activation='relu'),
    layers.Dense(1)
])
```

- We compile the model using the Adam optimizer and mean squared error as the loss function, suitable for regression tasks:

```
model.compile(optimizer='adam', loss='mean_squared_error')
```

- The model is trained on the synthetic data for 200 epochs:

```
model.fit(x, y, epochs=200, batch_size=32, verbose=1)
```

- After training, the model makes predictions on the same x values:

```
predictions = model.predict(x)
```

8. Finally, we plot the model's predictions against the original data to visualize how well the MLP has approximated the complex function:

```
plt.scatter(x, y, label='Original')
plt.scatter(x, predictions, label='Predicted', color='red')
plt.title("Original vs Predicted")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()
```

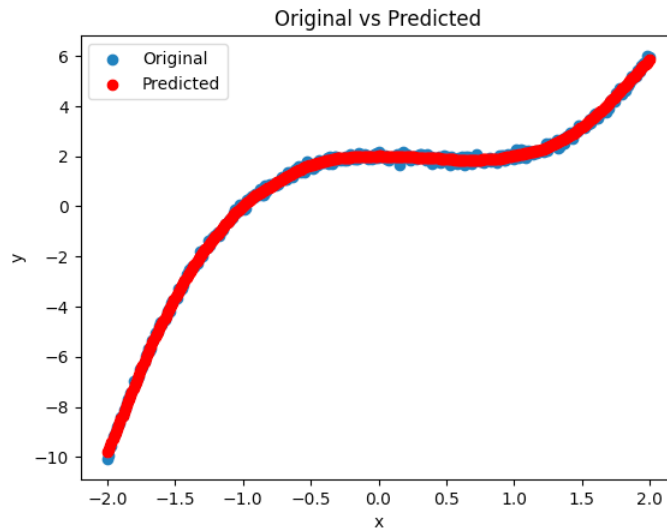


Figure 4.11: The original function and function approximated by MLP

This code demonstrates how an MLP can learn to approximate a non-linear function, showcasing its capability as a universal function approximator. In the next section, we will explore **convolutional neural networks (CNNs)**.

Understanding CNNs

MLPs were fun, but as you must have observed while playing with MLP code in the previous section that the time to learn increases as the complexity of input space increases. Moreover, the performance of MLPs is just second to the ML algorithms. Whatever you can do with MLP, there's a high probability you can do it slightly better using the ML algorithms you learned about in *Chapter 3, Machine Learning for IoT*. This inherent challenge in neural networks, including MLPs, was partly responsible for the second AI winter that occurred roughly between 1987 and 1993. During this period, the limitations of existing AI technologies, combined with unmet expectations and reduced funding, led to a significant

slowdown in AI research and development. Yet, this period also set the stage for future breakthroughs and advancements in the field.

The landscape of neural networks underwent a remarkable transformation in the 2010s with the development of **deep neural networks (DNNs)**. This era marked a pivotal shift from traditional ML techniques to more sophisticated DL models. Today, DL has achieved impressive milestones, often reaching or even surpassing human-level performance in various tasks.

In the dynamic field of computer vision, DL continues to make groundbreaking advancements. One of the notable recent developments is in the area of image synthesis and manipulation. For example, NVIDIA's research on **GANs (Generative Adversarial Networks)** has led to the creation of photorealistic images and videos, a leap forward in both creative applications and realistic rendering (for more details, check out <https://developer.nvidia.com/blog/synthesizing-high-resolution-images-with-stylegan2/>).

Another significant area is object detection and segmentation in real time, vital for autonomous vehicles and augmented reality applications. Companies such as Waymo (check out Waymo at CVPR 2023: <https://waymo.com/blog/2023/06/waymo-at-cvpr-2023-at-the-cutting-edge-of-autonomous-driving-research/>) and Tesla have made strides in leveraging DL for enhanced perception in self-driving cars, demonstrating remarkable capabilities in real-world scenarios.

Additionally, DL has revolutionized medical imaging. AI models, trained on vast datasets of X-rays, MRIs, and CT scans, now assist radiologists in detecting and diagnosing diseases with unprecedented accuracy and speed. Projects such as Google Health's research on breast cancer detection using AI showcase the potential of DL in enhancing diagnostic procedures.

In the realm of consumer technology, DL has been pivotal in improving the capabilities of smartphone cameras. Techniques such as super-resolution, advanced portrait modes, and low-light photography enhancements are direct results of DL algorithms integrated into modern smartphones.

Moreover, the integration of DL in retail and surveillance has led to sophisticated systems capable of crowd analysis, behavior prediction, and personalized advertising, transforming how businesses interact with customers and monitor environments. A key player in this success story has been CNNs. CNNs have become the backbone of most modern computer vision systems, thanks to their ability to effectively capture spatial hierarchies in images. In the upcoming section, we will delve into CNNs, exploring the mathematical foundations that make them so effective and examining some of the most influential CNN architectures that have shaped the field.

Different layers of CNN

CNN consists of three main types of layers – **convolution layers**, **pooling layers**, and **fully connected layers**. Fully connected layers are nothing but layers of MLP; they are always the last few layers of the CNN and perform the final task of classification or regression. Let's see how the convolution and max pooling layers work.

The convolution layer

This is the core building block of CNNs. It performs a mathematical operation similar to convolution (cross-correlation, to be precise) on its input, normally a 3D image. It's defined by kernels (filters). The basic idea is that these filters stride through the entire image and extract specific features from the image.

Before going into further details, let's first see the convolution operation on a two-dimensional matrix for simplicity. The following diagram shows the operation when one pixel placed at position [2, 2] of a 5×5 **2D image** matrix convolves with a 3×3 filter:

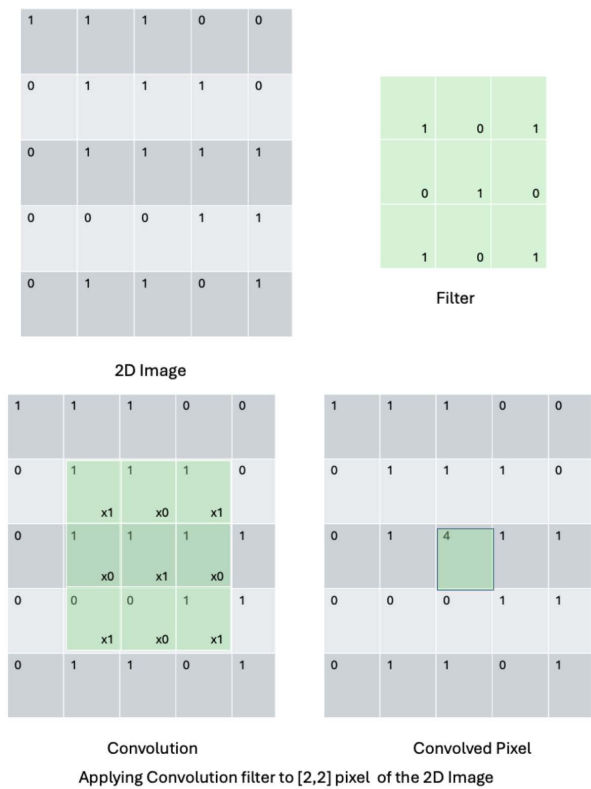


Figure 4.12: Convolution operation at a single pixel

The convolution operation involves placing the filter with the pixel at the center, and then performing element-wise multiplication between the filter elements and the pixel, along with its neighbors. Finally, we sum the product. Since convolution operation is performed on a pixel, the filters are conventionally odd-sized, such as 5×5, 3×3, or 7×7. The size of the filters specifies how much neighboring area it's covering.

The important parameters when designing the convolution layers are as follows:

- The size of the filters ($k \times k$).
- The number of filters in the layer, also called **channels**. The input color image is present in the three RGB channels. The number of channels is conventionally increased in the higher layers, resulting in deeper information in higher layers.
- The number of pixels the filter strides (s) through the image. Conventionally, the stride is one pixel so that the filter covers the entire image, starting from the top left to the bottom right.
- The padding to be used while convolving. Traditionally, there are two options, either valid or same padding. In **valid** padding, there's no padding at all, and thus the size of the convolved image is less than that of the original. In **same** padding, the padding of zeros is done around the boundary pixels, so the size of the convolved image is the same as that of the original image. The following screenshot shows the complete **convolved image**. The green square of size 3×3 is the result when the padding is valid, and the complete 5×5 matrix on the right is the result when the padding is the same:



Figure 4.13: Convolution operation applied on a 5×5 image

The green square on the right will be the result of valid padding. For the same padding, we will get the complete 5×5 matrix, shown on the right-hand side.

Pooling layer

The convolution layer is followed conventionally by a pooling layer. The purpose of the pooling layer is to progressively reduce the size of the representation, as well as the number of parameters and computations in a network. Thus, it down-samples the information as it propagates through the network in a feed-forward manner.

Here, again, we have a filter; traditionally, people prefer a filter of size 2×2, which moves with a stride of two pixels in both directions. The pooling process replaces the four elements under the 2×2 filter by either the maximum value of the four (**max pooling**) or the average value of the four (**average pooling**). In the following diagram, you can see the result of the pooling operation on a **2D single-channel slice of an image**:

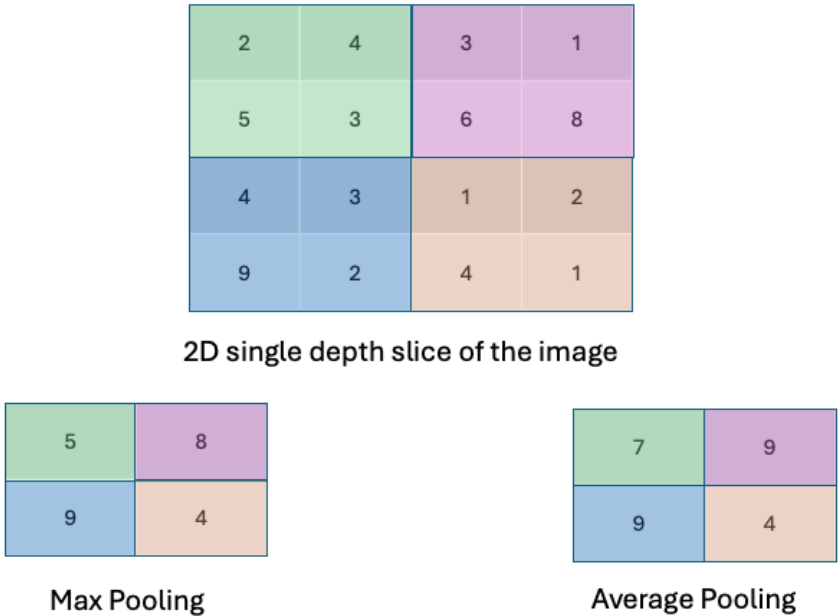


Figure 4.14: A max pooling and average pooling operation on a 2D single-depth slice of an image

Multiple convolution pooling layers are stacked together to form a deep CNN. As an image propagates through the CNN, each convolutional layer extracts specific features. The lower layers extract the gross features, such as shape, curves, and lines, while the higher layers extract more abstract features, such as eyes and lips. The image, as it propagates through the network, reduces in dimensions but increases in depth. The output from the last convolutional layer is flattened and passed to fully connected layers, as shown in the following diagram:

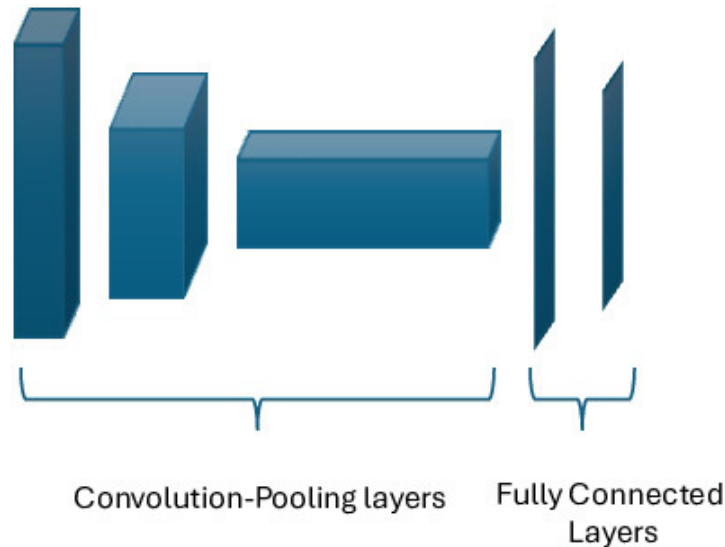


Figure 4.15: The basic architecture of a CNN network

The values of the filter matrix are also called **weights**, and they are shared by the whole image. This sharing reduces the number of training parameters. The weights are learned by the network using the backpropagation algorithm. Since we will be using the auto-differentiation feature of Keras, we will not calculate the exact expression for the weight update for convolution layers.

Milestones in CNN architectures – from LeNet to MediaPipe

The journey of CNNs through the years is a tale of innovation and breakthroughs, each new architecture building upon the last and pushing the boundaries of what's possible in computer vision:

- It all began with **LeNet-5** in 1998, developed by Yann LeCun and his team (<https://ieeexplore.ieee.org/abstract/document/726791>). This was the first successful application of a CNN and was used to recognize handwritten digits. LeNet-5's architecture, comprising two convolutional and three fully connected layers, established the foundational design for future CNNs. It introduced stacking convolutional and pooling layers, which became a standard in the field.
- Fast-forward to 2012, and **AlexNet** burst onto the scene, winning the ImageNet competition (https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf). It was the first DL model to achieve this feat, signaling a major shift in the landscape of computer vision. AlexNet's architecture, much larger than LeNet-5, featured eight layers, including five convolutional and three fully connected layers. It brought into play the ReLU activation function and overlapping pooling, techniques that have since become staples in CNN designs.

- In 2014, the **VGG-16** architecture was introduced by Karen Simonyan and Andrew Zisserman (<https://arxiv.org/pdf/1409.1556>). With 13 convolutional and 3 fully connected layers, VGG-16 was notable for its depth, comprising a total of 138 million parameters. It demonstrated the effectiveness of deeper networks, a principle that has guided much of the subsequent CNN research.
- The year 2015 saw the emergence of two more influential architectures – **Inception-v1/v3** (https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Szegedy_Rethinking_the_Inception_CVPR_2016_paper.pdf) and **ResNet-50** (https://openaccess.thecvf.com/content_cvpr_2016/papers/He_Deep_Residual_Learning_CVPR_2016_paper.pdf). The Inception models, starting with v1 and refined in v3, were among the first to use batch normalization and factorization methods for more efficient computation. Inception-v3 boasted 48 layers and 24 million parameters. Conversely, ResNet-50 introduced skip connections, a revolutionary concept that allowed the construction of even deeper networks, up to 50 layers, without running into the problem of vanishing gradients.
- In 2016, **Xception**, built on the principles of Inception-v3, took a leap forward by replacing Inception modules with depthwise separable convolutions (https://openaccess.thecvf.com/content_cvpr_2017/papers/Chollet_Xception_Deep_Learning_CVPR_2017_paper.pdf). This 71-layer deep network, with 23 million parameters, was another step in the evolution of DL architectures, emphasizing the role of depth in network design.
- Beyond these, two architectures deserve special mention – **U-Net** (https://link.springer.com/chapter/10.1007/978-3-319-24574-4_28) and **HRNet** (<https://arxiv.org/pdf/1908.07919.pdf>). U-Net, initially designed for biomedical image segmentation, was known for its efficiency with fewer training samples and its precise segmentation capabilities. Its unique U-shaped design set it apart. HRNet maintained high-resolution representations throughout the network, a key feature for tasks requiring detailed spatial information.
- Another groundbreaking addition to this lineage of CNN architectures is **YOLO (You Only Look Once)**, introduced by Joseph Redmon et al. (https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Redmon_You_Only_Look_CVPR_2016_paper.pdf). YOLO revolutionized real-time object detection with its unique approach. Unlike traditional object detectors that would first propose regions and then classify each region, YOLO framed object detection as a single regression problem, directly predicting bounding boxes and class probabilities in one evaluation. This not only simplified the object detection pipeline but also significantly increased its speed, allowing for real-time applications. YOLO's architecture is specially designed to process an entire image during training and testing, which enables it to encode contextual information about classes as well as their appearance. This global perspective reduces false positives significantly, making it a robust choice for applications such as surveillance, where speed and accuracy are paramount. The development and subsequent improvements of YOLO, including YOLOv2 and YOLOv3, have continually pushed the boundaries of speed and accuracy in object detection tasks.

- Lastly, **MediaPipe**, developed by Google, though not a single CNN architecture, has been pivotal in building perception pipelines, particularly for applications involving video and audio data (<https://arxiv.org/pdf/1906.08172v1.pdf>). Its framework allows the integration of various perception components, showcasing the adaptability of CNNs to a range of applications.

Each of these architectures, from LeNet to MediaPipe, has played a crucial role in advancing computer vision. They have not only solved complex problems but also paved the way for future innovations, influencing a wide range of applications, from healthcare to autonomous vehicles. As we move forward, the lessons learned from these pioneering designs continue to inspire and challenge the field, driving it toward new frontiers of possibility.

ResNet50 for image classification

As we progress through the next chapters, we're going to dive into the practical applications of various CNN architectures, exploring their strengths across a range of image-processing tasks. From image classification and object detection to segmentation and localization, these architectures offer a toolkit of solutions for complex visual challenges. To kickstart this journey, we'll begin with an illustrative example that showcases the remarkable capabilities of pre-trained models. Our focus will be on **ResNet50**, a model trained on the vast and diverse **ImageNet dataset**. This example demonstrates the immediate utility of leveraging pre-trained models, and it also provides a glimpse into how these powerful networks can be adapted and applied to different types of image-related tasks. By understanding and utilizing these models, we can harness the full potential of DL in computer vision, opening a world of possibilities in both research and practical applications:

1. The first step is to import the necessary modules:

```
import keras
from keras.applications.resnet50 import ResNet50
from keras.applications.resnet50 import (
    preprocess_input, decode_predictions)
import numpy as np
import matplotlib.pyplot as plt
```

2. Then, we load the ResNet50 model from the **Keras Applications** library. The `weights='imagenet'` parameter tells us that we are loading the ResNet50 model trained on the ImageNet dataset:

```
model = ResNet50(weights='imagenet')
```

3. Let's read our image. We download an image from the net and then load it, using Keras' `load_image` function. The loaded image is converted to a NumPy array to be passed to the model and preprocessed:

```
!wget https://upload.wikimedia.org/wikipedia/commons/3/3f/
Walking_tiger_female.jpg
```

```
img_path = 'Walking_tiger_female.jpg'
img = keras.utils.load_img(img_path, target_size=(224, 224))
x = keras.utils.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)
```

4. We pass the pre-processed image to the model and get model prediction with the help of Keras' predict function:

```
preds = model.predict(x)
```

5. Here, you can see the downloaded image along with the prediction and prediction probability:

```
plt.imshow(img)
plt.axis('off')

# Get top prediction
top_pred = decode_predictions(preds, top=1)[0][0]
plt.title(f"Prediction: {top_pred[1]} ({top_pred[2]*100:.2f}%)")
plt.show()
```

Prediction: tiger (90.19%)



Figure 4.16: A figure of a walking female tiger with the model prediction on top

On Spot, right. Our journey through CNNs, exploring various architectures, and finally, witnessing ResNet 50 accurately identifying a tiger from its image showcases the power of DL in visual recognition tasks. Moving on from the realm of images, we will now turn our attention to sequences and patterns over time with RNNs. The next section will delve into how RNNs capture temporal dependencies, opening new avenues for understanding and predicting sequential data.

Recurrent Neural Networks

The models that we have studied so far respond only to present input. You present them an input, and based on what they have learned, they give you a corresponding output. But this is not the way humans work. When you are reading a sentence, you do not interpret each word individually; you take the previous words into account to conclude its semantic meaning.

RNNs are able to address this issue. They use feedback loops, which preserve information. The feedback loop allows the information to be passed from the previous steps to the present. The following diagram shows the basic architecture of an RNN and how the feedback allows the passing of information from one step of the network to the next (**Unroll**):

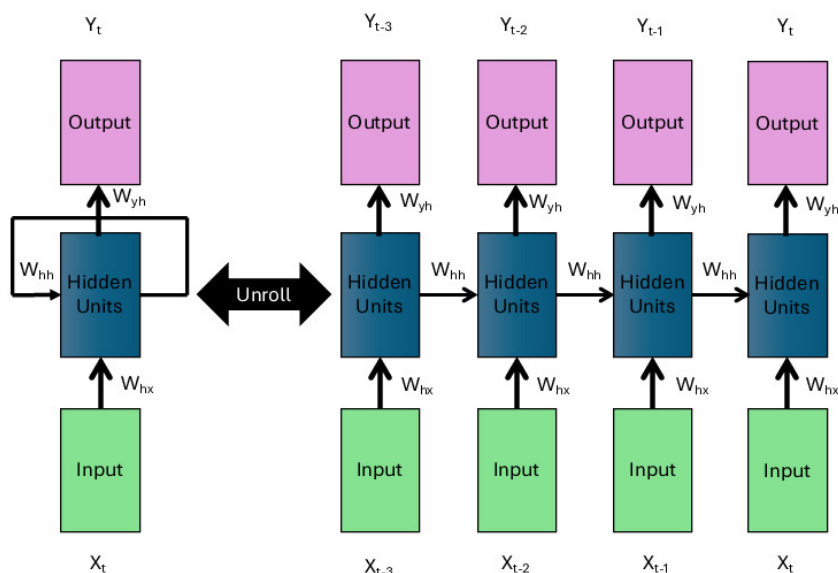


Figure 4.17: An RNN

In the preceding diagram, X represents the inputs. It's connected to the neurons in the hidden layer by weights W_{hx} . The output of the hidden layer, h , is fed back to the hidden layer via weights W_{hh} and contributes to the output, O , via weights W_{yh} . We can write the mathematical relationships as follows:

$$h_t = g(W_{hx}X_t + W_{hh}h_{t-1} + b_h)$$

$$O_t = g(W_{yh}h_t + b_y)$$

Here, g is the activation function, and b_h and b_y are the biases of the hidden and output neurons, respectively. In the preceding relationship, X , h , and O are all vectors; W_{hx} , W_{hh} , and W_{yh} are all matrices. The dimensions of the input, X , and the output, O , depend upon the dataset you are working on, and the number of units in the hidden layer, h , are decided by you; you will find many papers where

researchers have used 128 number of hidden units. The preceding architecture shows only one hidden layer, but we can have as many hidden layers as we want. RNNs have been applied in the field of natural language processing, and they have also been applied to analyze time series data, such as stock prices.

RNNs learn via an algorithm called **backpropagation through time (BPTT)**, which is a modification of the backpropagation algorithm that takes into account the time series nature of data. Here, the loss is defined as the sum of all the loss functions at times $t=1$ to $t=T$ (i.e., the number of time steps to be unrolled), such as the following:

$$\mathcal{L} = \sum_{t=1}^T \mathcal{L}^{(t)}$$

Here, $L^{(t)}$ is the loss at time t ; we apply the chain rule of differentiation, like before, and derive the weight updates for weights W_{hx} , W_{hh} , and W_{yh} .

Note

We are not deriving the expression for weight updates in this book because we will not be coding it. DL frameworks such as Keras provide built-in support for standard DL layers, such as RNN, LSTM, and GRU, and handle the complexities of BPTT internally using automatic differentiation. For those of you interested in delving into the mathematical details, check out the following references:

On the difficulty of training Recurrent Neural Networks by Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio (<https://arxiv.org/pdf/1211.5063>)

Learning Long-Term Dependencies with Gradient Descent is Difficult, Yoshua Bengio, Patrice Simard, and Paolo Frasconi (www.iro.umontreal.ca/~lisa/pointeurs/ieeetrnn94.pdf)

Also, this list would be incomplete if we didn't mention Colah's blog (<https://colah.github.io/posts/2015-08-Understanding-LSTMs/>) and Andrej Karpathy's blog (<https://karpathy.github.io/2015/05/21/rnn-effectiveness/>) for an excellent explanation of RNNs and some of their cool applications

We present the RNN with one input each timestep and predict the corresponding output. BPTT works by unrolling all input timesteps. The errors are calculated and accumulated for each timestep, and later, the network is rolled back to update the weights. One of the disadvantages of BPTT is that when the number of time steps increases, the computation also increases. This makes the overall model computationally expensive. Moreover, due to multiple gradient multiplications, the network is prone to the vanishing gradient problem.

To solve this issue, a modified version of BPTT, truncated BPTT, is often used. In truncated BPTT, data is processed one timestep at a time, and the BPTT weight update is performed periodically for a fixed number of time steps. We can enumerate the steps of the truncated-BPTT algorithm as follows:

1. Present the sequence of K_1 time steps of input and output pairs to the network.
2. Calculate and accumulate the errors across K_2 time steps by unrolling the network. Typically, $K_2 \geq K_1$, which means that the network might be unrolled for fewer steps than the inputs presented.

3. Accumulated gradients are used to update the weights by rolling up the network.

The performance of the algorithm depends on two hyperparameters, K_1 and K_2 . K_1 represents the number of time steps for which the network is presented, with previous inputs performing a weight update. This parameter affects the training speed and how frequently the weights are updated. K_2 , the number of time steps for which the network is unrolled to calculate the gradients, is crucial for capturing the dependencies in the sequence. It needs to be large enough to capture the temporal structure within the data but not so large that it becomes computationally infeasible, or leads to vanishing or exploding gradients. Choosing the optimal values of K_1 and K_2 requires balancing computational efficiency with the ability of a model to learn dependencies over time.

Long short-term memory

Hochreiter and Schmidhuber in 1997 proposed a modified RNN model, called the **long short-term memory (LSTM)**, as a solution to overcome the vanishing gradient problem. The hidden layer in the RNNs is replaced by an LSTM cell.

The LSTM cell consists of three gates – the **forget gate**, **input gate**, and **output gate**. These gates control the amount of long-term memory and the short-term memory generated and retained by the cell. The gates all have the sigmoid function, which squashes the input between 0 and 1. In the following diagram, we can see how the outputs from various gates are calculated. If the expressions seem daunting to you, don't worry – we will use Keras' LSTM class to implement the LSTM cell, as shown in the following diagram:

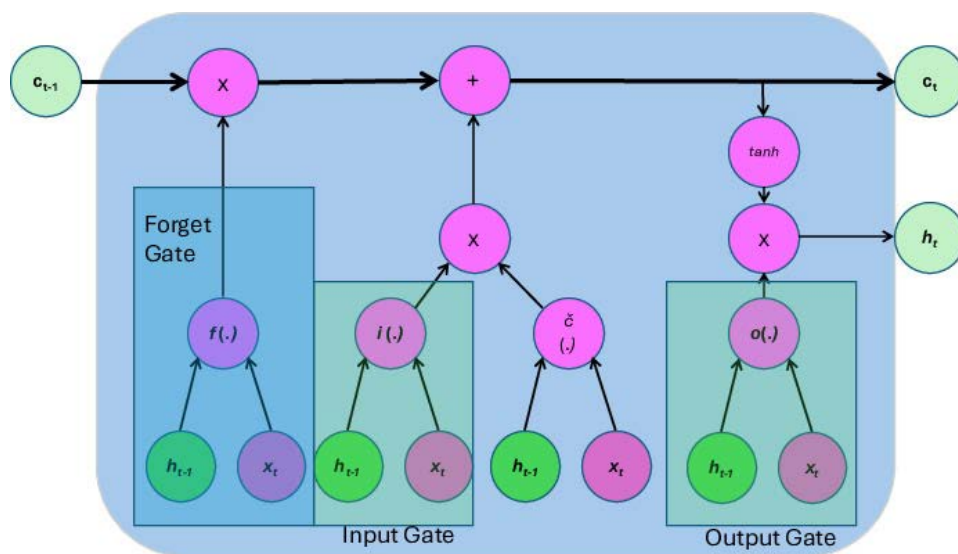


Figure 4.18: The basic LSTM cell, x is the input to the cell, h is the short-term memory, and c is the long-term memory. The subscript refers to the time

At each time step, t , the LSTM cell takes three inputs – the input, x_t , the short-term memory, h_{t-1} , and the long-term memory, c_{t-1} . It then outputs the long-term memory, c_t , and short-term memory, h_t . The subscript to x , h , and c refer to the timestep.

The forget gate, $f(\cdot)$, controls the amount of short-term memory, h , to be remembered for further flow in the present time step. It decides what information to discard from the cell state. Mathematically, we can represent the forget gate $f(\cdot)$ as:

$$f(\cdot) = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f)$$

where σ represents the sigmoid activation function, W_{fx} and W_{fh} are the weights controlling the influence of input, x_t , and the short-term memory, h_{t-1} , and b_f is the bias of the forget gate.

The input gate, $i(\cdot)$, controls the amount of input and working memory influencing the output of the cell. The input gate decides what new information is added to the cell state. We can express it as follows:

$$i(\cdot) = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i)$$

The output gate, $o(\cdot)$, controls the amount of information that's used to update the short-term memory. It decides what information from the cell state to output as the hidden state and is expressed by the following:

$$o(\cdot) = \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o)$$

As well as these three gates, the LSTM cell also computes the candidate hidden state, $\tilde{c}$, which along with the input and forget gate is used to compute the amount of long-term memory, c_t :

$$\tilde{c}(\cdot) = \tanh(W_{cx}x_t + W_{ch}h_{t-1} + b_c)$$

$$c_t = c_{t-1} \circ f(\cdot) + i(\cdot) \circ \tilde{c}(\cdot)$$

The circle represents the element-wise multiplication. The new value of the short-term memory is then computed as follows:

$$h_t = \tanh(c_t) \circ o(\cdot)$$

By using these mechanisms, LSTMs selectively remember and forget information, which makes them effective for a wide range of sequence learning tasks.

Gated recurrent unit

A **gated recurrent unit (GRU)** is another modification of RNN. It has a simplified architecture compared to LSTM and overcomes the vanishing gradient problem. It takes only two inputs, the input, x_t , at time t , and memory, h_{t-1} , from time $t-1$. There are only two gates, **Update Gate** and **Reset Gate**, as shown in the following diagram:

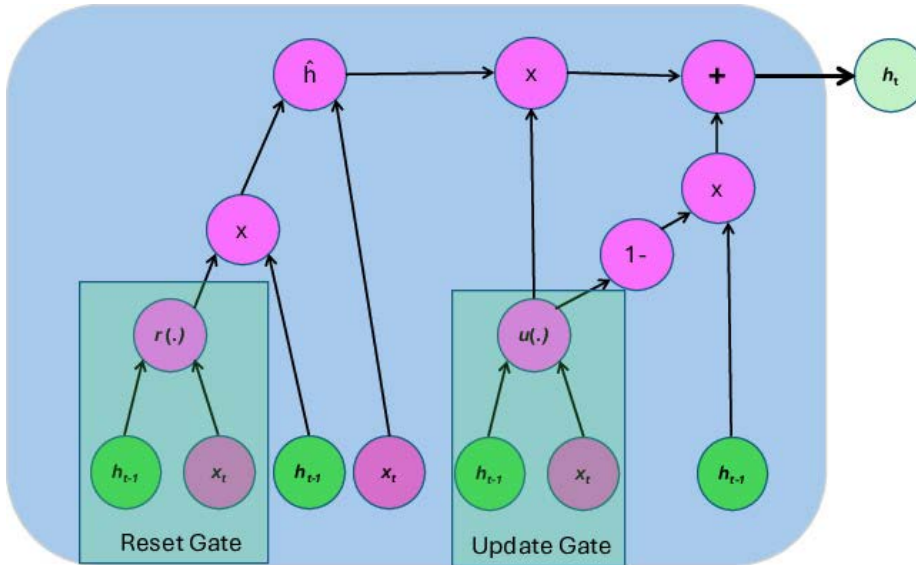


Figure 4.19: The architecture of a basic GRU cell

The update gate controls how much previous memory to keep, and the reset gate determines how to combine the new input with the previous memory. We can define the complete GRU cell with the following four equations:

- **Update gate:** $u(\cdot) = \sigma(W_{ux}x_t + W_{uh}h_{t-1} + b_u)$
- **Reset gate:** $r(\cdot) = \sigma(W_{rx}x_t + W_{rh}h_{t-1} + b_r)$
- **Candidate activation:** $\tilde{h} = \tanh(W_{\tilde{h}x}x_t + W_{\tilde{h}h}(r_t \circ h_{t-1}) + b_{\tilde{h}})$
- **Final memory at current time step:** $h_t = (u_t \circ \tilde{h}_t) + (1 - u_t) \circ h_{t-1}$

While both GRU and LSTM cells are designed to mitigate the vanishing gradient problem, GRUs are simpler with fewer parameters, making them faster to compute in many cases, although this comes at the cost of potentially reduced capacity to model complex sequences compared to LSTMs.

Exploring Autoencoders

The models we have explored so far learned using supervised learning. In this section, we will learn about autoencoders. They are feedforward, non-recurrent neural networks, and they learn through unsupervised learning. They find applications in image reconstruction, clustering, machine translation, and much more. They were initially proposed in the 1980s by Geoffrey E. Hinton and the PDP group (<http://www.cs.toronto.edu/~fritz/absps/clp.pdf>).

The autoencoder basically consists of two cascaded neural networks – the first network acts as an encoder; it takes the input, x , and encodes it using a transformation, h , to an encoded signal, y , as shown in the following equation:

$$y = h(x)$$

The second neural network uses the encoded signal, y , as its input and performs another transformation, f , to get a reconstructed signal, r , as follows:

$$r = f(y) = f(h(x))$$

The loss function is the MSE with error, e , defined as the difference between the original input, x , and the reconstructed signal, r :

$$e = X - r$$

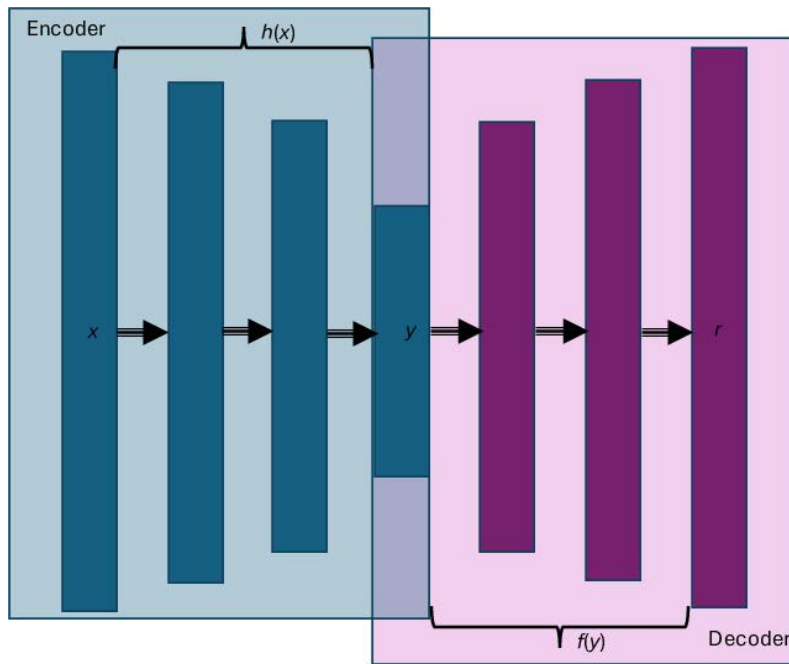


Figure 4.20: The basic architecture of an autoencoder

The preceding diagram shows an autoencoder, with **Encoder** and **Decoder** highlighted separately. Autoencoders may have weight sharing – that is, the weights of the decoder and encoder are shared. This is done by simply making them transpose each other; this helps the network learn faster, as the number of training parameters is less. There is a large variety of autoencoders – for example, sparse autoencoders, denoising autoencoders, convolution autoencoders, and variational autoencoders.

The connection between autoencoders and Transformers

As we delve into the world of autoencoders, it's worth noting a fascinating parallel in the landscape of neural network architectures – Transformers. Much like autoencoders, Transformers, which have become the driving force behind LLMs, also rely on an encoder-decoder structure.

Autoencoders are designed to encode input data into a compressed latent space and then reconstruct the input from this space, often used for tasks such as noise reduction or dimensionality reduction in images. Conversely, Transformers, introduced by Vaswani et al. in the paper *Attention Is All You Need*, use encoders to process input data (such as text) into a rich, contextualized representation, and they use decoders to generate output based on this representation, such as translating a sentence from one language to another.

The encoder in a Transformer processes the input sequence and creates a context-rich encoding. The decoder then takes this encoding and, step by step, generates an output sequence, whether it be text in a new language, a summary of a given paragraph, or even a continuation of a given text prompt. At the heart of both architectures is the transformation of data into a new form that captures the essence of the input in a useful way for reconstruction or prediction.

While the underlying principles of encoding and decoding are shared, Transformers leverage attention mechanisms to weigh the importance of different parts of the input data differently. This allows for more dynamic use of information and has led to significant breakthroughs in natural language processing tasks. The versatility and power of Transformers have been demonstrated across a range of applications, including but not limited to machine translation, text summarization, and as the backbone of generative pre-trained models that can compose text, code, and even create art from textual descriptions.

Denoising autoencoders

A denoising autoencoder learns from a corrupted (noisy) input; we feed the encoder network the noisy input, and the reconstructed image from the decoder is compared with the original denoised input. The idea is that this will help the network learn how to denoise an input. It learns to capture important features from the input data that are useful for reconstruction. This often involves understanding the underlying structure in the data, which may include relationships between pixels or patterns that are consistent across noise variations.

After training, the encoder part of the autoencoder can be used for dimensionality reduction. The reduced representation (i.e., the encoded features, y) can then be used for downstream tasks such as classification or regression. However, the removal of the decoder is typically for feature extraction purposes; the entire autoencoder is still necessary when the task is denoising new inputs.

Variational autoencoders

Another very popular autoencoder is a **variational autoencoder (VAE)**. It mixes the best of both worlds – DL and the Bayesian inference.

VAEs have an additional stochastic layer; this layer, after the encoder network, samples data using a Gaussian distribution, and the one after the decoder network samples data using Bernoulli's distribution.

VAEs can be used to generate images. VAEs allow you to set complex priors in the latent and learn powerful latent representations. We will learn more about them in a later chapter.

Introduction to OpenVINO and TinyML

OpenVINO and TinyML represent two influential technologies that are transforming IoT and edge computing, by bringing powerful AI capabilities to devices with limited computational resources.

OpenVINO, short for **Open Visual Inference and Neural Network Optimization**, is a toolkit from Intel that facilitates the fast-track development of high-performance computer vision and DL applications. It enables developers to deploy pre-trained DL models through a high-level C++ or Python inference engine API, integrated with application logic.

OpenVINO optimizes neural network models for fast execution on Intel hardware, crucial for the low-latency requirements of edge devices in IoT. It supports a wide range of DL models and is compatible with many common frameworks, such as TensorFlow and PyTorch, allowing flexibility in development. OpenVINO enhances the performance of neural network inference on edge devices, which often have constrained computational resources. It's designed for various Intel platforms, enabling AI analytics across IoT devices such as drones, cameras, and industrial robots.

Tiny Machine Learning (TinyML) is the field of ML technologies capable of performing on-device analytics at extremely low power, ideal for running on compact, battery-powered IoT devices.

TinyML models are highly optimized to run on microcontrollers and other low-power devices, extending battery life and enabling long-term deployment without frequent maintenance. By processing data locally, TinyML can provide immediate insights and actions without the need for cloud connectivity, which is essential for real-time applications in remote or network-constrained environments. Processing data on a device helps maintain privacy and security, as sensitive information does not need to be transmitted to the cloud. TinyML enables smarter devices without the need for expensive or power-hungry hardware, reducing both operational costs and device costs.

Both OpenVINO and TinyML are part of a growing ecosystem of tools that empower developers to implement AI in the real world, pushing the boundaries of what's possible with IoT and edge devices. They enable smarter analytics and decision-making at the edge, which is crucial for the evolving needs of connected devices and the IoT.

Using Keras 3 for Hyperparameter Tuning

Hyperparameter tuning is a critical step in optimizing the performance of our deep learning models. Unlike model parameters—which are learned during training—hyperparameters (such as the number of filters in a convolutional layer, the learning rate, or the dropout rate) must be set before training. In a

previous chapter, we saw how scikit-learn's GridSearchCV can be used to search over hyperparameters for conventional ML models. In this section, we will leverage **Keras Tuner** to automatically search for the best hyperparameters for a CNN using Keras 3. You will need to install Keras Tuner for this:

```
pip install keras_tuner
```

Keras Tuner provides a high-level API that integrates directly with your Keras models and supports various search algorithms (e.g., Random Search, Hyperband, and Bayesian Optimization). This allows us to efficiently explore different configurations and find the optimal settings for our model.

Keras Tuner streamlines hyperparameter optimization by letting you declare tunable parameters directly in your model-building function. For every parameter you want to tune—such as the number of filters in a convolutional layer, the kernel size, the dropout rate, or the learning rate—you use methods from the `HyperParameters` object (often named “hp”). For example, you can define an integer range with `hp.Int`, a set of choices with `hp.Choice`, or a floating-point range with `hp.Float`. Keras Tuner then systematically searches through these defined ranges across multiple trials to identify the combination that maximizes your chosen metric, such as validation accuracy. This approach greatly simplifies the experimentation process and helps you optimize your model's architecture and training settings efficiently.

Below is an example code snippet that demonstrates how to use Keras Tuner to build and tune a CNN for CIFAR 10 dataset. (For more information on CIFAR10 dataset the link is: <https://www.cs.toronto.edu/~kriz/cifar.html>)

1. We start with exporting all the necessary python modules. Beside Keras, we will be using Keras Tuner.

```
import tensorflow as tf
import keras as K
import keras_tuner as kt
```

2. Next, we define a CNN model-building function for CIFAR-10 using Keras, where key hyperparameters like the number of convolutional filters, kernel sizes, dense layer units, dropout rate, and learning rate are dynamically set using the `hp` object. The network comprises two convolutional blocks with max pooling, followed by flattening and dense layers ending with a softmax output layer for classification, and it's compiled with an Adam optimizer using sparse categorical cross entropy loss. Keras Tuner will explore different combinations of these hyperparameters to optimize the model's performance.

```
def build_model(hp):
    model = K.Sequential()

    # First Convolutional Block
    model.add(K.layers.Conv2D(
        filters=hp.Int('conv_filters', min_value=32,
```



```

        max_value=128, step=32),
        kernel_size=hp.Choice('conv_kernel', values=[3, 5]),
        activation='relu',
        input_shape=(32, 32, 3)
    ))
model.add(K.layers.MaxPooling2D(pool_size=(2, 2)))

# Second Convolutional Block
model.add(K.layers.Conv2D(
    filters=hp.Int('conv_filters_2', min_value=32,
        max_value=128, step=32),
    kernel_size=hp.Choice('conv_kernel_2', values=[3, 5]),
    activation='relu'
))
model.add(K.layers.MaxPooling2D(pool_size=(2, 2)))

# Fully Connected Layers
model.add(K.layers.Flatten())
model.add(K.layers.Dense(
    units=hp.Int('dense_units', min_value=64,
        max_value=256, step=64),
    activation='relu'
))
model.add(K.layers.Dropout(rate=hp.Float(
    'dropout_rate', min_value=0.2,
    max_value=0.5, step=0.1
)))
model.add(K.layers.Dense(
    1, activation='sigmoid')) # Binary classification

# Compile with tunable learning rate
model.compile(
    optimizer=K.optimizers.Adam(
        hp.Float('learning_rate', min_value=1e-4,
            max_value=1e-2, sampling='LOG')
    ),
    loss='binary_crossentropy',
    metrics=['accuracy']
)
return model

```

3. For hyperparameter tuning, we create a tuner object using a **Random Search** strategy, where the `build_model` function defines the model architecture with tunable hyperparameters.

The 'objective' parameter, set to 'val_accuracy', tells the tuner to optimize based on validation accuracy, while 'max_trials' defines the maximum number of different hyperparameter combinations to try—more trials allow a broader search of the parameter space. The 'executions_per_trial' parameter specifies how many times to train each model configuration to average out randomness, and the 'directory' and 'project_name' options determine where the tuning results and logs will be saved, helping you organize experiments.

```
tuner = kt.RandomSearch(
    build_model,
    objective='val_accuracy',
    max_trials=5, # Increase this for a broader search
    executions_per_trial=1,
    directory='tuner_dir',
    project_name='cifar10_tuning'
)
```

4. We load the CIFAR-10 dataset from Keras and normalize the images.

```
(x_train,y_train),(x_test,y_test)=K.datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1
x_train = x_train.astype('float32') / 255.0
x_test  = x_test.astype('float32') / 255.0
```

5. Next we run the hyperparameter search. We use a reduced number of epochs for demonstration. For real tuning, consider using more epochs.

```
tuner.search(
    x_train, y_train,
    epochs=5,
    validation_data=(x_test, y_test)
)
```

6. After the search you can know the best hyperparameters

```
best_hp = tuner.get_best_hyperparameters(num_trials=1)[0]
print("Best hyperparameters found:")
print(f"Conv Filters: {best_hp.get('conv_filters')}")
print(f"Conv Kernel: {best_hp.get('conv_kernel')}")
print(f"Conv Filters 2: {best_hp.get('conv_filters_2')}")
print(f"Conv Kernel 2: {best_hp.get('conv_kernel_2')}")
print(f"Dense Units: {best_hp.get('dense_units')}")
print(f"Dropout Rate: {best_hp.get('dropout_rate')}")
print(f"Learning Rate: {best_hp.get('learning_rate')}")
```

Once you obtain the best hyperparameters, you typically use them to rebuild and retrain your model on your complete training data, often for a longer period to fine-tune performance. You can retrieve the best model directly using `tuner.get_best_models(num_models=1)[0]` or rebuild the model using the best hyperparameters (`best_hp`) and then evaluate it on your test set, save the final model for deployment.

Summary

In this chapter, we covered some basic and useful deep neural network models. We started with a single neuron and saw its power and its limitations. The MLP was built for both regression and classification tasks. We also introduced the backpropagation algorithm. The chapter progressed to CNN, with an introduction to the convolution layers and pooling layers. We learned about some of the successful CNNs and used the first CNN, LeNet, to perform handwritten digit recognition. From the feed-forward MLPs and CNNs, we moved on to RNNs. We introduced LSTM and GRU networks. We learned about autoencoders and, finally, were introduced to the OpenVino and TinyML frameworks.

In the next chapter, we will start with a totally new type of AI model genetic algorithm. Like neural networks, it too is inspired by nature. We will use what we learned in this chapter and some of the upcoming chapters in the case studies we'll do later in the book.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/ds>



Optimization Techniques in IoT

In the previous chapter, we explored an array of **deep learning (DL)** based algorithms that have transformed our ability to recognize patterns, detect nuances, reconstruct complex data, and even generate new content in vision, speech, and text. While, at present, DL is on top in terms of both application and employability, it has close competition with **evolutionary algorithms (EAs)**. The algorithms are inspired by the natural process of evolution, the world's best optimizers. Yes, even we are the result of years of genetic evolution. In this chapter, you will be introduced to the fascinating world of evolutionary algorithms and learn about a specific type of EA, **genetic algorithms (GAs)**, in more detail. In this chapter, you will learn about the following:

- Replace with Optimization
- Replace with Deterministic and Analytic Methods
- Replace with Natural Optimization Methods
- Replace with Introduction to Genetic Algorithms
- Replace with Frameworks for EAs
- Replace with Implementing GAs
- Replace with EAs and Recent Developments

Technical requirements

To run the code in this chapter, besides Keras 3, we will need to install two more packages:

- PyGAD (<https://pygad.readthedocs.io/en/latest/>). To install, use the following command:

```
pip install pygad
```

- **Distributed EAs in Python (DEAP)** (<https://deap.readthedocs.io/en/master/>). To install, use the following command:

```
pip install deap
```

Optimization

Optimization is not a new word; we used it earlier with respect to both **machine learning (ML)** and DL algorithms, where we used the Keras auto differentiator to find the optimum model weights and biases using a form of gradient descent algorithm. In this section, we will learn a little more about optimization, optimization problems, and different techniques used to perform optimization.

In its most basic terms, **optimization** is the process of finding the best possible solution to a problem. The idea is to find the best solution, and obviously, when we talk about the best solution, it means there exists more than one solution. At its core, it's about making improvements: whether that's shortening the duration of a commute, increasing the efficiency of a machine, or enhancing the accuracy of a predictive model. The “best solution” is a relative term here, implying that while multiple solutions may exist, our goal is to identify the one that offers the greatest benefit according to our criteria. In optimization, we try to adjust our variable parameters/processes/inputs to find the optimal value of a given function, which could be to minimize costs, errors, or time, or to maximize efficiency, profit, or performance. Normally, the variables constitute the inputs; for these, we have functions called **objective functions**, **loss functions**, or **fitness functions**, and as output, we expect the cost/loss or fitness. The cost or loss should be minimized, and if we define fitness, then it should be maximized. Here, we vary the inputs (variables) to achieve a desired (optimized) output.

Note

I hope you can appreciate that calling it loss/cost or fitness is just a matter of choice. The function that calculates the cost and needs to be minimized, if we just add a negative sign to it then we expect the modified function to be maximized. As an example, minimizing $2 - x^2$ over the interval $-2 < x < 2$ is the same as maximizing $x^2 - 2$ over the same interval.

Our daily lives are full of many such optimization tasks. What will be the best route to take to the office? Which project should I do first? What topics should I read before an interview so that my success rate in the interview is maximized? The following diagram shows the basic relationship between *Input Variables*, the *Function* to be optimized, and the *Cost/Output*:

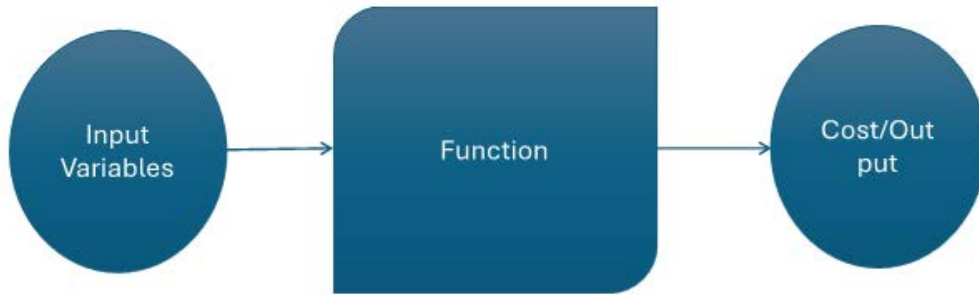


Figure 5.1: Relationship between input, the function to be optimized, and the output

The aim of optimization is to fine-tune the inputs so that we achieve the lowest possible cost while still meeting the requirements set by the constraints. The way the cost, constraints, and inputs relate to each other mathematically adds to the problem's difficulty. A central consideration is whether the cost and constraints form what's known as a **convex** shape. A **convex problem** is one where the cost and constraints curve outward and slope upwards, much like the shape of a bowl. This is important because it guarantees that there's a best solution that we can find if we look broadly enough. The following figure illustrates a convex cost function, a scenario where we can expect to find a solution that fits our needs.

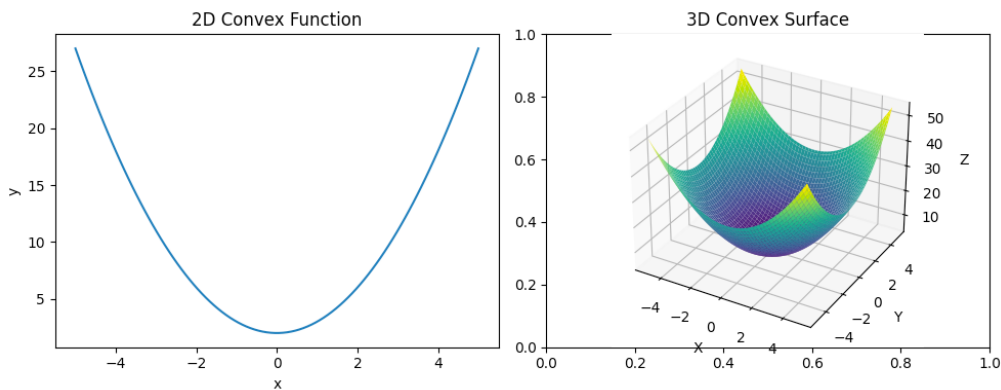


Figure 5.2: A parabolic convex function and its 3D surface counterpart

If, on the other hand, the cost function or constraints are non-convex, meaning the curve can dip and peak like a roller coaster, the optimization problem becomes more complex. In such cases, finding a solution is not guaranteed, and even if a solution exists, locating it among many possible local minima (low points) becomes a significant challenge. Unlike convex problems, where a global minimum is assured, non-convex problems may have multiple feasible solutions, making it hard to identify the optimal one.

There are various methods to solve optimization problems in mathematics and computer programming. Let's find out a little about each of them next.

Deterministic and Analytic Methods

When the objective function is smooth with a continuous second derivative, then we know from the knowledge of calculus that at a local minimum the following are true.

The gradient of the objective function, $f(x)$, at minima, x^* , is zero, that is $\nabla f(x) = 0$. This condition indicates that there is no slope at the minimum point, implying a “flat” spot or a turning point on the curve of the function.

The second derivative (Hessian $H(x^*) = \nabla^2 f(x)$) is positively definite. This condition ensures that the point is indeed a minimum, as a positive definite Hessian indicates that the function curves upwards at x^* .

In such conditions, for some problems, it is possible to find the solution analytically by determining the zeros of the gradient and verifying the positive definiteness of the Hessian matrix at the zeros. So, in these cases, we can explore the search space iteratively for the minima of the objective function. There are various search methods; let's see them.

Gradient descent method

This method is widely used for optimization in ML and DL. We learned about gradient descent and how it works in earlier chapters, and we saw that the search direction is the direction of the gradient descent, $-\nabla f(x)$. It is also called the **Cauchy-Euler method** because it was created by Cauchy, in 1847, and since then it has been very popular. In it, we start from an arbitrary point on the objective function surface and change the variables (in earlier chapters, these were the weights and biases) along the direction of the gradient. Mathematically, it is represented as follows:

$$x_{n+1} = x_n - \alpha_n \nabla f(x_n)$$

Here α_n is the step size (variation/learning rate) at iteration n . Gradient descent algorithms have worked well in training DL models, but they have some severe drawbacks:

- The performance of the optimizer used depends greatly on the learning rate and other constants. If you change them even slightly, there is a big possibility that the network may not converge. It is because of this that sometimes researchers call training a model an art, or alchemy.
- Since these methods are based on derivatives, they do not work for discrete data. We cannot reliably apply it when the objective function is non-convex, which is the case in many DL networks (especially models using a non-linear activation function).

- The presence of many hidden layers can result in many local minima, and there is a strong possibility that the model gets stuck in a local minimum. Here, you can see an example of the objective function with many local minima.

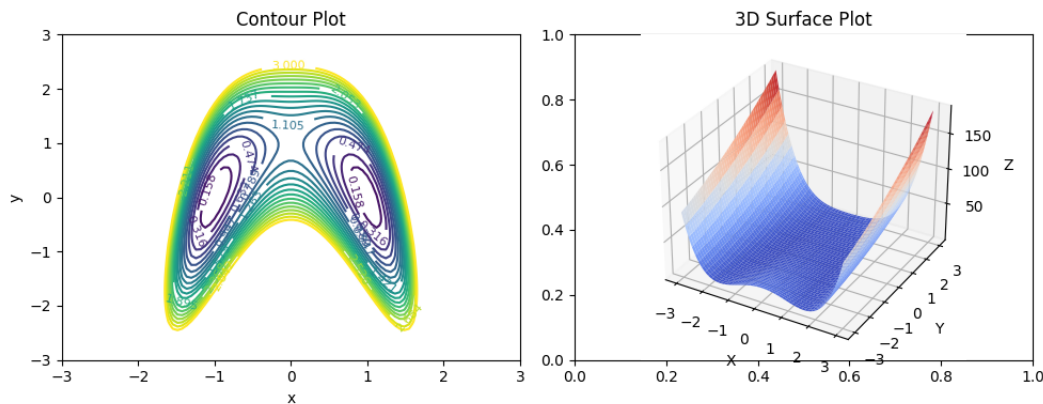


Figure 5.3: A cost function with many local minima. The one on the left is the surface plot and the one on the right shows the contour plot of the same cost function. The dark red points in the image correspond to minima.

There are many variants of the gradient descent method, each designed to improve upon the basic algorithm in different ways. These variants are widely supported across DL frameworks due to their effectiveness in optimizing complex neural networks. Some of the most popular ones include the following:

- **Stochastic gradient descent (SGD):** A modification of the gradient descent algorithm that updates the model's weights using only a single sample at a time. This makes the optimization process faster and can lead to better generalization of unseen data.
- **Adam optimizer:** This combines the advantages of two other extensions of SGD, AdaGrad and RMSProp. Adam is known for its adaptive learning rate properties, making it effective for a wide range of problems.
- **AdaGrad optimizer:** This adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of all past squared gradients. It's particularly suited for dealing with sparse data.
- **RMSProp optimizer:** An extension of AdaGrad that seeks to resolve its radically diminishing learning rates by using a moving average of squared gradients. This makes it more suited for online and non-stationary problems.

Note

To learn more about different optimizing algorithms, a good source is the paper by Sebastian Ruder titled *An overview of gradient descent optimization algorithms*; here is its link: <https://arxiv.org/pdf/1609.04747.pdf>.

Each of these optimizers tweaks the original gradient descent algorithm to improve convergence rates and performance across various types of neural network architectures and datasets.

The Newton-Raphson method

The **Newton-Raphson method** is based on the second-order **Taylor series** expansion of the objective function, $f(x)$, around the point, x^* :

$$f(x) = f(x^*) + \nabla f(x^*)(x - x^*)^T + \frac{(x - x^*)}{2!} H(x - x^*)^T + \dots$$

Here, x^* is the point about which the Taylor series is expanded, x is a point near x^* , the superscript, T , represents the transpose, and H is the Hessian matrix with elements given by the following:

$$h_{mn} = \frac{\partial^2 f}{\partial x_m \partial x_n}$$

Taking the gradient of the Taylor series expansion and equating to 0, we get this:

$$\nabla f(x) = \nabla f(x^*) + (x - x^*)H = 0$$

Assuming the initial guess as x_0 , the next point, x_{n+1} , can be obtained from the previous point, x_n , using this:

$$x_{n+1} = x_n - H^{-1} \nabla f(x_n)$$

The method uses both the first and the second partial derivatives of the objective function to find the minima. At iteration k , it approximates the objective function by a quadratic function around $x(k)$ and moves toward its minima.

Since computing the Hessian matrix is computationally expensive and not normally known, a large number of algorithms exist around approximating the Hessian; these techniques are called **quasi-Newton methods**. They can be represented as follows:

$$x_{n+1} = x_n - \alpha_n A_n \nabla f(x_n)$$

α_n is the step size (variation/learning rate) at iteration n , and A_n is the approximation to the Hessian matrix at iteration n . We construct a sequence of approximations to the Hessian, such that the following is true:

$$\lim_{n \rightarrow \infty} A_n = H^{-1}$$

Two popular quasi-Newton methods are as follows:

- Davidon-Fletcher-Powell algorithm
- Broyden-Fletcher-Goldfarb-Shanno algorithm

Note

When the approximation, A_n , to the Hessian is the identity matrix, the Newton method becomes the gradient descent method.

The major disadvantage of the Newton method is that it is not scalable to problems with a high-dimensional input feature space.

Natural Optimization Methods

Natural optimization methods are powerful algorithms that draw inspiration from the efficient problem-solving strategies observed in nature. These methods mimic processes such as evolution, swarm behavior, and even physical laws to find optimal solutions in complex situations. A key advantage is that they often don't rely on calculating derivatives of the objective function. This versatility makes them applicable to a wide range of problems involving discrete variables and non-continuous objective functions, scenarios where traditional optimization techniques might struggle.

Simulated annealing

Simulated annealing is a stochastic method. It is inspired by the physical process of annealing, in which a solid is first heated to a high enough temperature so that it melts, and then the temperature is decreased slowly; this allows the particles of the solid to arrange themselves in the lowest possible energy state and thus produce a highly structured lattice.

The algorithm begins with an initial state where random values are assigned to each variable. At each step, one or more variables are randomly selected, and a new random value is tested for those variables. If this change improves the objective function, the new assignment is always accepted, and the current state of the system updates. However, if the change worsens the objective function, it might still be accepted with a probability, P . This probability depends on the temperature, T (a control parameter), and the magnitude of the negative change in the objective function. If the change is not accepted, the current state remains unchanged. The probability, P , that we will change from state i to state j is given as follows:

$$P(i \Rightarrow j) = \exp\left(\frac{f(x_i) - f(x_j)}{T}\right)$$

As T approaches 0, the simulated annealing algorithm's behavior becomes similar to deterministic optimization methods, such as gradient descent, in that it becomes increasingly focused on moving directly toward the objective function's minimum without accepting worse solutions.

Particle swarm optimization

Particle swarm optimization (PSO) was developed by James Kennedy and Russell Eberhart in 1995. It is based on the social behavior of animals, such as a flock of birds. You must have noticed in the sky; birds fly in a V shape. Those who have studied bird behavior tell us that birds fly like this when in search of food or a better location, with the one leading being nearest to the desired source.

Now, when they fly, the leading bird does not remain the same; instead, it changes as they move. The bird in the flock that sees the food sends a sound signal, and all other birds then collect around that bird in a V fashion. This is a continuous repetitive process and has served birds well for millions of years.

PSO takes inspiration from this bird behavior and uses it to solve optimization problems. In PSO, every single solution is a bird (called a **particle**) in the search space. Each particle has a fitness value, which is evaluated by the fitness function to be optimized; they also have velocities, which direct the flying of the particles. The particles fly through the problem search space by following the current optimum particle.

The particles are moved around in the search space guided by two best fitness values: one their own best-known position in the search space (**particle best (pbest)**), the other the entire swarm's best-known fitness value (**global best (gbest)**). As improved positions are discovered, they are used to guide the movements of the particles of the swarm. This process is repeated, and it is hoped that an optimum solution will eventually be discovered.

PSO is particularly effective for solving continuous optimization problems, where the goal is to find the best parameters within a continuous domain. This makes it a powerful tool for a wide array of applications, ranging from tuning ML model parameters to optimizing engineering designs and financial models.

Genetic algorithms

When we look around the world and see different species, a question naturally arises: why are these sets of features stable and not others; why should the majority of animals have two legs or four legs, and why not three? Is it possible that the world that we see today is the result of many iterations in a grand optimization algorithm?

Let's imagine there is a cost function that measures survivability, which should be maximized. The characteristics of the organisms of the natural world fit into a topological landscape. The level of survivability (measured through adaptation) represents the elevation of the landscape. The highest points correspond to the most fit conditions, and the constraints are provided by the environment and through interaction between different species.

Then, the process of evolution can be thought of as a grand optimization algorithm that selects which characteristics produce a species of organism fit for survival. The peaks of the landscape are populated by living organisms. Some peaks are broad and hold a wide range of characteristics encompassing many organisms, while other peaks are very narrow and allow only very specific characteristics.

We can extend this analogy to include valleys between the peaks separating different species. We can also think that humankind might be at the global maximum peak of this landscape, since we have intelligence and the ability to alter the environment and ensure better survivability, even in extreme environments.

Thus, the world with different life forms can be thought of as a big search space, with different species as the result of many iterations of a grand optimization algorithm. This idea forms the basis of GAs.

In the next section, we'll explore genetic algorithms, a standout choice among the various optimization methods discussed. These algorithms stand out on the **internet of things (IoT)** for their robustness in optimizing network configurations, resource allocation, and adaptive decision-making processes. By mimicking natural evolutionary strategies, they adeptly navigate the complexities inherent to IoT systems, setting the stage for innovative solutions to emergent challenges.

Introduction to Genetic Algorithms

According to the work of Charles Darwin, the famous biologist, the animal and plant species that we see today have emerged due to millions of years of evolution. The process of evolution is guided by the principle of **survival of the fittest**, selecting the organisms that have a better chance of survivability. The plants and animals that we see today are the results of millions of years of adaptation to the constraints imposed by the environment. At any given time, a large number of varied organisms may coexist and compete for the same environmental resources.

The organisms that are most capable of both acquiring the resources and procreation are the ones whose descendants will have more chances of survival. Organisms that are less capable, on the other hand, will tend to have few or no descendants. Over time, the entire population will evolve, containing on average organisms that are fitter than the previous generations.

What makes this possible? What decides that a person will be tall, and a plant will have a particular shape of leaf? All this is encoded like a set of rules in the program on the blueprint of life itself—genes. Every living organism on Earth has this set of rules and they describe how that organism is designed (created). Genes reside in chromosomes. Each being has a different number of chromosomes and they contain thousands of genes. For example, we Homo sapiens have 46 chromosomes and these chromosomes contain about 20,000 genes. Each gene represents a specific rule: a person will have blue eyes, will have brown hair, will be a female, and so on. These genes pass from parents to offspring through the process of reproduction.

There are two ways by which genes pass from parents to offspring:

- **Asexual reproduction:** In this, the child is the duplicate copy of the parent. It happens during a biological process called **mitosis**; lower organisms such as bacteria and fungi reproduce via mitosis. Only one parent is needed in this.

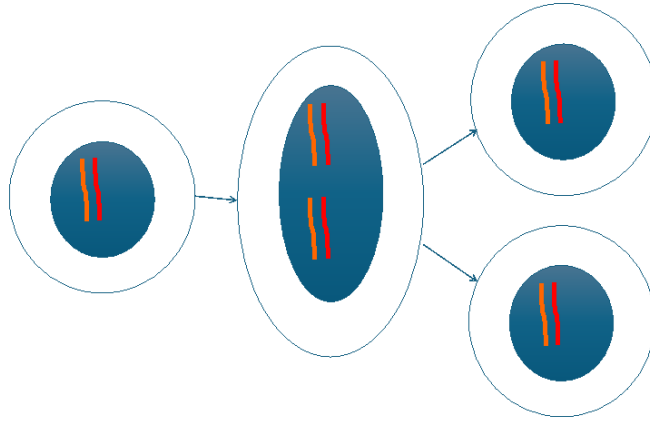


Figure 5.4: The process of mitosis: the chromosomes of the parent first double, and the cell divides into two

- **Sexual reproduction:** This happens via a biological process called **meiosis**. In this, two parents are involved initially; each parent cell undergoes a process of **crossover**, where a part of one chromosome gets interchanged with a part of another chromosome. This modifies the genetic sequence; the cells then divide into two, but with only half the number of chromosomes each. The cells containing half the number of chromosomes (**haploid**) from the two parents then meet to form a **zygote**, which later, through mitosis and cell differentiation, results in the production of an **offspring** similar to, yet different from, the parents.

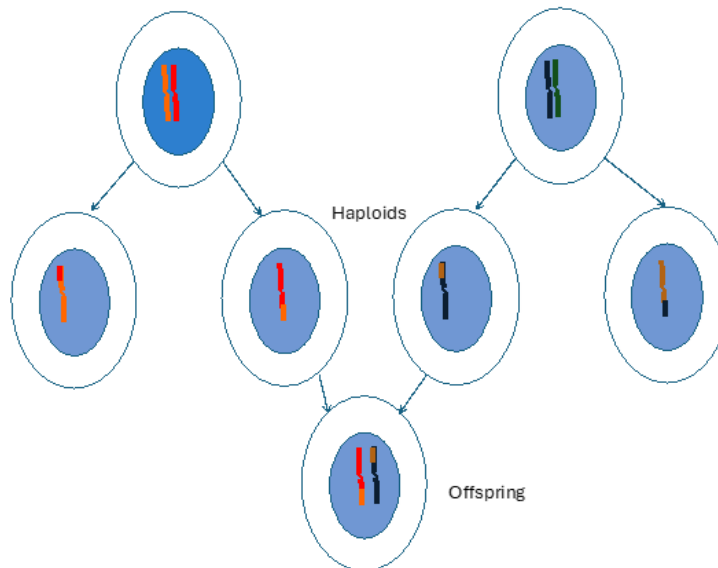


Figure 5.5: The process of meiosis: parents' cell chromosomes undergo crossover, a part of one chromosome overlaps and changes position with part of another chromosome

Another interesting thing that happens in the natural process of selection and evolution is the phenomenon of **mutation**. Here, the genes undergo a sudden change and generate a completely new gene, which was not present in either parent. This phenomenon generates further diversity.

Sexual reproduction through generations is supposed to bring about evolution and ensure that organisms with the fittest characteristics have more descendants.

The genetic algorithm

Let's now learn how can we implement the GA. This method was developed by John Holland in 1975. It was shown that it could be used to solve an optimization problem by his student Goldberg, who used GAs to control gas pipeline transmission. Since then, GAs have remained popular, and have inspired various other evolutionary programs.

To apply GAs to solving optimization problems using the computer, as the first step, we will need to encode the problem variables into genes. The genes can be a string of real numbers or a binary bit string (series of 0s and 1s). This represents a potential solution (individual) and many such solutions together form the population at time, t . For instance, consider a problem where we need to find two variables, a and b , such that the two lie in the range $(0, 255)$. For binary gene representation, these two variables can be represented by a 16-bit chromosome, with the higher 8 bits representing gene a and the lower 8 bits for b . The encoding will need to be later decoded to get the real values of the variables a and b .

The second important requirement for GAs is defining a proper **fitness function**, which calculates the fitness score of any potential solution (in the preceding example, it should calculate the fitness value of the encoded chromosome). This is the function that we want to optimize by finding the optimum set of parameters of the system or the problem at hand. The fitness function is problem-dependent. For example, in the natural process of evolution, the fitness function represents the organism's ability to operate and to survive in its environment.

Once we have decided the encoding of the problem solution in genes and decided upon the fitness function, the GA will then follow these steps:

1. **Population initialization:** We need to create an initial population, where all chromosomes are (usually) randomly generated to yield an entire range of possible solutions (the search space). Occasionally, the solutions may be seeded in areas where optimal solutions are likely to be found. The population size depends on the nature of the problem but typically contains several hundred potential solutions encoded into chromosomes.
2. **Parent selection:** For each successive generation, based on the fitness function (or randomly), we next select a certain proportion of the existing population. This selected proportion of the population will then breed to form a new generation. This is done by the method of **tournament selection**: a fixed number of individuals are randomly selected (tournament size) and the individual with the best fitness score is chosen as one of the parents.

3. **Reproduction:** We next generate the successive generation from those selected in step 2, through genetic operators such as **crossover** and **mutation**. These genetic operators ultimately result in a child (next generation) population of chromosomes that is different from the initial generation but at the same time shares many of the characteristics of its parents.
4. **Evaluation:** The offspring generated are then evaluated using the fitness function, and they replace the least-fit individuals in the population to keep the population size unchanged.
5. **Termination:** During the *Evaluation* step, if any of the offspring achieve the objective fitness score or the maximum number of generations is reached, then the GA process is terminated. Otherwise, steps 2 to 4 are repeated to produce the next generation.

Let us learn about the two operators that are important for the success of GAs, crossover and mutation, in more detail.

Crossover

To perform the crossover operation, we select a random position on the chromosome of two parents, and then the genetic information is swapped between them around this point, with a probability, P_x . This results in two new offspring. When the crossover takes place over a random point, it is called a **one-point crossover** (or **single-point crossover**).

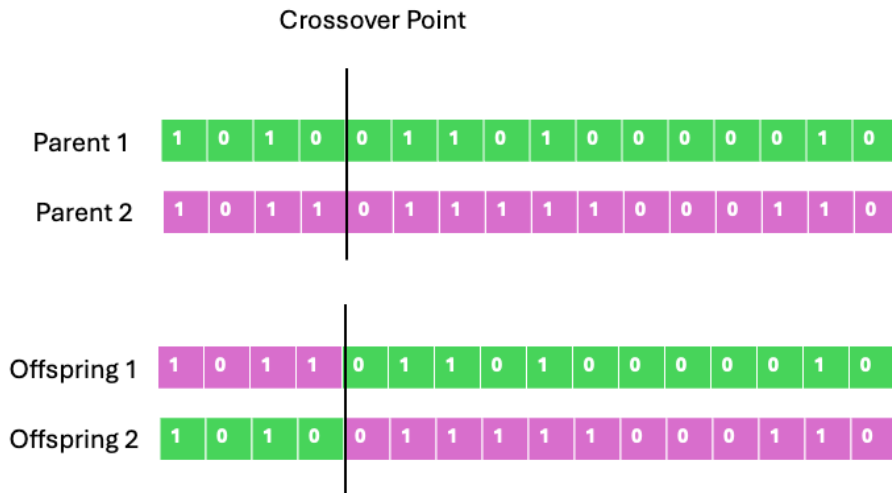


Figure 5.6: One-point crossover: a random point is selected in the parent's chromosomes and the corresponding gene bits are swapped

We can also have more than one point where parents' genes are swapped; this is called a **multi-point crossover**.

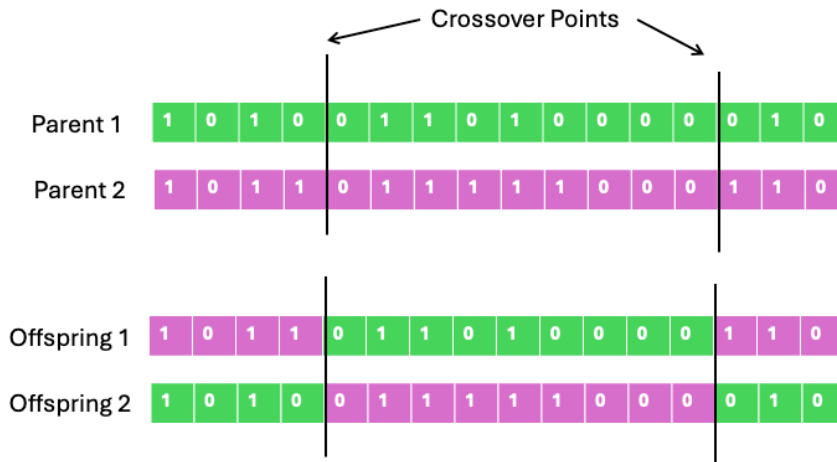


Figure 5.7: Multi-point crossover: there is more than one point where the genes of the parents are swapped. This is an example of a double-point crossover.

There are many types of crossovers people have tried, for example, uniform crossover, order-based crossover, and cyclic crossover.

Mutation

While crossover combines existing genetic material, ensuring variety and potentially speeding up the search, it doesn't introduce entirely new solutions. This is where the mutation operator plays a crucial role – it maintains and introduces diversity into the population. The mutation operator works by randomly modifying certain genes (bits) of child chromosomes with a probability, P_m .

One common mutation type is the **bit-flip mutation**. In your 16-bit chromosome example, a bit-flip mutation would cause a single bit to change its state (from 0 to 1 or vice versa).

Another type is **random resetting**, where a gene is randomly assigned a new value within its possible range.

Choosing the right mutation probability (P_m) is critical. A low P_m can lead to genetic drift (loss of diversity), whereas a high P_m might disrupt good solutions. The ideal mutation probability encourages the algorithm to sacrifice short-term fitness for potential long-term improvements.

Pros and cons

GAs sound cool, right? Now, before we try and build code around them, let's point out certain advantages and disadvantages of GAs.

Advantages

GAs offer a compelling suite of advantages for solving optimization problems, particularly shining where traditional gradient-based methods may not be effective. Their unique characteristics enable them to address a wide range of challenges across various fields. Here are some key points that highlight their strengths:

- GAs are equally adept at optimizing both continuous and discrete variables, making them applicable in diverse scenarios.
- Unlike gradient descent methods, GAs do not rely on derivative information. This means they can work with non-continuous and non-differentiable fitness functions, broadening their applicability.
- They can simultaneously explore multiple areas of the cost surface, improving the chances of finding global optima.
- GAs handle a large number of variables efficiently, with computational time increasing modestly, making them suitable for complex optimization problems.
- The generation of populations and calculation of fitness values can be executed in parallel, making GAs ideal for parallel computing environments and significantly reducing computation times.
- The crossover and mutation mechanisms enable GAs to avoid being trapped in local minima, facilitating the search for global solutions in complex landscapes.
- GAs work well with numerically generated data, experimental data, and even analytical functions, demonstrating their flexibility and utility in a wide range of applications.
- Their adaptability and robustness make GAs particularly effective for tackling large-scale optimization challenges, offering innovative solutions where other methods struggle.

These points underscore the power and flexibility of GAs as a premier choice for optimization, capable of navigating complex landscapes and delivering solutions where traditional methods may not suffice. They can be used to optimize either continuous or discrete variables.

Disadvantages

While GAs offer significant advantages in solving complex optimization problems, they are not a one-size-fits-all solution. Certain limitations affect their applicability and efficiency in specific contexts:

- For well-behaved convex functions, gradient-based approaches tend to converge more quickly. In these cases, the inherent characteristics of GAs may not offer a competitive edge in speed.
- Although a large solution population allows GAs to explore the search space thoroughly, this can also lead to slower convergence rates compared to methods that take more direct routes to optimal solutions.

- Crafting an effective fitness function is crucial for the success of GAs. However, this can be challenging, especially in problems where the evaluation criteria are not straightforward or are computationally expensive to calculate.

Understanding these limitations is essential for effectively applying GAs and ensuring they are used in scenarios where their benefits can be fully realized.

Frameworks for EAs

Python, with its rich scientific computing ecosystem, provides several excellent frameworks that streamline the implementation and experimentation with EAs. In this section, we'll delve into two popular choices: PyGAD and distributed EAs in Python (**DEAP**).

PyGAD – GA Python library

PyGAD is a user-friendly library built on top of NumPy, emphasizing simplicity for both practitioners and researchers. Its design allows for easy customization of various EA components. It supports a wide range of genetic operators (selection, crossover, mutation). It also provides flexibility to create custom operators. It integrates seamlessly with ML libraries such as PyTorch, TensorFlow, and Keras. It also offers built-in visualization tools for analysis. It is best suited when ease of use, flexibility in defining operators, and integration with other Python libraries are important considerations.

DEAP – distributed EAs in Python

DEAP is a mature and flexible framework that promotes experimentation and the development of novel EA techniques. It prioritizes transparency in algorithm representation. It allows for the implementation of almost any EA, from GAs and genetic programming to multi-objective optimization.

It is a versatile toolbox for building custom EAs with minimal effort. It supports complex operations such as mating, mutation, and selection with simple Python functions. It has capabilities for parallel computing, making it suitable for large-scale optimization problems. It also has tools for multi-objective optimization, including support for various selection methods such as NSGA-II and SPEA2.

Note

PyGAD is an excellent choice for beginners in EAs or those seeking a user-friendly and easy-to-learn framework. It excels in projects where integration with ML libraries, such as scikit-learn, TensorFlow, PyTorch, Keras, and so on, is important. PyGAD offers built-in support for various genetic operators and allows customization for specific needs.

DEAP caters to more experienced users or those tackling complex, multi-objective optimization problems. It prioritizes flexibility and control, allowing for the creation of highly customized EAs. DEAP's modular design facilitates experimentation with novel EA techniques.

Let us now try some hands-on exercises using these two frameworks.

Implementing GAs

Ready to see the power of GAs in action? In this section, we'll dive into code examples demonstrating how to harness evolutionary principles for solving real-life problems. Get ready to experiment and build your own GA-powered solutions.

Guess the word

In this program, we use GAs to guess a word. The GA will know the number of letters in the word and will guess those letters until it finds the right answer. We define each gene as an alphanumeric character, with strings of these genes forming chromosomes representing potential solutions. The fitness function evaluates the accuracy of a guess by the number of character matches with the target word. To illustrate the framework's versatility, we'll implement it in both the PyGAD and DEAP frameworks, comparing their performance and ease of use in solving this problem:

1. We begin first with the PyGAD framework. We begin by importing the `pygad` library along with the `string` library for potential string operations. Our target for this exercise is the `PYTHON345er` passcode. To evaluate the efficacy of each solution generated during the GA's evolutionary process, we define a fitness function named `fitness_func`. This function calculates a fitness score by comparing the ASCII value of each character in the solution against the ASCII value of the corresponding character in the target passcode, `PYTHON345er`. The fitness score increases by one for every character that matches, providing a simple measure of how close a given solution is to the target passcode:

```
import pygad
import string

target_passcode = "PYTHON345er"

def fitness_func(ga_instance, solution, solution_idx):
    fitness = sum(1 for i in range(
        len(target_passcode)
    ) if solution[i] == ord(target_passcode[i]))
    return fitness
```

2. We expand our setup by defining the search space for our GA. First, we create our genes, the building blocks that will form the solution, `alphabet`, which includes both ASCII letters (uppercase and lowercase) and digits with the help of Python's `string` library. From this `alphabet`, we generate `gene_space`, which consists of the ASCII values for each character in the `alphabet`. This approach allows our GA to work with numerical representations of characters. To align with our target passcode, `PYTHON345er`, we set the number of genes (`num_genes`) in each

solution to match the length of `PYTHON345er`. This ensures that each solution generated by the GA will have the same number of characters as our target passcode:

```
alphabet = string.ascii_letters + string.digits
gene_space = [ord(char) for char in alphabet]
num_genes = len(target_passcode)
```

3. We initialize a GA instance using the PyGAD library. This instance is configured to run for 200 generations, with 5 parents selected for mating in each generation. The fitness of solutions is evaluated using our previously defined `fitness_func`. We specify that each population will consist of 20 solutions, with the number of genes per solution and the gene space set according to our `PYTHON345er` target passcode. Additionally, we set the mutation rate to affect 10% of the genes, introducing variability and aiding in the exploration of the solution space:

```
ga_instance = pygad.GA(
    num_generations=200,
    num_parents_mating=5,
    fitness_func=fitness_func,
    sol_per_pop=20,
    num_genes=num_genes,
    gene_space=gene_space,
    mutation_percent_genes=10
)
```

4. We run the GA with `ga_instance.run()`, then retrieve the best solution along with its fitness and index. The solution is converted from ASCII values back into a string. Finally, we print the cracked passcode, revealing the outcome of the GA's search process:

```
ga_instance.run()
solution, solution_fitness, solution_idx = ga_instance.best_solution()
solution_str = ''.join(chr(int(gene)) for gene in solution)
print("Passcode cracked:", solution_str)
```

5. Next, let us try the same in DEAP, and so instead of PyGAD, we import DEAP. The gene pool remains the same alphabet. The fitness function is also logically the same as earlier, but the format is changed to match the DEAP format:

```
import random
from deap import base, creator, tools
import string

target_passcode = "PYTHON345er"
alphabet = string.ascii_letters + string.digits
```

```
# Define the fitness function
def fitness(individual):
    return sum(
        individual[i] == target_passcode[i]
        for i in range(len(target_passcode))
    ),
```

6. In this step, we set up the GA framework using DEAP. We define a `FitnessMax` class to maximize our fitness function, indicating we aim to maximize the number of matching characters with the target passcode. An `Individual` class is also created to represent each candidate solution as a list with a fitness attribute of type `FitnessMax`. The toolbox is then initialized to streamline genetic operations. It registers functions for creating individual genes from our alphabet, forming individuals and populations, and executing genetic operations such as evaluation, mating, mutation, and selection. Specifically, we use `cxUniform` for crossover, `mutShuffleIndexes` for mutation, and `selTournament` for selection, with parameters set to encourage diversity and exploration in our GA's search process. For detailed explanations of each setting and operation, refer to the DEAP documentation.

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)

toolbox = base.Toolbox()
toolbox.register("attr_char", random.choice, alphabet)
toolbox.register(
    "individual", tools.initRepeat,
    creator.Individual, toolbox.attr_char,
    n=len(target_passcode)
)
toolbox.register(
    "population", tools.initRepeat,
    list, toolbox.individual
)

toolbox.register("evaluate", fitness)
toolbox.register("mate", tools.cxUniform, indpb=0.5)
toolbox.register("mutate", tools.mutShuffleIndexes, indpb=0.1)
toolbox.register("select", tools.selTournament, tournsize=5)
```

7. In this code, we define the main function to execute the GA. We start by setting a random seed for reproducibility and initializing a population of 500 individuals. The algorithm is configured to run for five hundred generations, with probabilities for crossover and mutation set at 0.7 and 0.2, respectively. During each generation, we select offspring, apply crossover and mutation with the specified probabilities, and correct any length mismatches post-genetic operations to ensure offspring match the target passcode's length. The fitness of the modified

offspring is evaluated, and the population is updated accordingly. After iterating through the generations, we identify and print the best individual in terms of fitness, providing insight into the algorithm's effectiveness at approximating the target passcode:

```
def main():
    random.seed(33)
    population = toolbox.population(n=500)

    # Parameters
    probab_crossover = 0.7
    probab_mutation = 0.2
    generations = 500

    for gen in range(generations):
        offspring = toolbox.select(population, len(population))
        offspring = list(map(toolbox.clone, offspring))

        # Apply crossover
        for child1, child2 in zip(offspring[::2],
                                offspring[1::2]):
            if random.random() < probab_crossover:
                toolbox.mate(child1, child2)
                del child1.fitness.values
                del child2.fitness.values

        # Apply mutation
        for mutant in offspring:
            if random.random() < probab_mutation:
                toolbox.mutate(mutant)
                del mutant.fitness.values

        # After crossover and mutation in your
        # genetic algorithm loop
        for ind in offspring:
            if len(ind) != len(target_passcode):
                print("Length mismatch detected after genetic
                      operations.")
                ind[:] = ind[:len(target_passcode)]
        # Evaluate the individuals with an invalid fitness
        invalid_ind = [
            ind for ind in offspring if not ind.fitness.valid]
        fitnesses = map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit
```

```

population[:] = offspring

# Check for a solution
fits = [ind.fitness.values[0] for ind in population]

best_ind = tools.selBest(population, 1)[0]
print("Best Individual=", ''.join(str(x) for x in best_ind),
      ", Fitness = ", best_ind.fitness.values[0])

```

8. Now, let us run the code:

```

if __name__ == "__main__":
    main()

```

We can see that both PyGAD and DEAP were able to guess the word correctly. PyGAD offers a straightforward approach, making it accessible for those new to GAs or seeking simplicity in implementation. On the other hand, DEAP, with its comprehensive set of tools and customizable framework, caters to more complex scenarios and offers greater flexibility. This versatility, however, comes with a slightly steeper learning curve, requiring users to familiarize themselves with its components and how to best leverage them for their specific problems.

Optimizing neural network weights with GAs

In this section, we'll harness the power of GAs to optimize the weights of a neural network. Combining evolutionary computing with ML, we'll demonstrate how natural selection principles can enhance neural network performance. Using PyGAD and Keras, we'll establish a framework that iteratively evolves network weights to minimize binary cross-entropy loss and boost prediction accuracy. This technique highlights the adaptability of GAs in tackling the complexities of neural network optimization. The following code is adapted from the XOR example in the PyGAD docs:

1. We start by importing the necessary libraries: Keras for building and managing our neural network, PyGAD for implementing GAs, and NumPy for numerical operations. We then define a function, `evaluate_solution`, which plays a crucial role in our GA setup. This function evaluates the fitness of each solution generated by the GA. We use the solution vector, representing a potential set of weights for our neural network, to make predictions on our input data. The fitness of each solution is calculated based on how well these predictions match our target outputs, using the **binary cross-entropy** loss as our metric. The lower the loss, the higher the fitness score, guiding our GA toward solutions that enhance the neural network's predictive accuracy:

```

import keras
import pygad
import numpy as np

```

```
def evaluate_solution(ga_engine, solution_vector, index):
    global inputs, targets, ga_keras, nn_model
    pred = pygad.kerasga.predict(
        model=nn_model,
        solution=solution_vector,
        data=inputs
    )
    loss_function = keras.losses.BinaryCrossentropy()
    fitness_score = 1.0 / (
        loss_function(targets, pred).numpy()+ 1e-8
    )
    return fitness_score
```

- Next, we define a callback function, `generation_callback`, which is called at the end of each generation during the GA's execution. We use it to print the current generation number and the fitness score of the best solution found so far, giving us insights into the algorithm's progress and effectiveness at optimizing the neural network's weights:

```
def generation_callback(ga_instance):
    print(f"Generation = {ga_instance.generations_completed}")
    current_fitness = ga_instance.best_solution()[1]
    print(f"Fitness = {current_fitness}")
```

- Here is our Keras model. It is a simple model so that we can visualize the weights once GA finds it:

```
# Constructing the neural network model
input_node = keras.layers.Input(shape=(2,))
hidden_layer = keras.layers.Dense(4, activation="relu")(
    input_node)
output_node = keras.layers.Dense(2, activation="softmax")(
    hidden_layer)

nn_model = keras.Model(inputs=input_node, outputs=output_node)
```

- Since our model is very basic, we keep simple data and we train it for the XOR problem. Here, we define inputs and outputs:

```
# Data for XOR problem
inputs = np.array([[0, 0],
                   [0, 1],
                   [1, 0],
                   [1, 1]])
```



```
targets = np.array([[1, 0],
                    [0, 1],
                    [0, 1],
                    [1, 0]])
```

5. Now, we configure and initialize the PyGAD setup to optimize the weights of the Keras model. First, we create an instance of `pygad.kerasga.KerasGA`, linking it to our model and specifying that we want to explore 10 different solutions simultaneously. We then set the number of generations the GA will run for 200 iterations, indicating how many iterations of evolution we'll allow for our solutions. We also state that five parents will be mating in each generation, which influences the diversity and convergence of our solution space. The initial population's weights are directly derived from our Keras model through the `ga_keras.population_weights` property. With these configurations in place, we instantiate the PyGAD optimizer, `ga_optimizer`, providing it with the number of generations, the number of parents mating, the initial population weights, our custom fitness function, `evaluate_solution`, to assess solution quality, and a callback function, `generation_callback`, to track progress after each generation. This setup kickstarts the process of using GAs to find the optimal set of weights for our neural network model, aiming to improve its performance on our specified task:

```
# Setting up PyGAD with Keras
ga_keras = pygad.kerasga.KerasGA(
    model=nn_model, num_solutions=10)
generations = 200
parents_mating = 5
initial_pop_weights = ga_keras.population_weights

# Initializing PyGAD instance
ga_optimizer = pygad.GA(
    num_generations=generations,
    num_parents_mating=parents_mating,
    initial_population=initial_pop_weights,
    fitness_func=evaluate_solution,
    on_generation=generation_callback
)
```

6. Now, we run the PyGAD optimizer:

```
ga_optimizer.run()
```

7. Here is the evolution of the Fitness function as generations evolve:

```
ga_optimizer.plot_fitness(
    title="Fitness Evolution", linewidth=4)
```

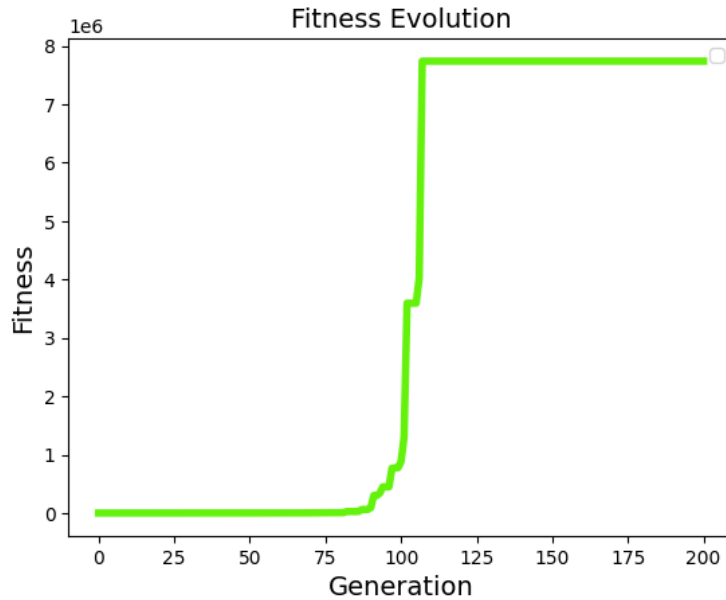


Figure 5.8: Evolution of the fitness function with generations

8. Let us now access the best solution and evaluate its loss and accuracy:

```
best_solution, best_fitness, best_index = ga_optimizer.best_
solution()
print(f"Best Fitness: {best_fitness}")
print(f"Best Solution Index: {best_index}")

# Making predictions using the best solution
best_predictions = pygad.kerasga.predict(model=nn_model,
                                          solution=best_solution,
                                          data=inputs)

print(f"Predictions:\n{best_predictions}")

# Evaluating the best model
crossentropy = keras.losses.BinaryCrossentropy()
print("Crossentropy Loss:", crossentropy(
    targets, best_predictions).numpy())

# Accuracy of the model
accuracy_metric = keras.metrics.BinaryAccuracy()
accuracy_metric.update_state(targets, best_predictions)
print(f"Model Accuracy: {accuracy_metric.result().numpy()}")
```

The results are as follows:

```
Best Fitness: 7739380.763515218
Best Solution Index: 0
Predictions:
[[1.0000000e+00 2.3234287e-08]
 [1.3747472e-11 1.0000000e+00]
 [2.4629969e-22 1.0000000e+00]
 [1.0000000e+00 9.4765156e-09]]
Crossentropy Loss: 1.192093e-07
Model Accuracy: 1.0
```

Here, we represent the weights:

```
print(pygad.kerasga.model_weights_as_matrix(
    model=model, weights_vector=best_solution))
```

The weights are as follows:

```
[array([[ -7.29260329,  6.37755817, -2.31631866, -0.14540028],
        [ 4.81178216, -7.71485941,  4.08579291,  0.2815448 ]]),
 array([ -1.07567367, -0.35969691, -1.42715861,  2.79960323]),
 array([ -7.00547248,  5.6856117 ],
        [ -7.43096919,  3.69133582],
        [ 0.27536305, -1.24861577],
        [ 1.92976453, -0.82562848])), array([ 5.8421343 ,
        -4.02149466])]
```

Through this code, we've explored the integration of GAs with neural network models using PyGAD and Keras, showcasing an effective method to optimize neural network weights. This approach highlights the versatility of GAs in ML optimization tasks, providing insights into the optimization process in a straightforward and accessible manner.

EAs and Recent Developments

Evolutionary Algorithms (EAs) draw inspiration from Darwinian principles, finding their roots in the early days of computing. While these nature-inspired methods held promise, their widespread adoption was often hampered by one major drawback: computational expense. The process of simulating generations of solutions, evaluating their fitness, and iteratively refining them demanded resources that were often not readily available. However, the landscape is rapidly changing. With the remarkable increase in computational power, we're witnessing a renaissance in EA research and applications. This section will explore recent developments, underscoring how EAs are now solving complex problems across various fields with unparalleled efficiency. From optimizing network structures in telecommunications and enhancing ML models to addressing logistical and engineering design challenges, EAs are demonstrating their versatility and adaptability. Moreover, the integration of EAs with other computational techniques is paving the way for innovative hybrid approaches, further expanding their applicability and effectiveness in tackling today's most pressing challenges.

Model architecture using EAs

In the previous section, we used GAs to find optimal weights for a multilayered perceptron model, but what about the architecture? How do we know how many layers to keep, and how many units in each layer? If I am building a CNN for a vision task, then what would be the optimal number of CNN layers? How many kernels should be in each layer? Anyone who has worked in DL can attest to the fact that, more than model training, finding the optimal architecture is a pain. But fear not; researchers have shown that EAs can be effectively used to determine the optimal model architecture for a specific task. In this section, we will cover two approaches, one on finding the optimum architecture for CNN, which is based on the paper, *Genetic CNN*, by Lingxi Xie and Alan Yuille (<https://arxiv.org/abs/1703.01513>). Despite a great architecture, suboptimal hyperparameters can hinder model performance. This is another place where EAs can help—we can use them to estimate the optimal hyperparameters for a given task and architecture.

GA for CNN architecture

In *Chapter 4*, we learned about different DL models, such as MLP, CNN, RNN, and so on. Now, we will see how we can use GAs with these DL models.

In the paper, *Genetic CNN* (<https://arxiv.org/pdf/1703.01513>), the authors proposed a strategy to use GAs to estimate CNN architecture. The key idea is to represent the architecture of a CNN as a binary string, which can then be evolved to find optimal network structures. The family of networks is encoded into fixed-length binary strings.

The CNN is divided into stages, each stage contains several nodes (e.g., v_s, k_s) representing individual convolutional layers. Each bit in the string controls whether a connection exists between specific nodes. Only connections from a lower-numbered node to a higher-numbered node are allowed. This keeps things orderly. Each node is followed by batch normalization and ReLU activation.

Each bit of the bit string represents the presence or absence of the connection between one convolution layer (node) and the other. For each stage, they employ $1 + 2 + \dots + (K_s + 1) = K_s(K_s + 1)/2$ bits to encode the inter-node connections. The first bit represents the connection between $(v_{s,1}, v_{s,2})$, the next two bits represent the connection between $(v_{s,1}, v_{s,3})$ and $(v_{s,2}, v_{s,3})$, and so on. If L is the total length of this binary string, then 2^L is the total number of different network architectures that GA can create by changing the bit string. For example, If we have a network with 3 stages and 3, 4, and 5 nodes per stage, then for stage 1 ($K_{s=1} = 3$), we require 3 bits, for stage 2 ($K_{s=2} = 4$), we require 6 bits, and for the third stage ($K_{s=3} = 5$), 10 bits are required, making $L=19$, and therefore a possibility of $2^L = 524288$ different architectures.

To understand this better, let's consider a two-stage network (each stage will have the same number of filters and filter size). Let's say stage S_1 consists of four nodes (that is $K_s = 4$), thus the total number of bits required to encode it is $(4 \times 3 \times \frac{1}{2}) = 6$. The number of convolutional filters in stage 1 is 32; also, we ensure that the convolutional operation does not change the spatial dimensions of the image (for example, padding is the same). The following diagram shows the respective bit string encoded and corresponding convolution layer connections. The connections in red are default connections and

are not encoded in the bit string. The first bit encodes the connection between (a_1, a_2) , the next two bits encode the connection between (a_1, a_3) and (a_2, a_3) , and the last three bits encode the connection between (a_1, a_4) , (a_2, a_4) , and (a_3, a_4) :

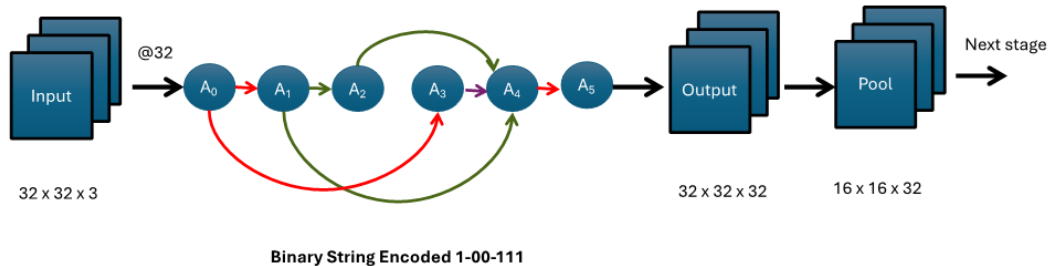


Figure 5.9: Bit string encoded and corresponding convolution layer connections

Stage 1 takes a $32 \times 32 \times 3$ input; all the convolution nodes in this stage have 32 filters. The red connections are default connections not encoded in the bit string. The green connection represents the connections according to the encoded bit string 1-00-111. The output of stage 1 goes to the pooling layer and reduces by half in the spatial dimension.

Stage 2 has five nodes, and thus will need $(5 \times 4 \times \frac{1}{2}) = 10$ bits. It will take the input from stage 1 with dimensions of $16 \times 16 \times 32$. Now, if we take the number of convolution filters in stage 2 as 64, then after the pooling of the output, it would be $8 \times 8 \times 64$. If you are interested in seeing how this works, you can refer to the GitHub repository: <https://github.com/aqibsaeed/Genetic-CNN>.

Researchers continue to explore such evolutionary approaches, with the paper, *Regularized Evolution for Image Classifier Architecture Search* (<https://ojs.aaai.org/index.php/AAAI/article/view/4405>), introducing a refinement called **regularized evolution** to further improve the search process. This method progressively evolves a population of network architectures (genotypes), with older architectures facing an increased likelihood of being replaced. The evolved **AmoebaNet-A** models achieved state-of-the-art classification accuracy on the CIFAR-10 and ImageNet datasets. It also demonstrates competitive results with significantly reduced computational cost compared to **reinforcement learning (RL)**-based and other complex **AutoML** methods.

Hyperparameter optimization using EAs

Finding the optimal set of hyperparameters (learning rate, batch size, optimizer choice, etc.) is crucial for maximizing the performance of DL models. EAs provide a powerful alternative to manual tuning or less flexible methods such as grid search. For this task, each individual in the EA population represents a potential set of hyperparameters. These can be encoded as binary strings, numerical vectors, or even more complex structures. The fitness of an individual is determined by training a model with its corresponding hyperparameters and evaluating performance on a validation set. Selection, crossover, and mutation are tailored to the hyperparameter representation to generate new and potentially better solutions. Let us go through some work employing GAs for hyperparameter tuning.

A 2017 paper, *Population Based Training of Neural Networks* (<https://arxiv.org/pdf/1711.09846.pdf>), introduces a hybrid optimization method that combines the benefits of parallel and sequential hyperparameter optimization techniques. PBT accelerates the tuning process without requiring sequential runs, optimizing hyperparameters online and adapting them throughout training based on performance. This approach is decentralized, efficient, and applicable to various domains, including deep RL and supervised learning tasks. PBT's effectiveness is demonstrated through its application to deep RL models, machine translation, and GAN training, showcasing improvements in performance and computational efficiency.

The paper by Lisha Li et al. (<https://arxiv.org/pdf/1603.06560.pdf>), introduces **Hyperband**, a novel method for hyperparameter optimization that focuses on adaptive resource allocation to evaluate and select configurations. By formulating optimization as a pure-exploration adaptive resource allocation problem, Hyperband significantly speeds up the process, allowing for the evaluation of many more configurations than traditional methods. It combines early stopping strategies to allocate resources efficiently, demonstrating faster performance than Bayesian optimization algorithms across various learning tasks. The approach is designed to be general-purpose, making minimal assumptions, and its effectiveness is supported by both theoretical analysis and empirical results.

EAs in AutoML and AutoRL

The increasing complexity inherent in modern ML and RL calls for intelligent automation of their design and optimization processes. EAs, inspired by the principles of natural selection, provide a robust and versatile solution to drive advancements in **automated ML (AutoML)** and **automated RL (AutoRL)**. EAs, with their bio-inspired mechanisms, excel in optimizing high-dimensional, non-convex problems typical in AutoML and AutoRL, where traditional gradient-based methods often falter due to local optima entrapment or the need for derivative information.

In AutoML, the goal is to automate the process of selecting and optimizing ML models, including preprocessing steps, model selection, and hyperparameter tuning. EAs approach this challenge by treating each component of the ML pipeline as genes within a chromosome, evolving these through generations to optimize a predefined fitness function, typically accuracy or predictive performance on a validation set. The use of EAs in AutoML has been demonstrated effectively in works such as Real et al. (2019, <https://ojs.aaai.org/index.php/AAAI/article/view/4405>), where they applied EAs to **neural architecture search (NAS)**, yielding architectures that rival or exceed manually designed models in terms of efficiency and performance.

The flexibility of EAs, allowing for the exploration of a vast search space without predefined pathways, makes them particularly suited for NAS within AutoML. They can discover innovative model architectures by recombining and mutating existing architectures, a process that has led to the development of highly efficient and task-specific models. This evolutionary approach to model discovery is further supported by systems such as Google's AutoML-Zero (<https://blog.research.google/2020/07/automl-zero-evolving-code-that-learns.html>), which explores the space from scratch, starting with simple mathematical operations to build ML models, underscoring the potential of EAs in driving foundational innovations in model architecture design.

AutoRL extends the concept of automation to RL, aiming to optimize policies for decision-making tasks without extensive manual tuning. EAs contribute to AutoRL by optimizing the policy parameters or the structure of the policy itself, navigating through the policy space to find efficient solutions for complex environments. This is particularly relevant in scenarios where the reward landscape is sparse or deceptive, conditions under which gradient-based optimization methods might struggle. The work by Such et al. (2018, <https://arxiv.org/pdf/1712.06567.pdf>) illustrates the power of EAs in AutoRL, where deep neuroevolution algorithms were used to evolve policies for challenging environments, achieving competitive results with traditional deep RL methods.

Moreover, EAs in AutoRL benefit from their inherent parallelizability, enabling the evaluation of multiple policies across different environments simultaneously. This feature significantly accelerates the search process, a crucial advantage given the computationally intensive nature of training RL models. It also allows for a more thorough exploration of the policy space, increasing the likelihood of discovering novel and effective policies.

The integration of EAs into AutoML and AutoRL presents a paradigm shift toward more exploratory, adaptive optimization techniques that promise to unlock new possibilities in automated system design. However, challenges remain, such as the computational cost associated with evaluating large populations of candidates and the need for effective mechanisms to maintain diversity within the population to prevent premature convergence on suboptimal solutions.

Future directions in leveraging EAs for AutoML and AutoRL include the development of more efficient evaluation strategies, such as surrogate modeling or transfer learning, to reduce the computational burden. Additionally, there is ongoing research into hybrid models that combine the exploratory strengths of EAs with the efficiency of gradient-based methods, aiming to balance exploration and exploitation in the optimization process.

Summary

This chapter introduced an interesting nature-inspired algorithm family: GAs. We covered various standard optimization algorithms, varying from deterministic models, to gradient-based algorithms, to EAs. The biological process of evolution through natural selection was covered. We then learned how to convert our optimization problems into a form suitable for GAs. Crossover and mutation, two very crucial operations in GAs, were explained. While it is not possible to extensively cover all the crossover and mutation methods, we did learn about the popular ones.

We applied what we learned to three very different optimization problems. We used it to guess a word. The example was of a five-letter word; had we used simple brute force, it would take a search of a 615 search space. We used GAs to optimize the CNN architecture; again, note that, with 19 possible bits, the search space is 219. Then, we used it to find the optimum hyperparameters for an LSTM network.

In the next chapter, we will talk about another intriguing learning paradigm: RL. This is another natural learning paradigm, in the sense that in nature, we normally do not have supervised learning; rather, we learn through our interactions with the environment. In the same manner, here, the agent is not told anything except the rewards and punishments it receives from the environment after its action.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/ds>



6

Reinforcement Learning for IoT

Reinforcement Learning (RL) is very different from both supervised and unsupervised learning. It's the way most living beings learn—interacting with the environment. In this chapter, we'll study different algorithms that are employed for RL. You'll learn what RL is, see how it's different from supervised learning and unsupervised learning, understand the various elements of RL, and explore some fascinating applications of RL in the real world. Additionally, you'll gain an understanding of the OpenAI interface for training RL agents, delve into Q-learning and use it to train an RL agent, and discover **Deep Q-Networks (DQNs)** to train an agent to play Atari. You'll also learn about the policy gradient algorithm, use stable baselines to train an RL agent, and finally, train an RL agent to walk using **Deep Deterministic Policy Gradient (DDPG)**.

As you progress through the chapter, you'll learn about the following:

- Introduction to RL
- Simulated Environments
- Introducing Gymnasium
- Q-learning
- Q-network
- Policy gradients

Technical requirements

For this chapter, you will need the following:

- Gymnasium:

```
pip install gymnasium[all]
```

- Stable Baselines3:

```
pip install "stable-baselines3[extra]>=2.0.0a4"
```

Introduction to RL

Have you ever observed infants and how they learn to turn over, sit up, crawl, and even stand? Have you watched how baby birds learn to fly? The parents throw them out of the nest, they flutter for some time, and they slowly learn to fly. All of this learning involves a component of the following:

- **Trial and error:** The baby tries different ways and is unsuccessful many times before finally succeeding in doing it.
- **Goal-oriented:** All of the efforts are put toward reaching a particular goal. The goal for the human baby can be to crawl, and for a baby bird, it can be to fly.
- **Interaction with the environment:** The only feedback that they get is from the environment.

The human baby learning to crawl or baby bird learning to fly are both examples of RL in nature.

RL, in **Artificial Intelligence (AI)**, can be defined as a computational approach to goal-directed learning and decision-making, from interaction with the environment, under some idealized conditions. Let's elaborate upon this since we'll be using various computer algorithms to perform the learning—it's a computational approach. In all of the examples that we'll consider, the agent (learner) has a specific goal that it's trying to achieve—it's a goal-directed approach. The agent in RL isn't given any explicit instructions. It only learns from its interaction with the environment. This interaction with the environment, as shown in the following diagram, is a cyclic process. The agent can sense the state of the environment, and the agent can perform specific well-defined actions on the environment. This causes two things: first, a change in the state of the environment occurs, and second, a reward is generated (under ideal conditions). This cycle continues:

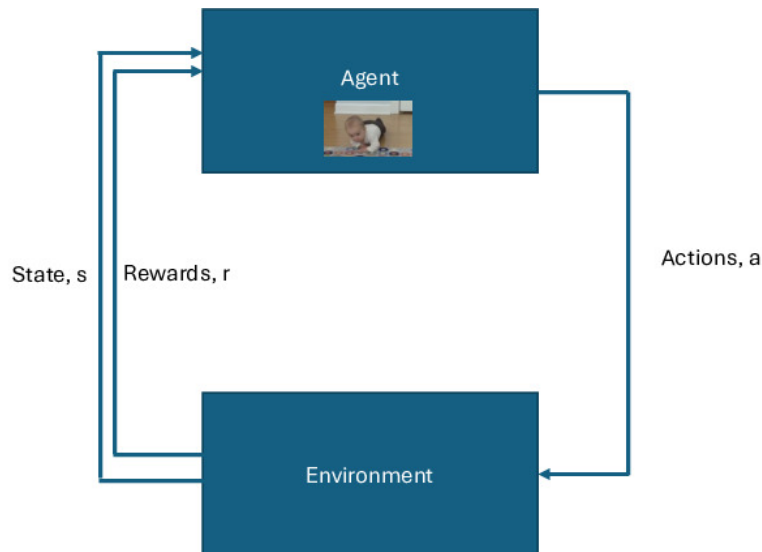


Figure 6.1: RL paradigm

Unlike supervised learning, the agent isn't presented with any examples. The agent doesn't know what the correct action is. Also, unlike with unsupervised learning, the agent's goal isn't to find some inherent structure in the input (the learning may yield some structure, but that isn't the goal). Instead, its goal is to maximize the rewards (in the long run).

RL terminology

Before learning different algorithms, let's accustom ourselves to RL terminology. For illustration purposes, let's consider two examples: an agent finding a route in a maze and an agent steering the wheel of a **Self-Driving Car (SDC)**. The two are illustrated in the following diagram:

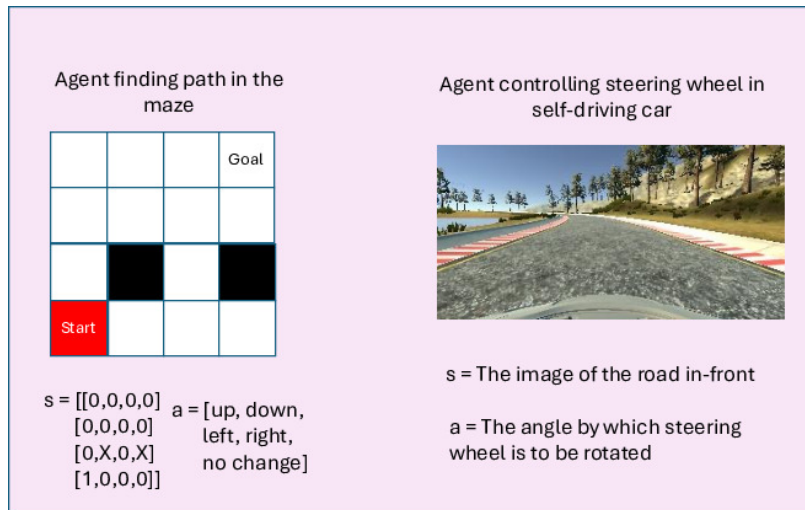


Figure 6.2: Two example RL scenarios

Before going further, let's acquaint ourselves with common RL terms:

- **States s :** The states can be thought of as a set of tokens (or representations) that can define all of the possible states the environment can be in. The state can be continuous or discrete. For example, in the case of an agent finding a path through a maze, the state can be represented by a 4×4 array, with 0 representing an empty block, 1 representing a block occupied by the agent, and X representing the state that can't be occupied. The states here are discrete in nature. In the case of an agent steering the wheel, the state is the view in front of the SDC. The image contains continuous valued pixels.
- **Actions $a(s)$:** Actions are the set of all possible things that the agent can do in a particular state. The set of possible actions, a , depends on the present state, s . Actions may or may not result in a change of state. They can be discrete or continuous. The agent in the maze can perform five discrete actions [**up**, **down**, **left**, **right**, and **no change**]. The SDC agent, on another hand, can rotate the steering wheel in a continuous range of angles.

- **Reward** $R(s, a, s')$: It's a scalar value returned by the environment when the agent selects an action. It defines the goal; the agent gets a higher reward if the action brings it near the goal, and a low (or even negative) reward otherwise. How we define a reward is totally up to us—in the case of the maze, we can define the reward as the Euclidean distance between the agent's current position and goal. The SDC agent reward can be that the car is on the road (positive reward) or off the road (negative reward).
- **Policy** $\pi(s)$: It defines a mapping between each state and the action to take in that state. The policy can be deterministic—that is, a well-defined policy for each state. As for the maze agent, a policy can be that if the top block is empty, the agent should move up. The policy can also be stochastic—that is, where an action is taken by some probability. It can be implemented as a simple look-up table, or it can be a function dependent on the present state. The policy is the core of the RL agent. In this chapter, we'll learn about different algorithms that help the agent to learn the policy.
- **Value function** $V(s)$: It defines the goodness of a state in the long run. It can be thought of as the total amount of reward the agent can expect to accumulate over the future, starting from the state s . You can think of it as long-term goodness as opposed to the immediate goodness of rewards. What do you think is more important, maximizing the reward or maximizing the value function? Yes, you guessed right: just as in chess, we sometimes lose a pawn to win the game a few steps later, and so the agent should try to maximize the value function. There are two ways in which the value function is normally considered:
 - **Value state function** $V^\pi(s)$: It's the goodness of the state following the policy π . Mathematically, at state s , it's the expected cumulative reward from following the policy π :

$$V^\pi(s) = E \left[\sum_{t=0}^{\infty} \gamma^t R_t \mid s_0 = s, \pi \right]$$

- **Action value function (or Q-function)** $Q^\pi(s, a)$: It's the goodness of a state s , taking action a , and thereafter following policy π . Mathematically, we can say that for a state-action pair (s, a) , it's the expected cumulative reward from taking action a in state s and then following policy π :

$$Q^\pi(s) = E \left[\sum_{t=0}^{\infty} \gamma^t R_t \mid s_0 = s, A_0 = a, \pi \right]$$

γ is the discount factor, and its value determines how much importance we give to the immediate rewards compared to rewards received later on. A high value of discount factor decides how far into the future an agent can see. An ideal choice of γ in many successful RL algorithms has been a value of 0.97.

- **Model of the environment**: It's an optional element. It mimics the behavior of the environment, and it contains the physics of the environment; in other words, it defines how the environment will behave. The model of the environment is defined by the transition probability to the next state.

Note

An RL problem is mathematically formulated as a **Markov Decision Process (MDP)**, and it follows the Markov property—that is, the current state completely characterizes the state of the world.

Deep RL

Diving deeper into the world of deep RL, this section discusses different methods employed for training RL algorithms and showcases some of the most impressive and successful applications of RL. RL algorithms can be classified into two main categories based on their primary focus in the learning process. Those categories are value-based methods and policy-based methods:

- **Value-based methods:** In these methods, the algorithms take the action that maximizes the value function. The agent here learns to predict how good a given state or action would be. Hence, here, the aim is to find the optimal value. An example of the value-based method is **Q-learning**. Consider, for example, our RL agent in a maze. Assuming that the value of each state is the negative of the number of steps needed to get from that box to the goal, at each time step, the agent will choose the action that takes it to a state with optimal value, as in the following diagram. So, starting from a value of -6, it'll move to -5, -4, -3, -2, -1, and eventually reach the goal with the value 0.
- **Policy-based methods:** In these methods, the algorithms predict the best policy that maximizes the value function. The aim is to find the optimal policy. An example of the policy-based method is **policy gradients**. Here, we approximate the policy function, which allows us to map each state to the best corresponding action.

We can use neural networks as a function approximator to get an approximate value of either policy or value. When we use deep neural networks as a policy approximator or value approximator, we call it **Deep Reinforcement Learning (DRL)**. DRL has, in the recent past, given very successful results. Hence, in this chapter, our focus will be on DRL.

Some successful applications of RL

In the last few years, RL has been successfully used in a variety of tasks, especially in game-playing and robotics. Let's acquaint ourselves with some success stories of RL before learning its algorithms:

- **AlphaGo Zero:** Developed by Google's DeepMind team, AlphaGo Zero is focused on mastering the game of Go without any human knowledge. It starts from an absolutely blank slate (tabula rasa). AlphaGo Zero uses one neural network to approximate both the move probabilities and values. This neural network takes as input the raw board representation. It uses a Monte Carlo Tree search guided by the neural network to select the moves. The RL algorithm incorporates a look-ahead search inside the training loop. It was trained for forty days using a 40-block residual

CNN and, over the course of training, it played about 29 million games (a big number!). The neural network was optimized on Google Cloud using TensorFlow, with 64 GPU workers and 19 CPU parameter servers. You can access the paper at <https://www.nature.com/articles/nature24270>.

- **AI-controlled sailplanes:** Microsoft developed a controller system that can run on many different autopilot hardware platforms such as Pixhawk and Raspberry Pi 3. It can keep the sailplane in the air without using a motor, by autonomously finding and catching rides on naturally occurring thermals. The controller helps the sailplane to operate on its own; it detects and uses thermals to travel without the aid of a motor or a person. They implemented it as a partially observable MDP. They employ the Bayesian RL and use the Monte Carlo tree search to search for the best action. They've divided the whole system into level planners—a high-level planner that makes a decision based on experience and a low-level planner that uses Bayesian RL to detect and latch onto thermals in real time. You can see the sailplane in action at Microsoft News: <https://news.microsoft.com/features/science-mimics-nature-microsoft-researchers-test-ai-controlled-soaring-machine/>
- **Locomotion behavior:** In the *Emergence of Locomotion Behaviours in Rich Environments* paper (<https://arxiv.org/pdf/1707.02286.pdf>), DeepMind researchers provided the agents with rich and diverse environments. The environments presented a spectrum of challenges at different levels of difficulty. The agent was provided with difficulties in increasing order; this led the agent to learn sophisticated locomotion skills without performing any reward engineering.

RL is being applied in diverse fields with impressive outcomes. In robotics, RL enables robots to perform complex tasks such as pancake flipping and precision archery, showcasing its potential to enhance robotic adaptability and efficiency in dynamic environments. In the gaming industry, RL's capabilities extend to complex multiplayer video games and simulation modeling, improving strategies in logistics and manufacturing to enhance forecasting and strategic planning. Additionally, RL is transforming personalization systems in technology platforms, optimizing user engagement by tailoring content and notifications to individual preferences at companies such as YouTube and Facebook. In the financial sector, RL has been effectively used to optimize trading executions, exemplified by JPMorgan Chase's LOXM system (<https://informaconnect.com/the-latest-in-loxm-and-why-we-shouldnt-be-using-single-stock-algos/>), which utilizes RL for trading orders to maximize speed and pricing efficiency. These applications underscore the versatile and transformative impact of RL across various sectors, promising further innovations and improvements in technology and business processes.

Now that we have been familiarized with some interesting applications of RL, let us explore the tools that can help us build our own RL agents.

Simulated Environments

Since RL involves trial and error, it makes sense to train our RL agent first in a simulated environment. While a large number of applications exist that can be used for the creation of an environment, some popular ones include the following:

- **Gymnasium:** Gymnasium is an open source platform for developing and comparing RL algorithms, forked from OpenAI Gym. Now maintained by the Farama Foundation, it includes updates and new features to meet the evolving needs of the field. As OpenAI Gym is no longer maintained, Gymnasium has become the preferred choice for researchers. In this chapter, we'll use Gymnasium to train our RL agents.
- **Unity ML-Agents SDK:** It allows developers to transform games and simulations created using the Unity editor into environments where intelligent agents can be trained using DRL, evolutionary strategies, or other machine learning methods through a simple-to-use Python API. It works with Keras and provides the ability to train intelligent agents for two-dimensional/three-dimensional and VR/AR games. You can learn more about it at <https://github.com/Unity-Technologies/ml-agents>.
- **Gazebo:** In Gazebo, we can build three-dimensional worlds with physics-based simulation. Gazebo, together with the **Robot Operating System (ROS)** and the OpenAI gym interface, is **gym-gazebo** and can be used to train RL agents. To learn more about this, you can refer to the whitepaper at <https://arxiv.org/pdf/1903.06278>.
- **Blender learning environment:** It's a Python interface for the Blender game engine, and it also works over OpenAI gym. It has Blender as its base. A free three-dimensional modeling software with an integrated game engine, this provides an easy-to-use, powerful set of tools for creating games. It provides an interface to the Blender game engine, and the games themselves are designed in Blender. We can then create a custom virtual environment to train an RL agent on a specific problem (<https://github.com/LouisFoucard/gym-blender>).

Now that we have covered various simulated environments, let's dive into one of the more important ones: Gymnasium.

Introducing Gymnasium

Gymnasium is an open source platform for developing and comparing RL algorithms, initially forked from the well-known OpenAI Gym. This toolkit is maintained by the Farama Foundation, which has taken over its development to ensure continuous improvement and integration of new features and environments. Gymnasium builds on the solid foundation provided by OpenAI Gym, enhancing it with bug fixes, new features, and updates that keep pace with the evolving needs of the RL community. OpenAI Gym, once the standard for such environments, is no longer actively maintained, making Gymnasium the preferred choice for researchers and developers in this field. In this chapter, we will use Gymnasium to train our RL agents, leveraging its robust and comprehensive environment offerings to

explore the capabilities of modern RL techniques. To start, you'll first have to install gym. For Python 3.10+, you can install gym using pip:

```
pip install gymnasium
```

In this chapter, we will be playing around with Atari games as well, so doing this install will be helpful:

```
pip install "gymnasium[atari, accept-rom-license]"
```

OpenAI Gym supports various environments, from simple text-based ones to three-dimensional ones. The environments supported in the latest version can be grouped as follows:

- **Atari:** This environment provides a wide variety of classical Atari/arcade games.
- **Box2D:** It includes environments that include toy games with physics-based control, such as car racing.
- **Classic control:** This contains classical control theory problems, such as balancing a cart pole.
- **MuJoCo:** MuJoCo multi-joint dynamics with contact provides environments that support robotics, graphics, animation, and similar fields. It requires the MuJoCo engine, which was made freely available by DeepMind (<https://mujoco.org/>).
- **Toy text:** It's a simple text-based environment, for example, blackjack—very good for beginners.

In addition to these, Gymnasium also supports a wide number of third-party environments.

Let us start by seeing the available environments in Gymnasium. To get a list of all available environments in our installation, we can use the following code:

```
import gymnasium as gym
from gym import envs
print(envs.registry.values())
```

This will provide a list of all installed environments along with their environment ID, which is a string. It's also possible to add your own environment in the gym registry. To create an environment, we use the `make` command with the environment name passed as a string. For example, to create a game using the Pong environment, the string we need will be `Pong-v4`. The `make` command creates the environment, and the `reset` command is used to activate the environment. The `reset` command returns the environment in an initial state. The state is represented as an array:

```
import gymnasium as gym
env = gym.make('Pong-v4', render_mode= 'rgb_array')
obs, info = env.reset()
env.render()
```

The state space of `Pong-v4` is given by an array of the $210 \times 160 \times 3$ size, which actually represents the raw pixel values for the Pong game. On the other hand, if you create a `Blackjack-v1` environment, the state is defined by a $500 \times 600 \times 3$ array. We can visualize the environment using the `render` command. The following diagram shows the rendered environment for the `Pong-v4` and `Blackjack-v1` environments at the initial state:



Figure 6.3: Rendered environment for `Pong-v4` (left) and `Blackjack-v1` (right)

For every environment, there is a set of valid actions that can be taken, defined in `env.action_space`. For example, in a game where the action space consists of six discrete values. We can select a random action using the `sample()` function from this space. The chosen action is then applied to the environment through the `step()` function, which executes the action and returns the new state, the obtained reward, and two Boolean flags (truncated and terminated) indicating whether the episode has ended due to time limit or the game reached the goal state. It also includes some debug information that, while not typically used by the RL agent, can be valuable for development. The following example demonstrates a game of Pong where the agent plays randomly. We store each game state in an array called `frames` for later visualization:

```
frames = []
for _ in range(300):
    frames.append(env.render())
    action = env.action_space.sample()
    obs, reward, terminated, truncated, info = env.step(action)

    if terminated or truncated:
        observation, info = env.reset()
```

We can use IPython and Matplotlib to visualize these frames as video-right in a Jupyter notebook. Here is the code:

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

fig, ax = plt.subplots()
ax.axis('off') # Turn off axes

patch = ax.imshow(frames[0], animated=True)

def animate(i):
    patch.set_data(frames[i])

anim = animation.FuncAnimation(
    fig, animate, frames=len(frames), interval=100)
HTML(anim.to_jshtml())
```

Alternatively, we can record gameplay for later review using the `save_video` function from the Gymnasium library's `utils` module, which allows for saving rendered frames as videos with customizable triggers and parameters. The provided script leverages this functionality in the `Pong-v4` environment. It sets up an automated session where the game is played using randomly chosen actions, and upon reaching game termination or truncation, it captures and stores the gameplay footage. Each session is initiated with the environment being reset, and the process repeats until a set number of steps is completed. This approach is particularly useful for generating and analyzing gameplay data to train or evaluate AI models, enabling thorough testing and refinement in various simulated scenarios. Finally, to ensure optimal resource management, the environment is properly closed at the script's conclusion:

```
import gymnasium as gym
from gymnasium.utils.save_video import save_video

env = gym.make("Pong-v4", render_mode="rgb_array_list")
_ = env.reset()
step_start_index = 0
ep_index = 0

for step_index in range(5000):
    action = env.action_space.sample()
    _, _, terminated, truncated, _ = env.step(action)
```

```

    if terminated or truncated:
        save_video(
            env.render(),
            "videos",
            fps=env.metadata.get("render_fps", 30),
            step_starting_index=step_start_index,
            episode_index=ep_index
        )
        step_start_index = step_index + 1
        ep_index += 1
        env.reset()
env.close()

```

The output of this code is an untrained RL agent attempting to play a game of Pong. Despite having no prior training, the agent managed to lose the game by 20-2. That's not bad, right? Watch the video to observe its initial performance:

https://youtube.com/shorts/vRu_eOdJmX4?feature=shared.

To explore the diverse capabilities of Gymnasium and understand the specifics of its various environments, refer to the detailed documentation available at Gymnasium's official site (<https://gymnasium.farama.org/>). Experiment with different environments to gain practical experience. Now, let's shift our focus back to the core concepts of RL algorithms, starting with an in-depth look at Q-learning.

Q-learning

In his doctoral thesis titled *Learning from delayed rewards* (1989), Watkins introduced the concept of Q-learning. The goal of Q-learning is to learn an optimal action selection policy. Given a specific state, s , and taking a specific action, a , Q-learning attempts to learn the value of the state s . In its simplest version, Q-learning can be implemented with the help of look-up tables. We maintain a table of values for every state (row) and action (column) possible in the environment. The algorithm attempts to learn the value—that is, how good of an idea it is to take a particular action in the given state.

We start by initializing all of the entries in the Q-table to zero; this ensures all states have a uniform (and hence equal chance) value. Later, we observe the rewards obtained by taking a particular action and, based on the rewards, we update the Q-table. The update in Q-value is performed dynamically with the help of the Bellman equation, given by the following:

$$Q(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha(R_t + \gamma \max_a Q(s_{t+1}, a_t))$$

Here, α is the learning rate. The figure that follows explains the basic Q-learning algorithm.

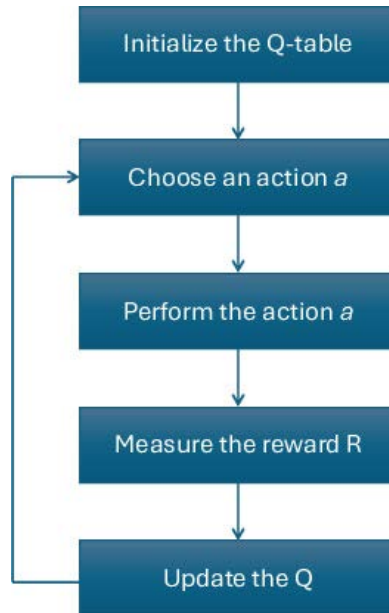


Figure 6.4: Simple Q-learning algorithm

Note

If you're interested, you can read the 240-page Watkins doctoral thesis at http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.

At the end of learning, we'll have a good Q-table, with optimal policy. An important question here is: how do we choose the action at the second step? There are two alternatives; first, we can choose the action randomly. This allows our agent to explore all of the possible actions with equal probability but, at the same time, ignores the information it has already learned. The second way is that we can choose the action for which the value is maximum. Initially, all of the actions have the same Q-value; however, as the agent will learn, some actions will get a high value and others a low value. In this case, the agent is exploiting the knowledge it has already learned. So what's better: exploration or exploitation? This is called the **exploration-exploitation trade-off**. A natural way to solve this problem is by relying on what the agent has learned, but at the same time, sometimes just exploring. This is achieved via the use of the **epsilon greedy algorithm**. The basic idea is that the agent chooses the actions randomly with the probability, ϵ , and exploits the information learned in previous episodes by a probability, $(1-\epsilon)$. The algorithm chooses the best option (greedy) most of the time $(1-\epsilon)$ but sometimes (ϵ) , it makes a random choice. Let's now try to implement what we learned in a simple toy text problem.

Taxi drop-off using Q-learning

The simple Q-learning algorithm involves maintaining a table of the size $m \times n$, where m is the total number of states and n is the total number of possible actions. Therefore, we have chosen a problem from the toy-text group since their state and action space is small. For illustrative purposes, we have chosen the Taxi-v3 environment. The goal of our agent is to choose a passenger at one location and drop them off at another. The agent receives +20 points for a successful drop-off and loses 1 point for every time step it takes. There's also a 10-point penalty for illegal pick-up and drop-off. The state space has walls shown by bushes and four location marks, R, G, Y, and B respectively. The taxi is shown by a yellow taxi icon. The pick-up and drop-off location can be either of these four location marks. The pick-up point is colored blue, and the drop-off is colored purple. The Taxi-v3 environment has a state space of size 500 and an action space of size 6, making a Q-table with $500 \times 6 = 3000$ entries.



Figure 6.5: Taxi-v3 environment

For the taxi problem, where an agent must pick up a passenger at one location and drop them off at another, the Q-learning algorithm works by updating a Q-table that represents the state-action values. Each entry in the Q-table corresponds to a potential state the agent might be in and an action the agent might take. Over time, by exploring the environment and updating the Q-values based on the rewards received, the agent learns which actions are best to take in each state.

The following is a Python implementation of the Q-learning algorithm applied to the taxi problem using the Gymnasium library:

1. We start by importing the necessary modules and creating our environment. Since here, we just need to make a look-up table, using Keras won't be necessary. As mentioned previously, the Taxi-v3 environment has five hundred possible states and six possible actions:

```
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
env = gym.make('Taxi-v3', render_mode="rgb_array")
obs, info = env.reset()
env.render()
```

2. We initialize the Q-table of the 500×6 size with all zeros, and define the hyperparameters: γ , the discount factor, and α , the learning rate. We also set the values for maximum episodes (one episode means one complete run from reset to termination or truncated being True) and maximum steps in an episode the agent will learn for:

```
m = env.observation_space.n # size of the state space
n = env.action_space.n # size of action space
print("The Q-table will have {} rows and {} columns, resulting
in \ total {} entries".format(m,n,m*n))

# Intialize the Q-table and hyperparameters
gamma = 0.97
max_episode = 1000
max_steps = 100
alpha = 0.7
epsilon = 0.3

Q = np.zeros([m,n])
```

3. Now, for each episode, we choose the action with the highest value, perform the action, and update the Q-table based on the received rewards and future state using the Bellman equation:

```
for i in range(max_episode):
    s = env.reset()
    state_index = s[0]
    done = False
    for _ in range(max_steps):
        p = np.random.rand()
        if p > epsilon or (not np.any(Q[state_index,:])):
            a = env.action_space.sample() #explore
        else:
            a = np.argmax(Q[state_index,:]) # exploit
        s_new, r, terminated, truncated, _ = env.step(a)
        Q[state_index,a] = (
            (1-alpha)*Q[state_index,a] +
            alpha*(r + gamma*np.max(Q[s_new,:])))
        )
        s = s_new
        state_index = s#[0]
        if terminated or truncated:
            break
```

4. Let us compare the Q-table before and after the training:

```
plt.figure(figsize=(5, 15))
sns.heatmap(Q, annot=False, cmap='coolwarm')
plt.title('Q-table Heatmap after learning')
plt.xlabel('Actions')
plt.ylabel('States')
plt.show()
```

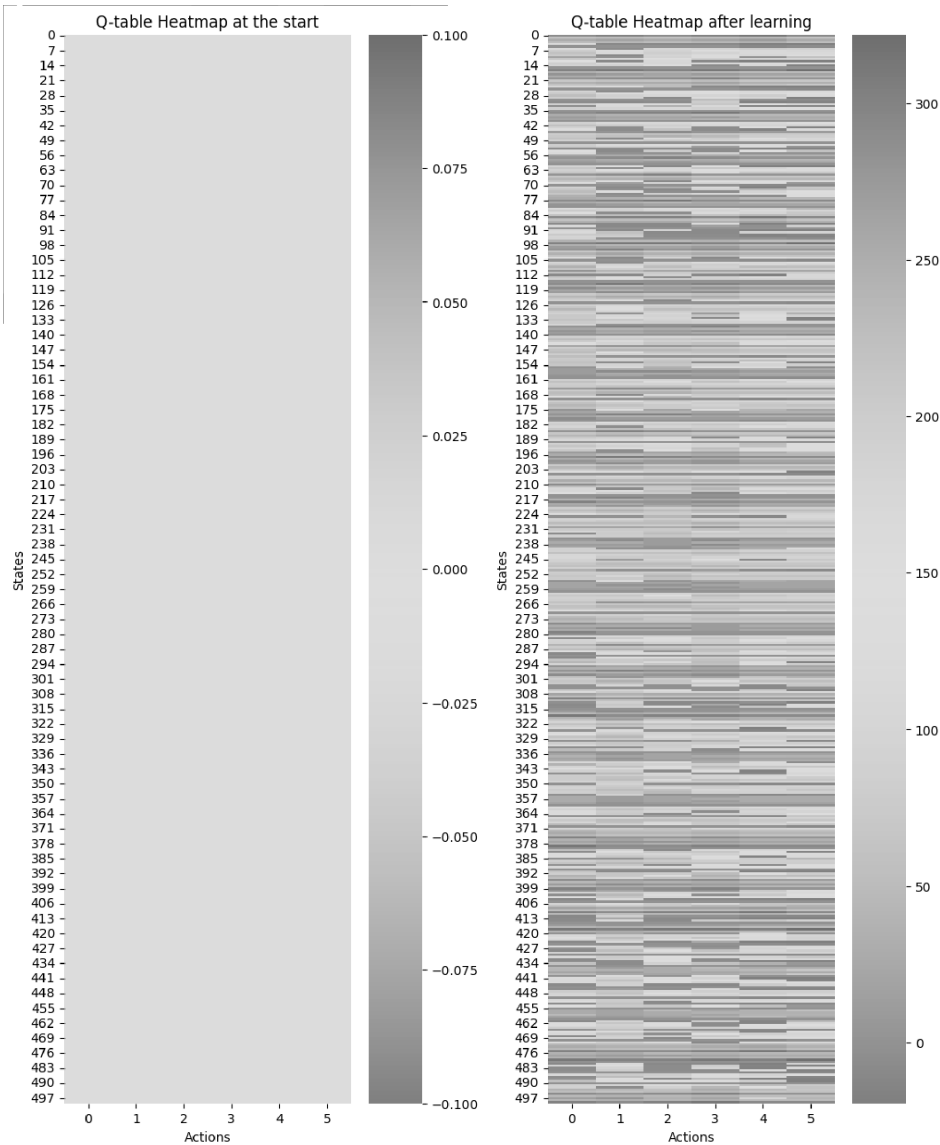


Figure 6.6: Q-table before and after training

5. Here is our trained agent making the travel decisions for picking the person up and dropping them off:

```
s = env.reset()
done = False
frames = []
frames.append(env.render())

s = s[0]
step_starting_index = 0
episode_index = 0
max_steps = 400
# Test the learned Agent
for i in range(max_steps):
    a = np.argmax(Q[s,:])
    s, _, terminated, truncated, _ = env.step(a)
    frames.append(env.render())
    if terminated or truncated:
        break
```

You can use the `animate` function we defined in the preceding section to see the video (one such clip can be found at https://www.youtube.com/watch?v=DFvRN5Fn_kg).

Witnessing an agent flawlessly navigate through its environment after learning can feel almost magical. As we continue on this journey, we'll now venture into more complex environments with intricate state spaces. This will challenge our agent further, testing the robustness and adaptability of its learned strategies. Let's see how our agent evolves to meet these new challenges.

Q-network

The simple Q-learning algorithm involves maintaining a table of the size $m \times n$, where m is the total number of states and n is the total number of possible actions. This table-based method becomes impractical for environments with large state or action spaces due to memory and computational constraints. This means we can't use it for large state and action space. An alternative is to replace the table with a neural network acting as a function approximator, approximating the Q-function for each possible action. The weights of the neural network in this case store the Q-table information (they match a given state with the corresponding action and its Q-value). When the neural network that we use to approximate the Q-function is a deep neural network, we call it a DQN.

The neural network takes the state as its input and calculates the Q-value of all of the possible actions. Let us first try Q-network on our old taxi problem.

Taxi drop-off using Q-network

In the taxi drop-off scenario, we can employ a neural network with 500 input neurons representing the state as a 1×500 **one-hot vector**, and 6 output neurons, each corresponding to the Q-value of a specific action. The network's goal is to approximate the Q-values so they align with the target Q-values derived from the Bellman equation:

$$Q_{target} = R + \gamma \max_a Q(s_{t+1}, a)$$

We train the network to minimize the squared error between the predicted Q-values (Q_{pred}) and the target Q-values (Q_{target}), expressed by the loss function:

$$loss = E_{\pi} [Q_{target}(s, a) - Q_{pred}(s, W, a)]^2$$

This approach updates the network's weights through backpropagation to minimize this loss, effectively learning the Q-value function. Let's begin by constructing a simple neural network with one hidden layer and six output neurons to approximate these Q-values:

1. We start by importing various libraries and modules necessary for building neural network models and performing numerical and graphical operations. It uses TensorFlow and Keras for model creation and optimization, specifically importing functions to define models and layers, as well as a **Stochastic Gradient Descent (SGD)** optimizer. Additionally, it includes Gymnasium for creating and managing environments for agent-based simulations, NumPy for numerical operations, and both Matplotlib and Seaborn for data visualization, enabling the creation of plots and statistical graphics:

```
import tensorflow as tf
from keras import Model
from keras.layers import Input, Dense
from keras.optimizers import SGD
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

2. Next, we define the QNetwork class using Keras. It is a custom neural network model built specifically for the Q-learning algorithm. The class inherits from Model and initializes a network with a single dense layer, specifying the number of inputs m and neurons n . The model is compiled with mean squared error as the loss function and SGD as the optimizer, where the alpha learning rate can be adjusted. The call method defines the forward pass, get_action computes and returns the action with the highest Q-value for a given state, learnQ updates the model weights based on the target Q-values Q_hat , and Qnew provides

the Q-values for a given state. This setup allows the model to learn and make decisions based on the state input from the environment:

```
class QNetwork(Model):
    def __init__(self, m, n, alpha):
        super(QNetwork, self).__init__()
        # Define the architecture
        self.input_layer = Input(shape=(m,))
        self.dense_layer = Dense(
            n, kernel_initializer='random_normal',
            bias_initializer='random_normal'
        )

        # Compile model
        self.compile(optimizer=SGD(learning_rate=alpha),
                      loss='mse')

    def call(self, inputs):
        x = self.dense_layer(inputs)
        return x

    def get_action(self, s):
        Q_values = self(tf.convert_to_tensor([s],
                                              dtype=tf.float32))
        # Extract the first element to get a scalar
        action = tf.argmax(Q_values, axis=1).numpy()[0]
        return action, Q_values.numpy()

    def learnQ(self, s, Q_hat):
        self.train_on_batch(
            x=tf.convert_to_tensor([s], dtype=tf.float32),
            y=tf.convert_to_tensor([Q_hat], dtype=tf.float32)
        )

    def Qnew(self, s):
        return self(tf.convert_to_tensor([s],
                                          dtype=tf.float32)).numpy()
```

3. Now we initialize an environment using Gymnasium. Specifically, it creates an instance of the 'Taxi-v3' environment, which is set to return observations as RGB arrays:

```
env = gym.make('Taxi-v3', render_mode="rgb_array")
obs, info = env.reset()
```

4. The code segment below sets up the foundational elements for implementing a Q-learning algorithm with a Q-Network approach in the Taxi-v3 environment. It starts by extracting the sizes of the state space (m) and action space (n) from the environment, which define the dimensions of the Q-table. This table will have m rows and n columns, resulting in $m*n$ total entries, each entry representing a Q-value for a state-action pair. The script then initializes the Q-table and sets several hyperparameters:

```
m = env.observation_space.n # size of the state space
n = env.action_space.n # size of action space
gamma = 0.97
max_episode = 50
max_steps = 10
alpha = 0.7
epsilon = 0.3
```

5. This line of code initializes an instance of the `QNetwork` class defined earlier, creating a Q-learning neural network model named `QNN`. The network is configured with the sizes of the state space (m) and action space (n), along with the learning rate (α). This model will be used to approximate the Q-values for different state-action pairs in the Taxi-v3 environment. This means it will essentially be serving as a function approximator for the Q-table in traditional Q-learning, providing a more scalable approach to handling large or continuous state spaces:

```
QNN = QNetwork(m, n, alpha)
```

6. Next, we train a Q-learning agent using a neural network to optimize actions in the Taxi-v3 environment, balancing exploration and exploitation, and updating Q-values through repeated episodes:

```
rewards = [] # To store total rewards per episode

for i in range(max_episode):
    s = env.reset() # Start with new environment
    S = np.identity(m)[s[0]] # One-hot encode the state
    done = False
    rtot = 0 # Reset total reward for the episode

    for jj in range(max_steps):
        # Get action from Q-network(not using Q-values yet):
        a, _ = QNN.get_action(S)
        p = np.random.rand()
        if p > epsilon:
            # Explore: random action:
            a = env.action_space.sample()
```

```

    # print(i, jj, a, p)
    # Perform action:
    s_new, r, truncated, terminated, _ = env.step(a)
    rtot += r # Accumulate reward

    # Prepare next state
    S_new = np.identity(m)[s_new] # One-hot encode the new
                                   # state
    Q_new = QNN.Qnew(S_new) # Get new Q-values from network
    maxQ = np.max(Q_new) # Find max Q-value for updating

    # Update target Q-value for training
    Q_hat = QNN.Qnew(S) # Get current Q-values
    Q_hat[0, a] = r + gamma * maxQ # Apply Bellman equation
    QNN.learnQ(S, Q_hat) # Train network

    S = S_new # Update state

    if done:
        break # Exit loop if the episode is finished

    rewards.append(rtot) # Store total reward for this episode

# Print average reward
print("Total reward per episode is: " + str(sum(rewards) / max_
episode))

```

7. Let us now visualize the performance of an RL agent by plotting the total rewards per episode and a running average of these rewards over a specified window size (fifty episodes):

```

window_size = 50 # Example window size for averaging
smoothed_rewards = [
    np.mean(rewards[i-window_size:i+1]) if i >= window_size
    else np.mean(rewards[:i+1]) for i in range(len(rewards))
]

plt.figure(figsize=(10, 5))
# Original rewards, more transparent:
plt.plot(rewards, label='Rewards per Episode', alpha=0.5)
plt.plot(
    smoothed_rewards,
    label='Running Average',
    color='black') # Smoothed rewards
plt.xlabel('Episode')
plt.ylabel('Total Reward')

```

```
plt.title('Total Reward per Episode with Running Average')
plt.legend()
plt.grid(True)
plt.show()
```

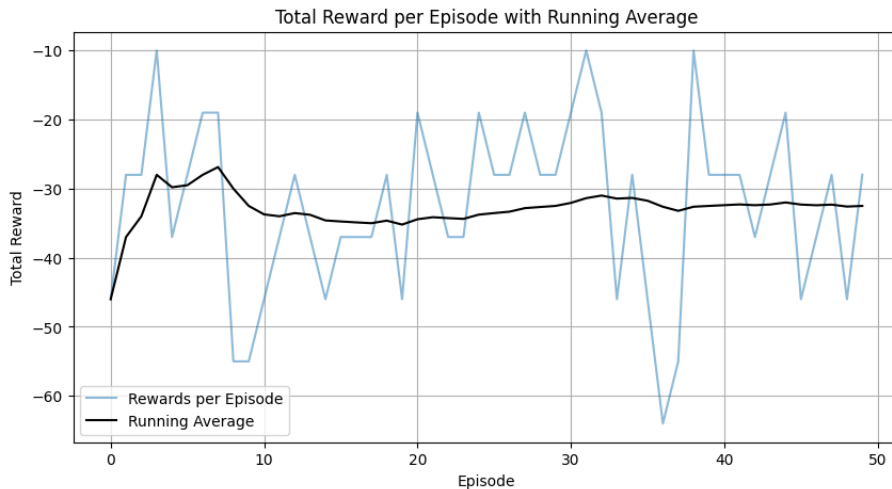


Figure 6.7: Total reward per episode with running average

This should have performed well, but even after one thousand training episodes, the network still receives a high negative reward, suggesting it hasn't learned effectively. Instead of showing improved decision-making, the network's actions appear random, performing worse than a basic Q-table approach. This issue is evident in the reward plot (Figure 6.7) from the training phase, where we would expect rewards to progressively increase as the network learns. Instead, the plot shows rewards fluctuating erratically, similar to a random walk, which indicates no significant learning progress.

What happened? Why is the neural network failing to learn, and can we make it better?

Consider the scenario when the taxi should go west to pick up and, randomly, the agent chooses west. The agent gets a reward and the network will learn that, in the present state (represented by a one-hot vector), going west is favorable. Next, consider another state similar to this one (correlated state space): the agent makes the west move again, but this time it results in a negative reward, so now the agent will unlearn what it had learned earlier. Hence, similar state actions with divergent targets confuse the learning process. This is called **catastrophic forgetting**. The problem arises here because consecutive states are highly correlated and so, if the agent learns in sequence (as it does here), this extremely correlated input state space won't let the agent learn.

Can we break the correlation between sequential inputs presented to the network? Absolutely. By employing a **replay buffer**, we store experiences as tuples of (state, action, reward, new state) regardless of the order in which they occur. Actions are chosen randomly, ensuring a diverse range of states and

outcomes. This buffer acts like a training dataset where experiences are sampled randomly to reduce correlation. This technique, known as **experience replay**, not only helps in breaking the sequence of correlated states but also enhances learning by repeatedly using past experiences, recalling rare events, and generally optimizing the use of accumulated data. Essentially, it transforms the problem into a form of supervised learning where the random sampling of inputs helps the network to generalize better.

Another challenge with our approach is the immediate updating of target Q-values, which can introduce undesirable correlations. In Q-learning, we aim to minimize the **Temporal Difference (TD)** error between the predicted Q and the target Q. Instant updates to the target Q create a correlation with the weights being adjusted, leading to a scenario akin to **chasing a moving target**, which can hinder stable learning. To address this, we can employ fixed Q-targets using two networks: one for predicting Q-values and another for the target Q-values. The predictive Q-network updates its weights continuously, while the target Q-network's weights are updated less frequently. This strategy helps stabilize the learning process by providing a consistent target during the updates.

Finally, we make one more small change: right now, our epsilon value remains constant throughout the learning process. However, ideally, our exploration strategy should evolve as learning progresses. Initially, when the agent has limited knowledge, it should explore extensively. Over time, as it learns more about the environment, it should increasingly rely on this knowledge, thus reducing the need for exploration. We can implement this by decreasing epsilon gradually as the agent becomes more familiar with the environment, mimicking the natural progression from exploration to exploitation.

Equipped with these tricks, let's now build a DQN to play an Atari game!

DQN to play an Atari game

The DQN we'll learn here is based on a DeepMind paper (<https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf>). At the heart of DQN is a deep convolutional neural network that takes the raw pixels of the game environment (just like any human player would see) as input, captured one screen at a time. As output, it returns the value for each possible action. The action with the maximum value is the chosen action:

1. The first step is to import all the essential libraries and modules for building and training an RL agent:

```
import random
import gymnasium as gym
import math
import numpy as np
from collections import deque
import tensorflow as tf
import keras
from keras.models import Sequential
from keras.layers import Dense, Conv2D, MaxPooling2D, Flatten
from keras.optimizers import Adam
```

2. Next, we set two constants: EPOCHS and THRESHOLD. EPOCHS indicates the number of times the learning or training process will run through the entire game. THRESHOLD is set to 10, which is the cutoff determining when our agent has achieved sufficient performance:

```
EPOCHS = 500
THRESHOLD = 10
```

3. Next, we define a DQN Python class. The architecture incorporates a primary neural network and a secondary target network to stabilize learning by providing consistent targets for Q-value updates. The model structure includes several convolutional layers designed to process visual inputs effectively:

```
class DQN():
    def __init__(self, env_string, batch_size=64,
                  IM_SIZE = 84, m = 4):
        self.memory = deque(maxlen=5000)
        self.env = gym.make(
            env_string, render_mode = "rgb_array_list")
        input_size = self.env.observation_space.shape[0]
        action_size = self.env.action_space.n
        self.batch_size = batch_size
        self.gamma = 1.0
        self.epsilon = 1.0
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995
        self.IM_SIZE = IM_SIZE
        self.m = m

        alpha=0.01
        alpha_decay=0.01
        # if MONITOR: self.env = gym.wrappers.Monitor(
        # self.env, '../data/'+env_string, force=True)

        # Init model
        self.model = Sequential()
        self.model.add( Conv2D(
            32, 8, (4,4), activation='relu', padding='valid',
            input_shape=(IM_SIZE, IM_SIZE, m)))
        #self.model.add(MaxPooling2D())
        self.model.add( Conv2D(
            64, 4, (2,2), activation='relu',padding='valid'))
        self.model.add(MaxPooling2D())
        self.model.add( Conv2D(
            64, 3, (1,1), activation='relu',padding='valid'))
```



```

self.model.add(MaxPooling2D())
self.model.add(Flatten())
self.model.add(Dense(256, activation='elu'))
self.model.add(Dense(action_size, activation='linear'))
self.model.compile(
    loss='mse', optimizer=Adam(learning_rate=alpha))
self.model_target = keras.models.clone_model(self.model)

```

4. The class stores data (state, action, reward, next state) in a memory buffer to minimize correlations between sequential updates, thus enhancing the stability and efficiency of training:

```

def remember(self, state, action, reward, next_state, done):
    self.memory.append((
        state, action, reward, next_state, done))

```

5. It uses an **epsilon-greedy strategy** (choose_action) to balance exploration with exploitation, gradually decreasing epsilon to favor more exploitative actions as learning progresses:

```

def choose_action(self, state, epsilon):
    if np.random.random() <= epsilon:
        return self.env.action_space.sample()
    else:
        return np.argmax(self.model.predict(
            state, verbose=0))

```

6. Additionally, the network preprocesses images by converting them to grayscale and resizing, focusing on critical visual features for decision-making (preprocess_state and combine_images):

```

def preprocess_state(self, img):
    if img.ndim != 3 or img.shape[2] != 3:
        raise ValueError(
            "Input image must be height x width x 3 dimensions")
    img_temp = img[31:195] # Choose the important area of
                           # the image
    img_temp = tf.image.rgb_to_grayscale(img_temp)
    img_temp = tf.image.resize(
        img_temp, [self.IM_SIZE, self.IM_SIZE],
        method=tf.image.ResizeMethod.NEAREST_NEIGHBOR
    )
    img_temp = tf.cast(img_temp, tf.float32)
    return img_temp[:, :, 0]

```

```
def combine_images(self, img1, img2):
    if len(img1.shape) == 3 and img1.shape[0] == self.m:
        im = np.append(
            img1[1:,:,:,:], np.expand_dims(img2,0), axis=2)
        return tf.expand_dims(im, 0)
    else:
        im = np.stack([img1]*self.m, axis = 2)
        return tf.expand_dims(im, 0)
#return np.reshape(state, [1, 4])
```

7. Experience replay (the replay function) is a significant feature. It samples a minibatch from the buffer and trains the model on the minibatch:

```
def replay(self, batch_size):
    x_batch, y_batch = [], []
    minibatch = random.sample(
        self.memory,
        min(len(self.memory), batch_size)
    )
    for (state, action, reward,
        next_state, done) in minibatch:
        y_target = self.model_target.predict(
            state, verbose=0)
        y_target[0][action] = reward if done else reward + \
            self.gamma * np.max(
                self.model.predict(next_state,
                                   verbose=0)[0])
        x_batch.append(state[0])
        y_batch.append(y_target[0])

    self.model.fit(
        np.array(x_batch), np.array(y_batch),
        batch_size=len(x_batch), verbose=0
    )
    print("test, length {len(x_batch)}")
    # epsilon = max(epsilon_min, epsilon_decay*epsilon)
    # decrease epsilon
```

8. The training loop runs over multiple episodes, updating the network based on a combination of recent and past experiences, with the aim of achieving an average score above a set threshold, updating the target model weights intermittently to keep them closely tracking the primary model:

```
def train(self):
    scores = deque(maxlen=100)
    avg_scores = []
```

```

for e in range(EPOCHS):
    state, info = self.env.reset()
    state = self.preprocess_state(state)
    state = self.combine_images(state, state)
    done = False
    i = 0
    while not done:
        action = self.choose_action(state, self.epsilon)
        next_state, reward, truncated, terminated,
        _ = self.env.step(action)
        done = terminated or truncated
        next_state = self.preprocess_state(next_state)
        next_state = self.combine_images(
            next_state, state)
        #print(next_state.shape)
        self.remember(state, action, reward,
                      next_state, done)
        state = next_state
        self.epsilon = max(
            self.epsilon_min,
            self.epsilon_decay*self.epsilon
        ) # decrease epsilon
        i += reward

    scores.append(i)
    mean_score = np.mean(scores)
    avg_scores.append(mean_score)
    if mean_score >= THRESHOLD:
        print('Solved after {} trials'.format(e))
        return avg_scores
    if e % 10 == 0:
        print('[Episode {}] - Average Score:
              {}'.format(e, mean_score))
        self.model_target.set_weights(
            self.model.get_weights())
        self.replay(self.batch_size)

    print('Did not solve after {} episodes'.format(e))
    return avg_scores

```

9. Now we initialize an instance of the DQN class for the BreakoutDeterministic-v4 environment:

```

env_string = 'BreakoutDeterministic-v4'
agent = DQN(env_string)

```

10. Next, we display summaries of the neural network architectures for both the primary model (`agent.model`) and the target model (`agent.model_target`) within the DQN agent. Since they are clones, they are exactly the same in architecture:

```
agent.model.summary()
agent.model_target.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 20, 20, 32)	8,224
conv2d_4 (Conv2D)	(None, 9, 9, 64)	32,832
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 64)	0
conv2d_5 (Conv2D)	(None, 2, 2, 64)	36,928
max_pooling2d_3 (MaxPooling2D)	(None, 1, 1, 64)	0
flatten_1 (Flatten)	(None, 64)	0
dense_2 (Dense)	(None, 256)	16,640
dense_3 (Dense)	(None, 4)	1,028

Total params: 95,652 (373.64 KB)
 Trainable params: 95,652 (373.64 KB)
 Non-trainable params: 0 (0.00 B)

Figure 6.8: DQN model architecture

11. Now, let us train. Please remember that training an RL agent can take millions of episodes:

```
scores = agent.train()
```

Here is the reward plot of our agent.

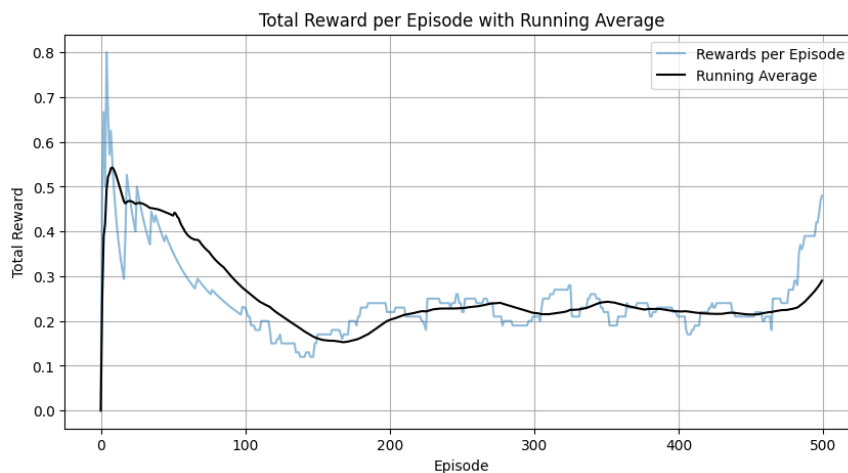


Figure 6.9: Reward plot of DQN agent learning to play Atari (Breakout)

After an intensive training regimen of five hundred episodes, we're beginning to see the fruits of our labor: the agent's average reward is steadily climbing. This process, executed on an A100 GPU, unfolded over the span of twelve hours. However, for the agent to truly excel and master the game, a significant increase in the number of episodes, well into the thousands, is necessary. Moreover, fine-tuning hyperparameters such as the discount factor gamma, the learning rate, and epsilon, as well as exploring different model architectures, are steps that could further enhance performance. RL is a resource-intensive endeavor, but with continued effort and computational power, remarkable achievements are within reach.

Double DQN

Now, recall that we're using a max operator to both select an action and evaluate an action. This can result in overestimated values for an action that may not be an ideal one. We can take care of this problem by decoupling the selection from the evaluation. Double DQN addresses this by using two separate Q-networks with different weights. One network is used to select the best action using the epsilon-greedy policy, while the other evaluates the action to calculate the target Q-value. This decoupling helps prevent the overestimation bias inherent in the original DQN approach, leading to more accurate value estimates and improved policy performance.

To make it clearer, let's first see the case of the DQN. The action with maximum Q-value is selected; let W be the weight of the DQN. Then what we're doing is as follows:

$$Q_{target} = R + \gamma \max_a Q^W(s_{t+1}, \arg\max_a Q^W(s_t))$$

The superscript W tells us the weights used to approximate the Q-value. In double DQN, the equation changes to the following:

$$Q_{target} = R + \gamma \max_a Q^W(s_{t+1}, \arg\max_a Q^{W'}(s_t))$$

Note the change: now, the action is chosen using the Q-network with the weights W , and the max Q-value is predicted using a Q-network with weights W' . This reduces the overestimation and helps us to train the agent quickly and more reliably. You can access the Deep RL with Double Q-Learning paper at <https://arxiv.org/pdf/1509.06461>.

Dueling DQN

Dueling DQN decouples the Q-function into the value function and advantage function. The value function is the same as discussed earlier; it represents the value of the state independent of action. The advantage function, on the other hand, provides a relative measure of the utility (advantage/goodness) of action a in state s :

$$Q(s, a) = V^\pi(s) + A(s, a)$$

In dueling DQN, the same convolutional is used to extract features. However, in later stages, it's separated into two separate networks, one providing the value and another providing the advantage. Later, the two stages are recombined using an aggregating layer to estimate the Q-value. This ensures that the network produces separate estimates for the value function and the advantage function. The intuition behind this decoupling of value and advantage is that, for many states, it's unnecessary to estimate the value of each action choice. For example, in a car race, if there's no car in front, then the turn left or turn right actions are not required, and so there's no need to estimate the value of these actions on the given state.

This allows it to learn which states are valuable, without having to determine the effect of each action for each state.

At the aggregate layer, the value and advantage are combined such that it's possible to recover both V and A uniquely from a given Q . This is achieved by enforcing that the advantage function estimator has zero advantage at the chosen action:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha) - \max_{a' \in |A|} A(s, a'; \theta, \alpha)$$

Here, θ is the parameter of the common convolutional feature extractor, and α and β are the parameters for the advantage and value estimator network. The dueling DQN was proposed by Google's DeepMind team, too. You can read the complete paper at <https://arxiv.org/pdf/1511.06581>. The authors found that changing the preceding max operator with an average operator increases the stability of the network. The advantage, in this case, changes only as fast as the mean. Hence, in their results, they used the aggregate layer given by the following:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha) - \frac{1}{|A|} \sum_{a'} A(s, a'; \theta, \alpha)$$

The following screenshot shows the basic architecture of a dueling DQN:

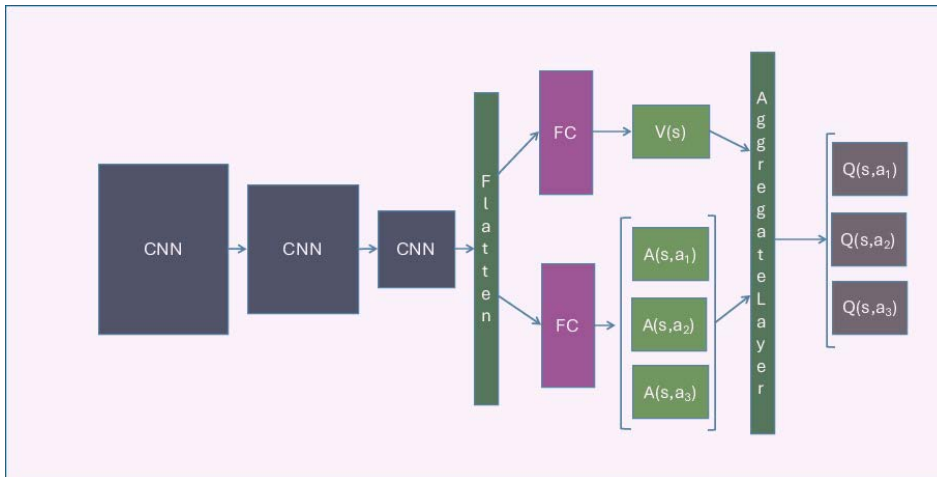


Figure 6.10: The basic architecture of a dueling DQN

The dueling DQN architecture significantly influences the learning process by enabling the network to distinguish between the intrinsic value of being in a given state and the value of taking specific actions in that state. This distinction is particularly advantageous in complex environments where certain actions may have little or no impact. By learning state values separately from the advantages of actions, dueling DQN can make more nuanced decisions, focusing computational resources on understanding which states are beneficial, rather than evaluating the effect of each action in every state. This leads to more efficient policy improvement, especially in cases with a large number of actions. Now that we have covered some value-based algorithms, let us move to policy-based ones by gaining a deeper understanding of policy gradients.

Policy Gradients

In the Q-learning-based methods, we generated a policy after estimating a value or Q- function. In policy-based methods, such as the policy gradient, we approximate the policy directly.

Continuing as earlier, here, we use a neural network to approximate the policy. In the simplest form, the neural network learns a policy for selecting the actions that maximize the rewards by adjusting its weights using the steepest gradient ascent. That's where the name *policy gradients* comes from.

In policy gradients, the policy is represented by a neural network whose input is a representation of states and whose output is action selection probabilities. The weights of this network are the policy parameters that we need to learn. The natural question arises: how should we update the weights of this network? Since our goal is to maximize rewards, it makes sense that our network tries to maximize the expected rewards per episode:

$$J(\theta) = E[R|\pi(a|s, \theta)]$$

Here, we've taken a parametrized stochastic policy π —that is, the policy determines the probability of choosing an action a given state s , and the neural network parameters are θ . R represents the sum of all of the rewards in an episode. The network parameters are then updated using gradient ascent:

$$\theta_{t+1} = \theta_t + \eta \nabla_{\theta} J(\theta) |_{\theta=\theta_t}$$

Here, η is the learning rate. Using the policy gradient theorem, we get the following:

$$\nabla_{\theta} J(\theta) \approx E \left[\sum_{t=0}^{T-1} R_t \sum_{i=0}^{T-1} \nabla_{\theta} \log \pi(a_i | s_i, \theta) \right]$$

Hence, instead of maximizing the expected return, we can use the function as log-loss (expected action and predicted action as labels and logits respectively) and the discounted reward as the weight to train the network. For more stability, it has been found that adding a baseline helps in variance reduction. The most common form of the baseline is the sum of the discounted rewards, resulting in the following:

$$\nabla_{\theta} J(\theta) \approx E \left[\sum_{t=0}^{T-1} \nabla_{\theta} \log \pi(a_t | s_t, \theta) \left(\sum_{t'=t}^{T-1} \gamma^{t'-t} R^{t'} - b(s_t) \right) \right]$$

The baseline $b(st)$ is as follows:

$$b(s_t) \approx E[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \dots + \gamma^{T-1} R_{T-1}]$$

Here, γ is the discount factor.

Why policy gradients?

Well, first of all, policy gradients, like other policy-based methods, directly estimate the optimal policy, without any need to store additional data (experience replay buffer). Hence, they're simple to implement. Secondly, we can train them to learn true stochastic policies. Finally, they're well suited for continuous action space.

DDPG for continuous action space

DDPG is a cutting-edge algorithm in RL that's adept at handling environments with **continuous action spaces**—a notable advantage over algorithms such as DQN, which are limited to discrete action spaces. As a policy gradient method, DDPG directly learns the optimal policy while maintaining a separate, slowly updating target network to stabilize training. It smartly combines ideas from DQN and policy gradients to work efficiently in both high-dimensional state and action spaces. To put DDPG into action, we'll use the `BipedalWalker` environment, which presents a complex, continuous control challenge perfect for showcasing DDPG's capabilities.

We will be using Stable Baselines to implement the RL agent this time. Stable Baselines is a collection of high-quality implementations of RL algorithms in Python, offering a simplified interface to rapidly prototype and benchmark models. It's designed for ease of use with pre-configured models and environments, facilitating a more user-friendly experience for both newcomers and experienced practitioners. What sets Stable Baselines apart is its focus on performance; the library is optimized to run faster using parallelization techniques, thus significantly reducing training time. This makes it an ideal tool for experimenting with various algorithms and hyperparameters efficiently. For more information, refer to the Stable Baselines documentation at <https://stable-baselines.readthedocs.io/en/master/index.html>:

1. We will start, as usual, by importing the modules we need. Besides `Gymnasium`, we will be using Stable Baselines as well:

```
import gymnasium as gym
import numpy as np

from stable_baselines3 import DDPG
from stable_baselines3.common.noise import (
    NormalActionNoise, OrnsteinUhlenbeckActionNoise)
from stable_baselines3.common.evaluation import evaluate_policy
```


2. Next, we define our environment. We are using the `BipedalWalker-v3` environment. It is a simple four-joint walking robot. The idea is to control the motor speeds for the four joints at both hips and knees:

```
env_id = "BipedalWalker-v3"
env = gym.make(env_id, render_mode="rgb_array")
```

3. To improve exploration by the agent in the environment, we introduce Gaussian (normal) noise to the actions taken by the policy. The noise has a mean of zero and a standard deviation of 0.1, scaled to the number of actions. This noise ensures that the agent does not always select the exact same action every time, allowing it to explore more diverse strategies. We also initialize the DDPG model for our environment, the specified noise model, and a **Multi-Layer Perceptron (MLP)** policy:

```
n_actions = env.action_space.shape[-1]
action_noise = NormalActionNoise(mean=np.zeros(n_actions),
                                  sigma=0.1 * np.ones(n_actions))
model = DDPG("MlpPolicy", env,
             action_noise=action_noise, verbose=1)
```

4. Before training the model, let us evaluate a random agent:

```
mean_reward, std_reward = evaluate_policy(
    model, env, n_eval_episodes=100)
```

The mean reward of the random agent is -92.54 ± 0.16 .

5. Now, we go for an exhaustive training of $1e6$ time steps. As you will observe, the Stable Baselines optimizations ensure that the model is trained in a few hours:

```
model.learn(total_timesteps=1000000, log_interval=100)
```

6. By the end of the training, `ep_rew_mean`- the average reward of an episode was 80.

You can see the video of the trained agent at https://www.youtube.com/watch?v=RF_bGZUV4as.

Observe how our bipedal robot now walks almost flawlessly. To truly appreciate its progress, compare this with the awkward stumbling of the untrained bipedal robot—it's a remarkable transformation!

The actor-critic algorithm

In the policy gradient method, we introduced the baseline to reduce variance. Even so, both action and baseline were changing simultaneously (look closely: the variance is the expected sum of rewards, or in other words, the goodness of the state or its value function).

Wouldn't it be better to separate the policy evaluation from the value evaluation? That's the idea behind the actor-critic method. It consists of two neural networks. One approximates the policy, called the actor network, and the other approximates the value, called the critic network. We alternate between a policy evaluation and a policy improvement step, resulting in more stable learning. The critic uses the state and action values to estimate a value function, which is then used to update the actor's policy network parameters so that the overall performance improves. The following diagram shows the basic architecture of the actor-critic network:

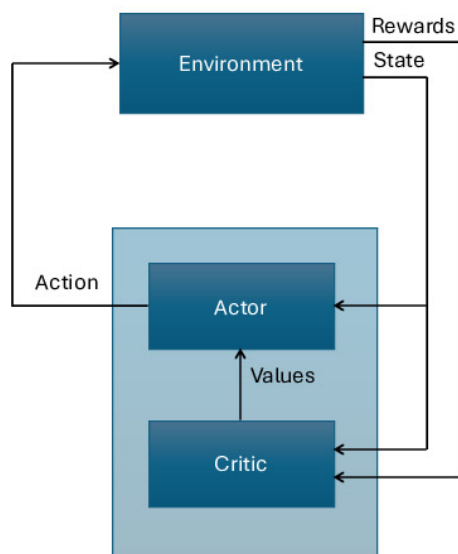


Figure 6.11: Actor-critic architecture

In this section on policy gradients, we've delved deep into sophisticated algorithms such as DDPG and A3C, exploring their capabilities and applications. By implementing these advanced techniques, we can gain a richer understanding of how RL can be adapted to a wide array of tasks, paving the way for further innovations in the field.

Summary

In this chapter, we learned about RL and how it's different from supervised and unsupervised learning. The emphasis of this chapter was on DRL, where deep neural networks are used to approximate the policy function, the value function, or even both. This chapter introduced OpenAI Gym, a library that provides a large number of environments to train RL agents. We learned about value-based methods such as Q-learning and used it to train an agent to pick up and drop passengers off in a taxi. We also used a DQN to train an agent to play an Atari game. This chapter then moved on to policy-based methods, specifically policy gradients. We covered the intuition behind policy gradients and used the algorithm to train a bipedal robot to walk using DDPG.

In the next chapter, we'll explore generative models and learn the secrets behind generative adversarial networks.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/ds>



Generative Models for IoT

Machine learning (ML) and **artificial intelligence (AI)** have profoundly impacted numerous fields, from agriculture and music to healthcare and defense. The success of AI/ML, beyond the advances in computational power, hinges on the generation and analysis of vast amounts of data. Traditionally, much of this data has been unlabeled, making the understanding of its inherent distribution a crucial ML task. This is where generative models come into play.

In recent years, the landscape of generative AI has evolved significantly, particularly in the context of the **internet of things (IoT)**. Generative AI, which includes models such as **generative adversarial networks (GANs)**, **variational autoencoders (VAEs)**, and more recently, Transformer-based models, has advanced to address the unique challenges and opportunities presented by IoT ecosystems.

As we explore the capabilities of generative AI within IoT, we'll delve into the latest advancements, methodologies, and practical applications that are transforming how we interact with and benefit from connected devices. After reading this chapter, you'll have covered the following:

- Introduction to Generative Models
- Understanding VAEs
- GANs
- KerasNLP

Introduction to Generative Models

Generative models are an exciting new branch of **deep learning (DL)** models that learn through unsupervised learning. The main idea is to generate new samples having the same distribution as the given training data; for example, a network trained on handwritten digits can create new digits that aren't in the dataset but are similar to them. Formally, we can say that if the training data follows the distribution $P_{data}(x)$, then the goal of generative models is to estimate the probability density function $P_{model}(x)$, which is similar to $P_{data}(x)$.

Generative models can be classified into two types:

- **Explicit generative models:** Here, the probability density function, $P_{model}(x)$, is explicitly defined and solved. The density function may be tractable as in the case of PixelRNN/CNN, or an approximation of the density function as in the case of VAE.
- **Implicit generative models:** In these, the network learns to generate a sample from $P_{model}(x)$, without explicitly defining it. GANs are an example of this type of generative model.

In this chapter, we will explore VAEs, an explicit generative model, and GANs, an implicit generative model. Generative models are instrumental in creating realistic samples and performing super-resolution, colorization, and more. They are also valuable for time series data, enabling simulations and planning. Additionally, these models help us understand the latent representations of data.

Recent advancements have extended the applications of generative models into the realm of IoT. By leveraging the strengths of AI and IoT, generative models enhance data augmentation, simulate diverse scenarios, and optimize resource allocation in real time. This integration, often referred to as **AIoT**, facilitates smarter, more efficient systems.

Generative models can learn the underlying distribution of data and generate new data samples that are statistically similar to the training data. This capability is particularly useful in the following IoT scenarios:

- **Anomaly detection:** In a smart factory, sensors continuously monitor the condition of machinery. A VAE can be trained on normal operational data to learn the typical behavior of the machines. When the VAE encounters data that deviates significantly from the learned pattern, it flags it as an anomaly. This helps in early detection of potential faults, reducing downtime and maintenance costs.
- **Predictive maintenance:** In the context of fleet management, vehicles are equipped with various sensors to monitor engine performance, fuel efficiency, and other critical parameters. Generative models can analyze this data to predict when a vehicle is likely to require maintenance. For instance, a GAN can simulate different failure scenarios and help in scheduling maintenance before a breakdown occurs.
- **Data augmentation:** In healthcare IoT applications, collecting a large amount of labeled data for training ML models can be challenging. **Conditional generative adversarial networks (cGANs)** can generate synthetic medical sensor data, such as heart rate or glucose level readings, to augment the training dataset. This improves the accuracy and robustness of diagnostic models without the need for extensive real-life data collection.
- **Realistic simulations:** Creating realistic driving scenarios is crucial for testing and validation when developing autonomous vehicles. Generative models such as **style-based GAN (StyleGAN)** can generate high-fidelity images of various driving environments, weather conditions, and traffic situations. These synthetic environments enable extensive testing of autonomous driving algorithms in a safe and controlled manner before deployment in the real world.

By leveraging the capabilities of generative models, IoT applications can become more intelligent, reliable, and efficient. This chapter explores several advanced generative models, including VAEs, GANs, and their variants such as **super-resolution generative adversarial networks (SRGANs)**, StyleGAN, and AnimeGAN. We will also discuss the implementation of a conditional GAN on the MNIST dataset. By understanding these models and their capabilities, we can unlock new possibilities for innovation in IoT.

Furthermore, the advent of transformer-based models has revolutionized personalization within IoT systems. These models, originally developed for **natural language processing (NLP)**, can be adapted to provide personalized recommendations and interactions in IoT environments. They enable more intuitive and context-aware services by understanding and predicting user behaviors and preferences.

As we delve into the capabilities of generative models within the IoT landscape, we will uncover the latest methodologies, practical applications, and future directions. From smart homes and cities to industrial automation and healthcare, generative AI is set to revolutionize how we interact with and benefit from connected devices, making them more intelligent, responsive, and adaptive.

So, sit tight as we start with VAEs.

Understanding VAEs

From *Chapter 4*, you should be familiar with autoencoders and their functions. VAEs are a type of autoencoder; when using one as a generator, we retain the (trained) decoder part, which can be used by feeding random latent features, z , to generate data similar to the training data. Now, if you remember, in autoencoders, the encoder results in the generation of low-dimensional features, z :

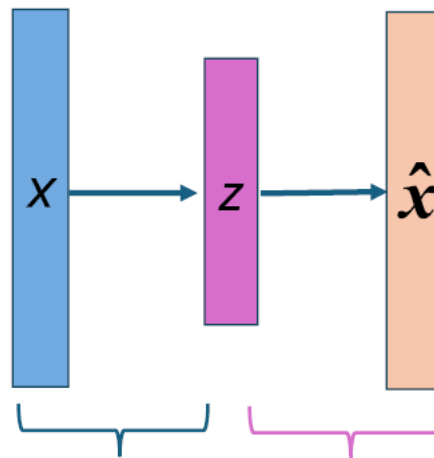


Figure 7.1: Architecture of autoencoders

The VAEs are concerned with finding the likelihood function $p(x)$ from the latent features, z :

$$p_{\theta}(x) = \int p_{\theta}(z) p_{\theta}(x|z) dz$$

This is an intractable density function, and it isn't possible to directly optimize it; instead, we obtain a lower bound by using a simple Gaussian prior $p(z)$ and making both the encoder and decoder networks probabilistic.

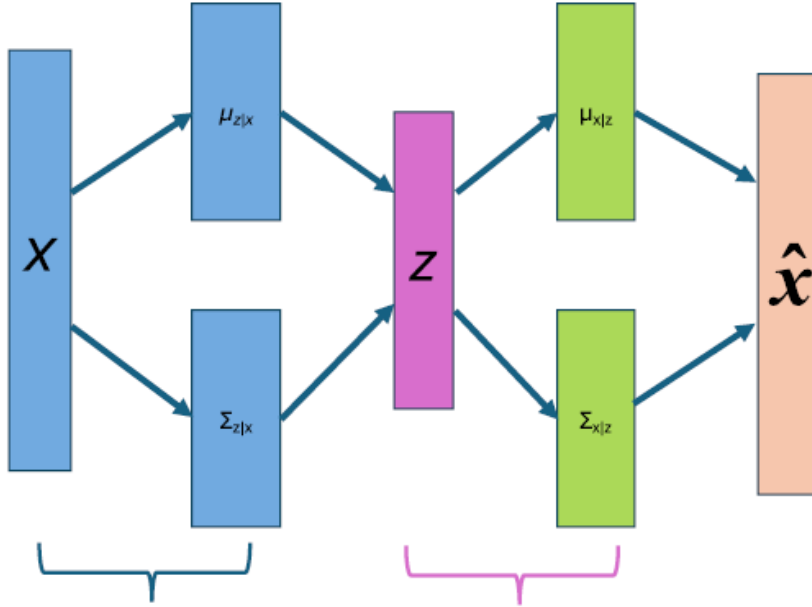


Figure 7.2: Architecture of a VAE

This allows us to define a tractable **lower bound** on the log likelihood, given by the following:

$$\log p_{\theta}(x^i) \geq \mathcal{L}(x^i, \theta, \phi) = E_z[\log p_{\theta}(x^i|z)] - D_{KL}(q_{\phi}(z|x^i) || p_{\theta}(z|x^i))$$

In the preceding, θ represents the decoder network parameters and ϕ the encoder network parameters. The network is trained by maximizing this lower bound:

$$\theta^*, \phi^* = \operatorname{argmax}_{\theta, \phi} \sum_{i=1}^N \mathcal{L}(x^i, \theta, \phi)$$

The first term in the lower bound is responsible for the reconstruction of the input data, also called **reconstruction loss**, and the second term, called the **regularization term**, ensures that the approximate posterior distribution is close to the prior. The regularization term ensures that the latent space distribution is smooth and continuous, which helps in generating new data points.

VAEs grasp the underlying structure, which enables them to generate new, similar data from this learned distribution. This makes VAEs particularly valuable for several applications:

- **Generating new images:** After training, the decoder alone is used to generate new images by sampling random points from the learned latent space distribution and mapping them back to the original data space.
- **Filling in missing data:** VAEs can reconstruct missing parts by encoding incomplete data into the latent space and then using the decoder to predict and fill in the gaps.
- **Creating realistic simulations:** By sampling from the latent space, VAEs can create realistic variations of the input data, useful for generating synthetic datasets for modeling and analysis.
- **Anomaly detection:** VAEs learn the normal data distribution during training. When new data is passed through the VAE, significant deviations between the reconstructed data and the input can indicate anomalies, making this method effective for detecting outliers in datasets, such as identifying fraud or equipment failures.

Let us now see VAEs in action. We will use VAEs for anomaly detection, a common and crucial task in preventive maintenance to identify potential equipment failures early.

Anomaly detection using VAEs

Anomaly detection involves identifying data points that deviate significantly from the norm, indicating potential issues or unusual patterns. In the context of IoT, anomaly detection is crucial for maintaining system health and reliability. IoT devices generate vast amounts of real-time data, and anomalies can signal malfunctions, security breaches, or other critical problems. By detecting these anomalies early, organizations can perform predictive maintenance, preventing equipment failures and minimizing downtime. This proactive approach ensures smoother operations, enhances safety, and reduces maintenance costs, making anomaly detection an essential component of IoT systems.

In this section, we are using VAE to perform anomaly detection on the MNIST dataset, a widely used dataset of handwritten digits. The idea is to train a VAE on normal digit images (digits 0-5 and 7-9), and then use it to identify anomalies, specifically the digit 6. Follow these steps:

1. First, we need to import all the necessary libraries. TensorFlow and Keras are essential for building and training our VAE model. We also import TensorFlow Probability for handling probabilistic aspects, NumPy for numerical operations, and Matplotlib for visualization. To ensure smooth training on GPUs, we enable memory growth to avoid out-of-memory errors:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import tensorflow_probability as tfp
import numpy as np
import matplotlib.pyplot as plt
```



```

from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score

# Enable memory growth for GPUs to avoid out-of-memory errors
physical_devices = tf.config.list_physical_devices('GPU')
for device in physical_devices:
    tf.config.experimental.set_memory_growth(device, True)

```

2. We begin by loading the MNIST dataset, which contains 60,000 training images and 10,000 test images of handwritten digits. Our focus is on identifying the digit 6 as an anomaly. Hence, we filter out 6 from the training and test datasets to use them later as anomalous data. The remaining data is normalized to have values between 0 and 1, and each image is flattened from a 28x28 matrix to a vector of 784 elements:

```

(x_train_, y_train_), (x_test_, y_test_) = keras.datasets.mnist.
load_data()

# Filter out digit '6' from the training and test sets
train_mask = y_train_ != 6
test_mask = y_test_ != 6

x_train, y_train = x_train_[train_mask], y_train_[train_mask]
x_test, y_test = x_test_[test_mask], y_test_[test_mask]

x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

x_train = x_train.reshape(-1, 28 * 28)
x_test = x_test.reshape(-1, 28 * 28)

```

3. The encoder is the first component of our VAE, responsible for compressing the input data into a lower-dimensional latent space. It consists of multiple dense layers with ReLU activation functions. The encoder outputs two vectors: the **mean** and the **logarithm of the variance** of the latent space. These vectors represent the distribution of the encoded data points in the latent space:

```

input_shape = x_train.shape[1]
latent_dim = 2 # Dimension of the latent space

# Encoder
encoder_inputs = keras.Input(shape=(input_shape,))
x = layers.Dense(512, activation='relu')(encoder_inputs)
x = layers.Dense(256, activation='relu')(x)
x = layers.Dense(128, activation='relu')(x)

```

```
z_mean = layers.Dense(latent_dim, name='z_mean')(x)
z_log_var = layers.Dense(latent_dim, name='z_log_var')(x)
```

4. To sample points from the latent space, we use a reparameterization trick. This involves defining a custom sampling function that generates random points from the latent space distribution. This trick is crucial for backpropagation, allowing gradients to flow through the stochastic part of the model during training:

```
# Sampling function
def sampling(z_mean, z_log_var):
    epsilon = tf.random.normal(shape=tf.shape(z_mean))
    return z_mean + tf.exp(0.5 * z_log_var) * epsilon

z = layers.Lambda(lambda x: sampling(x[0], x[1]))([z_mean, z_log_var])
```

5. Combining the encoder layers defined in step 4, and latent space, z , we get encoder:

```
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var, z],
name='encoder')
```

6. The decoder is the second component of our VAE, tasked with reconstructing the original input data from the sampled latent points. Similar to the encoder, the decoder consists of multiple dense layers with ReLU activation functions. The final layer of the decoder reconstructs the input data, aiming to match the original data as closely as possible:

```
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(128, activation='relu')(latent_inputs)
x = layers.Dense(256, activation='relu')(x)
x = layers.Dense(512, activation='relu')(x)
decoder_outputs = layers.Dense(
    input_shape, activation='sigmoid')(x)

decoder = keras.Model(latent_inputs, decoder_outputs,
name='decoder')
```

7. With the encoder and decoder defined, we combine them to create the VAE model. We also define the VAE loss function, which is a combination of the reconstruction loss (how well the reconstructed data matches the original data) and the KL divergence loss (how well the latent space distribution matches a standard normal distribution). The model is then compiled using the **Adam** optimizer:

```
outputs = decoder(encoder(encoder_inputs)[2])
vae = keras.Model(encoder_inputs, outputs, name='vae')
```

```
# Loss Function
reconstruction_loss = tf.reduce_mean(
    tf.keras.losses.binary_crossentropy(
        encoder_inputs, outputs))
reconstruction_loss *= input_shape

kl_loss = 1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var)
kl_loss = tf.reduce_mean(kl_loss)
kl_loss *= -0.5

vae_loss = reconstruction_loss + kl_loss
vae.add_loss(vae_loss)
vae.compile(optimizer=keras.optimizers.Adam())
```

8. We proceed to train the VAE model on the training data for 50 epochs with a batch size of 128. During training, the model learns to encode and decode the normal digit images. Validation data is also provided to monitor the model's performance and prevent overfitting:

```
vae.fit(x_train, x_train, epochs=50, batch_size=128, validation_
data=(x_test, x_test))
```

9. Once the model is trained, we use it to identify anomalies. We select digit 6 images as anomalies and preprocess them similarly to the training data. The VAE model reconstructs both normal and anomaly data. We then compute the reconstruction loss for both types of data. A threshold is set based on the 95th percentile of the reconstruction loss of normal data, which helps in distinguishing anomalies:

```
(x_anomaly, _) = x_test_[y_test_ == 6], y_test_[y_test_ == 6]

x_anomaly = x_anomaly.astype('float32') / 255.0
x_anomaly = x_anomaly.reshape(-1, 28 * 28)

# Compute reconstruction loss for normal and anomaly data
reconstructed = vae.predict(x_test)
reconstruction_loss_normal = np.mean(np.abs(
    reconstructed - x_test), axis=1)

reconstructed_anomaly = vae.predict(x_anomaly)
reconstruction_loss_anomaly = np.mean(np.abs(
    reconstructed_anomaly - x_anomaly), axis=1)
```

```
# Threshold for anomaly detection
threshold = np.percentile(reconstruction_loss_normal, 95)
```

10. To evaluate our anomaly detection model, we create labels for normal and anomaly data and make predictions based on whether the reconstruction loss exceeds the threshold. We calculate evaluation metrics such as accuracy, precision, recall, and F1 score to measure the model's performance and print these values:

```
# Predict anomalies
y_pred_normal = reconstruction_loss_normal > threshold
y_pred_anomaly = reconstruction_loss_anomaly > threshold

# Create labels for evaluation
y_true_normal = np.zeros_like(y_pred_normal)
y_true_anomaly = np.ones_like(y_pred_anomaly)

y_true = np.concatenate([y_true_normal, y_true_anomaly])
y_pred = np.concatenate([y_pred_normal, y_pred_anomaly])

# Calculate scores
accuracy = accuracy_score(y_true, y_pred)
precision = precision_score(y_true, y_pred)
recall = recall_score(y_true, y_pred)
f1 = f1_score(y_true, y_pred)

print(f'Accuracy: {accuracy}')
print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'F1 Score: {f1}')
```

Here is the result:

```
283/283 [=====] - 1s 2ms/step
30/30 [=====] - 0s 2ms/step
Accuracy: 0.9185
Precision: 0.5681601525262154
Recall: 0.6221294363256785
F1 Score: 0.5939212755356253
```

11. Finally, we visualize the results by plotting a histogram of reconstruction losses for normal and anomaly data. The threshold is marked on the plot, providing a clear visual representation of how well the model distinguishes between normal and anomalous data.

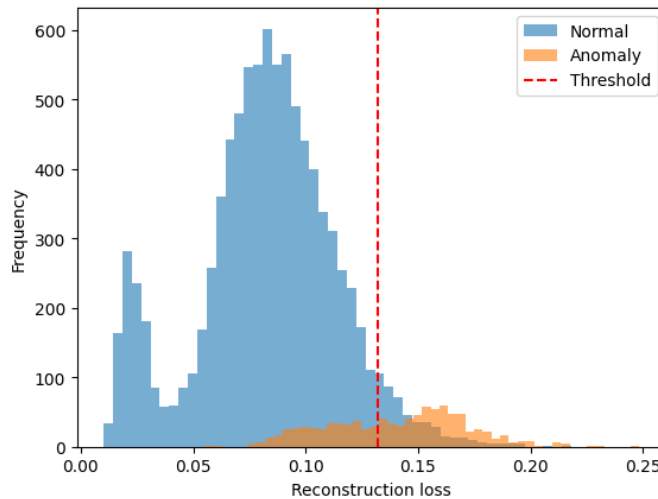


Figure 7.3: Histogram for reconstruction losses for normal and anomaly data

This section showcases the power of VAEs in identifying anomalies, highlighting their versatility and effectiveness. Anomaly detection is crucial in numerous real-life applications, such as fraud detection, where unusual patterns can indicate fraudulent activities, medical diagnosis, where early detection of anomalies can save lives, and predictive maintenance, where identifying irregularities can prevent equipment failures and reduce downtime. VAEs offer a robust tool for enhancing reliability and safety across these diverse domains. In the next section, we will cover GANs, another generative model.

GANs

GANs have become a cornerstone of modern ML, described by Yann LeCun as “*the most interesting idea in the last 10 years.*” These innovative networks have made significant strides across multiple fields. In computer vision, GANs excel in image and video generation, image translation, and super-resolution. In NLP, they’re pivotal for text generation and machine translation. Healthcare leverages GANs for medical image synthesis and data augmentation, enhancing diagnostic models. Cybersecurity and finance also benefit from GANs by generating synthetic data for training models and detecting anomalies. A 2022 review paper (<https://www.sciencedirect.com/science/article/pii/S2667318522000150>) highlights their extensive use in medical imaging, including segmentation and denoising, underscoring their transformative impact. With their widespread applications, GANs have become a focal point in AI/ML research, consistently showcasing their potential to revolutionize various industries.

GANs were proposed by Ian J. Goodfellow and Yoshua Bengio in the paper *Generative Adversarial Networks* in 2014 (<https://arxiv.org/abs/1406.2661>). They're inspired by the two-player game scenario. Like the two players of the game, in GANs, two networks—one called the **discriminative network** and the other the **generative network**—compete with each other. The generative network tries to generate data similar to the input data, and the discriminator network has to identify whether the data it's seeing is real or fake (that is, generated by a generator). Every time the discriminator finds a difference between the distribution of true input and fake data, the generator adjusts its weights to reduce the difference. To summarize, the discriminative network tries to learn the boundary between counterfeit and real data, and the generative network tries to learn the distribution of training data. As the training ends, the generator learns to produce images exactly like the input data distribution, and the discriminator can no longer differentiate the two. The general architecture of a GAN is as follows:

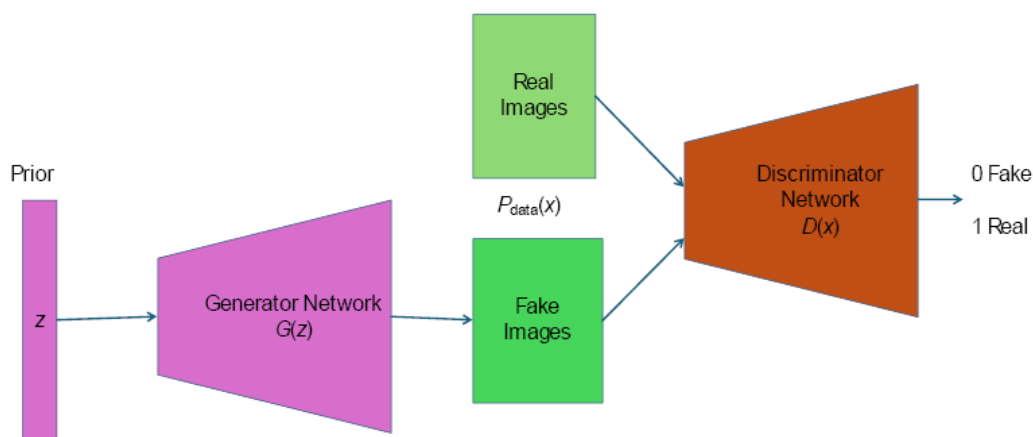


Figure 7.4: Architecture of GANs

Let's now delve deep into how GANs learn. Both the discriminator and generator take turns to learn. The learning can be divided into two steps:

1. Here the *Discriminator*, $D(x)$, learns. The *Generator*, $G(z)$, is used to generate *Fake Images* from random noise, z (which follows some *Prior* distribution, $P(z)$). The *Fake Images* from the *Generator* and the *Real Images* from the training dataset are both fed to the *Discriminator*, and it performs supervised learning, trying to separate fake from real. If $P_{data}(x)$ is the training dataset distribution, then the *Discriminator Network* tries to maximize its objective so that $D(x)$ is close to 1 when the input data is real, and close to 0 when the input data is fake. This can be achieved by performing the gradient ascent on the following objective function:

$$\theta_d^{max} \left[E_{x \sim P_{data}} \log D_{\theta_d}(x) + E_{z \sim P(z)} \log (1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

2. In the next step, the *Generator Network* learns. Its goal is to fool the *Discriminator Network* into thinking that generated $G(z)$ is real, that is, force $D(G(z))$ close to 1. To achieve this, the *Generator Network* minimizes the objective:

$$\theta_d^{min} \left[E_{z \sim P(z)} \log \left(1 - D_{\theta_d} \left(G_{\theta_g}(z) \right) \right) \right]$$

The two equations can be combined into a single equation represented as the following objective function:

$$\min_G \max_D V(D, G) = E_{x \sim P_{data}} \log D(x) + E_{z \sim P(z)} \log(1 - D(G(z)))$$

The two steps are repeated sequentially. Once the training ends, the discriminator is no longer able to discriminate between real and fake data and the generator becomes a pro at creating data very similar to the training data. However, training GANs isn't always straightforward due to instability issues. Recent advances have significantly improved GAN training stability. Techniques such as improved loss functions, regularization methods, and architectural changes have been proposed, leading to more reliable and efficient training processes. These developments have made GANs more robust and easier to train, opening even more applications.

Variants of GANs

GANs have proven their mettle as a powerhouse for generating realistic data, but the innovation doesn't stop at the original framework. Researchers and engineers have pushed the boundaries, crafting a diverse range of GAN variants, each with unique strengths and applications. Let's explore some of the most noteworthy additions to the GAN family.

cGAN

cGANs extend the original GAN framework by incorporating additional conditions or labels to both the generator and discriminator. This is achieved by feeding the conditioning information (e.g., class labels) along with the noise vector into the generator, and into the discriminator alongside real or generated data. This allows for controlled and targeted data generation. For example, in image generation, cGANs can produce images of specific categories by conditioning on class labels, leading to more diverse and specified outputs.

This conditioning modifies the GAN framework to generate data that adheres to the specified conditions. The objective function for a cGAN is as follows:

$$\min_G \max_D V(D, G) = E_{x, y \sim P_{data}(x, y)} \log D(x|y) + E_{z \sim P(z), y \sim P_{data}(y)} \log(1 - D(G(z|y)))$$

Here, $D(x/y)$ represents the probability that x is real given the condition y , and $G(z/y)$ generates data conditioned on y .

For further reading and practical implementation, refer to the seminal paper by Mirza and Osindero (2014) (<https://arxiv.org/pdf/1411.1784>).

StyleGAN

StyleGAN, developed by NVIDIA, introduces style transfer to GANs, enabling more nuanced and detailed control over the generated images. It uses a two-stage architecture where the input noise vector is first mapped to an intermediate latent space, called W , via a mapping network (eight-layer fully connected network). This intermediate space influences the image synthesis process at multiple scales through **adaptive instance normalization (AdaIN)** layers. This design allows for fine-grained control over style attributes at different levels of abstraction, resulting in highly realistic and detailed images. StyleGAN has led to breakthroughs in high-quality image generation and manipulation, captivating artists and designers alike.

Here is the link to the original StyleGAN paper: https://openaccess.thecvf.com/content_CVPR_2019/papers/Karras_A_Style-Based_Generator_Architecture_for_Generative_Adversarial_Networks_CVPR_2019_paper.pdf. NVIDIA improved upon their original idea with **StyleGAN2** (https://openaccess.thecvf.com/content_CVPR_2020/papers/Karras_Analyzing_and_Improving_the_Image_Quality_of_StyleGAN_CVPR_2020_paper.pdf). By replacing AdaIN with weight demodulation, removing progressive growing, and adding perceptual path length regularization, StyleGAN2 achieves higher image quality, reduces artifacts, and provides smoother latent space interpolations. These enhancements make StyleGAN2 a more robust and reliable tool for generating high-resolution, realistic images.

SRGAN

SRGAN represents a significant advancement in the field of image super-resolution, which involves upscaling low-resolution images to higher resolutions while preserving fine details and minimizing information distortion. Developed by Christian Ledig and colleagues, SRGAN is the first framework capable of inferring photo-realistic natural images for 4x upscaling factors. The generator in SRGAN is based on a deep residual network (**ResNet**) with skip connections, which helps in training deeper networks by mitigating the vanishing gradient problem.

The discriminator is designed to differentiate between real high-resolution images and the super-resolved images generated by the generator. It uses a series of convolutional layers with increasing feature maps and decreasing spatial dimensions, followed by dense layers and a final sigmoid activation function to output the probability of the input being a real image. SRGAN uses the **perceptual loss** function, which is a combination of content loss and adversarial loss. The **content loss** measures the perceptual similarity between the generated image and the ground truth. **Adversarial loss** encourages the generator to produce images that reside on the manifold of natural images.

AnimeGAN

AnimeGAN is a GAN designed to transform real-life photos into anime-style images. AnimeGAN follows the typical GAN structure, consisting of a generator and a discriminator that are trained adversarially. The generator's task is to create anime-style images from real photographs, while the

discriminator evaluates the authenticity of these generated images against real anime images. The unique aspect of AnimeGAN lies in its ability to maintain high fidelity to anime art styles while preserving the essential content of the original photographs.



Figure 7.5: Pictures of the author using AnimeGAN2

It utilizes a combination of adversarial loss, content loss, and style loss. The adversarial loss ensures that the generated images are indistinguishable from real anime images. The content loss preserves the structure and key features of the original photographs. The style loss ensures that the generated images maintain the distinct stylistic elements of anime.

You can access the complete list of different variants of GAN from the GAN Zoo GitHub repository: <https://github.com/hindupuravinash/the-gan-zoo>.

Applications of GANs

GANs have revolutionized the field of AI by enabling the generation of highly realistic data. GANs have found applications across various domains, including image synthesis, healthcare, fashion, and more. In this section, we will explore some of the most impactful applications of GANs:

- **Image generation:** GANs are widely used to generate high-quality, photorealistic images. This application is particularly useful in fields such as entertainment, advertising, and video game development. GANs can create realistic images of faces, objects, and landscapes, which can be used for various purposes, including creating avatars, enhancing visual effects, and generating new content for games and movies.
- **Super-resolution:** SRGANs are used to enhance the resolution of low-quality images. This technique is beneficial in medical imaging, satellite imaging, and video enhancement, where high-resolution images are crucial for accurate analysis and interpretation.
- **Image inpainting:** GANs can fill in missing parts of an image, a process known as **image inpainting**. This application is useful in restoring damaged photographs, removing unwanted objects from images, and enhancing the quality of old or incomplete images.
- **Style transfer:** GANs can transfer the style of one image to another, creating new images that combine the content of the original image with the style of a reference image. This technique is used in artistic applications, such as generating paintings in the style of famous artists, such as Van Gogh or Picasso.

- **Semantic image synthesis:** GANs can generate realistic images from semantic maps or sketches. This application is useful in urban planning, architecture, and interior design, where designers can create realistic visualizations of their concepts.
- **Domain adaptation:** GANs can transform images from one domain to another, such as converting day images to night images, summer scenes to winter scenes, or black-and-white images to color. This application is valuable in autonomous driving, where training data needs to cover various driving conditions.
- **Text-to-image synthesis:** GANs can generate images based on textual descriptions. This application is useful in creating visual content for stories, generating product images from descriptions, and designing new products. For example, a GAN can generate an image of a “red sports car with black stripes” based on the given text description.
- **Medical image synthesis:** GANs can generate synthetic medical images, which are useful for training ML models without compromising patient privacy. These synthetic images can augment existing datasets, improving the performance of diagnostic algorithms.
- **Drug discovery:** GANs assist in generating molecular structures with desired properties, accelerating the drug discovery process. By synthesizing novel molecules, GANs enable pharmaceutical companies to explore a broader chemical space, potentially leading to the development of more effective treatments.
- **Virtual try-on:** GANs enable virtual try-on experiences, allowing customers to visualize themselves wearing various clothing items without physically trying them on. This enhances the online shopping experience, reduces return rates, and boosts customer satisfaction.
- **Fashion design:** GANs can generate new clothing designs, textures, and patterns, streamlining the design process for fashion designers. By training on vast datasets of clothing designs, GANs can create avant-garde collections and help designers quickly prototype and iterate on new ideas.
- **Character and environment generation:** GANs are used to create realistic characters, environments, and props for video games and movies. This application enhances the visual quality of games and films, providing more immersive experiences for users.
- **Voice generation and animation:** GANs can generate realistic voices and animations, adding a new level of realism to virtual experiences. This application is useful in creating lifelike characters for games, movies, and virtual reality environments.
- **Data augmentation:** GANs can generate synthetic data to augment existing datasets, improving the robustness and generalization of ML models. This application is particularly useful in scenarios where labeled data is scarce or expensive to obtain.
- **Anomaly detection:** GANs can identify anomalies or outliers in data, which is useful for detecting fraud, network intrusions, or medical conditions. By learning the distribution of normal data, GANs can effectively flag data points that deviate from the norm.

- **3D object generation:** GANs can generate realistic 3D models from 2D images or sketches. This application is valuable in fields such as architecture, design, and engineering, where accurate 3D models are essential for visualization and prototyping

cGAN in Keras

In this section, we'll dive into cGANs. We will cover the meticulous process of building, training, and evaluating a cGAN using Keras. The code presented here is adapted from the Keras code examples available on the Keras site (<https://keras.io/examples/>):

1. First things first, we start by importing our trusty modules. These include Keras, TensorFlow, NumPy, and a few other sidekicks:

```
import keras
from keras import Sequential, Input, Model
from keras import ops
from keras.layers import (
    InputLayer, Conv2D, Dense, Flatten,
    Reshape, Conv2DTranspose,
    LeakyReLU, GlobalMaxPooling2D
)
from keras.optimizers import Adam
from keras.losses import BinaryCrossentropy
from keras.utils import plot_model
from tensorflow_docs.vis import embed
import tensorflow as tf
import numpy as np
import imageio
```

2. Before diving into the models, we define some critical hyperparameters. These include the following:

```
batch_size = 64
num_channels = 1
num_classes = 10
image_size = 28
latent_dim = 128
generator_in_channels = latent_dim + num_classes
discriminator_in_channels = num_channels + num_classes
```

Batch size determines how many samples are processed before the model's internal parameters are updated. Since we're likely dealing with grayscale images (MNIST digits), we only need 1 channel and 10 classes with a 28x28 image size. `latent_dim` decides the size of the random noise vector fed into the generator, giving it enough capacity to create diverse and complex images.

3. Now, let's get our dataset ready. We load the MNIST dataset, which includes both training and test sets:

```
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.  
load_data()  
all_digits = np.concatenate([x_train, x_test])  
all_labels = np.concatenate([y_train, y_test])
```

4. We then scale the pixel values to the $[0, 1]$ range and reshape the images to add a channel dimension:

```
all_digits = all_digits.astype("float32") / 255.0  
all_digits = np.reshape(all_digits, (-1, 28, 28, 1))
```

5. Next, the labels are one-hot encoded to match `num_classes`:

```
all_labels = keras.utils.to_categorical(all_labels, 10)
```

6. Finally, we create `tf.data.Dataset`, shuffle it, and batch it. Shuffling and batching the dataset ensures the model trains on varied data and improves training efficiency:

```
dataset = tf.data.Dataset.from_tensor_slices((  
    all_digits, all_labels))  
dataset = dataset.shuffle(buffer_size=1024).batch(batch_size)
```

7. Next up, we build our generator model using Keras' Sequential API. We use some dense layers to bulk up the noise vector, followed by batch normalization to keep everything stable. LeakyReLU activation functions add a bit of edge, and upsampling layers ensure our generated images grow in size, eventually reaching the desired dimensions:

```
generator = Sequential(  
    [  
        InputLayer((generator_in_channels,)),  
        Dense(7 * 7 * generator_in_channels),  
        LeakyReLU(negative_slope=0.2),  
        Reshape((7, 7, generator_in_channels)),  
        Conv2DTranspose(  
            128, (4, 4), strides=(2, 2), padding="same"),  
        LeakyReLU(negative_slope=0.2),  
        Conv2DTranspose(  
            128, (4, 4), strides=(2, 2), padding="same"),  
        LeakyReLU(negative_slope=0.2),  
        Conv2D(1, (7, 7), padding="same", activation="sigmoid"),  
    ],  
    name="generator",  
)
```

8. Now, let's talk about the discriminator. Again, we lean on Keras' Sequential API. This time, our layers include convolutional ones for feature extraction, more batch normalization for stability, LeakyReLU for non-linearity, and dropout layers to keep overfitting at bay:

```
discriminator = Sequential(
    [
        InputLayer((28, 28, discriminator_in_channels)),
        Conv2D(64, (3, 3), strides=(2, 2), padding="same"),
        LeakyReLU(negative_slope=0.2),
        Conv2D(128, (3, 3), strides=(2, 2), padding="same"),
        LeakyReLU(negative_slope=0.2),
        GlobalMaxPooling2D(),
        Dense(1),
    ],
    name="discriminator",
)
```

9. Now, we get to the heart of it: the cGAN. We fuse our generator and discriminator models. The generator receives a noise vector and a conditional label, creating images accordingly. Meanwhile, the discriminator gets both real and generated images, conditioned on the same label, and tries to tell them apart. In the `__init__` function, we initialize the discriminator, generator, and latent dimension. A seed generator ensures reproducibility, crucial for consistent results, and loss trackers for both the generator and discriminator to help monitor training progress:

```
class cGAN(keras.Model):
    def __init__(self, discriminator, generator, latent_dim):
        super().__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim
        self.seed_generator = keras.random.SeedGenerator(27)
        self.gen_loss_tracker = keras.metrics.Mean(
            name="generator_loss")
        self.disc_loss_tracker = keras.metrics.Mean(
            name="discriminator_loss")
```

10. We then add a `metrics` property to return these trackers:

```
@property
def metrics(self):
    return [self.gen_loss_tracker, self.disc_loss_tracker]
```

11. The `compile` method is overridden to set optimizers and the loss function, as it lets us specify separate optimizers for the generator and discriminator, as well as the loss function used for both:

```
def compile(self, d_optimizer, g_optimizer, loss_fn):
    super().compile()
    self.d_optimizer = d_optimizer
    self.g_optimizer = g_optimizer
    self.loss_fn = loss_fn
```

12. In the `train_step` method, unpack data and images and reshape labels to match image dimensions. We create random latent vectors and combine them with labels to form the input for the generator. We also concatenate labels with image conditions for both real and fake images. Finally, create labels for real and fake images and compute discriminator loss:

```
def train_step(self, data):
    # Unpack the data.
    real_images, one_hot_labels = data

    image_one_hot_labels = one_hot_labels[:, :, None, None]
    image_one_hot_labels = ops.repeat(
        image_one_hot_labels, repeats=[
            image_size * image_size
        ]
    )
    image_one_hot_labels = ops.reshape(
        image_one_hot_labels,
        (-1, image_size, image_size, num_classes)
    )

    batch_size = ops.shape(real_images)[0]
    random_latent_vectors = keras.random.normal(
        shape=(batch_size, self.latent_dim),
        seed=self.seed_generator
    )
    random_vector_labels = ops.concatenate(
        [random_latent_vectors, one_hot_labels], axis=1
    )

    generated_images = self.generator(random_vector_labels)
```

```

fake_image_and_labels = ops.concatenate(
    [generated_images, image_one_hot_labels], -1
)
real_image_and_labels = ops.concatenate([
    real_images, image_one_hot_labels], -1)
combined_images = ops.concatenate(
    [fake_image_and_labels, real_image_and_labels],
    axis=0
)

labels = ops.concatenate(
    [ops.ones((batch_size, 1)),
     ops.zeros((batch_size, 1))], axis=0
)

with tf.GradientTape() as tape:
    predictions = self.discriminator(combined_images)
    d_loss = self.loss_fn(labels, predictions)
    grads = tape.gradient(
        d_loss, self.discriminator.trainable_weights)
    self.d_optimizer.apply_gradients(
        zip(grads, self.discriminator.trainable_weights)
    )

random_latent_vectors = keras.random.normal(
    shape=(batch_size, self.latent_dim),
    seed=self.seed_generator
)
random_vector_labels = ops.concatenate(
    [random_latent_vectors, one_hot_labels], axis=1
)

misleading_labels = ops.zeros((batch_size, 1))

with tf.GradientTape() as tape:
    fake_images = self.generator(random_vector_labels)
    fake_image_and_labels = ops.concatenate(
        [fake_images, image_one_hot_labels], -1
    )
    predictions = self.discriminator(

```

```

        fake_image_and_labels)
        g_loss = self.loss_fn(
            misleading_labels, predictions)
        grads = tape.gradient(
            g_loss, self.generator.trainable_weights)
        self.g_optimizer.apply_gradients(zip(
            grads, self.generator.trainable_weights))

        self.gen_loss_tracker.update_state(g_loss)
        self.disc_loss_tracker.update_state(d_loss)
        return {
            "g_loss": self.gen_loss_tracker.result(),
            "d_loss": self.disc_loss_tracker.result(),
        }

```

13. Let us now use this class to create our conditional GAN and compile it with the Adam optimizer for both generator and discriminator and Binarycrossentropy loss:

```

cond_gan = cGAN(
    discriminator=discriminator, generator=generator,
    latent_dim=latent_dim
)
cond_gan.compile(
    d_optimizer=Adam(learning_rate=0.0003),
    g_optimizer=Adam(learning_rate=0.0003),
    loss_fn=BinaryCrossentropy(from_logits=True),
)

```

14. Now, we train the conditional GAN:

```
cond_gan.fit(dataset, epochs=50)
```

15. Finally, post-training, the generator can whip up new images based on specific labels. We first extract the trained generator, define the number of intermediate images to be generated between the interpolation start and end points, sample noise for interpolation, and then we can generate interpolated images between two chosen classes:

```

trained_gen = cond_gan.generator

# Choose the number of intermediate images
# that would be generated in
# between the interpolation + 2 (start and last images).
num_interpolation = 9
# Sample noise for the interpolation.
interpolation_noise = keras.random.normal(shape=(1, latent_dim))

```



```
interpolation_noise = ops.repeat(
    interpolation_noise, repeats=num_interpolation)
interpolation_noise = ops.reshape(
    interpolation_noise, (num_interpolation, latent_dim))

def interpolate_class(first_number, second_number):
    # Convert the start and end labels to one-hot
    # encoded vectors.
    first_label = keras.utils.to_categorical(
        [first_number], num_classes)
    second_label = keras.utils.to_categorical(
        [second_number], num_classes)
    first_label = ops.cast(first_label, "float32")
    second_label = ops.cast(second_label, "float32")

    # Calculate the interpolation vector between the two labels.
    percent_second_label = ops.linspace(
        0, 1, num_interpolation)[: , None]
    percent_second_label = ops.cast(
        percent_second_label, "float32")
    interpolation_labels = (
        first_label * (1 - percent_second_label) +
        second_label * percent_second_label
    )

    # Combine the noise and the labels and
    #run inference with the generator.
    noise_and_labels = ops.concatenate(
        [interpolation_noise, interpolation_labels], 1)
    fake = trained_gen.predict(noise_and_labels)
    return fake

start_class = 3
end_class = 7
fake_images = interpolate_class(start_class, end_class)
```

Here is the generated image:



Figure 7.6: Generated image using cGAN between classes 3 and 7

In this section, we implemented a cGAN on the MNIST dataset, demonstrating how we can generate diverse and high-quality data samples. This approach can be extended to IoT applications, where cGANs can create synthetic sensor data for testing, improve ML model training, and enhance the robustness of IoT systems. By generating synthetic data, we have illustrated the potential of cGANs to fill in the gaps where real data might be limited or difficult to collect.

With a solid understanding of how generative models can enhance IoT applications, we now turn our attention to another exciting domain: NLP. In the next section, we will delve into KerasNLP and explore how advanced NLP techniques can be deployed on edge devices, bringing powerful language processing capabilities directly to IoT ecosystems.

KerasNLP

NLP has become an integral part of our daily interactions with technology. From voice assistants to language translation services, NLP powers a wide range of applications. **KerasNLP**, a specialized extension of the popular Keras library, brings the power of NLP to developers, making it easier to build, train, and deploy sophisticated language models. It is a powerful and flexible library designed to simplify the development of NLP models. Built on the Keras API, KerasNLP provides a suite of modular components that can be used to create state-of-the-art NLP workflows.

KerasNLP is designed to provide high-level, easy-to-use interfaces for creating NLP models. Leveraging the simplicity and modularity of Keras, it offers a suite of tools and pre-built components for tasks such as tokenization, embedding, sequence modeling, and text generation. Here's a quick overview of what KerasNLP brings to the table:

- **Preprocessing utilities:** Tools for text tokenization, detokenization, and sequence padding
- **Pre-trained models:** Access to pre-trained models for various NLP tasks such as sentiment analysis, text classification, and machine translation
- **Custom layers and modules:** Specialized layers for building custom NLP models, including attention mechanisms, recurrent layers, and transformers
- **Seamless integration:** Compatibility with TensorFlow and Keras workflows, enabling easy integration with existing models and pipelines

Running NLP models on edge devices presents unique challenges and opportunities. Edge devices, such as smartphones, IoT devices, and embedded systems, often have limited computational resources and memory constraints. However, deploying NLP models on these devices can enable real-time, offline processing and enhance privacy by keeping data local.

KerasNLP is well suited for edge deployments due to its lightweight nature and flexibility. Here are some ways to leverage KerasNLP on edge devices:

- **Model compression:** Techniques such as quantization and pruning can reduce the model size and improve inference speed without significant loss in accuracy
- **Efficient architectures:** Using compact model architectures such as MobileBERT or DistilBERT, which are optimized for performance on edge devices
- **On-device training:** Fine-tuning pre-trained models on edge devices using small datasets to adapt to specific tasks or user preferences

Let's dive into some code examples to illustrate how KerasNLP can be utilized for NLP tasks on edge devices.

Sentiment analysis using pre-trained Tiny Bert

Before starting, let us know a little about **Bidirectional Encoder Representations from Transformers (BERT)**. **BERT** is a groundbreaking model in the field of NLP developed by Google. Unlike traditional models that read text input sequentially (left to right or right to left), BERT processes text bi-directionally. This means it considers the context from both directions, providing a deeper understanding of language and significantly improving performance on various NLP tasks.

To accommodate the constraints of edge devices with limited computational resources, we use **TinyBERT**, a compressed version of the original BERT model. TinyBERT retains the powerful language understanding capabilities of its larger counterpart while being optimized for efficiency and speed, making it ideal for deployment in resource-constrained environments. We use pre-trained TinyBERT for sentiment analysis:

```
import keras_nlp
import keras
classifier = keras_nlp.models.BertClassifier.from_preset(
    "bert_tiny_en_uncased_sst2")
# Note: batched inputs expected so must wrap string in iterable
classifier.predict(["I enjoy coding"])
```

The model gives output as an array of logits per class, so an array of [0, 0] is a 50% chance of positive. For the preceding text, I enjoy coding, the output is as follows:

```
array([[ -1.588,   1.891]], dtype=float16)
```

This means that it is 97% positive.

Fine-tuning a model in KerasNLP

In this section, we fine-tune the BERT base model on the IMDB review dataset. We specifically use the `bert_base_en_uncased` model, a 12-layered BERT model with 109.48 M parameters. The dataset is loaded using the TensorFlow dataset library. You can see from the following, just like Keras, with very few lines of code, we can fine-tune a pre-trained model:

```
import os
import tensorflow_datasets as tfds
import keras_nlp
import keras

# Set the backend to TensorFlow
os.environ["KERAS_BACKEND"] = "tensorflow"

# Load the IMdb dataset
imdb_train, imdb_test = tfds.load(
    "imdb_reviews",
    split=["train", "test"],
    as_supervised=True,
    batch_size=16,
)

# Load a pre-trained BERT model
classifier = keras_nlp.models.BertClassifier.from_preset(
    "bert_base_en_uncased",
    num_classes=2,
)

# Fine-tune the model on IMdb movie reviews
classifier.fit(imdb_train, validation_data=imdb_test)
```

Through evaluating on the test dataset, we get a sparse categorical accuracy of 92.87%:

```
classifier.evaluate(imdb_test)
```

KerasNLP supports the latest language models, such as Llama 3 and Gemma; for a complete list of available pre-trained models, please refer to https://keras.io/api/keras_nlp/models/.

If you have sufficient data, you can also train a model from scratch. KerasNLP allows you to define your own model architecture using its various layers or utilize the architecture of standard **large language models (LLMs)**.

Summary

This was an interesting chapter, and I hope you enjoyed reading it as much as I enjoyed writing it. A lot is happening in the generative AI space, and in this chapter, we could only scratch the surface. This chapter introduced generative models and their classification, namely implicit generative models and explicit generative models. The first generative model that was covered is VAEs; they're an explicit generative model and try to estimate the lower bound on the density function. The VAEs were implemented in Keras and were used for anomaly detection.

This chapter then moved on to a more popular explicit generative model: GANs. The GAN architecture, especially how the discriminator network and generative network compete with each other, was explained. Multiple variations of GANs and their special features were covered. We implemented a conditional GAN using Keras for generating handwritten digits. Lastly, we delved into KerasNLP and explored pre-trained language models for sentiment analysis tasks, fine-tuning a pre-trained model.

With this chapter, we mark the end of part one of this book. Until now, we concentrated on different ML and DL models, which we'll require to understand our data and use it for prediction/classification, and other tasks. From the next chapter onward, we'll be talking more about the data itself and how we can process the data in the present IoT-driven environment.

In the next chapter, we'll move toward distributed processing, a necessity when dealing with a large amount of data, and explore platforms that offer distributed processing.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/ds>



Part 3: Implementing Intelligent IoT Solutions in Diverse Domains

This part demonstrates how to apply AI to specific data types and domains. From time series forecasting and visual recognition to speech and text-based IoT applications, we will explore end-to-end workflows for building smart, responsive systems across cloud and distributed infrastructures.

This part has the following chapters:

- *Chapter 8, Distributed AI for IoT*
- *Chapter 9, AI Cloud Platforms for IoT*
- *Chapter 10, Deep Learning for Time Series Data from IoT*
- *Chapter 11, AI for Video and Image Data from IoT*
- *Chapter 12, AI for Text, Audio, and Speech Data from IoT*

8

Distributed Learning Using Keras

Imagine you're building a giant LEGO model with millions of pieces. Working alone, it would take ages! But what if you had a group of friends helping you? Each person could focus on a different section, and you could finish the model much faster.

Distributed training is like that but for machine learning models. Instead of one computer doing all the work, we use multiple computers (or “nodes”) working together as a team. This approach helps us tackle the challenges of training increasingly complex and data-hungry models in a reasonable amount of time.

In this chapter, we're going to cover the following main topics with regard to distributed training using Keras:

- What is Distributed Training?
- Keras 3 and Distributed Learning
- The Keras Distribution API
- Distributed Training in Action

Technical requirements

For this chapter, you'll need to have Keras 3, TensorFlow, and JAX installed. To fully leverage the code examples, having multiple GPU devices available will be beneficial. You can access the complete code for this chapter at the following link: <https://github.com/PacktPublishing/Hands-On-Artificial-Intelligence-for-IoT-Second-Edition/tree/main/Chapter08>.

What is Distributed Training?

As we venture into this chapter, we will explore a critical aspect of deploying **artificial intelligence (AI)** in the **Internet of Things (IoT)** environment: distributed learning. In earlier chapters, we examined how AI models are developed, trained, and deployed on IoT devices using frameworks such as Keras. Now, as we delve into distributed learning, we bridge the gap between traditional centralized training and the demands of IoT systems that span vast networks of devices.

As we know from the past chapters, IoT is an expansive network of interconnected devices embedded with sensors, actuators, and software. It's revolutionizing various sectors, from smart homes and cities to industrial automation and healthcare. In this transformative landscape, AI, particularly machine learning, acts as a catalyst, enabling these "smart" devices to learn from data, adapt to their environment, and make intelligent decisions autonomously.

A key challenge in realizing the full potential of AI for IoT lies in efficiently training and deploying machine learning models on resource-constrained devices. Traditional centralized learning approaches, where all data is collected and processed on a powerful central server, often fall short due to limitations in bandwidth, latency, and privacy concerns. This is where distributed learning comes into play.

Distributed training is a method in deep learning where the training of a neural network is spread across multiple computational devices or nodes. This technique is essential when dealing with large-scale models and datasets that exceed the capabilities of a single machine. By distributing the workload, we can significantly reduce the training time and handle more complex models, which would be impractical on a single device.

Distributed training involves several key components:

- **Multiple devices or nodes:** The training process is distributed across a cluster of devices or nodes, which can include CPUs, GPUs, or specialized hardware such as **tensor processing units (TPUs)**. Each device processes a subset of the data and contributes to the overall training process.
- **Communication protocols:** Efficient communication protocols are necessary to synchronize data, parameters, and gradients across the devices. Common protocols include **Message Passing Interface (MPI)** and **NVIDIA Collective Communications Library (NCCL)**. These protocols ensure that all devices are updated with the most recent model parameters, maintaining consistency across the distributed system.
- **Synchronization mechanisms:** During training, devices need to periodically synchronize to ensure consistent model updates. This involves averaging gradients or model parameters across all devices. Synchronization can be done using various strategies, such as synchronous training, where all devices update simultaneously, or asynchronous training, where updates happen independently.
- **Scalability:** The system must be scalable to accommodate more devices as the complexity of the model or the size of the dataset increases. Scalability ensures that distributed training remains efficient even as the number of devices or the amount of data grows.

Why distributed learning is crucial for IoT

In a typical IoT setup, data is collected from a myriad of sensors and devices dispersed across various locations. Sending all this data to a centralized server for processing and model training not only incurs significant communication costs but also poses latency and privacy concerns. This is where distributed learning comes into play, allowing models to be trained locally on edge devices, thereby reducing the need for data to travel over the network. This approach not only conserves bandwidth but also ensures that sensitive data remains closer to its source, enhancing security and privacy.

Moreover, the distributed nature of IoT systems often entails that data is inherently heterogeneous, with different devices generating data of varying quality and quantity. Distributed learning techniques are particularly well-suited to handle such diversity, enabling models to be trained on device-specific data while still contributing to a global model. This is achieved through various distributed learning paradigms such as federated learning, parameter server models, and model parallelism, all of which will be discussed in this chapter.

By leveraging distributed training techniques, AI can be more effectively integrated into IoT ecosystems, empowering devices with real-time learning capabilities that are both scalable and efficient. The following sections will explore these distributed learning strategies in depth, demonstrating how they can be implemented using Keras 3 to meet the unique challenges posed by IoT environments.

Keras 3 and Distributed Learning

Keras 3, the latest version of the widely used deep learning framework, is designed with flexibility, scalability, and ease of use in mind, making it an excellent choice for distributed learning. One of the standouts features of Keras 3 is its multi-backend support, allowing developers to choose from TensorFlow, PyTorch, or JAX as the underlying engine for their models. This flexibility not only broadens the framework's appeal but also ensures that distributed learning can be implemented efficiently across different environments and use cases. Next, we'll discuss how Keras 3 leverages its integration with these backends to facilitate distributed training, and the specific strategies it offers for this purpose.

Multi-backend support in Keras 3

Keras 3's ability to interface with multiple deep learning backends—TensorFlow, PyTorch, and JAX—provides developers with a versatile platform that can be adapted to various computational needs and hardware setups. Each of these backends brings unique strengths to distributed learning:

- **TensorFlow integration:** TensorFlow is well-known for its comprehensive support for distributed computing, making it a natural choice for Keras 3. With TensorFlow as the backend, Keras 3 can leverage TensorFlow's advanced distributed training strategies, including **MirroredStrategy**, **MultiWorkerMirroredStrategy**, **TPUStrategy**, and **ParameterServerStrategy**. TensorFlow's ecosystem also offers extensive tools for data processing, model optimization, and deployment, making it a robust option for large-scale distributed learning.

- **PyTorch integration:** PyTorch has gained popularity due to its dynamic computation graph and user-friendly API, which make it particularly well-suited for research and experimentation. Keras 3's integration with PyTorch enables distributed training through PyTorch's native support for data parallelism, **distributed data parallel (DDP)**, and other parallelization techniques. PyTorch's flexibility allows developers to implement custom distributed training loops with ease, providing fine-grained control over the training process. This is particularly beneficial in research settings where experimental models and unconventional training strategies are common.
- **JAX integration:** JAX is a rising star in the deep learning community, known for its ability to perform high-performance numerical computing and automatic differentiation. Keras 3's integration with JAX allows for distributed training that takes advantage of JAX's ability to compile and run code on accelerators such as GPUs and TPUs. JAX's functional programming paradigm and support for automatic vectorization (via `vmap`) and parallelization (via `pmap`) make it a powerful tool for implementing complex distributed learning algorithms. Keras 3 combined with JAX is particularly well-suited for scenarios requiring high computational efficiency and flexibility.

Distributed training strategies in Keras 3

Through its robust **distribution API**, Keras 3 provides a variety of strategies for implementing distributed learning, each tailored to the strengths of the underlying backend:

- **Data parallelism:** This is the most common approach to distributed training and is supported across TensorFlow, PyTorch, and JAX. In data parallelism, the dataset is split into smaller batches, and each batch is processed by a separate device or node. The models on each device are updated simultaneously, and the gradients are averaged and applied to the global model. Keras 3 abstracts much of the complexity involved in setting up data parallelism, making it accessible even to those new to distributed learning.
- **Model parallelism:** For extremely large models that cannot fit into the memory of a single device, model parallelism becomes essential. Keras 3 supports this through its integration with TensorFlow's `ParameterServerStrategy` module, PyTorch's model-sharding capabilities, and JAX's ability to split and distribute computational graphs across devices. This strategy allows different parts of a model to be trained on different devices, making it possible to handle models with extremely large parameter counts.
- **Federated learning:** While Keras 3 does not natively support federated learning out of the box, its integration with TensorFlow Federated and PyTorch's federated learning libraries allows developers to implement this distributed training paradigm. Federated learning is particularly useful in IoT environments where data is distributed across many devices, and privacy concerns prevent centralizing the data. This method allows each device to contribute to the model training without sharing its raw data, thus preserving privacy.

- **Hybrid approaches:** One of the key strengths of Keras 3 is its ability to support hybrid distributed learning approaches that combine the strengths of different strategies and backends. For example, a project could use TensorFlow for data parallelism on multiple GPUs while leveraging JAX's `pmap` feature for efficient parallelization across TPUs. This flexibility allows developers to optimize their distributed training setups according to the specific requirements of their models and the available hardware.
- **Custom training loops and strategies:** For advanced use cases, Keras 3 allows developers to create custom training loops that are tightly integrated with the distributed capabilities of TensorFlow, PyTorch, or JAX. This feature is particularly useful for implementing novel distributed training algorithms or for optimizing existing strategies to better fit the specific needs of a project. By enabling custom loops, Keras 3 provides the necessary tools to explore new frontiers in distributed learning research.

In summary, Keras 3, with its backend flexibility and unified distribution API, provides a robust framework for implementing various distributed training approaches, irrespective of the chosen backend. Whether you're dealing with large datasets, complex models, or privacy-sensitive data, Keras 3 equips you with the tools to scale your machine learning workflows and leverage the power of distributed computing. Let us dive deeper into the Keras distribution API in the next section.

The Keras Distribution API

In the previous section, we introduced the Keras 3 distribution API as a powerful tool for implementing distributed learning across various backends, including TensorFlow, PyTorch, and JAX. Now, we will explore this API in greater detail, examining the different strategies it offers, how to implement them, and the practical considerations involved in optimizing distributed training workflows.

The Keras 3 distribution API provides a high-level abstraction for distributed training, allowing developers to scale their models across multiple devices and nodes with minimal code changes. This API is designed to work with ease across different backends, making it versatile for use in various environments, from single-machine multi-GPU setups to large-scale distributed clusters. This API introduces several key features and tools to enable efficient scaling of models on various computational resources such as GPUs and TPUs.

Key features of the Keras 3 distribution API

The Keras 3 distribution API is designed to provide a flexible and powerful framework for distributed training, enabling developers to efficiently scale their machine learning workflows across diverse computational environments. Here, we delve into the key features that make this API indispensable for distributed learning.

Multi-backend support

One of the most significant strengths of the Keras 3 distribution API is its **multi-backend support**. This feature allows smooth integration with different machine learning backends, such as TensorFlow, PyTorch, and JAX, without disrupting the consistent API. The multi-backend support ensures that developers can leverage the specific strengths of each framework, whether it's TensorFlow's comprehensive suite of distributed strategies, PyTorch's dynamic computation graph, or JAX's powerful automatic differentiation and vectorization capabilities. Here are two of its most standout features:

- **Consistent API across backends:** The distribution API abstracts the differences between these backends, providing a unified interface for distributed learning. This means that the same code base can be used to run distributed training across different backends, making it easier to switch between them based on project requirements or available hardware.
- **Backend-specific optimizations:** While the API provides a consistent interface, it also takes advantage of backend-specific features for efficient distribution. For example, it can leverage TensorFlow's advanced distribution strategies, PyTorch's DDP, and JAX's sharding and parallelization capabilities. This ensures that distributed training is optimized for the underlying hardware and software stack.

Let's look at data and model parallelism next!

Data and model parallelism

The Keras 3 distribution API supports both **data parallelism** and **model parallelism**, which are essential techniques for distributing computations across multiple devices.

Data parallelism

In data parallelism, the dataset is split across multiple devices, and each device processes a portion of the data. This approach is particularly useful when dealing with large datasets that cannot fit into the memory of a single device. The API manages the synchronization of gradients across devices, ensuring that model updates are consistent and efficient.

The **DataParallel API** in Keras facilitates distribution for data parallelism by allowing developers to split datasets across multiple devices. This API can be initialized either by specifying a `DeviceMesh` class or directly listing the devices to be used. The `device_mesh` argument should be a 1D `DeviceMesh` instance, and if the mesh has multiple axes, the first axis is treated as the data parallel dimension, with a warning issued to the user. If a list of devices is provided instead, the `DataParallel` class will automatically construct a 1D mesh:

```
keras.distribution.DataParallel(  
    device_mesh=None,  
    devices=None,  
    auto_shard_dataset=True  
)
```

Additionally, if neither a mesh nor devices are specified, the `list_devices()` function is called to detect available devices and create a 1D mesh. The `auto_shard_dataset` argument, which defaults to `True`, ensures that the dataset is automatically sharded across the available devices. This class is particularly useful for optimizing the distribution of large datasets, making it easier to balance computational loads and improve training efficiency across multiple GPUs.

Model parallelism

Model parallelism is used when the model itself is too large to fit into the memory of a single device. The model is split across multiple devices, with each device responsible for computing a part of the model. This approach is crucial for handling very large models, such as those used in natural language processing or complex computer vision tasks.

The distribution API simplifies the implementation of these parallelism strategies by providing high-level abstractions that handle the intricacies of distributing data and model computations across devices: the `ModelParallel` class.

The `ModelParallel` class in Keras is designed to distribute model variables across multiple devices, enabling the sharding of both variables and input data. Unlike `DataParallel`, which replicates variables across all devices, `ModelParallel` allows for partitioning variables, making it especially useful for large models that need to be distributed across multiple GPUs. To set up `ModelParallel`, you need two key components: a `DeviceMesh` class and a `LayoutMap` class. The `DeviceMesh` contains information about the physical devices, and its axes' names determine how variables and data are laid out across these devices. The `LayoutMap` maps specific model variable paths to their corresponding `TensorLayout`, offering fine-grained control over the distribution of each part of the model. A significant advantage of `ModelParallel` is the ability to adjust the sharding factor by updating the `DeviceMesh` shape, providing flexibility in how the model is distributed. Before applying this distribution, you can list all variable paths in your model to determine the most appropriate layout mapping for each variable. The `ModelParallel` class also allows you to specify a `batch_dim_name` argument, which is the axis in the device mesh used to distribute data; if not provided, the first axis of the device mesh is used by default.

DeviceMesh and TensorLayout

The concepts of `DeviceMesh` and `TensorLayout` are central to the Keras 3 distribution API, providing a structured way to manage distributed resources and data:

- **DeviceMesh:** This class represents a cluster of computational devices that are configured for distributed computation. A `DeviceMesh` maps physical devices (such as GPUs or TPUs) to a logical mesh structure, which facilitates the organization and management of distributed resources. This concept is closely aligned with `jax.sharding.Mesh` and `tf.dtensor`.

Mesh, providing a consistent way to structure device management across these frameworks. This logical structure allows developers to specify how computations and data are distributed across the devices in the mesh, making it easier to optimize resource usage and reduce communication overhead. The `DeviceMesh` class takes three arguments: `keras.distribution.DeviceMesh(shape, axis_names, devices=None)`, which define how tensors are distributed across the devices, which is crucial for efficient parallel computation. The first argument, `shape`, is a tuple or list of integers that specifies the shape of the device mesh. This shape determines the distribution strategy. For example, `(8, )` could be used for a data-parallel-only distribution, while `(4, 2)` could indicate both model and data parallelism.

Note

The `shape` argument should be configured based on the number of GPUs available. For instance, if you have only one GPU and just want to test the code, you can use `(1, 1)` as the shape to keep the setup simple.

The second argument, `axis_names`, is a list of strings that logically name each axis of the device mesh. These names help match and create the `TensorLayout` class when distributing data and variables across devices. The third argument, `devices`, is optional; when used, it provides a list of devices to use. If not specified, it defaults to all available devices on the local machine, as listed by `keras.distribution.list_devices()`.

Here's how you can create a `DeviceMesh`. Start by defining `shape`; the shape of the mesh determines how many devices you have and how they are organized. For example, if you have 8 devices, you might arrange them in a grid with 4 rows and 2 columns, so the shape would be `(4, 2)`:

```
shape = (4, 2)
```

The next step would be to assign logical names to each axis of the mesh, using `axis_names`. These names help us refer to each dimension of the mesh when specifying how data and computations are distributed. It is common to use `'data'` for the data parallelism axis and `'model'` for the model parallelism axis:

```
axis_names = ['data', 'model']
```

Then, specify the actual devices you are using, if you don't specify this, Keras will automatically use all available devices on your machine. And that's it! Now, you can create a `DeviceMesh`:

```
device_mesh = DeviceMesh(
    shape=shape,
    axis_names=axis_names,
    devices=devices
)
```

- **TensorLayout**: This class specifies how tensors are distributed across the `DeviceMesh`. It details the sharding of tensors along specified axes, allowing developers to control how data is partitioned and distributed across multiple devices. The `TensorLayout` class is aligned with `jax.sharding.NamedSharding` and `tf.dtensor.Layout`, providing a consistent approach to tensor distribution across different frameworks. The `axes` argument in `TensorLayout` maps the tensor's dimensions to the axis names defined in the `DeviceMesh`. For any dimensions that do not require sharding, `None` can be used as a placeholder. Additionally, the `device_mesh` argument specifies the `DeviceMesh` to be used, though the actual mapping of the tensor to physical devices isn't determined until this mesh is defined. This fine-grained control over tensor distribution is crucial for optimizing performance in distributed training, especially when working with complex models and large datasets.

`TensorLayout` specifies how each dimension of a tensor is mapped onto the axes of the `DeviceMesh`. Suppose you have a tensor with the `(batch_size, feature_size)` shape. You might want to shard the `batch_size` dimension across the `'data'` axis (to distribute batches across devices) and replicate the `feature_size` dimension across all devices (each device has a full copy of the features). You can represent this mapping as follows:

```
tensor_layout = TensorLayout(  
    axes= ['data', None],  
    device_mesh=device_mesh  
)
```

It is important to understand the difference between sharding and replication. In sharding, we split the tensor along a dimension so that each device gets a slice of the data. In replication, each device gets a full copy of the tensor along that dimension. In the preceding example, we're distributing the workload by splitting the data batches across devices (data parallelism), while each device has the full set of features.

- **LayoutMap**: Keras introduces the `LayoutMap` class, which allows you to specify the `TensorLayout` for any weights or intermediate tensors from a global perspective. `LayoutMap` is a dict-like object that maps string keys to `TensorLayout` instances. However, it differs from a regular Python dictionary in that the string key is treated as a **regular expression (regex)** when retrieving values. This feature allows for flexible and dynamic mapping, as it enables you to define a naming schema for `TensorLayout` and retrieve corresponding instances efficiently. Typically, the key used to query `LayoutMap` is the `variable.path` attribute, which uniquely identifies each variable. When inserting values, you can use a tuple or list of axis names, which will automatically be converted into a `TensorLayout`. Additionally, the `LayoutMap` can optionally include a `DeviceMesh` to populate the `TensorLayout.device_mesh` parameter if it is not explicitly set. When retrieving a layout, if an exact match is not found, all existing keys in the `LayoutMap` are treated as regex patterns and matched against the input key. If multiple matches are found, a `ValueError` is raised, ensuring clarity in the layout specification. If no match is found, `None` is returned, providing a clear indication that the layout needs to be specified.

Managing the layouts of multiple tensors and variables can become complex, especially in large models. `LayoutMap` helps by allowing you to define layouts for variables globally, using patterns to match variable names. You can initialize `LayoutMap` with `DeviceMesh`:

```
layout_map = LayoutMap(device_mesh=device_mesh)
```

We can also use variable names or patterns to specify the `TensorLayout` for different parts of your model. Consider the following two examples.

In the first example, we shard the kernels of all layers named `'dense'` along the `'model'` axis (model parallelism):

```
layout_map['dense/kernel'] = TensorLayout (
    axes=[None, 'model'],
    device_mesh=device_mesh
)
```

Here, `axes=[None, 'model']` means the first dimension is replicated (`None`), and the second dimension is sharded over the `'model'` axis.

In the second example, we replicate all biases (do not shard):

```
layout_map['.*bias'] = TensorLayout (
    axes=[None],
    device_mesh=device_mesh
)
```

Here, the `'.*bias'` regex pattern matches any variable name that ends with `'bias'`, and `axes=[None]` means the variable is replicated across all devices.

If multiple patterns match a variable name, Keras will raise an error to prevent ambiguity. It's important to design your patterns carefully to avoid overlaps.

These classes provide the foundational tools for managing distributed training at a low level, offering flexibility and control over how computations are executed across multiple devices.

Global programming model

The Keras 3 distribution API provides a **global programming model** that allows developers to compose applications that operate on tensors in a global context as if they were working with a single device. This abstraction significantly simplifies the development process by automatically managing the distribution of computations across multiple devices.

- **Global context operation:** Developers can write their applications as though they are working on a single device, but the API handles the underlying distribution, ensuring that the program scales seamlessly across different computational environments. This model abstracts away the complexities of distributed computation, enabling developers to focus on the application logic.

- **Seamless scaling:** By decoupling the application from the sharding directives, the same code base can be used to run on a single device, multiple devices, or even multiple clients, without modification. This flexibility is crucial for environments where computational resources may vary, such as in cloud-based or edge-computing scenarios.

Single program, multiple data expansion

The **single program, multiple data (SPMD)** expansion approach is a powerful feature of the Keras 3 distribution API that involves distributing the program and tensors according to sharding directives:

- **Flexible distribution:** Through SPMD, the same program and tensors are distributed across devices according to sharding directives. This approach allows for running the same application on various configurations—from a single device to multiple devices or clients—while preserving the global semantics of the program.
- **Global semantics preservation:** SPMD ensures that the application's global semantics are maintained, meaning that the program behaves consistently across different distributed environments. This decoupling of application logic from the specific sharding strategy enables flexible scaling without compromising the correctness of the application.

This feature is particularly powerful in frameworks such as JAX and TensorFlow, where the distribution API leverages backend-specific capabilities to optimize the distribution and execution of computations.

Automatic device detection

The distribution API includes features for **automatic device detection**, simplifying the setup process for distributed training:

- **Simplified setup:** The API can automatically detect the available devices (such as GPUs, TPUs, or CPUs) and configure them for distributed training. This reduces the manual effort required to set up distributed environments, allowing developers to focus on optimizing their models and training procedures.
- **Dynamic configuration:** Automatic device detection ensures that the API dynamically adjusts to the available resources, making it easier to deploy models in environments where the hardware may change or scale up dynamically, such as in cloud-based deployments.

Usage of DataParallel and ModelParallel classes

The distribution API provides the `DataParallel` and `ModelParallel` classes, which abstract the complexity involved in parallel computation:

- **DataParallel:** This class is used for distributing data across devices, ensuring that each device processes a portion of the data simultaneously. The API handles the synchronization of gradients and model updates, making it easy to implement data parallelism without dealing with the complexities of inter-device communication.

- **ModelParallel:** This class is used for distributing model computations across multiple devices. It is particularly useful for large models that cannot fit into the memory of a single device. The API ensures that different parts of the model are computed in parallel, optimizing resource usage and reducing training time.

Integration with backend-specific features

The Keras 3 distribution API is designed to leverage **backend-specific features** to optimize distributed training:

- **Backend-specific optimizations:** Depending on the backend in use, the API can automatically take advantage of optimizations such as TensorFlow's distribution strategies, PyTorch's DDP, or JAX's sharding and parallelization capabilities. This ensures that distributed training is not only possible across different environments but is also optimized for performance and resource utilization.
- **Cross-backend compatibility:** The API's design ensures that developers can switch between backends with minimal code changes, enabling them to leverage the most appropriate backend for their specific use case or hardware environment.

The Keras 3 distribution API is a powerful and flexible framework for distributed training, offering a wide range of features designed to simplify the process of scaling machine learning workflows across multiple devices and environments. With its multi-backend support, advanced parallelism capabilities, and robust abstractions for managing distributed resources, the distribution API enables developers to efficiently scale their AI models and make the most of the available computational resources.

In the next section, we will provide practical examples of using the Keras 3 distribution API with different backends, demonstrating how to implement distributed training in real-world scenarios. We'll also explore best practices for optimizing distributed workflows to ensure that your AI models are trained efficiently and effectively, regardless of the scale of your deployment.

Distributed Training in Action

Building on the powerful capabilities of the Keras 3 distribution API, this section delves into the practical implementation of distributed deep learning across multiple backends, including JAX, TensorFlow, and PyTorch. The API simplifies scaling by providing tools for both data and model parallelism, efficiently distributing workloads across GPUs, TPUs, and multiple hosts. Through a global programming model, it abstracts the complexities of parallel computation, automatically managing device allocation and tensor layout. This flexible approach allows developers to deploy their AI models on varying scales while maintaining consistent application logic and global semantics.

In this section, we will walk through a practical example of setting up distributed training using the Keras 3 distribution API. We'll demonstrate how to configure the environment for distributed learning, select the appropriate backend, and initialize a simple model for training across multiple devices. This example illustrates how to leverage the API to efficiently distribute both data and computation, ensuring optimal utilization of available hardware resources. Follow these steps:

1. We begin by configuring the necessary environment for distributed training. In this example, we'll use **JAX** as the backend, although you can also choose **TensorFlow** or **PyTorch**. This step involves setting the backend, importing the essential libraries, and preparing the environment for distributed learning. This foundational setup is essential for the subsequent steps, where we will demonstrate the implementation of distributed training using the Keras 3 distribution API:

```
import os
os.environ["KERAS_BACKEND"] = "jax"
import keras
from keras import layers
from keras.distribution import (
    DeviceMesh, TensorLayout, DataParallel)
import jax
import tensorflow as tf
import numpy as np
from tensorflow import data as tf_data
```

2. To efficiently distribute computations, it's essential to know which GPU devices are available on your system. In this step, we use the JAX library to detect and list all GPU devices. This information is vital for setting up the distribution of data and models across these devices, ensuring that the computational load is balanced and that the training process is optimized for performance:

```
devices = jax.devices("gpu")
```

3. To effectively manage the distribution of workloads across multiple GPUs, it's crucial to define how these devices will be utilized in the training process. We can do this using `DeviceMesh`. This mesh determines the structure of the distribution, specifying how data and model computations are allocated across the available GPUs. We have two GPUs, so by setting `shape` to `(2, 1)`, we instruct the system to split the data across 2 GPUs while keeping the model intact, ensuring efficient parallel processing. `axis_names` provides logical labels for these axes, helping to clarify the parallelism strategy:

```
mesh = DeviceMesh(
    shape=(2, 1),
    axis_names=["data", "model"],
    devices=devices
)
```

- Next, we create a `TensorLayout` that maps the tensor's dimensions to the axes of the `DeviceMesh`. By setting the `axes` argument to `( "model", "data" )`, we specify that the model's dimensions are distributed along the first axis of the mesh, and the data is distributed along the second axis. This layout ensures that the computation is aligned with the structure of the `DeviceMesh`, allowing for optimal sharding of the model and data across the devices:

```
tensor_layout = TensorLayout(
    axes=( "model", "data" ),
    device_mesh=mesh
)
```

- Next, we initialize `DataParallel` by specifying the list of available devices. This setup ensures that the dataset is automatically sharded across the GPUs, enabling each device to process a different portion of the data in parallel. By defining `data_parallel` in this way, we create a 1D device mesh from the specified GPUs, streamlining the data distribution and ensuring that each GPU contributes effectively to the training process:

```
data_parallel = DataParallel(devices=devices)
```

- Before we can begin training our model, we need to prepare the dataset that will be distributed across the GPUs. In this step, we generate synthetic input data and labels using NumPy and then convert them into a TensorFlow dataset. This dataset will be batched and ready for parallel processing across the GPUs. We create a synthetic data array, `x`, for training using NumPy with random values that simulate 128 samples of image data with a shape of `(28, 28, 1)`. We also generate `y`, another NumPy array with random values, representing 128 corresponding output labels with 10 classes. Finally, we use **TensorFlow's Dataset API (TFDS)** to combine these inputs and labels into a dataset, which is then batched with a size of 16, making it ready for training:

```
x = np.random.normal(size=(128, 28, 28, 1))
y = np.random.normal(size=(128, 10))
dataset = tf_data.Dataset.from_tensor_slices((x, y)).batch(16)
```

- The next step is to set the distribution strategy for our training process. We accomplish this by setting the `DataParallel` instance as the active distribution strategy using `keras.distribution.set_distribution()`. This step ensures that the dataset is sharded and processed according to the configuration we've established, with the workload being distributed across the specified GPUs:

```
keras.distribution.set_distribution(data_parallel)
```

- With the distribution strategy in place, we now define our neural network model using Keras. This model will be replicated across all devices in the `DeviceMesh`, meaning that all model weights, including **random number generator (RNG)** states, optimizer states, and metrics, will be duplicated on each device. The dataset used in `model.fit` or `model.evaluate` will

automatically be split across the batch dimension and distributed to all devices. Thanks to the global context provided by the distribution strategy, there's no need for manual aggregation of losses or metrics; the computations are managed seamlessly across the distributed environment:

```
inputs = layers.Input(shape=(28, 28, 1))
y = layers.Flatten()(inputs)
y = layers.Dense(
    units=200, use_bias=False, activation="relu")(y)
y = layers.Dropout(0.4)(y)
y = layers.Dense(units=10, activation="softmax")(y)
model = keras.Model(inputs=inputs, outputs=y)
```

9. Let us now compile the model and fit it:

```
model.compile(loss="mse")
model.fit(dataset, epochs=3)

>>> Epoch 1/3
8/8 _____ 1s 51ms/step - loss: 0.9363
Epoch 2/3
8/8 _____ 0s 2ms/step - loss: 0.8517
Epoch 3/3
8/8 _____ 0s 3ms/step - loss: 0.8116
8/8 _____ 0s 11ms/step - loss: 0.7647
```

10. And now, we can evaluate our model:

```
print(model.evaluate(dataset))

>>> 0.7740380764007568
```

By setting up data parallelism and defining, compiling, and training our model, we effectively utilized multiple GPUs to accelerate the training process, ensuring efficient distribution of both data and computation across devices.

After exploring the use of `DataParallel` for distributing data across multiple devices, it's important to recognize that there are scenarios where data parallelism alone may not be sufficient, particularly when dealing with extremely large models. In such cases, where model weights exceed the memory capacity of a single accelerator, `ModelParallel` becomes a critical tool. This approach allows you to distribute not just the data but also the model's weights and activation tensors across multiple devices, making it possible to train models that would otherwise be too large to handle. In the following section, we will explore model parallelism, where we will distribute different parts of the model across multiple GPUs to further optimize training for large and complex models.

Optimizing large models with ModelParallel and LayoutMap

`ModelParallel` becomes essential when your model's weights are too large to fit on a single accelerator. This strategy allows you to distribute your model's weights or activation tensors across all the devices within the `DeviceMesh`, enabling horizontal scaling for large models. Unlike `DataParallel`, where weights are fully replicated across all devices, `ModelParallel` requires a more tailored approach to weight distribution to achieve optimal performance. We do this using the Keras `LayoutMap` class. Let us see how to do this continuing with our previous example:

1. We start by configuring the environment to use JAX as the backend, which is currently required for the distribution API. The following code initializes the necessary libraries and sets up the environment for distributed training with model parallelism:

```
import os
os.environ["KERAS_BACKEND"] = "jax"
import keras
from keras import layers
from keras.distribution import (
    DeviceMesh, TensorLayout, ModelParallel)
import jax
import numpy as np
from tensorflow import data as tf_data
```

2. As in our previous setup, we use the JAX library to detect and list the GPU devices available on our machine. Since this machine has two GPUs, the following code will retrieve and display both devices:

```
devices = jax.devices("gpu")
print(devices)

Output
>>> [CudaDevice(id=0), CudaDevice(id=1)]
```

3. In this step, we configure the `DeviceMesh` to set up model parallelism, which is crucial when dealing with large models that cannot fit on a single GPU. We define the mesh with a shape of `(1, 2)`, meaning the grid has 1 “row” and 2 “columns.” This configuration focuses on distributing the model itself across two GPUs. By splitting the model between the devices, different parts of the model are processed on different GPUs, allowing us to handle larger models that require more memory than a single GPU can provide. The data remains whole and is processed through the model segments on each GPU:

```
mesh = DeviceMesh(
    shape=(1, 2),
    axis_names=["data", "model"],
    devices=devices
)
```

Note

This differs from our earlier data parallelism setup, where we used a `DeviceMesh` with a shape of `(2, 1)`. In that configuration, the emphasis was on distributing the data across two GPUs, with each GPU processing the entire model independently. In contrast, model parallelism focuses on distributing the model across the GPUs, which requires more frequent communication between the devices but allows for handling much larger models.

4. Next, we create a `LayoutMap` to define how different parts of the model are distributed across the devices in the `DeviceMesh`. The `LayoutMap` is essentially a dict-like object that maps model variable paths to `TensorLayout` instances, allowing us to control the sharding of these variables. We initialize the `LayoutMap` with the mesh and then specify the layout for various model components. For example, `layout_map["d1/kernel"] = (None, "model")` indicates that the kernel weights of layer `d1` should be distributed across the GPUs along the "model" axis, while not being sharded along the first dimension. Similarly, `layout_map["d1/bias"] = ("model",)` fully shards the bias of layer `d1` across the "model" axis. Lastly, `layout_map["d2/output"] = ("data", None)` ensures that the output of layer `d2` is sharded along the "data" axis for the first dimension but not the second. This configuration allows for precise control over how model variables are distributed, optimizing the performance of model parallelism by efficiently managing large model components across multiple GPUs:

```
layout_map = LayoutMap(mesh)
layout_map["d1/kernel"] = (None, "model")
layout_map["d1/bias"] = ("model",)
layout_map["d2/output"] = ("data", None)
```

5. Next, we set up the `ModelParallel` distribution strategy by using the `layout_map` parameter we just created. We initialize the `ModelParallel` instance with the `layout_map` that defines how the model variables are distributed across the GPUs. Additionally, we specify `batch_dim_name` as "data", indicating that the data should be distributed along the "data" axis of the `DeviceMesh`:

```
model_parallel = ModelParallel(
    layout_map=layout_map,
    batch_dim_name="data"
)
```

6. With the `ModelParallel` strategy configured, the final step is to set this distribution strategy as active by calling the following:

```
set_distribution(model_parallel)
```


7. To proceed with training, we first need to prepare some synthetic data. In this step, we generate random input data and corresponding labels using NumPy. The `x` array simulates 128 samples of image data, each with a shape of `(28, 28, 1)`, representing grayscale images. Similarly, the `y` array contains 128 samples, each with 10 possible classes. We then combine these inputs and labels into a TensorFlow dataset using `tf_data.Dataset.from_tensor_slices`, and batch the data with a size of 16. This synthetic dataset will be used to test the model parallelism setup, allowing us to simulate the training process across multiple GPUs:

```
x = np.random.normal(size=(128, 28, 28, 1))
y = np.random.normal(size=(128, 10))
dataset = tf_data.Dataset.from_tensor_slices((x, y)).batch(16)
```

8. With the synthetic data prepared, we now define the neural network model that will be trained using the model parallelism strategy:

```
inputs = layers.Input(shape=(28, 28, 1))
y = layers.Flatten()(inputs)
y = layers.Dense(
    units=200, use_bias=False, activation="relu")(y)
y = layers.Dropout(0.4)(y)
y = layers.Dense(units=10, activation="softmax")(y)
model = keras.Model(inputs=inputs, outputs=y)
```

9. With the model defined, we proceed to compile and train it. During this process, the model parallelism strategy is in effect, meaning the model's components and data are distributed across the GPUs as specified earlier:

```
model.compile(loss="mse")
model.fit(dataset, epochs=3)

>>> Epoch 1/3
8/8 _____ 0s 26ms/step - loss: 1.1570
Epoch 2/3
8/8 _____ 0s 3ms/step - loss: 1.0486
Epoch 3/3
8/8 _____ 0s 3ms/step - loss: 1.0191
8/8 _____ 0s 11ms/step - loss: 0.9607
```

10. Now, we can evaluate the model:

```
print(model.evaluate(dataset))

>>> 0.9406238198280334
```

In this process, we configured model parallelism using the Keras 3 distribution API to distribute large models across multiple GPUs. We set up a `DeviceMesh` and `LayoutMap` to control how model components and data are sharded. After defining, compiling, and training the model with synthetic data, we successfully leveraged parallel processing for efficient training.

Summary

In this chapter, we explored the implementation of distributed training using Keras 3, focusing on both data and model parallelism. We began by setting up the environment with JAX as the backend, identifying available GPUs, and configuring a `DeviceMesh` to manage distribution. Using a `LayoutMap`, we tailored how model components were sharded across devices. We then defined, compiled, and trained a neural network model using synthetic data, leveraging the power of parallel processing for efficient training across multiple GPUs.

In the next chapter, we will explore various AI cloud platforms tailored for IoT.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/ds>



AI Cloud Platforms for IoT

In the previous chapter, we explored distributed learning using Keras 3, diving into how distributed training techniques can optimize machine learning models, especially when handling large-scale **Internet of Things (IoT)** data. By leveraging Keras 3's capabilities for distributed learning, IoT applications can scale to process vast amounts of sensor data across multiple devices in parallel, enabling faster and more efficient training.

This chapter extends that discussion by shifting focus to AI cloud platforms for IoT, where we examine the tools and services provided by various cloud providers to analyze IoT-generated data. As IoT systems continue to expand, producing large streams of data in real time, cloud platforms play a pivotal role in ensuring that this data is captured, processed, and analyzed efficiently. From foundational services such as **Infrastructure as a Service (IaaS)**, **Software as a Service (SaaS)**, and **Platform as a Service (PaaS)** to advanced AI tools such as **machine learning (ML)** APIs and pretrained models, these platforms offer comprehensive ecosystems for IoT development and deployment.

Additionally, we will explore how major cloud providers such as **Amazon Web Services (AWS)**, Microsoft Azure, and IBM Watson integrate AI into IoT workflows, allowing developers to extract actionable insights from data streams.

This chapter covers the following topics:

- Introduction to Cloud Computing for IoT
- Cloud Computing Basics
- AI Services in Cloud Platforms
- Training and Deploying AI Models for IoT on Cloud Platforms
- Security and Privacy in IoT Cloud Platforms
- Emerging Trends and the Future of AI Cloud Platforms for IoT

By the end of this chapter, you will have a clear understanding of the current landscape of AI-driven cloud platforms for IoT analytics.

Technical requirements

To run the code in this chapter, you will need to have installed `boto3` and `azure-cognitiveservices-speech`; you can install them using `pip`.

Introduction to Cloud Computing for IoT

As the number of connected devices continues to grow, IoT has emerged as a transformative force across various industries, enabling real-time monitoring, automation, and control. However, IoT devices generate massive amounts of data that need to be processed, stored, and analyzed efficiently. This is where cloud computing becomes a critical enabler for IoT systems. The cloud offers scalable, flexible, and cost-effective solutions for managing vast IoT datasets, allowing organizations to harness the full potential of their connected devices without the need for extensive on-premises infrastructure.

IoT devices are often spread across diverse geographic locations and typically operate in constrained environments with limited computational power and storage capacity. Cloud integration provides a seamless solution by offloading the heavy lifting—such as data aggregation, storage, and processing—to centralized cloud platforms. With cloud computing, IoT applications can leverage advanced analytics, AI, and ML algorithms to derive actionable insights from the data collected by devices.

Cloud platforms enable real-time data processing, allowing IoT systems to analyze streams of information as they are generated. They also provide elastic scalability, meaning that resources can be scaled up or down dynamically based on the volume of IoT data, which is especially important for handling unpredictable spikes in data traffic. Moreover, cloud platforms offer built-in tools for security, device management, and data analytics, significantly reducing the operational complexity of managing IoT ecosystems.

While cloud computing provides the foundation for IoT scalability, several challenges still require thoughtful solutions. These include the following:

- **Data volume and velocity:** IoT devices can generate overwhelming amounts of data at high speeds. Think of industrial machines transmitting terabytes of sensor readings or autonomous vehicles collecting vast quantities of real-time information. Fortunately, cloud platforms with nearly limitless storage and computing power make it possible to manage this data influx with ease, ensuring that even the most data-intensive IoT applications remain responsive and efficient.
- **Data variety and complexity:** IoT data doesn't come in one tidy format. It spans from structured sensor data to unstructured media such as images and videos. Cloud platforms provide tools to integrate, clean, and transform this variety of data into usable formats, making it easier to apply advanced analytics and derive meaningful insights.
- **Security and privacy:** With sensitive IoT data being transmitted and stored in the cloud, security becomes a top concern. Cloud providers address these concerns with advanced measures such as encryption, multi-layer access controls, and real-time threat detection. These protections ensure that IoT data remains secure against cyber threats, unauthorized access, and breaches.

Now that we've explored the challenges of IoT and how cloud computing addresses them, let's dive deeper into the fundamentals of cloud computing. In the next section, we'll examine the core service models—IaaS, PaaS, and SaaS—and see how they power and scale today's IoT solutions.

Cloud Computing Basics

Cloud computing has transformed how data is processed, stored, and managed, especially in the world of IoT. With IoT systems generating vast data streams that demand real-time analysis, cloud infrastructure provides the flexibility and scalability needed to handle these challenges. This section covers the core cloud services—IaaS, PaaS, and SaaS—and their role in supporting IoT. By shifting heavy computational tasks to the cloud, organizations can tap into powerful analytics without investing in expensive hardware. We'll also explore public, private, and hybrid cloud models, highlighting their impact on IoT scalability, security, and flexibility. Understanding these foundations is key to unlocking the full potential of AI-driven IoT analytics.

Types of cloud services

As we have mentioned before, cloud computing services are generally categorized into three main models: IaaS, PaaS, and SaaS. These services provide varying levels of control, flexibility, and management over the underlying infrastructure and applications, making them particularly valuable for IoT implementations. Each model caters to different aspects of IoT development, deployment, and management, allowing organizations to choose the right mix based on their requirements.

IaaS

IaaS offers fundamental computing resources such as virtual machines, storage, and networking, allowing organizations to build and run applications without worrying about the underlying hardware. IaaS is ideal for IoT applications that require extensive data processing, storage, and real-time analytics. Features of IaaS include the following:

- **Scalability:** IaaS platforms provide elastic computing power, allowing IoT systems to handle the fluctuating volume of data generated by devices. As the number of connected devices increases, IaaS can automatically scale resources to ensure consistent performance.
- **Customization:** IoT developers have full control over the virtual machines, operating systems, and other infrastructure elements, enabling them to tailor the computing environment to specific needs, such as high-performance data processing or custom ML pipelines.
- **Real-time processing:** IaaS platforms enable IoT applications to process data in real time, ensuring timely decision-making and response actions, especially in critical sectors such as healthcare, transportation, and manufacturing.

Amazon Elastic Compute Cloud (Amazon EC2), and **Google Compute Engine** are examples of IaaS. EC2 provides scalable computing capacity in the cloud. IoT applications can leverage EC2 for real-time analytics, edge computing, and large-scale ML model deployment. Google's IaaS offering allows IoT developers to run large-scale workloads with customizable virtual machines. Google Compute Engine integrates with other Google Cloud services for IoT data analysis and AI-driven insights.

PaaS

PaaS offers a streamlined development environment with built-in tools, frameworks, and services for building, deploying, and managing applications. By abstracting away the complexities of infrastructure management, PaaS allows developers to focus on the core task of creating and deploying IoT applications, enhancing both efficiency and innovation.

By providing ready-to-use environments, PaaS accelerates development speed, enabling IoT developers to concentrate on writing and testing code rather than configuring servers and infrastructure. It also integrates essential tools, offering pre-built libraries, APIs, and services that simplify development. Features such as real-time analytics, AI integration, and data visualization come as part of the platform, making it easier to build and scale IoT applications.

Additionally, PaaS supports serverless computing, allowing developers to run IoT data processing tasks without managing servers. Serverless solutions, such as **AWS Lambda**, operate on a pay-per-use model, making them highly cost-efficient for handling unpredictable IoT workloads. Leveraging PaaS platforms such as **Microsoft Azure Functions** or AWS Lambda reduces infrastructure complexity and enables rapid deployment of real-time analytics and AI-driven IoT solutions.

SaaS

SaaS delivers complete, ready-to-use applications accessible via the cloud, removing the need for managing infrastructure or developing custom solutions. For IoT, SaaS platforms offer fully managed environments that handle everything from device connectivity to data analytics and visualization, enabling organizations to quickly deploy and scale IoT systems without deep technical expertise.

SaaS for IoT provides plug-and-play solutions that include dashboards, data analytics tools, and device management features, allowing businesses to rapidly implement IoT applications. These platforms are designed to grow with the system, automatically handling scalability as more devices are added. Furthermore, SaaS providers take care of maintenance, software updates, and security, freeing organizations from these operational burdens.

An example is **AWS IoT Core**, a fully managed SaaS platform that securely connects IoT devices to the cloud. It offers built-in device management, real-time analytics, and seamless integration with other AWS services such as **AWS IoT Analytics** and **Amazon SageMaker**, enabling advanced ML applications.

Another example is **IBM Watson IoT Platform**, which provides real-time monitoring, device management, and advanced analytics for IoT data. Watson IoT simplifies the management of large-scale IoT deployments while integrating AI-driven insights to help businesses extract value from their IoT data streams.

These SaaS platforms are invaluable for organizations looking to scale their IoT infrastructure without investing in extensive in-house expertise or hardware.

Cloud deployment models

When deploying IoT applications, choosing the right cloud deployment model is crucial for ensuring scalability, security, and cost-efficiency. Cloud deployment models can be broadly categorized into public, private, and hybrid clouds, each offering unique advantages based on organizational needs and constraints.

Public clouds

Public clouds, operated by third-party providers, offer computing resources such as servers, storage, and networking over the internet. For IoT applications, public clouds provide on-demand scalability, allowing businesses to scale resources as needed without the costs of physical infrastructure. The pay-as-you-go model makes public clouds highly cost-effective, especially for start-ups and organizations looking to deploy and scale IoT solutions quickly.

Public clouds are ideal for IoT systems due to their **elastic scalability**, which can easily handle unpredictable bursts of IoT data. Cloud providers manage and maintain the infrastructure, allowing IoT developers to focus solely on application development. With **global data centers**, public clouds also enable IoT operations to span multiple geographic regions seamlessly.

Best suited for large-scale IoT deployments, such as smart city infrastructure or connected vehicles, public clouds are the go-to choice where cost-efficiency and scalability are critical factors.

Private clouds

Private clouds are exclusive environments used by a single organization, either hosted on-premises or by a third-party provider. Offering greater control, security, and customization, private clouds are favored by industries with stringent regulatory needs, such as healthcare, finance, and government, where IoT data involves sensitive or mission-critical information.

With enhanced security, private clouds ensure data privacy and compliance since resources aren't shared with other organizations. They allow for the customization of infrastructure to meet specific IoT requirements, optimizing performance, security, and regulatory adherence. The isolation of workloads further reduces the risk of data breaches or unauthorized access.

They are best suited for IoT deployments in industries such as medical devices, critical infrastructure, and defense systems, where tight control over data and security is essential.

Hybrid clouds

Hybrid clouds blend the best of both public and private clouds, offering flexibility and efficiency. In this model, sensitive workloads run in the private cloud, while less critical tasks, such as large-scale data analysis, are handled in the public cloud. This makes hybrid clouds ideal for IoT applications that need to balance security, performance, and scalability.

For IoT, **data segmentation** ensures that sensitive data stays in the private cloud, while non-sensitive data is processed in the public cloud, optimizing both cost and security. The model offers flexibility, allowing organizations to scale public cloud resources during high traffic while maintaining core IoT operations in the private environment. **Edge computing** integration allows IoT devices to process data locally, sending only key information to the cloud, and reducing latency.

Best suited for use cases such as smart manufacturing or smart grid systems, hybrid clouds enable quick local processing for real-time operations and cloud management for long-term analytics.

To ensure the successful deployment of your IoT applications, consider the following best practices:

- **Assess data sensitivity:** Evaluate the sensitivity of your IoT data before selecting a cloud model. For highly regulated environments such as healthcare, consider private or hybrid clouds to ensure compliance with privacy regulations.
- **Leverage edge computing:** For low-latency or real-time processing needs, use a hybrid cloud approach with edge computing. Processing data closer to the source reduces latency and bandwidth costs, sending only critical information to the cloud.
- **Prioritize security:** Apply strong encryption and authentication for IoT data, especially in public clouds. Protect IoT devices and data streams against threats with robust security measures.
- **Optimize for scalability:** Ensure your cloud infrastructure can scale efficiently to handle IoT data volumes. Public clouds offer elasticity, but hybrid solutions can balance cost and performance for large-scale deployments.
- **Monitor and optimize costs:** Use cloud monitoring tools to track usage and identify areas to reduce costs, such as scaling down unused resources or optimizing data storage configurations.
- **Plan for integration:** Choose a cloud platform that integrates seamlessly with your IoT devices and legacy systems. Hybrid models offer the flexibility to connect older systems while maintaining necessary security.

By following these best practices, organizations can ensure their IoT applications run securely, efficiently, and cost-effectively, maximizing the potential of IoT data analytics. Now that you've seen how different cloud deployment models can shape the security, scalability, and cost efficiency of your IoT infrastructure, it's time to explore how cloud-based AI services elevate these solutions to new heights. In the next section, we will examine how ML and other AI capabilities can supercharge IoT applications, enabling real-time insights, predictive analytics, and automated decision-making.

AI Services in Cloud Platforms

The role of AI in IoT systems extends beyond simple automation; it enables these systems to learn from historical data, adapt to changing conditions, and make informed decisions. This ability to self-learn and self-correct helps IoT systems operate more efficiently, whether in predictive maintenance, energy management, or healthcare.

For instance, in *smart manufacturing*, AI-powered IoT systems can predict equipment failures before they occur by analyzing sensor data from machines. AI algorithms detect anomalies in the performance data, allowing the system to schedule maintenance proactively, reducing downtime and improving operational efficiency. Similarly, in *smart cities*, AI and IoT integration enables intelligent traffic management systems, which monitor real-time traffic conditions and adjust signals dynamically to optimize traffic flow, reducing congestion and travel time.

AI's ability to process and analyze large datasets in real time allows IoT systems to generate data-driven insights that were previously impossible. ML models can identify trends, correlations, and outliers in data streams, providing actionable intelligence that helps organizations make informed decisions. AI models also facilitate deeper analysis, such as predicting future outcomes based on historical data, a capability that is critical for sectors such as energy, logistics, and healthcare.

For example, in *energy management systems*, AI-driven IoT platforms can analyze energy consumption patterns and optimize resource use. By monitoring data from connected devices such as thermostats, smart meters, and lighting systems, AI can recommend energy-saving measures, optimize peak load times, and predict future energy demands. Google's **DeepMind** AI has been successfully applied in managing energy at data centers, where it uses IoT data to predict power usage and optimize cooling systems, reducing energy consumption by up to 40%.

In the *healthcare sector*, AI and IoT are being used to monitor patients remotely through connected wearables and sensors. AI algorithms analyze real-time data from devices such as glucose monitors or heart rate sensors, detecting irregularities and alerting healthcare providers before serious complications arise. One real-world example is **Philips' HealthSuite**, an AI-powered IoT platform that connects medical devices, collects patient data, and provides insights to healthcare professionals, improving patient outcomes and streamlining treatment protocols.

In summary, AI is not only enhancing the performance and decision-making capabilities of IoT systems but also transforming industries by enabling smarter, more responsive, and autonomous operations. Through real-time data analysis, predictive maintenance, and self-optimizing capabilities, AI makes IoT systems smarter and more capable of handling complex, data-driven environments. In the following sections, we will explore how cloud platforms provide a suite of AI services, such as ML APIs, pretrained models, and model management tools, to further enhance IoT systems.

ML APIs and tools

ML APIs offered by cloud platforms are transforming how IoT systems analyze and process data. These APIs provide ready-made, scalable solutions that can be easily integrated into IoT applications to add intelligence and decision-making capabilities without requiring expertise in building ML models from scratch. Cloud providers offer a wide range of ML APIs and tools, from image and speech recognition to more complex tasks such as predictive analytics and anomaly detection. In this section, we will explore some of the popular AI/ML APIs.

Amazon Rekognition

Amazon Rekognition is a cloud-based image and video analysis service offered by AWS. It empowers developers to add sophisticated visual search, and discovery features to applications without requiring expertise in ML or computer vision. Leveraging advanced deep learning technologies, Amazon Rekognition can swiftly analyze images and videos stored in Amazon S3, making it easier to process vast amounts of visual data.

Amazon Rekognition is a HIPAA-eligible service, meaning it complies with the U.S. **Health Insurance Portability and Accountability Act (HIPAA)** of 1996 requirements for processing, storing, and transmitting **protected health information (PHI)**. This makes it suitable for healthcare applications that handle sensitive patient data and require stringent privacy and security measures.

It offers a comprehensive suite of image analysis features that enable applications to interpret and understand the content within images. It can detect and classify objects, scenes, and concepts; recognize text in multiple languages; identify celebrities; analyze facial attributes; filter out unsafe content; and assess image properties such as quality and color. Additionally, users can create custom labels to detect objects specific to their unique use cases:

- **Object, scene, and concept detection:** Detects and classifies objects, scenes, concepts, and celebrities in images
- **Text detection:** Recognizes printed and handwritten text in images across a variety of languages
- **Unsafe content detection:** Identifies and filters explicit, inappropriate, and violent content with granular labels
- **Celebrity recognition:** Recognizes tens of thousands of celebrities across different categories such as politicians, athletes, actors, and musicians
- **Facial analysis:** Detects, analyzes, and compares faces, providing attributes such as gender, age range, and emotions
- **Custom labels:** Allows the creation of custom classifiers to detect objects specific to business needs, such as logos or proprietary products
- **Image properties analysis:** Evaluates image attributes such as quality, color balance, sharpness, and contrast

In addition to performing all the image analysis features, Amazon Rekognition extends its capabilities to video content, enabling applications to extract valuable insights from videos. It can track individuals as they move through frames, perform face liveness detection to ensure real-time user verification, and automatically segment videos to identify key transitions such as black frames or end credits. These additional features provide richer context and temporal understanding that are unique to video analysis:

- **People pathing:** Tracks identified individuals as they move across video frames, helping analyze movement patterns and behaviors
- **Video segmentation:** Automatically identifies and segments parts of a video, such as black frames or end credits, streamlining video editing and processing
- **Face liveness detection:** Verifies whether a live user is present during face verification processes, helping prevent spoofing attacks using photos or pre-recorded videos

You can use Amazon Rekognition via its SDK; it supports CLI, Java, Python, Ruby, Node.js, PHP, .NET, and JavaScript. To use the SDK, you will need to create a credential file on your system and create and download your access key ID and secret access key. This credential file will also contain Region information. To work with Python, you will need to install `boto3`:

```
pip install boto3
```

Let us try detecting objects on the road, which can be extremely useful for smart vehicles and self-driving cars. In this example, we'll be leveraging Amazon Rekognition to help a vehicle identify different objects around it, such as cars, traffic signs, and pedestrians, all in real time. This is critical in AI-driven automation systems, especially when dealing with dynamic environments such as road traffic. Follow these steps:

1. Before we start, the first step is importing essential libraries for interacting with Amazon Rekognition and handling the image processing tasks:

```
import getpass
import boto3
from PIL import Image, ImageDraw, ImageFont
import matplotlib.pyplot as plt
```

2. We use this section to securely input the AWS credentials required to connect to Rekognition. You can either enter the credentials in real time using `getpass` or make them environment variables, which are accessed automatically by the script:

```
access_key=getpass.getpass('Enter your AWS Access Key ID: ')
secret_key=getpass.getpass('Enter your AWS Secret Access Key:')
region = 'ap-south-1'
```

3. Next, we create a session with AWS and instantiate the Rekognition client:

```
session = boto3.Session(
    aws_access_key_id=access_key,
    aws_secret_access_key=secret_key,
    region_name=region
)

rekognition = session.client('rekognition')
```

4. We then capture the image and prepare it for analysis. In this case, we read the image and convert it to bytes. Here, we assume that a camera or external system is capturing an image, stored as `file_name`:

```
with open(file_name, 'rb') as image_file:
    image_bytes = image_file.read()
```

5. We now send the image bytes to Amazon Rekognition for object detection. We call `detect_labels()` from the Rekognition client, which identifies objects in the image. The `MaxLabels` parameter limits the number of labels returned, and `MinConfidence` ensures that only objects detected with a high confidence level are considered:

```
response = rekognition.detect_labels(
    Image={'Bytes': image_bytes},
    MaxLabels=10,
    MinConfidence=75
)

print('Detected labels in ' + file_name)
for label in response['Labels']:
    print(f"{label['Name']} : {label['Confidence']:.2f}%")
```

6. Now, we visualize the output by drawing bounding boxes around the detected objects. A red box is drawn around each detected object, and the label name is displayed above the box:

```
draw = ImageDraw.Draw(image)
img_width, img_height = image.size

for label in response['Labels']:
    for instance in label.get('Instances', []):
        box = instance['BoundingBox']
        left = img_width * box['Left']
        top = img_height * box['Top']
        width = img_width * box['Width']
        height = img_height * box['Height']
```

```
points = (  
    (left, top),  
    (left + width, top),  
    (left + width, top + height),  
    (left, top + height),  
    (left, top)  
)  
  
# Draw the bounding box  
draw.line(points, fill='red', width=3)  
  
# Draw the label name  
draw.text((left, top - 10), label['Name'], fill='red')  
  
del draw
```

7. Here is the image with proper labels:



Figure 9.1: Traffic image with detected objects in red bounding boxes

This approach can be extended to real-time applications such as self-driving cars or smart vehicles.

Other useful Amazon services for IoT and AI

While we've explored object detection using Amazon Rekognition, AWS offers several other powerful services tailored to specific IoT and AI applications. Let's look at two noteworthy examples that can enhance IoT-driven projects beyond just visual data processing.

Amazon Lookout for Equipment – predictive maintenance for industrial IoT

Amazon Lookout for Equipment is an API that focuses on predictive maintenance for industrial IoT applications. It uses ML to analyze sensor data from equipment, detecting abnormal behavior and predicting potential equipment failures before they occur. In large industrial setups, machinery downtime can be extremely costly, and this API helps prevent unexpected breakdowns.

Consider an industrial factory that uses this API to monitor machines on a production line. The sensors embedded in the machines continuously generate data such as vibration, temperature, and pressure. Lookout for Equipment analyzes this data in real time, detecting early signs of wear and tear or unusual patterns in machine behavior. By identifying these anomalies, the system can automatically schedule maintenance well before a machine breaks down, significantly reducing downtime and maintenance costs.

This service is invaluable for any industrial IoT solution where equipment monitoring and preventing unplanned downtime are critical. By predicting failures, the API allows manufacturers to optimize their maintenance schedules and enhance operational efficiency.

Update on Amazon Lookout for Equipment

At the time of writing this book, Amazon Lookout for Equipment is available for use. However, it is worth noting that Amazon will discontinue offering the service to new customers starting October 17, 2024. Existing customers will still have access via both the AWS Management Console and API, with AWS continuing to invest in security, availability, and performance improvements. However, no new features will be introduced for this service.

For new users looking for predictive maintenance solutions, Amazon suggests exploring the AWS Partner Network, where multiple partners offer specialized solutions. These can be found on the **Asset Maintenance and Reliability** page in the AWS Solutions Library, providing a quick *time to value* without needing extensive expertise in predictive maintenance.

Alternatively, for users who prefer building their own solutions, Amazon SageMaker offers a comprehensive suite of tools to help develop, train, and deploy ML models for predictive maintenance with fully managed infrastructure and workflows.

Amazon Fraud Detector – adapting to IoT anomaly detection

Although primarily designed for fraud detection, Amazon Fraud Detector can be adapted for anomaly detection in IoT scenarios where financial or transactional data is being processed. It uses ML models to identify suspicious behavior, making it highly useful in situations where IoT devices handle transactional data, such as in smart payment systems or financial applications.

Imagine a smart **point-of-sale (POS)** IoT device deployed in retail stores. The device processes hundreds of transactions daily. By integrating Amazon Fraud Detector, the system can monitor purchase patterns and detect anomalies, such as unusually high purchase amounts or multiple transactions within a short time frame. If any transaction deviates from normal behavior, the API can flag it for further investigation, preventing fraudulent activities.

For IoT systems handling sensitive financial data, such as smart payment terminals or connected ATMs, ensuring the security of transactions is crucial. Amazon Fraud Detector adds a layer of security, detecting fraudulent activities by monitoring financial anomalies in real time.

Using the Azure Speech SDK in Python to capture speech commands

The Azure AI Services Speech SDK allows you to integrate speech recognition into your applications. Here's how you can set up and use the Speech SDK to take speech commands in Python:

1. To begin, you'll need an Azure subscription. If you don't already have one, you can create a free account. Afterward, create a Speech service resource in the Azure portal to obtain your subscription key and service Region. For development, ensure that Python 3.6 or higher is installed on your system. Additionally, make sure the device running your script has access to a microphone, and that permissions are configured for it to function properly. This setup is essential for speech-related projects using Azure services.

2. Make sure you have installed the Azure Speech SDK:

```
pip install azure-cognitiveservices-speech
```

3. We kick things off by importing the Azure AI Services Speech SDK. This package contains all the tools we'll need for speech recognition:

```
import azure.cognitiveservices.speech as speechsdk
```

4. Next, we need to provide our Azure Speech service credentials. This includes our subscription key and the Region where our service is hosted. This step ensures our program can connect to Azure to process the speech:

```
speech_key = "YOUR_SPEECH_KEY"
service_region = "YOUR_SERVICE_REGION"
speech_config = speechsdk.SpeechConfig(
    subscription=speech_key,
    region=service_region
)
```


5. Now that we've set up the service, we create a speech recognizer. This will listen to what we say through the microphone:

```
speech_recognizer = speechsdk.SpeechRecognizer(  
    speech_config=speech_config  
)
```

6. Once the recognizer is ready, it waits for us to speak into the microphone. When we do, it will try to recognize what we've said. The `recognize_once()` method listens for a single utterance and processes it:

```
print("Speak into your microphone.")  
result = speech_recognizer.recognize_once()
```

7. Here's where we check whether our speech was recognized successfully or whether there were issues. We handle three cases—recognized speech, no match, or cancellation due to an error:

```
if result.reason == speechsdk.ResultReason.RecognizedSpeech:  
    print("Recognized: {}".format(result.text))  
    process_command(result.text)  
elif result.reason == speechsdk.ResultReason.NoMatch:  
    print("No speech could be recognized: {}".format(  
        result.no_match_details))  
elif result.reason == speechsdk.ResultReason.Canceled:  
    print("Speech Recognition canceled: {}".format(  
        cancellation_details.reason))
```

8. Now, we're adding a bit of fun! This function takes what we said and checks whether it's a valid command, such as "turn on the light." Based on that, it either prints the action or informs us that the command is not recognized:

```
def process_command(command_text):  
    if "turn on the light" in command_text.lower():  
        print("Turning on the light.")  
    elif "turn off the light" in command_text.lower():  
        print("Turning off the light.")  
    else:  
        print("Command not recognized.")
```

9. Finally, we wrap everything up by calling our `recognize_speech()` function when we run the script:

```
if __name__ == "__main__":  
    recognize_speech()
```

In the `process_command` function, you can expand the code to interact with your IoT devices. For instance, you can send messages to IoT devices via Azure IoT Hub, **Message Queuing Telemetry Transport (MQTT)** or any communication protocol that your devices support. This approach can be used in various IoT applications, such as **smart lighting**, where voice commands can turn lights on or off.

For example, smart light bulbs (such as **Philips Hue** or **LIFX**) can be controlled using this method. The required tech stack would include **Azure IoT Hub**, the **MQTT protocol**, and integration with IoT edge devices such as **Raspberry Pi** or **ESP8266**. This allows voice-controlled automation for a wide range of IoT devices such as thermostats, security cameras, and more.

Pretrained models in the cloud

When it comes to IoT applications, building ML models from scratch can be time-consuming and resource intensive. This is where pretrained models come in, offering a shortcut to advanced AI functionality. These models have already been trained on large datasets by cloud service providers, which means you can take advantage of their capabilities without going through the hassle of training them yourself. You just plug them into your IoT system, tweak them if needed, and you're ready to go!

Let's face it, IoT devices generate massive amounts of data, and analyzing that data in real time requires some serious computing power. By leveraging pretrained models, you can focus on solving problems rather than building complex models. Here are some of the key benefits of using pretrained models in IoT applications:

- **Faster deployment:** Pretrained models are ready to use right out of the box. Whether you're dealing with image recognition, anomaly detection, or speech processing, these models save time by eliminating the need for training on your own data.
- **Cost-effective:** Training a model from scratch requires vast computational resources. With pretrained models, that heavy lifting has already been done. This means you can minimize the compute costs and still get high-quality results.
- **Access to cutting-edge technology:** Cloud platforms such as Google Cloud and AWS offer pretrained models that are built by some of the brightest minds in AI. You get to tap into this technology without needing a team of data scientists.
- **Customizability:** Many pretrained models can be fine-tuned for your specific use case. You can take a general-purpose model and adapt it to your IoT application without needing to start from scratch.

Let's dive into a couple of the popular cloud platforms and the pretrained models they offer, specifically for IoT applications:

- Google Cloud's **TensorFlow Hub** is a comprehensive repository of pretrained models for tasks such as image classification, object detection, and text analysis. Models such as MobileNet for efficient image classification and SSD MobileNet for object detection are readily available.

For example, a smart security system could use the pretrained SSD MobileNet V2 model from TensorFlow Hub to recognize intruders in real time, enabling immediate alerts and responses without the latency of cloud processing.

- On AWS, **SageMaker JumpStart** provides a rich library of pretrained models that can be deployed directly for various tasks, including image classification, natural language processing, and time-series forecasting. Models such as ResNet50 for image recognition and BERT for language understanding are accessible through SageMaker. For instance, a smart agriculture system could utilize a pretrained ResNet50 model to monitor crops for signs of disease, analyzing images captured by drones or sensors to detect anomalies and notify farmers promptly.
- Microsoft **Azure's AI Services** offers a suite of pretrained models across vision, speech, language, and decision-making tasks through easy-to-use APIs. Services such as the Computer Vision API and the Speech-to-Text API enable the integration of AI into IoT devices without the need for custom model training. For example, a smart assistant in a retail environment can leverage the Azure Speech-to-Text API to process voice commands from customers, facilitating hands-free interactions and improving customer service.

By utilizing these pretrained models, developers can significantly reduce development time and focus on integrating AI capabilities into their IoT applications. These models have been trained and optimized for performance, making them ideal for deploying in environments where computational resources may be limited. Incorporating pretrained models such as Inception V3 and **You Only Look Once (YOLO)** for object detection, or **Large Language Models (LLMs)** such as T-5, GPT-2, and so on for text generation can enhance IoT solutions, providing advanced functionalities such as image recognition, real-time analytics, and natural language understanding. We've just walked through how cloud-based AI services can supercharge IoT solutions with real-time insights and predictive analytics. Next, let's explore how to train and deploy AI models for IoT on cloud platforms. We'll look at how to tailor pretrained models—or build them from scratch—to give your IoT applications a distinct competitive edge.

Training and Deploying AI Models for IoT on Cloud Platforms

Have you ever felt that pretrained AI models are almost perfect but not quite tailored to your specific IoT needs? While these models are powerful and can save you a significant amount of time, sometimes you need something more customized. That's when training your own model becomes a game-changer. In this section, we'll delve into how you can take that crucial step and the tools that make it possible.

AI cloud platforms such as Google Vertex AI, Google AutoML, Amazon SageMaker, Amazon Bedrock, and their counterparts in Microsoft Azure provide robust tools to simplify this process. They offer advanced ML services that help businesses extract valuable insights from IoT data. These platforms empower IoT applications with AI capabilities that enable systems to learn, adapt, and make decisions autonomously, thereby improving efficiency, reliability, and user experience.

Let's explore how each of these leading cloud platforms can elevate your IoT projects, complete with technical details to guide you.

Google Vertex AI

Google Vertex AI is an end-to-end ML platform that unifies Google Cloud's existing ML offerings into a cohesive experience. It allows you to build, deploy, and scale AI models effortlessly (*Figure 9.2*). Vertex AI combines the user-friendliness of AutoML with the flexibility of custom training, making it suitable for both novices and experienced data scientists.

Here are some of the key technical features of Vertex AI:

- **Unified UI and API:** Manage all ML workflows through a single interface
- **Custom model training:** Use Jupyter notebooks integrated with BigQuery and Cloud Storage
- **AutoML capabilities:** Automate the training of high-quality models with minimal effort
- **Hyperparameter tuning:** Optimize models using Google's Vizier service
- **Edge deployment:** Export models to TensorFlow Lite or Coral Edge TPU for edge devices

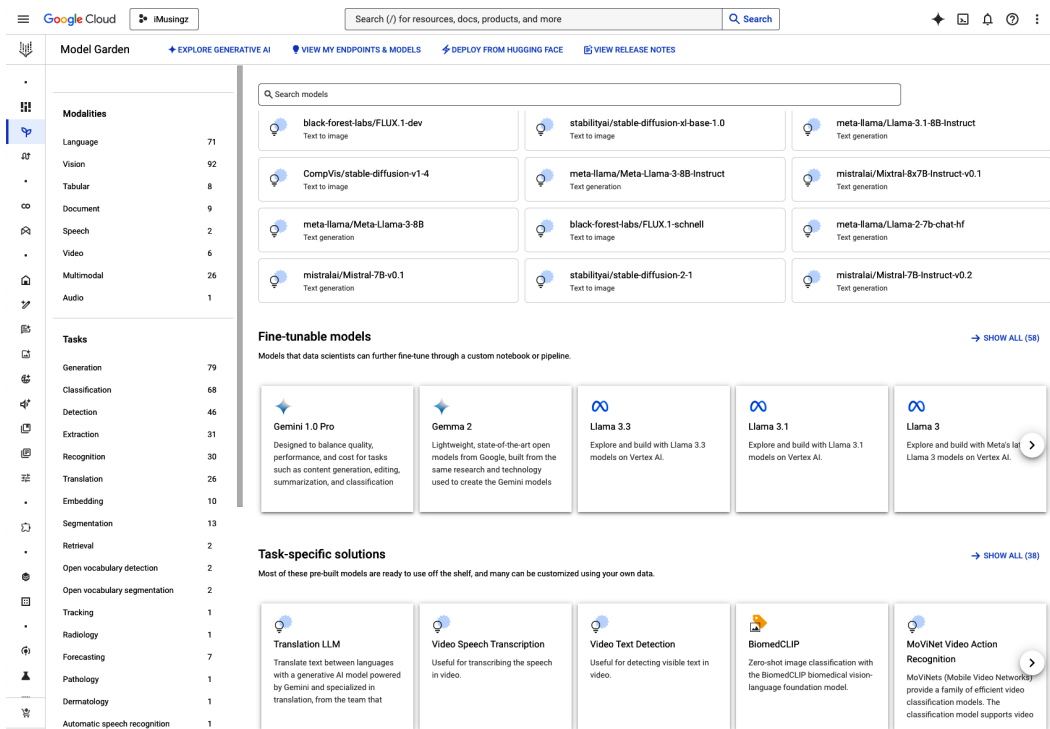


Figure 9.2: Model Garden available in Vertex AI

Imagine you're running a fleet of autonomous drones for agricultural monitoring. With Vertex AI, you can train a custom image classification model to identify crop diseases from aerial images. Using **TensorFlow Extended (TFX)** pipelines, you can preprocess data, train the model, and deploy it at scale. Vertex AI's ability to handle large datasets and complex models ensures your drones make accurate, real-time decisions while covering vast agricultural fields.

Google AutoML

Google AutoML is a suite within Vertex AI that democratizes the creation of custom ML models. It automates the model training process, making it accessible even if you lack extensive ML expertise.

The key technical features of AutoML include the following:

- **AutoML Vision, Natural Language, Translation, Tables, and Video:** These are specialized services for different data types
- **Transfer learning:** Build on top of Google's pretrained models
- **Simple interface:** Upload data, train models, and evaluate performance through an intuitive UI
- **Model export:** Deploy models to edge devices or mobile platforms

Suppose you're developing a smart retail system that analyzes customer behavior through in-store cameras. Using AutoML Vision, you can train a model to recognize shopping patterns, such as which aisles attract the most customers. The process involves uploading your video data, labeling it accordingly, and letting AutoML handle the rest. The resulting model can then be integrated into your IoT system to provide real-time analytics and insights.

Amazon SageMaker

Amazon SageMaker is a fully managed service that equips developers and data scientists with the tools to build, train, and deploy ML models at scale.

Key technical features of Amazon's SageMaker are as follows:

- **SageMaker Studio:** An integrated development environment for ML
- **Built-in algorithms:** Access to optimized algorithms for common tasks
- **SageMaker Autopilot:** Automates model creation while allowing full control and visibility
- **Distributed training:** Support for training models across multiple GPUs and instances
- **Edge deployment:** Use SageMaker Neo to optimize models for various hardware platforms

Consider a manufacturing plant aiming to reduce equipment downtime. By leveraging SageMaker's built-in anomaly detection algorithms, you can analyze sensor data from machinery to predict failures before they occur. Using SageMaker's distributed training, you can process large volumes of

data quickly. The trained model can then be deployed on AWS IoT Greengrass devices for real-time monitoring and alerts, enhancing operational efficiency.

Amazon Bedrock

Amazon Bedrock is a new service that offers access to pretrained foundation models from leading AI companies, including Amazon's own models.

Amazon's Bedrock has some cool technical features, such as the following:

- **Foundation models access:** Utilize models such as Jurassic-2, Claude, and Amazon's Titan
- **Customization:** Fine-tune these models with your data securely
- **Serverless experience:** No need to manage infrastructure
- **Integration with AWS services:** Easily incorporate models into applications using AWS tools

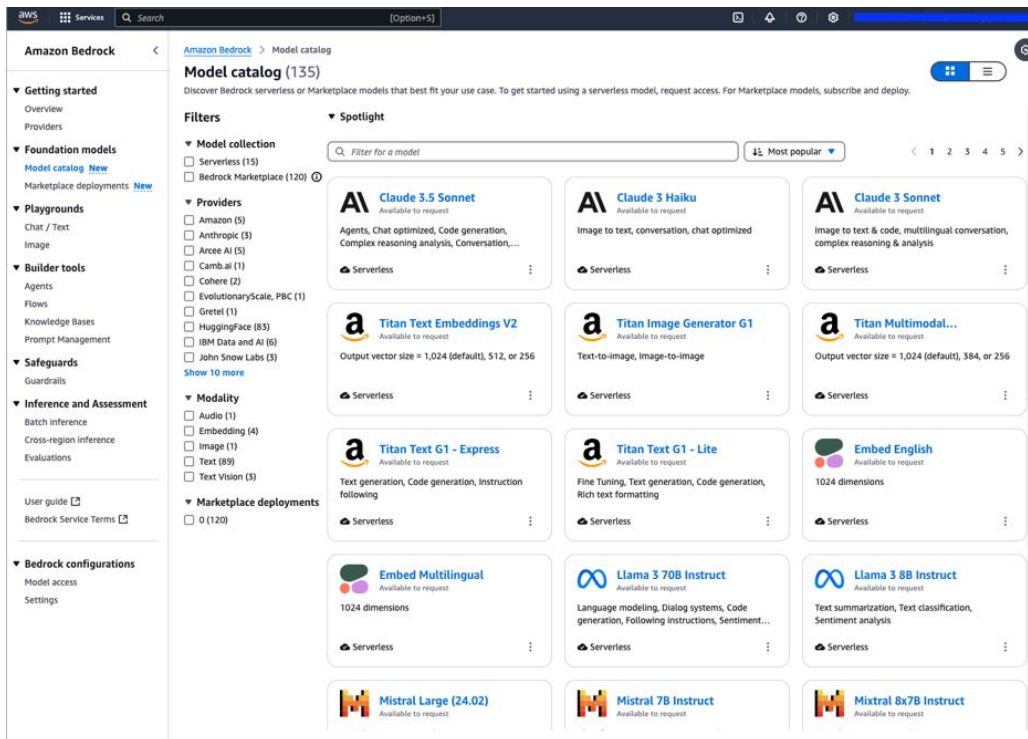


Figure 9.3: Available models in Amazon Bedrock

Suppose you're developing a voice-activated assistant for smart homes that requires advanced natural language understanding. With Amazon Bedrock, you can fine-tune a foundation model such as Claude to recognize and process complex voice commands specific to home automation. By integrating this

model with AWS IoT Core, your assistant can control devices, schedule tasks, and provide information, all while learning from user interactions to improve over time.

Security and Privacy in IoT Cloud Platforms

The proliferation of IoT devices has revolutionized industries by enabling real-time data collection and analytics, but it also brings significant security and privacy challenges when integrated with cloud platforms. This section delves into these challenges, explores strategies to ensure data privacy, examines the security features of major cloud providers such as AWS, Azure, and Google Cloud, and outlines best practices for securing IoT deployments on the cloud.

Securing IoT data is a complex endeavor due to the inherent characteristics of IoT ecosystems and their integration with cloud computing. Some of the reasons for this complexity are as follows:

- **Heterogeneity of devices:** IoT ecosystems consist of a vast array of devices with varying hardware capabilities, operating systems, and communication protocols. This heterogeneity makes it difficult to implement uniform security measures across all devices. Many IoT devices have limited processing power and storage capacity, restricting the implementation of robust security algorithms.
- **Massive scale and connectivity:** The sheer number of interconnected devices increases the attack surface, providing more entry points for potential attackers. Continuous connectivity and data transmission between devices and cloud services can be exploited to intercept or manipulate data.
- **Weak authentication and authorization:** Many IoT devices lack strong authentication mechanisms, making them susceptible to unauthorized access. Default passwords and lack of user authentication protocols can lead to device compromise, allowing attackers to gain control over devices and the sensitive data they handle.
- **Data privacy concerns:** IoT devices collect vast amounts of personal and sensitive data. Ensuring that this data is securely transmitted and stored is a significant challenge. Data breaches can lead to privacy violations, financial losses, and legal repercussions.
- **Firmware and software vulnerabilities:** Outdated firmware and software can contain vulnerabilities that attackers may exploit. Many IoT devices do not support over-the-air updates, making it difficult to patch security flaws promptly.

Protecting the privacy of data on cloud platforms is paramount, requiring robust measures to prevent unauthorized access and data breaches. There are various strategies that we can implement to protect sensitive data in the cloud. Here are a few such strategies:

- **Data encryption:** Encrypting data both at rest and in transit is crucial for maintaining privacy. Strong encryption algorithms such as AES-256 can be used to secure data stored in the cloud. Implementing **Transport Layer Security (TLS)** protocols protects data during transmission between IoT devices and cloud services.

- **Access control mechanisms:** Implementing robust access control policies ensures that only authorized entities can access sensitive data. **Role-based access control (RBAC)** and **attribute-based access control (ABAC)** models can define permissions based on user roles and attributes.
- **Anonymization and pseudonymization:** Anonymizing or pseudonymizing sensitive data helps protect individual identities. This involves removing or encrypting **personally identifiable information (PII)** before storing or processing the data in the cloud.
- **Compliance with regulations:** Compliance with data protection regulations such as GDPR, HIPAA, and CCPA is essential. Cloud platforms often provide tools to help organizations adhere to these regulations, such as data residency options and comprehensive audit logs.
- **Secure APIs and communication protocols:** Using secure communication protocols such as MQTT over TLS and HTTPS ensures that data transmitted between devices and cloud services is protected. APIs should be secured using authentication tokens and implement rate limiting to prevent abuse.

Security features of major cloud providers

Major cloud providers offer a range of security features tailored to IoT deployments, helping organizations protect their data and devices. Let us explore how leading cloud services enhance IoT security.

Amazon

Amazon provides a range of specialized services designed to enhance the security, management, and communication of IoT devices, ensuring that IoT ecosystems are scalable, efficient, and secure. Here's a look at some of the key services AWS offers for IoT applications:

- **AWS IoT Device Defender:** This service provides ongoing security management for IoT devices. It helps audit configurations and monitor devices to detect abnormal behavior using ML and rule-based detection. It also offers automatic alarms and actionable insights for securing IoT devices in real time (<https://aws.amazon.com/iot-device-defender/>).
- **AWS IoT Device Management:** This service allows for secure onboarding, organizing, monitoring, and managing IoT devices at scale. You can perform actions such as firmware updates, device reboots, and troubleshooting, all remotely, which is essential for IoT systems that require constant and scalable device management (<https://aws.amazon.com/iot-device-management/>).
- **AWS IoT Core:** AWS IoT Core provides secure communication between IoT devices and the cloud. It uses mutual authentication and encryption to ensure that only trusted devices can send data, making it fundamental for secure IoT data exchange (<https://aws.amazon.com/iot-core/>).

- **AWS Key Management Service (KMS):** AWS KMS offers encryption key management, which is critical for protecting sensitive data stored and transmitted in IoT applications. KMS integrates with many AWS services, allowing for seamless data security across various cloud resources.
- **AWS Identity and Access Management (IAM):** IAM helps define fine-grained permissions and access controls, ensuring that only authorized individuals and devices can access specific IoT resources and data. This is crucial for maintaining security in IoT ecosystems.

Microsoft Azure

Microsoft Azure offers a comprehensive set of security features to help safeguard IoT devices, data, and communications. These services are designed to provide end-to-end protection, from device-level security to managing access control and monitoring threats in real time. Here are some of the key security solutions offered by Azure for IoT environments:

- **Azure Sphere:** Azure Sphere delivers end-to-end security for IoT devices by combining a secured **microcontroller unit (MCU)**, a Linux-based operating system, and cloud services. This integration ensures device security throughout its lifecycle. It's particularly useful for industries requiring robust security for connected devices, from smart factories to consumer electronics (<https://azure.microsoft.com/en-us/products/azure-sphere>).
- **Azure IoT Hub:** Azure IoT Hub enables secure, bidirectional communication between IoT devices and the cloud. It supports per-device authentication and access control, allowing you to manage billions of devices securely. It also integrates with Azure Event Grid and IoT Edge for hybrid applications, making it a scalable solution for large IoT deployments.
- **Azure Security Center for IoT:** This service provides unified security management and advanced threat protection for IoT environments. It continuously monitors the IoT security posture, detects threats, and offers recommendations for improving security. This service is crucial for protecting IoT systems against evolving cyber threats (<https://learn.microsoft.com/en-us/azure/iot-hub/>).
- **Azure RBAC:** Azure RBAC helps implement fine-grained access control, allowing administrators to define permissions based on roles. This ensures that users or services have only the access they need, reducing the potential for security breaches (<https://learn.microsoft.com/en-us/azure/role-based-access-control/overview>).
- **Azure Entra ID (formerly Azure Active Directory):** Azure Entra ID provides identity and access management, ensuring that only authorized users can access IoT resources. It's a vital service for managing secure access to both IoT devices and related cloud resources, further enhancing security across the ecosystem.

Here is an informative blog on different ways to secure your Azure services: <https://www.microsoft.com/en-us/security/blog/2024/01/10/5-ways-to-secure-identity-and-access-for-2024/>.

Google

Google Cloud offers robust security tools designed to protect IoT deployments, ensuring device security, secure communication, and data protection across the cloud. These services provide essential safeguards for IoT systems, enhancing their resilience to threats. Key offerings include the following:

- **Cloud IoT Core:** Ensuring secure device connections, Cloud IoT Core supports industry-standard protocols such as MQTT and HTTP, with TLS encryption. It facilitates bidirectional communication between IoT devices and Google Cloud, enabling secure data transfer, device authentication, and authorization across the cloud infrastructure (<https://cloud.google.com/architecture/connected-devices/iot-platform-product-architecture>).
- **Cloud IAM:** Google Cloud IAM offers fine-grained permissions, allowing precise control over which users and devices can access specific resources. This is essential for implementing least privilege principles in large IoT ecosystems, ensuring only authorized devices and users have the right access (<https://cloud.google.com/iam/docs/>).
- **Vulnerability scanning and threat detection:** Google Cloud's Security Scanner integrates with IoT deployments to detect vulnerabilities in infrastructure and applications. This enables proactive risk mitigation, helping to identify potential threats before they can cause harm, and providing an added layer of security to IoT environments (<https://cloud.google.com/security-command-center/docs/concepts-vulnerabilities-findings>).
- **Encryption by default:** Google Cloud ensures data protection by encrypting data both at rest and in transit. AES-256 is used for encrypting stored data, while TLS secures data during transfer, maintaining confidentiality and integrity throughout IoT communication (an interesting read on how Google ensures data safety can be found here: <https://services.google.com/fh/files/misc/google-workspace-encryption-wp.pdf>).

Ensuring the security and privacy of IoT data is critical for building trust and maintaining system integrity, but it's also essential to look ahead. In the next section, we will explore groundbreaking developments—from edge AI to quantum computing—that promise to reshape how IoT ecosystems are designed, deployed, and secured.

Emerging Trends and the Future of AI Cloud Platforms for IoT

As we navigate the ever-evolving landscape of technology, it's thrilling to witness how AI and the IoT are converging to redefine our world. Let's explore some of the groundbreaking trends shaping the future of AI cloud platforms for IoT and how they might impact us all.

Edge AI and hybrid cloud for IoT

Imagine your smart devices not just collecting data but also processing and acting on it instantly, right where it's generated. This is the power of edge AI. By bringing computation and intelligence to the edge of the network—closer to the IoT devices themselves—we reduce latency, enhance privacy, and improve real-time decision-making.

But here's where it gets even more exciting: combining edge processing with cloud AI creates a hybrid cloud IoT model. This synergy allows for immediate data processing on the edge while leveraging the vast computational resources of the cloud for more complex analyses. For instance, in a smart factory, edge devices can monitor equipment in real time to prevent failures, while the cloud analyzes long-term data to optimize production processes. This blend offers the best of both worlds—speed and efficiency at the edge and powerful analytics and storage in the cloud.

A Hybrid Edge-Cloud System for Networking Service Components Optimization Using the Internet of Things

This paper proposes a hybrid edge-cloud system for optimizing networking service components in IoT environments. It introduces the IoT-SCOM model, which combines edge and cloud computing to improve performance, reduce latency, and enhance resource utilization. The system consists of three layers: IoT, Mobile Edge Computing, and Mobile Cloud Computing, offering scalability and efficiency for data-intensive applications (<https://www.mdpi.com/2079-9292/12/3/649>).

Let's look at quantum computing next.

Quantum computing in IoT

Now, let's take a leap into the future with quantum computing in IoT. Quantum computers operate on principles vastly different from classical computers, enabling them to process complex computations at unprecedented speeds. For IoT, this means the potential to analyze massive datasets from countless devices almost instantaneously.

Think about the impact this could have on areas such as climate modeling or financial forecasting, where processing vast amounts of data quickly is crucial. Quantum computing could revolutionize IoT data analytics by providing deeper insights and solving problems that are currently beyond our reach. While still in the experimental stage, the fusion of quantum computing and IoT holds promise for breakthroughs we can only begin to imagine.

Quantum computing applications for Internet of Things

This paper explores how quantum computing can revolutionize IoT in areas such as network optimization, endpoint computation, security, and sensor precision. Quantum computing can improve IoT networks' efficiency by optimizing data transmission, enhance security with quantum encryption, and offer faster data processing at endpoints. The paper also discusses emerging applications such as quantum digital marketing and quantum-secured smart locks, while highlighting challenges such as the current limitations in quantum hardware and integration with IoT infrastructure (<https://ietresearch.onlinelibrary.wiley.com/doi/epdf/10.1049/qtc2.12079>).

As we stand on the cusp of these innovations, it's clear that the intersection of AI and IoT is set to transform our world profoundly. Embracing these emerging trends not only fuels technological advancement but also opens up new possibilities for solving some of the most pressing challenges we face today. It's an exciting time to be part of this journey, and I can't wait to see where it takes us next!

Cloud computing is a vast landscape, and in this chapter, we have only been able to cover the surface. However, I hope the links shared throughout will help those of you who are interested in exploring more deeply into this exciting domain.

Summary

This chapter provided a comprehensive exploration of how cloud computing empowers IoT, addressing both the challenges and solutions that arise in this dynamic field. It began by introducing the fundamental concepts of cloud computing and its critical role in overcoming key IoT challenges, such as data storage, processing power, and scalability.

We delved into the basics of cloud computing, explaining the different types of cloud services—IaaS, PaaS, and SaaS—and how each service model caters to various IoT needs. The discussion extended to cloud deployment models, including public, private, and hybrid clouds, highlighting their respective advantages and applications in IoT deployments.

The chapter outlined best practices for deploying IoT applications, providing practical guidance to optimize performance and security. We then shifted focus to the integration of AI services in cloud platforms, examining essential ML APIs and tools that enhance IoT functionalities. A detailed look at Amazon Rekognition demonstrated how cloud-based AI can be harnessed for image and video analysis in IoT solutions.

Further, we explored other valuable Amazon services such as Amazon Lookout for Equipment for predictive maintenance in industrial IoT and Amazon Fraud Detector for anomaly detection. A hands-on section guided you through using the Azure Speech SDK in Python to capture speech commands, showcasing the implementation of voice control in IoT devices.

Addressing the crucial aspect of security and privacy in IoT cloud platforms, the chapter reviewed the security features of major cloud providers—Amazon, Microsoft Azure, and Google—emphasizing strategies to protect IoT data and infrastructure.

Finally, we looked ahead to the emerging trends and future of AI cloud platforms for IoT, discussing the rise of edge AI and hybrid cloud models that combine edge processing with cloud capabilities. The potential impact of quantum computing in IoT was also considered, highlighting how it could revolutionize data analytics and processing speeds in IoT ecosystems.

In the next chapter, we will move towards time series data and understand how to build AI models around them.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/ds>



Deep Learning for Time-Series Data from IoT

In *Chapters 3 and 4*, we laid the groundwork for understanding **Machine Learning (ML)** and **Deep Learning (DL)**, diving into various algorithms and neural network architectures that form the backbone of modern AI solutions. Specifically, we explored **Recurrent Neural Networks (RNNs)** and **Long Short-Term Memory (LSTM)** networks as powerful tools for handling sequential data due to their ability to capture time-dependent patterns. Building on that foundation, this chapter focuses specifically on applying these DL models to time-series data generated from IoT devices.

Time-series data is at the heart of IoT applications, driving everything from the real-time monitoring of industrial equipment to predictive maintenance in smart cities. Given the volume and velocity of this data, scalable approaches are crucial. This chapter not only revisits RNNs and LSTMs, applying them to IoT data, but also introduces SparkML, a framework designed to handle large-scale data processing. By leveraging SparkML, we can efficiently manage the extensive time-series data produced in IoT environments.

In this chapter, we're going to cover the following main topics:

- Introduction to Time-Series Data
- The Importance of Time-Series Analysis in IoT
- Working with Large Time-Series Data Using SparkML
- Review of RNNs and LSTMs for Time-Series Analysis
- Pretrained Models for Time-Series Forecasting

Technical requirements

To run the codes in this chapter you will require to download the Spark ML library and Meta's Prophet library using these commands:

```
pip install pyspark
python -m pip install prophet
```

The code in the chapter are available on GitHub at the URL: <https://github.com/PacktPublishing/Hands-On-Artificial-Intelligence-for-IoT-Second-Edition/tree/main/Chapter10>

Introduction to Time-Series Data

In the **Internet of Things (IoT)**, data drives decision-making and fuels innovation. Among the diverse types of data that IoT systems generate, time-series data is particularly critical. This chapter explores the definition, characteristics, and significance of time-series data in IoT, setting the foundation for advanced analysis and applications.

What is time-series data?

Time-series data is a collection of data points recorded over time at consistent intervals, each associated with a timestamp indicating the moment of observation. While the intervals are typically regular (e.g., hourly or daily), irregularities can occur due to network disruptions or sensor failures.

The defining feature of time-series data is its temporal dimension—the order of data points matters. This aspect captures how variables change over time, enabling the analysis of trends, seasonality, and other temporal patterns. Time-series data can be categorized as follows:

- **Univariate:** Single variable measured over time (e.g., temperature)
- **Multivariate:** Multiple variables recorded simultaneously (e.g., temperature, humidity, and pressure), common in IoT systems where sensors measure various parameters at each time step

How is time-series data different?

Time-series data has several unique attributes compared to other data types:

- **Temporal dependency:** Unlike cross-sectional data, which represents a snapshot of variables at a single point, time-series data shows dependency across time steps. The value at any time point is often influenced by previous values, introducing autocorrelation. This temporal dependency necessitates specialized analysis techniques, such as the **Autoregressive Integrated Moving Average (ARIMA)**, LSTM networks, and newer models such as the **Temporal Fusion Transformer (TFT)**.

- **Seasonality and trends:** Time-series data often exhibits recurring seasonal patterns (daily, weekly, or yearly) and long-term trends (upward or downward movements over extended periods). Recognizing these patterns is crucial for accurate forecasting and anomaly detection. Advances in decomposition techniques, such as **Seasonal-Trend decomposition using Loess (STL)**, help isolate these components for better analysis.
- **Irregular observations:** While data is typically collected at fixed intervals, real-world factors such as sensor malfunctions can cause gaps or irregularities. Techniques such as Kalman filters or DL models for imputation, such as those based on **Generative Adversarial Networks (GANs)**, are often employed to address these challenges, preserving the integrity of the time series.

Time-series data is integral to various IoT applications across industries, offering predictive insights and enabling automation. Here are some key use cases:

- **Predictive maintenance:** Sensors monitor equipment health, collecting time-series data that can be analyzed to detect anomalies and predict failures before they occur. Techniques such as anomaly detection with autoencoders help IoT (*Chapter 7*) systems schedule maintenance more efficiently, reducing downtime and cutting costs.
- **Energy management:** Smart grids and industrial IoT setups generate time-series data on energy consumption. Analysis of this data helps optimize energy usage, predict demand, and implement energy-saving strategies using methods such as time-series clustering and reinforcement learning-based optimization.
- **Environmental monitoring:** IoT sensors in agriculture, forests, and urban areas track variables such as temperature, humidity, and air quality. Time-series analysis aids in precision agriculture, early warnings for natural disasters, and sustainable resource management. Recent studies on edge computing in environmental monitoring have demonstrated the benefits of processing time-series data close to the source, minimizing latency.
- **Healthcare monitoring:** Wearables and medical sensors produce time-series data on vital signs and activity levels. This data is crucial for real-time health monitoring, anomaly detection, and personalized healthcare interventions. Advances in AI, such as continuous monitoring with DL models such as **Gated Recurrent Unit (GRU)** networks, have enabled more accurate and timely insights from this data.

Understanding time-series data's characteristics is key to harnessing the potential of IoT technologies. In the following chapters, we will dive into techniques for preprocessing, analyzing, and modeling time-series data, transforming raw data into actionable insights that drive intelligent decision-making in IoT systems.

The Importance of Time-Series Analysis in IoT

Time-series analysis plays a crucial role in unlocking the full potential of IoT systems. By examining patterns, trends, and relationships within time-series data, organizations can gain valuable insights

and make data-driven decisions. Two key aspects highlight the importance of time-series analysis in IoT: real-time monitoring and predictive analytics.

Real-time monitoring

IoT devices generate continuous streams of time-series data, making real-time monitoring an indispensable feature for critical industries such as manufacturing, healthcare, and transportation. Analyzing this data as it flows allows organizations to quickly detect anomalies, identify potential problems, and respond proactively.

A prominent example is Bosch, which has implemented IoT-based real-time monitoring in its automotive manufacturing plants. Sensors attached to machines and production lines stream time-series data that Bosch analyzes to detect deviations from normal operating conditions. This setup has significantly reduced unplanned downtime and improved **Overall Equipment Effectiveness (OEE)**, demonstrating the power of real-time insights in industrial IoT.

Real-time monitoring is increasingly supported by advancements in edge computing and federated learning, enabling faster processing of data closer to the source, reducing latency, and improving responsiveness.

Predictive analytics

Predictive analytics is another crucial application of time-series analysis in IoT. By leveraging historical time-series data and advanced models, organizations can forecast future trends, anticipate equipment failures, and optimize operations. This proactive approach allows for better resource allocation and reduces costly disruptions.

Rolls-Royce provides a notable case with its predictive maintenance solution. By analyzing time-series data from sensors embedded in aircraft engines, Rolls-Royce predicts component failures and schedules maintenance before issues arise. This strategy has reduced maintenance costs, increased aircraft availability, and bolstered safety. This is further enhanced by models such as LSTM networks and reinforcement learning, which excel at processing complex time-series data for more accurate predictions.

Recent advancements in hybrid ML models, which combine the strengths of traditional statistical techniques such as ARIMA with DL models, offer even more precise forecasting for predictive maintenance and other IoT applications.

Challenges in handling time-series data in IoT

Despite its benefits, time-series analysis in IoT also comes with challenges, particularly in terms of data volume and data quality.

Data volume

IoT systems continuously generate vast amounts of data, often in real time, which can strain storage, processing, and analysis systems. Scalable infrastructure and efficient data management strategies are essential to cope with these high volumes.

Many industries face the challenge of managing large-scale time-series data in their IoT environments, such as in smart factories where thousands of sensors generate real-time data. Cisco's Connected Factory solution (https://www.cisco.com/c/dam/m/es_la/internet-of-everything-ioe/industrial/assets/pdfs/cisco-connected-factory.pdf) addresses this challenge by providing scalable infrastructure for collecting, processing, and analyzing this data. In such industrial IoT applications, companies often rely on cloud platforms such as AWS and Microsoft Azure to handle the massive data streams, ensuring scalable and flexible data processing and storage.

Data quality

Ensuring data quality is another critical challenge in IoT environments. Time-series data collected from sensors can be noisy or inconsistent or contain missing values. Poor data quality can lead to inaccurate analysis and flawed decision-making. Robust data preprocessing techniques, such as data cleaning, outlier detection, and imputation, are necessary to address data quality issues.

For example, in IoT-based healthcare systems, data quality issues can arise due to network disruptions, device malfunctions, or inconsistencies in data collection. Implementing advanced data preprocessing algorithms to handle missing values and ensure data integrity is crucial before performing time-series analysis.

As IoT ecosystems expand, the volume and complexity of time-series data will only grow. Addressing these challenges requires a combination of cutting-edge technologies, such as edge AI, robust data management, and domain-specific expertise. The importance of effective time-series analysis will continue to increase, driving innovation across industries—from smart cities to precision agriculture—where IoT systems are transforming traditional operations.

Working with Large Time-Series Data Using SparkML

As the volume of time-series data grows—especially in IoT environments—traditional methods may struggle with scalability, latency, and resource constraints. Apache Spark provides a distributed computing framework that excels at handling large-scale data processing, and its ML library, **SparkML**, offers various tools and algorithms to work with big data, including time-series data.

While SparkML does not have a dedicated time-series module like some libraries (such as **statsmodels** or **Prophet**), it provides the essential building blocks for scalable time-series analysis, such as feature engineering, regression, and classification. By leveraging Spark's distributed architecture, time-series models can be trained on large datasets across clusters, enabling the efficient processing of IoT data streams.

Spark's core advantage is its ability to scale data processing across a distributed cluster. This is essential for IoT environments where time-series data is generated in real time by thousands or millions of devices. SparkML allows users to preprocess, transform, and model these large datasets efficiently.

Spark's DataFrames and **Resilient Distributed Dataset (RDD)** abstractions allow for the parallel processing of time-series data in a fault-tolerant manner.

Time-series data often requires rolling or sliding window operations to compute metrics such as moving averages or lag features. Spark's built-in windowing functions (via `org.apache.spark.sql.expressions.Window`) allow efficient computation of these features in a distributed environment.

With Spark's SQL-like query capabilities, you can perform time-based aggregations, such as hourly or daily averages, which are crucial in IoT data analysis. SparkML provides feature scaling techniques (e.g., `MinMaxScaler` or `StandardScaler`) to normalize time-series data, ensuring better performance in ML models.

While SparkML does not natively support time-series forecasting models (e.g., ARIMA), it offers regression techniques that can be adapted for time-series forecasting by using lagged features.

SparkML's Pipeline API enables the creation of end-to-end ML workflows. For time-series data, this could include steps for feature engineering (lagging or differencing), model training, and evaluation, all of which can be managed in a single workflow that is both scalable and efficient. Pipelines can include stages such as data cleaning, scaling, model fitting, and hyperparameter tuning using cross-validation (`CrossValidator`) or train-validation splits (`TrainValidationSplit`).

To demonstrate SparkML's capabilities for time-series analysis, let's walk through a practical example using the decision tree algorithm, one of the native models available in SparkML. We will be using the UCI Machine Learning Repository's Air Pollution dataset.

The Air Pollution dataset contains time-series data of various air pollutants, such as **Carbon Monoxide (CO)**, **Nitrogen Dioxide (NO2)**, and **Ozone (O3)**, along with meteorological variables such as temperature and humidity. The data was collected hourly from an Italian city (<https://archive.ics.uci.edu/ml/machine-learning-databases/00360/AirQualityUCI.zip>).

Our goal is to build a decision tree model using SparkML to predict the concentration of a specific air pollutant based on the other variables in the dataset. Here's a step-by-step overview of the process:

1. We start by importing the necessary libraries for our SparkML example. These include the following:

```
from pyspark.sql import SparkSession
import pandas as pd
import numpy as np
from pyspark.sql.window import Window
from pyspark.sql.functions import lag, col
from pyspark.ml.feature import MinMaxScaler
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import DecisionTreeRegressor
from pyspark.ml.evaluation import RegressionEvaluator
```

2. Next, we initialize the Spark session:

```
spark = SparkSession.builder.appName(
    "TimeSeriesAnalysis").getOrCreate()
```

3. Let us load the dataset and select the useful columns:

```
df_pandas = pd.read_csv(
    'AirQualityUCI.csv',
    sep=';', decimal=',',
    parse_dates=[['Date', 'Time']])
df_pandas = df_pandas[
    ['Date_Time', 'CO(GT)', 'PT08.S1(CO)',
     'NMHC(GT)', 'C6H6(GT)', 'PT08.S2(NMHC)',
     'NOx(GT)', 'PT08.S3(NOx)']
]
df_pandas = df_pandas.dropna()
```

4. Since our goal is to predict CO levels, it's important to first examine the statistical distribution of the CO (GT) values before building the model. This will give us insights into the data quality and potential preprocessing steps:

```
print(df_pandas['CO(GT)'].describe())

# Check for missing or negative values
missing_values = df_pandas['CO(GT)'].isnull().sum()
negative_values = (df_pandas['CO(GT)'] < 0).sum()
print(f"Missing Values: {missing_values}, Negative Values: {negative_values}")
```

```
Output>>>
count      9357.000000
mean       -34.207524
std         77.657170
min        -200.000000
25%         0.600000
50%         1.500000
75%         2.600000
max         11.900000
Name: CO(GT), dtype: float64
Missing Values: 0, Negative Values: 1683
```

5. Since negative values are not physically meaningful for CO levels, we have two options: either remove the rows with negative values or impute them using techniques such as median imputation or interpolation. Given that CO values on nearby days are likely to be similar, we opt for the interpolation method:

```
df_pandas['CO(GT)'] = df_pandas['CO(GT)'].apply(
    lambda x: np.nan if x < 0 else x)
```

```
# You can then apply interpolation to fill missing values
df_pandas['CO(GT)'] = df_pandas['CO(GT)'].interpolate()
```

6. SparkML does not support column names containing periods ('.'), so we need to transform the column names to make them compatible with Spark by replacing the periods with underscores ('_'):

```
df_pandas.columns = [
    col.replace('.', '_')
    for col in df_pandas.columns
]
```

7. Next, we create a Spark DataFrame from our dataset. In this example, since we started with a CSV file, this step is necessary. However, in many cases, organizations store their data directly in Spark-based data warehouses, so this step might be skipped. In the following code, you can see how we convert the data to a Spark DataFrame:

```
data = spark.createDataFrame(df_pandas)
data.show(5)
```

Date_Time	CO(GT)	PT08_S1(CO)	NMHC(GT)	C6H6(GT)	PT08_S2(NMHC)	N0x(GT)	PT08_S3(N0x)
10/03/2004 18.00.00	2.6	1360.0	150.0	11.9	1046.0	166.0	1056.0
10/03/2004 19.00.00	2.0	1292.0	112.0	9.4	955.0	103.0	1174.0
10/03/2004 20.00.00	2.2	1402.0	88.0	9.0	939.0	131.0	1140.0
10/03/2004 21.00.00	2.2	1376.0	80.0	9.2	948.0	172.0	1092.0
10/03/2004 22.00.00	1.6	1272.0	51.0	6.5	836.0	131.0	1205.0

only showing top 5 rows

Figure 10.1: Top five rows in a Spark DataFrame

8. We will now preprocess the dataset, focusing on creating lag features for time-series prediction. For simplicity, we'll create lag features for the CO (GT) column, which represents the concentration of carbon monoxide in the atmosphere:

```
window = Window.orderBy("Date_Time")
data_with_lags = (
    data.withColumn(
        "lag_1", lag(col("`CO(GT)`"), 1).over(window)
    ).withColumn(
        "lag_2", lag(col("`CO(GT)`"), 2).over(window)
    ).withColumn(
        "lag_3", lag(col("`CO(GT)`"), 3).over(window)
    ).withColumn(
        "lag_4", lag(col("`CO(GT)`"), 4).over(window)
    ).withColumn(
```

```

        "lag_5", lag(col("`CO(GT)`"), 5).over(window
    )
)
data_clean = data_with_lags.dropna()
data_clean.show(5)

```

Date_Time	CO(GT)	PT08_S1(CO)	NMHC(GT)	C6H6(GT)	PT08_S2(NMHC)	NOx(GT)	PT08_S3(NOx)	lag_1	lag_2	lag_3	lag_4	lag_5
[01/01/2005 05.00.00]	1.4	1004.0	-200.0	4.8	753.0	181.0	879.0	1.9	2.7	2.5	1.6	1.595876288659794
[01/01/2005 06.00.00]	1.5	1001.0	-200.0	5.3	777.0	171.0	859.0	1.4	1.9	2.7	2.5	1.6
[01/01/2005 07.00.00]	1.4	974.0	-200.0	4.5	736.0	168.0	888.0	1.5	1.4	1.9	2.7	2.5
[01/01/2005 08.00.00]	1.1	915.0	-200.0	3.0	653.0	169.0	973.0	1.4	1.5	1.4	1.9	2.7
[01/01/2005 09.00.00]	1.0	939.0	-200.0	3.0	649.0	145.0	996.0	1.1	1.4	1.5	1.4	1.9

only showing top 5 rows

Figure 10.2: Top five rows of Spark DataFrame after adding lag features

9. Scaling the features is essential to prevent the model from being biased toward features with larger numeric values. In time-series data, the raw values of some features can vary significantly:

```

assembler = VectorAssembler(
    inputCols=["lag_1", "lag_2", "lag_3", "lag_4", "lag_5"],
    outputCol="features"
)
scaler = MinMaxScaler(
    inputCol="features",
    outputCol="scaled_features"
)
training_data = assembler.transform(data_clean)
scaler_model = scaler.fit(training_data)
scaled_data = scaler_model.transform(training_data)

```

10. Now we use a DecisionTree regressor and fit it on our scaled data:

```

dt = DecisionTreeRegressor(
    featuresCol="scaled_features",
    labelCol="CO(GT)"
)
dt_model = dt.fit(scaled_data)

```

11. Finally, we make the predictions using our trained regressor:

```

predictions = dt_model.transform(scaled_data)

predictions.select(
    "Date_Time", "`CO(GT)`", "prediction"
).show(5)

```

Date_Time	CO(GT)	prediction
01/01/2005 05.00.00	1.4	1.908670011325342
01/01/2005 06.00.00	1.5	1.5199851487067881
01/01/2005 07.00.00	1.4	1.5199851487067881
01/01/2005 08.00.00	1.1	1.5199851487067881
01/01/2005 09.00.00	1.0	1.179554701734488

only showing top 5 rows

Figure 10.3: Top five rows showing ground truth and prediction

12. Let us evaluate the model. The **Root Mean Squared Error (RMSE)** value is 0.71, indicating that the model has learned reasonably well. In the next section, we will explore using an LSTM model, which we expect to yield even better performance:

```
evaluator = RegressionEvaluator(
    labelCol="CO(GT)",
    predictionCol="prediction",
    metricName="rmse"
)
# Compute RMSE (Root Mean Squared Error)
rmse = evaluator.evaluate(predictions)
print(f"Root Mean Squared Error (RMSE): {rmse}")

Output >>>
Root Mean Squared Error (RMSE): 0.7121596162653802
```

The model evaluation shows an RMSE of 0.71, indicating a good fit to the data. This suggests that the model has captured the underlying patterns well.

In the next section, we will implement an LSTM model, which is particularly suited for time-series data and is expected to improve performance by capturing temporal dependencies more effectively.

Review of RNNs and LSTMs for Time-Series Analysis

RNNs and LSTM networks, introduced in *Chapter 4*, have proven to be powerful tools for analyzing and predicting time-series data. In this section, we will recap the key concepts covered in the DL chapter and discuss the adjustments and considerations needed when applying these architectures to IoT data.

Recap of RNN and LSTM concepts

RNNs are a class of neural networks designed to handle sequential data, making them well suited for time-series analysis. Unlike feedforward neural networks, RNNs maintain a hidden state that allows them to capture dependencies and patterns over time. The output at each time step depends not only on the current input but also on the previous hidden state.

LSTMs are a special type of RNN that addresses the vanishing gradient problem, which can occur when training RNNs on long sequences. LSTMs introduce a memory cell and gating mechanisms (input gate, forget gate, and output gate) that allow them to selectively remember or forget information over long periods. This enables LSTMs to capture both short-term and long-term dependencies in time-series data effectively.

In *Chapter 4*, we discussed the architecture and working principles of RNNs and LSTMs, including the mathematical formulations and the flow of information through the networks. We also covered the training process, such as **Backpropagation through Time (BPTT)**, and the various tasks to which these models can be applied.

Adjustments and considerations for IoT data

When we extend these concepts to IoT, there are a few key adjustments and considerations to ensure the successful application of RNNs and LSTMs for time-series analysis.

Handling multivariate time series

IoT data is often multivariate, meaning it consists of multiple interconnected sensor readings collected over time (e.g., temperature, humidity, and pressure). RNNs and LSTMs can be adjusted to handle these multivariate time series by taking multiple input sequences at each time step.

You need to design the input layer to accommodate multiple features for each time step, allowing the network to learn from multiple sensor streams simultaneously.

Dealing with missing data and noise

IoT data often suffers from noise and incompleteness, typically caused by sensor failures or transmission issues. To address these challenges, data preprocessing plays a crucial role in handling missing values and mitigating noise. Missing data can be imputed using methods such as linear interpolation or more sophisticated approaches, such as Gaussian processes. Noise can be reduced through techniques such as smoothing or Kalman filters. Additionally, to prevent overfitting to noisy data during model training, regularization methods such as dropout or L2 regularization should be applied, ensuring the model generalizes better to unseen data.

Data compression and real-time processing

IoT devices often generate data continuously in real time, and processing this data efficiently is crucial. Instead of training on large historical datasets, models can be trained incrementally as new data arrives. Techniques such as **online learning** or **stateful LSTMs** (which maintain hidden states across sequences) are useful here.

Since IoT devices may have limited computational power, RNNs/LSTMs can be optimized to run on edge devices with reduced memory footprints. Techniques such as **quantization** and **model pruning** can be used to shrink the size of the model while maintaining accuracy.

Time-series seasonality and trend considerations

Many IoT time-series datasets exhibit seasonal trends (e.g., daily or weekly patterns). LSTMs can be further enhanced to handle this by incorporating **Fourier features** or **exogenous inputs** that represent known seasonal patterns. Additionally, to capture both trend and seasonality in IoT data, hybrid models can be developed by combining LSTMs with classical time-series models such as ARIMA, allowing the deep network to learn residual patterns not captured by traditional methods.

With these adjustments, you can use RNNs and LSTMs for time-series analysis, particularly in the context of IoT data.

LSTM for CO level prediction

LSTMs, as we discussed earlier, are well suited for time-series data. In this section, we will now try predicting CO levels using an LSTM model:

1. As always, we start by importing all the necessary packages:

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
from keras.models import Sequential
from keras.layers import LSTM, Dense, Dropout
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
```

2. Next, we load the dataset and perform the same preprocessing steps as in the previous section using SparkML. This includes handling missing values, replacing negative CO values, interpolating, and scaling the features:

```
df = pd.read_csv(
    'AirQualityUCI.csv',
    sep=';', decimal=',',
    parse_dates=[['Date', 'Time']]
)
df = df[
    ['Date_Time', 'CO(GT)', 'PT08.S1(CO)',
     'NMHC(GT)', 'C6H6(GT)', 'PT08.S2(NMHC)',
     'NOx(GT)', 'PT08.S3(NOx)']
]
df = df.dropna()
df['CO(GT)'] = df['CO(GT)'].apply(
```

```

        lambda x: np.nan if x < 0 else x)
df['CO(GT)'] = df['CO(GT)'].interpolate()
scaler = MinMaxScaler()
df_scaled = scaler.fit_transform(
    df[['CO(GT)', 'PT08.S1(CO)', 'NMHC(GT)',
        'C6H6(GT)', 'PT08.S2(NMHC)', 'NOx(GT)',
        'PT08.S3(NOx)']]
)
df_scaled = pd.DataFrame(
    df_scaled,
    columns=['CO(GT)', 'PT08.S1(CO)', 'NMHC(GT)',
        'C6H6(GT)', 'PT08.S2(NMHC)', 'NOx(GT)',
        'PT08.S3(NOx)'])
)

```

3. To predict future CO levels, we will use data from the previous five days. We define a function to create sequences of data for this purpose (remember how easily this was handled in SparkML):

```

def create_sequences(data, time_steps=5):
    X, y = [], []
    for i in range(len(data) - time_steps):
        X.append(data[i:(i + time_steps), :])
        y.append(data[i + time_steps, 0]) # Predicting 'CO(GT)'
    return np.array(X), np.array(y)

```

4. Now, we will create our training and testing datasets. We'll use 80% of the data for training and 20% for testing:

```

time_steps = 5 # Looking back 5 time steps
X, y = create_sequences(df_scaled.values, time_steps)

# Split the data into training and testing sets
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

```

5. Next, we build an LSTM model in Keras. The model consists of two LSTM layers, each with 50 units, followed by two dropout layers for regularization. Finally, a dense layer is used to predict the CO levels:

```

model = Sequential()
model.add(LSTM(
    units=50, return_sequences=True,
    input_shape=(X_train.shape[1], X_train.shape[2])
))

```

```

model.add(Dropout(0.2))
model.add(LSTM(units=50))
model.add(Dropout(0.2))
model.add(Dense(1))

```

6. We can check the model architecture using the `keras.plot_model` function.:

```
keras.utils.plot_model(model, show_shapes=True)
```

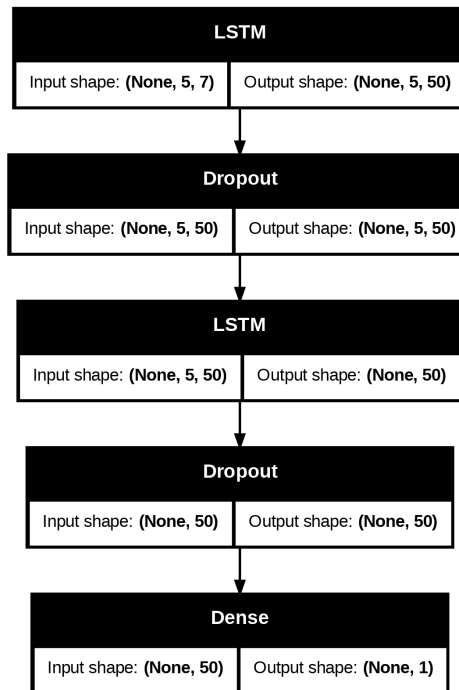


Figure 10.4: Keras LSTM architecture for CO prediction

7. Now, we compile the model and train it using the `fit` function. We use the Adam optimizer and **Mean Squared Error (MSE)** as the loss function. During training, 20% of the training data is used for validation:

```

model.compile(optimizer='adam', loss='mean_squared_error')
history = model.fit(
    X_train, y_train,
    epochs=20, batch_size=32,
    validation_split=0.2
)

```

```
Output>>>
Epoch 1/20
187/187 _____ 8s 10ms/step - loss: 0.0137 - val_
loss: 0.0116
Epoch 2/20
187/187 _____ 2s 7ms/step - loss: 0.0063 - val_
loss: 0.0073
Epoch 3/20
187/187 _____ 2s 11ms/step - loss: 0.0047 - val_
loss: 0.0061
Epoch 4/20
187/187 _____ 3s 14ms/step - loss: 0.0036 - val_
loss: 0.0047
Epoch 5/20
187/187 _____ 2s 8ms/step - loss: 0.0035 - val_
loss: 0.0050
Epoch 6/20
187/187 _____ 2s 8ms/step - loss: 0.0032 - val_
loss: 0.0052
Epoch 7/20
187/187 _____ 3s 9ms/step - loss: 0.0033 - val_
loss: 0.0045
Epoch 8/20
187/187 _____ 2s 7ms/step - loss: 0.0034 - val_
loss: 0.0046
Epoch 9/20
187/187 _____ 3s 11ms/step - loss: 0.0029 - val_
loss: 0.0046
Epoch 10/20
187/187 _____ 2s 13ms/step - loss: 0.0030 - val_
loss: 0.0044
Epoch 11/20
187/187 _____ 2s 8ms/step - loss: 0.0032 - val_
loss: 0.0046
Epoch 12/20
187/187 _____ 2s 8ms/step - loss: 0.0029 - val_
loss: 0.0049
Epoch 13/20
187/187 _____ 2s 8ms/step - loss: 0.0028 - val_
loss: 0.0046
Epoch 14/20
187/187 _____ 2s 8ms/step - loss: 0.0028 - val_
loss: 0.0044
Epoch 15/20
187/187 _____ 2s 8ms/step - loss: 0.0028 - val_
loss: 0.0047
```

```

Epoch 16/20
187/187 _____ 4s 14ms/step - loss: 0.0029 - val_
loss: 0.0046
Epoch 17/20
187/187 _____ 2s 8ms/step - loss: 0.0028 - val_
loss: 0.0044
Epoch 18/20
187/187 _____ 3s 8ms/step - loss: 0.0028 - val_
loss: 0.0050
Epoch 19/20
187/187 _____ 1s 8ms/step - loss: 0.0031 - val_
loss: 0.0045
Epoch 20/20
187/187 _____ 2s 7ms/step - loss: 0.0028 - val_
loss: 0.0046

```

8. With a training loss of 0.0028 and a validation loss of 0.0046 after 20 epochs, we can infer that the model has learned well. However, using more epochs with `EarlyStopping` could potentially yield even better results. Now, let's evaluate the model on the test data:

```

y_pred = model.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print(f"Root Mean Squared Error (RMSE): {rmse}")

Output>>>
59/59 _____ 1s 10ms/step
Root Mean Squared Error (RMSE): 0.05522588172402996

```

9. The following is the plot comparing the true CO levels (in blue) with the predicted CO levels (in orange). As you can see, the two match closely, indicating that the model has performed well:

```

plt.figure(figsize=(10, 6))
plt.plot(y_test, label='True CO(GT)')
plt.plot(y_pred, label='Predicted CO(GT)')
plt.legend()
plt.show()

```

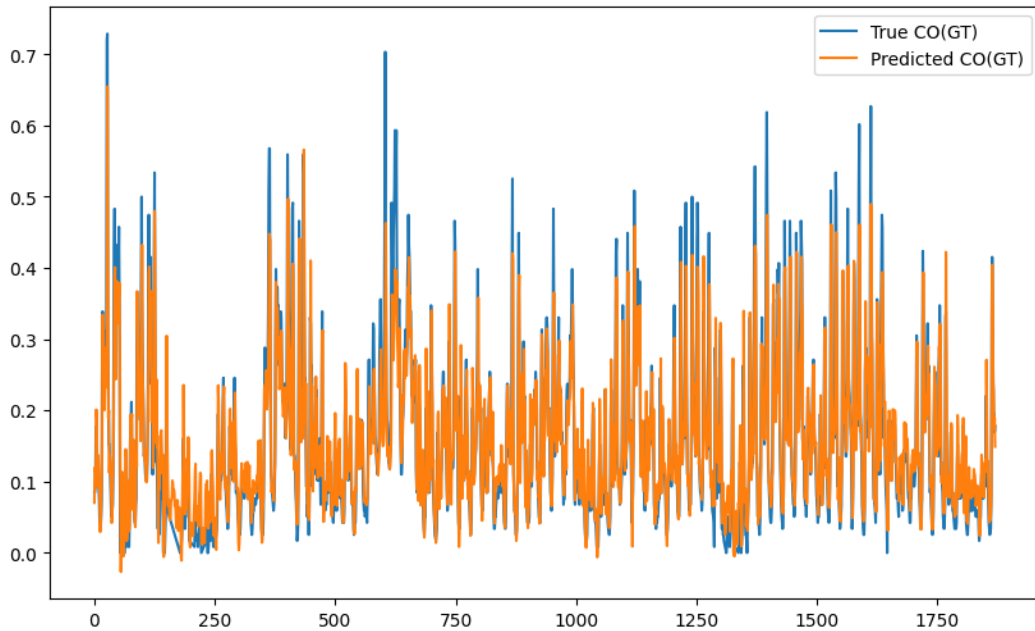


Figure 10.5: True and predicted CO levels

In this section, we built an LSTM model to predict CO levels using time-series data. We preprocessed the data by handling missing values, scaling the features, and creating sequences for model input. After splitting the data into training and testing sets, we trained an LSTM model with two LSTM layers and dropout regularization. The model achieved a reasonably low RMSE of 0.71, and the comparison between the true and predicted CO levels showed a good match, indicating that the model learned the patterns effectively.

To further improve the model's performance, we could do the following:

- **Increase the epochs with EarlyStopping:** This would allow us to train the model longer while preventing overfitting by stopping training when the validation loss stops improving.
- **Hyperparameter tuning:** Adjusting parameters such as batch size, number of LSTM units, or dropout rate could improve model performance.
- **Feature engineering:** Adding more relevant features or using domain-specific transformations could enhance the model's ability to capture complex patterns.
- **Use more advanced architectures:** We could explore using more sophisticated models such as bidirectional LSTMs or GRUs to capture more temporal dependencies in the data.

In the next section, we will explore using pretrained models to enhance our predictions. Pretrained models can provide a strong foundation by leveraging the knowledge learned from large datasets, which can be fine-tuned on our specific CO prediction task. This approach often improves accuracy and reduces the amount of training time required.

Pretrained Models for Time-Series Forecasting

Using pretrained models offers the advantage of avoiding the need to build and train models from scratch. Pretrained models, especially those developed using DL and statistical approaches, can quickly adapt to a wide range of forecasting problems, including IoT data from air pollution sensors. These models have been developed and trained on large datasets and can be fine-tuned for specific use cases. In this section, we will explore some popular pretrained models, including Prophet and Chronos.

Meta's Prophet

Prophet is an open source library developed by Facebook (now Meta) for forecasting time-series data. It is designed to be easy to use and robust to missing data, shifts in trends, and outliers. It is particularly useful for making predictions where there are trends, seasonality, and holidays or events affecting the data. Prophet is built to handle various time-series forecasting challenges, such as missing data, outliers, and changes in trends, making it suitable for business forecasting, such as sales or inventory management, where the time series exhibits certain patterns.

Prophet models a time series as an additive combination of the three components *trend* ($g(\tau)$), *seasonality* ($s(\tau)$), and *holidays and special events* ($h(\tau)$). Trend captures the long-term increase or decrease in the data over time. Prophet allows for piecewise linear or logistic growth models to capture this trend. Seasonality captures recurring patterns in the data, such as daily, weekly, or yearly seasonality. Prophet can model multiple seasonalities using **Fourier series**. Holidays and special events models the impact of specific dates or events, such as holidays, that might temporarily affect the data. In addition, there is an error term (ϵ) introduced that accounts for noise and irregularities that the model cannot explain:

Prophet's flexibility in capturing the trend is one of its strengths. It can model two types of trends:

- **Piecewise linear trend:** The model can fit data where the growth rate changes over time, allowing for breakpoints or "changepoints" where the trend shifts direction
- **Logistic growth trend:** For data that follows an S-shaped curve (e.g., populations or product adoption that grows rapidly and then plateaus), Prophet uses a logistic growth model with an upper limit (capacity)

Prophet provides confidence intervals (also called **uncertainty intervals**) for predictions. These intervals are generated using Monte Carlo sampling and help to give an indication of how confident the model is about future predictions. Prophet is robust to missing data and outliers. If the data has gaps or sudden spikes, the model can handle these without requiring complex preprocessing steps, which is particularly useful for real-world forecasting. Prophet is user-friendly, and even non-experts can use it to forecast time-series data without much fine-tuning.

Now, we will use Prophet to predict CO levels. To make the task more challenging and insightful, we will include additional features in our prediction. These features are the following:

- **PT08.S1(CO), PT08.S2(NMHC), and PT08.S3(NOx) (sensor readings)**: These columns represent sensor outputs related to specific pollutants.
- **NMHC(GT) (non-methane hydrocarbons)**: This represents a concentration of non-methane hydrocarbons in the air.
- **C6H6(GT) (benzene)**: Benzene concentration is one of the pollutant concentrations in the air.
- **NOx(GT) (nitrogen oxides)**: This value measures the concentration of nitrogen oxides.

By incorporating these features, we will perform a multivariate time-series prediction for CO levels, capturing the relationships between various pollutants and CO concentration over time.

Before proceeding, let's take a closer look at the data to ensure there are no outliers or erroneous values. We will plot the distribution of each variable to identify any potential issues:

```
plt.figure(figsize=(15, 10))

# Create subplots for each variable with negative values
variables = [
    'CO(GT)', 'PT08.S1(CO)', 'NMHC(GT)', 'C6H6(GT)',
    'PT08.S2(NMHC)', 'NOx(GT)', 'PT08.S3(NOx)'
]

for i, variable in enumerate(variables):
    plt.subplot(3, 3, i+1)
    sns.histplot(df[variable], kde=True, color='blue')
    plt.title(f'{variable} Distribution Before Cleaning')
    plt.xlabel(variable)
    plt.ylabel('Frequency')

plt.tight_layout()
plt.show()
```

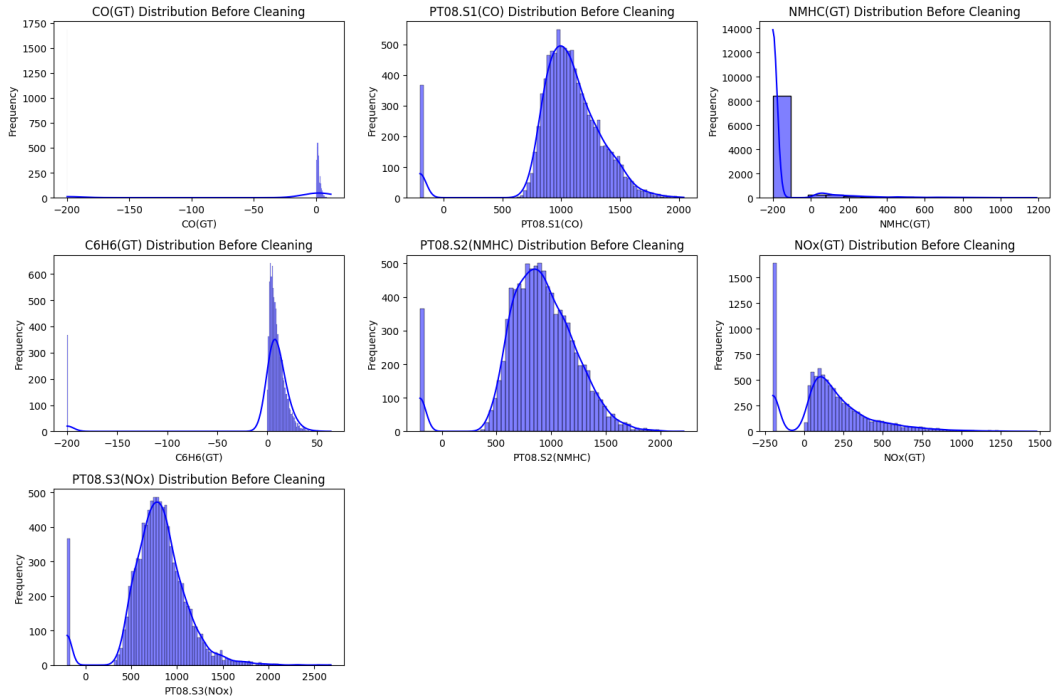



Figure 10.6: Distribution of features in air quality data

We can observe that, similar to CO values, some of the other variables also contain negative values. Since these negative values are not valid, we need to remove them. As before, we will use the interpolation method to fill in the missing values:

```
for column in variables:
    df[column] = df[column].apply(
        lambda x: np.nan if x < 0 else x
    ).interpolate()
```

If we plot the variables again, you'll notice that the negative values have been successfully removed:

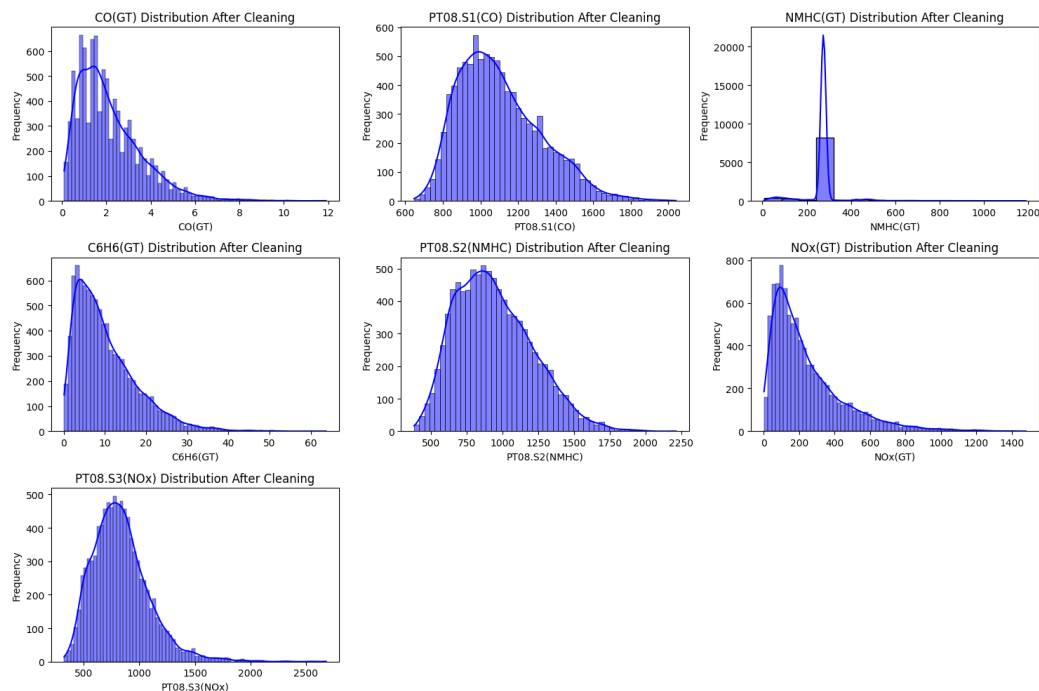


Figure 10.7: Distribution of features in air quality data after cleaning

Since Prophet is a **General Additive Model (GAM)** that handles seasonality and trend components independently. It doesn't assume that the values in the time series need to be within a specific range (such as $[0, 1]$). Therefore, scaling the data is not required. Prophet can work directly with the raw values of the time series and is robust to variations in scale. With that in mind, we can move forward without additional scaling or preprocessing.

Let's now begin using Prophet to model and predict CO levels:

1. Prophet requires the date-time column to be renamed as `ds` and the target variable we want to predict as `y`. Additionally, we need to ensure that the date-time values are correctly formatted. To achieve this, we will use the `to_datetime` function:

```
data = df[
    ['Date_Time', 'CO(GT)', 'PT08.S1(CO)',
     'NMHC(GT)', 'NOx(GT)']
].copy() #Create copy to avoid modifying original DataFrame
data.rename(
    columns={'Date_Time': 'ds', 'CO(GT)': 'y'},
    inplace=True
```

```

)
# Convert 'ds' column to datetime with explicit format
data['ds'] = pd.to_datetime(
    data['ds'],
    format='%d/%m/%Y %H.%M.%S'
)

```

2. Now, we define the Prophet model and add additional regressors we want to consider in our prediction. Then, we fit the model to our data:

```

model = Prophet()

#Add the additional regressors(you can choose relevant ones)
model.add_regressor('PT08.S1(CO)')
model.add_regressor('NMHC(GT)')
model.add_regressor('NOx(GT)')

# Step 3: Fit the model
model.fit(data)

```

3. Let's generate the forecast for the next 365 days. Since we don't have the future values for PT08.S1(CO), NMHC(GT), and NOx(GT), we will assume these values remain constant at their mean:

```

future = model.make_future_dataframe(periods=365)

future['PT08.S1(CO)'] = df['PT08.S1(CO)'].mean()
future['NMHC(GT)'] = df['NMHC(GT)'].mean()
future['NOx(GT)'] = df['NOx(GT)'].mean()
forecast = model.predict(future)

```

4. Let's now take a look at the forecasted values. Prophet provides several key outputs—`yhat`, which is the predicted value, and `yhat_lower` and `yhat_upper`, which represent the lower and upper confidence intervals, respectively:

```

forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']
].tail()

```

5. Prophet provides built-in plotting functions to visualize the forecast results, including the predicted values and confidence intervals:

```
fig = model.plot(forecast)
```

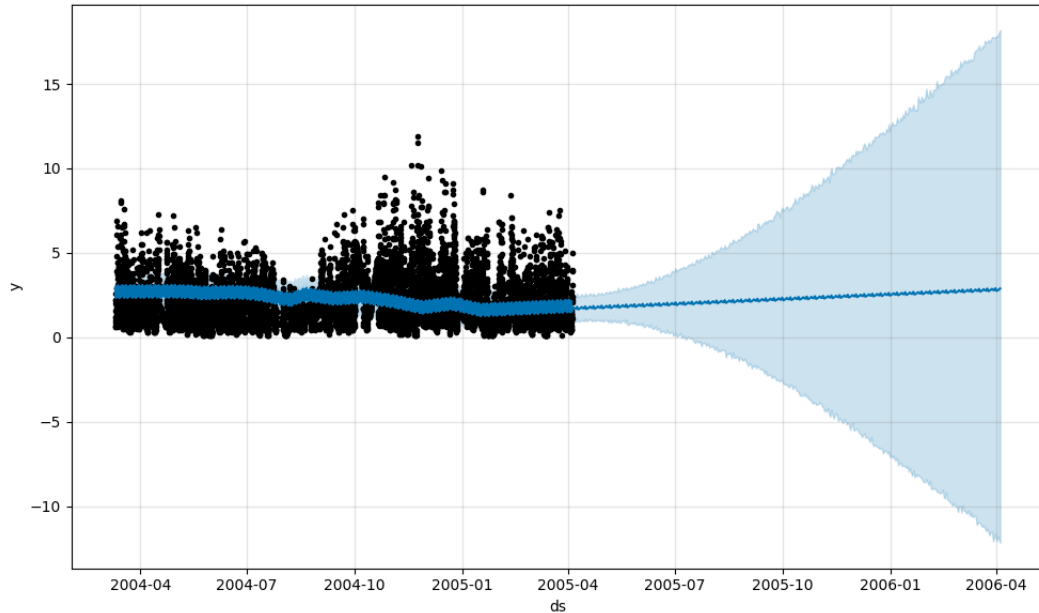


Figure 10.8: Model forecast with the actual data points (black dots) and the forecasted values (blue line) along with the uncertainty intervals (shaded region)

We can see that the model fits the historical data reasonably well, staying close to the actual values before the forecast period begins. The forecast shows that the CO levels are expected to stay relatively steady, with the uncertainty increasing as we forecast further into the future. The widening of the confidence intervals indicates that the model's predictions become less certain over time.

Prophet also allows us to view plots of the components that contribute to the forecast, namely trend, weekly seasonality, daily seasonality, and the impact of any additional regressors. To view these components we use the `plot_components()` method.

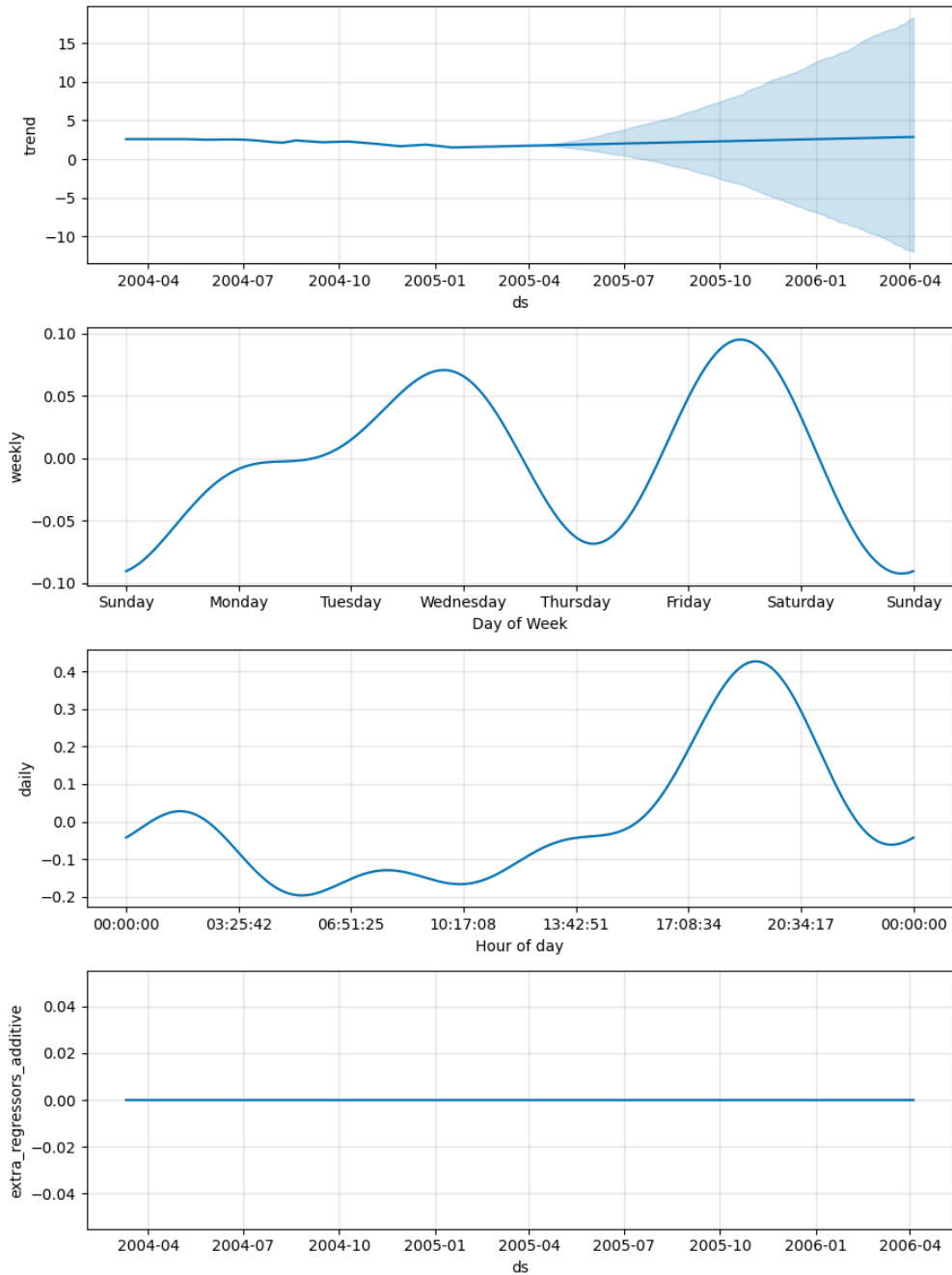


Figure 10.9: Decomposition output from Prophet, showing the components that contribute to the forecast: trend, weekly seasonality, daily seasonality, and the impact of any additional regressors

The trend represents the overall direction of the time series over the observed period. The blue line indicates that the trend for CO levels is fairly flat, with slight fluctuations but no significant increase or decrease over time. The second graph shows the weekly seasonality component; it captures how CO levels fluctuate over the days of the week. From the graph, we can see that CO levels peak on **Wednesday** and **Friday**, and dip on **Saturday** and **Sunday**. The third graph displays the daily seasonality, showing how CO levels change throughout the day. We can see that the CO levels peak in the evening, around 5 P.M. to 8 P.M., which could be related to increased traffic or other activities that contribute to higher CO levels during that time.

The last graph shows the effect of any additional regressors (in our case, the `PT08.S1(CO)`, `NMHC(GT)`, and `NOx(GT)` variables). We can see from the graph that the contribution of the extra regressors is flat and very close to 0, indicating that they do not have a significant additive effect on the forecast in our case.

In addition to its strong performance and ease of training, Prophet is lightweight and well suited for deployment on edge devices. Its fast prediction speed allows it to run efficiently, even on low-power hardware, making it ideal for resource-constrained environments.

Other pretrained models

Chronos, developed by Intel, is designed for time-series forecasting, anomaly detection, and feature engineering using DL models. It offers a variety of pretrained models for time-series tasks. Chronos offers several DL architectures, such as LSTMs, **Temporal Convolutional Networks (TCNs)**, and **NBEATS** for forecasting. Each of these models is designed to capture sequential dependencies in time-series data. Chronos models tend to perform well, especially on complex datasets with non-linear relationships, such as air quality data where multiple pollutants and environmental factors affect each other. Chronos is based on DL, so models such as LSTM and TCN are larger compared to Prophet. However, the model can be pruned and quantized for edge deployment. Training DL models is slower than Prophet, but after training, prediction speeds can be improved using optimizations such as **TensorFlow Lite** or **ONNX** for deployment.

Hugging Face's Transformers library contains a wide range of open source models, including transformer-based architectures that are adaptable for time-series forecasting. Hugging Face models leverage **transformers**, which were initially designed for NLP, but they have proven effective for time-series tasks due to their ability to capture long-range dependencies using attention mechanisms. Transformer models, when adapted to time-series data, can be effective in capturing both short-term and long-term dependencies, thanks to their self-attention mechanism. Hugging Face models are pretrained but can be fine-tuned on your specific dataset for more accurate predictions.

Summary

In this chapter, we covered what time-series data is and how it differs from other tabular data. We explored SparkML and used its built-in `DecisionTreeRegressor` to predict CO levels using the air quality dataset from the UCI Machine Learning Repository.

Next, we recapped RNNs and LSTMs, discussing why they are well suited for time-series forecasting. We built an LSTM model using Keras and trained it on the same air quality data.

We then shifted to pretrained models, fine-tuning Prophet on the dataset for more efficient predictions. Other pretrained models, such as Chronos and Hugging Face Transformers, were briefly discussed in terms of accuracy and edge deployment.

Throughout the chapter, we focused on the practical aspects of edge deployment, data cleaning, and model optimization to ensure efficient real-time forecasts on constrained devices.

In the next chapter, we will work with video and image data from IoT devices, exploring techniques for real-time visual data analysis.

References

- Pooja Nagpal and Sarika Chaudhary. “Health Monitoring Multifunction Band Using IOT.” In Proceedings of International Conference on IoT Inclusive Life (ICIIL 2019), NITTTR Chandigarh, India, pp. 159-165. Springer Singapore, 2020. https://doi.org/10.1007/978-981-15-3020-3_14.
- Saverio De Vito, Ettore Massera, Marco Piga, Luca Martinotto, and Girolamo Di Francia. “On field calibration of an electronic nose for benzene estimation in an urban pollution monitoring scenario.” *Sensors and Actuators B: Chemical* 129, no. 2 (2008): 750-757. <https://www.sciencedirect.com/science/article/abs/pii/S0925400507007691>.

Join our community on Discord

Join our community’s Discord space for discussions with the authors and other readers:

<https://packt.link/ds>



Leveraging AI for Visual Data from IoT

In this chapter, we'll delve into the role of **artificial intelligence (AI)** in processing visual data from IoT systems. Visual data plays a critical role in IoT applications, powering solutions from smart surveillance to industrial automation and healthcare monitoring. This chapter builds on concepts introduced earlier in this book. Specifically, in *Chapter 4*, we explored **convolutional neural networks (CNNs)** and discussed key architectures that underpin modern computer vision systems. Here, we'll extend that foundation, focusing on how these and other AI techniques can be tailored to handle the unique challenges of visual data in IoT environments.

Processing visual data in IoT introduces distinct challenges, such as handling large data volumes, ensuring real-time analysis, and deploying models on resource-constrained devices. This chapter addresses these aspects, emphasizing practical solutions for real-world IoT scenarios.

In this chapter, we're going to cover the following main topics:

- A Brief Introduction to Visual Data in IoT Ecosystems
- Challenges in Handling Visual Data in IoT
- Leveraging AI Techniques for Visual Data in IoT
- Attention Mechanisms Capturing the Spotlight in Visual AI

Technical requirements

To run the code examples in this chapter, you'll need to install three libraries:

```
pip install ultralytics  
pip install mediapipe  
pip install transformers
```


These commands will install **Ultralytics** (which includes YOLOv5), **MediaPipe** for computer vision tasks, and the **Transformers** library from Hugging Face.

A Brief Introduction to Visual Data in IoT Ecosystems

Efficiently leveraging visual data in IoT systems requires a nuanced understanding of its unique characteristics, associated challenges, and transformative potential. Let's look at it like this – visual data, including images and videos, plays a transformative role in IoT ecosystems, and unlike conventional IoT data streams such as time series data or tabular sensor logs, visual data captures the spatial and contextual richness of the real world. This enables IoT systems to perform advanced tasks, such as object recognition, activity analysis, and environmental monitoring, that are beyond the scope of traditional data sources.

The significance of visual data in IoT stems from its ability to provide granular, actionable insights. For example, while temperature sensors might identify anomalies in heat levels, cameras integrated with AI can discern whether the anomaly is caused by a system malfunction or a fire hazard. This contextual understanding is essential for applications such as autonomous vehicles, smart surveillance, and precision agriculture, where decision-making relies heavily on visual information.

Despite its potential, processing visual data in IoT environments introduces notable challenges. High data dimensionality demands significant computational, storage, and network resources. Real-time applications require low-latency AI solutions, often constrained by the limited power and processing capabilities of IoT devices. Additionally, privacy concerns are amplified due to the sensitive nature of visual data, necessitating robust encryption and anonymization techniques.

However, these challenges are met with equally compelling opportunities. The scale and complexity of visual data enable IoT systems to uncover patterns and insights that enhance decision-making across diverse applications. Innovations in AI, such as lightweight models and self-supervised learning, are making it feasible to deploy advanced computer vision techniques in resource-constrained IoT settings.

Handling Challenges in Visual Data in IoT

Processing visual data in IoT ecosystems presents unique challenges, primarily due to the high volume, bandwidth constraints, real-time requirements, and data privacy concerns associated with visual data. Each of these challenges demands tailored technical solutions to enable efficient and secure IoT operations. This section explores the key obstacles that are faced in managing visual data within IoT deployments and discusses potential approaches to address these challenges.

But before we get into handling the unique challenges that visual data introduces due to its inherent characteristics, we'll look at some of the challenges:

- **High dimensionality and redundancy in video streams:** Videos consist of consecutive frames, each a high-dimensional matrix of pixel values. A single second of Full HD video at 30 **frames per second (FPS)** generates 30 frames, each containing over 2 million pixels. This volume quickly scales up in IoT systems with multiple camera feeds. Much of the data across frames is redundant due to minimal changes between consecutive frames.
- **Temporal and spatial correlations in visual data:** Spatial patterns within frames are best captured by CNNs, while temporal patterns across frames, such as object movement, require models such as RNNs, 3D CNNs, or attention-based transformers such as **Time-Space Transformer (TimeSformer)**.

IoT systems often require spatial and temporal analyses to be combined to enable real-time event detection, such as identifying abnormal activities in surveillance feeds.

Hardware constraints

IoT devices are often deployed in environments with limited computational resources and energy supplies. These constraints demand careful planning in how we design and deploy AI models and systems for visual data analysis:

- **Constrained devices and edge computing:** IoT devices often have limited memory, processing power, or energy. Processing large visual data on these resource-restricted devices can be challenging. Pushing computation to the edge (e.g., via NVIDIA Jetson Nano or Google Coral TPU) reduces latency and prevents large volumes of visual data from being transmitted to the cloud.
- **Lightweight models and optimization techniques:** Deep learning models tailored for constrained hardware, such as **MobileNet**, **EfficientNet**, and **YOLO-Nano**, maintain robust performance while operating efficiently. Pruning, quantization, or distillation are frequently used to shrink model size and reduce computational complexity without severely sacrificing accuracy.
- **Balancing accuracy, latency, and energy efficiency:** High-resolution visual data improves model accuracy but increases computational and energy costs. Techniques such as **resolution scaling**, **frame skipping**, and **model quantization** can mitigate these trade-offs. Techniques such as implementing **dynamic inference pipelines** allow models to adapt complexity based on the available resources, ensuring low latency without significant accuracy loss.

Now that we've briefly touched on various constraints in handling visual data on IoT devices, let's talk about another important issue: data privacy and ethics.

Mitigating data privacy and ethical issues

Perhaps the most profound challenge in handling visual data in IoT systems centers on privacy and ethics. When we deploy cameras in public spaces, we're not just collecting data – we're capturing moments of people's lives, often without their explicit awareness or consent. This raises fundamental questions about privacy, consent, and the responsible use of technology. The nature of visual data often introduces sensitive and identifiable information, creating heightened concerns over privacy and ethical usage. Facial recognition in public spaces, a common application of IoT visual systems, can lead to surveillance misuse and privacy violations. Persistent monitoring using IoT-enabled cameras risks private activities or data being captured unintentionally.

There are various approaches we can implement to mitigate these ethical concerns:

- **Anonymization:** This involves blurring or masking sensitive regions, such as faces or license plates, before analysis. AI models such as segmentation networks can automate this process effectively. To further safeguard visual data, end-to-end encryption can be applied during both storage and transmission. This ensures that the data remains secure and inaccessible, even if it's intercepted by unauthorized parties.
- **Federated learning:** This decentralized **machine learning (ML)** approach enables AI models to be trained directly on IoT devices, edge nodes, or local servers without raw data needing to be transferred to a central cloud. By keeping data on-device, federated learning (refer to *Chapter 8* for more on federated learning) significantly reduces privacy risks and enhances data security while still allowing for collaborative AI improvements across multiple locations.
- **Bias detection and fairness enforcement:** Ensuring fairness and mitigating biases in visual AI systems is critical for equitable and ethical IoT applications. AI models trained on unrepresentative datasets often exhibit biases, particularly in tasks such as facial recognition or object detection. For instance, a model trained predominantly on images of one demographic group may underperform for other groups, leading to discriminatory outcomes in real-world applications such as surveillance or access control.

By addressing these challenges through a combination of advanced AI techniques and ethical considerations, IoT systems can leverage the full potential of visual data while minimizing risks and inefficiencies.

Examples of datasets for bias detection and fairness enforcement

When developing AI and IoT systems, using well-curated datasets is key to identifying and addressing bias. The following datasets have been specifically created to support fairness in model training and evaluation, helping ensure systems perform equitably across different demographic, environmental, and contextual conditions:

- **FairFace:** FairFace is a race-balanced face image dataset designed to address biases in facial recognition systems. It includes 108,501 images representing seven race groups: White, Black,

Indian, East Asian, Southeast Asian, Middle Eastern, and Latino. Images were curated from the YFCC-100M Flickr dataset, a large-scale public dataset, and annotated with demographic attributes such as race, gender, and age. FairFace can be used to train and evaluate AI models to ensure consistent accuracy across diverse demographic groups. For example, an IoT-based facial recognition system can be tested for fairness by assessing its performance across these demographic categories (<https://arxiv.org/pdf/1908.04913v1>).

- **Diversity in Object Recognition (DIOR):** DIOR is a large-scale object detection dataset with diverse geographic and socio-cultural contexts. It includes annotated images from different environmental and lighting conditions, ensuring a variety of perspectives. DIOR can help IoT models avoid biases related to underrepresented object categories or specific environments. For example, a smart agricultural monitoring system can benefit from diverse plant and terrain representations (<https://arxiv.org/pdf/1909.00133>).
- **Pilot Parliaments Benchmark (PPB):** PPB evaluates gender and skin-tone biases in commercial face analysis systems. It includes images from parliament members worldwide, providing a balanced dataset for testing facial analysis fairness. PPB can assess IoT visual systems' fairness when deployed for public safety or governance use cases (<https://proceedings.mlr.press/v81/buolamwini18a/buolamwini18a.pdf>).

In this section, we covered various techniques for mitigating biases, privacy, and ethical issues related to visual data in IoT. The next section will detail AI approaches to tasks such as object detection and pose detection that involve leveraging pretrained models.

Leveraging AI Techniques for Visual Data in IoT

The integration of AI with IoT technology has transformed the processes of capturing, analyzing, and utilizing visual data. As IoT devices generate vast volumes of images and videos, AI techniques provide the necessary tools to extract meaningful insights, enabling applications ranging from smart surveillance to predictive maintenance. In this section, we'll look at how advanced AI methodologies, such as computer vision, deep learning, and edge computing, empower IoT systems to process and interpret visual data effectively.

Deep learning architectures

The role of deep learning architectures in processing and analyzing visual data is key to the success of IoT applications. CNNs form the backbone of image data processing. CNNs and their derivatives have a layered architecture that extracts hierarchical features, making them indispensable for tasks such as object detection, image segmentation, and classification in IoT environments. These architectures, tailored to handle the complexities of image and video data, enable efficient feature extraction, object detection, and activity recognition. By optimizing models for edge devices, they ensure real-time performance while addressing the constraints of limited computational resources. Let's look at some popular deep learning architectures that are useful in IoT.

MobileNet

MobileNet is a CNN architecture explicitly designed for mobile and edge devices where computational resources are limited. It's optimized for speed and size while maintaining competitive performance in visual recognition tasks, making it an ideal choice for IoT applications such as facial recognition, object detection, and gesture recognition on low-power devices. To achieve this efficiency, MobileNet uses **depthwise separable convolutions** to reduce the number of parameters and computational costs. This technique breaks down a standard convolution (which combines spatial and depth operations into one step) into two parts:

- **Depthwise convolution:** Filters are applied to each input channel independently, focusing on spatial feature extraction
- **Pointwise convolution:** A 1x1 convolution is applied to combine the features from all channels, focusing on depth (inter-channel relationships)

This separation of convolutions reduces the computational load and the number of parameters compared to traditional convolutions. As a result, MobileNet models are compact, with fewer parameters (e.g., MobileNetV2 has only 3.4 million parameters compared to 60 million in models such as VGG16). This allows them to run efficiently on low-power devices, achieving faster inference times (up to 10 times faster on certain tasks) without requiring high-end **graphics processing units (GPUs)**. While lightweight, MobileNet retains high accuracy, making it suitable for tasks requiring real-time insights on edge devices.

Let's try it out:

1. Let's begin by importing the MobileNetV2 model from the Keras library, along with modules for image processing and decoding predictions:

```
from keras.applications import MobileNetV2
from keras.preprocessing import image
from keras.applications.mobilenet_v2 import (
    preprocess_input, decode_predictions)
import numpy as np
```

2. Load the pretrained MobileNetV2 model:

```
model = MobileNetV2(weights='imagenet')
```

For reference, the image we used is shown here:



Figure 11.1: The sample image input to the MobileNetV2 model

3. Next, we must load the image, resize it to the required input size for MobileNetV2 (224x224 pixels), and apply the necessary preprocessing:

```
img_path = 'sample_image.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)
```

4. Here are the predictions:

```
predictions = model.predict(x)
print('Predicted:', decode_predictions(predictions, top=3)[0])
```

Here, we print the top three predictions. As you can see, the highest prediction is `mountain_bike`, with approximately 46% confidence. The second most likely prediction is `bicycle-built-for-two`, at around 16% confidence, followed by `crash_helmet` with less than 1% confidence. Since this is a classification model, it provides multiple predictions with associated confidence scores to indicate how confidently the model associates each class label with the given input.

Here's the output:

```
1/1 ----- 2s 2s/step
Predicted: [('n03792782', 'mountain_bike', 0.4605968),
            ('n02835271', 'bicycle-built-for-two', 0.16410403),
            ('n03127747', 'crash_helmet', 0.009096457)]
```

As you can see, the model correctly predicts the image as containing a bike with the highest probability.

EfficientNet

EfficientNet is a family of CNNs designed to optimize both accuracy and efficiency. Unlike conventional models, which scale their architecture arbitrarily, EfficientNet systematically scales the following three dimensions using a compound scaling method:

- **Depth:** Increasing the number of layers allows for more complex feature extraction
- **Width:** Expanding the number of channels in each layer enables the network to learn richer representations
- **Resolution:** Using higher-resolution input images helps capture finer details

This approach ensures that the model achieves state-of-the-art performance with significantly fewer computational resources compared to traditional CNNs.

EfficientNet-B0 serves as the foundation for other variants. It uses a MobileNet-like inverted bottleneck structure, a computationally efficient building block designed for mobile devices. It also incorporates **squeeze-and-excitation (SE) blocks**, which improve the network's ability to focus on important features, further enhancing efficiency and accuracy.

SE blocks

SE blocks are a powerful architectural enhancement used in deep learning models to improve the network's ability to model interdependencies between feature channels. Introduced in *Squeeze-and-Excitation Networks*, SE blocks enable the network to “focus” on the most important feature channels by dynamically reweighting them based on their relevance to the task (<https://arxiv.org/pdf/1709.01507>).

EfficientNet models are designed to be computationally lightweight. EfficientNet-B0 has only 5.3 million parameters, significantly fewer than traditional models such as ResNet-50 (25.5M) and InceptionV3 (23.9M). It also requires fewer **floating-point operations per second (FLOPS)**, a measure of computational cost.

EfficientNet's ability to scale across IoT devices and environments while maintaining efficiency and accuracy ensures its place as a leading architecture for visual data processing in edge computing.

Just change the model in the previous code to `model = EfficientNetB0(weights='imagenet')` and you can test on the same image with EfficientNet.

You Only Look Once

You Only Look Once (YOLO) is an object detection model that predicts bounding boxes and class probabilities in a single pass. YOLO models (e.g., YOLOv5 and YOLOv8, with the latest being v9) are optimized for speed and accuracy, making them ideal for real-time IoT applications such as autonomous drones, smart surveillance, and robotic navigation.

YOLO processes an image in three key stages:

1. **Grid division:** The input image is divided into an $S \times S$ grid. Each grid cell is responsible for detecting objects whose center falls within the cell. Each cell predicts a fixed number of bounding boxes (e.g., two or three), along with the object class probabilities and confidence scores:



Figure 11.2: The input image with a 5x5 grid in white

2. **Bounding box prediction:** For each predicted bounding box, YOLO predicts the following aspects:
 - **Coordinates (x, y):** The center of the bounding box relative to the grid cell
 - **Dimensions (w, h):** The width and height of the box, normalized concerning the image's dimensions

- **Confidence score:** The probability that the box contains an object and the accuracy of the bounding box
- 3. **Class prediction:** Each grid cell predicts class probabilities for all possible object classes, conditioned on the presence of an object in the cell.

YOLO combines the confidence score and class probabilities to assign a final score to each bounding box. **Non-maximum suppression (NMS)** is then applied, a technique that filters out overlapping boxes by selecting the box with the highest confidence score and suppressing any other boxes that significantly overlap with it. This ensures that only the most likely detections are retained.

YOLO processes an image in a single forward pass, making it exceptionally fast. Models such as YOLOv5 (a smaller and faster version of YOLO) can achieve up to 140 FPS on modern GPUs, enabling real-time applications in IoT. YOLO is highly scalable, with variants such as YOLOv3, YOLOv4, YOLOv5, and YOLOv8 offering improved performance on resource-constrained devices. Lightweight versions such as YOLOv5s are specifically designed for edge deployment in IoT. YOLO supports multi-object detection, which makes it versatile for IoT.

YOLO's ability to combine speed, accuracy, and flexibility makes it uniquely suited for IoT applications. Its streamlined framework, lightweight versions, and scalability allow deployment to occur on diverse devices, from high-performance servers to edge IoT sensors. This versatility ensures that YOLO remains a cornerstone of real-time visual data processing in IoT systems.

Scalability across YOLO versions

As YOLO has evolved, each new version has introduced enhancements to improve performance, efficiency, and adaptability. The following highlights key improvements across versions, showing how YOLO has scaled to meet the needs of increasingly complex and diverse computer vision tasks:

- **YOLOv3:** Introduced multi-scale predictions and improved performance for small object detection.
- **YOLOv4:** Incorporated **Cross-Stage Partial Network (CSPNet)** and other optimizations to enhance accuracy and speed.
- **YOLOv5:** Focused on model scaling, compatibility with edge devices, and user-friendly implementation.
- **YOLOv8:** Introduced an anchor-free detection head, enhancing flexibility and simplifying the detection process.
- **YOLOv9:** Implemented **programmable gradient information (PGI)** and the **Generalized Efficient Layer Aggregation Network (GELAN)** to address information bottlenecks and improve gradient flow, resulting in higher accuracy and efficiency (<https://arxiv.org/abs/2402.13616>).
- **YOLOv10:** Eliminated the need for NMS by employing consistent dual assignments during training and optimized model components for superior performance with reduced computational overhead.

- **YOLOv11:** Introduced significant improvements in architecture and training methods, including enhanced feature extraction capabilities, optimized training pipelines, and adaptability across various environments, making it a versatile choice for a wide range of computer vision tasks.

YOLO in action

Now that we've explored how YOLO has evolved across versions, let's see it in action. In this hands-on example, we'll use a pre-trained YOLO model to detect objects in an image and visualize the results with bounding boxes and labels.

1. Let's begin by importing the YOLO model from the `ultralytics` library, along with `matplotlib` and `cv2` for image visualization and drawing bounding boxes, respectively:

```
from ultralytics import YOLO
import matplotlib.pyplot as plt
import cv2
```

2. Next, we'll use a pre-trained model from the YOLO family. Here, we'll choose the `yolov5su.pt` model file, which is a small and efficient version suitable for resource-constrained devices:

```
model = YOLO("yolov5su.pt")
```

3. Now, we'll use the YOLO model to predict objects in the image:

```
image_path = "/content/pexels-photo-5851037.jpeg"
results = model(image_path)
```

4. At this point, we can visualize the results. To do so, we'll define a function, `draw_boxes`, that we'll use to draw boxes over all the objects that were identified in the image. The functions take in the original image, as well as the box's coordinates and their classes:

```
def draw_boxes(image, boxes, class_names):
    # xyxy format [x1, y1, x2, y2]
    for i, box in enumerate(boxes.xyxy):
        x1, y1, x2, y2 = box.tolist()
        conf = boxes.conf[i].item() # Confidence score
        cls = int(boxes.cls[i].item()) # Class index
        class_name = class_names[cls]
        label = f"{class_name} {conf:.2f}"

        # Draw the bounding box
        cv2.rectangle(
            image,
            (int(x1), int(y1)),
            (int(x2), int(y2)),
            (255, 0, 0), 8
```

```
    )
    # Add the label
    cv2.putText(
        image, label,
        (int(x1), int(y1) - 10),
        cv2.FONT_HERSHEY_SIMPLEX,
        3, (0, 255, 0), 2, cv2.LINE_AA
    )
    return image
```

5. Now, we'll use the function on the result we obtained. We can see three bounding boxes, with two boxes for the person and one for the bicycle. We have also added the confidence values alongside each label:

```
result = results[0]
image = result.orig_img.copy()
boxes = result.boxes #
# Get class names
class_names = result.names

# Draw the bounding boxes and labels
annotated_image = draw_boxes(
    image, boxes, class_names
)
# Convert image color for matplotlib (BGR to RGB)
annotated_image = cv2.cvtColor(
    annotated_image, cv2.COLOR_BGR2RGB
)
# Display the image
plt.figure(figsize=(10, 10))
plt.imshow(annotated_image)
plt.axis("off")
plt.show()
```

Here, you can see the image annotated with bounding boxes, along with the corresponding object predictions for each box:

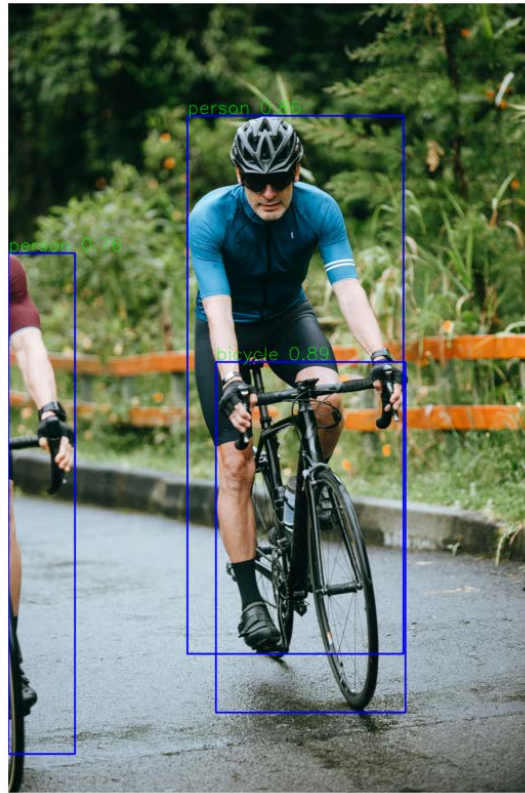


Figure 11.3: Predictions using YOLO with bounded boxes and confidence

This code demonstrates how to use a pre-trained YOLO model to detect objects in an image and visualize the results with bounding boxes and labels. Even on a CPU with only two cores, the model takes about 380 ms for an image. The ability to achieve such fast inference times on a basic two-core CPU highlights the accessibility and practicality of YOLOv5 for edge deployments.

Applications of YOLO beyond object detection

YOLO is renowned for object detection, but its versatility enables a wide range of applications across computer vision tasks. Let's consider various use cases; these have been structured as subsections for clarity and continuity.

YOLO use cases

Though YOLO is primarily known for object detection, it can be extended to a wide range of computer vision tasks. When paired with tracking algorithms such as SORT or DeepSORT, YOLO enables continuous object tracking across video frames for wildlife observation, traffic surveillance, or sports analytics. With segmentation extensions such as YOLO-Seg, it can identify pixel-level masks, facilitating defect detection in manufacturing or lane boundary recognition in autonomous vehicles. Adapting YOLO to detect key points supports pose estimation in fitness tracking, augmented reality, and animal behavior analysis. Its ability to detect multiple instances of objects also makes it suitable for counting tasks in settings such as inventory management or crowd monitoring. Additionally, YOLO's bounding boxes provide precise object localization for robotics (pick-and-place) or security (intruder detection), and when integrated with temporal data, YOLO is used for action recognition in surveillance or industrial safety. It can detect anomalies by spotting unusual objects on production lines or in medical imaging and plays a key role in autonomous driving by identifying vehicles, pedestrians, and road signs for tasks such as obstacle avoidance and lane departure warnings. Finally, YOLO powers gesture recognition for gaming, VR, and smart home interfaces, and provides broader scene understanding by analyzing the spatial relationships among multiple objects.

MediaPipe

MediaPipe is an open source, cross-platform framework developed by Google. It simplifies the creation of real-time ML pipelines for tasks such as pose estimation, face and hand tracking, and gesture recognition. MediaPipe enables developers to deploy efficient and lightweight solutions across various platforms, including mobile devices, desktops, and the web.

At its core, MediaPipe uses a **directed acyclic graph (DAG)** to represent computational pipelines. A DAG is a collection of nodes (representing operations) connected by directed edges (representing data flow) with no cycles, ensuring the execution of the pipeline is organized and efficient. Each node in the graph performs a specific operation, such as image processing or inference. This approach allows different components to be customized and integrated with ease.

MediaPipe offers several key advantages: first, it's *cross-platform compatible*, supporting Android, iOS, desktop, and web platforms. It's also *lightweight* and *efficient*, using optimized models to ensure low latency and real-time performance, especially on mobile and edge devices. For even better performance, MediaPipe leverages *hardware acceleration* through GPUs or specialized hardware such as the **Neural Networks API (NNAPI)** on mobile devices. Its *multi-modal processing capabilities* allow it to handle data from various sources, including video, audio, and sensors, enabling complex tasks that combine different modalities. MediaPipe also provides ready-to-use solutions, including *pre-built options* for pose estimation, hand tracking, face detection, and object detection. Furthermore, its *extensibility* allows developers to execute custom ML models alongside MediaPipe components by being compatible with TensorFlow Lite.

These features make MediaPipe suitable for a wide range of applications, including the following:

- **Pose estimation:** Fitness applications use pose estimation to provide real-time feedback during yoga or dance workouts, while AR systems utilize it for interactive experiences with virtual characters.
- **Hand and gesture recognition:** This technology enables touchless control of devices in smart homes, allows for immersive gaming experiences, and can be used to develop sign language recognition systems (you can refer to the Google blog for details on such an implementation: https://ai.google.dev/edge/mediapipe/solutions/customization/gesture_recognizer).
- **Face detection and tracking:** MediaPipe powers virtual makeup applications and interactive filters on social media. It also finds use in biometric authentication systems for enhanced security.
- **3D object detection:** AR applications leverage 3D object detection to overlay digital information onto real-world objects, while robots use it for navigation and object manipulation.
- **Healthcare and wellness:** MediaPipe enables real-time posture monitoring for ergonomic assessments in workplaces and facilitates fall detection systems for elderly care, enhancing safety and well-being.

Now, let's learn how to use MediaPipe for pose estimation on a video:

1. We'll start by importing the necessary modules – that is, `mediapipe`, `numpy`, and `cv2`:

```
import cv2
import mediapipe as mp
import numpy as np
```

2. Next, we'll initialize the MediaPipe pose estimation pipeline by creating a `mp.solutions.pose` object. We'll also import drawing utilities from `mp.solutions.drawing_utils` so that we can visualize the pose landmarks later:

```
mp_pose = mp.solutions.pose
mp_drawing = mp.solutions.drawing_utils
```

3. Now, let's define a function, `process_frame`, that will process each frame that's captured from the video. This function takes a video frame as input. First, it converts the input frame from the BGR color space (which is the default format used by OpenCV) to the RGB color space, which is required by MediaPipe. Then, it processes the RGB frame using the pose estimation pipeline (`pose.process`) to detect pose landmarks. If pose landmarks are detected, it uses the drawing utilities to draw those landmarks and their connections on the original frame. This helps with visualizing the estimated pose:

```
def process_frame(frame, pose):
    # Convert the frame to RGB
```

```

frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

# Process the frame and get the results
results = pose.process(frame_rgb)

# Draw the pose annotations on the frame
mp_drawing.draw_landmarks(
    frame,
    results.pose_landmarks,
    mp_pose.POSE_CONNECTIONS,
    landmark_drawing_spec=mp_drawing.DrawingSpec(
        color=(0,255,0),
        thickness=2,
        circle_radius=2
    ),
    connection_drawing_spec=mp_drawing.DrawingSpec(
        color=(255,0,0),
        thickness=2,
        circle_radius=2
    )
)

return frame

```

4. Next, we'll define another function called `process_video`. This function takes the path to an input video and an output path as arguments. It starts by opening the video file using `cv2.VideoCapture`. It retrieves the video's width, height, and FPS to use when saving the processed output video. After, it creates a `cv2.VideoWriter` object to save the processed video with the same properties as the input video. Then, it iterates through each frame of the video, applies the `process_frame` function to detect and draw pose landmarks, and writes the processed frame to the output video. Finally, it releases the video capture object:

```

def process_video(input_path, output_path):
    cap = cv2.VideoCapture(input_path)

    # Get video properties
    width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
    height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
    fps = int(cap.get(cv2.CAP_PROP_FPS))

    # Create VideoWriter object
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    out = cv2.VideoWriter(
        output_path, fourcc, fps,

```

```
        (width, height)
    )
    with mp_pose.Pose(
        min_detection_confidence=0.5,
        min_tracking_confidence=0.5
    ) as pose:
        while cap.isOpened():
            ret, frame = cap.read()
            if not ret:
                break

            # Process the frame
            processed_frame = process_frame(
                frame, pose
            )

            # Write the frame
            out.write(processed_frame)

    cap.release()
    out.release()
```

5. Finally, we can use the `process_video` function to process our video. We need to provide the path to the input video file and the desired path for the output video file:

```
input_video_path = "sample.mp4"
output_video_path = "output_video.mp4"
process_video(input_video_path, output_video_path)
```

After running this code, you'll see a new video where the detected pose landmarks (key body joints) are overlaid on each frame. The landmarks will be connected by lines, forming a skeletal representation of the person's pose in the video. You can check out the video containing the pose estimation results at <https://youtube.com/shorts/V7eVXkE4vF4?feature=share>.

Recurrent and temporal convolutional networks for video analysis

IoT applications often involve video data that requires not only spatial analysis but also a deep understanding of temporal relationships across frames. Traditional convolutional models, such as those shown previously in this chapter, excel in spatial feature extraction but lack explicit mechanisms to process sequential dependencies in video data.

In contrast, architectures such as 3D CNNs, RNNs, LSTMs, and **temporal convolutional networks (TCNs)** are designed specifically to address these challenges, making them uniquely suited for tasks that demand temporal awareness. In this section, we'll have a brief look at these deep learning architectures.

3D CNNs

3D CNNs extend traditional 2D CNNs by incorporating a third dimension (time) into the convolutional kernels. This allows the network to capture spatial (height, width) and temporal (time) features from a video simultaneously. 3D CNNs analyze temporal dynamics across a sequence of frames while maintaining strong spatial feature extraction. They are very effective at recognizing complex activities in video data (e.g., recognizing actions such as walking, running, or jumping) and are suitable for applications requiring frame-level precision in both spatial and temporal domains, such as the following:

- **Sports analytics:** Analyzing player movements and strategies
- **Surveillance:** Detecting anomalies and suspicious activities
- **Healthcare monitoring:** Tracking patient movements and vital signs

RNNs and LSTMs

RNNs are designed to process sequential data by maintaining a hidden state that captures information from previous time steps. LSTMs enhance RNNs by solving issues such as vanishing gradients, allowing them to capture long-term dependencies. They are effective at understanding relationships between sequential events. LSTMs can retain context over long video sequences, enabling robust predictions. They are ideal for predicting future actions or events based on observed sequences, such as the following:

- **Gesture recognition:** Recognizing hand movements or body gestures in videos (e.g., sign language recognition)
- **Predictive analytics:** Anticipating future actions, such as predicting pedestrian movement in autonomous driving
- **Video captioning:** Generating text descriptions for video sequences by capturing temporal dynamics

TCNs

While RNNs and LSTMs are powerful for sequential data, they can be computationally expensive, especially for long sequences. TCNs offer an alternative approach since they replace the sequential processing of RNNs/LSTMs with temporal convolutions that capture relationships across time using dilated convolutional filters. This allows TCNs to process sequences in parallel rather than sequentially, improving efficiency and scalability. They have been used in various applications:

- **Video summarization:** Identifying the most important parts of a video.
- **Complex activity recognition:** Recognizing activities with long-range dependencies

- **Anomaly detection:** Identifying unusual patterns or behaviors over time (e.g., detecting rare events in surveillance footage)

While YOLO, MobileNet, EfficientNet, and MediaPipe excel in spatial feature extraction and real-time applications, they fall short in tasks requiring temporal understanding. RCNs and TCNs, including 3D CNNs, RNNs, LSTMs, and TCNs, are uniquely suited for analyzing sequential data where both spatial and temporal relationships are critical.

This section was both challenging and diverse since we explored models such as EfficientNet, YOLO, and MediaPipe for visual data. We also noted that while these models are effective for images or slow-frame-rate video, they are not ideal for real-time video detection, where recurrent or temporal networks become necessary. In the next section, we'll examine another intriguing architecture: **vision transformers (ViTs)**.

Attention Mechanisms Capturing the Spotlight in Visual AI

In recent years, **attention mechanisms** have revolutionized the field of AI, emerging as a fundamental building block in modern ML models. Initially popularized by transformer models in **natural language processing (NLP)** (e.g., BERT and GPT), attention mechanisms have had a significant impact in the field of computer vision. By enabling models to dynamically focus on the most relevant parts of the input data, attention mechanisms have proven invaluable in processing complex visual information, such as images and videos.

Unlike convolutional layers, which operate over fixed local receptive fields, attention mechanisms capture both **local and global dependencies**. This is particularly beneficial in IoT systems, where visual data often contains intricate patterns spread across spatial and temporal dimensions.

ViTs – a paradigm shift in visual AI

ViTs bring the power of attention mechanisms to computer vision, fundamentally rethinking how image data is processed. Instead of relying on convolutional operations to extract local features, ViTs treat images as sequences of patches, similar to how text is processed in natural language models. Here's a deeper breakdown of how ViTs work:

1. **Divide the image into fixed-size patches:** An input image is divided into a grid of non-overlapping patches, each of size $p \times p$ pixels. For example, a 224×224 image could be split into 16×16 patches, resulting in $(224/16)^2 = 196$ patches. Each patch is flattened into a 1D vector by concatenating its pixel values. If the patch size is 16×16 and the image has three color channels (RGB), the flattened vector for each patch will have $16 \times 16 \times 3 = 768$ elements. Flattening patches allows the model to treat image patches like tokens in a sequence, akin to words in a sentence, enabling the application of transformers designed for sequential data.

2. **Add positional embeddings:** Unlike convolutional networks, which inherently respect spatial relationships due to their local receptive fields, transformers lack a built-in notion of order or position. To retain spatial information, each patch embedding is combined with a positional embedding, which encodes the patch's position in the original image. This positional encoding is a learnable vector that's added to each patch embedding during training. The positional embedding ensures that the model understands relationships between patches, such as which patches are adjacent or how they are spatially arranged.
3. **Pass through transformer layers:** The patch embeddings with positional encodings are fed into multiple transformer layers, each comprising multi-head self-attention, feed-forward neural network, layer normalization, and residual connections. ViTs stack multiple such transformer layers (e.g., 12, 16, or more) to progressively learn richer and more abstract features of the image.
4. **Classification token and output:** To perform classification, ViTs introduce a special learnable embedding called the [CLS] token. This token is prepended to the sequence of patch embeddings and is designed to aggregate information from all patches. At the end of the transformer layers, the embedding of the [CLS] token is passed through a final classification head (a fully connected layer) to produce the output. By using the [CLS] token, ViTs can efficiently summarize global information about the image without requiring additional pooling operations.

Let's see how we can use a ViT for image classification. We'll use Google's `google/vit-base-patch16-224` model, which was pre-trained on **ImageNet-21k** (14 million images, 21,843 classes) at a resolution of 224x224 pixels and then fine-tuned on **ImageNet 2012** (1 million images, 1,000 classes) at the same resolution:

1. We'll start by importing the required modules. We'll be using the Hugging Face's `transformers` library for this example:

```
from transformers import (
    ViTForImageClassification, ViTFeatureExtractor)
from PIL import Image
import torch
```

2. Next, we'll load the pre-trained ViT model and its corresponding feature extractor from Hugging Face's Model Hub:

```
model = ViTForImageClassification.from_pretrained(
    "google/vit-base-patch16-224"
)
feature_extractor = ViTFeatureExtractor.from_pretrained(
    "google/vit-base-patch16-224"
)
```

3. At this point, we'll load the image using the PIL library and then prepare it for the model using the feature extractor. This step ensures the image is in the correct format and size for the ViT:

```
image = Image.open("sample_image.jpg")
inputs = feature_extractor(images=image, return_tensors="pt")
```

4. Now, we'll pass the preprocessed image to the ViT model to get the predictions:

```
outputs = model(**inputs)
```

5. Finally, we'll extract the predicted class label from the model's output (*Figure 11.2* is the input image):

```
logits = outputs.logits
# model predicts one of the 1000 ImageNet classes
predicted_class_idx = logits.argmax(-1).item()
print(
    "Predicted class:",
    model.config.id2label[predicted_class_idx]
)
```

This is the result:

```
Predicted class: mountain bike, all-terrain bike, off-roader
```

As you can see, the model correctly predicts that the image is a mountain bike.

If you are interested in learning more about ViTs, including fine-tuning them on your own data, here's a helpful resource: <https://github.com/NielsRogge/Transformers-Tutorials/tree/master/VisionTransformer>.

Spatial-temporal attention – unlocking the power of video analysis

While ViTs have demonstrated exceptional performance for static images, video data introduces additional challenges due to the temporal dimension. IoT systems, such as surveillance networks or autonomous vehicles, often need to analyze video streams, requiring models to capture both spatial and temporal dependencies. Spatial dependencies refer to relationships between objects within a frame, while temporal dependencies capture how objects move and interact across frames. Let's explore a transformer model specifically designed to handle these spatial-temporal relationships: **TimeSformer**.

TimeSformer

TimeSformer is a groundbreaking architecture that adapts the transformer model for video analysis by addressing both spatial and temporal dependencies in video data. Unlike traditional convolutional models for video processing, which often use 3D convolutions to encode spatial and temporal information jointly, TimeSformer employs a transformer-based approach, separating spatial and temporal attention to achieve computational efficiency and flexibility.

Similar to ViTs, TimeSformer divides each video frame into non-overlapping patches, which are then flattened into a sequence of 1D vectors. The sequence of patch embeddings from all frames in the video is processed together, making the model capable of analyzing both spatial and temporal patterns.

To capture these patterns effectively, TimeSformer separates attention into two distinct dimensions:

- **Spatial attention:** This focuses on relationships within individual frames, identifying spatial features such as objects and their positions. It's similar to the attention mechanism in ViTs, but it operates on each frame independently.
- **Temporal attention:** This captures relationships between frames to understand motion and time-dependent changes, such as a person walking or an object falling. It analyzes how the features within patches change over time, capturing the dynamics of the video.

These attention mechanisms operate sequentially, ensuring computational efficiency by avoiding simultaneous computation across all spatial and temporal dimensions. Each transformer layer alternates between spatial and temporal attention, with positional embeddings added to maintain the order of patches and frames. Positional embeddings provide information about the location of each patch within a frame and the position of each frame within the video sequence, which is crucial for understanding spatial and temporal relationships.

The final output from TimeSformer is typically passed through a classification head for video-level tasks, such as action recognition or scene classification.

Let's walk through a complete pipeline by using TimeSformer for video classification. We'll need a video – for this example, we'll download the following video from Github:

```
!wget https://github.com/Educat8n/images/raw/refs/heads/main/dogs.mp4
```

In addition to transformers library, we'll need the `av` module to support video decoding inside `torchvision`. You can install it by using the following command:

```
pip install av
```

Let's start:

1. First, import the required model and utilities:

```
from transformers import (
    TimesformerForVideoClassification, AutoImageProcessor
)
from torchvision.io import read_video
from torchvision.transforms import (
    Compose, Resize, CenterCrop, ToPILImage
)
import torch
```

2. Now, load the pretrained TimeSformer model. This model has been trained on the Kinetics 400 dataset and can recognize common actions in videos:

```
model = TimesformerForVideoClassification.from_pretrained(
    "facebook/timesformer-base-finetuned-k400"
)
processor = AutoImageProcessor.from_pretrained(
    "facebook/timesformer-base-finetuned-k400"
)
```

3. Next, we'll load the video and extract its frame. Note that here, we're only extracting the first eight frames to speed up the classification process:

```
video_path = "dogs.mp4"
video, _, _ = read_video(video_path, pts_unit='sec')
frames = video[:8]
```

4. After, convert each frame from a PyTorch tensor into a PIL image and resize it to 224x224. This is the input size required by TimeSformer:

```
transform = Compose([
    Resize(256),
    CenterCrop(224),
    ToPILImage()
])
processed_frames = [transform(
    frame.permute(2, 0, 1)) for frame in frames]
```

5. Now, we'll use `AutoImageProcessor` to prepare inputs. This processor converts images into model-compatible tensors and includes position embeddings and normalization. We'll pass the processed frames through the model without computing gradients for inference:

```
inputs = processor(
    images=processed_frames,
    return_tensors="pt",
    do_rescale=False
)
with torch.no_grad():
    outputs = model(**inputs)
```

6. Finally, we'll retrieve the top predicted label from the model's output and print it:

```
pred_class = outputs.logits.argmax(-1).item()
print(
    "Predicted action:",
    model.config.id2label[pred_class]
)
```

The model for the video predicted carrying baby as an action. Cool, right? Feel free to experiment with other videos or explore different pretrained models from Hugging Face to expand your understanding.

TimeSformer's ability to capture both spatial and temporal information in video data effectively makes it a powerful tool for various IoT applications:

- **Activity recognition:** Understanding human actions or events in videos, such as recognizing gestures in a smart home or detecting falls in elderly care systems
- **Video summarization:** Creating concise summaries of long videos by identifying the most important segments
- **Anomaly detection:** Detecting unusual events or behaviors in video streams, such as identifying suspicious activities in surveillance footage

TimeSformer is not the only model incorporating spatial-temporal attention for video analysis. Several other transformer-based and hybrid architectures aim to address the challenges of video understanding, each with a unique approach:

- **Video Swin Transformer (VideoSwin):** This architecture extends the Swin Transformer (designed for images) to video data. It's a hierarchical ViT that combines spatial and temporal modeling using a shifted window mechanism. This allows it to process high-resolution videos efficiently by focusing attention on local windows within each frame and across frames.

- **Multiscale Vision Transformer (MViT):** MViT is designed specifically for videos, emphasizing multiscale processing. It captures information at different spatial and temporal resolutions, allowing it to capture both fine-grained details and long-range motion patterns. This makes it suitable for tasks that require understanding both local and global contexts in videos.
- **TokenLearner:** This model takes a different approach by dynamically selecting the most important tokens (patches) from video frames. This reduces redundancy and computational cost, making it more efficient for processing long videos. By focusing on the most informative parts of the video, TokenLearner can achieve good performance with fewer computational resources.
- **Video Vision Transformer (ViViT):** ViViT is similar to TimeSformer in that it uses separate spatial and temporal attention mechanisms. However, it introduces variations in how spatial and temporal information is fused, allowing for more flexible and efficient processing. ViViT uses a factorized self-attention mechanism to reduce computational complexity while capturing spatial-temporal relationships effectively.

To help you make a better choice, here's a comparative table of the various models discussed in this chapter:

Model	Parameters (Millions)	FLOPS (Billions)	Typical FPS (Edge Device)	Suitable For
MobileNetV2	~3.4M	~0.3 GFLOPS	~60–100 FPS	Low-power edge devices, real-time detection
EfficientNet-B0	~5.3M	~0.4 GFLOPS	~50–80 FPS	Medium edge devices, high accuracy
YOLOv5s	~7.5M	~17.1 GFLOPS	~80–140 FPS	Real-time object detection, multiple classes
ViT (Base)	~86M	~55 GFLOPS	~5–10 FPS	High-resource GPU/TPU, detailed scene understanding
TimeSformer	~121M	~121 GFLOPS	~2–5 FPS	Video classification, temporal analysis

These are just a few examples of the evolving landscape of transformer-based models for video analysis. As research in this area progresses, we can expect even more innovative architectures that push the boundaries of video understanding in IoT applications.

Summary

In this chapter, we explored the evolving role of AI in processing visual data within IoT ecosystems. Visual data presents both unique opportunities and challenges, requiring advanced AI techniques for efficient processing and meaningful analysis.

We began by examining the unique role of visual data in IoT, distinguishing it from other IoT data types, such as time series or tabular data.

Then, we discussed the high volume and redundancy in video streams, hardware constraints on resource-limited IoT devices, and privacy risks associated with sensitive visual data. Strategies such as data compression, lightweight models, encryption, and ethical AI practices were highlighted as key enablers for mitigating these challenges.

The core of this chapter focused on leveraging AI techniques for visual data in IoT and introduced powerful architectures such as MobileNet, EfficientNet, and YOLO for object detection and classification in real-time, resource-constrained environments. Additionally, frameworks such as MediaPipe were explored for real-time perception tasks.

We also highlighted attention mechanisms, with ViTs and their groundbreaking shift from convolution-based to attention-based architectures.

In the next chapter, we'll transition to exploring how to use AI to gather text, audio, and speech data from IoT while focusing on how NLP, audio signal processing, and speech recognition technologies are integrated into IoT systems to create seamless and interactive user experiences.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/ds>



AI for Text, Audio and Speech

Data from IoT

In this chapter, we'll explore how to work effectively with text, audio, and speech data originating from IoT devices. As IoT ecosystems become increasingly integrated into our daily lives – through voice assistants, smart home hubs, industrial monitoring systems, and wearable sensors – non-visual data (e.g., textual logs, voice commands, and environmental audio) takes on a critical role. We'll cover preprocessing techniques, **natural language understanding (NLU)**, **natural language generation (NLG)**, audio feature extraction, audio command recognition, and speech-to-text transcription. Finally, we'll discuss how to integrate these capabilities into a voice-assisted IoT environment while considering performance, scalability, and privacy.

In this chapter, we're going to cover the following main topics:

- Introduction to Text, Audio, and Speech Data
- Fundamentals of Text Data Processing
- Working with Audio Data in IoT

Technical requirements

To run the code examples provided in this chapter, you'll need the `nltk`, `transformers`, and `torch` Python modules. Install them by using the following command:

```
pip install nltk transformers torch nltk
```

Additionally, audio-based examples require `torchaudio` and `librosa`, which you can install via the following command:

```
pip install torchaudio librosa soundfile
```

The code and the `requirements.txt` file, which contains version details for this chapter, are available at <https://github.com/PacktPublishing/Hands-On-Artificial-Intelligence-for-IoT-Second-Edition/tree/main/Chapter12>.

Introduction to Text, Audio, and Speech Data

The IoT domain is often associated with devices that generate numeric sensor readings – temperature values, humidity levels, pressure measurements, and more. However, the IoT landscape also includes rich non-visual data sources that can deeply enhance user interaction, automate complex processes, and provide actionable insights beyond raw numbers. In particular, text, audio, and speech data have emerged as critical components of modern IoT ecosystems. By integrating textual alerts, voice commands, and natural language interfaces, IoT devices can offer more intuitive, user-friendly experiences. They can also support new functionalities that cannot be achieved through numeric data alone.

One of the most noticeable trends in consumer IoT is the rise of voice-controlled interfaces. From smart speakers such as Amazon Echo and Google Nest Hub to voice-enabled wearables, the ability to issue commands verbally allows for effortless, hands-free interaction. Instead of navigating a mobile application or pressing physical buttons, users can simply say, “Turn on the living room lights” or “Play jazz music.” This natural human-machine dialogue reduces friction, making IoT devices more accessible, especially for those with mobility or vision impairments. In industrial settings, voice-based controls allow technicians to operate machinery or retrieve information without pausing their tasks, increasing efficiency and safety.

Beyond voice commands, IoT devices frequently generate textual data. Many sensors or connected devices produce logs, alerts, and diagnostic messages as text. For instance, a factory machine might send textual warnings when it detects abnormal vibration patterns. A home security system could log textual alerts whenever motion is detected, indicating time, location, and sensor ID. This textual data, once processed and understood through NLP techniques, can be transformed into structured insights, enabling predictive maintenance, anomaly detection, and automated reporting.

In addition, non-visual data adds significant value in diverse IoT scenarios. Let’s look at a couple of these scenarios here:

- **Drones and robotics:** Pilots and operators can issue commands verbally to drones, while drones can provide status updates or flight logs in text form. This makes human-robot interaction more natural and scalable.
- **Healthcare and wearables:** Users can inquire about their health metrics verbally, and devices can respond with spoken summaries. Textual reports from IoT-equipped medical devices aid in remote patient monitoring.

In short, incorporating text, audio, and speech data into IoT solutions enables more intuitive user interfaces, improved accessibility, and richer context-aware functionalities.

Fundamentals of Text Data Processing

Textual IoT data can take many forms: device logs, alert messages, maintenance notes, or transcripts of spoken commands. Before we can leverage sophisticated NLP techniques, we need to transform raw text into cleaner, more manageable forms. Let's start by focusing on data collection and preprocessing, exploring how to gather raw textual data from various IoT endpoints and convert it into a cleaner, more consistent form. This foundation is critical for any AI pipeline, as the quality of preprocessing directly affects the accuracy and reliability of downstream models.

Data collection and preprocessing

In any AI pipeline, the quality of input data greatly influences the performance of downstream models. For textual IoT data, preprocessing is crucial to ensure that models receive clean and consistent inputs. IoT textual data can come from various sources, each requiring slightly different handling before analysis and modeling.

Let's explore some data sources:

- **Logs from IoT hubs:** Many IoT devices send periodic status updates, warnings, or errors in the form of textual logs. These could be plaintext files or structured JSON files. For example, a smart thermostat might log entries like the ones shown here:

```
2024-12-14 09:30:21 INFO: Thermostat set to 22°C
2024-12-14 09:32:10 WARN: Temperature sensor reading delayed
```

These logs need cleaning and meaningful keywords must be extracted (e.g., Thermostat, Temperature, and delayed).

- **Cloud-based notifications:** IoT devices may use messaging protocols such as **Message Queuing Telemetry Transport (MQTT)** or **Advanced Message Queuing Protocol (AMQP)** to publish alerts. A security camera might send messages like these:

```
ALERT: Motion detected at front door - 2024-12-14 10:00:05
```

These notifications often require timestamps to be parsed and device IDs to be extracted or converted into standardized formats before analysis.

- **User queries or commands:** Users might interact with IoT dashboards or mobile applications by typing text commands:

```
"Turn on the living room lights"
"Increase the water pump flow rate"
```

Understanding these commands involves normalizing text, identifying the device/entity mentioned, and mapping the command to an action.

Now that we've covered different data sources, let's understand what an ML pipeline for textual data looks like.

Outlining an ML pipeline

Before feeding textual data to ML models or NLP pipelines, we must apply a series of transformations. Here are the most commonly used steps:

1. **Normalization:** Convert all text into lowercase to ensure uniformity. Also, handle accented characters or Unicode normalization, ensuring that words such as “résumé” and “resume” are represented consistently.
2. **Tokenization:** Split text into individual units – usually, words or subwords. For example, “Turn on the living room lights” becomes “[“turn”, “on”, “the”, “living”, “room”, “lights”]”.
3. **Stopwords removal:** Remove common words that do not add semantic value, such as “the,” “and,” and “or.” This step helps reduce noise and allows us to focus on more meaningful words.
4. **Stemming:** Roughly cuts words down to their stem – for example, “running,” “runs,” or “ran” will become “run.”
5. **Lemmatization:** Use vocabulary and morphological analysis to reduce words to their dictionary form – for example, “better” becomes “good” (when lemmatizing adjectives). Lemmatization is often preferred for precise NLP tasks, but stemming is faster and simpler.
6. **Handle special characters and numbers:** Remove or handle punctuation and symbols, and possibly convert numbers into tokens such as “<num>” or keep them if needed. For IoT logs, numbers (such as 22°C or 45% humidity) may carry essential information, so you might choose to retain numbers or transform them into normalized units.

Now that we are familiar with the major steps of an ML pipeline, let’s build one.

Example of an ML pipeline

Let’s cover a step-by-step example that utilizes NLTK and SpaCy. We’ll assume that we have a file named `iot_logs.txt` that contains the following sample entries:

```
2024-12-14 09:30:21 INFO: Thermostat set to 22°C
2024-12-14 09:32:10 WARN: Temperature sensor reading delayed
Please turn on the living room lights.
```

Let’s get into it:

1. We’ll start by importing the essential tools for NLP using the `nltk` library. These include modules for tokenizing text, removing stopwords, lemmatizing words, and handling regular expressions. This sets up everything we need to preprocess and clean text data effectively:

```
import nltk
import re
import string
from nltk.corpus import stopwords
```

```
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
```

2. We'll proceed by downloading the necessary NLTK datasets and resources. This includes tokenization models (punkt and punkt_tab), a list of stopwords, the WordNet lexical database for lemmatization, and additional multilingual WordNet data (omw-1.4). These downloads ensure we have all the tools required for effective text preprocessing:

```
nltk.download('punkt')
nltk.download('punkt_tab')
nltk.download('stopwords')
nltk.download('wordnet')
nltk.download('omw-1.4')
```

3. Next, we'll initialize the tools needed for text preprocessing. We'll create a lemmatizer using WordNetLemmatizer to reduce words to their base forms and load a set of English stopwords to filter out common, non-essential words during text analysis:

```
lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english'))
```

4. Now, we'll open the iot_logs.txt file in read mode to load its contents. The file's data is read and stored in the variable data:

```
with open('iot_logs.txt', 'r') as f:
    data = f.read()
```

5. After, we must convert the text from the data variable so that it's in lowercase and store it in text. This ensures consistency by removing case sensitivity, which is essential for accurate text processing:

```
text = data.lower()
```

6. At this point, we must normalize the text by replacing occurrences of °C with the word degrees for easier readability and processing. Then, we must use a regular expression to remove all punctuation, leaving only numbers and letters intact. After this normalization step, the cleaned text is printed so that we can check the results:

```
text = text.replace(
    '°c', ' degrees'
) # Convert °C to 'degrees' for simplicity
text = re.sub(
    f'[{re.escape(string.punctuation)}]', '', text
) # remove punctuation except numbers and letters

print("\nAfter Normalization and Removing Punctuation:")
print(text)
```

Here's the output of this code:

```
>>>
After Normalization and Removing Punctuation:
20241214 093021 info thermostat set to 22 degrees
20241214 093210 warn temperature sensor reading delayed
please turn on the living room lights
```

7. Now, we'll tokenize the normalized text into individual words using `word_tokenize`, breaking the text into smaller units called tokens. These tokens are then printed, allowing us to see the text split as manageable pieces that we can use for further analysis:

```
tokens = word_tokenize(text)
print("\nTokenized:")
print(tokens)
```

Here's the output:

```
Tokenized:
['20241214', '093021', 'info', 'thermostat', 'set', 'to',
'22', 'degrees', '20241214', '093210', 'warn', 'temperature',
'sensor', 'reading', 'delayed', 'please', 'turn', 'on', 'the',
'living', 'room', 'lights']
```

8. Next, we must filter out common stopwords from the tokenized words using a list comprehension. This step removes non-essential words such as “and,” “the,” and “is,” leaving only the meaningful tokens. The cleaned tokens are then printed for review:

```
filtered_tokens = [
    w for w in tokens if w not in stop_words
]
print("\nAfter Stopwords Removal:")
print(filtered_tokens)
```

We'll get the following output:

```
After Stopwords Removal:
['20241214', '093021', 'info', 'thermostat', 'set', '22',
'degrees', '20241214', '093210', 'warn', 'temperature',
'sensor', 'reading', 'delayed', 'please', 'turn', 'living',
'room', 'lights']
```

9. Finally, we'll lemmatize the filtered tokens by reducing each word to its base or root form using `lemmatizer`. This step ensures that variations of words, such as plurals or different tenses, are unified. Here, the lemmatized tokens are printed to show the processed output:

```
lemmatized_tokens = [
    lemmatizer.lemmatize(w) for w in filtered_tokens
]
```

```
print("\nAfter Lemmatization:")
print(lemmatized_tokens)
```

Here's the result:

```
After Lemmatization:
['20241214', '093021', 'info', 'thermostat', 'set', '22',
'degree', '20241214', '093210', 'warn', 'temperature', 'sensor',
'reading', 'delayed', 'please', 'turn', 'living', 'room',
'light']
```

Here, you might notice the following:

- The timestamps remain as tokens. Depending on your use case, you may remove them or convert them into `<TIME>` tokens.
- The word “degrees” got lemmatized to “degree.”
- Common stopwords such as “the” and “to” were removed.
- Lemmatization reduced “lights” to “light.”

Using these steps, we can streamline the way we process any incoming textual data from IoT sources into a standardized format suitable for further analysis, whether it's keyword extraction, topic modeling, intent detection, or feeding it into an ML or deep learning model.

Encoding text

Once raw textual data from IoT devices has been cleaned, normalized, tokenized, and possibly lemmatized, the next step is to transform these words into numerical representations that ML models can understand. This process, known as text encoding, is crucial because most AI models operate on vectors, not raw strings.

Encoding text can be done at varying levels of sophistication. The representation you choose can significantly affect your model's performance, interpretability, and generalization ability. There are various approaches to encoding text:

- **One-hot encoding:** One-hot encoding represents each token (word) as a vector of the same length as the vocabulary, with all zeros except for a single one in the position corresponding to that token. It's simple to implement and easy to understand. However, it results in very high-dimensional, sparse vectors that do not capture semantic meaning (e.g., “light” and “lamp” are just as unrelated as “light” and “banana” in one-hot space). Suppose your vocabulary is `["temperature", "light", "sensor", "turn"]`. Using one-hot encoding, the word “light” is encoded as `[0, 1, 0, 0]`. This is good for very simple tasks or as a baseline, but it's rarely used in complex IoT scenarios.

- **Word embeddings:** Word embeddings such as Word2Vec and GloV are dense, low-dimensional vectors (often 50 to 300 dimensions) that capture semantic meaning and relationships between words. Words with similar meanings have similar embeddings. Using word embeddings is efficient; it even captures semantic similarity between words. However, it's context-independent – the embedding for a word is the same, regardless of its usage in a sentence, which can be problematic for words with multiple meanings. For example, Word2Vec might learn that “temperature” and “humidity” have similar embeddings because they often appear in similar IoT contexts, while “light” and “turn” might appear in different contexts. Offer a good balance of complexity and performance; this is suitable for many IoT use cases where you need semantic understanding but not necessarily deep context.
- **Contextual embeddings:** Transformer-based models such as BERT produce embeddings that depend on the entire sentence context. The word “light” in “living room light” will have a different embedding than “light” in “light weight sensor,” reflecting subtle differences in meaning. Contextual embeddings provide state-of-the-art performance in many NLP tasks as they can capture context dynamically. However, it results in larger models that are more computationally expensive and more complex to deploy at scale. This is ideal for nuanced tasks such as intent detection, command disambiguation, and more complex language understanding in IoT, albeit more resource-intensive.

Now that we understand some of the theory, let's look at a practical implementation.

Example implementation

Let's consider a small set of IoT-related sentences and demonstrate how to encode them by using Word2Vec for static embeddings and a pretrained BERT model for contextual embeddings:

1. We'll start by importing the essential libraries for text processing. This includes `nltk` for natural language tasks, `re` for handling regular expressions, and `string` for working with text characters. The `word_tokenize` function from `nltk` is being imported to break down text into individual tokens. Finally, we'll download the `punkt` and `punkt_tab` tokenizer models to ensure proper functionality for tokenizing sentences and words:

```
import nltk
import re
import string
from nltk.tokenize import word_tokenize
nltk.download('punkt')
nltk.download('punkt_tab')
```

2. Next, we'll define a list of example sentences representing commands or messages, possibly from IoT devices or related systems. These sentences cover actions such as controlling devices, monitoring conditions, and reporting issues, providing a small dataset for text analysis or preprocessing:

```
sentences = [
    "Turn on the living room light",
    "Increase the thermostat temperature",
    "Sensor reading delayed due to network issue",
    "Check humidity levels in the greenhouse"
]
```

- Now, we'll define a helper function, `simple_preprocess`, to standardize and tokenize text. This function converts the text into lowercase for consistency, removes punctuation using a regular expression, and tokenizes it into words using `word_tokenize`. This function is applied to each sentence in the list of sentences, resulting in a list of tokenized sentences:

```
def simple_preprocess(text):
    text = text.lower()
    text = re.sub(
        f'[{re.escape(string.punctuation)}]', '', text
    )
    tokens = word_tokenize(text)
    return tokens

tokenized_sentences = [
    simple_preprocess(s) for s in sentences
]
print("Tokenized Sentences:")
print(tokenized_sentences)
```

The output will be as follows:

```
Tokenized Sentences:
[['turn', 'on', 'the', 'living', 'room', 'light'], ['increase',
'the', 'thermostat', 'temperature'], ['sensor', 'reading',
'delayed', 'due', 'to', 'network', 'issue'], ['check',
'humidity', 'levels', 'in', 'the', 'greenhouse']]
```

- Now that we've done that, we'll extract all tokens from the tokenized sentences into a single list and create a unique vocabulary by converting it into a set. The vocabulary is then sorted alphabetically and printed, giving us a clear view of all distinct words present in the dataset:

```
all_tokens = [
    token for sent in tokenized_sentences for token in sent
]
vocab = list(set(all_tokens))
vocab.sort()
print("Vocabulary:", vocab)
```

The response is shown here:

```
Vocabulary: ['check', 'delayed', 'due', 'greenhouse',
'humidity', 'in', 'increase', 'issue', 'levels', 'light',
'living', 'network', 'on', 'reading', 'room', 'sensor',
'temperature', 'the', 'thermostat', 'to', 'turn']
```

5. Next, we'll map each word in the vocabulary to a unique index using dictionary comprehension. This `word_to_index` dictionary assigns each word a numerical identifier, which is useful for tasks such as encoding text for ML models:

```
word_to_index = {w: i for i, w in enumerate(vocab)}
```

6. Now, we'll define a `one_hot_encode` function to create a one-hot-encoded vector for a given token. This function initializes a vector of zeros with a length equal to the vocabulary size. If the token exists in the `word_to_index` dictionary, the corresponding index in the vector is set to 1. This encoding is demonstrated with the example token "thermostat". Here, we can see how it maps to a binary vector where only the index for "thermostat" has been activated:

```
def one_hot_encode(token, word_to_index):
    vector = [0]*(len(word_to_index))
    if token in word_to_index:
        vector[word_to_index[token]] = 1
    return vector

# Example: encode "thermostat"
thermostat_vector = one_hot_encode(
    "thermostat", word_to_index
)
print("One-hot for 'thermostat':", thermostat_vector)
```

Here's the one-hot-encoded 'thermostat' value:

```
One-hot for 'thermostat': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 1, 0, 0]
```

7. Now, let's try word embeddings. We'll use the Gensim library to train a `Word2Vec` model on the tokenized sentences. The model learns 50-dimensional vector representations (embeddings) for words based on their context while considering a window size of five words. After training for 10 epochs, we retrieve the embedding for the word 'thermostat', which represents its semantic meaning as a dense numerical vector. This vector can be used in downstream tasks such as similarity analysis or ML:

```
from gensim.models import Word2Vec

# Train a Word2Vec model
w2v_model = Word2Vec(
```

```

    sentences=tokenized_sentences,
    vector_size=50, window=5,
    min_count=1, workers=4
)
w2v_model.train(
    tokenized_sentences,
    total_examples=w2v_model.corpus_count,
    epochs=10
)

# Get embedding for a word
thermostat_embedding = w2v_model.wv["thermostat"]
print(
    "Word2Vec embedding for 'thermostat':\n",
    thermostat_embedding
)

```

Here, we can see the Word2Vec-encoded embedding vector for the word 'thermostat':

```

Word2Vec embedding for 'thermostat':
[-0.01427785  0.00248062 -0.01435238
-0.00448586  0.00743716  0.01166521
  0.00240026  0.00420425 -0.00822123  0.01444707
-0.01261343  0.00929421
-0.01643505  0.00406795 -0.0099551  -0.00849571
-0.00621964  0.01130703
  0.01159536
-0.00995255  0.0015448  -0.01698785  0.01562211  0.0185174
-0.00548613  0.00159988  0.00148695  0.01095375
-0.01721281  0.00116747
  0.01374174  0.00446148  0.00225555 -0.01864281  0.01696442
-0.0125275
-0.00598699  0.00698934 -0.00154697  0.00282321  0.00356525
-0.01365327
-0.01945076  0.01807933  0.01240023
-0.01382502  0.00680445  0.00041468
  0.00950904 -0.01424117]

```

8. Now, we'll try contextual embeddings. To do so, we'll load a pre-trained BERT model and tokenizer using the Hugging Face Transformers library. Here, BertTokenizer is responsible for converting raw text into tokens that BERT can understand, while BertModel provides a pre-trained deep learning model capable of generating embeddings for these tokens. By using the bert-base-uncased version, we ensure that the text is lowercase during tokenization and leverages a widely used general-purpose model:

```

from transformers import BertTokenizer, BertModel
import torch

```

```
# Load a pretrained BERT model
tokenizer = BertTokenizer.from_pretrained(
    'bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')
```

9. Let's tokenize the sentence "Turn on the living room light" using the BERT tokenizer. The tokenizer function converts the text into token IDs and attention masks required for input into the BERT model. The `return_tensors='pt'` argument ensures that the output is in PyTorch tensor format, which is necessary for feeding the data into the pre-trained BERT model:

```
sentence = "Turn on the living room light"
inputs = tokenizer(sentence, return_tensors='pt')
```

10. Next, we'll run the tokenized input through the BERT model in inference mode, using `torch.no_grad()` to save memory. The model returns contextualized embeddings for each token in the input sequence, stored in `last_hidden_state`. These embeddings capture the meaning of each token concerning its context, making them suitable for various NLP tasks:

```
with torch.no_grad():
    outputs = model(**inputs)
    # outputs is a tuple:
    # (last_hidden_state, pooler_output, hidden_states)
    last_hidden_state = outputs.last_hidden_state
```

11. Finally, we'll print the shape of the `last_hidden_state` tensor to understand the dimensions of the contextual embeddings. Here, we'll locate the index of the "light" token in the tokenized input while accounting for the [CLS] token offset. Using this index, we can extract the contextual embedding for "light" from the `last_hidden_state` tensor, showcasing the first 10 dimensions as an example of the token's rich semantic representation:

```
print(
    "Shape of contextual embeddings:",
    last_hidden_state.shape
)
# Retrieve the embedding for the token "light"
tokens = tokenizer.tokenize(sentence)
light_index = tokens.index(
    "light"
) + 1 # +1 for CLS token offset
light_embedding = last_hidden_state[
    0, light_index, :
].numpy()
print(
```

```

    "Contextual embedding for 'light':\n",
    light_embedding[:10], "...
) # Show first 10 dimensions for brevity

```

You can see the contextual embedding for the word 'light' here:

```

Shape of contextual embeddings: torch.Size([1, 8, 768])
Contextual embedding for 'light':
[ 0.16429132 -0.39750677 -0.19628845
 -0.12849177  0.9441986  -0.34701523
  0.19516525 -0.20958914 -0.24018592 -1.0789583 ] ...

```

Here, the embedding for "light" is influenced by the sentence "Turn on the living room light". If we had a different sentence, such as "The sensor emits a very light signal," the embedding for "light" would differ, reflecting its context (an attribute of the signal rather than a home appliance).

Choosing the right encoding

One-hot: This is good for very simple tasks or as a baseline, but it's rarely used in complex IoT scenarios.

Word embeddings (Word2Vec/GloVe): These offer a good balance of complexity and performance, which makes them suitable for many IoT use cases where you need semantic understanding but not necessarily deep context.

Contextual embeddings (BERT/DistilBERT): These are ideal for nuanced tasks such as intent detection, command disambiguation, and more complex language understanding in IoT, albeit more resource-intensive.

In the next few sections, we'll apply these embeddings to actual NLP tasks, such as intent classification and entity recognition, enabling IoT systems to interpret and act on text commands and logs more effectively.

NLU in IoT contexts

NLU is a branch of AI that helps machines interpret and understand human language. In the IoT world, NLU enables devices and systems to comprehend text-based instructions, alerts, and feedback. By extracting structured information – such as the user's intent, the device they are referring to, and the parameters they specify – NLU allows IoT systems to respond appropriately, making human-machine interaction more natural and efficient. We can employ various NLU techniques:

- **Intent recognition:** Distinguish between different user requests – such as turning on a device, checking sensor readings, or adjusting a setting. It determines what the user wants the system to do – for example, the intent of `Turn on the living room light` is `TURN_ON_DEVICE`.

- **Entity extraction:** Identify which device, parameter, or location the user is talking about (e.g., thermostat, living room, or 22 degrees). It identifies important words or phrases (entities). For example, the entity of `Turn on the living room light` is `living room light`. By combining intent and entity recognition, an IoT system can, for instance, turn on the correct device at the specified location or set the exact parameter requested by the user.
- **Semantic analysis:** Understand the broader meaning, sentiment, and topics present in IoT-related text data, such as user feedback on a smart home system. In **sentiment analysis**, the system determines whether user feedback is positive, negative, or neutral. For example, if a user says `The vacuum cleaner never starts when I tell it to`, the system can detect frustration and log negative sentiment. In **topic modeling**, the system identifies underlying themes in large sets of IoT logs. For example, analysis of thousands of maintenance logs might reveal that `overheating motors` and `network delays` are the most frequent categories of issues.

Approaches to NLU

To perform the NLU tasks mentioned in this section, you can use the following approaches:

Classical ML: Models such as logistic regression, SVM, or Naive Bayes. These are good starting points but may require manual feature engineering.

Deep learning (RNNs, LSTMs, BiLSTMs, CNNs): These learn features from text automatically and generally outperform classical ML on large datasets.

Transformers (BERT, DistilBERT): State-of-the-art models that use self-attention to understand context. These models can be fine-tuned on custom IoT data for superior accuracy in intent classification and entity recognition tasks.

NLU pipeline implementation using a transformer

To demonstrate how to build an NLU pipeline, we'll create a small synthetic dataset of IoT commands. Then, we'll use a pretrained BERT model to classify the intent of these commands.

Here, we'll create a synthetic dataset of user commands directed at a smart home. Each command will have a corresponding intent label. In a real-world scenario, you would have a labeled dataset of IoT commands (e.g., collected from user logs or manual annotations):

1. We'll start by importing the necessary libraries for text classification using a pre-trained BERT model. This includes `pandas` for handling data, `PyTorch` for tensor operations, and Hugging Face's tools such as `AutoTokenizer`, `AutoModelForSequenceClassification`, and `Trainer` for fine-tuning. Additionally, we'll use `sklearn` to split data and datasets so that we can structure it for training:

```
import pandas as pd
import torch
```

```
from transformers import (
    AutoTokenizer, AutoModelForSequenceClassification,
    Trainer, TrainingArguments
)
from sklearn.model_selection import train_test_split
from datasets import Dataset
```

2. Next, we'll create a synthetic dataset of IoT commands and their corresponding intents to simulate device control and monitoring tasks. This data is stored in a pandas DataFrame, where each command (e.g., "turn on the living room light") is paired with an intent label (e.g., "TURN_ON_DEVICE"). This dataset forms the basis for training and evaluating a text classification model:

```
commands = [
    "turn on the living room fan",
    "turn on the bedroom light",
    "turn off the kitchen fan",
    "turn off the bedroom light",
    "set thermostat to 22 degrees",
    "increase humidity by 5 percent",
    "decrease brightness in the hallway",
    "check the bedroom temperature",
    "what is the current humidity level?"
]

intents = [
    "TURN_ON_DEVICE",
    "TURN_ON_DEVICE",
    "TURN_OFF_DEVICE",
    "TURN_OFF_DEVICE",
    "SET_TEMPERATURE",
    "ADJUST_PARAMETER",
    "ADJUST_PARAMETER",
    "QUERY_STATUS",
    "QUERY_STATUS"
]

df = pd.DataFrame({
    'command': commands,
    'intent': intents
})
```


- Now, we'll map each unique intent to a numeric label using a dictionary and add a new label column to the DataFrame. This mapping (`label_to_id`) is useful for encoding the intent categories as integers, which are required for training ML models. The resulting DataFrame now includes both the textual intent and its corresponding numeric label:

```
label_to_id = {
    label: i for i,
    label in enumerate(df['intent'].unique())
}
id_to_label = {v: k for k, v in label_to_id.items()}
df['label'] = df['intent'].map(label_to_id)
```

- Next, we'll split the dataset into training and testing sets using an 80-20 split. The `train_test_split` function ensures the data is shuffled and divided, with the training set used to train the model and the test set reserved for evaluating its performance. The split data is then printed for review:

```
train_df, test_df = train_test_split(
    df, test_size=0.2, random_state=42
)
```

- At this point, we'll convert the training and testing DataFrames into Hugging Face Dataset objects, making them compatible with the Trainer API. Then, we'll remove unnecessary columns such as intent and metadata columns (`__index_level_0__`), keeping only the essential command and label columns needed for model training and evaluation:

```
train_dataset = Dataset.from_pandas(train_df)
test_dataset = Dataset.from_pandas(test_df)

# Keep only necessary columns for the model
train_dataset = train_dataset.remove_columns([
    "intent", "__index_level_0__"
])
test_dataset = test_dataset.remove_columns([
    "intent", "__index_level_0__"
])
```

6. Now, we'll use `AutoTokenizer` to tokenize the command text in the training and test datasets. Here, `tokenize_function` applies padding and truncation to ensure all sequences are of a uniform length (`max_length=16`). The tokenized datasets are then stripped of the original command text, leaving only the encoded inputs (`input_ids`, `attention_mask`) and labels. Finally, the datasets are formatted into PyTorch tensors, making them ready for model training and evaluation:

```
tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')

def tokenize_function(example):
    return tokenizer(
        example["command"],
        truncation=True,
        padding='max_length',
        max_length=16
    )
train_encoded = train_dataset.map(
    tokenize_function, batched=True
)
test_encoded = test_dataset.map(
    tokenize_function, batched=True
)
train_encoded = train_encoded.remove_columns(["command"])
test_encoded = test_encoded.remove_columns(["command"])

train_encoded.set_format(
    "torch",
    columns=["input_ids", "attention_mask", "label"]
)
test_encoded.set_format(
    "torch",
    columns=["input_ids", "attention_mask", "label"]
)
```

7. Next, we'll initialize a pre-trained BERT model for sequence classification using `AutoModelForSequenceClassification`. The model is configured with a classification head that has the same number of output labels as the unique intents in our dataset (`num_labels=len(label_to_id)`). This setup tailors the model for predicting the intent of IoT commands:

```
model = AutoModelForSequenceClassification.from_pretrained(
    'bert-base-uncased',
    num_labels=len(label_to_id)
)
```

8. Now, we'll define `TrainingArguments` to configure the training process, specifying parameters such as the number of epochs (`num_train_epochs=10`), batch size (`per_device_train_batch_size=4`), and evaluation strategy (`evaluation_strategy="steps"`). The `Trainer` class will be initialized with the BERT model, training arguments, and the tokenized training and evaluation datasets. This setup streamlines the fine-tuning process for the intent classification task:

```
training_args = TrainingArguments(
    output_dir="results",
    evaluation_strategy="steps",
    num_train_epochs=10,
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    logging_steps=10,
    load_best_model_at_end=True,
    save_total_limit=1,
    save_steps=10
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_encoded,
    eval_dataset=test_encoded
)
```

9. Let's initiate the training process using `trainer.train()`, which fine-tunes the BERT model on the training dataset based on the provided arguments. After training, we'll evaluate the model on the test dataset with `trainer.evaluate()`, assessing its performance on unseen data. This completes the training workflow for intent classification:

```
trainer.train()
trainer.evaluate()
```

Here are the training logs:

```
Step   Training Loss   Validation Loss
[40/40 02:28, Epoch 20/20]
Step   Training Loss   Validation Loss
10     1.146800       1.650992
20     0.605900       1.992197
30     0.314000       1.704657
40     0.230800       1.639256
[1/1 : < :]
{'eval_loss': 1.6392557621002197,
```

```
'eval_runtime': 0.2583,
'eval_samples_per_second': 7.742,
'eval_steps_per_second': 3.871,
'epoch': 20.0}
```

10. Finally, we'll test the fine-tuned model with a new command, "turn on the fan". The command is tokenized and prepared as input for the BERT model, excluding "token_type_ids" since they are unnecessary for single-sentence inputs. In inference mode (`torch.no_grad()`), we pass the inputs to the model and extract the predicted label by taking the argmax of the logits. The predicted label ID is then mapped back to its corresponding intent using `id_to_label`, and we print both the command and its predicted intent. The model correctly identifies the intent as "TURN_ON_DEVICE":

```
test_command = "turn on the fan"
inputs = tokenizer(
    test_command,
    return_tensors="pt",
    truncation=True,
    padding='max_length',
    max_length=32
)
# Remove token_type_ids from inputs
inputs = {
    key: value
    for key, value in inputs.items()
    if key != "token_type_ids"
}
with torch.no_grad():
    outputs = model(**inputs)
    predicted_label_id = outputs.logits.argmax(
        dim=-1
    ).item()

predicted_intent = id_to_label[predicted_label_id]
print(f"Command: {test_command}")
print(f"Predicted Intent: {predicted_intent}")
```

Let's see the command that was identified and the predicted intent from the model:

```
Command: turn on the fan
Predicted Intent: TURN_ON_DEVICE
```

Extending to entity recognition

For entity recognition, you'd typically use a token classification model (e.g., `AutoModelForTokenClassification`) and label each token with entity tags (e.g., B-DEVICE, I-DEVICE, B-VALUE, etc.). You could create a similar synthetic dataset where each word in the command is labeled. After training, the model would identify which part of the command corresponds to a device ("living room light"), a value ("22 degrees"), or a parameter ("humidity").

With appropriately labeled data, you can extend the same logic for semantic analysis and topic modeling.

As IoT devices become more intelligent, integrating **large language models (LLMs)** – such as GPT, Claude, or LLaMA – can dramatically enhance how we **interact with, interpret, and automate** IoT systems. LLMs are built on **transformer architectures**, which allow them to process long-range dependencies in text and generate context-aware responses. Here's a quick example of using OpenAI's API to extract intents/entities from user input:

```
import openai
import os
# Set your API key as an environment variable
os.environ["OPENAI_API_KEY"] = "Your_Open_AI_Key"
client = openai.OpenAI()
def analyze_command(command):
    response = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=[{
            "role": "user",
            "content": f"Extract the intent and device
                        from this IoT command: '{command}'"
        }]
    )
    return response.choices[0].message.content
# Example usage
print(analyze_command("Turn on the greenhouse lights"))
```

You'll get a structured natural language response similar to the following:

```
Intent: TURN_ON_DEVICE; Device: greenhouse lights
```

Additionally, correct prompting can enable LLMs to summarize lengthy sensor logs. An IoT assistant can be developed by prototyping with voice UI. Combining LLM outputs with lightweight logic (rules/regex/APIs) creates hybrid systems that are both **smart and controllable**. Once validated, these LLM prototypes can be transitioned to fine-tuned, lightweight models (such as `DistilBERT`) for deployment.

Working with Audio Data in IoT

As IoT devices become more integrated into daily life, the ability to interact using voice commands and process environmental sounds is increasingly valuable. From smart home assistants that respond to spoken requests to industrial sensors that listen for machine anomalies, audio data opens up hands-free, intuitive, and accessible interaction channels.

This section will guide you through the end-to-end process of working with audio data for IoT applications.

Audio processing and feature extraction

Raw audio waveforms are challenging for AI models to handle directly. Preprocessing steps enhance signal quality and extract representative features. Some of the important steps include the following:

- **Noise reduction:** Filter out background noise or static.
- **Normalization:** Adjust volume levels to a consistent range.
- **Silence removal:** Trim silent segments at the start or end of recordings.
- **Feature extraction:** Convert audio signals into time-frequency representations (e.g., **Mel-Frequency Cepstral Coefficients (MFCCs)**). MFCCs approximate human auditory perception, making them widely used in speech and audio recognition.

Audio command recognition

Audio command recognition involves classifying short spoken utterances into predefined categories or actions. For IoT, these actions might be “turn on device,” “turn off device,” “increase temperature,” or “query status.”

To build an effective audio command recognition system, you first need to prepare your audio clips so that each one contains a single, clearly spoken command. This ensures that your model learns to classify distinct utterances rather than grappling with multiple instructions in one sample. Once you have these clean, isolated clips, the next step is to extract meaningful features. Common approaches involve converting raw waveforms into MFCCs or generating spectrogram representations. These transformations highlight the frequency and temporal characteristics of the audio, making it easier for neural networks to recognize patterns.

After feature extraction, the next step is to train a classifier. Neural network architectures such as CNNs, RNNs, or transformer-based models can all be effective. A CNN, for example, can handle MFCCs or spectrograms as 2D inputs, making it well-suited for image-like feature matrices. Once trained, your model will be able to predict the class of an incoming audio command. Based on this prediction, you can trigger the corresponding IoT action in real time – turning on a device, adjusting a parameter, or querying a status – thereby enabling seamless voice-controlled interactions.

When refining your model, consider several key tips. CNNs are often a good starting point for audio tasks, thanks to their ability to automatically learn feature hierarchies from spectrogram-like inputs. To improve your model's robustness, apply regularization techniques and data augmentation strategies, such as adding background noise or shifting audio clips slightly in time. This helps the model generalize better to real-world conditions. If you only have a limited amount of data, look into transfer learning or pretrained audio models that can leverage knowledge learned from large-scale datasets. With these practices in place, you can build a more accurate, reliable, and flexible audio command recognition system for your IoT applications.

Speech recognition and transcription

When your IoT device must handle more complex queries than simple fixed commands, **automatic speech recognition (ASR)** becomes essential. By converting spoken language into text, ASR unlocks the ability to apply various NLP techniques – such as intent classification and entity extraction – to fully understand user requests. You can use open source ASR models such as *Wav2Vec2* or *Whisper* for on-device processing, or rely on cloud-based APIs from providers such as AWS and Google for high-quality transcription with minimal setup. Once you have the transcribed text, all the NLP techniques discussed earlier (e.g., extracting the user's intent, identifying parameters, or recognizing named entities) can be applied to the query. Tools such as Hugging Face *Wav2Vec2* models for on-device ASR, the *SpeechRecognition* library for integrating with different engines, and reliable cloud services all help streamline this process, enabling your IoT system to respond to nuanced voice commands effectively.

Voice assistants and IoT integration

Voice assistants, such as Amazon's Alexa or Google Assistant, offer a powerful paradigm for IoT integration by already combining ASR, NLU, and **text-to-speech (TTS)** capabilities. By leveraging their built-in language processing pipelines, you can connect these voice assistants to IoT APIs so that users can interact with devices solely through voice commands. This approach not only simplifies user interaction but also broadens accessibility, allowing for hands-free control and immediate responses from a variety of connected systems.

One common approach is to build a custom skill or action for the voice assistant platform you've chosen. In this method, you define specific intents – such as “Play the song *Bad Habits* by Ed Sheeran” or “Switch on the geyser” – and map them directly to IoT actions. When a user utters a command that matches one of your defined intents, the assistant then calls the corresponding IoT API and carries out the requested task. For those concerned about privacy or latency, running a lightweight ASR model on an edge device, such as a Raspberry Pi, can keep the entire voice pipeline local, minimizing dependency on cloud services and preserving user data on-premises.

Ultimately, once the assistant identifies the user's intent, it seamlessly triggers the appropriate IoT action. Whether it's adjusting the lighting, controlling home appliances, or checking sensor data, the combination of voice assistants and IoT integration creates a convenient, efficient, and intuitive user experience.

Using audio commands to control an IoT device

Consider that you have an IoT smart plug controlling a kitchen fan. A user says, "Turn on the kitchen fan." We'll show a simplified example using a pretrained speech recognition model and a simple classifier for commands.

In this example, we'll use a Hugging Face ASR model (e.g., a Wav2Vec2 variant) for transcription and a simple if-statement as a classifier. In practice, you'd use a more robust classifier trained on multiple classes. For example, you can use the intent classifier that we trained in the previous section to detect intents:

1. We'll start by importing the necessary libraries for speech-to-text processing. This includes `torch` for handling tensors, `librosa` for audio processing, and `soundfile` for reading and writing audio files. From Hugging Face, we'll import `Wav2Vec2Processor` for audio preprocessing and the `Wav2Vec2ForCTC` model for transcribing speech into text. Additionally, `requests` will be included so that audio files can be downloaded if needed:

```
import torch
import librosa
from transformers import (
    Wav2Vec2Processor, Wav2Vec2ForCTC
)
import requests
import soundfile as sf
```

2. Next, we'll define a mock function, `control_iot_device`, to simulate controlling an IoT device based on an intent. It handles specific intents such as `"TURN_ON_DEVICE"` and `"TURN_OFF_DEVICE"` by printing corresponding actions for the specified device (e.g., turning a fan on or off). For unrecognized intents, it prints a default `Unknown intent` message, mimicking a simple IoT control system:

```
def control_iot_device(intent, device):
    if intent == "TURN_ON_DEVICE":
        print(f"Action: Turning on the {device}.")
    elif intent == "TURN_OFF_DEVICE":
        print(f"Action: Turning off the {device}.")
    else:
        print("Action: Unknown intent.")
```


- Now, we'll load an audio file (`speech.wav`) using `librosa.load`, resampling it to a sampling rate of 16 kHz (`sr=16000`), as required by the `Wav2Vec2` model. The audio data is stored in `y` as a NumPy array, while `sr` confirms the sampling rate used. This prepares the audio for preprocessing and transcription:

```
audio_path = 'speech.wav'
y, sr = librosa.load(audio_path, sr=16000)
```

- Next, we'll load the pre-trained `Wav2Vec2` processor and model from Hugging Face. Here, `Wav2Vec2Processor` handles audio preprocessing, such as feature extraction and tokenization, while the `Wav2Vec2ForCTC` model performs speech-to-text transcription. The specific model, "`facebook/wav2vec2-base-960h`", will be trained on 960 hours of English speech data and optimized for general speech recognition tasks:

```
processor = Wav2Vec2Processor.from_pretrained(
    "facebook/wav2vec2-base-960h"
)
model = Wav2Vec2ForCTC.from_pretrained(
    "facebook/wav2vec2-base-960h"
)
```

- At this point, we'll preprocess the audio data using `Wav2Vec2Processor`, converting it into a format suitable for the model, including padding and PyTorch tensors. The processed audio is then passed through the `Wav2Vec2ForCTC` model in inference mode (`torch.no_grad()`), generating logits as output. By taking `argmax` over the logits, we obtain the predicted token IDs, which are decoded back into human-readable text using the processor. Finally, we'll print the transcribed text, which represents the spoken content of the audio:

```
inputs = processor(
    y, sampling_rate=sr,
    return_tensors="pt",
    padding="longest"
)
with torch.no_grad():
    logits = model(inputs.input_values).logits
    pred_ids = torch.argmax(logits, dim=-1)
    transcription = processor.decode(pred_ids[0])
    print("Transcription:", transcription)
```

In human-readable format, the input is as follows:

```
Transcription: TURN ON THE KITCHEN FAN
```

- Let's analyze the transcribed text to determine the intent and target device. If the transcription contains "turn on" and "fan", we classify the intent as "TURN_ON_DEVICE" for "kitchen fan". Similarly, if it contains "turn off" and "fan", we classify the intent as "TURN_OFF_DEVICE". If neither condition is met, the intent is marked as "UNKNOWN", and the device remains unspecified. This logic demonstrates a **simple rule-based approach** to intent recognition. You can replace this rule-based approach with the fine-tuned intent classification model we trained in the *NLU pipeline implementation using a transformer* section:

```
command_text = transcription.lower()
if "turn on" in command_text and "fan" in command_text:
    intent = "TURN_ON_DEVICE"
    device = "kitchen fan"
elif "turn off" in command_text and "fan" in command_text:
    intent = "TURN_OFF_DEVICE"
    device = "kitchen fan"
else:
    intent = "UNKNOWN"
    device = ""
```

- After determining the intent and device from the transcription, we'll call the `control_iot_device` function with the identified intent and device. In this case, the transcription "TURN ON THE KITCHEN FAN" results in "TURN_ON_DEVICE" as the intent and "kitchen fan" as the device:

```
control_iot_device(intent, device)
Action: Turning on the kitchen fan.
```

Extending the approach

You can integrate entity extraction to identify location ("kitchen") or parameters ("increase speed"). You can also use a real IoT API endpoint rather than a `print` statement.

In this section, we explored how to handle audio data from start to finish. After preprocessing and feature extraction, you can build audio command recognition models or use speech recognition to obtain text for NLU. Ultimately, you can integrate these capabilities into a voice assistant framework to control IoT devices seamlessly. The provided code example outlines a basic pipeline that can be refined and expanded as your IoT ecosystem and requirements grow.

Summary

This chapter focused on handling and analyzing non-visual IoT data – specifically text, audio, and speech. It began by highlighting the importance of textual alerts, logs, and voice commands in IoT ecosystems, emphasizing how these modalities provide more intuitive user interfaces and richer context than raw numeric sensor readings. We covered fundamental preprocessing steps such as normalization, tokenization, stopword removal, and lemmatization, ensuring that textual data is clean and ready for analysis. We also introduced various encoding methods, from simple one-hot vectors to advanced contextual embeddings such as BERT, enabling more robust NLU.

Beyond text, we delved into audio processing and feature extraction. We covered how to classify audio commands, leverage speech recognition to transcribe spoken requests, and apply NLU to interpret them. The integration of voice assistants and IoT APIs was also discussed, showing how ASR, NLU, and device control can create seamless, voice-driven IoT experiences.

In the next chapter, we'll pivot to personal and home IoT applications. There, we'll delve into wearable fitness devices and smart home technologies, address privacy concerns, and explore integrating home IoT with smart assistants to create a more connected, secure, and user-centered living environment.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/ds>



Part 4: Applying AI and IoT in Real-World Scenarios

The final part of the book focuses on practical use cases across personal health, industrial automation, and smart city systems. Through case studies and hands-on examples, we will see how AI and IoT can be used together to drive sustainability, efficiency, and innovation in real-world environments.

This part has the following chapters:

- *Chapter 13, AI for Personal and Home IoT*
- *Chapter 14, AI for Industrial IoT*
- *Chapter 15, AI for Smart Cities IoT*

AI for Personal and Home IoT

In this chapter, we explore the landscape of personal and home **Internet of Things (IoT)** applications. Personal IoT devices such as wearables and fitness trackers are revolutionizing health monitoring, while smart home gadgets are automating our living spaces. We delve into how **artificial intelligence (AI)** enhances these IoT devices – from predicting health anomalies with smartwatches to automating home climate and lighting. We examine the role of edge computing in enabling real-time responses and protecting user data privacy by processing information locally. Throughout the chapter, Python-based examples illustrate how to analyze wearable data and control smart home devices. We also address critical privacy and security concerns in personal IoT ecosystems, highlighting encryption, data anonymization, and ethical AI practices to safeguard users. Case studies of real-world implementations are provided to ground these concepts: from a smartwatch detecting a heart condition to a voice assistant empowering a smart home for a user with disabilities. By the end of the chapter, you will have an advanced understanding of personal and home IoT applications, the AI and edge techniques that power them, and strategies to manage privacy and security in this domain.

In this chapter we look at:

- Wearable and Fitness Devices
- Smartwatch Health Monitoring in Action
- Smart Home Devices
- Privacy Concerns in Personal IoT
- Getting Started with Your Own Smart IoT Projects

Technical Requirements

To run the code examples in this chapter, no additional installations are required. You'll be using `scikit-learn`, `pandas`, `numpy`, and `matplotlib`, which you've already installed. Access the chapter's code examples at this link: <https://github.com/PacktPublishing/Hands-On-Artificial-Intelligence-for-IoT-Second-Edition/tree/main/Chapter13>.

Wearable and Fitness Devices

Wearable IoT devices have become mainstream tools for health and fitness tracking. Fitness trackers and smartwatches are common examples – these gadgets continuously monitor metrics such as steps taken, heart rate, sleep patterns, and calories burned. Advances in sensors and miniaturization allow wearables to capture vital signs (heart rate, blood oxygen, skin temperature, and so on) non-invasively. Modern smartwatches not only count steps but also include features such as **electrocardiogram (ECG)** and blood oxygen sensors, bridging the gap between consumer wellness devices and medical-grade monitors. Deloitte Global (<https://www2.deloitte.com/us/en/insights/industry/technology/technology-media-and-telecom-predictions/2022/wearable-technology-healthcare.html>) estimated that 320 million consumer health and wellness wearables would ship in 2022 (up from earlier years), and this could reach 440 million by 2024. Recent data suggests this figure may be surpassed. For instance, IDC (<https://www.idc.com/getdoc.jsp?containerId=prUS52322724>) reported 113.1 million shipments in just the first quarter of 2024, indicating a potential for exceeding initial yearly forecasts. This growth reflects how people increasingly use wearables not just for fitness goals (such as step counts or running pace) but for more sophisticated health monitoring, such as tracking heart rhythms for potential arrhythmias, managing chronic conditions such as diabetes through continuous glucose monitoring, or even detecting early signs of Parkinson's tremors. The market is also seeing growth in specialized devices such as medical-grade patches and wearables with health monitoring capabilities, contributing to the overall expansion. This evolution is driven by advancements in sensor technology, AI-driven analytics that provide personalized health insights, and increasing clinical validation of these devices. While challenges remain, such as data privacy concerns and the need for greater physician integration, the future of wearable health technology appears promising, with projections estimating a market size of \$386.5 billion by 2033 (<https://www.meticulousresearch.com/product/wearable-technology-market-5320>).

Wearable health devices come in various forms, such as activity bands, smartwatches, smart patches, smart clothing, and many others.

Many healthcare providers are beginning to incorporate data from consumer wearables into patient care. For instance, patients may share fitness tracker data with doctors to monitor lifestyle factors or use **Food and Drug Administration (FDA)**-cleared smartwatch features to catch early signs of heart disease. Overall, wearables serve as personal IoT health nodes, **collecting real-time data** that can promote preventive care and healthier habits.

AI integration in wearables for health insights

Raw data from wearable devices becomes far more powerful when combined with AI. AI algorithms – particularly **machine learning (ML)** – can analyze streams of sensor data to derive health insights and even predictions. A prominent example is the Apple Watch's irregular heart rhythm notification feature, which uses an AI algorithm to detect patterns of atrial fibrillation in heart rate data. This feature received FDA approval in 2018 and has been credited with alerting users to seek medical help

for hidden heart issues. In one reported case, an Apple Watch's heart monitoring app warned a user of an abnormal heart rate, prompting a hospital visit that revealed a serious condition – essentially, the smartwatch predicted a potential heart problem before symptoms became obvious. (You will get more details in the *Smartwatch health monitoring in action* section.)

Predictive analytics in wearables can go beyond heart health. Researchers are training models for a range of applications:

- **Activity recognition:** ML models can classify what activity a person is doing (walking, running, sitting, cycling) based on motion sensor patterns. Accelerometer and gyroscope data from a smartwatch can feed a classification model (such as a **convolutional neural network**, or **CNN**) to recognize activities or exercises. This helps provide automated workout logging or coaching. For instance, a trained model could detect when the user starts jogging versus cycling by recognizing distinct motion “signatures.”
- **Anomaly detection:** AI can learn an individual's normal vital patterns and flag anomalies. For example, a wearable's continuous heart rate variability data can be analyzed to predict cardiac irregularities or stress events. If the system learns your typical resting heart rate and circadian rhythm, it can raise an alert when your heart rate is elevated at rest or your temperature is abnormally high, potentially indicating illness.
- **Fall and accident detection:** Especially for elderly users, wearables use AI to detect falls by analyzing sudden accelerometer drops and impacts. Many smartwatches now have fall-detection algorithms that notify emergency contacts if a hard fall is detected and the user doesn't respond in time. These algorithms often use ML models trained on motion data for falls versus normal movements, achieving high sensitivity in controlled tests (often >90% accuracy).
- **Health event prediction:** Data can be used to track changes in heart rate patterns and oxygen saturation, which helps predict atrial fibrillation episodes or asthma attacks before they fully occur. ML can also correlate patterns (such as increased resting heart rate plus decreased activity levels) with possible infections or chronic condition flare-ups, enabling early intervention.

These features turn wearables from passive data loggers into proactive health assistants. Instead of just displaying graphs, the devices provide insights, notifying you that “your heart rate pattern looks abnormal today” or “you haven't slept well lately, which correlates with your elevated stress markers.” Notably, understanding the importance of AI is essential in this context – users and doctors need to trust and understand the reasoning behind alerts. Simpler models such as decision trees or rule-based systems are sometimes favored for interpretability when informing medical decisions.

Despite promising results, deploying AI for wearables faces challenges. Models must be robust to noisy data from daily life and diverse users. False positives (for example, a smartwatch falsely detecting a fall or heart issue) can annoy users or cause alarm, so algorithms are tuned to balance sensitivity and specificity. Moreover, personalization is key – AI models may adapt to an individual's baseline by learning from their data over time (a form of on-device training or **transfer learning** (TL)) so that the system gets more accurate for that user. We'll discuss next how edge computing facilitates such personalization without compromising privacy.

Edge computing for real-time health monitoring

Wearable devices generate data continuously and often need to react in real time (for instance, to alert a user of a heart rate spike or a fall immediately). Edge computing refers to processing data on the device itself or on nearby local nodes (such as a smartphone or home gateway) instead of sending everything to a cloud server. In personal health IoT, edge computing has several benefits:

- **Low latency:** By analyzing data on-device or on a phone, wearables can provide instant feedback without the delay of uploading to the cloud. For a use case such as detecting an arrhythmia or a fall, milliseconds matter – local processing triggers immediate alerts (vibration, sound, or message to emergency services) with minimal lag.
- **Privacy preservation:** Health data is sensitive. Keeping the data local (on the user's watch/phone) means raw sensor streams need not be transmitted over the internet at all. This reduces exposure of personal data. For example, a stress monitoring app could analyze your heart rate and microphone data locally to detect stress in your voice without ever uploading those voice snippets to a server. Edge processing keeps the data on the patient's device, which is safer from a privacy perspective.
- **Bandwidth and power efficiency:** Constantly streaming high-frequency sensor data (such as accelerometer readings at 50 Hz) to the cloud can be power-intensive for a wearable and consume bandwidth. Edge computing allows devices to send only meaningful results or summaries to the cloud instead of raw data. For instance, a wearable might locally determine “no fall detected in the last minute” rather than streaming the full accelerometer trace. This selective upload conserves battery and network resources.
- **Robustness:** Edge devices can function offline or with intermittent connectivity. A fall detector in a smartwatch will still work in a remote area with no internet since it doesn't rely on a cloud connection. Similarly, local AI means the service isn't lost if a cloud server is unreachable. The system is more resilient, which is crucial for health or safety functions.

There are trade-offs to edge computing on wearables. Watches and fitness bands have limited CPU and memory, constraining the size and complexity of AI models that can run on them. Techniques such as model compression, efficient **neural network (NN)** architectures, or even hardware accelerators (special chips for AI in phones/watches) are used to overcome these limits. In some designs, a hybrid approach is used: the wearable does basic processing and filtering, then sends intermediate results to either a smartphone (acting as an edge gateway) or the cloud for heavier analysis. For example, a wearable ECG patch might do onboard signal preprocessing and detection of obvious anomalies but transmit summary metrics to a phone app that has a more powerful AI model to make a detailed assessment.

An alternative edge strategy is to offload computation to nearby edge devices such as smartphones or home hubs. A smartwatch might relay sensor data to your phone over Bluetooth; the phone, with its better processor, runs the AI model (this is sometimes called **fog computing**, an intermediate layer between the device and the cloud). This still retains the benefits of local control – the data stays in your personal devices. Research has shown that edge/fog computing for wearables can significantly reduce the amount of data sent to the cloud, thereby improving privacy and reducing cloud costs.

Now that we are familiar with edge computing and how it empowers wearables to be responsive and privacy-conscious, let's explore how we can analyze the data generated by these devices using Python.

Analyzing wearable data with Python

From a developer or data scientist's perspective, working with wearable data often involves Python, given its rich ecosystem of data analysis and ML libraries. Once data is collected from devices (either via device APIs or exported files), Python can be used to clean, visualize, and draw insights from this data. Let's consider an example workflow of analyzing fitness tracker data using Python. For this example, we are using data exported from Apple Health collected over a 3-month period, January 2020-March 2020:

1. We start by importing the necessary libraries for our project: `pandas` and `numpy` for data manipulation and numerical operations; `matplotlib` and `seaborn` for visualization; and several modules from `scikit-learn`, including `StandardScaler` for feature scaling, `train_test_split` for dividing our dataset, `RandomForestRegressor` for building our ensemble regression model, and metrics such as `mean_absolute_error` and `mean_squared_error` for evaluating performance:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import (
    mean_absolute_error, mean_squared_error
)
```

2. Next, we load our dataset from the CSV file into a `pandas DataFrame` while parsing the 'Date' column as datetime objects. Then, we examine the shape of our `DataFrame` to understand its dimensions, check the data types and overall structure with the `info()` method, and finally, review the count of missing values per column to assess data quality before proceeding further:

```
df = pd.read_csv(
    'Apple-Fitness-Data.csv',
    parse_dates=['Date']
)
print("DataFrame Shape:", df.shape)

print("\nDataFrame Info:")
df.info()
print("\nMissing Values per column:")
print(df.isnull().sum())
```

Here is a screenshot of the output:

```
DataFrame Shape: (1412, 6)

DataFrame Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1412 entries, 0 to 1411
Data columns (total 6 columns):
#   Column      Non-Null Count  Dtype
---  ---
0   Date        1412 non-null   datetime64[ns]
1   Time        1412 non-null   object
2   Steps       1412 non-null   float64
3   Distance    1412 non-null   float64
4   Energy      1412 non-null   float64
5   Speed       1412 non-null   float64
dtypes: datetime64[ns](1), float64(4), object(1)
memory usage: 66.3+ KB

Missing Values per column:
Date      0
Time      0
Steps     0
Distance  0
Energy    0
Speed     0
dtype: int64
```

Figure 13.1: Descriptive details of the DataFrame

3. After cleaning our data, we begin by stripping any extra whitespace from the 'Time' column to ensure consistency. Following this, we convert the 'Time' values into proper datetime objects using the specified format so that they can be effectively utilized in time-based analyses. Finally, we examine the distribution of our key variables – 'Steps', 'Distance', 'Energy', and 'Speed' – by generating descriptive statistics, which helps us understand the central tendencies and variability of these metrics:

```
df['Time'] = df['Time'].str.strip()

df['Time'] = pd.to_datetime(
    df['Time'],
    format='%H:%M:%S'
).dt.time

print("\nDescriptive Statistics for Key Variables:")
key_variables = ['Steps', 'Distance', 'Energy', 'Speed']
print(df[key_variables].describe())
```

The following screenshot shows the statistical details of our data:

Descriptive Statistics for Key Variables:				
	Steps	Distance	Energy	Speed
count	1412.000000	1412.000000	1412.000000	1412.000000
mean	120.076487	0.068727	2401.897868	2.741370
std	173.881897	0.099386	3477.745410	1.302993
min	0.000000	0.000000	0.011963	0.511035
25%	14.000000	0.007423	272.880317	1.599697
50%	34.000000	0.020665	685.212173	2.709679
75%	166.250000	0.096437	3327.710405	3.887836
max	1103.000000	0.714210	22065.907901	4.997037

Figure 13.2: Descriptive statistics of the data

4. Subsequently, we explore the relationships between our numerical variables by first selecting all numeric columns from the DataFrame. We compute a correlation matrix to understand how these variables relate to each other, and then we visualize this matrix using a heatmap; this helps us quickly identify any strong or weak correlations in our data:

```
print("\nCorrelation Matrix for Numerical Variables (excluding  
Time):")  
numerical_cols = df.select_dtypes(  
    include=['number']  
) .columns  
plt.figure(figsize=(10, 8))  
corr = df[numerical_cols].corr()  
sns.heatmap(corr, annot=True, cmap='coolwarm', fmt=".2f")  
plt.title('Correlation Heatmap')  
plt.show()
```

A plot of the correlation matrix is shown next; we can see that steps, distance, and energy (burned) are quite correlated.

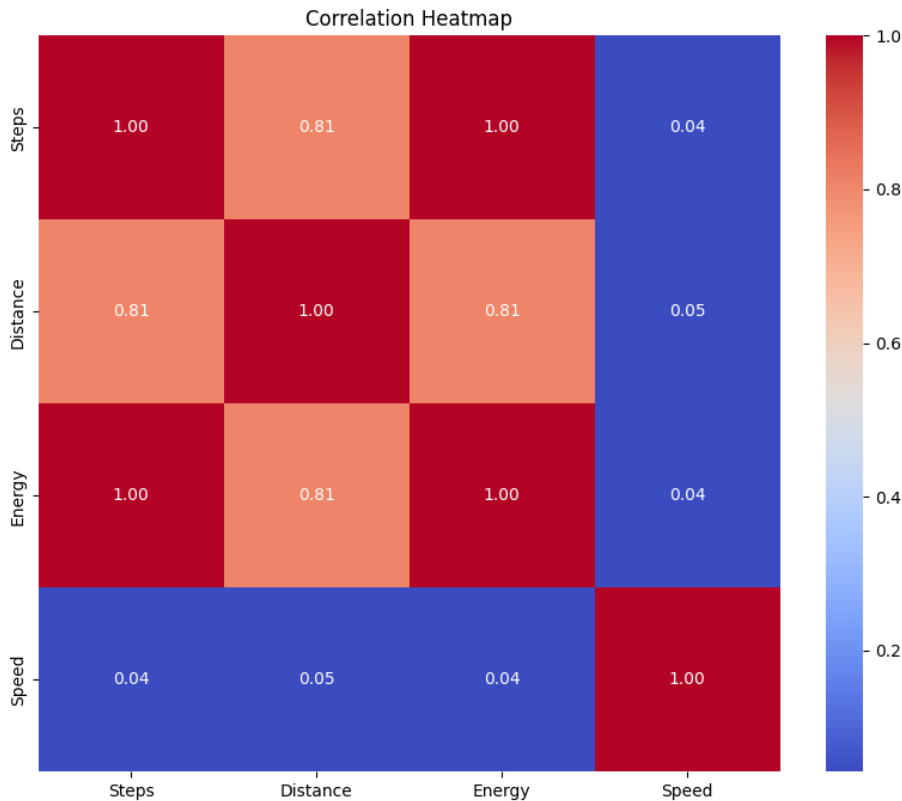


Figure 13.3: Heatmap of the correlation matrix

5. Next, we combine the 'Date' and 'Time' columns to create a unified 'Datetime' column—this prepares our data for time series analysis. To ensure data quality, we remove any duplicate records from the DataFrame:

```
df['Datetime'] = pd.to_datetime(
    df['Date'].dt.strftime('%Y-%m-%d') + ' ' +
    df['Time'].astype(str)
)
df.drop_duplicates(inplace=True)
```

6. We then enrich our dataset by creating additional features such as 'Avg Steps per Day', 'Day of Week', 'Hour of Day', 'Rolling Avg Steps (7 days)', 'Activity Level', and 'Active'—all aimed at improving our model's ability to learn from the data:

```
df['Avg Steps per Day'] = df.groupby('Date')['Steps']
    .transform('mean')

df['Day of Week'] = df['Datetime'].dt.day_name()
df['Hour of Day'] = df['Datetime'].dt.hour

df['Rolling Avg Steps (7 days)'] = df[
    'Steps'
].rolling(
    window=7, min_periods=1
).mean().fillna(method='bfill')

def categorize_activity(steps):
    if steps < 2000:
        return 'Low'
    elif steps < 7000:
        return 'Medium'
    else:
        return 'High'

df['Activity Level'] = df['Steps'].apply(
    categorize_activity
)
df['Active'] = (df['Steps'] > 5000).astype(int)
```

7. We plot the daily step counts over time using our data and iterate over a list of key variables—'Steps', 'Distance', 'Energy', and 'Speed'—to create histograms with kernel density estimates, which help us understand the distribution of each variable. Finally, we generate boxplots for the same set of variables, allowing us to examine the spread and detect any outliers in the data:

```
sns.set_style("whitegrid")
plt.figure(figsize=(20, 10))

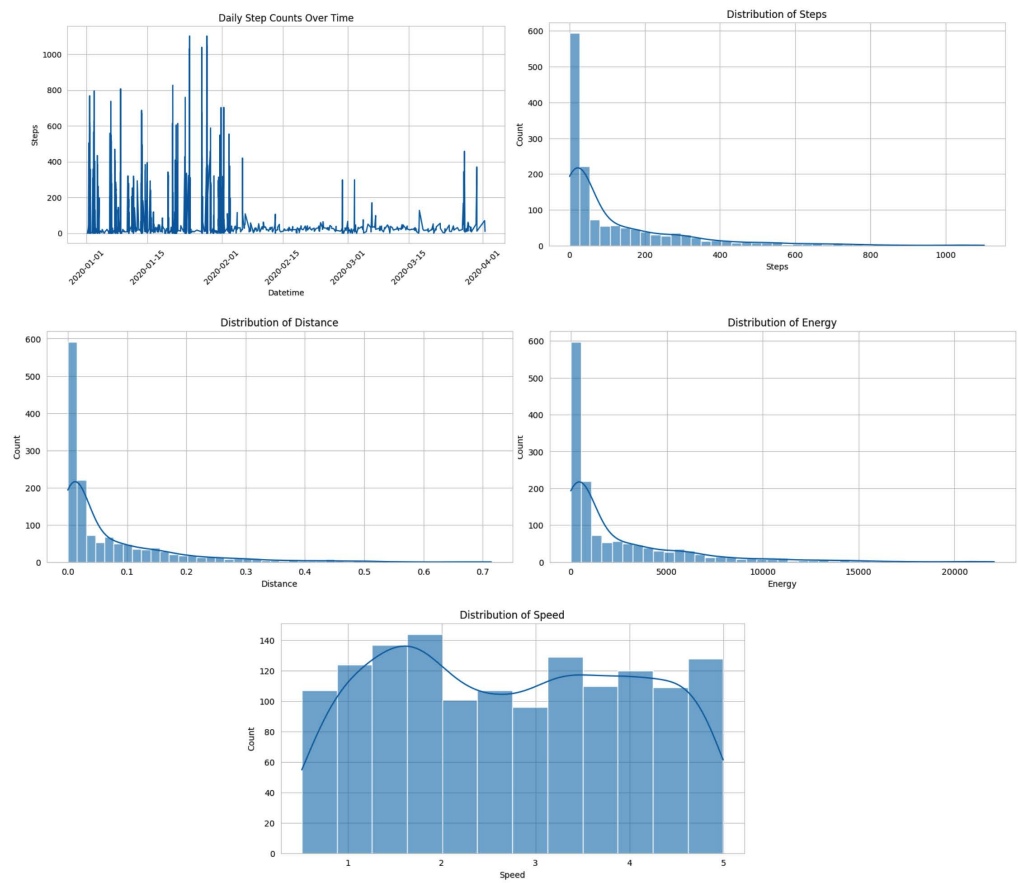
plt.subplot(3, 2, 1)
plt.plot(df['Datetime'], df['Steps'])
plt.title('Daily Step Counts Over Time')
plt.xlabel('Datetime')
plt.ylabel('Steps')
plt.xticks(rotation=45)

variables = ['Steps', 'Distance', 'Energy', 'Speed']
for i, var in enumerate(variables):
    plt.subplot(3, 2, i + 2)
    sns.histplot(df[var], kde=True)
    plt.title(f'Distribution of {var}')

plt.figure(figsize=(20, 10))

for i, var in enumerate(variables):
    plt.subplot(2, 2, i+1)
    sns.boxplot(y=df[var])
    plt.title(f'Boxplot of {var}')
```

The plots are shown next; we see daily step counts plotted over time, revealing both dramatic spikes of intense activity and calmer stretches of lower movement. Meanwhile, the histograms for distance, energy, and speed show that most days hover around moderate values, with a few outliers hinting at extra-energetic bursts. Looks like the wearer of this fitness device has a knack for mixing superhero-level workouts with well-deserved rest days.



The box plots shown next confirm the skewed nature of our data.

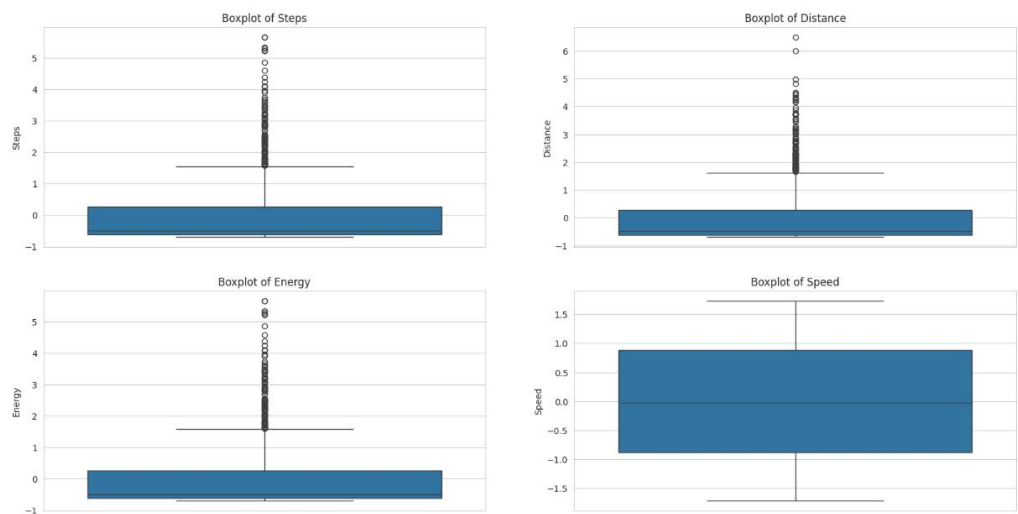


Figure 13.4: Daily activity plots

8. We then explore the relationships between our variables and daily activity patterns. This process helps us identify typical patterns and any outliers in our data, ensuring that the visualizations are neatly arranged before displaying them:

```
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
sns.regplot(x='Steps', y='Distance', data=df)
plt.title('Step Count vs. Distance')

plt.subplot(1, 2, 2)
sns.regplot(x='Steps', y='Energy', data=df)
plt.title('Step Count vs. Energy Burned')

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
sns.boxplot(x='Day of Week', y='Steps', data=df)
plt.title('Step Count by Day of Week')
plt.xticks(rotation=45)

plt.subplot(1, 2, 2)
sns.boxplot(x='Hour of Day', y='Steps', data=df)
plt.title('Step Count by Hour of Day')

plt.tight_layout()
plt.show()
```

The plots show that strong upward trend between step count and distance—every extra step pushes us that little bit farther, though there's just enough scatter to hint at individual stride quirks. On the right, the nearly perfect diagonal line linking step count to energy burned shows a direct conversion from footsteps to calories. It's as if every step is a currency that buys you a bit of distance and a measurable chunk of energy expenditure, painting a crystal-clear picture of how our daily movement adds up.

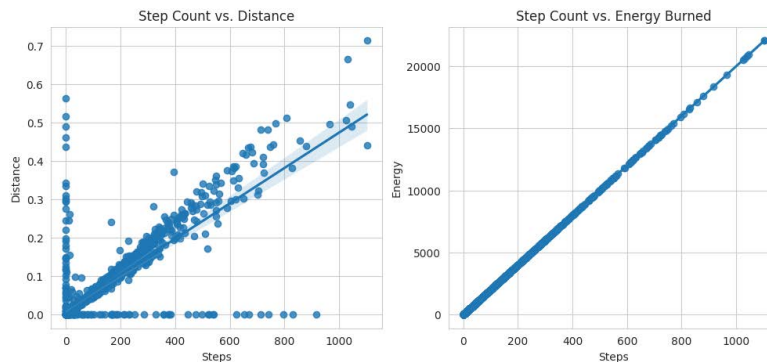


Figure 13.5: Plot between step count and distance and step count and energy burned

Next, we see how step counts vary across different days of the week, highlighting which days might encourage a more active lifestyle—though the outliers reveal some individuals surpassing the norm. On the right, step counts by the hour showcase a natural ebb and flow throughout the day, with activity levels climbing from morning into midday before gradually tapering off in the evening.

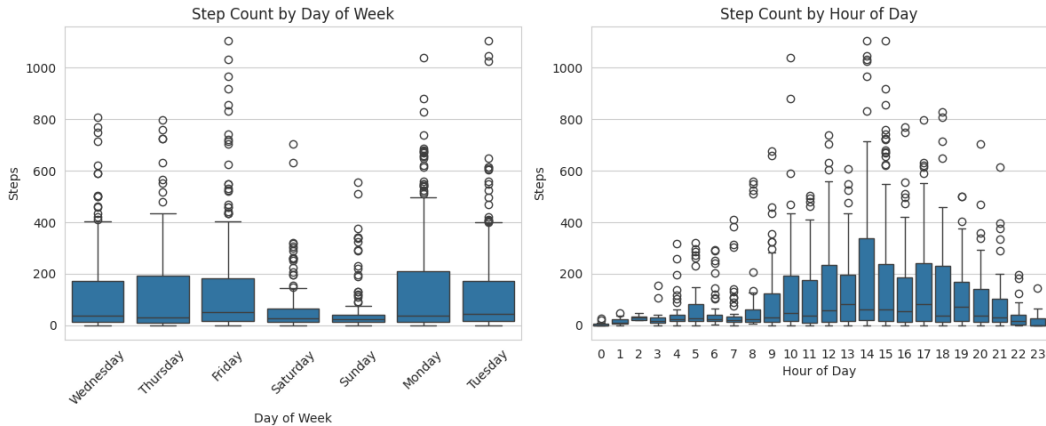


Figure 13.6: Step count by day of the week and by hour of the day

1. We begin by enriching our dataset with features that capture its temporal dynamics. First, we generate lagged versions of key metrics—'Steps', 'Distance', and 'Energy'—to reflect their values from the previous three time periods, which helps our model understand trends over time. We compute rolling statistics, calculating both the mean and standard deviation of 'Steps' over 3-day and 7-day windows to smooth out short-term fluctuations and highlight longer-term patterns. Finally, we encode cyclical aspects of time by transforming 'Hour of Day' and the day of the week into sine and cosine components, effectively capturing the periodic nature of these features:

```
for col in ['Steps', 'Distance', 'Energy']:
    for lag in range(1, 4):
        df[f'{col}_lag_{lag}'] = df[col].shift(
            lag
        ).ffill()

for window in [3, 7]:
    df[f'Step Count_rolling_mean_{window}'] = df[
        'Steps'
    ].rolling(
        window=window, min_periods=1
    ).mean().bfill()
    df[f'Step Count_rolling_std_{window}'] = df[
        'Steps'
    ].rolling(
        window=window, min_periods=1
    ).std().bfill()
```

```

    ].rolling(
        window=window, min_periods=1
    ).std().bfill()

df['Hour_sin'] = np.sin(
    2 * np.pi * df['Hour of Day'] / 24)
df['Hour_cos'] = np.cos(
    2 * np.pi * df['Hour of Day'] / 24)
df['DayOfWeek_sin'] = np.sin(
    2 * np.pi * df['Datetime'].dt.dayofweek / 7)
df['DayOfWeek_cos'] = np.cos(
    2 * np.pi * df['Datetime'].dt.dayofweek / 7)

```

2. We identify all numerical columns in our DataFrame and filter out any binary features, such as 'Active', that don't require scaling. Using the StandardScaler module, we transform the remaining features, ensuring they're on a comparable scale for our models. Looks like our data just got a much-needed fitness makeover!

```

numerical_cols = df.select_dtypes(
    include=['number']
).columns

cols_to_scale = [
    col for col in numerical_cols
    if col not in ['Active']]

scaler = StandardScaler()
df[cols_to_scale] = scaler.fit_transform(
    df[cols_to_scale]
)
display(df.head())

```

3. We sort the DataFrame chronologically to maintain the temporal sequence, and then perform our desired splits for training (70%), validation (15%), and testing (15%):

```

df = df.sort_values(by='Datetime')

train_ratio = 0.7
val_ratio = 0.15
test_ratio = 0.15

total_rows = len(df)
train_index = int(total_rows * train_ratio)
val_index = int(total_rows * (train_ratio + val_ratio))

```

```
df_train = df[:train_index]
df_val = df[train_index:val_index]
df_test = df[val_index:]

print(f"Shape of df_train: {df_train.shape}")
print(f"Shape of df_val: {df_val.shape}")
print(f"Shape of df_test: {df_test.shape}")
```

4. Now, we define the set of features we believe are crucial for capturing daily step behavior, including distance, energy burned, and various lagged or rolling metrics. Finally, we split out our training DataFrame into `X_train`—containing only the selected features—and `y_train`, which holds the 'Steps' column that our model aims to predict:

```
features = [
    'Distance', 'Energy Burned', 'Flights Climbed',
    'Walking Double Support Percentage',
    'Walking Speed', 'Avg Steps per Day', 'Hour of Day',
    'Step Count_lag_1', 'Step Count_lag_2',
    'Step Count_lag_3', 'Distance_lag_1', 'Distance_lag_2',
    'Distance_lag_3', 'Energy Burned_lag_1',
    'Energy Burned_lag_2', 'Energy Burned_lag_3',
    'Step Count_rolling_mean_3',
    'Step Count_rolling_std_3', 'Step Count_rolling_mean_7',
    'Step Count_rolling_std_7', 'Hour_sin', 'Hour_cos',
    'DayOfWeek_sin', 'DayOfWeek_cos'
]

missing_features = set(features) - set(
    df_train.columns)
if missing_features:
    print(f"Warning: The following features are not in df_train
    and will be excluded: {missing_features}")
    features = list(set(features) - missing_features)

X_train = df_train[features]
y_train = df_train['Steps']
```

5. We now instantiate a `RandomForestRegressor` class with a fixed random state to ensure reproducibility. Following this, we train our model using the training features (`X_train`) and the target variable (`y_train`), enabling it to learn the complex patterns in our wearable data that drive step counts:

```
model = RandomForestRegressor(random_state=42)
model.fit(X_train, y_train)
```

6. We then prepare our validation set; using our trained Random Forest model, we predict step counts for the validation data. To assess the accuracy of our predictions, we compute the **mean absolute error (MAE)** and the **root mean squared error (RMSE)**, which provide insight into the average prediction error and the variability of these errors:

```
X_val = df_val[features]
y_val = df_val['Steps']

y_pred = model.predict(X_val)

mae = mean_absolute_error(y_val, y_pred)
rmse = np.sqrt(mean_squared_error(y_val, y_pred))

print(f"Mean Absolute Error (MAE): {mae}")
print(f"Root Mean Squared Error (RMSE): {rmse}")
>>> Output
Mean Absolute Error (MAE): 0.1029288911096441
Root Mean Squared Error (RMSE): 0.35762909088476863
```

7. Lastly, let us visualize our predicted step count and ground truth:

```
plt.figure(figsize=(8, 6))
plt.scatter(y_val, y_pred)
plt.xlabel("Actual Step Count")
plt.ylabel("Predicted Step Count")
plt.title("Actual vs. Predicted Step Count (Validation Set)")
plt.plot(
    [min(y_val), max(y_val)],
    [min(y_val), max(y_val)],
    color='red', linestyle='--'
) # Add a diagonal line
plt.show()
```

Here is the plot; we can see, despite our data being quite a roller coaster, that the results are not that bad:

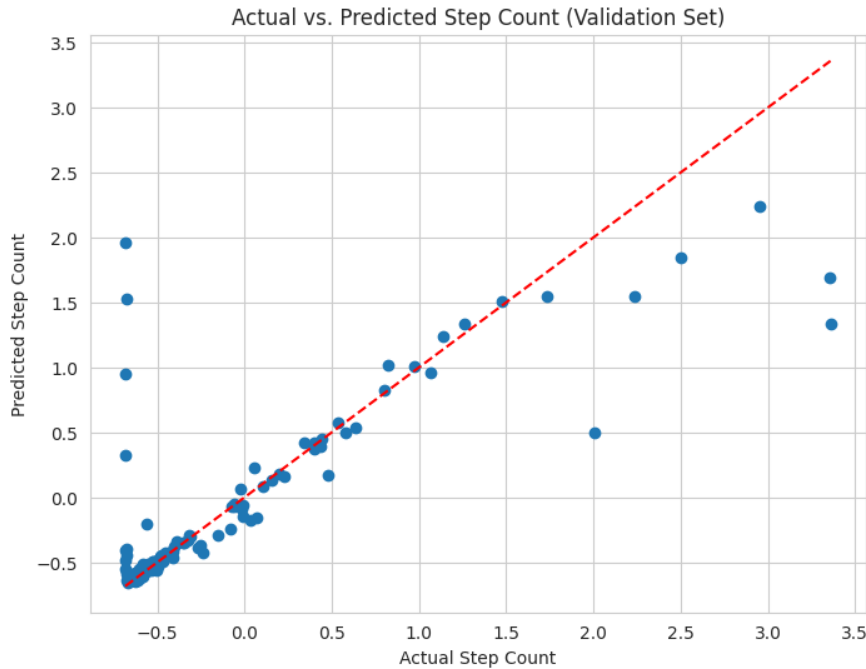


Figure 13.7: Predicted versus actual step count

For more physiological signal analysis (such as detailed heart rate variability or ECG waveform analysis), specialized Python libraries are available. One is the **Waveform Database (WFDB)** toolkit, which has a `wfdb` Python package for reading and processing biosignals. Researchers use it to load data from sources such as **PhysioNet** (a repository of medical sensor data) and perform signal processing in Python. Another useful library is **NeuroKit** (`neurokit2` in Python), which provides high-level functions to analyze biosignals – for example, detecting heartbeats in an ECG signal or peaks in a **photoplethysmography (PPG)** signal. These toolkits can take raw signals from a wearable and output features such as heart rate variability, breathing rate, or stress indicators, enabling advanced health analysis and even real-time feedback if integrated into an application.

In the next section, we will explore some real-world cases where AI, coupled with wearable devices, made a real difference in people's lives.

Smartwatch Health Monitoring in Action

To illustrate the power of wearables in real life, consider a smartwatch that potentially saved a life by early detection of a heart problem. In 2018, a 73-year-old man in New Jersey was alerted by his Apple Watch that his heart rate was irregularly low while at rest. He sought medical attention and discovered he had a serious underlying heart condition that could have led to a heart attack. This case exemplifies how the integration of IoT sensors and AI can yield life-saving insights. The watch's optical heart sensor collected data, and its onboard AI algorithms (tuned to detect atrial fibrillation or other anomalies) recognized an off-pattern and generated an alert. Without this wearable and its intelligent monitoring, the condition might have gone unnoticed until a much more dangerous event occurred (<https://www.cbsnews.com/philadelphia/news/apple-watch-saves-mans-life-frank-haggerty-lawrenceville-new-jersey/>).

Another case study comes from a clinical trial context: Stanford's Apple Heart Study (2019) enlisted over 400,000 Apple Watch users to validate the device's ability to detect **atrial fibrillation**. The study found that the watch could accurately identify irregular pulse rhythms in many cases and inform users to seek medical advice, demonstrating the real-world implementation of AI-driven health monitoring at scale. Several participants in the study received notifications of possible atrial fibrillation and, upon clinical evaluation, were found to indeed have notable heart arrhythmias. This large-scale case highlighted not only individual stories but also showed that wearables could be feasibly used as a widespread early warning system for cardiac health (<https://med.stanford.edu/news/all-news/2019/03/apple-heart-study-demonstrates-ability-of-wearable-technology.html>).

Wearables have also been used in specialized health interventions. For instance, in diabetic care, companies have introduced IoT **continuous glucose monitors (CGMs)** that sync with smartphones. In one scenario, a patient's glucose levels (captured every few minutes by a wearable sensor) were fed into an AI-driven insulin pump system. The AI learned the patient's daily patterns and could predict dangerous blood sugar drops, adjusting insulin delivery preemptively. A case study reported that such closed-loop systems (often called "artificial pancreas") significantly improved glucose control for participants, reducing incidents of hypoglycemia. This is a complex use of personal IoT: the wearable CGM provides real-time biometric data, an edge AI on a paired device makes rapid decisions, and the IoT insulin pump acts – all autonomously and continuously.

Wearable and fitness IoT devices, backed by AI and edge computing, are not just tech fads but meaningful health tools. The case studies show improved outcomes and even lives saved. As adoption grows, especially with medical-grade wearables (smart patches; implantables) joining the fray, personal IoT stands to become an integral part of healthcare delivery. The key is ensuring accuracy, user trust, and privacy as these devices become ever more involved in our well-being. Next, let us see how AI and IoT combined can bring smartness to home devices.

Smart Home Devices

Smart homes consist of IoT devices embedded in household fixtures and appliances, all connected to provide enhanced control, automation, and insight. A typical smart home might include the following:

- **Smart lighting:** Internet-connected light bulbs (for example, Philips Hue; LIFX) that can be controlled via smartphone or voice. They allow scheduling (lights turn on at sunset), remote control, and adaptive coloring (dimnable or color-changing based on mood or time of day).
- **Smart climate control (HVAC):** Smart thermostats such as **Google Nest** or **Ecobee** learn your heating and cooling preferences and schedule. They allow fine-grained control of home temperature, often featuring remote sensors in different rooms and integration with weather forecasts to optimize HVAC usage.
- **Smart security systems:** This covers smart door locks (keyless entry via phone or codes), smart doorbells with cameras (such as Ring; Nest Hello) that let you see and talk to visitors remotely, and Wi-Fi security cameras for both indoors and outdoors. Motion sensors, window/door sensors, and alarm sirens also integrate into home security IoT ecosystems, sending alerts to your phone on detection of intruders or unusual activity.
- **Smart appliances:** Many appliances now come with IoT features. Refrigerators can track inventory and suggest grocery orders; washing machines can be started or monitored from an app; robotic vacuum cleaners (such as Roomba) can be scheduled or commanded via voice; smart TVs and speakers tie into the home network to allow voice or app control of media. Even devices such as smart coffee makers or smart ovens exist, which can be programmed remotely or through routines (for example, brewing coffee automatically when you wake up).
- **IoT sensors and miscellaneous gadgets:** These include smart smoke and CO detectors that send phone alerts, smart water leak sensors near plumbing, smart blinds that open/close on schedule, and occupancy sensors to detect if people are in a room. They all connect to central hubs or directly to Wi-Fi to report data and respond to commands.

A core feature of smart home IoT is interconnectivity. Devices often communicate through a home hub or platform (using protocols such as Wi-Fi, Zigbee, Z-Wave, and Bluetooth). For example, a motion sensor (when it detects movement) might trigger smart lights to turn on and a security camera to start recording. To manage this, users typically have a smart home app or hub that ties everything together, such as **Apple HomeKit**, **Amazon Alexa**, **Google Home**, **Samsung SmartThings**, or open source platforms such as Home Assistant. These platforms unify control, so a single interface or voice command can coordinate multiple devices.

The convenience and capabilities of smart home IoT are rapidly expanding. According to industry reports, the number of connected devices in homes is growing exponentially, and with that comes a converged home experience where appliances and systems work in concert. For instance, you can set a “Good Night” routine that locks all doors, turns off lights, lowers thermostats, and arms the security system – all with one tap or voice command. The appeal is not just convenience but also

energy efficiency (turning off things that aren't needed), security (remote monitoring and instant alerts), and personalization (home reacts to individual preferences).

It's worth noting that setting up a smart home can range from simple (**Plug and Play (PnP)** Wi-Fi bulbs controlled by a phone) to complex (multiple devices from different manufacturers integrated via a central automation system). There are open source automation systems such as **openHAB** and **Home Assistant** that tech-savvy users deploy on devices such as **Raspberry Pi** to customize their smart homes deeply.

AI-driven home automation and intelligence

While remote control and scheduling are useful, the true power of smart home IoT emerges when combined with AI to create an autonomous, learning home. AI algorithms can learn from the homeowners' behaviors and preferences and then automatically adjust devices for comfort, efficiency, and safety. Some key applications of AI in home automation include the following:

- **Predictive heating/cooling (thermostat AI):** Smart thermostats already save energy by learning schedules, but newer models use AI to be even smarter. Google's Nest Learning Thermostat, for example, employs ML to quickly develop a personalized temperature schedule for the user. It observes when you adjust the temperature and uses sensors and your phone's location to infer when you're home or away. Over time, it automatically creates a heating/cooling schedule that keeps you comfortable when you're home and conserves energy when you're away. Independent studies cited by Nest found these thermostats can significantly reduce heating and cooling costs by fine-tuning climate control.
- **Adaptive lighting:** AI enhances smart lighting through features such as adaptive lighting (also called **human-centric lighting**). An AI-driven lighting system can learn user preferences for brightness and color temperature at different times. For instance, it might learn that in the evenings, you prefer warm, dim lighting in the living room but bright, cool light in the morning for the kitchen. Using this knowledge, the system automatically adjusts bulbs without manual input. If it knows your typical schedule, it can even turn lights on in certain rooms as you usually use them or dim them as bedtime nears. AI can also integrate lighting with other sensors: for example, if the system sees via a sensor or your smart TV that you've started watching a movie, it might gradually dim the lights.
- **Security threat detection:** Smart home security devices increasingly use AI for object and person recognition. Instead of just recording motion, an AI-enhanced security camera can distinguish between a person, a pet, or a falling tree branch. This reduces false alarms and can alert homeowners only to true intrusions. Companies such as Xailient (<https://www.biometricupdate.com/202311/smart-home-market-embracing-edge-biometrics-for-improved-security-and-privacy>) are integrating computer vision AI into smart cameras to do real-time facial recognition and object detection on the camera itself. AI can also detect abnormal sound patterns through smart speakers or dedicated sensors.

- **Personalized automation routines:** AI in home automation can learn routine tasks and start doing them proactively. Amazon Alexa's Hunches feature is a good example. Alexa learns your daily usage patterns of devices – say, you typically turn off the porch light and lock the door at 10 P.M. If one night you forget, Alexa might proactively suggest, “I have a hunch you left the porch light on; would you like me to turn it off?” Now, it can even automatically act on these hunches with your permission. This is AI working across devices: it notices deviations in multi-device state and takes initiative. Google Assistant and HomeKit similarly enable setting up scenes or using ML to suggest routines (such as Siri Suggestions for HomeKit actions).
- **Voice and gesture recognition:** AI-driven **natural language processing (NLP)** allows voice assistants to understand complex or contextual commands (“Set the living room lights to the usual evening ambiance”). AI also enables smart displays to recognize gestures or your presence – for example, a camera might notice you’ve walked into a room and greet you or display your schedule automatically. These interfaces improve automation by making interactions more intuitive and personalized.
- **Appliance efficiency and maintenance:** AI can optimize appliance usage. For instance, a smart fridge with AI might observe that you tend to open it frequently around 6 P.M. (dinnertime) and thus pre-cool itself slightly to maintain temperature when those openings occur (saving energy overall). Similarly, AI can do predictive maintenance – by monitoring an appliance’s performance (say, the energy draw of your HVAC), it can detect when the system is straining and might need a filter replacement or servicing. Nest has tested an AI that watches for irregularities in heating/cooling patterns to detect HVAC problems early. This heads-up can save money and prevent breakdowns by prompting timely maintenance.

In all these examples, the theme is a shift from **explicit user-controlled automation** to **AI-driven implicit automation** – the home begins to anticipate and respond to needs without being told explicitly every time. It’s a collaborative process: the AI learns from the users (possibly with some feedback mechanism) and gradually handles more tasks in the background.

It’s important to implement fail-safes: users should be able to override AI decisions, and the system should request confirmation for significant actions until trust is established. AI-driven automation works best when it’s personalized – what one household wants (lights off at 10, temp at 68°F) might differ from another. Hence, the learning aspect is crucial so that automation truly matches the occupants.

AI makes smart homes context-aware and predictive. Instead of just following pre-set rules, the home adapts to your lifestyle: it learns to anticipate your needs (comfort; convenience) and react to problems (security threats; device failures) in an intelligent way. As technology advances, we expect homes to exhibit even more “common sense.” The combination of IoT data and AI algorithms transforms a collection of gadgets into a coordinated, intelligent home system.

Smart home devices, much like wearables, can benefit from edge computing – performing data processing on local devices (or a home hub) instead of relying solely on cloud servers. Given that a home may contain dozens of IoT devices, the volume of data (and potential for sensitive information) is significant.

Case study: an integrated smart home implementation

Let us consider a real home case study of a residence outfitted (<https://pmc.ncbi.nlm.nih.gov/articles/PMC8198920/>) with comprehensive IoT automation. In this case study, we explore the practical implementation of the **qToggle** system within a real-life smart home setting, demonstrating the potential and flexibility of IoT technology. The home chosen for this scenario is a two-story house consisting of five rooms, two bathrooms, a kitchen, a pantry, a shed, a garage, and a garden. qToggle has been successfully implemented to manage several critical aspects, including temperature, lighting, energy consumption, security, access control, and garden irrigation.

Temperature control and A/C management

The qToggle system is actively controlling temperature within the house to ensure thermal comfort and reduce energy usage. The home utilizes nine smart thermostats—six Wi-Fi-enabled smart thermostats and three battery-powered Bluetooth thermostats—allowing temperature control on a per-room basis. Users benefit from remote accessibility via the **qToggle app**, which enables temperature adjustments from any location. Scheduling features further enhance energy efficiency, automatically adjusting temperatures during the day or when the house is unoccupied.

Additionally, the home's air conditioning system is integrated through smart plugs controlled via qToggle, enabling convenient on/off functionality remotely. While the current implementation supports basic control, an AI-driven extension could include predictive temperature adjustments. ML algorithms could analyze user behavior patterns and external weather data to autonomously optimize comfort and energy consumption.

Lighting control

Lighting throughout the house is managed via qToggle, ensuring convenience, energy conservation, and enhanced security. The system employs **SONOFF Touch switches**, supporting manual control through the qToggle app and voice commands via **Google Home** and **Amazon Alexa**. An implemented security feature simulates occupancy by scheduling lights to periodically activate when the residents are away.

AI could further augment this system by adapting lighting based on occupancy prediction, ambient natural light levels, and personalized user preferences, significantly improving efficiency and user experience.

Energy and power monitoring

The home is equipped with a sophisticated renewable energy setup, incorporating thirty-three photovoltaic panels connected to two inverters (**Fronius** and **MPPSolar**). The power generated can either be utilized immediately, stored in batteries, or injected back into the grid. Smart power meters based on **ESP8266** continuously monitor critical electrical parameters, providing real-time data and remote emergency cut-off capabilities.

To further capitalize on this implementation, AI algorithms can optimize energy usage by forecasting solar power generation, balancing battery storage and grid injection intelligently, and proactively identifying equipment anomalies or inefficiencies.

Access control and security

The qToggle system currently manages access control for gates and the garage using Wi-Fi-enabled smart relay boards, allowing manual control through the qToggle app or voice commands via Google Assistant. The security system, featuring a **Paradox MG5050** alarm panel integrated with a **Raspberry Pi** module, offers full remote arm/disarm capabilities and partial arming options for nighttime security.

AI integration could greatly enhance security by enabling facial recognition for entry, anomaly detection for unusual activity patterns, and predictive analytics to identify potential security breaches before they occur.

Irrigation system

The automated irrigation system, built around a Raspberry Pi, manages garden watering through pop-up sprinklers controlled by electric valves and pressure switches. Users can choose between manual and automatic irrigation modes, including scheduled watering adjusted by configurable morning and evening factors.

An intelligent AI-based approach could integrate weather forecasts, soil moisture sensors, and plant-specific irrigation needs to optimize water usage, reduce waste, and further decrease manual management requirements.

System architecture and communication

The overall qToggle system architecture features one master hub and six secondary hubs, managing various functionalities such as HVAC, lighting, power management, garage access, alarms, and irrigation. Communication between hubs relies primarily on wireless connections, with some critical systems connected via wired networks. An extensible API ensures interoperability and scalability, with robust logging mechanisms to ensure reliability and ease of debugging.

In the future, incorporating AI analytics into system logs and operational data could proactively detect system failures or performance degradation, improving overall system resilience and simplifying maintenance tasks.

Through this comprehensive implementation of qToggle, the case study demonstrates how IoT and AI technologies can collaboratively enhance smart home functionality, efficiency, and security.

Privacy Concerns in Personal IoT

While personal and home IoT devices offer considerable benefits, they also introduce **significant privacy and security concerns**. These devices often collect sensitive data and, if not secured properly, can become entry points for cyberattacks or lead to unintended data exposure. In this section, we discuss common security risks in personal IoT, methods to enhance privacy (such as encryption and local data processing), ethical considerations for AI in IoT, and real-world case studies of privacy breaches with lessons learned.

Attackers can violate privacy (peeping through cameras; tracking your daily routine), cause financial loss (crank up your thermostat to waste energy, or use your devices in botnets that consume your bandwidth), or even endanger safety (disabling alarms or unlocking doors without permission). A particularly sobering scenario is attacks on **medical IoT wearables**. If a hacker gained access to a wearable insulin pump or pacemaker, they could potentially harm the patient. In 2017, the FDA issued a recall of 465,000 St. Jude pacemakers to patch a cybersecurity hole that could let attackers drain the battery or alter pacing remotely (<https://beckerlaw.com/blog/fda-recalls-465000-pacemakers-cyber-security-threat/>). Fortunately, no known malicious exploits occurred, but it shows the stakes when IoT devices interact directly with health.

Another risk is the creation of **botnets** out of IoT devices. Gartner analysts estimated that at one point, 40% of smart home appliances globally were being used as parts of botnet attacks (often without the owners' knowledge). Appliances such as smart fridges or smart TVs might seem innocuous – who cares if a hacker can turn your fridge light on? – but attackers use them as “dumb robots” to launch cyberattacks elsewhere. Even if it doesn't hurt the device owner directly, it's a problem (and it can degrade performance or internet speed for the owner too).

To conclude, the **attack surface** for personal IoT is broad: each device is a computer that can be compromised. And unlike a phone or PC, IoT devices often lack user interfaces or security software, so the compromises can go unnoticed. They might quietly spy or become foot soldiers in larger attacks. This is why security is a top concern as we fill our homes and wearables with IoT gadgets.

Real-world incidents illustrate what can go wrong in personal IoT privacy – and what we can learn from them. Let's take a look at some examples next.

Case study 1: The smart camera breach (Ring cameras)

In late 2019, reports emerged of strangers talking to children through Ring cameras placed inside homes. How did this happen? Attackers didn't break Ring's encryption or servers; instead, they took advantage of reused passwords from unrelated breaches. Many Ring users had accounts with weak credentials that had been leaked elsewhere. The hackers tried those credentials on Ring and succeeded in several cases, granting them access to the camera feed and two-way audio. In one creepy instance, a hacker taunted an 8-year-old girl in her bedroom through a Ring camera. Mitigation: Ring responded by urging **two-factor authentication (2FA)** and implemented new security requirements. They also added user notifications when a new device logs in so that account owners are alerted to any unusual

access. The lesson is twofold: users must use unique, strong passwords and enable 2FA; IoT providers should enforce these best practices (now, many require 2FA by default). Also, sharing device access properly (Ring lets you add shared users instead of sharing the main login) avoids sharing credentials. This breach underscores that even if the video feed is encrypted, the system is only as secure as the user's login (<https://consumer.ftc.gov/consumer-alerts/2023/05/rings-privacy-failures-led-spying-and-harassment-through-home-security-cameras>).

Case study 2: The smart thermostat data leak

Not a single event, but researchers have shown that even innocuous IoT data can leak personal patterns. A study found that smart thermostat data uploaded to the cloud could be analyzed to infer occupancy patterns (DOI:10.1109/ICC.2015.7248381) – essentially, someone hacking or subpoenaing that data could tell when you are likely home or away (privacy issue; could aid burglars). Mitigation here involves companies adopting privacy-preserving data aggregation. For example, instead of storing raw temperature readings with timestamps per user, perhaps only store aggregated statistics across many homes or none at all in the cloud. Some smart thermostats have moved more logic on-device to minimize cloud reliance for such sensitive inferences.

Case study 3: Strava military base exposure

The fitness tracking app Strava compiled an anonymous heatmap of user activity worldwide. It seemed like a cool visualization of billions of running/cycling routes. However, analysts found that in war zones such as Afghanistan, the heatmap lit up in places where there shouldn't be civilian exercise – these corresponded to foreign military bases where soldiers were using fitness trackers (<https://www.bbc.com/news/technology-42853072>). Even though individual names weren't shown, the aggregate data gave away sensitive locations and routines. This was a privacy oversight: Strava did allow users to opt out of public sharing, but many military personnel didn't realize their default settings would share data publicly. The result was the US and other militaries had to update protocols, advising troops to disable GPS tracking or use strict privacy settings. This case emphasizes that companies should default to privacy for sensitive contexts and better inform users. Mitigation: Strava now puts more prominent privacy options and has more granular controls (and, possibly, excluding certain areas). Users, on their side, should be mindful of what “sharing” means – just because data is anonymized doesn't mean it cannot be deanonymized with external knowledge.

Case Study 4: IoT data breach (ORVIBO Smart Home)

In 2019, a Chinese IoT company ORVIBO left an Elasticsearch database publicly exposed without a password. It contained data for millions of smart home customers: email addresses, hashed passwords, precise geolocation, device names (such as “Bedroom Camera”), and even video feed records. About 2 billion logs were exposed. This is a stark example of cloud-side failure. Mitigation: After discovery, the database was secured. But the damage was done – such data could allow hackers to target specific homes or blackmail individuals (“I have your camera logs”). This shows the importance of cloud security

practices: companies must secure databases, use proper authentication, and not store more data than needed (why keep old logs?). For users, it's a reminder that using IoT services means some trust in the provider's security. Choosing reputable brands with a good security track record or offline-capable systems can reduce risk (<https://www.forbes.com/sites/daveywinder/2019/07/02/confirmed-2-billion-records-exposed-in-massive-smart-home-device-breach/>).

These case studies emphasize the need for mitigation strategies; some suggested strategies to increase security and privacy are the following:

- Use strong, unique credentials and 2FA for all IoT accounts. Many breaches start with credential stuffing or guessing.
- Keep devices updated. Patch known holes – many attacks target devices with outdated firmware.
- Segment IoT devices to a network where if they are compromised, damage is limited.
- Review privacy settings on apps/devices. Opt out of unnecessary sharing. Periodically check what data is being collected (many devices now have privacy menus or online dashboards).
- Encryption and secure protocols should be non-negotiable – avoid devices that use unencrypted communications for sensitive functions.
- **Plan for failure:** Have a way to notice and respond if a device is acting oddly (cover a camera if it's not supposed to be on, and so on). Some smart home systems can alert if, say, a new unknown device joins the network – which could catch an intruder.

On the industry side, governance and standards are improving. The **IoT Security Foundation (IoTSF)** and governments are creating guidelines (such as requiring unique default passwords, vulnerability disclosure policies, and so on). As these take effect, new devices should gradually become more secure out of the box.

In conclusion, while personal IoT has seen its share of breaches and scares, each incident pushes stakeholders (companies, consumers, regulators) to raise the bar for privacy and security. By learning from these cases and implementing robust safeguards, we can enjoy the benefits of IoT without unduly sacrificing privacy or safety.

Getting Started with Your Own Smart IoT Projects

For readers eager to bring the concepts from this chapter into the real world, here are a few accessible **do-it-yourself (DIY)** project ideas that use affordable components such as Raspberry Pi, Arduino, and wearable sensors. These projects not only reinforce learning but can also become stepping stones to more advanced personal or home IoT systems.

To begin, you only need a basic microcontroller, one to two sensors, and a few jumper wires. Python, Arduino IDE, and platforms such as Edge Impulse or Home Assistant can guide you through most setup and deployment workflows.

Component	Purpose
Raspberry Pi 4/Pico	Local computation, IoT gateway
Arduino Nano/UNO	Low-power sensor interface
DHT11/DHT22	Temperature and humidity sensing
MPU6050/ADXL345	Motion and fall detection
MAX3010/MAX30102	Heart rate and SpO2 monitoring
ESP32/ESP8266	Wi-Fi-enabled device control

Here is a list of DIY project ideas; try them out:

- **Smart room monitor (Raspberry Pi + DHT11 sensor + light sensor):** Build a basic smart home node that tracks room temperature, humidity, and light levels. Program it to trigger a fan or light based on sensor values, and visualize data using Python and `matplotlib`.
- **Fall detection wearable (Arduino Nano and accelerometer):** Create a simple wearable using an MPU6050 sensor to detect sudden falls. Use threshold-based logic or train a lightweight model to differentiate between normal movement and falls, then trigger alerts via Bluetooth.
- **Voice-controlled light automation (ESP32 + Relay + Google Assistant):** Connect a smart relay to control your home lamp via voice commands using IFTTT and ESP32. This project brings together cloud-based voice AI and local hardware automation.
- **Heart rate logger with edge AI (Raspberry Pi Pico + MAX30100):** Collect heart rate data and run a simple ML model (for example, anomaly detection) on-device using MicroPython or TinyML libraries such as Edge Impulse. Optionally, log or plot heart rate trends over time.
- **Personal activity classifier (Smartphone + Python):** Use a smartphone's accelerometer and gyroscope data to train an activity recognition model in Python that classifies walking, running, or sitting. Export and analyze the data using `pandas` and `scikit-learn`.

Remember, the only limit is your imagination. With a few components and the knowledge gained in this book, you can begin building your own intelligent wearables or smart home devices. Whether it's a health tracker or a voice-controlled light, creativity and curiosity are your best tools. Start small, dream big!

Summary

In this chapter, we explored the fascinating world of personal and home IoT applications, highlighting how these devices are rapidly transforming our daily lives. Wearable IoT devices, such as fitness trackers and smartwatches, are revolutionizing health monitoring by tracking vital signs, predicting potential health issues, and even detecting emergencies such as falls or cardiac anomalies. These devices leverage AI and ML to analyze data and offer personalized health insights, significantly enhancing preventive healthcare.

The integration of AI and edge computing further enhances these devices, allowing for real-time responsiveness and robust data privacy through local processing. Python-based examples provided practical insights into analyzing wearable data, illustrating the potential for developers and data scientists to extract meaningful health insights and predictions.

Smart homes, another major focus, leverage IoT technology for automation, convenience, security, and energy efficiency. Real-life implementations, such as the qToggle smart home system, demonstrated comprehensive control over lighting, temperature, security, and garden irrigation. We also explored how AI could augment these systems, enabling predictive analytics for optimized energy use, intelligent security features such as facial recognition, and adaptive irrigation systems responsive to weather conditions and soil moisture levels.

Privacy and security remain critical considerations. Through several real-world case studies, we learned essential strategies for safeguarding personal IoT ecosystems, including encryption, data anonymization, and robust user authentication mechanisms.

As we've seen, personal and home IoT technologies, empowered by AI and edge computing, promise significant benefits while demanding careful management of privacy and security concerns.

In the next chapter, we'll shift our focus to AI applications in **Industrial IoT (IIoT)**, exploring compelling examples where AI has notably enhanced efficiency, productivity, and safety in various industrial settings.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/ds>



AI for IIoT

Imagine a world where machines not only talk to each other but also learn from every vibration, temperature change, and minute fluctuation in their environment. This is the promise of the **Industrial Internet of Things (IIoT)** fused with AI. Companies from diverse backgrounds are today realizing the importance of AI and, hence, are incorporating it into their ecosystems. In this chapter, we will explore how AI is revolutionizing IIoT across various sectors. We'll dive deep into practical, real-world case studies that show how industries use AI to predict equipment failures, optimize energy consumption, and even guide precision farming. You'll see detailed code examples (upgraded to Keras 3) and learn about the architecture behind these systems. By the end of this chapter, you will have covered the following:

- Introduction to AI-Powered IIoT
- Predictive Maintenance using AI
- Electrical Load Forecasting in Industry
- IIoT in Agriculture – Precision Farming with Edge AI

Introduction to AI-Powered IIoT

At the crossroads of IoT, robotics, big data, and **machine learning (ML)**, industrial companies are witnessing a technological revolution that brings both immense opportunities and new challenges. The proliferation of low-cost sensors, expansive cloud platforms, and powerful edge infrastructure has accelerated the adoption of AI in industrial settings. As a result, AI-powered IIoT systems are rapidly transforming the way companies deliver products, offer services, and engage with customers and partners.

This evolution lays the foundation for what is known as IIoT. IIoT describes a vast, interconnected network of sensors, machines, and devices deployed across industrial environments—from factories and power grids to farms and beyond. These devices continuously generate massive volumes of real-time data. When combined with advanced AI algorithms, this data becomes a powerful asset, enabling organizations to optimize operations, reduce downtime, and make smarter, data-driven decisions.

In many modern factories, for example, predictive analytics is used to monitor the health of machinery in real time, drawing insights from sensor readings that reveal subtle shifts in vibration or temperature. Such early warnings can avert costly breakdowns and allow maintenance teams to intervene only when necessary, creating a more efficient approach than fixed service schedules. Some industrial sites also employ AI-driven computer vision to inspect products on the assembly line, detecting defects that human inspectors might overlook and ensuring higher quality output. At the same time, logistics operations benefit from AI-powered scheduling and route optimization, where algorithms respond to changing conditions—such as traffic or sudden demand surges—by adjusting deliveries and inventory flows in near real time.

Despite these breakthroughs, the integration of AI into industrial processes also introduces important considerations. Companies often face the task of aligning data formats and communication protocols across legacy systems, which can be a formidable undertaking. Cybersecurity measures must be robust and continually updated since connecting critical infrastructure to broader networks creates potential vulnerabilities. The sheer scale of data collected by sensors can pose additional challenges around storage, processing, and regulatory compliance, prompting organizations to invest in scalable architectures and strict governance practices. Yet these hurdles have not slowed the momentum of AI-powered IIoT, which continues to drive innovation across multiple sectors, creating new opportunities for competitive advantage and reshaping industrial landscapes in profound ways. As these AI-driven technologies keep advancing, they reveal a variety of applications that demonstrate the transformative impact of IIoT in diverse industries, leading us to some interesting use cases that highlight how these solutions are redefining operations and delivering value.

A large number of companies from diverse backgrounds are realizing the importance and impact of incorporating data analysis and AI into their ecosystems. From increasing their operations, supply chain, and maintenance efficiency to increasing employee productivity, to creating new business models, products, and services, there is no facet where AI has not been explored. Following, we list some interesting use cases of AI-powered IIoT in industries:

- **Predictive maintenance:** Perhaps the most celebrated IIoT use case, predictive maintenance involves monitoring the condition of machines through sensors (vibration, temperature, pressure, and so on) and using AI to predict faults or **remaining useful life (RUL)**. By identifying patterns that precede equipment failures, companies can service machines just in time – reducing unplanned downtime and maintenance costs (<https://a-team.global/blog/iiot-predictive-maintenance/>). For example, IIoT sensors on an oil pipeline or factory assembly robot stream hundreds of data points per second; AI models analyze these in real time to find anomalies and warn of impending failures (<https://www.xenonstack.com/blog/energy-grid-management>). This shift from reactive or schedule-based maintenance to condition-based maintenance improves reliability and safety.
- **Energy optimization and load forecasting:** Industrial facilities and utilities deploy AI to forecast electricity demand and optimize energy usage. Smart grids ingest data from smart meters and substation sensors (voltage, current, power flows) to predict short-term load fluctuations. AI-driven load forecasting helps balance generation with demand and manage

energy storage. In fact, load forecasting was one of the first commercial uses of **neural network (NN)** AI in industry (https://www.ieee.org/ns/periodicals/PES/Articles/PE_MayJun2022-Artificial.pdf). Modern **deep learning (DL)** models can ingest weather data, historical load, and even calendar events to forecast future consumption with high accuracy, enabling dynamic adjustments (for example, shedding non-critical loads during peaks to save costs). We'll see an example of how such forecasting can be implemented in the *Electrical load forecasting in industry* section.

- **Asset tracking:** Also called **asset management**, involves monitoring the location and condition of key physical assets. Historically, companies relied on **radio-frequency identification (RFID)** tags or barcodes to keep tabs on their equipment, which limited them to basic location tracking. Today, thanks to advanced AI algorithms and IIoT connectivity, asset tracking has evolved into a more dynamic process. Every piece of equipment can now be continuously monitored in real time, allowing organizations to optimize logistics, maintain proper inventory levels, detect inefficiencies, and even plan for replacements or upgrades more effectively.
- **Fleet management and maintenance:** The transport industry has been using AI for fleet management by optimizing routes for about a decade. The availability of many low-cost sensors and the advancement of edge computing devices have now made it possible for transport companies to collect and use the data received from these sensors to not only optimize the logistics by better vehicle-to-vehicle communication and preventive maintenance but to accelerate safety. Installing systems such as drowsiness detection, risky behavior caused due to fatigue or distraction can be detected, and the driver can be asked to take countermeasures.

Deploying AI in IIoT involves unique challenges: huge data volumes, the need for real-time response, intermittent connectivity, and security concerns. A key architectural consideration is *where* the AI computation happens – on the edge (near the data source; for example, on the factory floor or in the field) or in the cloud (central servers or data centers). Often, the answer is a hybrid approach. A common pattern is an **edge-cloud hierarchy**: edge nodes perform immediate, local analysis and actions and also act as gateways sending filtered data to the cloud (<https://openautomationsoftware.com/blog/iiot-edge-computing-vs-cloud-computing/>). The cloud collects data from many edges to train global models or do heavy analysis and then sends insights or updated models back to the edge. This way, you get the best of both: fast, reliable local control and powerful centralized intelligence.

Predictive Maintenance using AI

Heavy machinery and equipment are the backbone of any industry, and like all physical objects, they deteriorate, age, and fail. Initially, companies used to perform **reactive maintenance**; that is, maintenance was done once the equipment failure was reported. This used to cause unplanned downtime. For any industry, unscheduled, unplanned downtime can cause a significant resource crunch and drastically reduce efficiency, production, and, hence, profits. To deal with these problems, industries shifted to preventive maintenance.

In **preventive maintenance**, regular scheduled routine checks are performed at predetermined intervals. Preventive maintenance requires keeping a record of equipment and its scheduled maintenance. The third industrial revolution, where computers were introduced into industries, made it easy to maintain and update these records. While preventive maintenance saves the industry from most unplanned downtimes, it still isn't the best alternative since regular checks can be an unnecessary expenditure. The following diagram outlines an example of the four industrial revolutions:

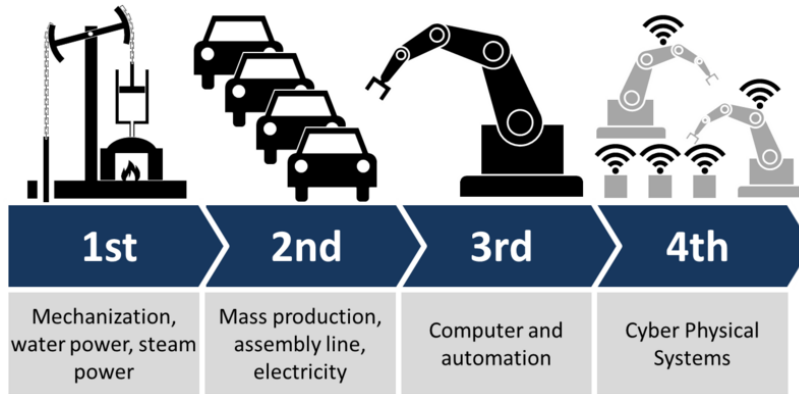


Figure 14.1: Industrial revolutions (Image source: https://commons.wikimedia.org/wiki/File:Industry_4.0.png)

The current trend of automation and digitalization has led to the fourth industrial revolution, also called **Industry 4.0**. This has allowed companies to deploy **machine-to-machine (M2M)** and **machine-to-human (M2H)** communication, along with AI-powered analytical algorithms, enabling predictive maintenance, which predicts the breakdown before it occurs using past data. Predictive maintenance strategies have enormously eased the maintenance and management of company resources.

The main idea behind predictive maintenance is to predict when equipment breakdown might occur based on condition-monitoring data. Sensors are used to monitor the condition and performance of equipment during its normal operation; depending on the equipment, different types of sensors may be used. Some common condition monitoring parameters/sensor values are as follows:

- Vibration sensors mainly used to detect misalignment, imbalance, mechanical looseness, or wear on pumps and motors
- Current/voltage sensors to measure the current and voltage supplied to an electric motor
- Ultrasound analysis to detect leakage in pipe systems or tanks or mechanical malfunctions of movable parts and faults in electrical equipment
- Infrared thermography to identify temperature fluctuations
- Liquid quality sensors—for example, wine sensors that detect the presence of different elements in the liquid

- Acoustic emission sensors that capture high-frequency stress waves to identify early crack formation or micro-fractures
- Fiber optic sensors for distributed sensing along pipelines or large structures, measuring parameters such as strain and temperature continuously
- Image-based sensors integrated with machine vision systems to inspect surfaces for defects or signs of degradation
- Gas sensors designed to detect chemical leaks or changes in ambient composition
- Pressure sensors that monitor pressure variations, which may signal system abnormalities
- Humidity sensors that measure moisture levels to help detect conditions that could lead to corrosion or equipment failure

To implement predictive maintenance, the most important thing is to identify the conditions that need to be monitored. The sensors required to monitor these conditions are then deployed. Finally, the data from the sensors is collected to build a model. Before delving into a hands-on example of how to do preventive maintenance, let us first go through some real-world examples.

Preventive maintenance real-world examples

Preventive maintenance is a proactive approach that involves regular and systematic inspections, servicing, and repairs to prevent equipment failures and extend asset lifespans. By implementing predictive maintenance strategies, companies across various industries have achieved significant cost savings and operational efficiencies. Here are real-world examples illustrating the impact of preventive maintenance:

- **Cement industry:** Titan America, a prominent cement manufacturer, sought to enhance its maintenance practices to reduce unplanned downtime and associated costs. By implementing a **reliability-centered maintenance (RCM)** program and integrating a **computerized maintenance management system (CMMS)**, the company achieved a 30% reduction in unplanned maintenance activities. This strategic shift not only improved equipment reliability but also led to substantial operational and maintenance savings. For an in-depth exploration, visit <https://www.getmaintainx.com/case-studies/how-titan-america-decreased-unplanned-maintenance-by-30-with-maintainx>.
- **Oil and gas sector:** A leading oil and gas company aimed to optimize its **operational expenditures (OpEx)** by refining its maintenance strategies. Through a comprehensive analysis and the adoption of preventive maintenance practices, the company reported cost reductions ranging from 10% to 30%, culminating in an overall OpEx reduction of 10%. This initiative underscored the financial benefits of transitioning from reactive to preventive maintenance approaches. For additional insights, refer to <https://www.solomoninsight.com/resources/oil-major-slashes-opex-maintenance-cost-continuous-improvement-case-study>.

- **Automotive manufacturing:** A US-based manufacturer of **original equipment manufacturer (OEM)** and aftermarket automotive parts faced challenges with equipment downtime and maintenance inefficiencies. By implementing a preventive maintenance program, the company achieved a 48% reduction in conveyor system incidents and a 24% decrease in downtime. Additionally, over 400 overdue work orders were closed, and an oil sampling program was initiated, contributing to enhanced equipment reliability and operational efficiency. Dive deeper into it by reading the full case study: <https://tbmcg.com/case-study/maintenance-intervention-cuts-downtime>.
- **Telecommunications industry:** A large telecommunications company conducted a study to evaluate the financial impact of preventive maintenance on its operations. The findings revealed that implementing predictive maintenance resulted in an average **return on investment (ROI)** of 545%. This significant ROI was attributed to extended equipment lifespans, reduced repair costs, and minimized downtime, highlighting the substantial economic advantages of preventive maintenance. Discover more details about this transformation: <https://www.micromain.com/preventive-maintenance-study/>.
- **Wind energy sector:** In the wind energy sector, maintenance costs constitute a significant portion of operational expenses. A study focusing on wind farm operators demonstrated that integrating preventive maintenance with corrective strategies led to a 40% reduction in maintenance costs and a 1.8% increase in production availability. These improvements were achieved by optimizing maintenance schedules and adopting condition-based monitoring techniques. Gain further insights here: <https://arxiv.org/pdf/2105.06666>.

These examples underscore the transformative impact of preventive maintenance across diverse industries. By proactively addressing equipment health, companies can achieve substantial cost savings, enhance operational efficiency, and extend the lifespan of critical assets.

Predictive maintenance: hands-on examples

Picture a large manufacturing plant with dozens of machines (pumps, motors, **computer numerical control (CNC)** machines). Unexpected breakdowns in these machines can halt production and cost thousands of dollars per minute in downtime. To avoid this, the plant deploys IoT sensors on critical components (for example, vibration sensors on motors and temperature sensors on bearings) and uses an AI-driven predictive maintenance system. The goal is to continuously monitor equipment health and **predict failures before they happen** so that maintenance can be scheduled at optimal times. This approach has been adopted in many industries – from factories to railways and jet engines – to improve reliability. For example, in energy grids, utilities gather real-time data from transformers and use ML to predict outages, enabling preventative repairs. AI models can identify subtle precursors to failure that human operators might miss, such as a combination of slight vibration increase and temperature fluctuation that often precedes a motor failure.

Predictive maintenance data often comes as time series sensor readings. Each machine provides a stream of measurements (such as vibration amplitude, temperature, pressure, RPM (rotations per minute), and so on) over time. In addition, we may have metadata such as machine age, usage patterns, or maintenance history. There are two common modeling approaches:

- **Failure classification:** Use sensor data (current or recent history) to classify whether a machine is in “normal” condition or will fail soon (within a certain horizon). This is treated as a binary classification (or multi-class if predicting specific failure modes). It’s useful when you want an alert that a failure is likely imminent.
- **RUL regression:** Estimate how much time is left before the machine fails (a regression problem). This is common in scenarios where machines run until failure in simulations or testing – for example, NASA’s C-MAPSS turbofan engine dataset provides run-to-failure sensor data from a simulated jet engine, and the task is to predict each engine’s remaining operational cycles. RUL prediction helps plan maintenance lead time (for example, “This machine has ~5 days of life left”).

Let us start with some hands-on examples of preventive maintenance.

Traditional ML for failure prediction

For our hands-on example, we will demonstrate a simplified example using a public dataset. The AI4I 2020 Predictive Maintenance Dataset (a synthetic dataset from a research study) contains 10,000 samples of simulated industrial machine data (UCI ML link: <https://archive.ics.uci.edu/dataset/601/ai4i+2020+predictive+maintenance+dataset>). Each sample represents a snapshot of a machine at a point in time with features such as the following:

- Air temperature (K)
- Process temperature (K)
- Rotational speed (rpm)
- Torque (Nm)
- Tool wear (min)

It also includes a target label indicating if the machine experienced a failure at that point (and the type of failure). While it is not time-series data for each machine over many cycles, it still serves to illustrate a classification approach to predictive maintenance by predicting failure occurrences from sensor readings.

We will use a traditional ML model to predict machine failure based on sensor data. In practice, one might engineer features from time series (for example, recent trends or moving averages of readings), but to keep it simple, we’ll use the snapshot features as provided. A common choice is a tree-based ensemble (such as **Random Forest** or **XGBoost**), which handles feature interactions well and provides importance to each sensor. Here is how to implement this in Python: get the data, clean it, preprocess

it if necessary, split it into train-validation-test sets, define the model, train it, optimize the model if needed, and then use it for prediction.

1. We start with importing the Python modules that we will be using:

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import (
    classification_report, accuracy_score)
import requests
from io import BytesIO
from zipfile import ZipFile
```

2. We download the data and unzip it; to do this, we define a useful function, `download_and_unzip`. The function takes as input the URL, downloads from the given link, and unzips it. As a result, we will have an `'ai4i2020.csv'` data file:

```
def download_and_unzip(url, extract_to='.'):
    response = requests.get(url)
    # Raise an exception for bad status codes
    response.raise_for_status()

    with ZipFile(BytesIO(response.content)) as zip_file:
        zip_file.extractall(extract_to)

url = 'https://archive.ics.uci.edu/static/public/601/
ai4i+2020+predictive+maintenance+dataset.zip'
download_and_unzip(url)
```

3. Next, the data is loaded in a `DataFrame`:

```
data = pd.read_csv("ai4i2020.csv")
```

4. We extract the features in `X` and the target in `y`:

```
X = data[[
    "Air temperature [K]",
    "Process temperature [K]",
    "Rotational speed [rpm]",
    "Torque [Nm]", "Tool wear [min]"
]]
y = data["Machine failure"]
```

5. Split the data into training and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    random_state=27
)
```

6. Define a simple Random Forest classifier. Train the model using the `fit` method:

```
model = RandomForestClassifier(
    n_estimators=100, random_state=0
)
model.fit(X_train, y_train)
```

7. And finally, evaluate the trained model:

```
y_pred = model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

Here is a screenshot of the result; we get a 98.6% accuracy. The classification report indicates strong performance for class 0, with near-perfect precision, recall, and F1-score. However, the model's performance is weaker for class 1, particularly in recall and F1-score, suggesting it struggles to correctly identify instances of this class. The overall accuracy is high, but this might be skewed by the class imbalance evident from the support values, where class 0 has significantly more instances than class 1. Therefore, while generally accurate, the model is less effective at classifying class 1.

Accuracy: 0.9865					
	precision	recall	f1-score	support	
0	0.99	1.00	0.99	1942	
1	0.84	0.66	0.74	58	
accuracy			0.99	2000	
macro avg	0.92	0.83	0.87	2000	
weighted avg	0.99	0.99	0.99	2000	

Figure 14.2: Accuracy and classification report of the trained Random Forest on the test dataset

8. We also analyze the **receiver operating characteristic (ROC)** curve. The **area under the curve (AUC)** is 0.98, which is very close to the maximum possible value of 1. An AUC of 1 represents a perfect classifier. An AUC of 0.98 signifies that our model has a very high ability to distinguish between positive and negative classes.

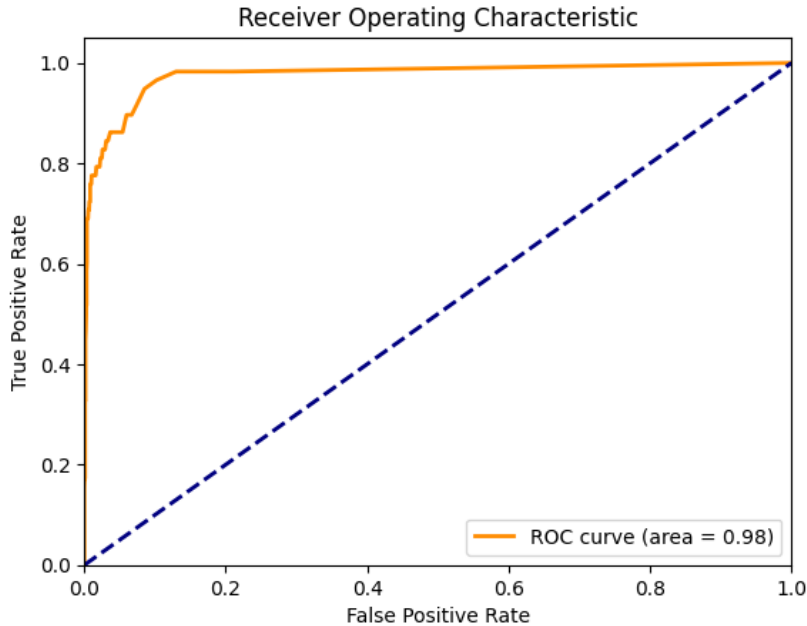


Figure 14.3: ROC curve for the positive class

The orange ROC curve is plotted far away from the diagonal dashed line (navy blue). The dashed line represents a random classifier, meaning a model that performs no better than chance. Our model's curve being significantly above this line shows it's much better than random guessing.

Predicting RUL using LSTM

For this hands-on exercise, we will use the NASA Turbofan Engine dataset (<https://data.nasa.gov/dataset/c-mapss-aircraft-engine-simulator-data?>).

The NASA Turbofan Engine dataset is a rich collection of multivariate time-series data designed for prognostics and health management. Each time series represents an individual engine unit, capturing sensor measurements and operational settings recorded at each cycle of operation. As engines operate over time, sensor readings gradually change, reflecting both normal operational conditions and the progression toward failure. In the training set, every engine is run until failure, allowing us to compute the RUL at each cycle by simply taking the difference between the final cycle and the current cycle. The test set, however, is truncated before failure occurs, and the true RUL values are provided in a separate file.

Because the dataset is inherently sequential—each row is a snapshot of the engine's state at a given cycle—it naturally lends itself to time-series modeling. This makes it an excellent candidate for advanced DL techniques such as **long short-term memory (LSTM)** networks or transformer-based models, which are designed to capture temporal dependencies in data. Moreover, as discussed in *Chapter 10*,

one can also leverage Hugging Face time-series models to fine-tune and predict RUL effectively. These approaches enable us to explore different architectures to model the degradation process of engines, ultimately aiming to predict the remaining cycles before failure. Let us dive in:

1. We begin by importing the essential libraries needed for data processing, visualization, and DL. These include tools for handling numerical data, splitting datasets, scaling features, and building an LSTM-based predictive model. Additionally, we suppress warnings to keep the output clean. This setup ensures we have everything ready to process NASA's turbofan engine data for predictive maintenance:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense, LSTM, Dropout
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
```

2. We specify the file paths for the dataset. The 'train_FD001.txt' training file contains historical sensor readings from engines until failure. The 'test_FD001.txt' test file includes sensor readings for engines but without failure labels. The 'RUL_FD001.txt' RUL file provides the actual RUL of engines in the test set:

```
train_file = 'train_FD001.txt'
test_file = 'test_FD001.txt'
rul_file = 'RUL_FD001.txt'
```

3. We load the training dataset into a pandas DataFrame. Since the file is space-separated and has extra blank columns, we remove unnecessary columns. Next, we assign meaningful column names. The first two columns represent the engine ID and operational cycle. Three columns store operational settings, and the remaining columns correspond to sensor readings:

```
train_df = pd.read_csv(train_file, sep=' ', header=None)
train_df.drop(
    train_df.columns[[26, 27]],
    axis=1, inplace=True
)
train_df.columns = [
    'id', 'cycle', 'op1', 'op2', 'op3'
] + [f'sensor{i}' for i in range(1, 22)]
```

4. We determine the RUL for each engine in the training dataset. First, we find the maximum cycle count for each engine, indicating the last recorded operational cycle before failure. We then merge this information with the training data and compute the RUL by subtracting the current cycle from the maximum cycle. This step provides the target variable needed to train our predictive maintenance model:

```
max_cycles_train = train_df.groupby(
    'id')['cycle'].max().reset_index()
max_cycles_train.columns = ['id', 'max_cycle']

train_df = train_df.merge(
    max_cycles_train, on='id', how='left'
)
train_df['RUL'] = train_df['max_cycle'] - train_df['cycle']
train_df.drop('max_cycle', axis=1, inplace=True)
```

5. We analyze the relationships between different features in the dataset by computing a correlation matrix. This helps us identify strongly related variables, potential redundancies, and key features that influence engine degradation. Understanding these relationships is crucial for building an effective predictive maintenance model:

```
sensor_cols = [f'sensor{i}' for i in range(1, 22)]
op_cols = ['op1', 'op2', 'op3']
features_to_analyze = [
    'cycle'] + op_cols + sensor_cols + ['RUL']
corr_matrix = train_df[features_to_analyze].corr()
plt.figure(figsize=(18, 16))
sns.heatmap(
    corr_matrix, cmap='coolwarm',
    annot=False, square=True, linewidths=.5
)
plt.title(
    'Correlation Heatmap of All Features', fontsize=15
)
plt.tight_layout()
plt.show()
```

Here is the correlation plot, where we can see that not all sensor values contribute to our target RUL.

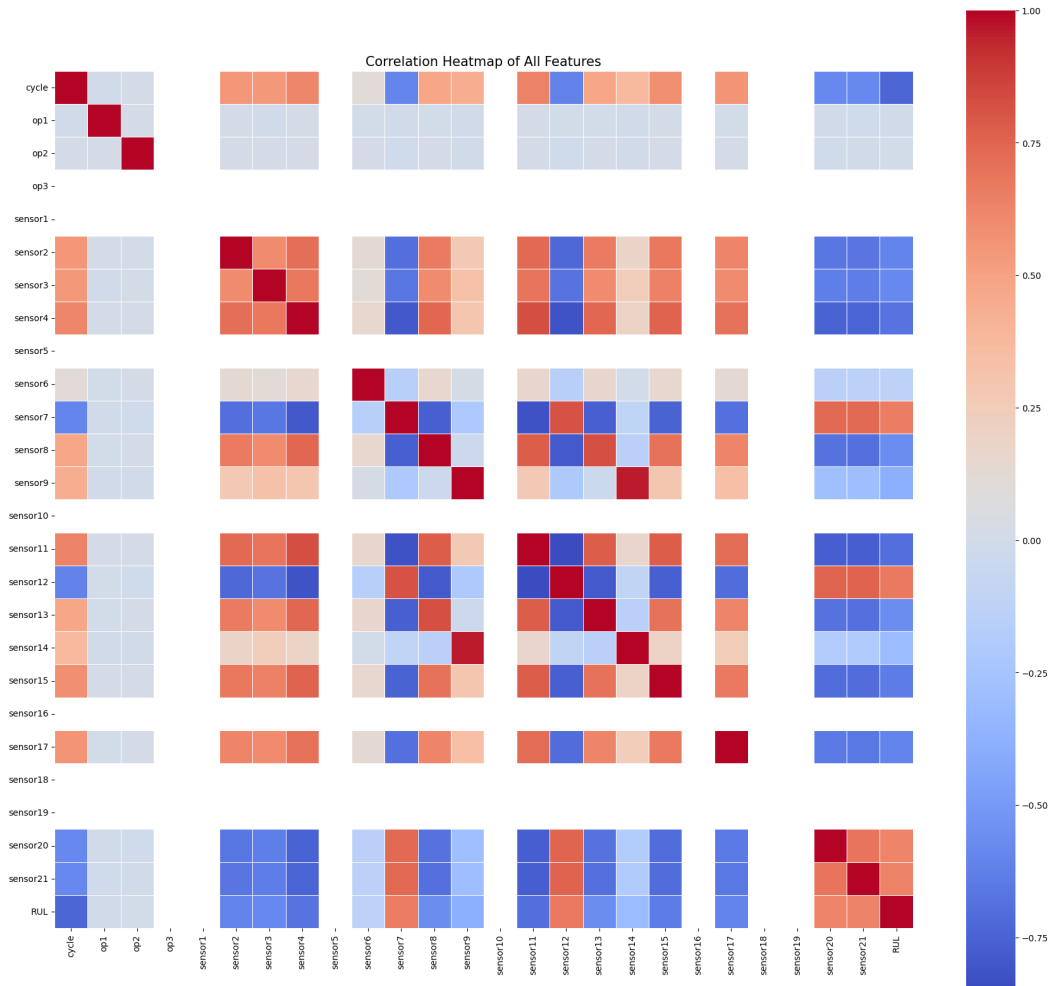


Figure 14.4: Correlation heatmap of all features

- We analyze how each feature correlates with the RUL. By sorting the correlation values, we can see which sensor readings and operational settings have the strongest relationships with engine degradation:

```
rul_correlation = corr_matrix[
    'RUL'].sort_values(ascending=False)
plt.figure(figsize=(12, 10))
rul_correlation.drop('RUL').plot(kind='bar')
```

```
plt.title('Feature Correlation with RUL', fontsize=15)
plt.axhline(y=0, color='r', linestyle='-', alpha=0.3)
plt.grid(axis='y', linestyle='--', alpha=0.5)
plt.tight_layout()
plt.show()
```

Here is the plot:

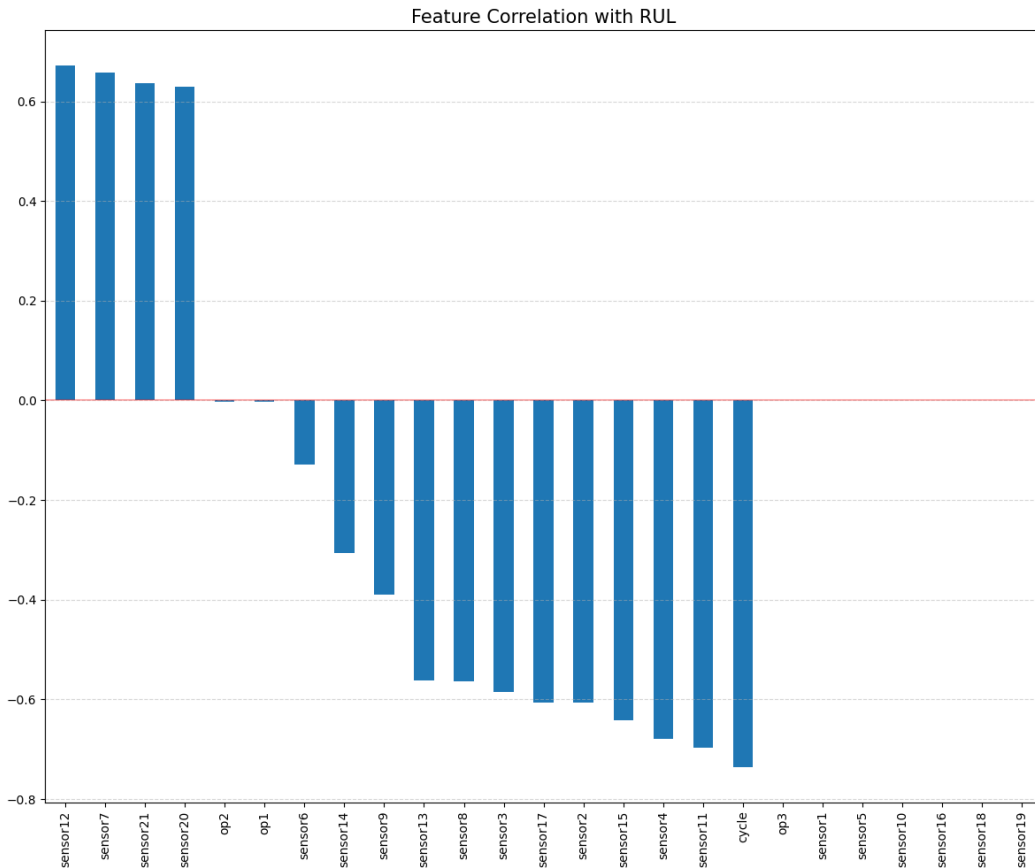


Figure 14.5: Feature correlation with RUL

7. We identify the sensors most strongly correlated with RUL. By computing the absolute correlation of each sensor with RUL, we rank them in descending order of importance:

```
high_corr_sensors = rul_correlation[
    sensor_cols
].abs().sort_values(ascending=False)
print("\nSensors ranked by absolute correlation with RUL:")
```

```

for sensor, corr in high_corr_sensors.items():
    print(f"{sensor}: {corr:.4f}")

# Select top correlated sensors
top_sensors = high_corr_sensors.index[:14].tolist()
print(f"\nSelected top {len(top_sensors)} correlated sensors:
{top_sensors}")

```

8. The top 14 most relevant sensors are selected for model training, along with `cycle` and the three `op` parameters. This feature selection step helps us reduce noise and improve model performance by focusing on the most informative sensor readings:

```

selected_sensors = top_sensors
features = [
    'cycle', 'op1', 'op2', 'op3'
] + selected_sensors
print(f"Selected features: {features}")

```

9. We create a function to prepare the data for the LSTM model by transforming it into sequences. Since LSTMs work best with sequential inputs, we create fixed-length sliding windows of sensor readings and operational data for each engine. For each engine, we extract a sequence of a defined length (for example, 50 cycles) and store the corresponding RUL as the target value. These sequences serve as inputs to the model, enabling it to learn patterns in sensor behavior leading up to failure:

```

def create_sequences(df, engine_ids, seq_length=50):
    sequences = []
    targets = []

    for engine_id in engine_ids:
        engine_data = df[df['id'] == engine_id][
            features].values
        engine_rul = df[df['id'] == engine_id][
            'RUL'].values

        for i in range(len(engine_data) - seq_length + 1):
            sequences.append(
                engine_data[i:i+seq_length])
            targets.append(engine_rul[i+seq_length-1])

    return np.array(sequences), np.array(targets)

```


10. We apply the function on the training data:

```
train_engine_ids = train_df['id'].unique()
print(f"Number of training engines: {len(train_engine_ids)}")
train_sequences, train_targets = create_sequences(
    train_df, train_engine_ids)
print(f"Training sequences shape: {train_sequences.shape},
targets shape: {train_targets.shape}")

>>> Output
Creating training sequences
Number of training engines: 100
Training sequences shape: (15731, 50, 18), targets shape:
(15731,)
```

11. We do the same for the test dataset:

```
test_engine_ids = test_df['id'].unique()
print(f"Number of test engines: {len(test_engine_ids)}")
test_sequences, test_targets = create_sequences(
    test_df, test_engine_ids)
print(f"Test sequences shape: {test_sequences.shape}, targets
shape: {test_targets.shape}")

>>> output
Creating test sequences
Number of test engines: 100
Test sequences shape: (8255, 50, 18), targets shape: (8255,)
```

12. We standardize the sensor readings and operational settings to ensure consistent data distribution. Standardization helps the LSTM model converge faster and improves its performance:

```
scaler = StandardScaler()
train_sequences_reshaped = train_sequences.reshape(
    -1, train_sequences.shape[2])
test_sequences_reshaped = test_sequences.reshape(
    -1, test_sequences.shape[2])

train_sequences_scaled = scaler.fit_transform(
    train_sequences_reshaped)
test_sequences_scaled = scaler.transform(
    test_sequences_reshaped)

train_sequences = train_sequences_scaled.reshape(
    train_sequences.shape)
test_sequences = test_sequences_scaled.reshape(
    test_sequences.shape)
```

13. We divide the training sequences into training and validation sets:

```
X_train, X_val, y_train, y_val = train_test_split(
    train_sequences, train_targets,
    test_size=0.2, random_state=42
)
```

14. We define a function to build an LSTM-based DL model to predict the RUL of engines. The model consists of two stacked LSTM layers that capture temporal dependencies in sensor data. Dropout layers help prevent overfitting by randomly disabling some neurons during training. Fully connected (Dense) layers refine the extracted features, with the final output layer predicting the RUL as a continuous value. The model is compiled using the **Adam optimizer** and **mean squared error (MSE)** loss, making it well suited for regression tasks:

```
def build_lstm_model(input_shape):
    model = Sequential()

    # LSTM layers
    model.add(LSTM(
        units=64, return_sequences=True,
        input_shape=input_shape
    ))
    model.add(Dropout(0.2))

    model.add(LSTM(units=32, return_sequences=False))
    model.add(Dropout(0.2))

    # Dense layers
    model.add(Dense(units=16, activation='relu'))
    model.add(Dense(
        units=1, activation='linear'
    )) # Output layer for RUL prediction

    model.compile(
        optimizer='adam',
        loss='mean_squared_error',
        metrics=['mae']
    )
    return model
```

15. We use the function to build the LSTM model:

```
input_shape = (
    train_sequences.shape[1],
    train_sequences.shape[2]
)
model = build_lstm_model(input_shape)
model.summary()
```

You can see the model summary in *Figure 14.6*:

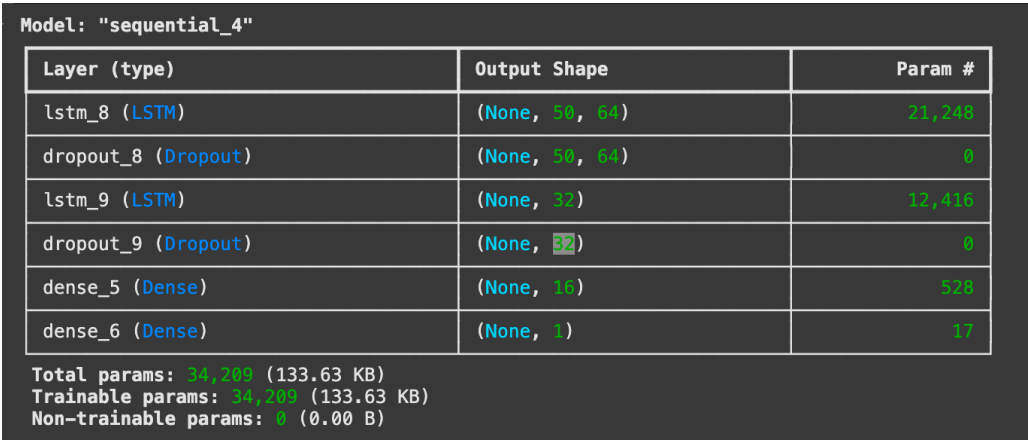


Figure 14.6: Model summary

16. Let us now fit the model on the training dataset:

```
history = model.fit(
    X_train, y_train,
    epochs=50,
    batch_size=32,
    validation_data=(X_val, y_val),
    verbose=1
)
```

17. Here is the code to plot loss and MAE for both the training and validation set. We can see that loss (MSE) and **mean absolute error (MAE)** both decrease as the model learns:

```
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
```

```
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper right')

plt.subplot(1, 2, 2)
plt.plot(history.history['mae'])
plt.plot(history.history['val_mae'])
plt.title('Model MAE')
plt.ylabel('MAE')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper right')

plt.tight_layout()
plt.show()
```

Here's what we see:

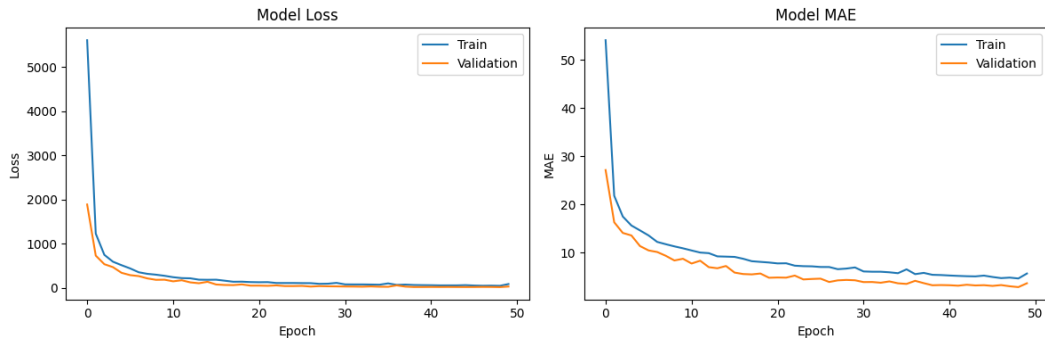


Figure 14.7: Loss and MAE plots while training

18. Next, we evaluate the model on the test data set. The MSE is 1927, and the MAE is 31.70. Since lower loss values indicate better performance, these metrics suggest a reasonable prediction accuracy, though the ideal value depends on the dataset's characteristics:

```
test_loss, test_mae = model.evaluate(
    test_sequences,
    test_targets, verbose=0
)
print(f"Test Loss: {test_loss:.4f}")
print(f"Test MAE: {test_mae:.4f}")
```

19. To gain deeper insight, we visualize the predicted versus actual RUL for both the training and test datasets. This helps us understand how well the model generalizes and highlights areas for potential improvement:

```
print("Making predictions")
train_predictions = model.predict(
    train_sequences).flatten()
test_predictions = model.predict(
    test_sequences).flatten()

# Plot results
print("Plotting results")
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.scatter(train_targets, train_predictions, alpha=0.5)
plt.plot(
    [0, max(train_targets)],
    [0, max(train_targets)],
    'r--'
)
plt.xlabel('Actual RUL')
plt.ylabel('Predicted RUL')
plt.title('Training Data: Actual vs Predicted RUL')

plt.subplot(1, 2, 2)
plt.scatter(test_targets, test_predictions, alpha=0.5)
plt.plot(
    [0, max(test_targets)],
    [0, max(test_targets)],
    'r--'
)
plt.xlabel('Actual RUL')
plt.ylabel('Predicted RUL')
plt.title('Test Data: Actual vs Predicted RUL')

plt.tight_layout()
plt.show()
```

Let's see the output:

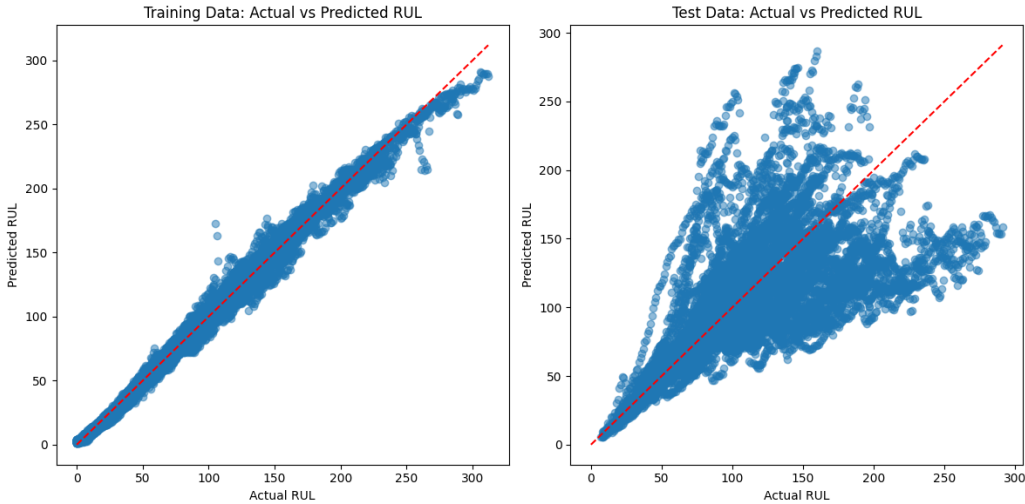


Figure 14.8: Predicted versus actual RUL values

From *Figure 14.8*, we can see that the predictions closely align with the actual values for the training dataset, indicating that the model has learned the patterns well. However, for the test dataset, while there is a general trend following the expected RUL values, some variance and overestimation errors appear, especially at higher RUL values.

This suggests that the model generalizes well but could still be optimized for better test performance. We can improve this by fine-tuning hyperparameters, adding more regularization, or experimenting with different architectures to reduce prediction errors.

Predictive maintenance for decision-making and actuation

Predicting failures is only part of the story – the end goal is to take action. In an industrial setup, AI systems can automatically trigger workflows when a prediction indicates risk. For example, if our model predicts a high chance of failure for a compressor pump within the next hour, the system could do the following:

- Raise an alert to technicians with recommended maintenance actions
- Perhaps automatically dial down the pump's load or switch to a backup unit if available.

This decision logic can be rule-based (if failure probability > X, then perform action Y) or learned via **reinforcement learning (RL)**, where an agent learns the optimal maintenance policy. A simple rule-based decision example in code might be the following:

```
failure_prob = 0.85 # output from a predictive model
if failure_prob > 0.8:
```

```
print("ALERT: Machine likely to fail soon. "  
      "Initiate maintenance protocol.")  
# Here we could programmatically schedule maintenance  
# or shut down equipment
```

In more complex scenarios, an RL agent could learn a maintenance scheduling policy by balancing the cost of downtime versus maintenance. For instance, the agent gets a negative reward if a failure occurs (due to no maintenance) or if maintenance is done too early (wasting component life). Over time, it could learn an optimal strategy for when to service machines. This kind of **AI-driven decision-making** turns predictions into tangible operational improvements.

Predictive maintenance advantages and disadvantages

Downtime disrupts industries across the board, hindering efficiency and impacting profitability. Predictive maintenance emerges as a crucial strategy to combat this, offering a proactive approach to asset management. By anticipating equipment failures, it promises significant advantages:

- **Minimized downtime and enhanced efficiency:** Predictive maintenance dramatically reduces unscheduled interruptions, ensuring smoother production flows and maximizing equipment effectiveness
- **Optimized maintenance:** Maintenance is precisely scheduled based on real-time equipment health, eliminating unnecessary interventions and freeing up resources
- **Cost savings:** Preventing major breakdowns, optimizing spare parts usage, and extending equipment lifespan lead to substantial cost reductions and improved ROI
- **Improved safety:** Early detection of potential failures mitigates risks, creating a safer working environment

However, predictive maintenance also presents challenges:

- **Initial investment:** Implementing these systems requires upfront costs for sensors, software, and skilled personnel
- **Complexity and data management:** Managing vast amounts of data and extracting actionable insights demands expertise and robust infrastructure
- **Data dependency:** Accuracy relies on high-quality data and system integrity
- **Organizational change:** Successful adoption requires a shift toward data-driven decision-making and proactive maintenance cultures

Despite these hurdles, the increasing sophistication and accessibility of predictive maintenance, driven by advancements in IIoT and AI, make it an increasingly compelling strategy for industries seeking operational excellence and a competitive edge.

Electrical Load Forecasting in Industry

Electricity is presently the most important energy vector in both the domestic and industrial sectors. Since, unlike fuels, it is hard and expensive to store electricity, there is a need for a precise coupling between its generation and demand. Electrical energy load forecasting, hence, is very vital. Depending upon the time range (forecasting horizon) electrical load forecasting is classified into the following three categories:

- **Short-term load forecasting (STLF):** The forecast is made for 1 hour to a few weeks
- **Medium-term load forecasting (MTLF):** The forecast duration spreads from a few weeks to a few months
- **Long-term load forecasting (LTLF):** Here, the forecasting is done from a few months to years

Depending upon the need and application one may have to plan either one or all of the previous load forecasting categories. In recent years, a lot of research work has been done in the area of STLF. STLF can assist industries by providing an accurate means to predict future load, which can help in precise planning, decrease operating costs, and, thus, increase profit and provide a more reliable electrical supply. STLF predicts future energy demands based on historical data (acquired through smart meters) and predicted weather conditions.

The load forecasting problem is a regression problem (*Chapter 3*). It can be modeled as a time-series problem or as a static model. Modeling load forecasting as time-series data is the most popular choice. With time-series modeling, we can use the standard ML time-series models such as **Autoregressive Moving Average (ARIMA)**, or we can make use of DL models such as **recurrent NNs (RNNs)** and **LSTM**.

STLF using LSTM

Here, we present the code for performing STLF with the help of LSTM. The data for training and testing is taken from the UCI ML website (<https://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption#>):

1. We begin by importing the necessary libraries for energy load forecasting using the UCI ML dataset. We include the `ucimlrepo` package to download the data directly from the UCI ML repo:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import (
    mean_squared_error, mean_absolute_error, r2_score)
import keras as K
```



```

from keras.models import Sequential
from keras.layers import Dense, LSTM, Dropout
from keras.callbacks import EarlyStopping
from datetime import datetime, timedelta
from ucimlrepo import fetch_ucirepo
import warnings
warnings.filterwarnings('ignore')

```

2. We retrieve the *Individual Household Electric Power Consumption* dataset from the UCI ML repository. This dataset contains energy consumption measurements collected from a household over a period of time:

```

individual_household_electric_power_consumption = fetch_
ucirepo(id=235)

```

3. We next preprocess the data. As the first step, we convert the date and time columns into a single datetime index to enable time-series analysis. The original 'Date' and 'Time' columns are then removed as they are no longer needed. Next, we ensure that all columns contain numerical values by converting them to the appropriate data type. Any non-numeric values or missing entries are handled by coercing errors and dropping rows with NaN values:

```

data = individual_household_electric_power_consumption.data.
features
data['datetime'] = pd.to_datetime(
    data['Date'] + ' ' + data['Time'],
    format='%d/%m/%Y %H:%M:%S'
)
data.set_index('datetime', inplace=True)

data.drop(['Date', 'Time'], axis=1, inplace=True)

for col in data.columns:
    data[col] = pd.to_numeric(data[col], errors='coerce')
data = data.dropna()

```

4. Since the original dataset contains minute-level readings, we aggregate the data to hourly intervals using the `resample` method. This transformation helps smooth out short-term fluctuations and provides a more manageable time series for forecasting:

```

data_hourly = data.resample('h').mean()

```

5. We do a little feature engineering. We extract useful time-based features from the datetime index to enhance the model's ability to learn patterns in energy consumption. These include the hour of the day, day of the week, month, and day of the year, which help capture daily, weekly, and seasonal variations:

```
data = data.copy()
# Extract time-based features
data.loc[:, 'hour'] = data.index.hour
data.loc[:, 'dayofweek'] = data.index.dayofweek
data.loc[:, 'month'] = data.index.month
data.loc[:, 'dayofyear'] = data.index.dayofyear
```

6. To further improve predictions, we compute rolling statistics such as the 6-hour rolling mean and standard deviation of energy consumption. These features help the model understand recent trends and fluctuations:

```
data.loc[:, 'rolling_mean_6h'] = data[
    'Global_active_power'].rolling(window=6).mean()
data.loc[:, 'rolling_std_6h'] = data[
    'Global_active_power'].rolling(window=6).std()
```

7. Additionally, we introduce lag features, which provide historical data points as inputs:

```
data.loc[:, 'lag_1h'] = data[
    'Global_active_power'].shift(1)
data.loc[:, 'lag_24h'] = data[
    'Global_active_power'].shift(24)
data.loc[:, 'lag_7d'] = data[
    'Global_active_power'].shift(7 * 24)
```

8. We define features that will be used as model input and normalize them using StandardScaler:

```
features = [
    'Global_active_power', 'Global_reactive_power',
    'Voltage', 'Global_intensity', 'Sub_metering_1',
    'Sub_metering_2', 'Sub_metering_3', 'hour',
    'dayofweek', 'month', 'dayofyear', 'rolling_mean_6h',
    'rolling_std_6h', 'lag_1h', 'lag_24h', 'lag_7d'
]

scaler = StandardScaler()
data_scaled = pd.DataFrame(
    scaler.fit_transform(data[features]),
    columns=features,
    index=data.index
)
```

9. Since we are using LSTM, as with the previous example we create sequences for time prediction; in the present example, we are keeping sequence length 24. You can see that we are choosing 'Global_active_power' as our output feature, as we want to forecast the electrical load:

```
def create_sequences(data, seq_length=24):
    """Create sequences for time series prediction"""
    xs, ys = [], []
    for i in range(len(data) - seq_length):
        x = data.iloc[i:(i + seq_length)].values
        y = data.iloc[i + seq_length][
            'Global_active_power']
        xs.append(x)
        ys.append(y)
    return np.array(xs), np.array(ys)

X, y = create_sequences(data_scaled)
```

10. Now, we split our X and y sequences into training and test datasets. The split ratio is 80 : 20:

```
split_idx = int(len(X) * 0.8)
X_train, X_test = X[:split_idx], X[split_idx:]
y_train, y_test = y[:split_idx], y[split_idx:]
```

11. Reshape them in a form suitable for model input, resulting in training data shape of (1639270, 24, 16) and testing data shape of (409818, 24, 16):

```
X_train = X_train.reshape(
    X_train.shape[0], 24, len(features))
X_test = X_test.reshape(
    X_test.shape[0], 24, len(features))
```

12. Now, we build a model with two LSTM layers and one Dense layer:

```
def build_lstm_model(input_shape):
    model = Sequential([
        LSTM(
            100, return_sequences=True,
            input_shape=input_shape
        ),
        Dropout(0.2),
        LSTM(50),
        Dropout(0.2),
        Dense(1)
    ])
```

```
model.compile(optimizer='adam', loss='mse')
return model
model = build_lstm_model((24, len(features)))
```

13. We use the **early stopping** feature from Keras to ensure that if validation loss stops improving, the training stops:

```
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=10,
    restore_best_weights=True
)
```

14. Let us now fit the model to our data:

```
history = model.fit(
    X_train, y_train,
    epochs=50,
    batch_size=32,
    validation_split=0.2,
    callbacks=[early_stopping],
    verbose=1
)
```

15. Here is the loss plot for the training and validation sets while training; we can see it is decreasing:

```
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(
    history.history['val_loss'],
    label='Validation Loss'
)
plt.title('Model Training History')
plt.xlabel('Epoch')
plt.ylabel('Loss (MSE)')
plt.legend()
plt.grid(True)
plt.show()
```

Here's our output:

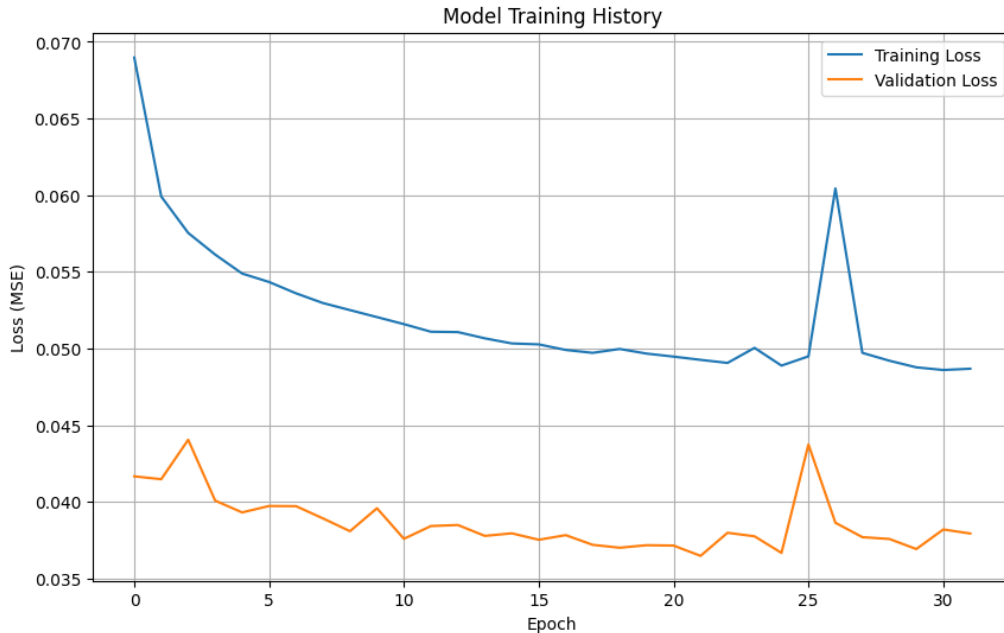


Figure 14.9: Loss plot for training and validation sets

16. We use the trained LSTM model to make predictions on the test dataset:

```
y_pred_scaled = model.predict(X_test)
```

17. Since our data was scaled before training, we must inverse-transform the predictions and actual values to their original scale for meaningful evaluation. To achieve this, we create dummy arrays of the same shape as the original dataset (with all zeros) and place the predicted or actual values in the first column. This ensures the inverse transformation correctly restores the original scale for global active power:

```
dummy = np.zeros((len(y_pred_scaled), len(features)))
dummy[:, 0] = y_pred_scaled.flatten()
y_pred = scaler.inverse_transform(dummy)[:, 0]

dummy = np.zeros((len(y_test), len(features)))
dummy[:, 0] = y_test
y_test_inv = scaler.inverse_transform(dummy)[:, 0]
```

18. Finally, we evaluate the model using MAE, **root MSE (RMSE)**, and **R² score** to assess its predictive accuracy:

```
mae = mean_absolute_error(y_test_inv, y_pred)
mse = mean_squared_error(y_test_inv, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test_inv, y_pred)

print(f'Mean Absolute Error (MAE): {mae:.4f}')
print(f'Root Mean Squared Error (RMSE): {rmse:.4f}')
print(f'R2 Score: {r2:.4f}')
```

We get an MAE of 0.0737, RMSE of 0.19, and an R-square score of 0.95, showing that the model has learned well the given the data.

19. We plot the actual versus predicted values to assess the model's performance. The plot shows that the predicted values closely follow the actual energy consumption trends, with both lines nearly overlapping. This indicates that our model is making highly accurate predictions, effectively capturing patterns in the data:

```
def plot_results(actual, predicted,
                 title='Load Forecasting Results'):
    plt.figure(figsize=(12, 6))
    plt.plot(actual, label='Actual')
    plt.plot(predicted, label='Predicted')
    plt.title(title)
    plt.xlabel('Time')
    plt.ylabel('Global Active Power (kW)')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()

plot_results(
    y_test_inv, y_pred,
    'Short-Term Load Forecasting (Hourly)'
)
```

Here's the result:

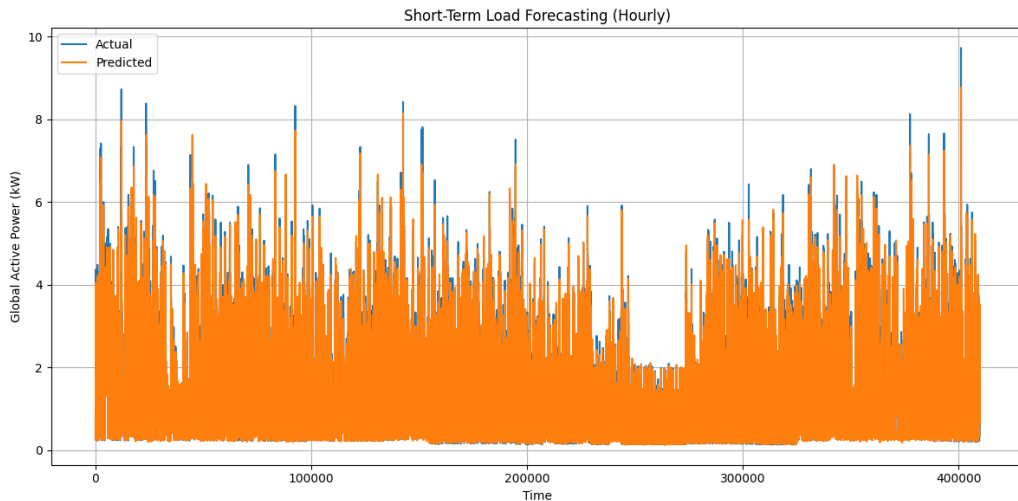


Figure 14.10: STLF

20. We use our trained LSTM model to predict energy consumption for the next 24 hours in a sequential manner. We start with the last available sequence from the test dataset as input. The model predicts the next value, which is then added to the sequence, replacing the oldest value. This process repeats for 24 steps, generating a rolling forecast:

```
last_sequence = X_test[-1].reshape(1, 24, len(features))

next_24h_predictions = []
current_sequence = last_sequence.copy()

for _ in range(24):
    # Predict the next hour
    next_hour_pred = model.predict(current_sequence)[0][0]
    next_24h_predictions.append(next_hour_pred)

    # Update the sequence for the next prediction
    new_row = current_sequence[0, 1:, :].copy()
    new_row = np.vstack([new_row, np.zeros((
        1, len(features)))]
    new_row[-1, 0] = next_hour_pred # Set the predicted value
    current_sequence[0] = new_row
```

21. Since the model outputs scaled values, we apply an inverse transformation to restore the original scale. We then create timestamps for the next 24 hours and plot the forecasted energy load. This visualization provides insights into short-term energy demand trends, which can assist in optimizing power distribution and resource planning:

```
dummy = np.zeros((
    len(next_24h_predictions), len(features)))
# Global_active_power is the first column:
dummy[:, 0] = next_24h_predictions
forecast = scaler.inverse_transform(dummy)[:, 0]

# Generate forecast dates
last_date = data.index[split_idx + len(y_test) - 1]
forecast_dates = [
    last_date + timedelta(hours=i+1) for i in range(24)
]

# Plot forecast
plt.figure(figsize=(12, 6))
plt.plot(
    forecast_dates, forecast,
    label='Forecast', marker='o'
)
plt.title('24-Hour Load Forecast')
plt.xlabel('Time')
plt.ylabel('Global Active Power (kW)')
plt.xticks(rotation=45)
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```


Finally, here's the output:

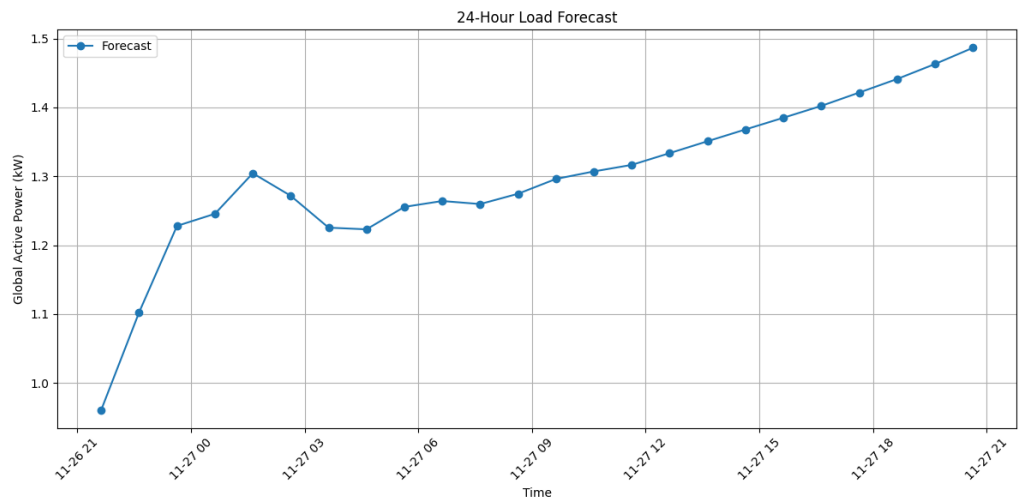


Figure 14.11: 24-hour load forecast

The plot shows the predicted global active power (kW) for the next 24 hours. The forecast captures variations in energy consumption, with an initial increase, followed by fluctuations, and a steady upward trend. This indicates the model's ability to recognize short-term patterns while projecting future demand. Such forecasts help in optimizing energy distribution and resource planning.

Note

For a comprehensive review of various strategies and models used in the electrical load forecasting, refer to this paper:

Fallah, S., Deo, R., Shojafar, M., Conti, M., and Shamshirband, S. (2018). Computational Intelligence Approaches for Energy Load Forecasting in Smart Energy Management Grids: State of the Art, Future Challenges, and Research Directions. Energies, 11(3), 596.

IIoT in Agriculture – Precision Farming with Edge AI

Now, let's turn to agriculture, which is quite different in domain but similar in IIoT and AI synergy. Consider a farm equipped with soil moisture sensors, weather stations, and perhaps drone surveillance. The objective is to optimize irrigation – water the crops only as needed, in each section of the field, to maximize growth while conserving water. AI can help by forecasting soil moisture levels and deciding irrigation schedules. Additionally, AI vision models can analyze images of crops to detect disease or nutrient deficiency, allowing targeted interventions. Agriculture often faces connectivity challenges (farms can be remote with limited internet), which is why **edge AI** becomes important: computations must sometimes happen on a local device (such as a ruggedized farm computer or even on the tractor) rather than relying on constant cloud access.

Agricultural IoT data includes time series from sensors (soil moisture, soil temperature, air humidity, ambient temperature, light, nutrient levels in soil, and so on), typically collected at regular intervals. It may also include discrete events (for example, rainfall events from weather data) and image data from drones or cameras. One real-world dataset, for example, comes from an edge IoT irrigation system for tomato crops (Data details: <https://data.mendeley.com/datasets/33cngpcrmx/2>). This dataset records environmental readings: temperature, humidity, soil moisture, plus soil nutrient (NPK) levels and external data such as wind speed and solar radiation. The data was used to train an AI model that predicts soil health and schedules underground drip irrigation optimally.

Many AI techniques come together in smart farming. For our focus, we highlight two: forecasting and decision-making (control):

- **Forecasting:** Predicting parameters such as soil moisture or crop water requirement in the near future. This could be as simple as predicting tomorrow's soil moisture given weather forecasts and current readings (a time-series regression) or as complex as predicting crop yield at the end of a season (which might use both time-series and other data such as satellite imagery over time). Techniques range from regression models to LSTM/transformers similar to the load forecasting discussed previously. For example, one could train a model to predict soil moisture 12 hours ahead; inputs might be recent moisture readings, predicted rainfall, temperature, and perhaps crop stage.
- **Decision-making (control):** Using AI to decide *actions* such as when to irrigate, how much water to apply, or where to apply fertilizer. This can be approached with rules (if predicted moisture drops below the threshold by morning, then irrigate 10 mm) or with RL (an agent learns an irrigation policy that maximizes crop yield while minimizing water use). In practice, many smart irrigation systems use a threshold-based control: for example, if soil moisture falls below X in a zone, pump water until it rises to Y . AI can enhance this by adjusting thresholds based on crop growth stage or weather (lower threshold if rain is expected). More advanced systems use optimization algorithms that consider the entire farm's needs and water availability.

Agriculture showcases the need for hybrid edge-cloud solutions. Edge AI devices on the farm (such as a local PC or IoT gateway) can perform immediate tasks such as reading sensor data, running a small NN to decide irrigation in real time, and directly controlling pumps. This reduces reliance on internet connectivity – indeed, Microsoft's **FarmBeats** project specifically designed edge components because rural connectivity can be poor. In FarmBeats, edge devices aggregate sensor and drone data, do initial processing (even some model inference) on the farm, and then sync with the cloud when possible. The cloud is used for heavier analytics, model retraining, and combining data from multiple farms to improve models. This architecture ensures that even if the farm goes offline temporarily, critical functions (such as keeping plants watered) continue via edge intelligence.

Next is an example of how such a system could be implemented in Python, where a trained ML model is used to predict if the soil moisture is low. Based on the prediction, an edge device can automatically activate a water pump, ensuring timely irrigation even without cloud access.

1. We begin by importing the essential libraries for data analysis, feature engineering, model training, and evaluation:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import OrdinalEncoder
import numpy as np
from sklearn.preprocessing import (
    PolynomialFeatures, StandardScaler)
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    f1_score, roc_auc_score, confusion_matrix,
    ConfusionMatrixDisplay)
```

2. Read the dataset in a pandas DataFrame; the dataset is downloaded from Mendeley (<https://data.mendeley.com/datasets/33cngpcrmx/2>):

```
df = pd.read_csv('tomato_irrigation_dataset.csv')
```

3. We drop the 'Crop Coefficient stage' column since it's categorical and not suitable for a correlation matrix. After that, we plot the correlations among the numerical features. The heatmap reveals that soil moisture has a weak correlation with most other features, suggesting that simple linear relationships may not be sufficient:

```
df_numeric = df.drop('Crop Coefficient stage', axis=1)

plt.figure(figsize=(12, 10))
correlation_matrix = df_numeric.corr()
sns.heatmap(
    correlation_matrix, annot=True,
    cmap='coolwarm', fmt=".2f"
)
plt.title('Correlation Matrix of Numerical Features')
plt.show()
```

Below you can see the heatmap of the correlation matrix of all the numerical features.

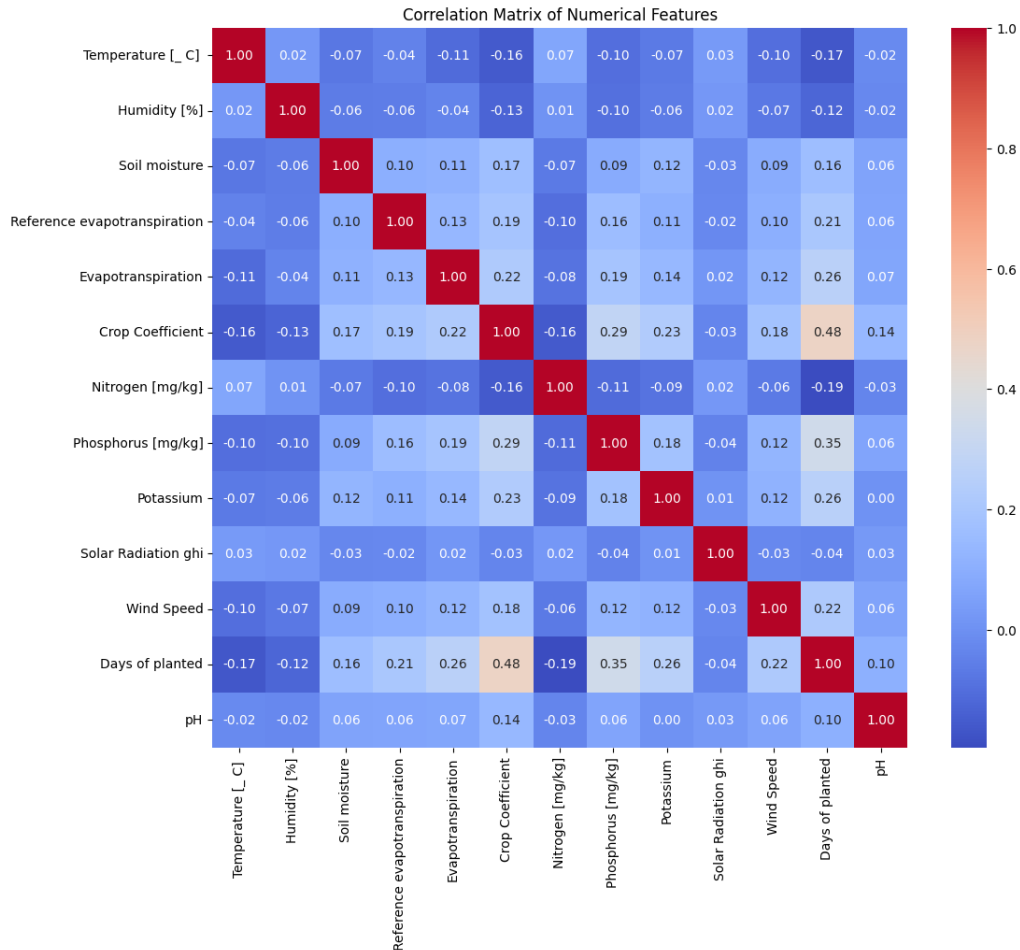


Figure 14.12: Correlation heatmap among different features

- We select the relevant numerical features from our dataset and then apply `PolynomialFeatures` with degree 2 to generate both squared terms and pairwise interactions. This transformation helps the model capture nonlinear relationships and interactions between features, which are especially useful when direct correlations with soil moisture are weak:

```
numerical_features = [
    'Temperature [°C]', 'Humidity [%]',
    'Soil moisture', 'Reference evapotranspiration',
    'Evapotranspiration', 'Crop Coefficient',
    'Nitrogen [mg/kg]', 'Phosphorus [mg/kg]',
    'Potassium', 'Solar Radiation ghi',
```

```

        'Wind Speed', 'Days of planted', 'pH'
    ]
    X = df[numerical_features]

    # Create interaction features (degree=2)
    poly = PolynomialFeatures(
        degree=2,
        interaction_only=False,
        include_bias=False
    )
    X_poly = poly.fit_transform(X)

```

5. We scale the polynomial features for consistent input ranges, then define a threshold of 400 to convert soil moisture into a binary label—1 for sufficient moisture and 0 for low moisture requiring irrigation:

```

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_poly)

# Define a threshold for soil moisture
threshold = 400 # Example threshold, adjust as needed
                # based on the distribution
y = (df['Soil moisture'] >= threshold).astype(int)

```

6. We split the data into training, validation, and test sets—70% for training, and 15% each for validation and testing—to train the model and evaluate its performance on unseen data:

```

X_train, X_temp, y_train, y_temp = train_test_split(
    X_scaled, y,
    test_size=0.3, random_state=42
)
X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp,
    test_size=0.5, random_state=42
)

```

7. We initialize a logistic regression model with selected hyperparameters, using L1 regularization to encourage feature sparsity, and train it on the training set:

```

params = {'C': 0.1, 'penalty': 'l1', 'solver': 'liblinear'}

# Initialize and train the model with best hyperparameters
final_model = LogisticRegression(
    max_iter=1000, **params)
final_model.fit(X_train, y_train)

```

8. We use the trained model to predict soil moisture status on the test set, allowing us to evaluate its real-world decision-making performance:

```
y_pred_final = final_model.predict(X_test)
```

9. We evaluate the model using standard classification metrics. The high accuracy, precision, recall, F1-score, and AUC-ROC indicate that the model performs exceptionally well in identifying low soil moisture conditions for irrigation decisions:

```
accuracy = accuracy_score(y_test, y_pred_final)
precision = precision_score(y_test, y_pred_final)
recall = recall_score(y_test, y_pred_final)
f1 = f1_score(y_test, y_pred_final)
auc_roc = roc_auc_score(y_test, y_pred_final)
```

```
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1-score: {f1}")
print(f"AUC-ROC: {auc_roc}")
```

```
>>> output
Accuracy: 0.9977777777777778
Precision: 1.0
Recall: 0.9957805907172996
F1-score: 0.9978858350951374
AUC-ROC: 0.9978902953586498
```

10. Let us now visualize the confusion matrix to examine the model's predictions. The result shows near-perfect classification, with only one misclassified instance, confirming the model's reliability for real-time irrigation decisions:

```
cm = confusion_matrix(y_test, y_pred_final)
disp = ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot()
plt.show()
>>> Output
```

The figure 14.1 shows the confusion matrix heatmap.

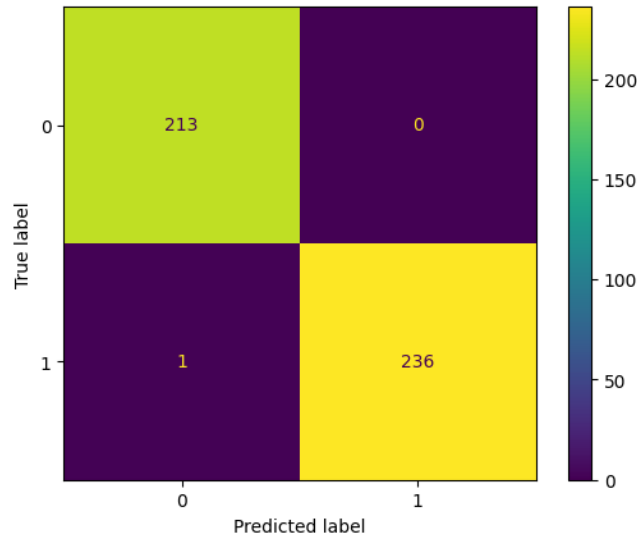


Figure 14.13: Confusion matrix heatmap

With a simple yet powerful ML model, we’re now able to classify soil moisture levels with high accuracy. This model can be deployed on an edge device such as a Raspberry Pi or Arduino, where the prediction (0 for dry, 1 for sufficient moisture) can directly trigger a relay to turn the water pump on or off. With a bit of creativity and the right sensors, we can transform data into decisions—and decisions into action.

Microsoft FarmBeats

Microsoft FarmBeats is a Microsoft Research project aimed at enabling data-driven farming, particularly in remote areas with limited resources. The project addresses the challenge of increasing global food production by leveraging technology to improve farm productivity and reduce costs. FarmBeats utilizes an end-to-end approach, integrating low-cost sensors, drones, and cloud-based computing with ML algorithms. It tackles the issues of lack of power and internet connectivity in farms by employing solutions such as solar-powered sensors and white space-based internet connectivity. By collecting and analyzing data on soil conditions, temperature, and moisture levels, FarmBeats aims to augment farmers’ knowledge and intuition, helping them make informed decisions to optimize yields. The project also has an educational component, “FarmBeats for Students,” designed to introduce precision agriculture concepts to students using affordable hardware and curriculum.

For more details, you can visit the Microsoft Research FarmBeats project page: <https://www.microsoft.com/en-us/research/project/farmbeats-iot-agriculture/>

The farm of Craig Blackburn in Australia integrated IoT sensors with an Azure IoT and Schneider Electric system to automate irrigation. From a smartphone, Blackburn could monitor soil moisture and operate irrigators, customizing settings per zone (<https://www.iothub.com.au/news/iot-in-action-how-data-driven-farming-can-help-feed-the-world-519263>). The results were substantial water savings and increased crop yields.

Emerging Trends: digital twins and AR/VR in IIoT

As IIoT evolves, emerging technologies such as digital twins and **augmented reality/virtual reality (AR/VR)** integration are becoming key enablers for smarter industrial systems. Digital twins are virtual replicas of physical assets that update in real time using sensor data, allowing operators to simulate different scenarios before applying changes in the real world. In sectors such as manufacturing and utilities, digital twins are used for equipment health monitoring, energy optimization, and what-if analyses.

Meanwhile, AR/VR is gaining traction in maintenance and training workflows. Technicians equipped with AR glasses can view live sensor overlays, step-by-step repair instructions, or remote expert guidance while working on complex machinery—reducing errors and downtime. These immersive experiences are not only improving operational efficiency but also transforming how workers interact with IIoT systems on the factory floor.

Summary

In this penultimate chapter, we explored how the fusion of AI and IIoT is revolutionizing industrial operations. The chapter detailed how smart sensors, real-time data, and advanced analytics drive predictive maintenance, energy load forecasting, and precision farming. Practical case studies and detailed code examples illustrated the use of traditional ML and DL models—such as Random Forests and LSTM networks—to predict equipment failures, optimize energy consumption, and manage industrial assets efficiently. The chapter explained techniques for failure classification and RUL prediction, emphasizing the shift from reactive to proactive maintenance strategies. Additionally, the chapter discussed electrical load forecasting using time-series analysis and LSTM models and examined how AI-driven decision-making improves fleet management and enhances safety in transportation. The integration of edge and cloud computing was highlighted as a key enabler for real-time analysis and robust system performance. Overall, the chapter demonstrated that AI-powered IIoT not only improves operational efficiency and reduces downtime but also paves the way for innovative business models and smarter cities, setting a strong foundation for the next wave of industrial transformation. By merging cutting-edge sensor technology with AI analytics, industries can proactively address challenges and unlock new growth opportunities globally.

The next chapter will showcase how AI and IoT can help to shape better cities.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/ds>



AI for Smart Cities IoT

The concept of **smart cities** has gained significant traction in recent years as urban areas grapple with challenges such as population growth, resource management, and environmental sustainability. Smart cities leverage technology and data to enhance the quality of life for citizens, improve the efficiency of urban services, and promote economic growth. At the heart of this transformation lies the convergence of AI and the IoT. This chapter introduces you to smart cities. Case studies will be used to demonstrate how the concepts learned in this book can be applied in developing various smart city components. While reading this chapter, you'll learn about the following:

- Why Do We Need Smart Cities?
- AI Applications in Smart Cities
- AI-Powered Urban Analytics
- Adapting IoT for Smart Cities and the Necessary Steps

Technical requirements

To run the code in this chapter, we will be using NumPy, pandas, Matplotlib, and scikit-learn standard libraries. You can run the code directly on Google Colab without any installation. The code is available at <https://github.com/PacktPublishing/Hands-On-Artificial-Intelligence-for-IoT-Second-Edition/tree/main/Chapter15>.

Why Do We Need Smart Cities?

If you look around you, you'll see cities bursting at the seams. Our urban centers, once symbols of progress and opportunity, are increasingly grappling with challenges that threaten to overwhelm their very fabric. From crippling traffic congestion to strained resources and environmental concerns, the pressure is mounting. Recent data paints a stark picture. Urban populations are exploding, with projections showing that by 2050, nearly 70% of the world's population will reside in cities (<https://www.un.org/uk/desa/68-world-population-projected-live-urban-areas-2050-says-un>). This unprecedented growth is placing immense strain on existing infrastructure and services.

The most difficult challenge is making resources and energy available to all of the inhabitants and, at the same time, avoiding environmental deterioration. At present, cities consume 75% of the world's resources and energy and generate 80% of greenhouse gases; while there's a trend toward green energy sources, we all understand that the Earth's resources, such as food and water, are limited. Another critical challenge is administration and management; with the growing population, strategies will be needed to prevent sanitation problems, mitigate traffic congestion, and thwart crime.

Traditional, reactive approaches to city management are simply no longer sufficient. We need a paradigm shift - a data-driven approach, and that's precisely what smart cities offer.

Smart cities leverage the power of technology and data to create urban environments that are more efficient, sustainable, and livable. At the heart of this transformation lies data. Think of historical data on traffic patterns, energy consumption, and citizen feedback. This rich repository of information, meticulously collected and analyzed, becomes the city's nervous system, enabling city planners and officials to make informed decisions like never before. Instead of guesswork, or reactive measures, smart cities are about dynamic urban adaptation, evolving in real time based on concrete evidence.

According to Techopedia (<https://www.techopedia.com/definition/31494/smart-city>), a smart city is a city that utilizes information and communication technologies to enhance the quality and performance of urban services (such as energy and transportation) so that there's a reduction in resource consumption, wastage, and overall costs. Deakin and Al Waer list four factors that contribute to the definition of a smart city:

- Using a wide range of electronic and digital technologies in the city infrastructure
- Employing ICT to transform living and working environment
- Embedding ICT in government systems
- Implementing practices and policies that bring people and ICT together to promote innovation and enhance the knowledge that they offer

Important note

The paper by Deakin and Al Waer defines a smart city and focuses on the transition required: Deakin, M., and Al Waer, H. (2011). From intelligent to smart cities. *Intelligent Buildings International*, 3(3), 140-152

Hence, a smart city would be a city that not only possesses ICT but also employs technology in a way that positively impacts the inhabitants.

Smart cities leverage IoT data and AI to improve urban life. Massive data streams from sensors - traffic cameras, air quality monitors, smart meters, and connected devices - offer rich insights into city operations. Applying AI to this data helps city officials optimize traffic flow, reduce energy consumption, streamline waste collection, and enhance public services.

In this chapter, we explore how AI techniques can turn raw smart city data into actionable intelligence. We focus on two hands-on examples: one on **AI-driven traffic optimization** and another on **crime hotspot prediction**.

AI Applications in Smart Cities

A smart city has lots of use cases for AI-powered IoT-enabled technology, from maintaining a healthier environment to enhancing public transport and safety. In the following diagram, you can see some of the use cases for a smart city:

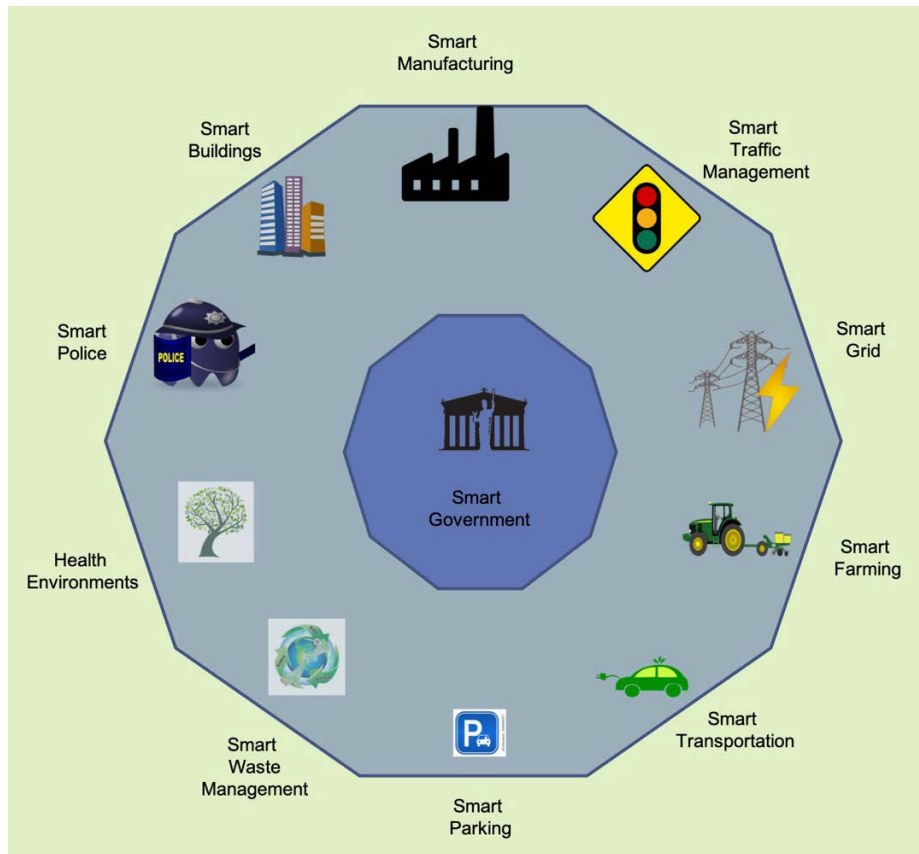


Figure 15.1: AI applications in smart cities

In this section, we'll provide an overview of the most popular use cases - some of them already implemented in smart cities across the world.

Smart traffic management

Smart traffic management utilizes advanced sensors, cameras, data analytics, and AI to monitor and optimize urban traffic flows in real time. By processing data from diverse sources, cities can dynamically adjust traffic signals, detect incidents promptly, and offer real-time routing guidance. The goal is to reduce congestion, enhance road safety, minimize emissions, and improve overall urban mobility. Let us look at some cities around the world that have successfully implemented smart traffic management systems:

- **Singapore:** Singapore began its journey in the late 1990s with **Electronic Road Pricing (ERP)** to manage congestion through dynamic tolls. Over time, the **Land Transport Authority (LTA)** integrated real-time sensors, CCTV cameras, and analytics, creating an advanced platform that adjusts traffic signals and responds rapidly to incidents (<https://www.lta.gov.sg/content/ltagov/en.html>).
- **Barcelona:** Barcelona adopted traffic sensors and CCTV networks and integrated these with public transportation systems to optimize flows. The city focuses on reducing private car dependency through well-coordinated traffic signals and comprehensive public transit.
- **Los Angeles:** With some of the world's most congested roads, Los Angeles employed road-surface sensors and cameras feeding data into a central system. In 2018, it introduced Advanced Transportation Controller cabinets with **vehicle-to-infrastructure (V2I)** and 5G connectivity, enabling direct communication with vehicles to display real-time traffic signal information. (<https://dpw.lacounty.gov/TNL/ITS/>).
- **Toronto:** Toronto's incremental strategy involves IoT sensors, data analytics, and integrated management platforms. Each technological enhancement aligns with broader urban development, ensuring traffic solutions scale effectively alongside population growth.

Smart traffic management systems offer considerable advantages, such as reducing congestion and emissions through dynamic traffic control, enabling prompt incident detection for faster response times, and improving safety while enhancing the overall travel experience. However, they also pose significant challenges, including high initial investment costs, the need for robust data privacy and cybersecurity measures, and ongoing requirements for scalability and maintenance to keep pace with evolving urban demands. We will look at some challenges further on in the chapter.

Smart parking

Anyone living in a city must have felt the struggle of finding a parking spot, especially during the holiday time. Smart parking can ease the struggle. Smart parking systems employ advanced technologies—such as sensors, data analytics, and automated mechanisms—to enhance the efficiency of urban parking infrastructures. By providing real-time information on parking availability and facilitating automated vehicle storage and retrieval, these systems aim to reduce congestion, minimize environmental impact, and improve the overall parking experience for users. Here are some real-world examples:

- **San Francisco's SFpark:** Launched in 2011, SFpark adjusts parking rates based on demand. Sensors embedded in parking spaces provide real-time occupancy data, and drivers can view availability on mobile apps or websites. This demand-based pricing reduced search times and lowered congestion and emissions.
- **Bold Lane car park in Derby, UK:** Known for advanced security since 1998, Bold Lane uses motion sensors, barcode ticketing, and CCTV to trigger alarms on unauthorized activity. No reported criminal incidents have occurred since its technology upgrade.
- **Automated parking systems in Wolfsburg, Germany:** Wolfsburg's automated robotic systems can retrieve vehicles in under two minutes, optimizing space usage and showcasing how automation can tackle urban parking challenges.
- **Nice, France:** The city uses thousands of U-Spot sensors and kiosks to track parking occupancy in real time. Data from MOV'Smart provides information on rates and alternative transport options, reducing illegal parking and improving overall efficiency.
- **Dubai, UAE:** Dubai's smart parking leverages NB-IoT sensors and integrates parking operations with the city's toll system to allow seamless entry and exit, supporting a broader Smart Dubai strategy focused on advanced urban mobility (<https://smartparkingsystems.com/en/smart-parking-systems-dubai-nb-iot-sensor-implementation/>).

Smart waste management

Modern cities face a mounting waste crisis with nearly *2 billion tons of solid waste* generated in 2016 - a figure expected to double to *4 billion tons by 2050*. Traditional waste collection methods, which rely on fixed schedules and manual pickups, struggle to keep up. Trucks often empty half-full bins, leading to wasted fuel and labor, while some bins overflow due to rigid routes. In fact, unnecessary garbage pickups can inflate annual collection costs by 70%, and inefficient routing increases carbon emissions by about 50%. These inefficiencies, combined with growing urban populations and sustainability pressures, have spurred interest in **smart waste management**.

Smart waste management refers to using advanced technology - sensors, data analytics, automation, and intelligent infrastructure - to optimize waste collection, processing, and disposal. By leveraging tools such as the IoT and AI, cities can transform waste management into a more efficient, cost-effective, and environmentally friendly system.

To understand how smart waste management works in practice, it is important to look at leading cities and regions that have implemented these systems. The following are some real cities implementing smart waste management:

- **Songdo, South Korea:** Built from scratch on reclaimed land, Songdo designed a truck-free, automated vacuum system that sucks waste through underground tubes to a central facility. Trash is automatically sorted and processed for recycling or incineration for energy, achieving a recycling rate of over 75%.

- **Barcelona, Spain:** Barcelona installed sensor-equipped bins that report fill levels, enabling collection only when needed. This system cuts redundant trips and lowers overflow incidents, saving on fuel and labor costs while reducing emissions.
- **San Francisco, USA:** The city combines strong recycling mandates with AI-powered sorting. Public bins outfitted with sensors report their fill status, reducing overflowing trash cans by up to 80%. At material recovery facilities, AI and robotics increase the accuracy of recycling streams.

Smart policing

Smart policing employs data-driven strategies, AI, IoT sensors, and various analytics tools to prevent and respond to crime more effectively. While these innovations offer benefits—such as quicker response times and enhanced investigative capabilities—concerns over bias and privacy have prompted debates and, in some areas, restrictions.

Let us now look at some countries around the globe and learn how they implement smart policing:

- **United States:** Santa Cruz, CA, deployed predictive policing in 2011 and initially observed a drop in crime but ultimately banned the technology in 2020 over concerns about bias. Los Angeles, CA, similarly reported early success with predictive software but faced public pressure over potential bias, leading to the program's suspension. Meanwhile, New York City, NY, concentrates on integrated surveillance through its Domain Awareness System, which aggregates data from thousands of cameras and other sensors.
- **Europe and United Kingdom:** London, UK, piloted live facial recognition and predictive policing in limited trials, relying on extensive CCTV systems for real-time alerts but faced ongoing concerns about accuracy and discrimination. In Germany, several states tested predictive tools such as PRECOBS for burglary prevention, but many authorities found the benefits unclear and instead pivoted toward intelligence-led and more traditional methods.
- **Asia:** China's extensive camera networks and AI-driven facial recognition, deployed under programs such as **Skynet**, have greatly expanded police surveillance capabilities but also intensified civil liberties concerns. In contrast, Singapore's **Safe City** system integrates sensors in lampposts, drones, and comprehensive data management to achieve highly efficient policing with minimal manpower. Indian cities are leveraging smart policing through extensive CCTV networks and the Aadhaar biometric system, aiming to enhance public safety (especially for women). New Delhi has deployed thousands of cameras paired with analytics to detect harassment or distress, and Delhi Police identified 3,000 missing children in days using facial recognition.
- **Middle East:** Known for high-tech police robots and analytics-driven surveillance; Dubai aims to have a significant portion of its force comprised of robotic units by 2030.

AI-powered surveillance, predictive analytics, IoT sensors (e.g., gunshot detection), body-worn cameras, and drones/robotics form the core of modern smart policing. While effective, these systems must balance security benefits with ethical and privacy considerations.

Beyond traffic, parking, waste, and policing, AI and IoT continue to reshape urban life:

- **Smart governance:** Cities use predictive analytics and real-time monitoring to optimize resource allocation. Blockchain-based records add transparency and security, as seen in various smart city initiatives worldwide.
- **Smart electric grids:** AI-driven forecasting anticipates power demands and integrates renewable sources, while microgrids enhance resilience. Cities such as Singapore and regions in Germany have demonstrated effective AI-based load balancing and predictive maintenance, reducing outages and operational costs.
- **Smart street lighting:** Sensor-equipped streetlights adjust brightness based on activity, cutting energy consumption significantly. Some cities integrate public Wi-Fi, traffic monitoring, and environmental sensors within streetlight networks, increasing both safety and connectivity.

AI's impact is magnified when combined with 5G, IoT, blockchain, and edge computing. 5G enables real-time AI decision-making, while IoT sensors generate valuable urban data. Edge computing ensures low-latency AI processing at traffic signals and in power grids and smart buildings, optimizing efficiency. Blockchain enhances data integrity and decentralization, allowing secure energy trading, smart contracts, and identity management in digital governance.

While AI in smart cities improves efficiency, sustainability, and security, challenges remain—*data privacy, algorithmic bias, cybersecurity risks, and interoperability*. Cities must adopt robust data governance frameworks, public engagement strategies, and ethical AI regulations to balance innovation with security and privacy.

AI-Powered Urban Analytics

This section showcases two practical code examples that highlight how AI can transform urban management. The first example uses reinforcement learning to optimize traffic light control at a simulated intersection, reducing congestion and wait times. The second example employs kernel density estimation to generate a crime heatmap, revealing spatial patterns from historical data. Together, these demonstrations illustrate how advanced analytics can enhance urban mobility, safety, and overall quality of life.

AI-driven traffic optimization

This example demonstrates how artificial intelligence, specifically a reinforcement learning approach using Q-learning (refer to *Chapter 6* to refresh on reinforcement learning concepts), can be applied to optimize traffic flow at an intersection. The code simulates a simple traffic environment with two crossing roads, where vehicles arrive randomly and form queues. By adjusting the traffic light between the vertical and horizontal directions, the Q-learning agent learns to minimize the total queue lengths and reduce waiting times. Let us dive in:

1. Import the required modules:

```
import numpy as np
import matplotlib.pyplot as plt
import random
```

2. We define the traffic environment for a single intersection. There are two directions: vertical and horizontal. The `light` variable indicates which direction is green, 0 signifies that vertical is green, while 1 means horizontal is green. Vehicles arrive randomly at each time with a given probability. When the light is green not all vehicles pass through - only vehicles up to `pass_rate` vehicles in that queue pass. The agent can choose to switch the light (`action=1`) or keep it as is (`action=0`). The reward is a negative total queue length (we want to minimize waiting vehicles):

```
class TrafficEnv:
    def __init__(self, max_queue=10, max_steps=100,
                  arrival_rate=0.3, pass_rate=1):
        # Maximum queue length (for state discretization)
        self.max_queue = max_queue
        # Number of time steps per episode
        self.max_steps = max_steps
        # Probability of vehicle arrival for each direction
        self.arrival_rate = arrival_rate
        # Vehicles that can pass when green
        self.pass_rate = pass_rate
        self.reset()

    def reset(self):
        self.queue_v = 0 # Vertical queue length
        self.queue_h = 0 # Horizontal queue length
        self.light = 0 # Start with vertical green
                        # (0) by default
        self.step_count = 0
        return (min(self.queue_v, self.max_queue),
                min(self.queue_h, self.max_queue),
                self.light)

    def step(self, action):
        if action == 1:
            self.light = 1 - self.light

        # Random vehicle arrivals in each direction
        if np.random.rand() < self.arrival_rate:
            self.queue_v += 1
        if np.random.rand() < self.arrival_rate:
```

```

        self.queue_h += 1

    # Allow vehicles to pass from the green-light queue
    if self.light == 0:
        passed = min(self.pass_rate, self.queue_v)
        self.queue_v -= passed
    else:
        passed = min(self.pass_rate, self.queue_h)
        self.queue_h -= passed

    # Calculate reward
    # (we want to minimize waiting vehicles)
    reward = - (self.queue_v + self.queue_h)

    self.step_count += 1
    done = self.step_count >= self.max_steps

    # Clip the queue lengths to max_queue
    # for a discrete state representation
    next_state = (
        min(self.queue_v, self.max_queue),
        min(self.queue_h, self.max_queue),
        self.light
    )
    return next_state, reward, done, {}

```

3. We set up Q-learning parameters and respective data structures. The light can be in one of the two states, 0 or 1, and the agents can have either action 0 (keep light) or action 1 (switch light):

```

max_queue = 10
env = TrafficEnv(
    max_queue=max_queue, max_steps=100,
    arrival_rate=0.3, pass_rate=1
)

q_table = np.zeros((max_queue+1, max_queue+1, 2, 2))

# Hyperparameters for Q-learning
alpha = 0.1      # Learning rate
gamma = 0.95     # Discount factor
epsilon = 0.2    # Exploration rate
episodes = 500   # Total number of episodes for training

# To track total rewards per episode for visualization
episode_rewards = []

```

4. Now we train the Q-learning agent:

```

for ep in range(episodes):
    state = env.reset()
    total_reward = 0
    done = False

    while not done:
        # Use epsilon-greedy policy to choose an action
        if np.random.rand() < epsilon:
            action = np.random.choice([0, 1])
        else:
            action = np.argmax(
                q_table[state[0], state[1], state[2]])

        # Take the action and observe the new state and reward
        next_state, reward, done, _ = env.step(action)
        total_reward += reward

        # Q-learning update rule
        best_next = np.max(q_table[
            next_state[0], next_state[1], next_state[2]
        ])
        q_table[state[0], state[1], state[2], action] = \
            (1 - alpha) * q_table[
                state[0], state[1], state[2], action] + \
            alpha * (reward + gamma * best_next)

        state = next_state

    episode_rewards.append(total_reward)

```

5. Next, we evaluate the trained policy and collect simulation data:

```

state = env.reset()
states_over_time = [state]
queue_v_list = [state[0]]
queue_h_list = [state[1]]
light_list = [state[2]]

for _ in range(env.max_steps):
    action = np.argmax(q_table[
        state[0], state[1], state[2]
    ])

```

```
next_state, reward, done, _ = env.step(action)
state = next_state
states_over_time.append(state)
queue_v_list.append(state[0])
queue_h_list.append(state[1])
light_list.append(state[2])
if done:
    break
```

6. And now, let us visualize the training performance:

```
plt.plot(episode_rewards, color='blue')
plt.title('Training Performance: Total Reward per Episode')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.grid(True)
```

Here is the reward plot:

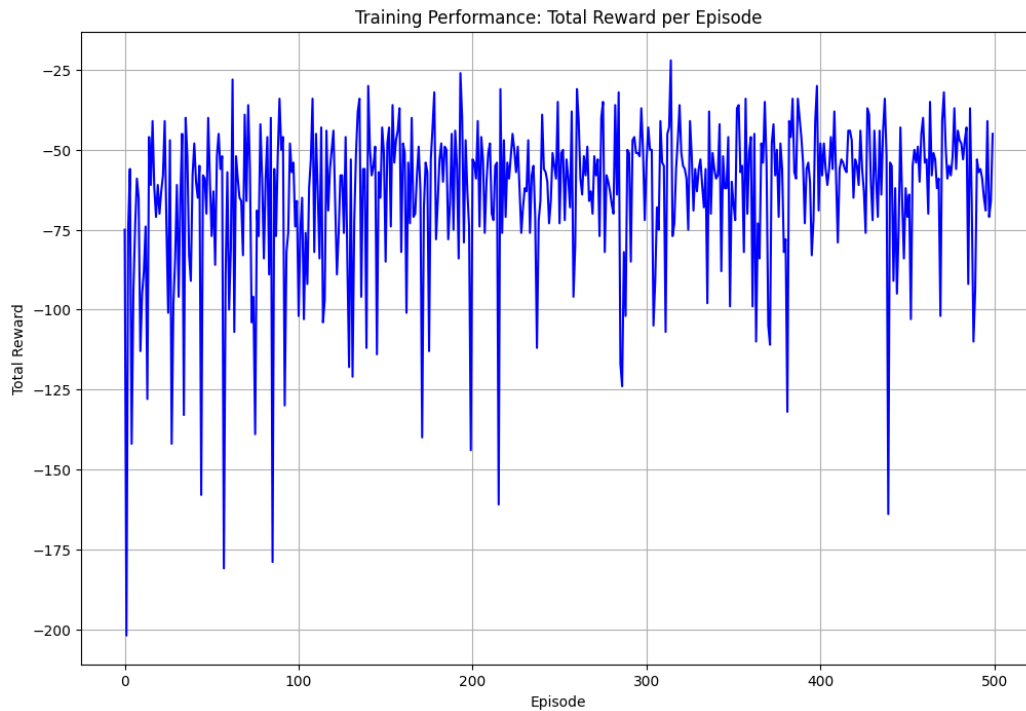


Figure 15.2: Total reward per episode for the Q-learning agent

7. Finally, let us see how waiting times are affected when the learned policy is applied:

```
plt.plot(queue_v_list, label='Vertical Queue', marker='o')
plt.plot(queue_h_list, label='Horizontal Queue', marker='o')
plt.title('Queue Lengths Over Time (Learned Policy Simulation)')
plt.xlabel('Time Step')
plt.ylabel('Queue Length')
plt.legend()
plt.grid(True)
```

The following plot shows the queue length on the two intersections over time.

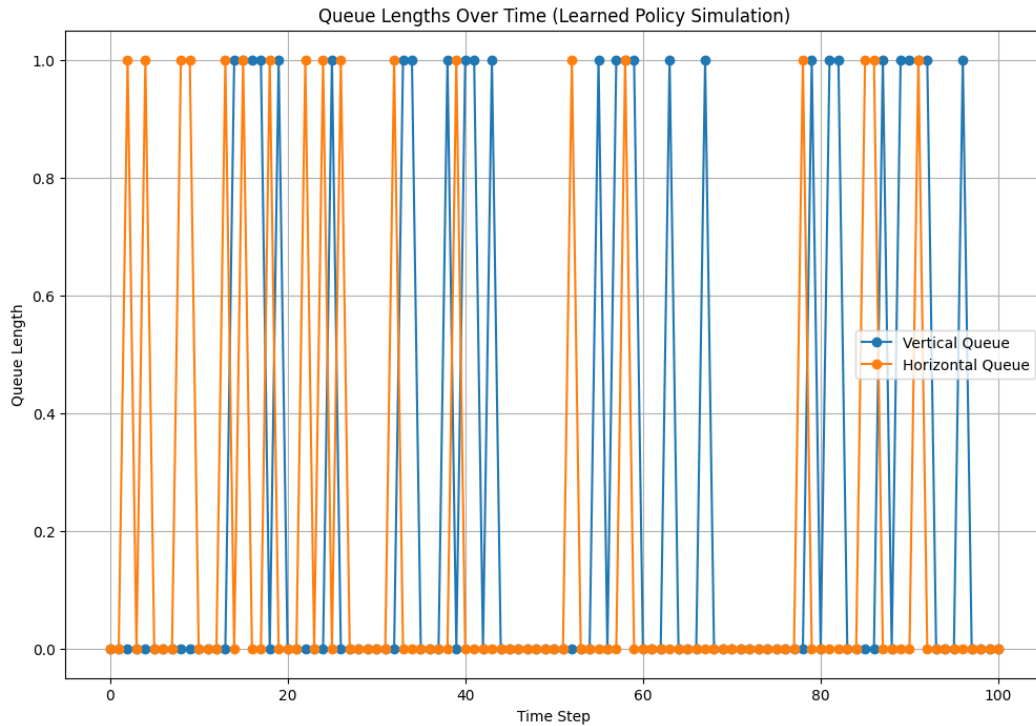


Figure 15.3: Queue length over time

This implementation demonstrates how AI-powered reinforcement learning (Q-learning) can optimize traffic flow at an intersection by dynamically adjusting traffic signals. The agent successfully learns to minimize total queue lengths and reduce waiting times by making data-driven switching decisions. The training performance plot highlights the gradual improvement in rewards over episodes, indicating the system's ability to adapt and refine its policy over time.

The learned policy effectively balances queue lengths in both vertical and horizontal lanes, as seen in the queue length simulation plot. This approach has real-world potential in smart city traffic management, reducing congestion, improving fuel efficiency, and enhancing urban mobility. With further enhancements, such AI-driven optimization can be extended to multi-intersection networks, real-time adaptive control, and integration with connected vehicle data, making modern cities more efficient and sustainable.

Crime hotspots prediction

Next, we'll see how AI can predict crime hotspots by simulating historical crime data and applying **Kernel Density Estimation (KDE)**. KDE is a non-parametric way to estimate the probability density function of a random variable. It places a smooth kernel (often Gaussian) on each data point and sums them to estimate the density across space. In our context, each historical crime point contributes to a "bump" in the density surface, helping us identify hotspots of activity where crime events are more concentrated. For this example, we generate synthetic crime incidents that mimic real-world clustering patterns. The code calculates a density estimate over a geographic grid, visualizing areas with higher probabilities of crime. This heatmap provides a clear, intuitive representation of potential crime concentrations, offering valuable insights for proactive law enforcement and urban planning strategies.

1. We begin by importing the necessary Python libraries for data processing, statistical modeling, and visualization:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.neighbors import KernelDensity
```

2. A function is defined to create simulated crime incidents that resemble real-world data (e.g., San Francisco crime data). It models crime hotspots by defining cluster centers based on typical San Francisco coordinates and then sampling crime incidents around these centers using a normal distribution. The function creates a pandas DataFrame mimicking real-world datasets, including columns for incident type, police district, location (lat/lon), date, and time:

```
def generate_crime_data(n_samples=1000, random_state=42):
    np.random.seed(random_state)
    # Define cluster centers (simulating hotspots)
    # around typical San Francisco coordinates
    centers = np.array([
        [37.78, -122.42],
        [37.76, -122.45],
        [37.80, -122.41]
    ])
    n_clusters = centers.shape[0]
    samples_per_cluster = n_samples // n_clusters
```

```

latitudes = []
longitudes = []

# Generate clustered data points for each hotspot
for center in centers:
    lat = np.random.normal(
        loc=center[0], scale=0.01,
        size=samples_per_cluster)
    lon = np.random.normal(
        loc=center[1], scale=0.01,
        size=samples_per_cluster)
    latitudes.extend(lat)
    longitudes.extend(lon)

# If there are leftover samples,
# generate them uniformly over a broader region
leftover = n_samples - samples_per_cluster * n_clusters
if leftover > 0:
    latitudes.extend(np.random.uniform(
        low=37.75, high=37.81, size=leftover))
    longitudes.extend(np.random.uniform(
        low=-122.48, high=-122.38, size=leftover))

# Create a DataFrame similar in structure to the
# SF police data
data = pd.DataFrame({
    'IncidentID': range(n_samples),
    'Category': np.random.choice(
        ['LARCENY/THEFT', 'ASSAULT',
        'ROBBERY', 'VANDALISM'],
        size=n_samples),
    'PdDistrict': np.random.choice(
        ['NORTHERN', 'SOUTHERN', 'BAYVIEW',
        'RICHMOND', 'TENDERLOIN'],
        size=n_samples),
    'Latitude': latitudes,
    'Longitude': longitudes,
    'Date': pd.to_datetime(np.random.choice(
        pd.date_range('2020-01-01', '2020-12-31'),
        size=n_samples)),
    'Time': np.random.choice(
        pd.date_range(
            '00:00', '23:59', freq='1min').time,

```

```

        size=n_samples)
    })
    return data

```

3. After defining the function, we generate 1,000 synthetic crime incidents:

```
crime_data = generate_crime_data(n_samples=1000)
```

4. To estimate crime-prone areas, we apply KDE on crime coordinates. The model assumes that each crime incident contributes to a smooth probability distribution over the surrounding region by using a **Gaussian kernel**. The bandwidth parameter controls the level of smoothing in the density estimation (lower values focus on small clusters, higher values generalize crime hotspots):

```

coords = crime_data[['Latitude', 'Longitude']].values
kde = KernelDensity(bandwidth=0.005, kernel='gaussian')
kde.fit(coords)

```

5. We create a grid over the city to evaluate crime density at every location. The region is divided into a 100x100 grid, covering the area between latitude (37.75 to 37.81) and longitude (-122.48 to -122.38):

```

lat_lin = np.linspace(37.75, 37.81, 100)
lon_lin = np.linspace(-122.48, -122.38, 100)
lon_grid, lat_grid = np.meshgrid(lon_lin, lat_lin)
grid_coords = np.vstack([
    lat_grid.ravel(), lon_grid.ravel()
]).T

```

6. Next, we compute crime density. The KDE model scores each grid point, and exponentiation converts log-density values into crime probability estimates. The density matrix is reshaped to match the grid for visualization:

```

log_density = kde.score_samples(grid_coords)
density = np.exp(log_density).reshape(lat_grid.shape)

```

7. We plot a heatmap to represent crime intensity across the region:

```

plt.figure(figsize=(10, 8))
plt.imshow(
    density, origin='lower', cmap='hot',
    extent=[-122.48, -122.38, 37.75, 37.81]
)
plt.scatter(
    crime_data['Longitude'],
    crime_data['Latitude'],
    s=10, c='blue', alpha=0.5,

```



```

    label='Crime Incidents'
)
plt.colorbar(label='Crime Density')
plt.title('Predicted Crime Hotspots Based on Historical Data')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.legend()
plt.show()

```

Here is the heatmap plot:

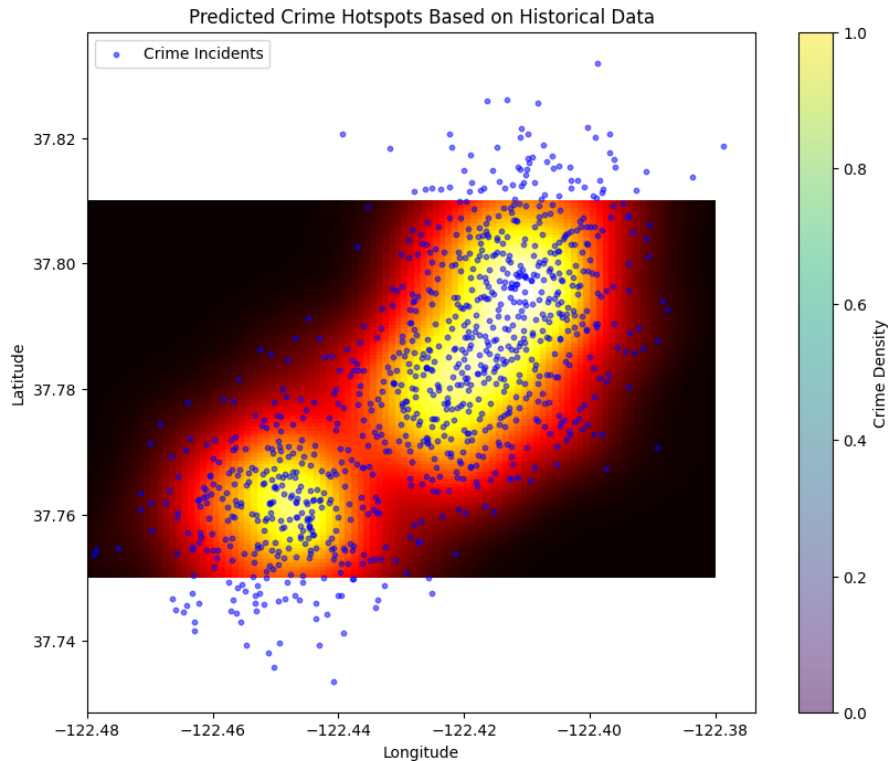


Figure 15.4: Heatmap of crime incidents

The heatmap visually represents predicted crime hotspots based on historical crime data using KDE. The bright yellow and red areas indicate regions with high crime density, suggesting these locations are frequent crime hotspots. The black and dark red areas show regions with low or no crime activity, meaning fewer incidents have been recorded in those zones. The blue dots represent individual crime incidents, and they align with the high-density regions, confirming that KDE effectively identifies historical crime clusters. We can see that crime events are concentrated in two main hotspot regions, suggesting repeat criminal activities in these locations.

For these examples, we used simulated data for ease of demonstration. In a real-world implementation, however, historical crime data would be essential to capture accurate trends and patterns. Many cities provide open data portals where such information is publicly available. For instance, San Francisco's Open Data Portal (<https://data.sfgov.org/>) offers comprehensive crime reports and other public safety datasets. Similarly, cities such as Chicago (<https://data.cityofchicago.org/>) and New York (<https://opendata.cityofnewyork.us/>) share detailed historical records that can be used to refine and enhance predictive models for crime hotspot analysis. You can experiment using tools such as QGIS (<https://qgis.org/>), Kepler.gl (<https://kepler.gl/>), and Python libraries such as scikit-learn, GeoPandas, and Folium to build your own smart city visualizations and predictions.

Adapting IoT for Smart Cities and the Necessary Steps

Building a smart city is not a one-day business, neither is it the work of one person or organization. It requires the collaboration of many strategic partners, leaders, and even citizens. The dynamics of such a collaboration are beyond the scope of this book, but, since the book is for AI enthusiasts and engineers, let's explore what the AI community can do - what are the areas that could provide us with a career or entrepreneurship opportunity? Any IoT platform will necessarily require the following:

- **A network of smart devices:** This includes sensors, cameras, actuators, and other connected devices that continuously collect data throughout the urban environment
- **Field (cloud) gateways:** These gateways aggregate data from low-power IoT devices, store it temporarily, and securely forward it to the cloud for further processing
- **Streaming data processors:** Dedicated systems that aggregate and route the high-volume, real-time data streams to data lakes and control applications
- **A data lake:** A centralized repository that stores all raw data—even those pieces that might seem insignificant at first glance
- **A data warehouse:** This system cleanses, structures, and organizes the collected data for efficient querying and analysis
- **Analytics and visualization tools:** Software that enables the exploration and visualization of sensor data to derive actionable insights
- **AI algorithms and techniques:** Advanced machine learning and AI models that analyze long-term data to identify patterns, predict trends, and enhance the performance of control systems
- **Control applications:** Systems designed to send commands to IoT actuators based on insights generated from the data
- **User applications:** Interfaces that connect smart devices with citizens, ensuring that technology improves daily life

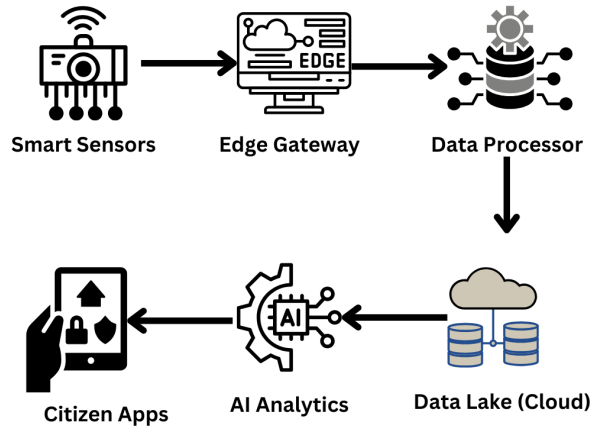


Figure 15.5: Smart city IoT architecture

The smart city IoT architecture consists of several interconnected layers that transform raw sensor data into meaningful urban actions (*Figure 15.5*). At the base, smart sensors collect real-time data on traffic, air quality, utilities, and more. This data is sent to edge gateways for initial processing, enabling quick, local decisions and reducing dependence on internet connectivity. The processed data flows through stream engines for real-time handling and is then stored in scalable cloud infrastructure such as data lakes and warehouses. Advanced analytics and AI models operate on this data to generate insights, forecasts, and alerts. Finally, these insights drive automated control systems and citizen-facing apps, closing the loop between sensing, thinking, and acting—an essential framework for sustainable and responsive urban management.

Additionally, issues surrounding security and privacy are paramount. Service providers must ensure that smart city services are secure and pose no threat to the well-being of citizens. User-friendly design is equally important to foster widespread adoption.

This expansive ecosystem not only drives urban innovation but also creates a multitude of career and entrepreneurial opportunities, especially for AI engineers. To truly capitalize on the data generated by IoT devices, the focus must extend beyond basic monitoring to sophisticated analysis. AI tools play a critical role in uncovering hidden correlations and predicting trends from historical sensor data. These insights empower control applications to make informed, proactive decisions in real time.

Ultimately, building a smart city is an iterative process. With each cycle, new layers of processing, analysis, and intelligent control are added, continuously enhancing the functionality and responsiveness of urban services.

One of the primary hurdles in smart city development is ensuring that the selected technologies work cohesively. Integrating disparate systems—such as IoT devices, data analytics platforms, and legacy infrastructure—can be complex and may require retrofitted solutions to bridge compatibility gaps. Collaboration also poses a significant challenge; successful smart city initiatives demand a paradigm

shift where city planners, technologists, local governments, businesses, environmental advocates, and citizens work in unison. This interdisciplinary approach requires long-term planning across multiple departments and sectors, which can be difficult to coordinate.

Despite these challenges, the benefits of a well-implemented smart city framework are substantial. Cities have always been crucibles of economic opportunity, but the rise of smart cities has redefined how and for whom these opportunities emerge. Far from the dystopian narrative of automation displacing workers, smart cities are proving to be dynamic employment incubators. By integrating technology with urban governance, they create jobs that span from high-skilled AI engineers to community-focused sustainability coordinators.

Smart cities drive economic growth through both direct and indirect job creation. Direct job creation occurs in sectors directly involved in the development, installation, and maintenance of smart infrastructure—for example, positions in manufacturing IoT devices, developing smart software systems, and managing urban data networks. Indirect job creation emerges as these projects stimulate broader economic activity by supporting ancillary sectors such as logistics, retail, and professional services, while also encouraging entrepreneurial ventures and innovation throughout the urban ecosystem.

Example of direct and indirect job creation by smart cities

In August 2024, the Indian Cabinet approved 12 new industrial smart cities under the **National Industrial Corridor Development Programme (NICDP)**. These projects are expected to generate around 1 million direct jobs and up to 3 million indirect jobs while attracting investments worth approximately ₹1.52 lakh crore. This initiative illustrates how smart city use cases not only modernize urban infrastructure and improve operational efficiency but also serve as a powerful engine for job creation and economic growth. This large-scale investment in industrial smart cities is a clear testament to the transformative potential of smart city projects in emerging markets.

Moreover, smart city initiatives contribute to environmental sustainability by reducing energy consumption and promoting resource efficiency. Ultimately, when implemented correctly, these technological advances lead to more economical urban management, improved public services, and an enhanced quality of life for citizens. In alignment with the United Nations **Sustainable Development Goals (SDGs)**, smart cities directly support several global targets. For example, **Goal 11 (Sustainable Cities and Communities)** is advanced through smarter urban planning and infrastructure. **Goal 13 (Climate Action)** is addressed via carbon reduction strategies such as smart grids and intelligent traffic systems. Likewise, **Goal 7 (Affordable and Clean Energy)** benefits from AI-driven energy optimization, while **Goal 6 (Clean Water and Sanitation)** is supported by smart water monitoring and leak detection systems. These integrated efforts underscore how digital transformation can serve not just innovation, but also the planet and its people.

By addressing integration challenges and fostering a collaborative mindset, cities can harness AI-powered IoT solutions to unlock significant economic, environmental, and social benefits.

You don't need big budgets or formal titles to contribute to smart city initiatives. Start small—build an app that shows real-time bus delays, create a dashboard to track local air quality, or visualize road accidents using open data. Reach out to local governments or NGOs to share your insights. Even a basic sensor network measuring temperature and noise in your neighborhood can add value. The important thing is to begin—your curiosity and skills can spark real impact.

Summary

AI-powered IoT solutions can help connect cities and manage multiple infrastructure and public services. This chapter covered diverse use cases of smart cities, from smart lighting and road traffic to connected public transport and waste management. From the successful case studies, we also learned that smart cities can lead to reduced costs for energy, optimized use of natural resources, safer cities, and a healthier environment. This chapter listed some of the open city data portals and the information available there. We use the tools learned about in this book to categorize synthetic crime data into hotspots. And finally, this chapter discussed some challenges and benefits in the building of smart cities.

Six years - that's the span between the first edition of this book in 2019 and this one in 2025, and what a difference those years have made in the world of AI. The acceleration of automation in our cities and organizations, the disruptive force of generative AI entering our daily routines, and the readily available power of open source pre-trained models have reshaped the technological landscape dramatically. It's a privilege to have been able to update this book to reflect these profound shifts. My aim has been to provide you with practical skills and insights to navigate this exciting era. The real impact, however, lies in your hands. Go forth, experiment, build, and contribute to the ongoing evolution of AI and IoT.

References

- SFpark Pilot Program, SFMTA (<https://www.sfmta.com/projects/sfpark-pilot-program>)
- "Types of Traffic Signal Systems", Toronto City Government. Retrieved April 3, 2025. (<https://www.toronto.ca/services-payments/streets-parking-transportation/traffic-management/traffic-signals-street-signs/traffic-signals-in-toronto/types-of-traffic-signal-systems/>)
- "Smart City Spotlight: Barcelona", Channel Futures, May 2016. Retrieved April 3, 2025. (<https://www.channelfutures.com/connectivity/smart-city-spotlight-barcelona>)
- "The World's Most Secure Buildings: Bold Lane Car Park", Hirsch Secure, September 2021. Retrieved April 3, 2025. (<https://www.hirschsecure.com/resources/blog/the-worlds-most-secure-buildings-bold-lane-car-park>)
- "Indian police use facial recognition app to reunite families with lost children," Anuradha Nagaraj, Reuters, Feb 2020. Retrieved April 3, 2025. (<https://www.reuters.com/article/world/indian-police-use-facial-recognition-app-to-reunite-families-with-lost-children-idUSKBN2081CT/>)

- “China launches SkyNet to catch Corrupt official overseas” BBC News. March 27, 2015. Retrieved April 3, 2025 (<https://www.bbc.com/news/world-asia-china-32082137>)
- “Tomorrow’s cities: just how smart is Songo?”, BBC News, September 2013. Retrieved April 3, 2025. (<https://www.bbc.com/news/technology-23757738>)
- Dubai is Introducing robotic policemen, to make 25% of the force by 2030, Emerging Technologies, World Economic Forum, May 2017. (<https://www.weforum.org/stories/2017/05/dubai-is-introducing-robotic-policemen-to-make-up-25-of-the-force-by-2030/>)
- Press Release, Ministry of Commerce and Industry, India (<https://pib.gov.in/PressReleasePage.aspx?PRID=2081537#:~:text=The%20Government%20of%20India%20has,development%20of%20trunk%20infrastructure%20packages.>)

Join our community on Discord

Join our community’s Discord space for discussions with the authors and other readers:

<https://packt.link/ds>



Index

Symbols

3D CNNs 302

3D object generation 200

A

actor-critic algorithm 182, 183

AdaGrad optimizer 125

Adam optimizer 125, 191, 383

adaptive instance normalization
(AdaIN) 197

adaptive lighting 357

Advanced Message Queuing
Protocol (AMQP) 313

adversarial loss 197

AI applications, in smart cities 409

smart parking 410

smart policing 412

smart traffic management 410

smart waste management 411

AI-controlled sailplanes 156

AI-driven decision-making 388

AI-driven implicit automation 358

AI/IoT platforms

supported services 15, 16

AI models, for IoT on cloud platforms 248

Amazon Bedrock 251

Amazon SageMaker 250

Google AutoML 250

Google Vertex AI 249, 250

AIoT 186

AI-Powered IIoT 367, 368

use cases 368, 369

AI-powered urban analytics 413

AI-driven traffic optimization 413-418

crime hotspots prediction 419-423

AI services, in cloud platforms 239

ML APIs 240

AI Techniques, for Visual Data in IoT

deep learning architectures 289

recurrent and temporal

convolutional networks 301

AlexNet 105

AlphaGo Zero 155

Amazon 253

Amazon Alexa 356, 359

Amazon Bedrock 251

features 251

Amazon Elastic Compute Cloud
(Amazon EC2) 236

- Amazon Fraud Detector** 244
- Amazon Go stores** 10
- Amazon Lookout for Equipment** 244
- Amazon Rekognition** 240-243
- Amazon SageMaker** 236, 250
 - features 250
- Amazon services, for IoT and AI** 244
 - Amazon Fraud Detector 244
 - Amazon Lookout for Equipment 244
- Amazon Web Services (AWS)** 233
- Ambient Pressure (AP)** 19
- Ambient Temperature (AT)** 19
- AmoebaNet-A models** 146
- analytic methods** 124
- AnimeGAN** 197, 198
- anomaly detection** 186, 199
 - with variational autoencoders (VAEs) 189-194
- anonymization** 288
- Apple HomeKit** 356
- area under the curve (AUC)** 375
- artificial intelligence (AI)** 3, 13, 185, 214, 285
 - used, for predictive maintenance 369
 - used, for preventive maintenance 370
- artificial neural networks** 93
 - activation function 93
 - learning rate parameter 93
 - loss function 93
- artificial neurons**
 - exploring 87
 - Exponential Linear Unit (ELU)
 - activation function 91
 - hyperbolic tangent activation function 88
 - Leaky Rectified Linear Unit (Leaky ReLU) activation function 90
 - Rectified Linear Unit (ReLU)
 - activation function 89
 - sigmoid activation function 88
 - softmax activation function 90
- asexual reproduction** 129
- asset management** 369
- Atari** 158
- atrial fibrillation** 355
- attention mechanisms**
 - for capturing spotlight, in Visual AI 303
- attribute-based access control (ABAC)** 253
- audio data** 312
- audio data, for IoT applications**
 - audio command recognition 331
 - audio commands, using 333-335
 - feature extraction 331
 - processing 331
 - speech recognition and transcription 332
 - voice assistants, and IoT intergration 332, 333
- augmented reality/virtual reality (AR/VR)** 405
- autoencoders** 113, 114
 - denoising autoencoder 115
 - variational autoencoders 115
- automated ML (AutoML)** 147, 148
- automated RL (AutoRL)** 147, 148
- automatic differentiation** 52, 94
- automatic speech recognition (ASR)** 332
- AutoML methods** 146
- Autoregressive Moving Average (ARIMA)** 389
- average pooling** 104
- AWS IoT** 16
- AWS IoT Analytics** 236
- AWS IoT Core** 236
- AWS Lambda** 236
- AWS offers for IoT applications**
 - AWS Identity and Access Management (IAM) 254

AWS IoT Core 253
 AWS IoT Device Defender 253
 AWS IoT Device Management 253
 AWS Key Management Service (KMS) 254

Azure Entra ID 254

Azure for IoT environments 254

Azure Entra ID 254
 Azure IoT Hub 254
 Azure RBAC 254
 Azure Security Center for IoT 254
 Azure Sphere 254

Azure IoT Hub 247, 254

Azure RBAC 254

Azure Security Center for IoT 254

Azure Speech SDK

to capture speech commands,
 in Python 245-247

Azure Sphere 254

B

backpropagation algorithm

backward pass 96
 compute loss 96
 forward pass 96
 repeat 96

backpropagation through time (BPTT) 110

Bayesian networks 47

Bayes probability theorem 64

Bidirectional Encoder Representations from Transformers (BERT) 208, 323, 324

big data approach

and IoT 9-11

BiLSTMs 324

binary cross-entropy 140

bit-flip mutation 133

Blender 157

botnets 361

Box2D 158

C

catastrophic forgetting 171

categorical-cross entropy loss function 93

Cauchy-Euler method 124

challenges handling, Visual Data in IoT 287

data privacy and ethical issues,
 mitigating 288
 examples of datasets 288, 289
 hardware constraints 287

challenges handling, time-series data

data quality 263
 data volume 262

channels 103

character and environment generation 199

Chronos 283

classical ML 324

classic control 158

classification, with SVMs 59, 60

kernel trick 62
 maximum margin hyperplane 60, 61
 wine, classifying 62-64

cloud computing 234

basics 235
 challenges 234, 235

cloud deployment models 237

hybrid clouds 238
 private clouds 237
 public clouds 237

Cloud IAM 255

Cloud IoT Core 255

cloud providers

Amazon 253, 254
 Google Cloud 255
 Microsoft Azure 254

cloud services

- IaaS 235
- PaaS 236
- SaaS 236
- types 235

CNN architectures

- milestones 105-107

combined cycle power plant (CCPP) 18

- features 19

comma-separated value (CSV) files 24

- reading, with NumPy module 29, 30
- reading, with pandas module 28
- with csv module 24-27

complex event processing (CEP) 12**computerized maintenance management system (CMMS) 371****computer numerical control (CNC) 372****conditional generative adversarial networks (cGANs) 186, 196**

- in Keras 200-207

content loss measures 197**contextual embeddings 318, 323****continuous action spaces 181****convex problem 123****convex shape 123****convolutional neural networks (CNNs) 100, 101, 324**

- convolution layer 102
- fully connected layers 101
- pooling layer 104, 105

convolved image 103**cross-entropy loss function 56****cross-industry standard process for data mining (CRISP-DM) 14**

- business understanding 14
- data preparation 14
- data understanding 14

- deployment 14

- evaluation 14

- modeling 14

crossover 130, 132

- multi-point crossover 133
- one-point crossover 132

Cross-Stage Partial Network (CSPNet) 294**cross-validation 77**

D

data 12, 21**data augmentation 186, 199****Database Management System (DBMS) 38****data collection 313****data-handling, capabilities**

- atomic 12
- micro batching 12
- windowing 12

data management (DM) 14**DataNode 42****DataParallel API 218****DataParallel class 223****data science**

- in IoT 12

data segmentation 238**datasets 34**

- combined cycle power plant dataset 18
- wine quality dataset 19, 20

DEAP 135**decision trees 48, 67-69**

- in scikit 69, 70
- nodes 68
- using 70, 71

Deep Deterministic Policy Gradient (DDPG) 151

- for continuous action space 181, 182

deep learning architectures 289

- EfficientNet 292
- MediaPipe 298-301
- MobileNet 290-292

deep learning (DL) 3, 13, 121, 185, 324, 369
 in regression 50

DeepMind AI 239

deep neural networks (DNNs) 101

Deep Q-Networks (DQNs) 151

- for playing Atari game 172-178

deep RL 155

denoising autoencoder 115

dependent variables 51

depthwise separable convolutions 290

deterministic methods 124

DeviceMesh class 219

directed acyclic graph (DAG) 298

discriminative network 195

DistilBERT 323, 324

distributed data parallel (DDP) 216

distributed training 214

- implementation 224-227
- important, for IoT 215
- key components 214
- large models, optimizing with ModelParallel and LayoutMap 228-231

Diversity in Object Recognition (DIOR) 289

domain adaptation 199

Double DQN 178

drug discovery 199

dueling DQN 178-180

dynamic inference pipelines 287

E

early stopping feature 393

Ecobee 356

edge AI

- for IoT 256
- used, for precision farming 398-405

edge-cloud hierarchy 369

edge computing 238, 342

- benefits 342
- data processing 43, 44
- data storage 43, 44

Edge IoT 11

- advantages 11
- key processes 11

EfficientNet 287, 292

- dimensions 292
- You Only Look Once (YOLO) 293

EfficientNet-B0 292

elastic scalability 237

electrical load forecasting 389

- STLF, using LSTM 389-398

electrocardiogram (ECG) 340

Electronic Road Pricing (ERP) 410

energy management systems 239

ensemble learning 72

- bagging 74, 75
- pasting 74, 75
- voting classifier 73, 74

entity extraction 324

epsilon greedy algorithm 162

epsilon-greedy strategy 174

ESP8266 247, 359

evolutionary algorithms (EAs) 121, 144

- frameworks 135
- hyperparameter optimization 146, 147
- in AutoML 147, 148
- in AutoRL 148
- model architecture 145

exhaust vacuum (V) 51

exogenous inputs 270

- experience replay** 172
- explicit generative models** 186
- exploration-exploitation trade-off** 162
- Exponential Linear Unit (ELU)**
 - activation function 91, 92

F

- FairFace** 288
- FarmBeats** 399
- fashion design** 199
- feature extraction** 331
- federated learning** 288
- FedEx SenseAware** 10
- fitness functions** 122, 131
- floating-point operations per second (FLOPS)** 292
- fog computing** 342
- fog network** 4
- Fourier features** 270
- Fourier series** 276
- frame skipping** 287
- frameworks, evolutionary algorithms (EAs)**
 - DEAP 135
 - PyGAD 135
- Friedman MSE** 69
- Fronius** 359

G

GANs applications

- 3D object generation 200
- anomaly detection 199
- character and environment generation 199
- data augmentation 199
- domain adaptation 199
- drug discovery 199
- fashion design 199

- image generation 198
- image inpainting 198
- medical image synthesis 199
- semantic image synthesis 199
- style transfer 198
- super-resolution 198
- text-to-image synthesis 199
- virtual try-on 199
- voice generation and animation 199
- gated recurrent unit (GRU)** 112, 113
- Gaussian kernel** 62, 421
- Gazebo** 157
- General Additive Model (GAM)** 279
- Generalized Efficient Layer Aggregation Network (GELAN)** 294
- generative adversarial networks (GANs)** 185-196
 - AnimeGAN 197, 198
 - applications 198
 - cGANs 196
 - SRGAN 197
 - StyleGAN 197
 - variants 196
- Generative AI** 13
- generative models** 185
 - anomaly detection 186
 - data augmentation 186
 - explicit generative models 186
 - implicit generative models 186
 - predictive maintenance 186
 - realistic simulations 186
- generative network** 195
- genetic algorithms (GAs)** 121-131
 - advantages 134
 - crossover 132
 - disadvantages 134
 - evaluation 132
 - for CNN architecture 145, 146

guess the word program 136-140
 implementing 136
 mutation 133
 neural network weights, optimizing 140-144
 parent selection 131
 population initialization 131
 reproduction 132
 termination 132
gini impurity 69
global best (gbest) 128
global data centers 237
global programming model 222
GloVe 323
Google AutoML 250
 features 250
Google Compute Engine 236
Google Home 356, 359
Google Nest 356
Google Vertex AI 249
 features 249, 250
gradient descent algorithms 48, 93
gradient descent method 124, 125
 AdaGrad optimizer 125
 Adam optimizer 125
 RMSProp optimizer 125
 stochastic gradient descent (SGD) 125
graphics processing units (GPUs) 290
group 34
gRPC 42
 reference link 42
gym-gazebo 157
Gymnasium 157-161
 URL 161

H

h5py 34-36
 reference link 37
Hadoop Distributed File System (HDFS) 42
 DataNode 42
 hdfs3, using with 42
 NodeName 42
 PyArrow's filesystem interface, using for 43
haploid 130
HDF5
 download link 34
 using, with h5py 36, 37
 using, with pandas 35
 using, with PyTables 34
HDF5, in pandas
 reference link 36
HDF Group
 URL 34
healthcare sector 239
**Health Insurance Portability and
 Accountability Act (HIPAA)** 240
Hierarchical Data Format (HDF) 34
Home Assistant 357
HRNet 106
human-centric lighting 357
hybrid clouds 238
 for IoT 256
Hyperband 147
hyperbolic tangent activation function 88
hyperparameter optimization
 with EAs 146
hyperparameters 78
hyperparameter tuning
 with Keras 3 116-120

I

- IBM Watson IoT Platform** 16, 237
- identically distributed (iid)** 64
- image generation** 198
- image inpainting** 198
- ImageNet-21k** 304
- ImageNet 2012** 304
- implicit generative models** 186
- Inception-v1/v3** 106
- independent variables** 51
- Industrial Internet of Things (IIoT)** 367
- Industry 4.0** 370
- Industry IoT (IIoT)** 6
- Infrastructure as a Service (IaaS)** 233
 - features 235, 236
- integrated smart home implementation**
 - case study** 359
 - access control and security 360
 - energy and power monitoring 359
 - irrigation system 360
 - lighting control 359
 - system architecture and communication 360
 - temperature control and A/C management 359
- intent recognition** 323
- International Telecommunication Unit (ITU)** 4
- Internet of Things Global Standards Initiative**
 - reference link 4
- Internet of Things (IoT)** 3-5, 21, 129, 185, 260
- IoT application deployment**
 - best practices 238
- IoT CGMs** 355

IoT cloud platforms

- privacy 252
- security 252

IoT data

- adjustments and considerations 269, 270
- breach case study 362, 363

IoT layers

- application layer 7
- device layer 6
- management layer 7
- network layer 7
- security layer 7
- service layer 7

IoT platforms 7

- criteria 7, 8

IoT reference model 6**IoT scenarios**

- non-visual data 312

IoT Security Foundation (IoTSF) 363**IoT verticals** 8

- connected healthcare 8
- smart agriculture 8
- smart buildings 8
- smart city 8
- smart retail 8
- supply chain/logistics 9

J**JavaScript Object Notation (JSON)** 32**JAX** 225**JSON files**

- using, with JSON module 32, 33
- using, with pandas module 33

K

Keras

- cGANs 200-207
- reference link 200

Keras 3 17, 76, 215

- distributed learning 215
- distributed training strategies 216
- features 18
- for hyperparameter tuning 116-120
- multi-backend support 215, 216

Keras 3 distribution API 217

- automatic device detection 223
- DataParallel class usage 223
- data parallelism 218, 219
- DeviceMesh 219
- features 217
- global programming model 222, 223
- integration, with backend-specific features 224
- LayoutMap 221, 222
- ModelParallel class usage 223
- model parallelism 219
- multi-backend support 218
- single program, multiple data (SPMD) 223
- TensorLayout 220, 221

Keras Applications library 107

KerasNLP 207

- fine-tuning model 209
- overview 207
- reference link 209
- using 208

Keras Tuner 117

kernel 62

Kernel Density Estimation (KDE) 419

K-means clustering 48

L

Land Transport Authority (LTA) 410

large language models

- (LLMs) 3, 209, 248, 330

Lasso regularization 77

LayoutMap class 221

Leaky ReLU activation function 90

lemmatization 314

LeNet-5 105

linear regression 48

- deep learning, using 50
- electrical power output prediction 50-54
- prediction 49, 50

local and global dependencies 303

logarithm vector 190

logistic regression 54

- cross-entropy loss function 56
- for classification 54, 55
- wine, classifying with logistic regressor 56-59

Long Short-Term Memory (LSTM) 111, 259, 268, 269, 302, 324, 376

- forget gate 111
- input gate 111
- output gate 111
- used, for predicting CO levels 270-275
- used, for STFL 389-398

long-term load forecasting (LTLF) 389

loss function 93, 122

M

- machine learning (ML) 12, 13, 122, 185, 233, 340, 367

- for IoT 46

machine-to-human (M2H) 370

- machine-to-machine (M2M)** 370
- Markov Decision Process (MDP)** 155
- Maximum A Posteriori (MAP) term** 65
- max pooling** 104
- mean absolute error (MAE)** 69, 353, 384
- mean squared error (MSE)** 383
- mean vector** 190
- MediaPipe** 107, 286
 - features 299
 - for pose estimation 299-301
- medical image synthesis** 199
- medical IoT wearables** 361
- medium-term load forecasting (MTLF)** 389
- meiosis** 130
- Mel-Frequency Cepstral Coefficients (MFCCs)** 331
- Message Queuing Telemetry Transport (MQTT)** 247, 313
- microcontroller unit (MCU)** 254
- Microsoft Azure Functions** 236
- Microsoft Azure's AI Services** 248
- Microsoft FarmBeats** 404
- Microsoft IoT-Azure IoT suite** 16
- min-max normalization** 76
- MirroredStrategy** 215
- mitosis** 129
- ML APIs** 240
 - Amazon Rekognition 240-243
- ML pipeline**
 - example 314-316
 - outlining 314
- MobileNet** 287, 290
 - depthwise convolution 290
 - pointwise convolution 290
- model improvement, tips and tricks** 76
 - cross-validation 77
 - feature scaling 76

- grid search 78
- hyperparameter tuning 78
- No Free Lunch theorem 78
- overfitting 76, 77
- modeling algorithms** 48
- modeling approaches, predictive maintenance**
 - failure classification 373
 - RUL regression 373
- ModelParallel class** 219, 224
- model pruning** 269
- model quantization** 287
- MongoClient object** 41
- MPPSolar** 359
- MQTT protocol** 247
- MuJoCo** 158
- Multilayer Perceptrons (MLPs)**
 - as universal function approximators 97
 - empirical proof 98
 - for regression and classification 94-97
 - non-linear functions, modeling 98-100
- multiple linear regression** 49
- multi-point crossover** 133
- Multiscale Vision Transformer (MViT)** 309
- MultiWorkerMirroredStrategy** 215
- mutation** 131-133
 - bit-flip mutation 133
 - random resetting 133
- MySQL** 40

N

- Naive Bayes** 64
 - for wine quality 65, 66
- National Industrial Corridor Development Programme (NICDP)** 425
- natural language generation (NLG)** 311

natural language processing
 (NLP) 187, 303, 358
natural language understanding (NLU) 311
natural optimization methods 127
 genetic algorithms 128, 129
 particle swarm optimization (PSO) 128
 simulated annealing 127
NBEATS 283
neural network (NN) 369
 architectures 342
Neural Networks API (NNAPI) 298
NeuroKit 354
Newton-Raphson method 126
NLU pipeline
 implementing, with transformer 324-330
nodeName 42
No Free Lunch theorem 78
noise reduction 331
non-maximum suppression (NMS) 294
non-probabilistic algorithms 47
normalization 314, 331
 min-max normalization 76
 z-score normalization 76
Not Only Structured Query Language
 (NoSQL) database 41
NumPy module
 CSV files, reading 29, 30

O

objective function 52
objects 37
object storage 37, 38
 features 38
offspring 130
one-hot encoding 317, 323

one-hot vector 167
one-point crossover 132
online learning 269
ONNX 283
openHAB 357
OpenPyXL
 using, for XLSX files 30, 31
OpenVINO 116
operational expenditures (OpEx) 371
optimization 122, 123
optimization algorithms 48
original equipment manufacturer
 (OEM) 372
Overall Equipment Effectiveness (OEE) 262

P

pandas module
 CSV files, reading 28
 HDF5, using with 35
 JSON files, using 33
 using, with XLSX files 31
Paradox MG5050 360
ParameterServerStrategy 215
particle 128
particle best (pbest) 128
particle swarm optimization (PSO) 128
perceptual loss function 197
personalized automation routines 358
personally identifiable
 information (PII) 253
photoplethysmography (PPG) signal 354
PhysioNet 354
Pilot Parliaments Benchmark (PPB) 289
Platform as a Service (PaaS) 233, 236
Plug and Play (PnP) 357

policy gradients 155, 180

- actor-critic algorithm 182, 183
- benefits 181
- DDPG, for continuous action space 181, 182

precision farming

- decision-making (control) 399
- forecasting 399
- with edge AI 398-405

predictive analytics 262**predictive analytics, in wearables**

- activity recognition 341
- anomaly detection 341
- fall and accident detection 341
- health event prediction 341

**predictive heating/cooling
(thermostat AI) 357****predictive maintenance 186, 368**

- advantages and disadvantages 388
- challenges 388
- condition monitoring parameters/
sensor values 370, 371
- for decision-making and actuation 387
- hands-on examples 372
- RUL, predicting with LSTM 376-387
- traditional ML, for failure prediction 373-376
- with AI 369

pretrained models, in IoT applications 247

- benefits 247

pretrained models, time-series**forecasting 283**

- prophet 276, 277-283

pre-trained TinyBERT

- using, in sentiment analysis 208

preventive maintenance 370

- real-world examples 371, 372

privacy concerns, in personal IoT 361

- IoT data breach case study 362
- smart camera breach case study 361
- smart thermostat data leak case study 362
- Strava military base exposure case study 362

private clouds 237**probabilistic algorithms 47****programmable gradient****information (PGI) 294****Prophet 263, 276-281**

- logistic growth trend 276
- piecewise linear trend 276

protected health information (PHI) 240**protobufs 42****public clouds 237****PyArrow 42**

- filesystem interface, for HDFS 43

PyGAD 135**PyTables 34**

- HDF5, using with 34

Python

- TXT files, using 22, 23
- wearable data, analyzing with 343-354

PyTorch 225**pywebhdfs 42**

Q

Q-function 154**Q-learning 155, 161, 162**

- used, for taxi drop-off 163-166

Q-network 166

- Double DQN 178
- DQN, for playing Atari game 172-178
- dueling DQN 178-180
- used, for taxi drop-off 167-172

qToggle app 359**quantization 269**

quantum computing

in IoT 256, 257

quasi-Newton methods 126

R

R² score 395

radial basis function 62

radio-frequency identification (RFID) 369

Random Forest 373

random number generator (RNG) states 226

random resetting 133

Random Search strategy 118

Raspberry Pi 247, 357

Raspberry Pi module 360

raw data 37

reactive maintenance 369

realistic simulations 186

real-time monitoring 262

receiver operating characteristic (ROC) 375

reconstruction loss 188

Rectified Linear Unit (ReLU)

activation function 89

recurrent neural networks (RNNs) 109,
110, 259, 268, 302, 324, 389

gated recurrent unit (GRU) 112, 113

long short-term memory 111, 112

review 268

regular expression (regex) 221

regularization 77

Lasso regularization 77

ridge regularization 77

regularization term 188

regularized evolution 146

reinforcement learning (RL)

146, 151, 152, 387

applications 155, 156

example scenarios 153, 154

terminology 153, 154

Relative Humidity (RH) 19

reliability-centered maintenance (RCM) 371

remaining useful life (RUL) 368

predicting, with LSTM 376-387

Remote Procedure Calls (RPCs) 42

replay buffer 171

residual network (ResNet) 197

Resilient Distributed Dataset (RDD) 263

ResNet50

for image classification 107, 108

resolution scaling 287

return on investment (ROI) 372

RFID journal, IoT

reference link 3, 4

ridge regularization 77

RL algorithms

policy-based methods 155

value-based methods 155

RL in Artificial Intelligence (AI) 152, 153

RMSProp optimizer 125

Robot Operating System (ROS) 157

role-based access control (RBAC) 253

root mean squared error (RMSE) 353

S

Safe City system 412

SageMaker JumpStart 248

same padding 103

Samsung SmartThings 356

scikit-learn naive Bayes module 65

Seasonal-Trend decomposition

using Loess (STL) 261

second-order Taylor series expansion 126

security threat detection 357

Self-Driving Car (SDC) 153

semantic image synthesis 199

- sentiment analysis** 324
 - pre-trained TinyBERT, using 208
- separator** 28
- sexual reproduction** 130
- short-term load forecasting (STLF)**
 - with LSTM 389-398
- sigmoid activation function** 88
- silence removal** 331
- simple linear regression** 49
- simulated annealing** 127
- simulated environments** 157
 - Blender 157
 - Gazebo 157
 - Gymnasium 157
 - Unity ML-Agents SDK 157
- single program, multiple data (SPMD) expansion** 223
- Skynet** 412
- smart camera breach case study** 361
- smart cities** 239, 407
 - AI applications 409
 - IoT, adapting 423-425
 - need for 407, 408
- smart city IoT architecture** 424, 425
- smart home devices** 356
 - AI-driven home automation and intelligence 357, 358
 - case study 359
- smart IoT projects**
 - do-it-yourself (DIY) project ideas 363, 364
- smart lighting** 247
- smart manufacturing** 239
- smart point-of-sale (POS) IoT device** 245
- smart thermostat data leak case study** 362
- smart waste management** 411
- smartwatch health monitoring** 355
- snakebite** 42
- softmax activation function** 90
- Software as Service (SaaS)** 233, 236
- SONOFF Touch switches** 359
- SparkML** 263
 - used, for working with large time-series data 263-268
- spatial-temporal attention** 305
- speech data** 312
- SQLite** 39, 40
 - URL 39
- squeeze-and-excitation (SE) blocks** 292
- standardization** 76
- stateful LSTMs** 269
- statsmodels** 263
- stemming** 314
- stochastic gradient descent (SGD)** 125
- stopwords removal** 314
- Strava military base exposure case study** 362
- Structured Query Language (SQL)** 39
- structured SQL data**
 - working with 38
- style-based GAN (StyleGAN)** 186, 197
 - reference link 197
- StyleGAN2**
 - reference link 197
- style transfer** 198
- super-resolution** 198
- super-resolution generative adversarial networks (SRGANs)** 187, 197
- supervised learning** 48
- supervised learning algorithms** 48
- support vector machines (SVMs)** 47
 - using, for classification 59, 60
- survival of the fittest principle** 129
- SVC classifier** 62

T

temporal convolutional networks
(TCNs) 283, 302

Temporal Difference (TD) error 172

Temporal Fusion Transformer (TFT) 260

TensorFlow 16, 225

TensorFlow 2.0 17

TensorFlow Extended (TFX) 250

TensorFlow Hub 247

TensorFlow Lite 283

TensorFlow's Dataset API (TFDS) 226

TensorLayout class 220, 221

text data 312

text data processing 313

text encoding 317, 318
example implementation 318-323

text-to-image synthesis 199

text-to-speech (TTS) 332

time-series analysis
handling, challenges 262
importance 261
LSTMs 268
predictive analytics 262
real-time monitoring 262
RNN 268

time-series data 260
features 260, 261
handling, challenges 263
multivariate 260
real-time monitoring 262
SparkML, used for working with 263-268
univariate 260
use cases 261

time-series forecasting
pretrained models 276

TimeSformer 306

activity recognition 308
anomaly detection 308
spatial attention 306
temporal attention 306
using 306-308
video summarization 308

Time-Space Transformer
(TimeSformer) 287

Tiny Machine Learning (TinyML) 116

tokenization 314

TokenLearner 309

topic modeling 324

tournament selection method 131

Toy text 158

TPUStrategy 215

transformer
NLU pipeline, implementing 324-330

transformers 283, 324

Transport Layer Security (TLS)
protocols 252

two-factor authentication (2FA) 361

TXT files 22
using, in Python 22, 23

U

Ultralytics 286

uncertainty intervals 277, 283

U--Net 106

United Nations Sustainable Development
Goals (SDGs) 425

Unity ML-Agents SDK 157

universal approximation theorem 97

unsupervised learning 48

unsupervised learning algorithms 48

V

- Vacuum (V)** 19
- valid padding** 103
- variational autoencoders**
 - (VAEs) 115, 185-189
 - anomaly detection 189-194
- vehicle-to-infrastructure (V2I)** 410
- VGG-16 architecture** 106
- Video Swin Transformer (VideoSwin)** 308
- Video Vision Transformer (ViViT)** 309
- virtual try-on** 199
- vision transformers (ViTs)** 303
 - working 303-305
- Visual Data in IoT**
 - AI Techniques, leveraging 289
 - attention mechanisms 303
 - challenges, handling 286-289
 - overview 286
- voice and gesture recognition** 358
- voice generation and animation** 199
- voting classifier** 73, 74

W

- Waveform Database (WFDB) toolkit** 354
- Wearable IoT devices** 340
 - AI integration 340, 341
 - data, analyzing with Python 343-354
 - edge computing 342
- weights** 105
- wine quality dataset** 19, 20
 - download link 19
 - reference link 56
- Word2Vec** 323
- word embeddings** 318, 323

X

- Xception** 106
- XGBoost** 373
- XLSX files**
 - OpenPyXL, using for 30, 31
 - pandas module, using 31

Y

- YOLO-Nano** 287
- YOLO versions**
 - YOLOv3 294
 - YOLOv4 294
 - YOLOv5 294
 - YOLOv8 294
 - YOLOv9 294
 - YOLOv10 294
 - YOLOv11 295
- You Only Look Once (YOLO)** 248, 293
 - applications, beyond object detection 297
 - bounding box prediction 293
 - class prediction 294
 - grid division 293
 - in action 295-297
 - key stages 293, 294
 - scalability 294
 - use cases 298

Z

- z-score normalization** 76
- zygote** 130



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

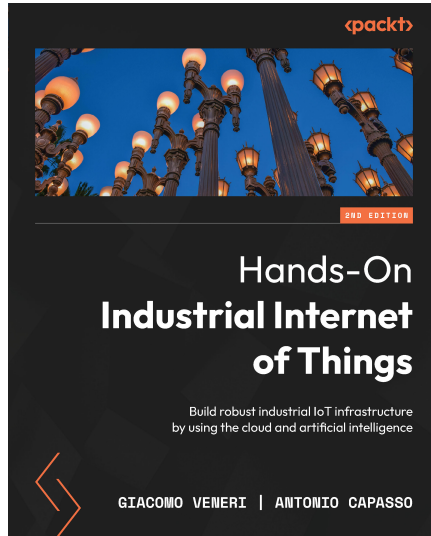
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

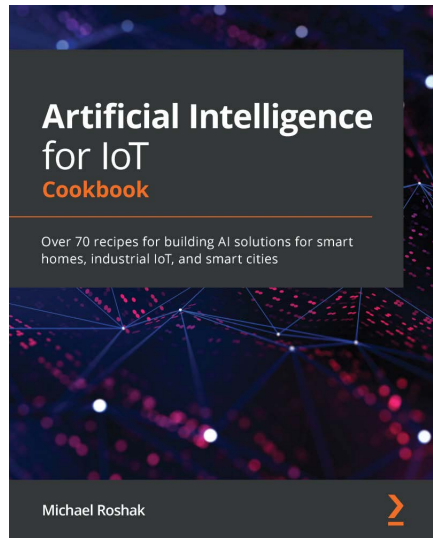


Hands-On Industrial Internet of Things

Giacomo Veneri, Antonio Capasso

ISBN: 978-1-83588-746-2

- Get a solid understanding of industrial processes, devices, and protocols
- Harness IoT technology to effectively manage industrial use cases
- Design and implement an IIoT network flow to continuously monitor the performance of your critical assets
- Get to grips with popular cloud-based platforms such as AWS and Azure
- Explore Edge devices and learn about Edge and fog computing to gather field data
- Apply diagnostic analytics to real-world data to answer critical workforce questions
- Develop AIoT technology for predictive maintenance



Artificial Intelligence for IoT Cookbook

Michael Roshak

ISBN: 978-1-83898-198-3

- Explore various AI techniques to build smart IoT solutions from scratch
- Use machine learning and deep learning techniques to build smart voice recognition and facial detection systems
- Gain insights into IoT data using algorithms and implement them in projects
- Perform anomaly detection for time series data and other types of IoT data
- Implement embedded systems learning techniques for machine learning on small devices
- Apply pre-trained machine learning models to an edge device
- Deploy machine learning models to web apps and mobile using TensorFlow.js and Java

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Hands-On Artificial Intelligence for IoT*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/978-1-83546-718-3>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly