‹packt›

Quantum Computing for AI with Python

Build quantum circuits with Qiskit and explore QML, quantum kernels, QNNs, and hybrid optimization



Vivian Aranha

Quantum Computing for AI with Python

Build quantum circuits with Qiskit and explore QML, quantum kernels, QNNs, and hybrid optimization.

Vivian Aranha

‹packt›

Quantum Computing for AI with Python

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: Deepesh Patel

Project Manager: Yash Basil

Content Engineer: Mamta Wardhani

Technical Editor: Simanta Rajbangshi

Indexer: Manju Arasan

Production Designer: Ajay Patule

Growth Lead: Ankur Mulasi

First published: November 2026

Production reference: 1291026

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-80808-833-9

www.packtpub.com

Contributor

About the author

Vivian Aranha is an AI educator, technology leader, and founder of School of AI, with over 20 years of industry experience. He earned a Bachelor's degree in Information Technology in 2004 and a Master's degree in Computer Science in 2006. His career spans web technologies, mobile app development for iOS and Android, blockchain solutions, and AI systems and applications.

Vivian has worked with Fortune 500 organizations, including The Washington Post, Delta Air Lines, and IBM. An instructor since 2009, he has trained professionals worldwide and now teaches AI globally. His courses have attracted over 2.5 million enrollments, with more than 500,000 students learning through School of AI, Udemy, Skool, and Maven.

Through *Quantum Computing for AI with Python*, Vivian aims to help readers understand how artificial intelligence and quantum computing work, how they fit together, and how to build and run machine learning models, neural networks, and quantum circuits with Python and Qiskit.

Table of Contents

Preface

Free benefits with your book

Chapter 1: Introduction to Artificial Intelligence and Quantum Computing

Technical requirements

Understanding the fundamentals of artificial intelligence

Understanding quantum computing

Preparing tools and libraries for AI and quantum computing development

- Choosing the tech stack •
- Installing Python and Jupyter Notebook •
- Installing essential AI libraries and frameworks •
- Installing and verifying Qiskit for quantum computing •

Running hands-on demonstrations with a linear regression model and a quantum circuit

- Building and testing a simple AI model using linear regression •
- Creating and simulating a basic quantum circuit in Qiskit •

Exploring foundational machine learning and deep learning concepts

Comparing supervised and unsupervised learning approaches

Building a supervised learning model for house price prediction

Summary

Chapter 2: Building Models with Machine Learning and AI Frameworks

Technical requirements

Exploring foundational Python libraries for machine learning

Using NumPy for numerical computing and array operations •

Creating a NumPy array •

Reshaping an array •

Applying operations on arrays •

Generating a random matrix •

Using Pandas for data manipulation and analysis •

Creating a Pandas DataFrame from a Python dictionary •

Reading data from a CSV file into a Pandas DataFrame •

Displaying the first rows of a DataFrame •

Generating descriptive statistics for numerical columns •

Handling missing values in a DataFrame •

Filtering rows in a DataFrame based on column values •

Applying scikit-learn for machine learning workflows •

Importing essential scikit-learn modules •

Creating a sample dataset for classification •

Splitting the data into training and testing sets •

Training a logistic regression model on the prepared data •

Making predictions and evaluating model accuracy •

Interpreting accuracy results by modifying input data •

Introducing feature engineering and data preprocessing

Implementing data preprocessing with Pandas and scikit-learn •

Handling missing values •

Encoding categorical values •

Scaling the numerical features •

Concatenating the final processed data •

Building and evaluating a classification model with scikit-learn
Importing the necessary libraries •
Creating a sample dataset •
Splitting, scaling, and training with the KNN classifier •
Making predictions on data and evaluating the model •
Summary

Chapter 3: Training Deep Learning and Reinforcement Learning Models

Technical requirements
Exploring neural networks and backpropagation in deep learning
Understanding convolutional neural networks for image processing
Applying neural networks to natural language processing with RNNs and transformers
Building an image classifier using CNNs
Using TensorFlow to create the image classifier •
Installing and importing required libraries in TensorFlow •
Loading the CIFAR-10 dataset for image classification •
Normalizing image data •
Defining a CNN architecture using Keras Sequential API •
Compiling and training the CNN •
Using PyTorch to create the image classifier •
Installing and importing the necessary libraries •
Loading the MNIST dataset •
Creating the data loaders •
Defining a CNN architecture for digit classification in PyTorch •
Initializing the CNN model, loss function, and optimizer in PyTorch •

Training the CNN model on MNIST data over multiple epochs in PyTorch •

Evaluating the trained CNN model and interpreting accuracy •

Training reinforcement learning with Q-learning and deep Q-networks

Understanding how Q-learning and deep Q-networks learn

Implementing reinforcement learning agents in Gym

Installing dependencies •

Creating the FrozenLake environment •

Training the agent with Q-learning •

Training the agent with a deep Q-network •

Summary

Chapter 4: Understanding the Foundations of Quantum Computing

Technical requirements

Understanding quantum computing through analogies

Comparing bits and qubits •

Explaining superposition •

Visualizing entanglement •

Exploring quantum parallelism •

Describing quantum measurement •

Running your first quantum experiment in Qiskit

Getting started with Qiskit •

Building a one-qubit circuit •

Simulating the experiment with shots •

Visualizing the flavor counts •

Describing qubits, ket notation, and quantum gates

Contrasting classical and quantum information •

Writing quantum states in ket notation •

Defining a qubit and its state •
Manipulating qubits with quantum gates •
Exploring probability, wave functions, and the Bloch sphere
Understanding probability and wave functions •
Representing a qubit on the Bloch sphere •
Measuring a quantum state •
Verifying quantum measurement in Python
Repeating the measurement with more shots •
Inspecting the quantum state before measurement •
Summary

Chapter 5: Building Quantum Circuits and Algorithms

Technical requirements
Exploring fundamental quantum gates and their operations
Understanding Pauli gates: X, Y, and Z •
Explaining the Hadamard gate and superposition •
Describing the CNOT gate and quantum entanglement •
Composing quantum circuits from gates
Implementing basic quantum circuits in Qiskit
Creating and simulating a basic quantum circuit •
Applying gates and measuring qubits in Qiskit •
Extracting and interpreting measurement results •
Implementing a Bell state circuit •
Exploring foundational quantum algorithms
Explaining Grover's search algorithm and quantum speedup •
Describing Shor's factoring algorithm and its impact •
Implementing Grover's algorithm with Qiskit

Summary

Chapter 6: Applying Quantum Computing to AI

Technical requirements

Exploring how quantum computing enhances AI capabilities

- Comparing classical and quantum machine learning approaches •
- Identifying key quantum machine learning techniques •
- Examining variational quantum classifiers •

Implementing a VQC model in Qiskit

Processing data with quantum feature maps and kernel methods

- Introducing feature mapping •
- Quantum kernel methods •

Training a quantum kernel-based classifier

Advancing AI with hybrid quantum neural networks and quantum Boltzmann machines

- Hybrid quantum-classical neural networks (HQCNNs) •
- Quantum Boltzmann machines (QBMs) •

Building a hybrid quantum-classical AI model

Summary

Chapter 7: Exploring AI-Quantum Use Cases and Optimization

Technical requirements

Understanding AI-quantum applications across different fields

- Explaining AI-assisted quantum chemistry simulations •
- Applying AI and quantum computing to financial modeling •
- Describing quantum-enabled drug discovery workflows •

Simulating a quantum chemistry problem with Qiskit

Importing libraries and defining the H_2 geometry •
Building the qubit Hamiltonian and configuring VQE •
Estimating H_2 ground-state energy with VQE •
Exploring quantum annealing for combinatorial optimization
Explaining quantum annealing and D-Wave QUBO hardware •
Combining AI predictions with quantum portfolio allocation •
Implementing AI-QC optimization for portfolio management
Importing the necessary libraries •
Preparing synthetic stock data and defining portfolio constraints •
Formulating the portfolio problem and configuring sampling VQE •
Solving the portfolio problem and interpreting the selected assets •
Summary

Chapter 8: Unlock the Code Bundle and the PDF Version

Unlock this book's free benefits in three easy steps

Other Books You May Enjoy

Index

Preface

Artificial intelligence and quantum computing are two of the most talked-about technologies of our time, and they are increasingly discussed together. AI is already part of everyday life, from product recommendations to medical imaging, but training and tuning models demand ever more computing power. Quantum computing takes a different approach to computation, using qubits, superposition, and entanglement, and researchers are exploring where it could help with problems in areas such as chemistry, finance, and optimization. Python is the common thread: it is the language behind the main AI libraries and behind Qiskit, which lets you build and run quantum programs on your own computer.

This book is a hands-on introduction to both fields and to the ground where they meet. You will start with the core ideas of AI and machine learning and build working models with scikit-learn, TensorFlow, and PyTorch. You will then learn the foundations of quantum computing, build quantum circuits and run algorithms such as Grover's search in Qiskit, and see how quantum methods can be applied to machine learning and to problems such as molecular simulation and portfolio optimization. Every chapter combines explanation with code you can run, so each concept is something you see working and not only read about.

Throughout the book, we stay honest about what these technologies can do today. The quantum examples run on simulators, and the datasets are small and often synthetic, so the results show how the methods work and not that quantum computers already outperform classical ones. By the end, you will be able to read and build simple AI and quantum programs, understand where the two fit together, and judge claims about quantum AI with a clear head.

Who this book is for

This book is designed for:

- Python developers who are curious about AI and quantum computing
- Data scientists and machine learning practitioners who want to explore quantum machine learning
- Students and self-taught learners looking for a first, practical introduction to both fields
- Software engineers and technology professionals who want to understand where AI and quantum computing are heading
- Professionals in areas such as finance, chemistry, and pharmaceuticals who want to see how these methods apply to their domain

To benefit from this book, you should have basic familiarity with Python and programming concepts such as variables, loops, and functions. No prior knowledge of AI, machine learning, or quantum computing is required, because the book introduces each idea as it is needed.

What this book covers

Chapter 1, *Introduction to Artificial Intelligence and Quantum Computing*, introduces the core ideas behind both fields and how they connect. You will set up Python, Jupyter, and the main libraries, including scikit-learn, TensorFlow, PyTorch, and Qiskit. You will then run your first two hands-on projects: a linear regression model that predicts house prices and a one-qubit quantum circuit simulated in Qiskit. The chapter also introduces machine learning and deep learning, and supervised and unsupervised learning.

Chapter 2, *Building Models with Machine Learning and AI Frameworks*, focuses on the data side of machine learning. You will use NumPy for numerical computing and Pandas for data manipulation, then prepare data through feature engineering: handling missing values, encoding categories, and scaling features. The chapter ends with a hands-on project that builds and evaluates a classification model with scikit-learn and interprets its results using accuracy and a confusion matrix.

Chapter 3, *Training Deep Learning and Reinforcement Learning Models*, moves from classical machine learning to neural networks. You will learn how neural networks, convolutional networks, recurrent networks, and transformers work, and then train image classifiers with TensorFlow and PyTorch on the CIFAR-10 and MNIST datasets. The second half introduces reinforcement learning, where you train agents with Q-learning and deep Q-networks using experience replay.

Chapter 4, *Understanding the Foundations of Quantum Computing*, builds the intuition and vocabulary you need for the quantum chapters. Using everyday analogies, it explains qubits, superposition, entanglement, and measurement, and then runs a first quantum experiment in Qiskit. You will learn ket notation, probability amplitudes, the wave function, and the Bloch sphere, and verify how measurement behaves using histograms and statevectors in Python.

Chapter 5, *Building Quantum Circuits and Algorithms*, shows how to control qubits. You will study the Pauli, Hadamard, and CNOT gates, compose them into circuits, and build and simulate a Bell state in Qiskit. The chapter then

introduces Grover's search algorithm and Shor's factoring algorithm, and you implement Grover's algorithm on a two-qubit search problem.

Chapter 6, *Applying Quantum Computing to AI*, brings the two halves of the book together. You will compare classical and quantum machine learning, and then train a variational quantum classifier and a quantum kernel classifier in Qiskit. You will also build a hybrid quantum-classical neural network with Qiskit and PyTorch and are introduced to quantum Boltzmann machines.

Chapter 7, *Exploring AI-Quantum Use Cases and Optimization*, looks at where AI and quantum computing are being applied, including chemistry, finance, and drug discovery. You will estimate the ground-state energy of the hydrogen molecule using Qiskit Nature and the variational quantum eigensolver, learn how quantum annealing and D-Wave approach combinatorial optimization, and work through a portfolio selection exercise using synthetic stock data.

To get the most out of this book

To benefit fully from this book, follow these recommendations:

- Install the required libraries at the beginning of each chapter using the provided pip commands
- Work through the code examples interactively in Jupyter Notebook, JupyterLab, or Google Colab
- Modify and experiment with the examples, for instance by changing the number of shots in a quantum circuit, the dataset, or the number of training epochs, to see how the results change
- Expect your results to differ slightly from the ones shown, because quantum measurements are probabilistic and many models start from random values
- Download the code files from the GitHub repository and compare them with your own work
- Read the chapters in order, since later chapters build on earlier ones, and return to earlier chapters when you meet an unfamiliar concept

Download the example code files

This book includes a complete downloadable code bundle containing all the example projects and files used throughout the chapters. We recommend

downloading the bundle so you can follow along smoothly and experiment with the examples.

Use the bundle as a practical starting point. Modify it, extend it, and apply what you learn by creating your own variations as you progress through the chapters.

Get the code bundle

If you bought the book directly from Packt:

1. Go to `packtpub.com`
2. Click your **profile picture** and select **Your Orders**
3. Find this book and click **Download Code**

If you bought this book from Amazon or any other channel partner:

1. Go to `packtpub.com/unlock` or scan the following QR code:



2. Search for this book
3. Sign up or log in to your free Packt account
4. Upload your proof of purchase and download the code bundle locally

Usage note: You're free to use and modify this code for personal learning and non-commercial projects.

Download the color images

Your purchase includes a color, DRM-free PDF copy of this book, ideal for viewing color images, screenshots, and diagrams. Refer to the *Free benefits with your book section* at the end of the *Preface* to unlock your PDF copy.

Conventions used

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

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter

handles. For example: "Use the `qc.h(0)` command to apply a Hadamard gate to the first qubit."

A block of code is set as follows:

```
from qiskit import QuantumCircuit

qc = QuantumCircuit(2)  # Two-qubit circuit
qc.h(0)  # Apply Hadamard gate
qc.cx(0, 1)  # Apply CNOT gate
qc.measure_all()
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "A **quantum circuit** is a series of quantum gates applied in a particular sequence. To download Python, select Downloads on the official Python website."

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

```
pip install numpy pandas scikit-learn tensorflow torch torchvision opencv-python
```

> **NOTE**
>
> Warnings or important notes appear like this.

> **TIP**
>
> Tips and tricks 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 or have any general feedback, please email us at `customercare@packt.com` and mention the book's 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 reported this to us. Please visit `http://www.packt.com/submit-errata`, click **Submit 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 .

Share your thoughts

Once you've read *Quantum Computing for AI with Python*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.



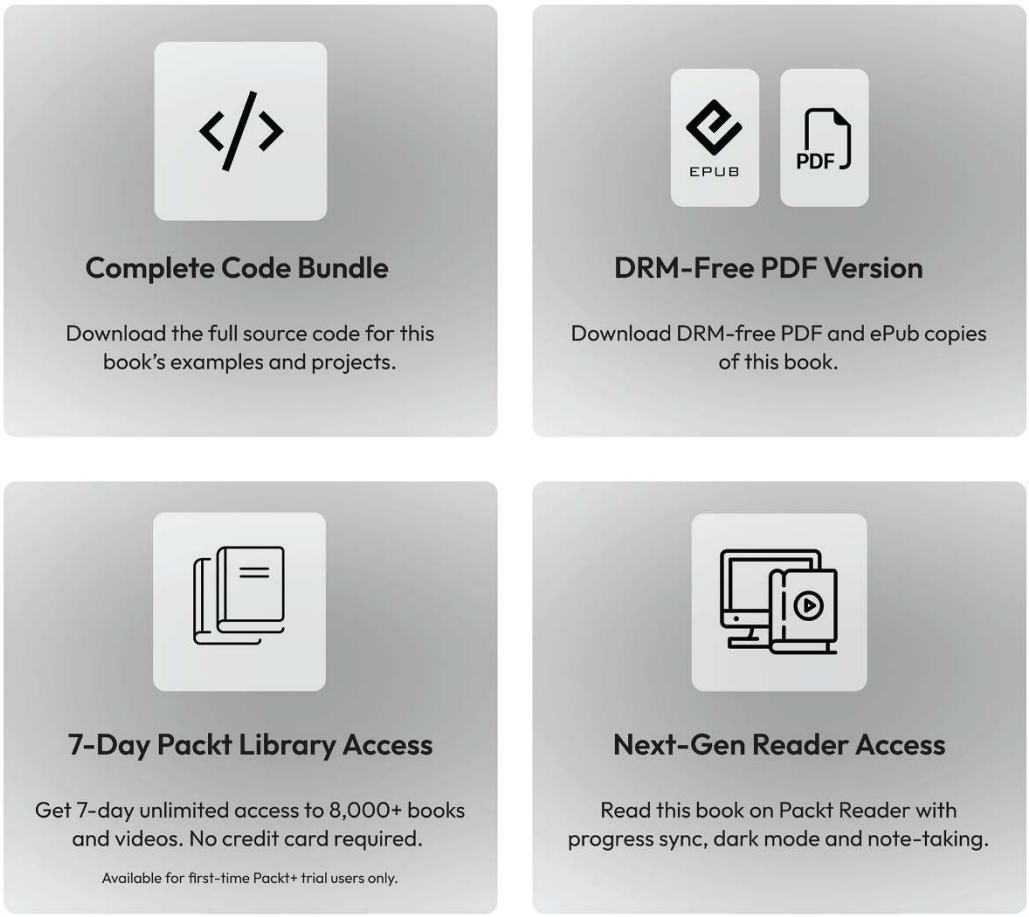
https://packt.link/r/1808088336

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

Free benefits with your book

This book includes free benefits designed to support your learning and help you apply what you learn effectively. Activate them now for instant access (see the *How to unlock* section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:



How to unlock

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

1

Introduction to Artificial Intelligence and Quantum Computing

Every so often, a technology changes what we expect computers to do. Artificial intelligence (AI) already lets machines learn from data, understand language, and make decisions. At the same time, quantum computing offers a fundamentally different way to process information, built to tackle problems that today's computers struggle with. This book brings the two fields together and shows you how to work with both using Python, so you can understand the ideas and build with them.

This chapter lays the groundwork for the rest of the book. You will learn what AI and quantum computing are and how they connect, set up your development environment with Python, Jupyter Notebook, Qiskit, and the main AI libraries, and run your first hands-on demonstrations: a linear regression model that predicts house prices and a simple quantum circuit. You will also learn the machine learning and deep learning foundations that later chapters build on. This chapter covers the following topics:

- Understanding the fundamentals of artificial intelligence
- Understanding quantum computing
- Preparing tools and libraries for AI and quantum computing development
- Running hands-on demonstrations with a linear regression model and a quantum circuit
- Exploring foundational machine learning and deep learning concepts

- Comparing supervised and unsupervised learning approaches
- Building a supervised learning model for house price prediction

Technical requirements

To follow the examples in this chapter, you will need the following:

- Python 3.12 or later, with `pip`
- A computer running Windows, macOS, or Linux
- JupyterLab to run notebooks
- The libraries `numpy`, `pandas`, `scikit-learn`, `tensorflow`, `torch`, `torchvision`, `opencv-python`, `matplotlib`, `qiskit`, and `qiskit-aer`

Creating a virtual environment first is recommended, as shown later in this chapter. Install all the required libraries with the following command:

```
pip install jupyterlab notebook numpy pandas scikit-learn tensorflow torch torchvision opencv-python
matplotlib qiskit qiskit-aer
```

The code files for this chapter are available at https://github.com/PacktPublishing/Quantum-Computing-for-AI-with-Python/blob/main/Chapter1.ipynb

NOTE

Download the code bundle and the PDF version of this book

Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Understanding the fundamentals of artificial intelligence

Artificial intelligence is the simulation of human intelligence in machines that can think, learn, and make decisions. AI enables computers to process data, recognize patterns, and automate complex tasks without human intervention.

Some of the key concepts of AI include:

- **Machine learning** (**ML**) refers to AI systems that learn from data and improve over time without explicit programming.

- **Deep learning** (**DL**) is a type of advanced machine learning model inspired by the human brain, using neural networks to process large datasets.
- **Natural language processing** (**NLP**) is AI's ability to understand and generate human language, with examples such as chatbots and language translation.
- **Computer vision** (**CV**) involves AI systems that analyze images and videos, such as facial recognition and object detection.
- **Reinforcement learning** (**RL**) is a type of AI that learns through trial and error by interacting with an environment, with examples such as self-driving cars and game AI.

AI has real-world applications in healthcare, finance, retail and e-commerce, and self-driving cars. In healthcare, you find AI-powered diagnoses, drug discovery, and robotic surgeries. In finance, AI is used for fraud detection, stock market prediction, and risk assessment. Retail and e-commerce benefit from personalized recommendations and AI-driven customer service. For self-driving cars, AI enables autonomous navigation and traffic prediction. AI plays a role in every single aspect of your life.

AI gives machines the ability to learn from data and act on it. A second field approaches computing from a very different direction by changing how information itself is stored and processed, and that is the focus of the next section.

Understanding quantum computing

Quantum computing is a new computing approach based on quantum mechanics, which allows computers to process complex calculations exponentially faster than classical computers. Unlike traditional computers that use bits, which are 0 or 1, quantum computers use **qubits**, which can exist in superposition, both 0 and 1 simultaneously, enabling massive parallel processing.

Some of the key concepts of quantum computing include:

- **Qubits** are quantum bits, the fundamental unit of quantum information. They are similar to classical bits but offer enhanced computing power.

- **Superposition** means a qubit can be in both the 0 and 1 states at the same time, allowing quantum computers to process multiple possibilities simultaneously.
- **Entanglement** is a phenomenon where two qubits become interconnected, so the state of one affects the other, even across vast distances.
- **Quantum gates** are operations that manipulate qubits. Examples include the Hadamard gate or CNOT gate, and you will look at some of these gates in future chapters.
- **Quantum speedup** is the ability for quantum computers to solve problems much faster than classical computers. Examples include factoring large numbers or simulating molecules.

You can find real-world applications of quantum computing in cryptography and cybersecurity, financial modeling, drug discovery, material science, and also in AI and machine learning. For cryptography and cybersecurity, you have quantum-safe encryption to protect sensitive data. In financial modeling, quantum algorithms are used for risk analysis and portfolio optimization. For drug discovery and materials science, simulating molecular structures helps develop new drugs and materials. In AI and machine learning, you have quantum-enhanced neural networks and optimization algorithms.

Think of your machines that are there right now; they might be a little slower, but in the future, using quantum computing, they will be way faster, or they will be thinking just like humans do, and sometimes maybe even faster.

NOTE

How do AI and quantum computing intersect?

AI and quantum computing are separate fields, but when you combine them, you get quantum AI, which creates a synergy. AI models need huge amounts of computation, and quantum computers can accelerate this process.

Some of the key ways quantum computing enhances AI include **quantum machine learning** (**QML**), which uses quantum computers to train AI models much faster. Quantum optimization also improves AI decision-making and efficiency in large datasets. Another area is **quantum neural networks** (**QNNs**), where quantum capabilities will enhance future AI systems.

> For example, in quantum AI for drug discovery, AI identifies drug candidates while quantum simulations predict molecular behavior. In finance, AI-powered trading models use quantum algorithms for better risk assessment. You also have AI for quantum computing, where AI optimizes quantum circuits to improve quantum computer performance. These use cases are among the interesting applications that people are working on right now, and you can expect to see more of them in the next year or two.

Preparing tools and libraries for AI and quantum computing development

To get started with hands-on AI and quantum computing projects, you need to set up the right tools and libraries. You will install Python and Jupyter Notebook, add essential AI libraries and frameworks, and set up Qiskit for quantum computing.

Choosing the tech stack

You will use Python as your main programming language because it is widely used for AI and quantum computing due to its simplicity, readability, and the large number of available libraries. For AI and machine learning, you will need libraries such as **NumPy** and **Pandas** for numerical computing and data manipulation, **scikit-learn** for machine learning algorithms such as classification, regression, and clustering, **TensorFlow** and **PyTorch** for DL and neural networks, and **OpenCV** for AI-powered image processing and CV.

For quantum computing, you will use Qiskit by IBM Quantum, which is a framework for designing and running quantum circuits. You will also encounter PennyLane, a library for QML that integrates with PyTorch and TensorFlow, and D-Wave Ocean SDK for working with quantum annealers in optimization problems.

With these tools and libraries in mind, you are ready to begin setting up Python, Jupyter Notebook, Qiskit, and the main AI frameworks.

Installing Python and Jupyter Notebook

To begin, install Python and Jupyter Notebook so you can run your code and experiments. Follow these steps:

1. Go to the official Python website at https://www.python.org
2. Select the **Downloads** section
3. Download Python by selecting the download button for your operating system (Windows, Linux, Unix, or macOS). The website will usually suggest the latest version for your computer
4. After downloading, install Python by following the prompts for your OS

Once Python is installed, you can verify the installation by opening your terminal or command prompt and running the following command to check the Python version:

```
python -V
```

This will return the version of Python installed. For example:

```
Python 3.12.7
```

Depending on your system, you might need to use `python3 -v` instead. The latest version at the time of this writing is 3.13.2, but version 3.12.7 will also work for this book.

Next, you should use a virtual environment to avoid package conflicts and permission issues. You can create and activate a virtual environment with the following commands:

```
python -m venv aiqc-env
# On Windows:
aiqc-env\Scripts\activate
# On macOS/Linux:
source aiqc-env/bin/activate
```

Once your virtual environment is activated, install Jupyter Notebook by running the following command in your terminal or command prompt:

```
pip install jupyterlab
```

This command will install Jupyter Notebook and all its essentials. You will see output indicating the installation progress and any requirements that the installer already satisfies.

Once Jupyter Notebook is installed, you are ready to open and use it for your AI and quantum computing projects.

To create a new notebook in Jupyter Notebook, go to the **File** menu, select **New**, and then select **Notebook** as follows:

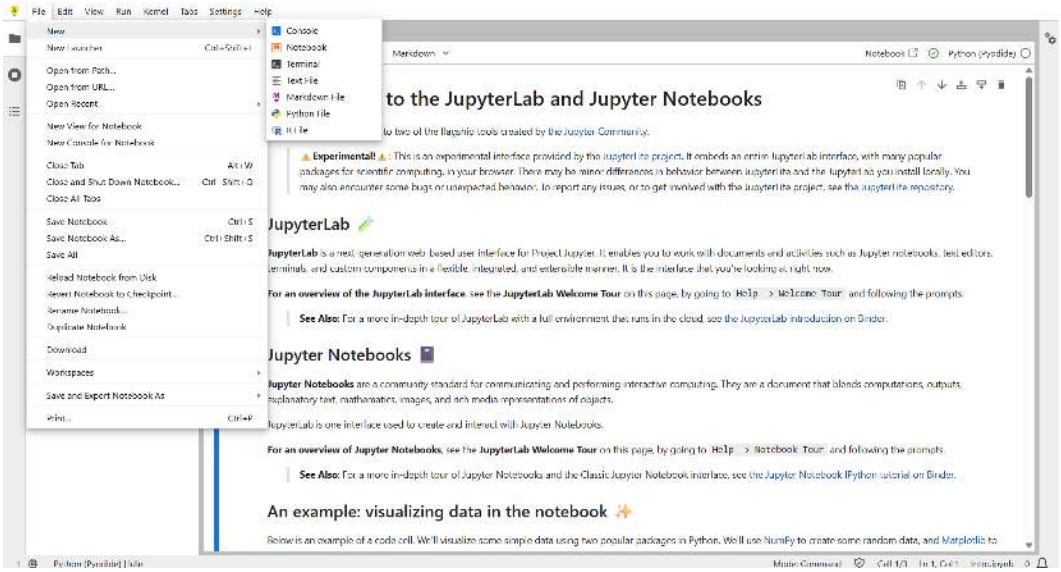


Figure 1.1: Jupyter Notebook interface with File menu and New Notebook option

You will use a notebook for most of the applications you build in this book.

When Jupyter Notebook is running, you might see various messages in the terminal. These messages are normal and indicate that your Jupyter Notebook server is active. Keep the terminal open, as it needs to run in the background while you work in your browser.

Installing essential AI libraries and frameworks

Next, you need to install some essential AI libraries. The libraries you will install include NumPy, Pandas, scikit-learn, TensorFlow, Torch, torchvision, and OpenCV-Python.

To install all these libraries at once, enter the following command in your terminal:

```
pip install numpy pandas scikit-learn tensorflow torch torchvision opencv-python matplotlib
```

NOTE

TensorFlow and PyTorch can have compatibility issues with certain Python versions or with each other. You should check the official installation guides for `TensorFlow` (`https://www.tensorflow.org/`) and

> PyTorch (https://docs.pytorch.org/docs/2.14/index.html) to ensure that you install versions compatible with your Python environment. If you encounter errors, refer to these guides for troubleshooting and version selection.

You can also install each library individually by running separate pip install commands for each one, but installing them together is more efficient. These AI libraries will be used for your applications, and if you need any additional ones later, you will learn how to install those as well.

Installing and verifying Qiskit for quantum computing

To install Qiskit for quantum computing, run the following command in your terminal:

```
pip install qiskit qiskit-aer
```

Qiskit supports most major operating systems, but some features might require additional system dependencies (such as C++ build tools or specific Python versions), and not all features are available on every platform. If you encounter issues or need platform-specific instructions, refer to the official Qiskit installation guide at https://qiskit.org/documentation/getting_started.html.

After installing Qiskit, you can verify the installation and check the version by running Python in your terminal and executing the following commands:

```
import qiskit
print(qiskit.__version__)
```

This will print the current version, for example:

```
1.4.0
```

You can also perform this verification in a Jupyter Notebook. Run the cell by selecting the play button in the notebook interface, and you will see the Qiskit version output, confirming that the installation was successful.

Running hands-on demonstrations with a linear regression model and a quantum circuit

The objectives of this exercise are to build and test a simple model using linear regression and then create and simulate a basic quantum circuit in Qiskit. These hands-on demonstrations will help you understand how to use Python libraries and Jupyter Notebook to work with both AI and quantum computing concepts.

Building and testing a simple AI model using linear regression

You will now create a linear regression model using the scikit-learn library to predict values based on input data.

To start, import the necessary libraries for linear regression and array operations:

```
import numpy as np
from sklearn.linear_model import LinearRegression
```

Next, define your sample data for the model:

```
# Sample Data
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
y = np.array([2, 4, 6, 8, 10])
```

In this example, `X` is reshaped into a column vector, which is required for machine learning models in scikit-learn. The values in `X` can be thought of as the number of hours studied (independent variable), and the corresponding values in `y` as the resulting test scores (dependent variable). So, `y` is more like the test score based on the number of hours you have studied. The more hours you put into your study, the higher your score will be.

To train the model, you first create a linear regression model and then fit it to your data. The following code creates and trains the model:

```
# Train Model
model = LinearRegression()
model.fit(X, y)
```

This code creates a linear regression model called `model`. The model will find the best straight-line equation for your data. When you call `model.fit(X, y)`, the model learns the relationship between `X` and `y`. In this case, it learns that `y` is two times `X`.

To make a prediction, you use the `predict` function. For example, to predict the test score for six hours of study, use the following code:

```
# Predict
print(model.predict([[6]]))
```

The output printed is as follows:

```
[12.]
```

If you change the input to 8, the model predicts the output as follows:

```
[16.]
```

This demonstrates how modeling and prediction work in artificial intelligence: you provide a set of values, the model learns from them, and it predicts the output for new inputs.

Creating and simulating a basic quantum circuit in Qiskit

You will now run a simple quantum circuit using Qiskit Aer, which is a simulator for quantum circuits.

To begin, import the necessary classes from Qiskit and Qiskit Aer. The following code imports the QuantumCircuit class, which lets you define quantum circuits, and Aer, which provides the simulator backend:

```
from qiskit import QuantumCircuit
from qiskit_aer import Aer
```

Next, create a quantum circuit with one qubit and one classical bit, and apply the Hadamard gate to the qubit. The Hadamard gate puts the qubit into a superposition state, meaning it is both 0 and 1 at the same time until measured:

```
# Create a Quantum Circuit with 1 Qubit and 1 Classical Bit
qc = QuantumCircuit(1, 1)
```

```
qc.h(0)  # Apply Hadamard gate
```

To record the result of measuring the qubit, add a measurement operation to the circuit. This step is important for getting the outcome counts after running the simulation:

```
# Add Measurement (Important for getting counts!)
qc.measure(0, 0)
```

So, you measure qubit zero and store the result in classical bit zero. Before measurement, the qubit is in superposition, and after measurement, it collapses to either a zero value or a one value. Measuring forces the quantum coin to land on either heads or tails.

To simulate the quantum circuit using the Aer simulator from Qiskit, select the simulator backend and run the circuit. The simulator mimics a real quantum computer, allowing you to test your circuit and record the results.

The following code selects the `Aer` simulator backend:

```
# Simulate using Aer simulator
simulator = Aer.get_backend('aer_simulator')
```

Next, run the quantum circuit using the simulator:

```
# Run the circuit
job = simulator.run(qc)
```

The following code retrieves the result object from the job:

```
result = job.result()
```

Print the measurement results to see how many times you get a 0 or 1:

```
# Print measurement results
print(result.get_counts())
```

The output printed is:

```
{'0': 495, '1': 529}  # Example output; actual counts may vary each run
```

Because the Hadamard gate creates a 50-50 split, you expect the counts for 0 and 1 to be almost equal. Each time you run the circuit, the results might vary slightly, but they should remain close to an even split.

In short, you built a one-qubit circuit, put it into superposition with a Hadamard gate, measured it, and read the counts from the Aer simulator. The near-even split between 0 and 1 is the visible signature of superposition. With a classical model and a quantum circuit both running in your environment, it is time to look at how machine learning and deep learning actually learn, which is the focus of the next section.

Exploring foundational machine learning and deep learning concepts

AI enables machines to perform tasks that typically require human intelligence, such as recognizing patterns, making decisions, and learning from experience. ML and DL are two key subfields of AI.

ML is a branch of AI that focuses on developing algorithms that allow computers to learn patterns from data and make predictions or decisions without being explicitly programmed. The way ML works is as follows:

1. First, you collect data, which is raw data gathered from various sources. Examples include images, text, and sensor data.
2. Then, you train the model. The machine learns patterns from data using an ML algorithm.
3. The trained model makes predictions or decisions based on new input data.
4. Finally, the model improves as more data is introduced to it.

Some of the key components of ML are as follows:

- A dataset is a collection of data used to train and test the model.
- Features are the characteristics or variables in the dataset that help you make predictions.
- The model is the mathematical function that learns from data and makes predictions.
- Training is the process of adjusting model parameters based on the input data.
- Evaluation is testing the model to measure its accuracy and effectiveness.

DL is an advanced subset of ML that uses artificial neural networks inspired by the human brain. These networks contain multiple layers of neurons that process data in hierarchical steps, allowing the model to automatically extract features and learn complex patterns, just like your brain.

Although DL is a subset of ML, the two differ in how much human effort they need, how much computing power they use, and how easy their results are to interpret.

The following table compares ML and DL across these dimensions:

Feature	Machine learning (ML)	Deep learning (DL)
Feature engineering	Requires manual feature selection	Learns features automatically
Computational power	Requires less	Needs high computational power
Performance on big data	Good but limited	Best suited for large datasets
Interpretability	Easier to understand	Acts like a "black box"

Table 1.1: Comparing machine learning with deep learning

Deep learning has a variety of real-world use cases. For example, in image recognition, self-driving cars use deep learning to detect pedestrians. In speech recognition, AI assistants such as Siri, Alexa, and Google Assistant rely on deep learning models to understand and process spoken commands. NLP is another area where deep learning is applied, powering chatbots, text summarization, and language translation.

Comparing supervised and unsupervised learning approaches

ML models are broadly categorized into supervised learning and unsupervised learning based on how they learn from data.

Let's talk about supervised learning, also referred to as learning from labeled data, first. The algorithm learns from labeled data, meaning each input has a

known correct output. An example can be predicting house prices based on size, location, and amenities.

Some of the supervised learning algorithms are regression and classification. Regression predicts continuous values. Examples include stock prices or temperatures. Classification predicts discrete categories. An example can be determining whether an email is spam or non-spam.

Next is unsupervised learning, which works with unlabeled data. The algorithm finds patterns in unlabeled data without predefined outcomes. An example can be grouping customers based on their purchasing behavior.

Some of the unsupervised learning techniques include clustering and dimensionality reduction. Clustering groups similar data points. An example can be customer segmentation. Dimensionality reduction, on the other hand, simplifies large datasets while retaining important information.

So, with all that we have learned so far, let's look at some real-world AI applications. AI is transforming industries worldwide. Here are some of the key applications:

- In healthcare, you can use AI-powered disease prediction, such as cancer detection. You can also find AI-assisted robotic surgeries and personalized treatment recommendations as important healthcare applications.
- In finance, AI is used for fraud detection in banking, stock market price prediction, and automated risk assessment. These examples represent real-world applications in finance.
- In retail and e-commerce, you encounter personalized product recommendations, such as those on Amazon or Netflix. AI-powered chatbots help with customer service, and demand forecasting supports inventory management.
- Self-driving cars use AI-based CV for detecting objects and reinforcement learning (RL) for autonomous navigation.
- In cybersecurity, AI detects phishing attacks and malware, and also provides real-time network monitoring for threat detection.

So, enough theory for now. Let's jump into our hands-on exercise, where we will build a simple AI model using linear regression.

Building a supervised learning model for house price prediction

The objective of this exercise is to build a simple AI model using linear regression to predict house prices based on the number of square feet.

Start by importing the essential libraries you will need for linear regression and data visualization in your Jupyter Notebook:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
```

Here's how these libraries help:

- Importing NumPy as np allows you to easily handle numerical operations and arrays, making mathematical calculations much easier.
- Importing `matplotlib.pyplot` as `plt` gives you the tools to create graphs and charts for visualizing your data.
- Importing `LinearRegression` from `sklearn.linear_model` brings in the `LinearRegression` class from scikit-learn, which you use to find relationships between numbers.

Next, let's define the data containing house sizes and prices. So, here, I have two parameters, or two variables: `X` is the square feet feature (independent variable), and `y` is the target price (dependent variable):

```
# Data: House sizes (sq ft) and corresponding prices
X = np.array([500, 700, 900, 1100, 1300]).reshape(-1, 1)  # Feature (sq ft)
y = np.array([150000, 180000, 210000, 250000, 280000])    # Target (price)
```

Next, create the linear regression model and train it:

```
# Create Linear Regression model
model = LinearRegression()

# Train the model on the data
model.fit(X, y)
```

To predict the price for a house with 1000 square feet, you use the trained model to estimate the value based on the relationship it learned from the data. The following code demonstrates how to make this prediction and print the result:

```
# Predict price for a house with 1000 sq ft
predicted_price = model.predict([[1000]])
print(f"Predicted Price for 1000 sq ft house: ${predicted_price[0]:,.2f}")
```

When you run this code, you see the predicted price for a 1,000-square-foot house displayed in the output:

```
Predicted Price for 1000 sq ft house: $230,500.00
```

Note that this output can vary on your side.

To visualize the relationship between house size and price, you create a scatter plot of the actual prices and overlay the regression line predicted by your model. The following code creates the scatter plot and regression line, adds axis labels, a title, and a legend, and then displays the plot:

```
plt.scatter(X, y, color='blue', label="Actual Prices")
plt.plot(X, model.predict(X), color='red', label="Regression Line (Best Fit)")
plt.xlabel("House Size (sq ft)")
plt.ylabel("House Price ($)")
plt.title("Linear Regression - House Price Prediction")
plt.legend()
plt.show()
```

The output is shown as follows:

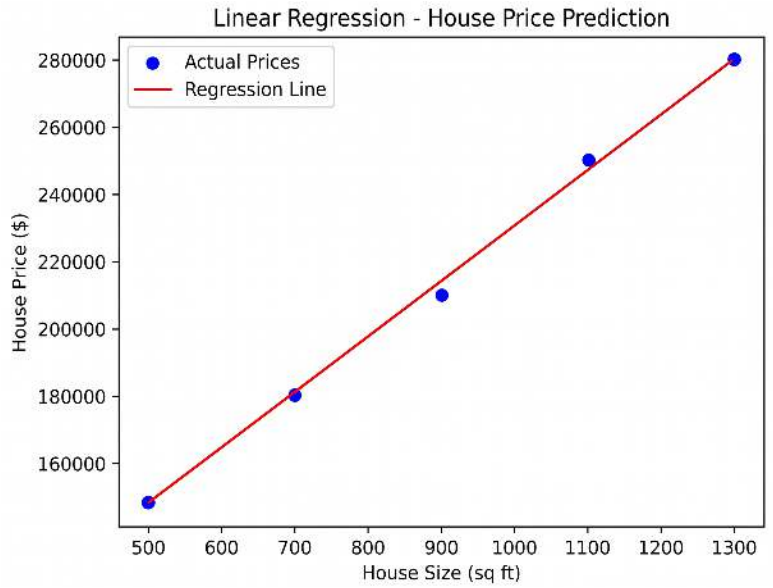


Figure 1.2: Linear regression model for house price prediction

The blue dots represent the actual house sizes and their corresponding prices, while the red regression line (best-fit line) shows the best-fit line that the model has learned.

A house with 1,000 square feet falls between 900 and 1,100 in your data, and the model calculates the price accordingly. The plot confirms that the model has learned a straight-line relationship between size and price, and this same pattern of learning from labeled data underlies the more advanced models in the chapters ahead. With your environment ready and both a classical model and a quantum circuit running, the summary that follows pulls together the key takeaways from this chapter.

Summary

AI and quantum computing solve problems in different ways. AI learns patterns from data to predict and decide, while quantum computing uses qubits, superposition, and entanglement to process information in ways classical bits cannot. Quantum advantage applies to specific problems such as molecular simulation and optimization, not to every task, and quantum AI remains an emerging area where expectations should stay grounded.

The most useful distinction to carry forward is between ML and DL. ML works well with smaller datasets and is easier to interpret, while DL learns features automatically but needs more data and computing power and behaves like a black box. Supervised learning learns from labeled examples, and unsupervised learning finds structure in unlabeled data. The house price model showed the core pattern behind all of it: fit a model to known examples, then use it to predict unseen inputs. Quantum results work differently, since you read them as counts across many runs rather than as a single answer.

In the next chapter, we will work with NumPy, Pandas, and Scikit-learn to prepare data for machine learning, handle missing values, encode categories, and scale features, and then build and evaluate the first classification model.

2

Building Models with Machine Learning and AI Frameworks

Chapter 1 introduced AI and quantum computing and got your environment ready. Every model you build from here on, classical or quantum, depends on data that is clean, structured, and in the right form. This chapter builds that foundation with the Python libraries that nearly every machine learning project relies on.

In this chapter, you will work with NumPy for numerical computing, Pandas for data manipulation, and Scikit-learn for building and evaluating models. You will learn to handle missing values, encode categories, and scale features so they work well together, and then put it all into practice by training a classification model that predicts whether a customer will buy a product. You will also learn to read its results using accuracy and a confusion matrix. This chapter covers the following topics:

- Exploring foundational Python libraries for machine learning
- Introducing feature engineering and data preprocessing
- Building and evaluating a classification model with Scikit-learn

Technical requirements

To follow the examples in this chapter, you will need the following:

- Python 3.12 or later, with `pip`
- JupyterLab, set up in *Chapter 1*
- The libraries `numpy` (version 1.18 or higher), `pandas` (version 1.0 or higher), and `scikit-learn` (version 1.2 or higher)

Install all the required libraries with the following command:

```
pip install "numpy>=1.18" "pandas>=1.0" "scikit-learn>=1.2"
```

> **NOTE**
>
> The quotes matter. Without them, the terminal treats > as redirection and creates files named `=1.18` instead of installing the packages.

The code files for this chapter are available at: https://github.com/PacktPublishing/Quantum-Computing-for-AI-with-Python/blob/main/Chapter2.ipynb

Exploring foundational Python libraries for machine learning

This section gives you a deep dive into the foundational Python libraries that are essential for machine learning and artificial intelligence. You will explore key tools such as NumPy, Pandas, and Scikit-learn, and see how they support your work in data analysis, feature engineering, and building classification models.

Using NumPy for numerical computing and array operations

NumPy is a library for numerical computing in Python. It provides support for multidimensional arrays and matrices, along with mathematical functions. Some of the key features of NumPy include efficient array operations, support for linear algebra, Fourier transforms, and random number generation. You will find that NumPy enables fast computations compared to standard Python lists.

To get started with NumPy, you first need to import the library using the common convention of the shortcut name `np`. This allows you to reference NumPy functions more easily throughout your code.

The following code imports NumPy as `np`:

```
import numpy as np
```

NumPy is especially useful for working with arrays, which are more efficient than Python lists for mathematical operations such as addition and multiplication. Arrays also support matrix operations, which are important for ML and AI tasks.

Creating a NumPy array

To create a NumPy array, you use the `np.array` function and provide a list of numbers. The resulting array is stored in a variable, and you can print it to see the output.

The following code creates a NumPy array with the numbers 1, 2, 3, 4, and 5, and prints the array:

```
arr = np.array([1, 2, 3, 4, 5])
print("Array:", arr)
```

The output is as follows:

```
Array: [1 2 3 4 5]
```

You might be thinking, what's the difference between NumPy arrays and Python lists? NumPy arrays are faster than lists and allow you to apply mathematical operations to all elements at once, a feature known as vectorized operations.

Reshaping an array

You can also reshape an array into a matrix, which is useful for organizing data in different formats. For example, you might want to convert a one-dimensional array into a two-dimensional matrix with a specific number of rows and columns.

The following code reshapes the array into a matrix with five rows and one column, and prints the reshaped matrix:

```
# Reshaping an array
matrix = arr.reshape(5, 1)
print("Reshaped Matrix:\n", matrix)
```

When you write `matrix = arr.reshape(5, 1)`, you are changing the shape of the array to a matrix with five rows and one column. The matrix variable stores this reshaped version and prints the following output:

```
Reshaped Matrix:
 [[1]
 [2]
 [3]
 [4]
 [5]]
```

You can experiment with different shapes, as long as the total number of elements matches.

Applying operations on arrays

You can perform various mathematical operations on NumPy arrays. For example, to square each element, you can write `arr_squared = arr ** 2`. This multiplies each element by itself and stores the result in `arr_squared`. When you print the squared array, you see the squares of the original elements: one squared is one, two squared is four, three squared is nine, four squared is sixteen, and five squared is twenty-five. The following code demonstrates squaring the elements:

```
# Performing mathematical operations
arr_squared = arr ** 2
print("Squared Array:", arr_squared)
```

The output produced is:

```
Squared Array: [ 1  4  9 16 25]
```

Generating a random matrix

To generate a random matrix, use `np.random.rand()`, for example, `np.random.rand(3, 3)`. This creates a three-by-three matrix of random numbers between zero and one. Here is the code for generating a random matrix:

```
# Generating a random matrix
random_matrix = np.random.rand(3, 3)
print("Random Matrix:\n", random_matrix)
```

When you run this code, you will see a random matrix with values between zero and one:

```
Random Matrix:
 [[0.49105925 0.77630105 0.02788947]
```

```
[0.35001937 0.88450674 0.78529975]
[0.06422064 0.95038461 0.06592808]]
```

Each time you run it, the numbers will be different.

It's useful in ML, where you can do random initialization of weights. This helps in simulations and probability experiments. Since the numbers are random, every time you run the code, you will get different results.

Array creation, reshaping, element-wise operations, and random number generation cover the NumPy operations you will use most often in ML work. NumPy handles numbers well, but real datasets arrive as tables with labeled columns and mixed data types, which is where Pandas takes over.

Using Pandas for data manipulation and analysis

Pandas is the most widely used library for data manipulation, data analysis, and preprocessing in ML. Some of the key features of Pandas are that it works with structured data such as tables, comma-separated values (CSV) files, Excel, and databases. It provides DataFrame and Series objects for handling data efficiently, and it supports data cleaning, filtering, transformation, and visualization.

> **NOTE**
>
> A **Series** is a one-dimensional labeled array that holds a single column of data, such as the ages of all customers. A **DataFrame** is a two-dimensional table of rows and columns, where each column is a Series, and all columns share the same row index. In short, a Series is one column, and a DataFrame is a collection of columns. When you select a single column from a DataFrame, such as `df["Age"]`, you get a Series back.

Pandas is imported in the code using the alias `pd`. The following code imports Pandas as `pd`:

```
import pandas as pd
```

Pandas helps you handle and analyze data efficiently, working like Excel in Python but much faster and more powerful.

Creating a Pandas DataFrame from a Python dictionary

To create a DataFrame, start by defining a dictionary, which is a collection of key-value pairs, where each key represents a column name and each value is a list containing the column data. For example, you can create a dictionary with the names, ages, and salaries of three people:

```
# Creating a DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35], 'Salary': [50000, 60000, 70000]}
```

This dictionary can then be converted to a Pandas DataFrame as follows:

```
df = pd.DataFrame(data)
print("DataFrame:\n", df)
```

The output printed is:

```
DataFrame:
      Name  Age  Salary
0   Alice   25   50000
1     Bob   30   60000
2 Charlie   35   70000
```

Here, the dictionary stores the data in a structured format similar to a table, with `Name`, `Age`, and `Salary` as the column names and their corresponding values as lists. Converting this dictionary to a DataFrame using `pd.DataFrame(data)` creates a Pandas DataFrame, which functions like an Excel table in Python. Printing the DataFrame displays the structured data as shown in the preceding output.

Reading data from a CSV file into a Pandas DataFrame

You can also read data directly from a CSV file into a Pandas DataFrame. The following line of code demonstrates how to do this:

```
# Reading data from a CSV file
df = pd.read_csv("data.csv")
```

It reads the data from the file and stores it in the `df` variable. You can replace the name of the file (`data.csv`) with the file that you have. Here is a sample `data.csv` file that you can create to follow along with this example:

```
Name,Age,Salary
Alice,25,50000
```

```
Bob,30,60000
Charlie,35,70000
```

Most real-world data is stored in CSV format. Many datasets available on platforms such as Kaggle (https://www.kaggle.com/) and GitHub (https://github.com/topics/dataset) are provided as CSV files. CSV files are popular because they can be easily imported and exported from Excel, databases, and websites, making them a common choice for data storage and transfer.

Displaying the first rows of a DataFrame

To show the first five rows of your DataFrame, use the `df.head()` function. If your DataFrame contains fewer than five rows, this function will display all available rows.

The following code prints the first five rows of the DataFrame:

```
print(df.head())
```

This output displays the first few rows of the DataFrame:

```
   Name  Age  Salary
0  Alice   25   50000
1    Bob   30   60000
2 Charlie   35   70000
```

If your DataFrame had one hundred rows, `df.head()` would still show only the first five. If there are only three rows, as in this example, it displays all three.

Generating descriptive statistics for numerical columns

To get some descriptive statistics for your numerical data, use the `df.describe()` function. This function generates summary statistics for all numerical columns in your DataFrame.

The following code prints descriptive statistics for the numerical columns in your DataFrame:

```
print(df.describe())
```

This displays the summary statistics for the numerical columns in the DataFrame as follows:

```
             Age   Salary
count   3.0      3.0
mean   30.0  60000.0
std     5.0  10000.0
min    25.0  50000.0
25%    27.5  55000.0
50%    30.0  60000.0
75%    32.5  65000.0
max    35.0  70000.0
```

The output includes several key statistics:

- `count`: The number of entries found for each column (3 for both Age and Salary).
- `mean`: The average value (30 for Age, 60,000 for Salary).
- `std`: The standard deviation, which measures variation in the data (5 for Age, 10,000 for Salary).
- `min` and `max`: The minimum and maximum values in each column.
- `25%`, `50%`, and `75%`: The 25th, 50th (median), and 75th percentiles, showing the data distribution across quartiles.

Using the `.describe()` method helps you understand data trends, distribution, and outliers in your existing data, which is useful for data analysis and machine learning.

Handling missing values in a DataFrame

Sometimes, your data may have missing values, which can affect your analysis or machine learning models. You can handle these missing values by replacing them with a specific value, such as zero, using the `df.fillna()` function. Setting `inplace=True` modifies the DataFrame directly.

The following code replaces all missing values in the DataFrame with zero:

```
df.fillna(0, inplace=True)
```

This approach ensures that any missing or not-a-number entries in your data are filled with zero, making the DataFrame ready for further processing.

Filtering rows in a DataFrame based on column values

You can filter your data to select particular rows based on a condition. In this example, you filter the DataFrame to keep only the rows where the value in the "Age" column is greater than 30.

The following code filters the DataFrame and prints the result:

```
filtered_df = df[df["Age"] > 30]
print("Filtered Data:\n", filtered_df)
```

This code prints the filtered DataFrame, `filtered_df`, which contains only the rows from the DataFrame where the Age column is greater than 30:

```
Filtered Data:
      Name  Age  Salary
2  Charlie   35   70000
```

In this case, only Charlie meets the condition, so the new DataFrame contains just Charlie's data. Filtering data in this way helps you select and analyze specific categories or subsets, which is useful in ML tasks.

You can now create, inspect, clean, and filter data in Pandas, which prepares it for modeling. The next step is to use that data to train a model, and Scikit-learn provides the tools to do it.

Applying scikit-learn for machine learning workflows

Scikit-learn is an ML library that provides tools for classification, regression, and clustering algorithms, as well as feature engineering, preprocessing tools, model evaluation, and hyperparameter tuning. You will use Scikit-learn for both supervised and unsupervised learning tasks, and its API is designed to work easily with NumPy and Pandas.

Importing essential scikit-learn modules

To get started with a basic Scikit-learn workflow, you will first import the necessary modules for splitting data, training a model, and evaluating its performance.

The following code imports the `train_test_split` function from `sklearn.model_selection`, which allows you to split your dataset into training and testing sets:

```
from sklearn.model_selection import train_test_split
```

Splitting your data ensures that your model is evaluated on data it has not seen during training. The training data is used to teach the model, whereas

the testing data is used to check how well the model performs.

Next, you import the `LogisticRegression` class from `sklearn.linear_model`. This is a machine learning algorithm used for classification tasks:

```
from sklearn.linear_model import LogisticRegression
```

The following code imports `accuracy_score`, a tool for measuring how well the model predicts the results:

```
from sklearn.metrics import accuracy_score
```

Creating a sample dataset for classification

To train and evaluate a machine learning model, you first need some sample data. Here, you create two variables: `X` for your input data (features) and `y` for your output labels (categories). The input data `X` is a list of lists, where each inner list contains a single integer value. This format is required because Scikit-learn expects the input features to be in a two-dimensional (2D) array. The labels `y` are a simple list of integers that correspond to the categories you want the model to predict.

The following code defines the sample data for your classification task:

```
# Sample data
X = [[1], [2], [3], [4], [5], [1], [2], [3], [4], [5], [1], [2], [3], [4], [5], [1], [2], [3], [4],
[5]]

y = [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]  # Labels
```

In this dataset, the numbers 1 through 5 are repeated multiple times in both `X` and `y` to provide enough examples for the model to learn from. Each value in `X` is wrapped in a list to maintain the required 2D structure. The goal is to train a model that can recognize the correct category in `y` based on the input value in `X`. For example, if the input is 3, the model should predict the category 3.

Splitting the data into training and testing sets

To split your data into training and testing sets, use the `train_test_split` function. This function takes your input data `X` and labels `y` and divides them into two groups: one for training the model and one for testing its accuracy.

The following code splits the data so that 80% is used for training and 20% for testing, with a fixed random state to ensure reproducibility:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

This returns a tuple of four values: `X_train`, `X_test`, `y_train`, and `y_test`. Here, `test_size=0.2` means that 20% of the data is set aside for testing, while the remaining 80% is used for training. You can adjust this ratio as needed, such as 10% for testing and 90% for training, or 30% and 70%. The `random_state=42` parameter ensures that the split is the same every time you run the function, which is important for consistent results.

Training a logistic regression model on the prepared data

Now that your data is ready, you can train a logistic regression model. First, you create the model by calling the `LogisticRegression()` function, and then fit the model to your training data using the `fit()` method:

```
model = LogisticRegression()
model.fit(X_train, y_train)
```

This step allows the model to learn the relationship between the input values (`X_train`) and the categories (`y_train`), and tries to find the best way to classify the numbers. In this example, the relationship is straightforward: the input values directly correspond to the output categories.

Making predictions and evaluating model accuracy

To make predictions using your trained model, assign the predicted values to a variable by calling the model's predict function on your test data:

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

Here, the model uses the patterns it learned during training to predict the categories for the unseen test data. The `y_pred` variable stores these predicted values.

Finally, to evaluate how well your model performed, use the `accuracy_score` function to compare the actual labels (`y_test`) with the predicted values (`y_pred`):

```
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

The `accuracy_score` function calculates the proportion of correct predictions by dividing the number of correct predictions by the total number of

predictions. This gives you a measure of how accurately your model classifies new data. The output produced here is:

```
Accuracy: 1.0
```

In this example, the accuracy is 1.0, or 100%, which means the model classified all test values correctly based on the input provided.

Interpreting accuracy results by modifying input data

Changing the input data affects the accuracy of your model. For example, if you only provide a single set of values, such as:

```
# Sample data
X = [[1], [2], [3], [4], [5]]
y = [1, 2, 3, 4, 5]  # Labels
```

And when you run the code, the accuracy drops to 0.0 because the model does not have enough data to make correct predictions:

```
Accuracy: 0.0
```

If you change some of the numbers in your dataset, such as by modifying one value, and run the code again, you may see the accuracy increase to 0.75. This happens because the model is able to match most of the values correctly, but not all, leading to 75% accuracy.

This process is important for interpreting the results of your machine learning workflow and for improving your models.

Now that you have seen how input data affects model accuracy, you are ready to explore feature engineering and data preprocessing to further improve your machine learning results.

Introducing feature engineering and data preprocessing

Before you feed data into ML models, you need to clean, transform, and structure it to improve performance. This is what brings us to the concept of feature engineering.

Feature engineering is the process of transforming raw data into meaningful features that help the model learn better.

Data preprocessing involves the following steps:

- Handling missing data: You can drop the missing values or fill them with the mean, median, or mode.
- Encoding categorical variables: You convert categorical variables into numerical form. For example, if you have a column with values such as "male" and "female," you can use one-hot encoding to transform these text values into numbers. One-hot encoding takes all the string values in your categorical data and converts them into numerical form, assigning numbers to each unique category.
- Feature scaling: You normalize or standardize the numerical features to improve model performance.
- Feature selection: You select the most relevant features to avoid overfitting. You select features based on the data you are using, and often data scientists or domain experts can help you decide which features are most important for training your model. Selecting the right features helps prevent your model from overfitting to the training data.

Cleaning, encoding, scaling, and selecting features often matter as much as the choice of algorithm itself. To see how these steps work in practice, the next section applies them to a small dataset using Pandas and Scikit-learn.

Implementing data preprocessing with Pandas and scikit-learn

Now let us put data preprocessing into practice using Pandas and Scikit-learn.

Start by importing Pandas and the preprocessing tools from Scikit-learn that you will need for data preparation:

```
import pandas as pd
from sklearn.preprocessing import OneHotEncoder, StandardScaler
```

You have imported pandas for handling structured data such as Excel sheets, tables, and data structures. `OneHotEncoder` converts categorical data into numbers. Finally, `StandardScaler` scales numerical data to ensure all features have the same scale and there are no ups and downs.

> **NOTE**
>
> Why is this important?

> ML only understands numbers, and that is why you convert text data (such as genders) into numbers using the `OneHotEncoder`. Additionally, features with large numbers, for example, salary, should not dominate over small numbers such as age. This is resolved by scaling using `StandardScaler`.

Next, create a sample dataset that includes both missing values and categorical variables:

```
# Sample dataset
data = {'Age': [25, 30, 35, 40, None],
        'Salary': [50000, 60000, 70000, None, 90000],
        'Gender': ['Male', 'Female', 'Male', 'Female', 'Male']}
df = pd.DataFrame(data)
print(df.head())
```

This produces the output as follows:

```
# Output:
    age  gender   salary
0  25.0    male  50000.0
1  30.0  female  60000.0
2   NaN  female  52000.0
3  22.0    None  58000.0
4  28.0    male      NaN
```

This DataFrame now contains numerical columns with missing values and a categorical column with string values.

Handling missing values

Next, you need to handle the missing values and fill them with the mean. Since the dataset is very small, you do not want to drop any rows. Instead, you use the mean of the existing values to fill in the missing entries for both the `Age` and `Salary` columns.

The following code fills missing values in the `age` and `salary` columns with their respective means:

```
df["Age"] = df["Age"].fillna(df["Age"].mean())
df["Salary"] = df["Salary"].fillna(df["Salary"].mean())
```

This approach looks for any `None` values in the columns, calculates the mean of the available values, and fills the missing entries with that mean. If you were

to remove the rows with missing values, you would lose data, which is not ideal for such a small dataset. Replacing missing values with zero could also distort the dataset, making the values for age or salary unrealistic. Using the mean provides a good approximation in this case.

Encoding categorical values

To encode the categorical variable, you use the `OneHotEncoder` from scikit-learn. The following code creates an `encoder` object with the parameter `sparse_output=False`, which ensures the output is not compressed and is easier to work with for small datasets:

```
encoder = OneHotEncoder(sparse_output=False)
```

You then apply the encoder to the 'Gender' column of your DataFrame. This converts the 'Male' and 'Female' values into a binary format, creating separate columns for each category:

```
gender_encoded = encoder.fit_transform(df[['Gender']])
```

Next, you convert the encoded data into a new DataFrame with columns named 'Male' and 'Female':

```
df_gender = pd.DataFrame(gender_encoded, columns=['Male', 'Female'])
```

One-hot encoding is necessary because algorithms do not understand text values such as 'male' and 'female'. By converting these to numbers, you avoid introducing bias that could occur if you simply assigned values such as male = 1 and female = 0, which might incorrectly suggest an order or ranking between the categories.

Scaling the numerical features

Next, you need to scale the numerical features in your dataset. The following code creates a StandardScaler object and applies it to the `Age` and `Salary` columns in your DataFrame:

```
scaler = StandardScaler()
df[['Age', 'Salary']] = scaler.fit_transform(df[['Age', 'Salary']])
```

`StandardScaler` transforms all numerical values so that they have a mean of zero and a standard deviation of one, making the data more similar to each

other. Scaling ensures that each feature contributes equally to the model's learning and one feature does not dominate another (for example, larger salary numbers could dominate smaller age numbers).

Concatenating the final processed data

To combine your processed features into a final dataset, you concatenate the scaled numerical data (`Age` and `Salary`) with the encoded categorical data (`Male` and `Female`). The following code performs this concatenation and prints the resulting preprocessed data:

```
# Concatenating final processed data
df_final = pd.concat([df[['Age', 'Salary']], df_gender], axis=1)
print("Preprocessed Data:\n", df_final)
```

The output produced is:

```
Preprocessed Data:
   Age    Salary  Male  Female
0 -1.5 -1.322876   0.0     1.0
1 -0.5 -0.566947   1.0     0.0
2  0.5  0.188982   0.0     1.0
3  1.5  0.000000   1.0     0.0
4  0.0  1.700840   0.0     1.0
```

In the output, you will see that the `Age` and `Salary` columns now have values close to zero, and the `Male` and `Female` columns are included as encoded features. This makes it much easier for an ML model to learn from your dataset and train effectively.

Your data is now numeric, complete, and scaled, which is the form most algorithms expect. The next section puts this preparation to work in a complete project, where you train a classification model and measure how well it performs.

Building and evaluating a classification model with scikit-learn

In this hands-on project, you will train a classification model using scikit-learn to predict whether a customer will buy a product based on their age and salary. The steps that we will follow are:

1. Import the necessary libraries.

2. Prepare the dataset.
3. Preprocess the data.
4. Train a classification model.
5. Evaluate the model.

Importing the necessary libraries

The following code imports Pandas for tabular data handling and NumPy for array operations:

```
import numpy as np
import pandas as pd
```

Next, you import the `train_test_split` function from scikit-learn to split your data into training and testing sets, and `StandardScaler` to scale your numerical features:

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

You then import the `KNeighborsClassifier`, which is the K-nearest neighbors algorithm used for classification tasks.

NOTE

K-nearest neighbors (**KNN**) is a classification algorithm that predicts the class of a new data point by looking at the training points closest to it. The value of k sets how many neighbors vote. With k = 3, the model finds the three closest training points, measured by the distance between their feature values, and assigns the class most of them share. KNN does not learn parameters during training. It stores the training data and compares new points against it at prediction time. Because it relies on distance, features must be on a similar scale, which is why you standardize age and salary before training.

The following code imports KNeighborsClassifier, which implements the KNN algorithm:

```
from sklearn.neighbors import KNeighborsClassifier
```

Finally, you import `accuracy_score` and `confusion_matrix` from scikit-learn's metrics module to evaluate your model's performance:

```
from sklearn.metrics import accuracy_score, confusion_matrix
```

You need all these libraries because your data must be cleaned, split, and scaled before you train a model. The k-NN algorithm predicts the class of a data point based on the closest data points in the training set. Evaluating accuracy and using a confusion matrix helps you understand how well your model performs.

Creating a sample dataset

You start by creating a sample dataset that includes the columns `Age`, `Salary`, and `Purchased`, where purchased is 0 for no and 1 for yes:

```
# Sample dataset
data = {'Age': [22, 25, 47, 52, 46, 56, 36, 23, 45, 55],
        'Salary': [20000, 25000, 50000, 65000, 50000, 70000, 40000, 23000, 60000, 72000],
        'Purchased': [0, 0, 1, 1, 1, 1, 0, 0, 1, 1]}
```

Next, you convert this dictionary into a Pandas DataFrame:

```
df = pd.DataFrame(data)
```

The goal is to predict whether a person will purchase a product (purchased = 1 or 0) based on their age and salary.

You then define the features and target for your classification task. The features are age and salary, and the target is the `Purchased` column:

```
# Define features and target
X = df[['Age', 'Salary']]
y = df['Purchased']
```

Here, the `X` variable contains the independent variables (`Age` and `Salary`) that influence the outcome, and the `y` variable contains the dependent variable (`Purchased`) that you want to predict.

Splitting, scaling, and training with the KNN classifier

To split your dataset into training and testing sets, use the following code. This separates 80% of the data for training and 20% for testing, ensuring reproducibility with a fixed random state:

```
# Split dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Next, standardize your features so that both age and salary have a mean of 0 and a standard deviation of 1. This prevents features with larger values, such as salary, from dominating the predictions:

```
# Standardize features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Now, train a k-NN classifier using three neighbors. The KNN algorithm predicts the class of a sample based on the majority vote of its three closest neighbors:

```
# Train K-Nearest Neighbors (KNN) classifier
model = KNeighborsClassifier(n_neighbors=3)
model.fit(X_train, y_train)
```

With these steps, you have prepared your data, scaled your features, and trained a KNN classification model that predicts whether a user will purchase based on age and salary. If the three nearest neighbors are, for example, [1, 1, 0], the prediction will be 1, as the majority is 1. If they are [0, 0, 1], the prediction will be 0.

Making predictions on data and evaluating the model

You can make predictions using your trained KNN model by applying the `predict()` method to your test data:

```
# Make predictions
y_pred = model.predict(X_test)
```

Next, compute the accuracy of your model by comparing the predicted values to the actual test labels. Print the model accuracy as a percentage:

```
# Compute accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy * 100:.2f}%")
```

To further evaluate your classification model, generate a confusion matrix. This matrix provides insight into the number of true positives, true negatives, false positives, and false negatives:

```
# Confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", conf_matrix)
```

When you run this code, you will see the model's accuracy and the confusion matrix printed as follows:

```
Model Accuracy: 100.00%
Confusion Matrix:
[[1 0]
 [0 1]]
```

The model achieves 100% accuracy; this means all test samples were classified correctly. The confusion matrix output shows that one person was correctly classified as not purchasing (0), and one person was correctly classified as purchasing (1). The confusion matrix summarizes the model's performance by displaying the counts of correct and incorrect predictions for each class.

With these steps, you have completed the process of building, training, and evaluating a classification model using scikit-learn, including interpreting the results using both accuracy and the confusion matrix.

Summary

NumPy, Pandas, and scikit-learn each solve a different part of the same problem. NumPy gives you fast numerical operations on arrays, pandas gives you labeled, table-like data you can clean and filter, and scikit-learn gives you a consistent workflow of splitting, fitting, predicting, and evaluating. The most important habit to carry forward is to always evaluate a model on data it has not seen. Very small or repetitive datasets can produce a perfect score that says little about real performance.

Data preparation shapes what a model can learn. Missing values need a deliberate strategy, categories must be converted to numbers without implying a false order, and features must be scaled so that large values do not dominate small ones, especially for distance-based models such as KNN.

Accuracy tells you how often a model is right, while a confusion matrix shows which kinds of mistakes it makes, so use both.

In the next chapter, you will build neural networks with TensorFlow and PyTorch to classify images, see how deep learning handles text, and train reinforcement learning agents using Q-learning and deep Q-networks.

3

Training Deep Learning and Reinforcement Learning Models

Chapter 2 showed how to prepare data and train classical ML models. Some problems go beyond what those models handle well, such as recognizing objects in photos, understanding sentences, or learning to play a game. They need models that learn useful features on their own, or that improve through experience. This chapter introduces the neural network and reinforcement learning techniques behind them.

You will build neural networks in both TensorFlow and PyTorch, train agents that learn from rewards, and see how choices such as data normalization and hyperparameters shape how well a model learns.

In this chapter, we will cover the following topics:

- Exploring neural networks and backpropagation in deep learning
- Understanding convolutional neural networks for image processing
- Applying neural networks to natural language processing with RNNs and transformers
- Building an image classifier using CNNs
- Training reinforcement learning agents with Q-learning and deep Q-networks
- Understanding how Q-learning and deep Q-networks learn
- Implementing reinforcement learning agents in Gym

Technical requirements

To follow the examples in this chapter, you will need the following:

- Python 3.12 or later, and JupyterLab, set up in *Chapter 1*
- The libraries `tensorflow`, `keras`, `numpy`, `matplotlib`, `torch`, `torchvision`, `gymnasium`, and `stable-baselines3`

Install all the required libraries with the following command:

```
pip install tensorflow keras numpy matplotlib torch torchvision gymnasium stable-baselines3
```

The code files for this chapter are available at: https://github.com/PacktPublishing/Quantum-Computing-for-AI-with-Python/blob/main/Chapter3.ipynb

NOTE

Download the code bundle and the PDF version of this book

Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Exploring neural networks and backpropagation in deep learning

Deep learning is a subset of machine learning (ML) that uses neural networks to learn from data. A **neural network** takes inspiration from the human brain and consists of layers of interconnected neurons. It takes input, processes it through hidden layers, and produces an output.

The structure of a neural network has three main layers as follows:

- The input layer, which takes in the raw data, such as images, text, or numbers
- The hidden layers, which perform mathematical transformations
- The output layer, which produces the final result, which can be a classification label

For example, in a three-layered neural network, you have an input, then hidden layer one, hidden layer two, and an output.

A neural network learns through **backpropagation**, which is a method to update its weights based on errors. The steps of backpropagation are as follows:

1. Forward propagation: Compute the output of the network.
2. Loss calculation: Measure the error between the predicted and actual values.
3. Backward propagation: Adjust weights using the gradient descent algorithm.
4. Repeat: Iterate multiple times (epochs) until the error is minimized.

The mathematical representation for backpropagation, if a neuron has weights (w), input (x), and bias (b), is given by the following formula, where f is the activation function:

$$y = f(w_1 x_1 + w_2 x_2 + \cdots + b)$$

Examples of activation functions include ReLU, sigmoid, or softmax.

Understanding convolutional neural networks for image processing

A **convolutional neural network** (**CNN**) is a special type of neural network used for image recognition and processing. Unlike regular neural networks, a CNN automatically detects patterns and features such as edges, textures, and shapes from images.

The architecture of a CNN includes the following layers:

- A convolutional layer that extracts features from the image using filters or kernels.
- A pooling layer reduces the size of the extracted features, which makes computation much faster.
- A fully connected layer classifies the image based on the extracted features.

For example, in a typical CNN structure for image classification, you start with an input image, then apply a convolutional layer, followed by a pooling layer, and finally a fully connected layer that produces the output.

The process of recognizing images with a CNN involves several steps: Convolution -> Pooling -> Flattening -> Classification.

Here is what each step does:

- During convolution, the network applies a filter (also called a kernel) to detect edges, colors, and textures.
- Pooling then down-samples the image while preserving important features.
- Flattening converts the pooled feature map into a 1D vector.
- Classification uses a fully connected layer with a softmax activation function to classify the images.

You can use deep learning techniques to process and understand text using neural networks. Next, you will learn about natural language processing with recurrent neural networks and transformers, and see how these models handle sequential data such as words and sentences.

Applying neural networks to natural language processing with RNNs and transformers

Natural language processing (**NLP**) is a branch of AI that enables machines to understand human language. **Recurrent neural networks** (**RNNs**) are neural networks designed for sequential data such as text, speech, and time series. One of the key features of RNNs is that they can remember previous words while processing text, using a loop structure to pass information from one step to the next.

However, RNNs have some problems, including the vanishing gradient issue. When processing long text sequences, RNNs sometimes struggle to remember earlier words, which leads to poor performance.

Transformers address this problem by processing all the words in a sequence simultaneously instead of sequentially. Transformers use self-attention to understand the relationships between words, which allows them to handle long-range dependencies very efficiently. This architecture powers models such as **GPT** (**Generative Pre-trained Transformer**), **BERT** (**Bidirectional Encoder Representations from Transformers**), and **T5** (**Text-to-Text Transfer Transformer**).

Some popular transformer-based models include:

- **BERT**: Used for question answering and text classification.
- **GPT**: Used for text generation and chatbots.

- **T5**: Converts NLP tasks into text generation problems, with T5 as an example model.

You have seen how CNNs extract features from images and how RNNs and transformers handle text. To see these ideas at work, the next section builds an image classifier from scratch, first in TensorFlow and then in PyTorch.

Building an image classifier using CNNs

In this hands-on exercise, we will train a CNN to classify images using TensorFlow and PyTorch. We will build:

- Image classifier using CNN (TensorFlow/Keras)
- Image classifier using CNN (PyTorch)

Start by setting up your Jupyter notebook environment for both projects.

Using TensorFlow to create the image classifier

In this section, you will build a CNN with TensorFlow and its high-level Keras API to classify images from the *CIFAR-10 dataset*. You will load and normalize the data, define the network, train it, and see how its accuracy improves as you add epochs.

Installing and importing required libraries in TensorFlow

To get started with TensorFlow, you first need to install the required dependencies. Run the following command in your Jupyter notebook to install TensorFlow, Keras, NumPy, and matplotlib:

```
pip install tensorflow keras numpy matplotlib
```

Once the installation is complete, you can import the necessary libraries for your projects:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
```

TensorFlow is a deep learning library designed for building and training image recognition models. Keras is a high-level API within TensorFlow that

simplifies the process of creating neural networks. The layers module provides the building blocks for your deep learning models. While matplotlib is included for data visualization, you are encouraged to use it as an assignment to visualize the data you will be working with.

Loading the CIFAR-10 dataset for image classification

You will now load the CIFAR-10 dataset, which is commonly used for training and testing image recognition models. The CIFAR-10 dataset contains 60,000 images divided into 10 different categories. Out of these, 50,000 images are used for training (`X_train`), and 10,000 images are used for testing (`X_test`). The labels, `y_train` and `y_test`, contain the class numbers ranging from 0 to 9. Each number corresponds to a specific category: for example, 0 is an airplane, 1 is a car, 2 is a bird, 3 is a cat, 4 is a deer, 5 is a dog, 6 is a frog, 7 is a horse, 8 is a ship, and 9 is a truck. If you need to check the exact mapping of these categories, you can search for CIFAR-10 images online. The following figure shows an example of the CIFAR-10 dataset:

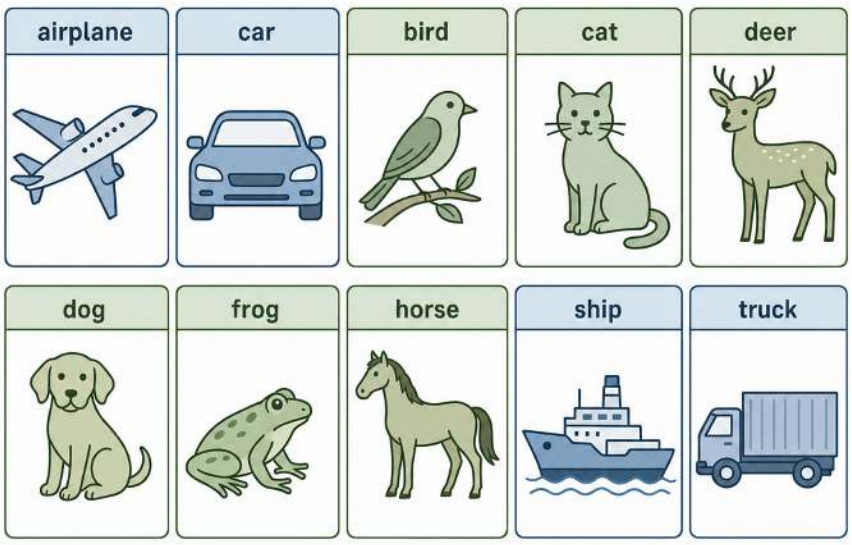


Figure 3.1: CIFAR-10 object categories

The following code loads the CIFAR-10 dataset using Keras:

```
# Load CIFAR-10 dataset (10 categories of images)
(X_train, y_train), (X_test, y_test) = keras.datasets.cifar10.load_data()
```

Normalizing image data

Before training your CNN, you need to normalize the pixel values in your image data. The CIFAR-10 dataset contains RGB values for each pixel in the range of 0 to 255. By dividing each pixel value by 255, you scale all values to fall between 0 and 1.

Normalizing the data in this way helps your deep learning models train faster and prevents large pixel values from dominating smaller ones.

The following code normalizes the pixel values in both your training and test datasets:

```
# Normalize pixel values
X_train, X_test = X_train / 255.0, X_test / 255.0
```

Defining a CNN architecture using Keras Sequential API

You define your CNN model using the Keras Sequential API, which allows you to stack layers in order. The model consists of convolutional, max pooling, flatten, and dense layers, each serving a specific purpose in feature extraction and classification.

The following code creates the CNN architecture for classifying CIFAR-10 images:

```
model = keras.Sequential([
    layers.Conv2D(32, (3,3), activation='relu', input_shape=(32,32,3)),
    layers.MaxPooling2D(2,2),
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D(2,2),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')  # 10 classes
])
```

The following layers are present in this code:

- The first convolutional layer uses 32 filters with a 3 × 3 kernel and ReLU activation, taking 32 x 32 RGB images as input.
- The max pooling layer reduces the image size to 16x16, keeping important features.
- The second convolutional layer uses 64 filters to detect more complex features.
- Another max pooling layer is present that reduces the size to 8 x 8.

- The flatten layer converts the feature map into a 1D array for the dense layers.
- The first dense layer has 128 neurons with ReLU activation to learn patterns from the extracted features.
- The final dense layer has 10 neurons with softmax activation, outputting probabilities for each of the 10 classes.

Compiling and training the CNN

Now you need to compile the model before training. The optimizer is set to Adam, the loss function is sparse categorical cross-entropy, and the metric tracked is accuracy. Then you train the model using the `fit()` function, passing in the training data, number of epochs, and validation data.

The following code compiles the model and starts training for one epoch, validating on the test set:

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=1, validation_data=(X_test, y_test))
```

The Adam optimizer automatically adjusts the weights to improve the model's accuracy. The sparse categorical cross-entropy loss function measures the error for multi-class classification problems, and you use it here because your labels are integers from 0 to 9, not one-hot encoded. The accuracy metric tracks how well the model is performing during training.

During training, the CNN extracts features from the images, the dense layer learns patterns, and the model adjusts its weights using the Adam optimizer. After training, the model predicts which category each image belongs to.

To evaluate the model on unseen test data and print the test accuracy, use the following code:

```
test_loss, test_acc = model.evaluate(X_test, y_test)
print("Test Accuracy:", test_acc)
```

This evaluates the trained model on the test set and prints the resulting accuracy.

When you execute the code, the model goes through the entire CIFAR-10 dataset and prints the test accuracy. After running for one epoch, you see the following output:

```
Test Accuracy: 0.5956000089645386
```

This means your model achieved about 59% accuracy after training for just one epoch.

If you increase the number of epochs, the model will take more time to train, but the accuracy should improve. To do this, change the `epochs` parameter in your model's fit function from 1 to 5:

```
model.fit(X_train, y_train, epochs=5, validation_data=(X_test, y_test))
```

As the training completes, you see the progress for each epoch and the final test accuracy:

```
Epoch 1/5
1563/1563 [==============================] - 14s 9ms/step - accuracy: 0.3864 - loss: 1.6914 -
val_accuracy: 0.5619 - val_loss: 1.2298
Epoch 2/5
1563/1563 [==============================] - 14s 9ms/step - accuracy: 0.5923 - loss: 1.1578 -
val_accuracy: 0.6361 - val_loss: 1.0682
Epoch 3/5
1563/1563 [==============================] - 15s 9ms/step - accuracy: 0.6553 - loss: 0.9923 -
val_accuracy: 0.6387 - val_loss: 1.0114
Epoch 4/5
1563/1563 [==============================] - 14s 9ms/step - accuracy: 0.6903 - loss: 0.8891 -
val_accuracy: 0.6660 - val_loss: 0.9584
Epoch 5/5
1563/1563 [==============================] - 14s 9ms/step - accuracy: 0.7164 - loss: 0.8059 -
val_accuracy: 0.6786 - val_loss: 0.9345
313/313 [==============================] - 1s 4ms/step - accuracy: 0.6817 - loss: 0.9255
Test Accuracy: 0.678600013256073
```

After training for five epochs, your model's test accuracy increases to about 68%, which is a significant improvement over the earlier result. The more epochs you train for, the better your model becomes at predicting the output, just as you improve your own skills with more practice.

The model you are training here, a CNN model, will become much better at predicting the output the more you train it for additional epochs.

Using PyTorch to create the image classifier

In this section, you will build a CNN for handwritten digit classification in PyTorch using the MNIST dataset. You will prepare the data with transforms

and data loaders, define the network as a class, train it, and measure its accuracy on unseen test data.

Installing and importing the necessary libraries

Start by installing and importing the required libraries. To install the necessary libraries, run the following command in your Jupyter notebook cell (or command prompt, or terminal):

```
pip install torch torchvision matplotlib
```

Next, import the required libraries for your PyTorch image classification project:

```
import torch
import torch.nn as nn
import torchvision.transforms
import torchvision.datasets as datasets
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import torchvision.transforms as transformsHere:
```

Here, the imports are as follows:

- `import torch` brings in PyTorch, the deep learning framework.
- `import torch.nn as nn` imports the neural network components, such as layers and loss functions.
- `import torchvision.transforms as transforms` handles image transformations such as converting images to tensors and normalizing them.
- `import torchvision.datasets as datasets` provides built-in datasets such as MNIST for handwritten digits.
- `from torch.utils.data import DataLoader` helps you load and batch data efficiently.
- `import matplotlib.pyplot as plt` is for visualizing data.

PyTorch is preferred for its easier debugging compared to TensorFlow, its dynamic computational graph, flexible model design, and faster training with GPUs.

Next, you define a transformation pipeline using `transforms.Compose()` that converts images to PyTorch tensors and normalizes their pixel values:

```
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
```

Here, `transforms.ToTensor()` converts the images into PyTorch tensors, and `transforms.Normalize((0.5,), (0.5,))` normalizes the pixel values so that instead of ranging from 0 to 255, they are scaled to be between -1 and 1, centered around 0. This normalization helps your CNN learn better by keeping the values in a similar range and preventing large numbers from dominating smaller ones.

Loading the MNIST dataset

To load your MNIST dataset, use the following code to specify the root directory, whether the data is for training or testing, whether to download it, and which transform to apply:

```
train_data = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_data = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
```

This loads the MNIST dataset, which contains handwritten digit images from 0 to 9. The `root='./data'` argument tells PyTorch to store the data in a local folder named 'data'. Setting `train=True` loads the training dataset (60,000 images), while `train=False` loads the testing dataset (10,000 images). The `download=True` argument ensures the dataset is downloaded if it is not already present, and `transform=transform` applies the transformations that convert images to tensors and normalize them.

Creating the data loaders

Next, create your data loaders to efficiently load images in batches for training and testing:

```
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=64, shuffle=False)
```

The data loader processes images in batches of 64 for faster training. Setting `shuffle=True` for the training loader randomizes the training data for each epoch, which improves training. For the test loader, `shuffle=False` keeps the test data in order, as shuffling is not needed for evaluation. Using data loaders is more efficient than loading one image at a time and allows for parallel processing, which is faster on GPUs.

Defining a CNN architecture for digit classification in PyTorch

You define your CNN for digit classification by creating a class that inherits from `nn.Module`. Inside the class, you implement the `__init__` function and set up the convolutional, pooling, and fully connected layers.

The following code defines the CNN class and its initialization method:

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(32 * 14 * 14, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = x.view(-1, 32 * 14 * 14)
        x = self.fc1(x)
        return x
```

In this architecture, the first convolutional layer takes a single input channel (for grayscale images) and outputs 32 channels, using a 3 x 3 kernel, a stride of 1 (moving 1 pixel at a time), and padding of 1 to keep the image size unchanged. The pooling layer uses a 2 x 2 max pooling operation to reduce the image size from 28 x 28 to 14 x 14, keeping only the most important features. The fully connected layer flattens the feature map from 32 x 14 x 14 to a 1D array and outputs 10 neurons, one for each digit class (0 to 9).

Initializing the CNN model, loss function, and optimizer in PyTorch

You initialize your model by creating an instance of the CNN class, then set up the loss function and optimizer for training. The loss function you use is cross-entropy loss, which measures how well the model predicts the correct label. For optimization, you use the Adam optimizer with a learning rate of 0.001, which automatically adjusts the model weights to reduce the loss. Adam is chosen because it is sufficient and works well for convolutional neural networks in PyTorch.

The following code initializes the model, loss function, and optimizer:

```
model = CNN()

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Adam adjusts the model weights automatically to reduce loss, and the learning rate (`lr`) determines how fast the model learns.

Training the CNN model on MNIST data over multiple epochs in PyTorch

You specify how many epochs you want to run by setting the range in your training loop. In this example, you run the training for 5 epochs, iterating over the training data in batches.

The following code runs the training loop for 5 epochs, performing forward propagation, loss computation, backpropagation, and optimizer steps for each batch:

```
for epoch in range(5):
    for images, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch+1}, Loss: {loss.item():.4f}")
```

This loop processes the entire dataset in each epoch, predicts the digits, computes the loss, performs backpropagation to adjust the weights, and updates the weights using the optimizer. The loss for each epoch is printed to help you track the model's progress.

Evaluating the trained CNN model and interpreting accuracy

After training your CNN model for five epochs, you need to evaluate its performance on the test dataset. To do this, you first set the variables `correct` and `total` to zero. Then, you use `torch.no_grad()` to ensure that the model is in evaluation mode and gradients are not calculated during testing.

The following code evaluates the trained model on the test data and prints the test accuracy:

```
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Test Accuracy: {100 * correct / total:.2f}%")
```

This code iterates over the test data, makes predictions, counts the number of correct predictions, and calculates the overall accuracy as a percentage.

When you run this evaluation after training, you see the loss values for each epoch and the final test accuracy:

```
Epoch 1, Loss: 0.0660
Epoch 2, Loss: 0.0374
Epoch 3, Loss: 0.0177
Epoch 4, Loss: 0.1829
Epoch 5, Loss: 0.1550
Test Accuracy: 98.23%
```

Training reinforcement learning with Q-learning and deep Q-networks

Reinforcement learning (RL) is a type of machine learning where an agent learns to perform actions in an environment to maximize rewards. Unlike supervised learning, reinforcement learning does not require labeled data; instead, you train the agent through trial and error as it interacts with the environment to maximize its reward.

RL involves an agent that interacts with an environment to maximize a numerical reward. Some of the key components you will work with in reinforcement learning are the agent, environment, state (S), action (A), reward (R), policy (π), and Q-value (Q(S, A)).

Here is what these components do:

- The **agent** is the AI model that makes decisions.
- The **environment** is the system where the agent operates, such as a game or a self-driving car.

- The **state** (S) represents the current situation of the agent.
- The **action** (A) is the set of moves the agent can make.
- The **reward** (R) is a numerical score given for good or bad actions.
- The **policy** (π) is the strategy used by the agent to decide the next action.
- The **Q-value (Q)** is the expected reward of taking action A in state S.

The RL workflow follows these steps:

1. The agent observes the state of the environment.
2. It selects an action based on a policy.
3. The environment gives a reward and transitions to a new state.
4. The agent then updates its policy to maximize future rewards.

This process repeats over multiple episodes until the agent learns the optimal strategy.

Understanding how Q-learning and deep Q-networks learn

Q-learning is a model-free RL algorithm that helps an agent learn optimal action selection. It uses a Q-table to store expected rewards for each state-action pair.

The Q-learning formula, also known as the Bellman equation, is expressed as follows:

$$Q(S,A) = Q(S,A) + \alpha[R + \gamma \max Q(S',A') - Q(S,A)]$$

Here:

- α is the learning rate, which controls how much new information affects existing Q values.
- γ is the discount factor, which determines how much future rewards are valued.
- S' is the next state, which is the state the agent transitions to after taking action (A).
- A' is the next action taken in the new state.

Next, let's look at **deep Q-networks** (**DQN**), which use neural networks to approximate Q-values instead of a Q-table.

The following table compares Q-learning and DQN across state representation, scalability, and memory usage:

Feature	Q-learning	Deep Q-networks (DQN)
State representation	Uses a Q-table.	Uses a neural network.
Scalability	Works well for small environments.	Suitable for complex environments such as games or robotics
Memory usage	Stores all state-action pairs.	Learns Q-values from experience.

Table 3.1: Key differences between Q-learning and deep Q-networks (DQN)

A Q-table is simple and transparent, but it only works when every state can be listed. A neural network removes that limit, and the next section puts both approaches into practice by training agents in Gym environments.

Implementing reinforcement learning agents in Gym

In this hands-on exercise, you will train an RL agent to solve the OpenAI Gym "FrozenLake and CartPole" environment. If you are not familiar with it, you are encouraged to look it up before proceeding. You can find the documentation at `https://gymnasium.farama.org/environments/classic_control/cart_pole`.

Installing dependencies

Let's begin by installing all the required dependencies for your reinforcement learning experiments in a Jupyter notebook.

The following command installs the necessary libraries: OpenAI Gym, NumPy, TensorFlow, PyTorch, and Stable-Baselines3:

```
pip install --upgrade gym numpy tensorflow torch stable-baselines3
```

OpenAI Gym provides a standard interface for creating and interacting with various reinforcement learning environments. Stable-Baselines3 is a library that provides implementations of various reinforcement learning algorithms, allowing you to easily train agents with a clean interface.

Once the dependencies are installed, you import NumPy and Gym in your notebook for mathematical calculations and to access reinforcement learning environments. The following code imports these libraries:

```
import numpy as np
import gym
```

You use Gym because it provides pre-built environments for reinforcement learning, such as FrozenLake-v1, which is a simple grid-based game where an agent learns to navigate without falling into holes.

Creating the FrozenLake environment

The next step is to create the FrozenLake-v1 environment and set up the state and action sizes for your reinforcement learning agent.

The following code creates the environment, retrieves the number of possible states and actions, and initializes the Q table with zeros:

```
# Create Environment
env = gym.make("FrozenLake-v1", is_slippery=False)
state_size = env.observation_space.n
action_size = env.action_space.n

Q_table = np.zeros((state_size, action_size))
```

In this setup, FrozenLake-v1 is a 4x4 grid-based game where each tile represents a state, resulting in 16 possible states. The agent can move in four directions: left, right, up, and down, which are the possible actions. Setting `is_slippery=False` disables randomness in movement, making the agent's actions more predictable.

Each cell in the Q table represents the expected future reward for taking a specific action in a particular state. The agent will learn over time which actions yield higher rewards, and a higher Q value indicates a better action for that state.

Next, you define the hyperparameters for the Q-learning algorithm, which control how the agent learns and explores the environment:

```
alpha = 0.1         # Learning rate
gamma = 0.99        # Discount factor
epsilon = 1.0       # Exploration rate
epsilon_decay = 0.995
epsilon_min = 0.01
episodes = 2000     # Number of games
```

The parameters in this code are as follows:

- `alpha` (learning rate) determines how much new knowledge overrides old knowledge. A higher value means faster learning but can be unstable, while a lower value is slower but more stable.
- `gamma` (discount factor) controls how much the agent values future rewards. A value close to 1 means the agent cares more about long-term rewards.
- `epsilon` (exploration rate) manages the balance between exploration and exploitation. Starting at 1.0, the agent explores randomly, and as epsilon decays toward 0.01, it increasingly exploits what it has learned.
- `epsilon_decay` gradually reduces epsilon to decrease randomness over time.
- `epsilon_min` ensures the agent never completely stops exploring.
- `episodes` is the number of games the agent will play to learn the optimal policy.

The following figure shows the layout of the FrozenLake environment that the agent will learn to navigate:

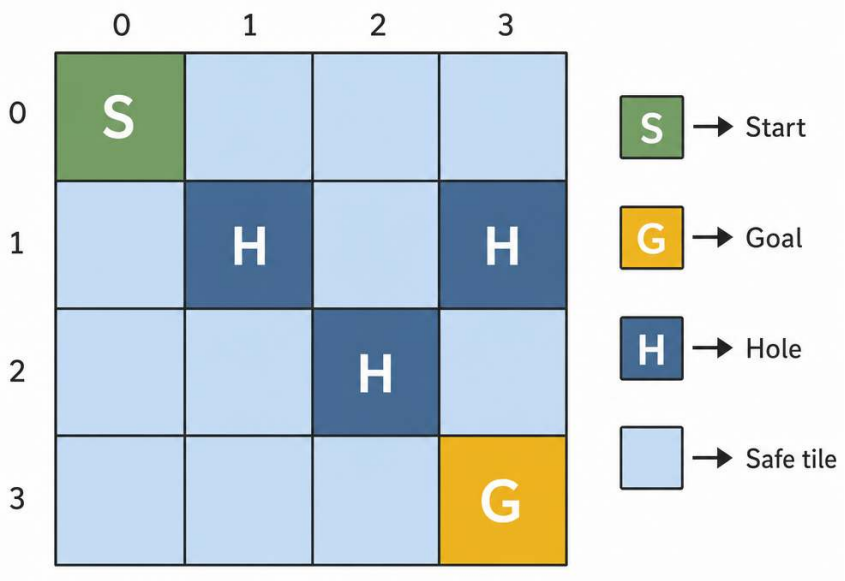


Figure 3.2: FrozenLake environment diagram

To prevent deprecation warnings from cluttering your output, you can import the warnings module and set a simple filter to ignore deprecation warnings.

The following code imports the warnings module and sets the filter:

```
import warnings
warnings.simplefilter(action='ignore', category=DeprecationWarning)
```

This approach is helpful if you are working with older versions of libraries and want to suppress unnecessary warning messages while running your agent.

Training the agent with Q-learning

To train the agent using the Q-learning algorithm, you iterate through each episode and update the Q-table based on the agent's actions and the rewards received from the environment.

The following code begins the training loop, initializing the state and done flag for each episode:

```
for episode in range(episodes):
    state = env.reset()[0]
    done = False
    while not done:
        # Choose action using epsilon-greedy strategy
        if np.random.rand() < epsilon:
            action = np.random.choice(action_size)  # Exploration
```

```
        else:
            action = np.argmax(Q_table[state])      # Exploitation
```

Within each episode, you use the epsilon-greedy strategy to choose actions. If a random number is less than epsilon, the agent explores by selecting a random action; otherwise, it exploits by choosing the action with the highest Q value for the current state.

TIP

Sidebar for the preceding code:
For `gym >=0.26`; for older versions, use: `state = env.reset()`

After selecting an action, you take the action in the environment and observe the resulting next state, reward, and whether the episode is done:

```
        # Take action and observe reward
        next_state, reward, done, _, _ = env.step(action)
```

TIP

Sidebar for the preceding code:
For `gym >=0.26`; for older versions, use: `next_state, reward, done, _ = env.step(action)`

You then update the Q-table using the Bellman equation, which incorporates the learning rate (`alpha`), the received reward, the discount factor (`gamma`), and the maximum Q value for the next state:

```
        # Update Q-table using Bellman equation
        Q_table[state, action] = Q_table[state, action] + alpha * (
            reward + gamma * np.max(Q_table[next_state]) - Q_table[state, action]
        )
```

This process allows the agent to learn the optimal actions to take in each state by balancing exploration and exploitation as it interacts with the environment.

This means if your action leads to a better state, you increase the Q value. If the action leads to a bad state, you decrease the Q value.

Next, set the current state to the next state so that the agent can continue learning from its new position.

The following code demonstrates how you update the state and decay epsilon to reduce exploration over time:

```
        state = next_state

    # Decay epsilon to reduce exploration over time
    epsilon = max(epsilon * epsilon_decay, epsilon_min)
```

Here, you gradually reduce the value of epsilon after each episode. This causes the agent to explore less and exploit more as training progresses.

Once you have completed the training loop, print a message to indicate that training is complete:

```
print("Training complete!")
```

This prints:

```
Training complete!
```

At the end of training, the agent learns the best path to the goal by adjusting its actions based on the rewards it receives and by reducing exploration over time. Your next step is to train an RL agent using deep Q-networks.

Training the agent with a deep Q-network

Let's start by importing the required libraries for building and training a DQN agent in PyTorch.

The following code imports PyTorch, neural network components, optimizers, random sampling, NumPy for mathematical operations, the Gym reinforcement learning library, and deque for storing past experiences:

```
import torch
import torch.nn as nn
import torch.optim as optim
import random
import numpy as np
from collections import deque
import gym
import gymnasium as gym
```

The following code imports the warnings module and sets the filter to ignore deprecation warnings:

```
import warnings
warnings.simplefilter(action='ignore', category=DeprecationWarning)
```

This approach is helpful if you are working with older versions of libraries and want to suppress unnecessary warning messages while running your agent.

Next, you create the CartPole environment and extract the state and action sizes:

```
env = gym.make("CartPole-v1")
state_size = env.observation_space.shape[0]
action_size = env.action_space.n
```

The `CartPole-v1` environment has four states: cart position, cart velocity, pole angle, and pole angular velocity. The agent can take two possible actions: move left or move right.

The following figure shows the four state variables of the CartPole environment:

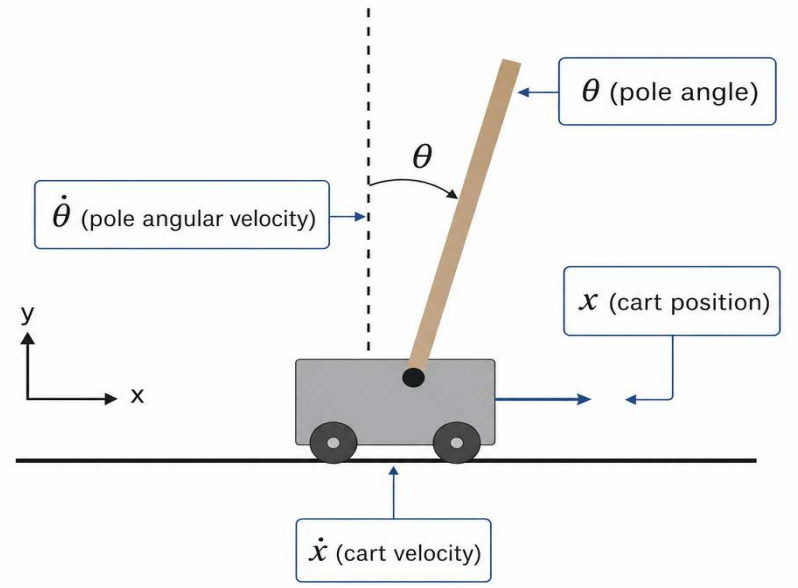


Figure 3.3: Cart-pole state variables

The output layer uses the number of neurons equal to the action size, and the activation function applied here is ReLU, which introduces non-linearity to help the network learn.

The following code defines the DQN class using PyTorch's `nn.Module`, which consists of three fully connected layers:

```
class DQN(nn.Module):
    def __init__(self, state_size, action_size):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(state_size, 24)
        self.fc2 = nn.Linear(24, 24)
        self.fc3 = nn.Linear(24, action_size)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)
```

To create your model and optimizer, you instantiate the model and define the optimizer and loss function as follows:

The following code creates the DQN model, sets up the `Adam` optimizer with a learning rate of `0.001`, and defines the mean squared error loss function:

```
model = DQN(state_size, action_size)
optimizer = optim.Adam(model.parameters(), lr=0.001)
loss_function = nn.MSELoss()
```

Adam automatically adjusts the weights to minimize loss, and the mean squared error loss helps the model learn by reducing prediction errors.

You now initialize your experience replay memory to store past experiences, including the state, action, reward, next state, and done flag. The memory is set to a maximum size of 2,000, so older experiences are removed once it is full. Experience replay is used to break the correlation between consecutive actions, allowing the model to learn from a diverse set of past experiences.

The following code initializes the experience replay memory and sets the key hyperparameters for training your DQN agent:

```
memory = deque(maxlen=2000)

gamma = 0.95          # Discount factor
epsilon = 1.0         # Exploration rate
epsilon_min = 0.01
epsilon_decay = 0.995
batch_size = 64
episodes = 1000
```

You set the discount factor `gamma` to `0.95`, which determines how much future rewards are valued. The exploration rate `epsilon` starts at 1.0 to ensure the agent explores randomly at first and gradually decays by multiplying by `epsilon_decay` after each episode, but never goes below `epsilon_min`. The `batch_size` is set to 64, meaning 64 experiences are sampled per training step, and the agent will train for `1000` episodes.

Next, you need to store each experience so the agent can learn from past state, action, reward, and next state transitions. The following function appends each experience to the memory:

```
def remember(state, action, reward, next_state, done):
    memory.append((state, action, reward, next_state, done))
```

The agent learns by replaying these stored experiences. To train from the experience replay, define a function that samples a random batch from memory and updates the model:

```
def replay():
    if len(memory) < batch_size:
        return

    minibatch = random.sample(memory, batch_size)
    for state, action, reward, next_state, done in minibatch:
        target = reward
        if not done:
            target += gamma * torch.max(model(torch.FloatTensor(next_state))).item()

        output = model(torch.FloatTensor(state))[action]

        # Ensure target has the same shape as output
        loss = loss_function(output.unsqueeze(0), torch.tensor([target], dtype=torch.float32))

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

If there are not enough experiences in memory, the training step is skipped. Otherwise, a random batch is sampled, the target Q value is computed, and backpropagation is performed to update the model weights.

Now, run the training loop for the specified number of episodes. For each episode, reset the environment, initialize the state, and iterate until the

episode is done. Choose actions using the epsilon-greedy strategy, store each experience, and update the agent using experience replay:

```
for episode in range(episodes):
    state = env.reset()[0]
    done = False
    total_reward = 0

    while not done:
        if np.random.rand() < epsilon:
            action = random.randrange(action_size)  # Explore
        else:
            action = torch.argmax(model(torch.FloatTensor(state))).item()  # Exploit

        next_state, reward, done, _, _ = env.step(action)  # For gym>=0.26; for older versions, use:
next_state, reward, done, _ = env.step(action)
        remember(state, action, reward, next_state, done)
        state = next_state
        total_reward += reward

        replay()
        epsilon = max(epsilon * epsilon_decay, epsilon_min)

    print(f"Episode {episode+1}, Total Reward: {total_reward}")

print("Training complete!")
```

This loop ensures the agent explores and exploits actions, stores each transition in memory, and trains using experience replay to improve performance over time.

As the training runs, you see the output for each episode, displaying the episode number and the total reward accumulated by the agent:

```
Episode 542, Total Reward: 1081.0
Episode 543, Total Reward: 1471.0
Episode 544, Total Reward: 1111.0
Episode 545, Total Reward: 947.0
Episode 546, Total Reward: 890.0
Episode 547, Total Reward: 778.0
Episode 548, Total Reward: 596.0
Episode 549, Total Reward: 356.0
Episode 550, Total Reward: 2420.0
Episode 551, Total Reward: 1266.0
Episode 552, Total Reward: 1498.0
Episode 553, Total Reward: 2123.0
Episode 554, Total Reward: 1339.0
Episode 555, Total Reward: 1769.0
```

```
Episode 556, Total Reward: 2018.0
Episode 557, Total Reward: 2122.0
Episode 558, Total Reward: 2424.0
```

The training continues for 1,000 episodes, and the agent's performance improves as the total reward increases and stabilizes over time. At the end of the training, the output confirms completion and shows the final reward for the last episode:

```
Episode 985, Total Reward: 420.0
Episode 986, Total Reward: 18.0
Episode 987, Total Reward: 194.0
Episode 988, Total Reward: 230.0
Episode 989, Total Reward: 203.0
Episode 990, Total Reward: 228.0
Episode 991, Total Reward: 209.0
Episode 992, Total Reward: 231.0
Episode 993, Total Reward: 235.0
Episode 994, Total Reward: 212.0
Episode 995, Total Reward: 277.0
Episode 996, Total Reward: 216.0
Episode 997, Total Reward: 13.0
Episode 998, Total Reward: 203.0
Episode 999, Total Reward: 17.0
Episode 1000, Total Reward: 183.0
Training complete!
```

You have now completed the DQN training process, where the agent learns to maximize its rewards through repeated episodes, using experience replay and the epsilon-greedy strategy to balance exploration and exploitation.

Summary

Neural networks learn by repeating a cycle of forward pass, loss calculation, and gradient-based weight updates. CNNs suit images because filters detect local patterns and pooling shrinks the data while keeping what matters. For text, transformers improve on RNNs by using self-attention to relate all words in a sequence at once. Normalizing inputs helps training, and more epochs raise accuracy only until the model starts to overfit, so validation results matter as much as training results. TensorFlow with Keras gets you a working model in fewer lines, while PyTorch makes you write the training loop yourself and gives you more control over it.

Reinforcement learning replaces labeled data with rewards, so the agent improves through trial and error. Epsilon-greedy exploration balances trying new actions against using what the agent already knows, and the learning rate and discount factor decide how fast it learns and how far ahead it looks. A Q-table works while every state can be listed, and a deep Q-network takes over when it cannot. Experience replay then keeps training stable by breaking the link between consecutive steps.

In the next chapter, you will turn to the foundations of quantum computing, using everyday analogies to compare classical and quantum computers, building and simulating a basic quantum experiment in Python, and exploring how quantum states, probability, and wave functions work.

4

Understanding the Foundations of Quantum Computing

The previous chapters introduced artificial intelligence, set up your tools, and showed how classical machines learn from data. Quantum computing asks a bolder question: what if the machine itself worked differently? The field can feel like a mystery, partly because its vocabulary is new and partly because quantum behavior defies everyday intuition. This chapter makes those ideas approachable. It begins with food analogies and everyday examples, and then moves step by step toward experiments you run yourself and the mathematics that explains them.

By the end of this chapter, you will be able to explain the difference between classical bits and qubits, describe superposition, entanglement, and measurement in your own words, and read a basic qubit state written in ket notation. You will have run your first quantum experiment in Python, seen how quantum results are visualized, and checked that the mathematics predicts what the simulator returns. Above all, you will gain the confidence to keep exploring the quantum world on your own. In this chapter, we will cover the following topics:

- Understanding quantum computing through analogies
- Running your first quantum experiment in Qiskit
- Describing qubits, ket notation, and quantum gates
- Exploring probability, wave functions, and the Bloch sphere
- Verifying quantum measurement in Python

Technical requirements

To follow the examples in this chapter, you will need the following:

- Python 3.12 or later, with `pip`, installed earlier in this book.
- A Jupyter notebook environment, set up earlier in this book.
- The libraries `qiskit`, `qiskit-aer`, and `matplotlib`.

Install all the required libraries with the following command:

```
pip install qiskit qiskit-aer matplotlib
```

The code files for this chapter are available at: https://github.com/PacktPublishing/Quantum-Computing-for-AI-with-Python/blob/main/Chapter4.ipynb

Understanding quantum computing through analogies

You already know what AI is and the many things you can do with it. Quantum computers are far less familiar, and even if you know a little about them, the topic can still feel confusing because it is so new. So imagine a magic kitchen, where ice cream flavors mix in impossible ways, fries communicate across the universe, and burritos hold many possibilities at once. In this section, you will use these tasty analogies to build intuition before any mathematics appears. Each analogy builds on the one before it, starting with the basic unit of information and ending with what happens when you finally look at a result.

Comparing bits and qubits

Start with the most basic comparison: a classical computer is like a sandwich, and a quantum computer is like a burrito. A classical computer works with bits, which are like choosing between two fixed options, such as a sandwich with either peanut butter or jelly, but never both at the same time.

A quantum computer uses qubits, which are more like a burrito filled with rice, beans, and cheese, all mixed together in one bite. Instead of picking a single option, a qubit can carry a blend of possibilities. That blend has a name, and the next analogy shows what it looks like.

Explaining superposition

Superposition is the name for that blend, and it works like an ice cream swirl. If a classical bit is like choosing either vanilla or chocolate, a qubit in superposition is like swirled vanilla and chocolate soft serve, which gives you both flavors in one spoonful. In a quantum computer, this means a qubit can be zero and one at the same time, in the same way that the swirl lets you taste vanilla and chocolate together.

Superposition describes a single qubit. Things become more interesting when two qubits are linked, and that is where the fries come in.

Visualizing entanglement

Entanglement is like having a pair of synchronized fries. Imagine you and a friend each order a plate of fries at different restaurants. If the fries are entangled, the two plates stay mysteriously in step: when you pick up a fry and find it dipped in ketchup, your friend's fry turns out to be dipped in ketchup too, even though nobody coordinated the plates. This is how entanglement works. Two qubits are connected so that their measurement results stay linked, no matter the distance between them.

> NOTE
>
> Entanglement does not let you send a message faster than light. The matching results only become visible when you and your friend compare notes afterward. This link is what makes entanglement valuable in quantum cryptography and secure communications, rather than as a way to transmit information instantly.

Entanglement links qubits to each other. The next analogy asks what a computer can do with many possibilities at once.

Exploring quantum parallelism

Quantum parallelism is like going to a buffet instead of ordering a single dish. A regular computer is like ordering one dish at a time: you pick a pizza, wait for it, eat it, and then order another dish. A quantum computer is like a buffet where you can put everything on your plate at once: pizza, sushi, tacos, pasta, anything. This lets quantum computers work on many possibilities at the same time, which makes them much faster for certain tasks.

There is a catch, though. When the meal ends, you can taste only one dish. Quantum algorithms work by arranging the possibilities so that the wrong answers cancel out and the right answer is the one you are most likely to be served. That is why quantum computers are faster only for specific problems, and not for every task. The moment of being measured has an analogy of its own.

Describing quantum measurement

Quantum measurement is like opening a mystery box cake. Imagine a cake hidden inside a box. Before you open it, the cake could be chocolate, vanilla, or strawberry, all at once, in a superposition of flavors. The moment you open the box, you discover the actual flavor, and all the other possibilities vanish. This is what happens when you measure a qubit: it collapses into a definite state, just like revealing the cake's flavor.

The odds are part of the recipe. If the batter is mixed so that two flavors are equally likely, opening many boxes gives you each flavor about half of the time, although you cannot predict what any single box contains.

Taken together, these analogies show that quantum computers are not simply faster versions of classical computers. They approach problems differently, like a chef who can prepare thousands of recipes at once instead of cooking them one by one, although only one dish reaches your table. To see this in action, you will build a quantum chocolate versus vanilla machine in the next section and watch it make its choices.

Running your first quantum experiment in Qiskit

In this section, you will turn the ice cream swirl into a working experiment: a single qubit that you put into superposition and then measure, many times over. You will create the circuit, apply a gate, measure the qubit, run the simulation, and chart the results.

Getting started with Qiskit

Qiskit is an open-source quantum computing framework developed by IBM. It lets you write quantum programs, test them on simulators, and run them on real quantum hardware, and it provides tools for building quantum circuits

and executing quantum algorithms. It works on Windows, macOS, and Linux. If you installed `qiskit`, `qiskit-aer`, and `matplotlib` from the technical requirements, you are ready to go. Otherwise, run the installation command from the technical requirements in your terminal first.

Building a one-qubit circuit

Start by importing the three tools you need for the experiment. The following code imports the building blocks:

```
from qiskit import QuantumCircuit
from qiskit_aer import Aer
import matplotlib.pyplot as plt
```

The following three imports each play a distinct role:

- `QuantumCircuit` is a tool for designing quantum experiments. Think of it as the blueprint of an experiment in a lab.
- `Aer` provides simulators that mimic a real quantum computer on your own machine.
- `matplotlib.pyplot` visualizes the results using charts and plots.

The following code creates the circuit, puts the qubit into superposition, and measures it:

```
# Step 1: Create a circuit with 1 qubit and 1 classical bit
qc = QuantumCircuit(1, 1)

# Step 2: Put the qubit in superposition (like a swirled ice cream mix)
qc.h(0)

# Step 3: Measure the qubit (forcing it to choose chocolate or vanilla)
qc.measure(0, 0)
```

The call `QuantumCircuit(1, 1)` creates a circuit with one qubit and one classical bit. The classical bit is needed because a measurement produces an ordinary 0 or 1, and that result has to be stored somewhere. Every qubit in Qiskit starts in the state zero.

The call `qc.h(0)` applies the Hadamard gate to qubit 0. It is the gate that stirs the swirl: it turns the definite zero into an even blend of zero and one, so the qubit has not yet chosen a flavor. You will see the mathematics behind this gate shortly.

Finally, `qc.measure(0, 0)` measures qubit 0 and stores the result in classical bit 0. This is the moment the mystery box is opened. The qubit must give up its blend and settle on chocolate (zero) or vanilla (one).

Simulating the experiment with shots

A **circuit** is only a blueprint until you run it. Qiskit Aer provides a simulator that mimics quantum behavior on a classical computer. It simulates ideal results without any noise, which makes it a convenient place to learn.

The following code selects the simulator, runs the circuit 100 times, and collects the results:

```
# Step 4: Simulate the experiment
simulator = Aer.get_backend("aer_simulator")
job = simulator.run(qc, shots=100)
result = job.result()
counts = result.get_counts(qc)
print(counts)
```

A single measurement gives just one random outcome, which tells you very little. So you repeat the experiment, and each repetition is called a shot. Here, `shots=100` runs the circuit 100 times. The variable `counts` is a dictionary that records how many times each outcome appeared, with the keys '0' and `'1'` standing for the two outcomes. The following output shows one run:

```
{'0': 57, '1': 43}
```

Your numbers will differ, and that is expected.

Visualizing the flavor counts

To chart the results, map the two outcomes to flavors and draw a bar chart with `matplotlib`. The following code does this:

```
# Step 5: Display the results
labels = ["Chocolate (0)", "Vanilla (1)"]
values = [counts.get("0", 0), counts.get("1", 0)]

plt.bar(labels, values, color=["saddlebrown", "gold"])
plt.xlabel("Flavor")
plt.ylabel("Count")
plt.title("Quantum Ice Cream Machine Results")
plt.show()
```

The `counts.get` calls read the count for each outcome and fall back to zero if an outcome never appeared, so both bars are always drawn in the right order and color. The `plt.xlabel` and `plt.ylabel` calls label the axes, and `plt.title` names the chart.

The following figure shows the flavor counts from the experiment:

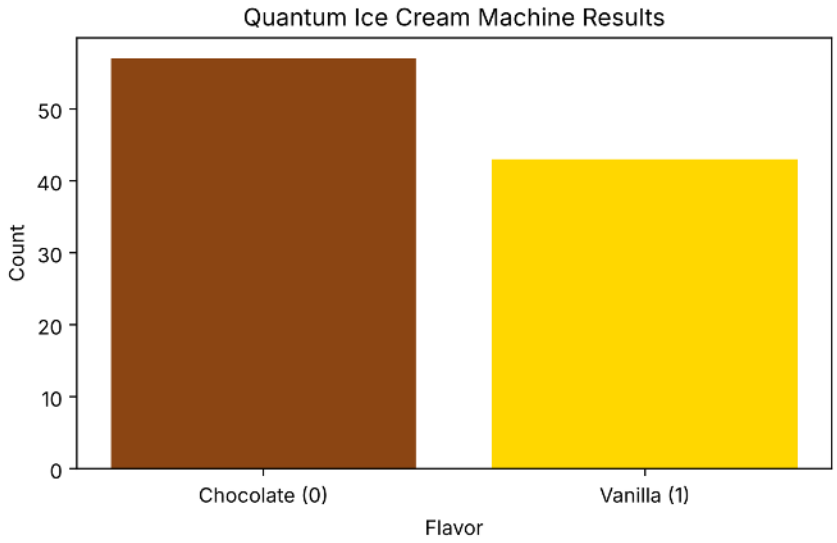


Figure 4.1: Quantum ice cream machine results. Chocolate and vanilla counts from 100 shots

The bars are close to an even split, but not identical. Run the cells again, and the numbers will change, sometimes with vanilla ahead and sometimes with chocolate. Two lessons stand out. First, every individual shot is unpredictable, like opening a single mystery box. Second, the pattern across many shots is stable and centers on 50/50, as if the odds were part of the recipe.

The Hadamard gate fixed those odds, but how does it do that, and why exactly half? Answering this requires the precise language that quantum computing uses to describe qubits, which comes next.

Describing qubits, ket notation, and quantum gates

The analogies and the experiment gave you intuition. This section turns that intuition into precise vocabulary. It begins with how classical and quantum computing store and process information, and then moves on to the notation and operations you need to describe a qubit.

Contrasting classical and quantum information

In classical computing, which includes everyday laptops and supercomputers, information is stored in bits, which are binary digits that are either 0 or 1. A classical bit is always in exactly one of these two states, which makes classical computation deterministic: the same input always produces the same output. Classical computers also work step by step, and some problems take an impractical amount of time to solve this way. Examples include factoring large numbers, simulating molecules for drug discovery, and solving large optimization problems.

Quantum computing uses quantum bits, or qubits, which can be in a superposition of 0 and 1 until they are measured. This lets quantum computers approach certain problems in a fundamentally different way, as the buffet analogy suggested. The following table summarizes the key differences between classical and quantum computing:

Feature	Classical computing	Quantum computing
Unit of information	Bit (0 or 1)	Qubit (a superposition of 0 and 1)
Reading the result	Always the stored value	0 or 1, chosen with probabilities set by the state
Operations	Logic gates such as AND and NOT	Quantum gates such as H and CNOT
Speed advantage	Baseline	Potentially large for specific problems, not for every task
Key principle	Boolean logic	Quantum mechanics

Table 4.1: Comparing classical computing with quantum computing

One row deserves extra care: the speed advantage. A quantum advantage is not automatic. It appears only for specific problems, which is why quantum computers are expected to complement classical machines rather than replace them. To describe how a qubit behaves, you first need a way to write its state down.

Writing quantum states in ket notation

Quantum states are written in ket notation, also called Dirac notation. In this notation, $|0\rangle$ and $|1\rangle$ are the two basic states of a qubit, the quantum counterparts of the classical bits 0 and 1. The vertical bar and the angle bracket together form a ket, and the label inside tells you which state the qubit is in. A ket represents a state vector, which is a list of numbers that fully describes the state.

You can read $|0\rangle$ as "the qubit is in state zero," and every qubit in Qiskit starts there. A superposition combines the two basic states and holds a probability distribution over them, as the next definition shows.

Defining a qubit and its state

A qubit is the basic unit of quantum information. Unlike a classical bit, it can exist in a superposition of $|0\rangle$ and $|1\rangle$. The following equation writes the state of any qubit as a combination of the two basic states:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

Here, α and β are complex numbers called probability amplitudes. They are not probabilities themselves. The probability of each outcome is the squared magnitude of its amplitude, so $|\alpha|^2$ is the probability of measuring 0 and $|\beta|^2$ is the probability of measuring 1. Because the qubit must give one of the two outcomes, the probabilities must add up to 1, as the following equation expresses:

$$|\alpha|^2 + |\beta|^2 = 1$$

This is called the normalization condition. The amplitudes are complex numbers rather than plain probabilities because the sign and angle of an amplitude, called its phase, affect how superpositions combine. A state describes where a qubit is, but programs also need to change it, and the tools for doing so are quantum gates.

Manipulating qubits with quantum gates

Quantum gates manipulate qubits just as classical logic gates, such as AND and NOT, manipulate bits. The following five gates are the ones you will meet most often:

- The Hadamard gate (H) turns a definite state, such as $|0\rangle$, into an equal superposition of $|0\rangle$ and $|1\rangle$.
- The Pauli X gate (X) flips $|0\rangle$ to $|1\rangle$ and $|1\rangle$ to $|0\rangle$, which makes it the quantum counterpart of the classical NOT gate.
- The CNOT gate (controlled NOT) flips a target qubit only when a control qubit is in the state $|1\rangle$, and it is the gate used to create entanglement between qubits.
- The phase gates (S, T, and Z) rotate the phase of the qubit state without changing the probabilities of the two outcomes.
- The Toffoli gate (CCNOT) flips a target qubit only when two control qubits are both in the state $|1\rangle$, which makes it a reversible version of the classical AND gate.
- The Hadamard gate is the one you used in the experiment. The following equation shows what it does to the state $|0\rangle$:

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

Both amplitudes equal $1/\sqrt{2}$, and squaring each of them gives 1/2. This means the qubit has a 50% chance of collapsing to 0 and a 50% chance of collapsing to 1 when measured, which explains the even split in your experiment. Applied to $|1\rangle$, the gate gives $1/\sqrt{2}$ ($|0\rangle$ - $|1\rangle$), which differs only by a minus sign. That sign is a phase. It does not change the odds of measuring 0 or 1, but it changes how the qubit behaves when later gates act on it.

You have now met the notation, the state, and the gates. What remains is the physics that gives these amplitudes their meaning, and the next section explores it through probability, wave functions, and a picture of the qubit on a sphere.

Exploring probability, wave functions, and the Bloch sphere

Quantum mechanics is the foundation of quantum computing. It introduces ideas that differ sharply from classical physics, such as probability that cannot be removed, superposition, and measurement that changes the state being measured. This section follows a path from the physics to the picture. It first

shows how probability arises from the wave function, then how the state of a qubit becomes a point on a sphere, and finally what measurement does to that point.

Understanding probability and wave functions

In classical mechanics, probability reflects what you do not know. A coin flip comes out heads or tails, but in principle the outcome is fixed by the details of the throw. In quantum mechanics, probability is built into nature itself, and it arises from the wave function, written ψ, which describes the quantum state.

A wave function, $\psi(x)$, describes the state of a quantum system and holds all the information about a particle, including its position and momentum. It is a complex function, which means it can have imaginary parts. The probability of finding a particle at a position x, comes from the squared magnitude of the wave function, as the following equation shows:

$$P(x) = |\psi(x)|^2$$

Strictly speaking, $P(x)$ is a probability density, so the probability of finding the particle between two positions a and b is the area under the curve between them:

$$\int_a^b |\psi(x)|^2 \, dx$$

Because the particle must be found somewhere, the total probability over all positions must equal 1, as the following equation expresses:

$$\int_{-\infty}^{\infty} |\psi(x)|^2 \, dx = 1$$

The qubit state from the previous section follows the same rule, with two outcomes instead of a continuous range of positions. The squared magnitudes of the amplitudes are the probabilities, and they must add up to 1. Whether the possibilities are positions along a line or the two outcomes of a qubit, the recipe is the same: square the amplitude to get the probability. For a single qubit, there is a way to see all of this at once, and it is called the Bloch sphere.

Representing a qubit on the Bloch sphere

The Bloch sphere is a geometric visualization of all the pure states of a single qubit. It shows the state of a qubit as a point on the surface of a sphere, and it helps you see how quantum gates rotate and transform qubit states in three-dimensional space. The state $|0\rangle$ sits at the north pole, and the state $|1\rangle$ sits at the south pole. Every other state lies somewhere between them. The following equation describes a general qubit state in terms of two angles that locate a point on the sphere:

$$\left|\psi\right\rangle = cos\frac{\theta}{2}\left|0\right\rangle + e^{i\phi}sin\frac{\theta}{2}|1\rangle$$

The following two angles play different roles:

- θ (theta) determines the probability of measuring 0 or 1.
- ϕ (phi) determines the phase of the state.

Comparing this form with the general state shows that $\alpha=\cos(\theta/2)$ and $\beta=e^{i\phi}\sin(\theta/2)$. The probabilities are therefore $P(0)=(\cos)^2(\theta/2)$ and $P(1)=(\sin)^2(\theta/2)$. At the north pole, $\theta=0$ and the outcome is always 0. At the south pole, $\theta=\pi$ and the outcome is always 1. Points on the equator, where $\theta=\pi/2$, give an even 50/50 split. The angle ϕ only moves the point around the sphere, so it leaves these probabilities unchanged, but it matters when gates later combine qubits.

The following figure shows the Bloch sphere with a state located by the angles θ and φ:

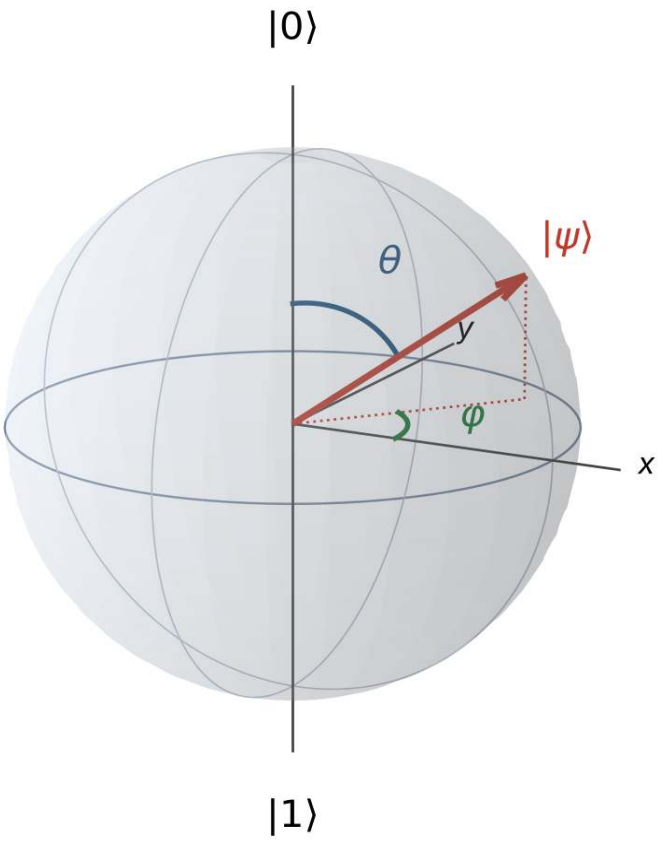


Figure 4.2: The Bloch sphere. The state |0⟩ is at the north pole and |1⟩ is at the south pole. The angles ϑ and φ locate the state |ψ⟩

In this picture, gates rotate the point. The X gate swaps the two poles, and the Hadamard gate moves the north pole onto the equator, along the x-axis. You will see exactly this in the final experiment of this chapter.

Measuring a quantum state

When you measure a qubit in the state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, it collapses into either $|0\rangle$ or $|1\rangle$. The probability of each outcome is given by the following equations:

$$P(0) = |\alpha|^2, P(1) = |\beta|^2$$

After the measurement, the qubit is no longer in a superposition. It stays in the state you found, so measuring it again right away gives the same result.

To see how unequal amplitudes behave, suppose a qubit is prepared in the following state:

$$|\psi\rangle = (\sqrt{3})/2|0\rangle + 1/2|1\rangle$$

Squaring the amplitudes gives $P(0) = \frac{3}{4}$ and $P(1) = \frac{1}{4}$, so you would expect about 750 zeros and 250 ones in 1,000 shots. The state prepared by the Hadamard gate is the special case where both amplitudes equal 1/√2, which is why your earlier experiment came out close to an even split.

You now have the full chain. A qubit state has amplitudes, measurement turns amplitudes into probabilities, and the Bloch sphere shows the state geometrically. The last step is to check this chain against the simulator.

Verifying quantum measurement in Python

In this section, you will test the theory you have just learned. First, you will repeat the experiment with more shots to see the probabilities emerge. Then you will inspect the state before measurement to confirm the amplitudes directly.

Repeating the measurement with more shots

Qiskit includes a built-in tool, `plot_histogram`, that turns a counts dictionary into a histogram. If you are continuing in the same notebook, the circuit `qc` and the `simulator` already exist, so you only need to import the tool and run 1,000 shots. The following code runs the circuit and plots the histogram:

```
from qiskit.visualization import plot_histogram

job = simulator.run(qc, shots=1000)
counts = job.result().get_counts(qc)
print(counts)
plot_histogram(counts)
```

The following output shows the counts from one run:

```
{'0': 530, '1': 470}
```

Run the cell again, and the numbers change. For example, a second run produced the following counts:

```
{'1': 505, '0': 495}
```

Notice that the order of the keys can differ between runs, which is why you should always read a count by its key, as the chart code earlier did.

NOTE

In a Jupyter notebook, leave `plot_histogram(counts)` as the last line of the cell so that the notebook displays the figure. Qiskit closes its figures automatically in notebooks, so calling `plt.show()` after it displays nothing. In a standalone Python script, call `plt.show()` after `plot_histogram(counts)` instead.

The following figure shows the histogram for the first run:

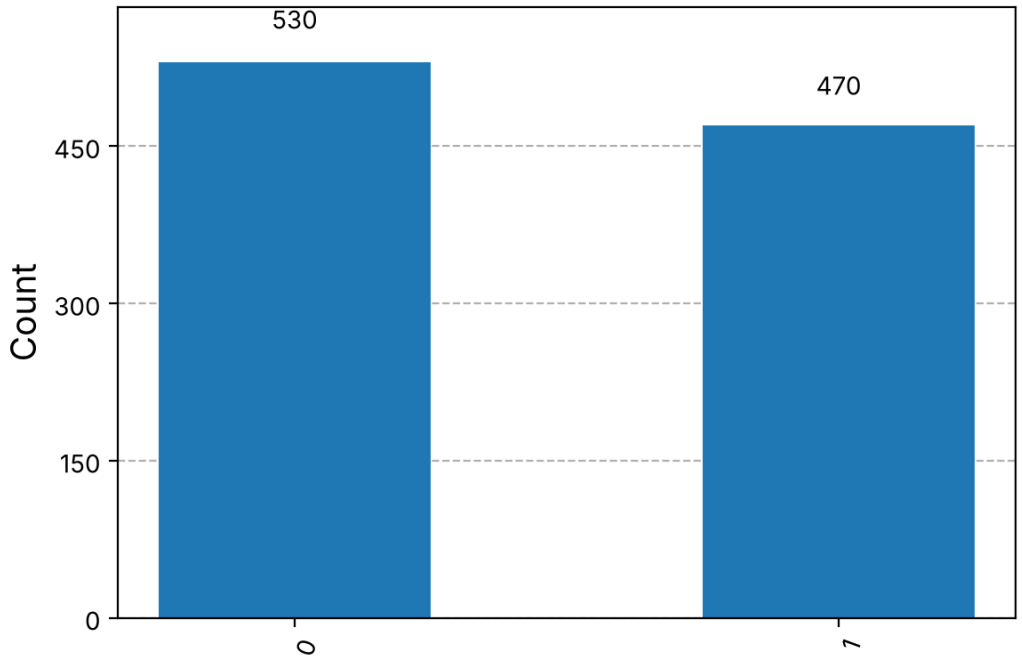


Figure 4.3: Histogram of 1,000 measurements of a qubit in equal superposition

Compare the two runs. Neither is exactly 500 and 500, and they differ from each other, yet both sit close to an even split. For 1,000 shots, the typical swing is about 16 counts in either direction, so any result from roughly 470 to 530 is normal. With only 100 shots, as in the first experiment, the typical swing is about 5 counts, which is why the ice cream bars looked less balanced.

This is the probabilistic nature of quantum measurement in action. Unlike a classical program, the same circuit can give different results on each run, but the distribution of the results is stable and matches the 50% predicted by the amplitudes. The histogram shows what happens after measurement. To confirm the numbers behind it, you can look at the qubit before it is measured.

Inspecting the quantum state before measurement

A simulator lets you do something real hardware cannot: look at the state itself before any measurement collapses it. The following code builds the same circuit without the measurement, extracts its state vector, and displays it on a Bloch sphere:

```
from qiskit import QuantumCircuit
from qiskit.quantum_info import Statevector
from qiskit.visualization import plot_bloch_multivector

qc_state = QuantumCircuit(1)
qc_state.h(0)

state = Statevector.from_instruction(qc_state)
print(state)
print(state.probabilities())
plot_bloch_multivector(state)
```

This circuit has no classical bit and no measurement, so the state vector shows the amplitudes themselves. The call `Statevector.from_instruction` calculates the state that the circuit produces, and probabilities() squares each amplitude for you. The following output shows the result:

```
Statevector([0.70710678+0.j, 0.70710678+0.j],
            dims=(2,))
[0.5 0.5]
```

The two amplitudes are 0.7071, which is 1/√2, and the `+0.j` shows that their imaginary parts are zero. Squaring them gives 0.5 each, which is the 50/50 split predicted by the equations and observed in the histogram. The following figure shows the same state on the Bloch sphere:

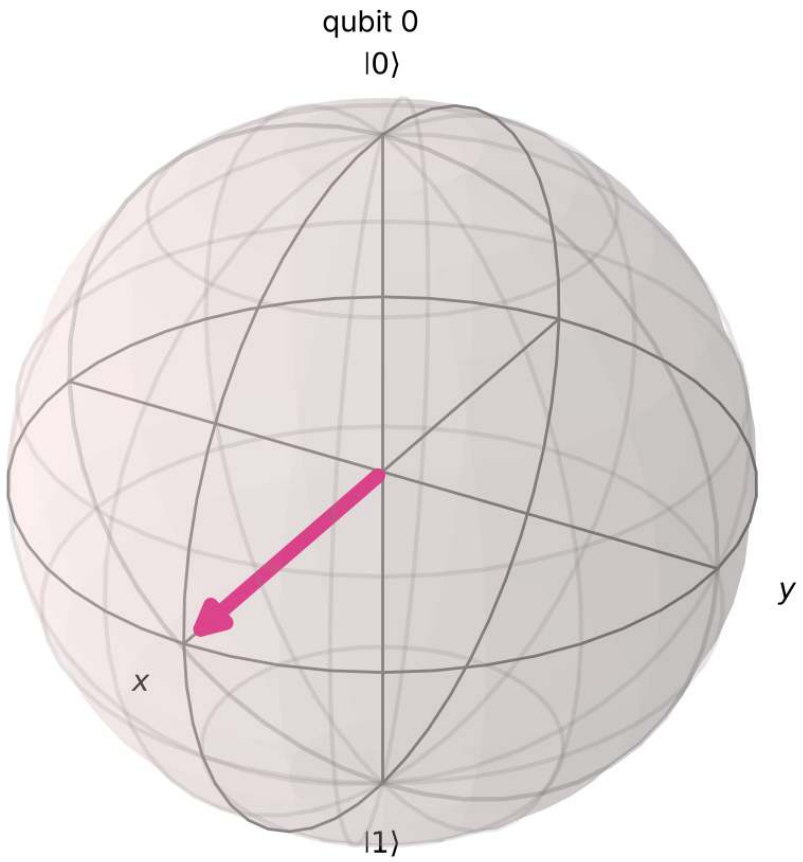


Figure 4.4: Bloch sphere showing the qubit after the Hadamard gate. The vector lies on the equator, along the x-axis

The arrow points to the equator, along the x-axis, which is where the Bloch sphere places an equal superposition with no phase. Everything in this chapter now connects. The swirl in the analogy, the even counts in the histogram, the amplitudes 1/√2 and the arrow on the equator are four views of the same qubit.

Summary

Quantum computing replaces the bit with the qubit, which holds a superposition of 0 and 1 described by two amplitudes. Squaring those amplitudes gives the probability of each result, and measurement collapses the qubit into a single definite outcome. That is why the same circuit gives different counts on each run, yet the overall distribution stays predictable. Quantum gates such as the Hadamard gate change the amplitudes, and the Bloch sphere shows each gate as a rotation. Entanglement links qubits so that

their results stay correlated without allowing instant messaging, and quantum parallelism helps only when an algorithm arranges for the right answer to become likely.

In practice, you built and simulated a one-qubit circuit in Qiskit, read its results as shot counts, and confirmed the theory by comparing the counts with the state vector. Remember that quantum advantage applies to specific problems and not to every task.

In the next chapter, we will put these foundations to work by learning how quantum gates operate, composing them into circuits in Qiskit, and exploring foundational algorithms such as Grover's and Shor's.

5

Building Quantum Circuits and Algorithms

In the previous chapter, you built the foundations of quantum computing: qubits, superposition, entanglement, and measurement. Those ideas explain what makes a quantum computer different, but they only become useful when you can put them to work. To do that, you need a way to control qubits and combine those controls into programs. That is the job of quantum gates and circuits.

This chapter turns concepts into practice. You will learn how individual gates change a qubit, how gates are assembled into circuits, and how to run those circuits on a simulator and read the results. You will then see how the same building blocks power two landmark quantum algorithms, and you will implement one of them yourself in Qiskit. By the end, you will be able to read, build, and run simple quantum circuits with confidence. In this chapter, we will cover the following topics:

- Exploring fundamental quantum gates and their operations
- Composing quantum circuits from gates
- Implementing basic quantum circuits in Qiskit
- Exploring foundational quantum algorithms
- Implementing Grover's algorithm with Qiskit

Technical requirements

To follow the examples in this chapter, you need the following installed and working:

- Python 3.12 or later
- Jupyter Notebook or JupyterLab
- Qiskit, Qiskit Aer, Matplotlib, and pylatexenc

You can install all the libraries with a single command:

```
pip install qiskit qiskit-aer matplotlib pylatexenc
```

The `pylatexenc` library is needed to draw circuits with `qc.draw('mpl')`. The code files for this chapter are available at the book's GitHub repository: https://github.com/PacktPublishing/Quantum-Computing-for-AI-with-Python/blob/main/Chapter5.ipynb

Exploring fundamental quantum gates and their operations

Quantum circuits are the foundation of quantum computing. We use quantum circuits to manipulate qubits in ways that are different from classical circuits. Instead of classical logic gates such as AND, OR, and NOT, we work with quantum gates that perform unique operations on qubits. In this section, we move from the Pauli gates to the Hadamard and CNOT gates, the building blocks you will use to create superposition and entanglement.

Understanding Pauli gates: X, Y, and Z

Quantum gates are unitary operations that transform qubits. Unlike classical logic gates, quantum gates are reversible. Let's start by looking at the Pauli gates: X, Y, and Z.

The Pauli gates perform fundamental quantum operations on qubits. The Pauli-X gate, or X gate, is equivalent to the classical NOT gate: it flips the state from $|0\rangle$ to $|1\rangle$ and vice versa. The matrix representation for the X gate is:

```
X = [0 1]
    [1 0]
```

Its effect on qubits is as follows:

$$X|0\rangle = |1\rangle, X|1\rangle = |0\rangle$$

This equation shows that applying X to the $|0\rangle$ gives $|1\rangle$, and applying X to $|1\rangle$ gives $|0\rangle$, just like a NOT operation.

The Pauli-Y gate, or Y gate, combines the X and Z gates with an imaginary unit i. Its matrix representation is:

```
Y = [0 -i]
    [i  0]
```

The effect on qubits is:

$$Y|0\rangle = i|1\rangle, Y|1\rangle = -i|0\rangle$$

When you apply Y to $|0\rangle$, you get i times $|1\rangle$, and Y applied to $|1\rangle$ gives minus i times $|0\rangle$.

The Pauli-Z gate, or Z gate, performs a phase flip on the qubit without changing its probability distribution. The matrix representation for Z is:

```
Z = [1  0]
    [0 -1]
```

The effect on qubits is:

$$Z|0\rangle = |0\rangle, Z|1\rangle = -|1\rangle$$

Applying Z to $|0\rangle$ leaves it unchanged, while Z applied to $|1\rangle$ gives minus $|1\rangle$. Together, the X, Y, and Z gates flip or twist the state of a single qubit, and the Hadamard gate, which we will look at next, is what places it in superposition.

Explaining the Hadamard gate and superposition

The Hadamard gate, often called the H gate, creates superposition by transforming a classical state into a quantum state.

The matrix representation of the Hadamard gate is:

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

When you apply the Hadamard gate to a qubit in the state $|0\rangle$, the result is:

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

and, for a state $|1\rangle$, it is:

$$H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\$$

The Hadamard gate is used in quantum algorithms such as Grover's search and quantum teleportation.

Describing the CNOT gate and quantum entanglement

The controlled NOT (CNOT) gate is a two-qubit gate where the state of the target qubit depends on the state of the control qubit. If the control qubit is in the state $|0\rangle$, the target qubit remains unchanged. If the control qubit is in the state $|1\rangle$, the target qubit is flipped. The matrix representation for the CNOT gate is:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

This gate is critical for creating entanglement in quantum computing.

With the Pauli gates for flipping states, the Hadamard gate for creating superposition, and the CNOT gate for entangling qubits, you now have the core gate set needed to build circuits, which is what we do next.

Composing quantum circuits from gates

A **quantum circuit** is a series of quantum gates applied in a particular sequence. Quantum circuits allow you to manipulate qubits by applying quantum gates in a specific order.

The following steps highlight how a quantum circuit works:

1. You start by initializing your qubits, for example, setting them to the $|0\rangle$ state.
2. Next, you apply quantum gates such as the Hadamard, Pauli, or CNOT gates to these qubits.
3. Finally, you measure the qubits, which collapses the quantum state to a classical result.

To see how these steps fit together, consider the Bell state, an entangled state of two qubits. You can create it with a short circuit. First, apply the Hadamard gate to the first qubit to put it in superposition. Then apply the CNOT gate to entangle the first qubit with the second. You will build and run this circuit in the hands-on sections that follow.

Implementing basic quantum circuits in Qiskit

In this hands-on exercise, you will turn the gate and circuit ideas from the previous sections into working Qiskit code. Using a simulator that runs on your own computer, you will build a single-qubit circuit that shows superposition and a two-qubit circuit that produces a Bell state. You will also see how measurement results emerge from repeated runs. This is the workflow you will use for every quantum program in this book.

Creating and simulating a basic quantum circuit

To begin working with quantum circuits in Qiskit, you first need to import the necessary modules for creating and simulating circuits.

The following code imports the `QuantumCircuit` class from `qiskit` and the `Aer` simulator from `qiskit_aer`:

```
from qiskit import QuantumCircuit
from qiskit_aer import Aer
```

The following two imports are used in this code:

- `QuantumCircuit` is used to create and manipulate the quantum circuit.
- `Aer` is Qiskit's quantum simulator, which allows you to run quantum circuits on a classical computer.

This is useful because quantum computers are expensive and not always available, so simulators help you debug and test circuits before deploying them on real hardware.

Next, you can create a quantum circuit with one qubit and one classical bit. The qubit performs quantum computation, and the classical bit stores the measurement result.

The following code creates a quantum circuit named `qc` with one qubit and one classical bit:

```
# Create a quantum circuit with 1 qubit and 1 classical bit
qc = QuantumCircuit(1, 1)
```

Applying gates and measuring qubits in Qiskit

To apply a Hadamard gate and measure a qubit in Qiskit, you first use the `qc.h(0)` command, which applies the Hadamard gate to qubit zero and creates a superposition:

```
# Apply Hadamard gate to create superposition
qc.h(0)
```

This means the qubit now has a 50 percent chance of being in the state $|0\rangle$ and a 50 percent chance of being in the state $|1\rangle$. The Hadamard gate is used here because, unlike classical bits that are either zero or one, quantum bits can be both at the same time.

Next, you measure the qubit to collapse the superposition:

```
# Measure the qubit
qc.measure(0, 0)
```

Measuring the qubit will force it into either $|0\rangle$ or $|1\rangle$, and the result is stored in the classical bit zero. Measurement is necessary because it destroys quantum information and converts quantum results into classical data you can understand.

To visualize the quantum circuit you have created, use the following code to draw it using the MPL (matplotlib) output:

```
qc.draw('mpl')
```

This command helps you visualize the quantum circuit structure.

Next, use the Qiskit Aer simulator to run your quantum circuit. The following code selects the Aer simulator backend:

```
simulator = Aer.get_backend('aer_simulator')
```

This selects the Aer simulator instead of a real quantum computer, allowing you to simulate quantum behavior on a classical computer.

To run your quantum circuit on the simulator, execute the following code:

```
job = simulator.run(qc, shots=1000)
result = job.result()
```

This code runs the quantum circuit (`qc`) on the simulator, executing it 1,000 times and storing the results in the result variable. Running the circuit multiple times is important because quantum results are probabilistic and non-deterministic, so multiple shots help you see the probability distribution of the outcomes.

Extracting and interpreting measurement results

To extract the measurement results from your quantum circuit simulation, use the following code to get the counts of each result and print them:

```
# Get the measurement results
counts = result.get_counts(qc)
print(counts)
```

This code retrieves the measurement results from the quantum simulator and stores how many times each result (such as 0 or 1) appeared. When you run this cell, you will see the number of times each outcome occurred as follows:

```
{'1': 482, '0': 518}
```

Running the cell multiple times will show different results due to the probabilistic nature of quantum measurement.

Implementing a Bell state circuit

To implement a Bell state circuit in Qiskit, you start by creating a quantum circuit with two qubits. This allows you to manipulate both Q0 and Q1.

The following code creates a two-qubit quantum circuit:

```
qc = QuantumCircuit(2) # Two-qubit Circuit
```

Next, you apply a Hadamard gate to the first qubit (Q0) to create superposition:

```
# Apply Hadamard gate to create superposition
qc.h(0)
```

After that, apply a CNOT (controlled NOT) gate, where Q0 is the control qubit and Q1 is the target qubit. This step entangles the two qubits, so that measuring one will instantly determine the state of the other:

```
# Applying a CNOT Gate
qc.cx(0, 1)
```

The CNOT gate creates entanglement between two qubits (measuring one affects the other), which is a unique quantum phenomenon.

Then proceed to measure all the qubits in the circuit. To measure all qubits in your quantum circuit, use the following code:

```
qc.measure_all()
```

This command measures both qubits, Q0 and Q1, and collapses their quantum states into classical bits, resulting in either 0 or 1 for each qubit.

After measuring, you continue by drawing the circuit and running the simulation. The following code completes the Bell state circuit, draws it, and simulates it using Qiskit's Aer simulator:

```
# Draw the circuit
qc.draw('mpl')
```

To run your quantum circuit on the simulator, execute the following code:

```
job = simulator.run(qc, shots=1000)
result = job.result()
```

To extract the measurement results from your quantum circuit simulation, use the following code to get the counts of each result and print them:

```
# Get the measurement results
counts = result.get_counts(qc)
print(counts)
```

When you run this code, you see output similar to:

```
{'11': 499, '00': 501}
```

This output shows that the two-qubit quantum circuit produces measurement results where both qubits are either 00 or 11, demonstrating the entanglement created by the Bell state circuit. Because the outcomes 01 and 10 never appear, the two qubits always agree, and this correlation is what makes the Bell state the standard first example of entanglement. With these basic circuits in place, we can now look at the algorithms that put them to work.

Exploring foundational quantum algorithms

Quantum algorithms use the unique properties of quantum mechanics, such as quantum parallelism and entanglement, to solve certain problems much faster than classical algorithms. In this section, we will examine two of the most important examples: Grover's algorithm for searching and Shor's algorithm for factoring.

Explaining Grover's search algorithm and quantum speedup

The problem with classical computing is that searching an unsorted database of N items takes $O(N)$ time.

> **TIP**
>
> If the Big O notation is new to you, read about it here: https://en.wikipedia.org/wiki/Big_O_notation

For example, if you have one million items, you may need to check all one million before finding the correct one. This is where Grover's search algorithm comes into play, as it helps with quantum speedup.

Grover's search algorithm is a quantum algorithm that enables fast searching in unsorted databases. It searches an unsorted database in $O(\sqrt{N})$, providing a quadratic speedup over classical search. So let's look at how Grover's algorithm works:

- Grover's algorithm amplifies the probability of the correct answer while reducing the probabilities of incorrect ones. It consists of initialization,

an oracle function, the Grover diffusion operator, iteration, and measurement.

- Initialization: You start with a uniform superposition of all possible states.
- Oracle function: Then you apply the oracle function, which marks the correct solution by inverting its phase.
- Grover diffusion operator: You then apply the Grover diffusion operator, which increases the probability of the correct answer.
- Iteration: You repeat steps two and three (the oracle function and the Grover diffusion operator) about $\sqrt{N}$ times.
- Measurement: Finally, you perform a measurement, during which the correct answer appears with high probability.

So that is about Grover's search algorithm. Let's talk about Shor's factoring algorithm. Let's see what the importance of factoring is.

Describing Shor's factoring algorithm and its impact

Factoring large numbers is classically hard and forms the basis of RSA (Rivest–Shamir–Adleman) encryption, since classical factoring algorithms run in exponential time $O(e^n)$. So, let's see how Shor's algorithm helps with the quantum speedup.

Shor's factoring algorithm is another foundational quantum algorithm with major implications for cryptography and secure communications. Shor's algorithm can factor large numbers exponentially faster. It has a quantum runtime of $O(\log^3 n)$.

Here is how Shor's algorithm works:

- You first choose a random number a such that $1 < a <$ N.
- You then check if a and N are already factors using the greatest common divisor (GCD).
- Then you find the period r of $a^r \, mod N$ using quantum phase estimation.
- You then use r to determine a factor of N.
- Finally, you measure and output the factor.

Shor's algorithm is important because it breaks RSA encryption, which is widely used for secure communications. A classical computer cannot efficiently factor large numbers, but a quantum computer can.

Implementing Grover's algorithm with Qiskit

In this exercise, you will put Grover's algorithm to work on the smallest useful case: a search space of four states, represented by two qubits, where the goal is to find the marked state $|11\rangle$. You will build the oracle that marks the target and the diffusion operator that amplifies it. You will then assemble the full circuit, run it on the simulator, and visualize the results. Because the search space is so small, a single Grover iteration is enough to find the answer with near certainty, which makes this a clear way to see the algorithm work.

Start by importing the required libraries:

```
from qiskit import QuantumCircuit
from qiskit.visualization import plot_histogram
from qiskit_aer import Aer
```

The `plot_histogram` function is used to visualize measurement results.

You are now ready to define the oracle for Grover's algorithm, which marks the $|11\rangle$ state in your two-qubit system. To define the oracle for Grover's algorithm, you create a function that marks the $|11\rangle$ state using a controlled-Z (CZ) gate.

The following code defines the oracle as a two-qubit quantum circuit and applies the CZ gate to mark the $|11\rangle$ state:

```
# Define the Oracle (marks the |11) state)
def grover_oracle():
    oracle = QuantumCircuit(2)
    oracle.cz(0, 1)  # Controlled-Z gate to mark |11)
    return oracle
```

This function creates a two-qubit quantum circuit, applies the controlled-Z gate to flip the phase of the $|11\rangle$ state, and returns the modified circuit. The oracle tags the correct answer, which is the $|11\rangle$ state in this case. In classical computing, searching takes O(n) time, but with Grover's algorithm, it takes O(√n) time, which is exponentially faster. The oracle is what enables this speedup.

Now you need to define the diffusion operator, which is a key part of Grover's algorithm. The diffusion operator amplifies the probability of measuring the marked state, making it more likely to be observed.

The following code defines the diffusion operator as a function in Qiskit:

```
# Define the Diffusion Operator (amplifies the correct state)
def diffusion_operator():
    diffusion = QuantumCircuit(2)
    diffusion.h([0, 1])    # Apply Hadamard gates
    diffusion.x([0, 1])    # Apply X gates
    diffusion.cz(0, 1)     # Apply Controlled-Z gate
    diffusion.x([0, 1])    # Apply X gates again
    diffusion.h([0, 1])    # Apply Hadamard gates again
    return diffusion
```

The diffusion operator works in several steps. First, you apply the Hadamard gates to convert all qubits to an equal superposition. Then, the X gates flip $|0\rangle$ to $|1\rangle$ and $|1\rangle$ to $|0\rangle$, which is a bitwise NOT operation. Next, you apply the CZ gate, which flips the phase of the $|00\rangle$ state. After that, you apply the X gates again to reverse the bit flip, and finally, you apply the Hadamard gates again to complete the reverse transformation.

NOTE

Think of the diffusion operator as an inversion around the mean. It amplifies the probability of measuring the correct answer, which is the marked state in Grover's algorithm.

To create a two-qubit quantum circuit for Grover's algorithm, you start by initializing the circuit with two quantum bits and two classical bits. The two qubits are used for computation, while the two classical bits store the measurement results:

```
qc = QuantumCircuit(2, 2)
```

Next, you initialize both qubits with Hadamard gates to create an equal superposition, so each state has an equal probability of occurring:

```
qc.h([0, 1])
```

After initialization, you apply Grover's oracle using the `.compose()` method. This attaches the oracle to the main circuit, marking the correct answer ($|11\rangle$)

by flipping its phase:

```
qc = qc.compose(grover_oracle())
```

Then, you apply the diffusion operator, also using `.compose()`. This amplifies the probability of the $|11\rangle$ state:

```
qc = qc.compose(diffusion_operator())
```

Finally, you measure both qubits and store the results in the classical bits. This collapses the quantum state into a classical bit string, such as 00, 01, 10, or 11:

```
qc.measure([0, 1], [0, 1])
```

To visualize the full quantum circuit, you can draw it using the following command:

```
qc.draw('mpl')
```

This displays the structure of your quantum circuit, showing each gate and measurement step.

To run the Aer simulator and execute your quantum circuit, use the following code:

```
simulator = Aer.get_backend('aer_simulator')
```

Next, execute the quantum circuit and collect the results:

```
job = simulator.run(qc, shots=1000)
result = job.result()
```

Running the circuit with multiple shots (in this case, 1,000) is important because quantum results are probabilistic. More shots provide better accuracy and a more reliable estimate of the probability distribution for each possible outcome.

To extract and display the measurement results, use the following code:

```
counts = result.get_counts(qc)
print(counts)
```

This code retrieves the measurement results from the simulation and prints how many times each possible state (such as '00', '01', '10', and '11') was

measured. Since you are using two qubits, these are the possible outcomes.

To visualize the results, plot a histogram of the measurement counts:

```
plot_histogram(counts)
```

This produces the following output:

```
{'11': 1000}
```

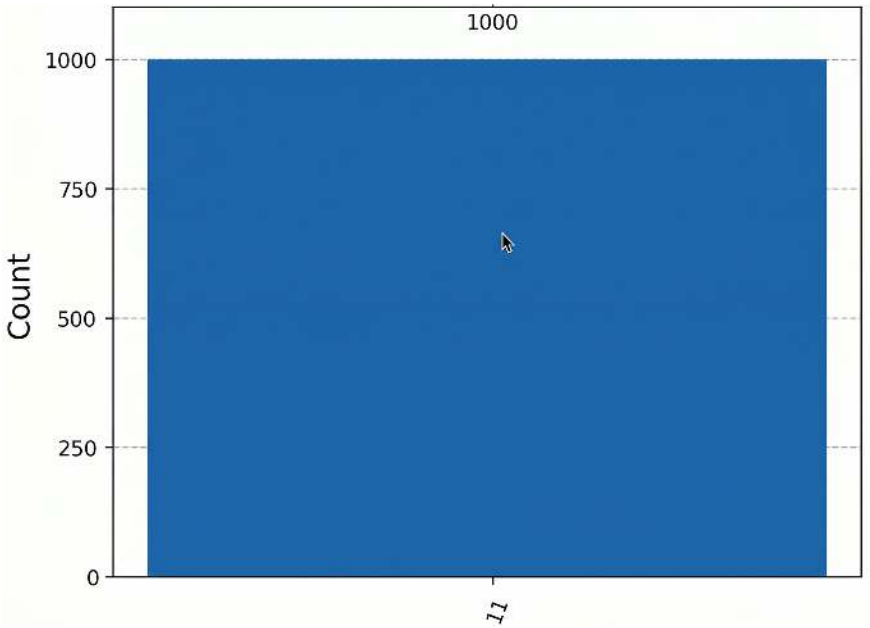


Figure 5.1: Histogram showing the count of measurement outcomes for a quantum circuit, with all counts in the '11' state

The histogram will show peaks for each measured state. After running Grover's algorithm, you should see a high probability for the '11' state, indicating a successful search. The histogram will typically show a strong peak at '11', with minimal counts for the other states. This demonstrates that Grover's algorithm has amplified the probability of measuring the marked state as expected.

Summary

Quantum gates are reversible, unitary operations, and a small set of them goes a long way. The Pauli gates flip a qubit's state or phase, the Hadamard gate creates superposition, and the CNOT gate entangles qubits. Combined in a circuit and ended with a measurement, they produce a Bell state. Because

measurement is probabilistic, you run a circuit many times and read the counts as a distribution, not as a single answer.

The same building blocks power quantum algorithms. Grover's algorithm uses an oracle to mark the answer and a diffusion operator to amplify it, giving a quadratic speedup for unstructured search. Shor's algorithm threatens RSA encryption by factoring large numbers far faster than any known classical method. Implementing Grover's algorithm in Qiskit showed that the pattern of marking, amplifying, and measuring is something you can build and verify yourself.

In the next chapter, we will apply these circuits and algorithms to artificial intelligence, exploring how quantum computing can enhance machine learning through variational quantum classifiers, quantum kernel methods, and hybrid quantum-classical neural networks.

6

Applying Quantum Computing to AI

In the previous chapter, you learned how quantum gates combine into circuits and how algorithms such as Grover's and Shor's use those circuits to attack specific problems such as search and factoring. Meanwhile, artificial intelligence is running into limits of its own. Modern models demand enormous computing power to train, optimization problems grow harder as models grow, and hidden patterns in complex data are difficult to extract. These pressures make AI a natural place to ask what quantum computing can contribute.

This chapter looks at where the two fields meet. You will see what quantum computing can and cannot offer machine learning, and then work with quantum models in Qiskit and PyTorch on small datasets that run on your own computer. You will see how classical data is turned into quantum states, how those states are used to classify data, and how a quantum circuit can sit inside a neural network that you train like any other. By the end, you will be able to build, train, and evaluate simple quantum machine learning models and judge their results realistically. This chapter covers the following topics:

- Exploring how quantum computing enhances AI capabilities
- Implementing a VQC model in Qiskit
- Processing data with quantum feature maps and kernel methods
- Training a quantum kernel-based classifier
- Advancing AI with hybrid quantum neural networks and quantum Boltzmann machines
- Building a hybrid quantum-classical AI model

Technical requirements

To follow the examples in this chapter, you need the following installed and working:

- Python 3.12 or later
- Jupyter Notebook or JupyterLab
- Qiskit, Qiskit Aer, Qiskit Machine Learning, Qiskit Algorithms, NumPy, Matplotlib, scikit-learn, and PyTorch

You can install all the libraries with a single command:

```
pip install qiskit qiskit-aer qiskit-machine-learning qiskit-algorithms numpy matplotlib scikit-learn
torch
```

The code files for this chapter are available at the book's GitHub repository: https://github.com/PacktPublishing/Quantum-Computing-for-AI-with-Python/blob/main/Chapter6.ipynb

Exploring how quantum computing enhances AI capabilities

Quantum computing offers game-changing advantages for AI and machine learning by enabling faster training, improved optimization, and enhanced feature extraction.

Before looking at specific AI models, consider the limitations of classical AI as follows:

- Deep learning and AI models require massive computational resources for training.
- Optimization problems, such as training neural networks and reinforcement learning, are hard to solve efficiently.
- Classical models also struggle to extract hidden patterns from complex data.

Quantum computing improves AI in three main ways:

- Faster computation: Quantum parallelism speeds up AI training; for example, solving complex neural networks and certain machine learning optimization problems runs exponentially faster.

- Better feature extraction: Quantum states capture multidimensional correlations in data more efficiently, and quantum feature mapping enhances classical machine learning models.
- Improved optimization: Quantum algorithms, such as the **quantum approximate optimization algorithm** (**QAOA**), help find optimal neural network weights.

For a real-world example, Google's Sycamore quantum processor outperformed the best classical supercomputer in optimization tasks, hinting at quantum computing's potential for deep learning.

With the limits of classical AI and the three areas where quantum computing can help in view, the next step is to see how classical and quantum machine learning compare side by side.

Comparing classical and quantum machine learning approaches

Quantum machine learning (**QML**) merges quantum computing and AI to build more learning algorithms. The following table compares classical and quantum ML:

Feature	Classical ML	Quantum ML
Computation	Sequential	Parallel due to superposition
Data processing	Limited scalability	Exponentially better feature mapping
Optimization	Hard for large datasets	Faster with quantum algorithms

Table 6.1: Comparing classical ML with quantum ML

The table shows that quantum machine learning offers potential gains in how data is represented and how models are optimized. These gains apply to specific problem types and depend on the algorithm and the hardware. With this comparison in mind, let's look at the techniques that put these ideas to work.

Identifying key quantum machine learning techniques

You can use several key QML techniques to enhance your AI models. Some of them are listed as follows:

- **Quantum feature mapping** transforms classical data into higher-dimensional quantum states, enabling more expressive representations. For example, **quantum kernel methods** use this mapping to improve machine learning classification tasks by measuring similarity in quantum feature space.
- **Quantum neural networks** (**QNNs**) replace classical neurons with quantum circuits, leading to faster model convergence.
- **Variational quantum classifier** (**VQC**) uses parameterized quantum circuits to classify data; you will explore this in detail in the next section.

A real-world example is IBM's development of quantum kernel estimation, which improves classical machine learning models using quantum-enhanced kernels.

Examining variational quantum classifiers

A **variational quantum classifier** (**VQC**) is a hybrid quantum-classical model that uses a quantum circuit to map input data to a quantum state and relies on a classical optimizer to train the quantum parameters. VQC is effective because it uses a quantum feature space for improved classification, operates efficiently on noisy intermediate-scale quantum (NISQ) devices, and has quantum-enhanced kernels that can outperform classical classifiers.

Implementing a VQC model in Qiskit

In this hands-on exercise, you will train a variational quantum classifier on a small synthetic dataset with two classes. The exercise shows how classical data is encoded into a quantum circuit, how a trainable circuit learns to separate the classes, and how to judge the model by its accuracy on unseen data. Because the dataset is small and runs on a simulator, you can focus on how the pieces fit together before moving to larger problems.

To begin, you need to install the required libraries for building and training a variational quantum classifier (VQC). Run the following command in your notebook to install and upgrade these packages:

```
pip install --upgrade qiskit qiskit-machine-learning qiskit-algorithms numpy matplotlib
```

The following code block imports `numpy` for numerical operations, `matplotlib` for plotting, `qiskit_aer` for quantum simulation, `StatevectorSampler` for simulating quantum state evolution, `VQC` for the variational quantum classifier, `FidelityQuantumKernel` for quantum similarity measurement, `ZZFeatureMap` and `RealAmplitudes` for quantum circuit construction, `COBYLA` for optimization, `PassManager`, `Optimize1qGatesDecomposition`, and `CommutativeCancellation` from the Qiskit transpiler to simplify the circuit before it runs, along with several scikit-learn modules for data handling and preprocessing:

```
# Import required libraries
import numpy as np
import matplotlib.pyplot as plt
from qiskit_aer import Aer
from qiskit.primitives import StatevectorSampler
from qiskit_machine_learning.algorithms.classifiers import VQC
from qiskit_machine_learning.kernels import FidelityQuantumKernel
from qiskit.circuit.library import ZZFeatureMap, RealAmplitudes
from qiskit_algorithms.optimizers import COBYLA
from qiskit.transpiler import PassManager
from qiskit.transpiler.passes import Optimize1qGatesDecomposition, CommutativeCancellation
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_classification
```

Next, let's generate and preprocess the synthetic binary dataset for quantum classification:

```
X, y = make_classification(n_samples=100,      # Total number of features
                           n_features=2,
                           n_informative=2,    # Ensure sum ≤ total features
                           n_redundant=0,      # No redundant features
                           n_clusters_per_class=1,
                           random_state=42)
```

You generate a synthetic dataset for quantum classification using the `make_classification` function from scikit-learn. This creates 100 samples with two features, both of which are informative for classification, and no

redundant features. Setting `random_state=42` ensures that you get the same data each time you run the code.

Next, you normalize the data so that each feature has mean zero and variance one. This normalization is important because quantum circuits are sensitive to input scales, and normalization improves training stability.

The following code normalizes the data:

```
# Normalize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

Finally, you split the normalized data into training and testing sets, using 80 percent of the data for training and 20 percent for testing. You do this with the `train_test_split` function, again using `random_state=42` for reproducibility. The following code splits the data:

```
# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)
```

The next step is to define the quantum feature map that will encode your classical data into quantum states.

The following code creates a quantum feature map using the `ZZFeatureMap` class, specifying a feature dimension of 2, two repetitions, and linear entanglement:

```
feature_map = ZZFeatureMap(feature_dimension=2, reps=2, entanglement='linear')
```

This feature map transforms your 2D classical data into quantum states by applying two layers of feature encoding and entangling the qubits in a linear chain. Quantum computers require this transformation because they cannot process classical data directly.

Next, you define the variational quantum circuit, or ansatz, using the `RealAmplitudes` class in Qiskit. This creates a trainable quantum circuit for classification tasks.

The following code defines the ansatz with two qubits, three repetitions of variational layers, and linear entanglement:

```
ansatz = RealAmplitudes(num_qubits=2, reps=3, entanglement='linear')
```

Here, the `entanglement='linear'` connects qubit 0 to qubit 1 in a linear fashion. The ansatz contains learnable parameters (theta) that will be trained during model optimization.

Next, you initialize the quantum sampler. In this case, you use the `StatevectorSampler`, which simulates the quantum circuit without requiring access to a real quantum computer.

The following code initializes the state vector sampler:

```
sampler = StatevectorSampler()
```

The `StatevectorSampler` is used for exact quantum state calculations. It provides accurate fidelity measurements, making it ideal for quantum kernel-based classification tasks.

To define the fidelity quantum kernel, you assign the quantum kernel to the `FidelityQuantumKernel`, set the feature map to the `feature_map` variable, and the fidelity to the sampler. The following code defines the fidelity quantum kernel using the feature map and sampler you created earlier:

```
quantum_kernel = FidelityQuantumKernel(feature_map=feature_map, fidelity=sampler)
```

This computes the similarity or fidelity between quantum states using the quantum feature map to generate kernel values. The quantum kernel replaces classical kernel methods with quantum similarity measures.

To set up the optimizer, assign the COBYLA optimizer with a maximum of 100 iterations:

```
optimizer = COBYLA(maxiter=100)
```

COBYLA stands for constrained optimization by linear approximation, and setting `maxiter=100` limits the optimization process to 100 iterations. COBYLA works well for small quantum circuits, which is why you use it here.

Next, define a custom pass manager for optimized transpilation to improve circuit execution for simulation:

```
pass_manager = PassManager([
    Optimize1qGatesDecomposition(basis=['u3', 'cx']),
    CommutativeCancellation()
])
```

This `PassManager` removes unnecessary quantum gates and optimizes the circuit for simulation. Optimizing the transpilation reduces noise and gate errors, making your quantum circuit execution more efficient.

Now you are ready to define and train your VQC model and then evaluate its accuracy on your test data.

The following code block defines the VQC model using the quantum feature map, trainable ansatz, optimizer, sampler, and pass manager you created earlier:

```
vqc = VQC(
    feature_map=feature_map,
    ansatz=ansatz,
    optimizer=optimizer,
    sampler=sampler,
    pass_manager=pass_manager  # Provide pass manager to avoid transpilation warnings
)
```

This creates your VQC model, which uses the quantum feature map, the trainable quantum ansatz, the COBYLA optimizer, the quantum sampler, and the `PassManager` to avoid transpilation warnings. The VQC uses quantum circuits to classify data.

Next, you train the VQC model on your training data:

```
vqc.fit(X_train, y_train)
```

After training, you evaluate the model by making predictions on your test data and calculating the accuracy:

```
y_pred = vqc.predict(X_test)

# Calculate accuracy
accuracy = np.mean(y_pred == y_test)
print(f"VQC Accuracy: {accuracy * 100:.2f}%")
```

This code predicts the labels for your test data, computes the accuracy, and prints it as a percentage.

When you run this, the model will be created and trained on your data:

```
VQC Accuracy: 75.00%
```

Since you are only using 100 data points, the accuracy may not be very high. In this run, the VQC accuracy is 75%. Using more data and more training can improve the accuracy.

In this exercise, you saw how a variational quantum classifier encodes data with a feature map, learns through a trainable circuit, and is judged by its accuracy on unseen data. The feature map at the heart of this model is also the foundation of quantum kernel methods, which we will explore next.

Processing data with quantum feature maps and kernel methods

Quantum computing introduces a new paradigm for data representation and computation, providing novel ways to process and classify data more efficiently than classical methods.

Introducing feature mapping

Feature mapping is the process of transforming input data into a higher-dimensional space to make it more separable by a classifier. The following table compares classical versus quantum feature mapping:

Feature mapping type	How it works	Pros	Cons
Classical	Uses mathematical functions (e.g., polynomial, RBF kernels) to transform data into higher dimensions.	Well understood, computationally feasible.	Computationally expensive for high-dimensional spaces.
Quantum	Encodes data into quantum states, leveraging superposition and entanglement to create richer feature spaces.	More expressive transformations, potentially exponentially better feature spaces.	Requires quantum hardware or simulators.

Table 6.1: Comparing classical vs. quantum feature mapping

The table shows that quantum feature mapping trades the low cost and familiarity of classical transformations for potentially richer representations. The price is that it needs a quantum simulator or hardware. To see how this mapping works in practice, consider its three stages.

The following steps explain how quantum feature mapping works:

1. The first step is classical data encoding, which converts classical features x into quantum states, which are $|\psi(x)\rangle$.
2. The next step is quantum circuit transformation, where we use a quantum feature map or QFM circuit to encode data.
3. The next step is quantum kernel calculation, where we measure the overlap or fidelity between quantum states to compute a similarity metric.

Quantum kernel methods

A kernel function in machine learning (ML) measures the similarity between two data points. This is especially important for algorithms such as **support vector machines** (**SVMs**), which perform classification in high-dimensional spaces by using these similarity measures.

Quantum kernel methods work by replacing classical kernels with a similarity measure between quantum-encoded feature vectors. Instead of computing dot products in classical space, you compute the fidelity of quantum states. The formula for the quantum kernel is:

$$K(x_i, x_j) = |\langle\psi(x_i)|\psi(x_j)\rangle|^2$$

This quantum similarity metric is then used in standard ML models. The advantages of quantum kernel methods include higher expressivity, which allows you to capture relationships that classical kernels cannot, efficiency on quantum hardware, and the ability to work well on **noisy intermediate-scale quantum** (**NISQ**) devices. Quantum kernel methods may outperform classical methods for large, complex datasets.

Training a quantum kernel-based classifier

The objective of this hands-on exercise is to train a quantum kernel classifier using Qiskit. This implementation will first encode the classical data into quantum states using a feature map, then compute the quantum kernel matrix, and train a classical SVM classifier with the quantum kernel.

To begin working with quantum feature mapping and kernel methods for AI, you need to import several essential libraries for quantum and classical machine learning tasks. The following code imports all the required libraries:

```
import numpy as np
from qiskit.primitives import StatevectorSampler
from qiskit_machine_learning.kernels import FidelityQuantumKernel
from qiskit.circuit.library import ZZFeatureMap
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons
```

Here, we've imported the following new libraries:

- `SVC` for classical support vector classification
- `make_moons` for generating a non-linearly separable dataset

Next, you will create a custom fidelity function for the quantum kernel. This function computes the fidelity, or similarity, between two quantum states using the `StatevectorSampler` primitive. The process involves combining two circuits by multiplying the first circuit with the inverse of the second circuit, running the combined circuit using the sampler, and extracting the probability of measurement outcomes.

The following code defines the custom fidelity function:

```
# Step 1: Custom Fidelity Function for Quantum Kernel
def custom_fidelity(sampler, circuit_1, circuit_2):
    """
    Compute the fidelity between two quantum states using the StatevectorSampler primitive.
    """
    combined_circuit = circuit_1.compose(circuit_2.inverse())  # Combine the circuits
    job = sampler.run([combined_circuit])
    result = job.result()
    probabilities = result.quasi_dists[0]
    # Fidelity is the probability of measuring the all-zero state
    num_qubits = combined_circuit.num_qubits
    zero_state = tuple([0] * num_qubits)
    fidelity = probabilities.get(zero_state)
    return fidelity
```

This function first combines the two circuits, runs the combined circuit with the sampler, retrieves the result, and then extracts the probability of measuring the all-zero quantum state, which represents the fidelity between the two quantum states.

So for that, you set fidelity as the probability of measurement. You write `fidelity = probability.get((0, 0))` and return the fidelity. This gives you back the fidelity value.

The reason you compute the fidelity here is that higher fidelity means higher similarity between the two quantum states. The quantum kernel uses fidelity as its similarity measure, so this calculation is essential. For example, if `circuit_1` and `circuit_2` are very similar, the fidelity will be close to 1.0. If they are different, the fidelity will be closer to 0.0. This gives you a clear idea of how similar the quantum states are.

To generate a synthetic dataset for quantum kernel classification, use the two-moons dataset, which is non-linearly separable and ideal for kernel-based models.

The following code generates 100 samples from the two-moons dataset with a small amount of noise to make the data more realistic:

```
X, y = make_moons(n_samples=100, noise=0.1, random_state=42)
```

Next, normalize the data and split it into training and testing sets. Normalizing the feature values to have a mean of 0 and a variance of 1 is important for quantum models, as quantum circuits process angles, and normalized values improve training stability.

The following code normalizes the data and splits it into 80% training and 20% testing sets:

```
# Normalize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)
```

Let's go ahead and define the quantum feature map. For this, you use the `ZZFeatureMap` with a feature dimension of 2, reps set to 2, and entanglement

set to 'linear'. This step encodes your classical data into quantum states, and the ZZFeatureMap applies ZZ entanglement to the qubits.

> **NOTE**
>
> Quantum circuits do not process classical data directly, so the feature map is necessary to transform classical data into quantum states.

The following code defines the quantum feature map:

```
feature_map = ZZFeatureMap(feature_dimension=2, reps=2, entanglement='linear')
```

Next, initialize the quantum sampler. You use the `StatevectorSampler`, which simulates quantum circuits exactly and avoids noise from real quantum hardware.

Here is the code to initialize the sampler:

```
sampler = StatevectorSampler()
```

Now, define the fidelity quantum kernel. You create a subclass called `CustomFidelityQuantumKernel` that inherits from `FidelityQuantumKernel`. In the initializer, you pass the feature map and sampler and set up the superclass with the feature map and fidelity set to None. You also define a `_compute_fidelity` method that uses your `custom_fidelity` function to compute the similarity between two quantum circuits.

The code for defining the custom fidelity quantum kernel is as follows:

```
class CustomFidelityQuantumKernel(FidelityQuantumKernel):
    def __init__(self, feature_map, sampler):
        super().__init__(feature_map=feature_map, fidelity=None)
        self.sampler = sampler

    def _compute_fidelity(self, circuit_1, circuit_2):
        return custom_fidelity(self.sampler, circuit_1, circuit_2)
```

This creates a subclass of `FidelityQuantumKernel` that uses your `custom_fidelity` function for computing quantum state similarity.

To compute the quantum kernel matrix, you first create an instance of your custom fidelity quantum kernel class and then evaluate the similarity for both training and test data.

The following code creates the quantum kernel and computes the kernel matrix for the training and testing data:

```
quantum_kernel = CustomFidelityQuantumKernel(feature_map=feature_map, sampler=sampler)

X_train_kernel = quantum_kernel.evaluate(X_train)
X_test_kernel = quantum_kernel.evaluate(X_test, X_train)
```

Here, `X_train_kernel` computes the similarity of the training data, and `X_test_kernel` computes the similarity of the test data with the training data.

The reason for using a quantum kernel in this case is that classical SVMs struggle with complex datasets, while quantum kernels use quantum states to measure similarity, providing an advantage for more complex data.

To train a classical SVM using the quantum kernel, you use the precomputed quantum kernel matrix as input to the SVM. This approach works well with custom kernels and allows you to classify nonlinearly separable data.

The following code trains a classical SVM model using the quantum kernel:

```
svm = SVC(kernel="precomputed")
svm.fit(X_train_kernel, y_train)
```

Next, you evaluate the model's accuracy by predicting the labels on the test data and comparing them to the true labels.

The following code predicts the test labels and computes the accuracy:

```
y_pred = svm.predict(X_test_kernel)

accuracy = np.mean(y_pred == y_test)
print(f"Quantum Kernel SVM Accuracy: {accuracy * 100:.2f}%")
```

The output generated is:

```
Quantum Kernel SVM Accuracy: 60.00%
```

In this run, the accuracy was 60 percent, so treat it as a starting point and not a benchmark because the value of a quantum kernel depends on the data and the feature map. In the next section, we will move from kernels to neural networks and combine quantum circuits with classical layers in a hybrid model.

Advancing AI with hybrid quantum neural networks and quantum Boltzmann machines

Quantum neural networks (**QNNs**) are a new frontier in AI, combining classical neural networks with quantum circuits to enhance learning capabilities. They have potential applications in optimization, classification, and generative models.

Hybrid quantum-classical neural networks (HQCNNs)

A **hybrid quantum-classical neural network** (**HQCNN**) consists of classical layers, such as fully connected layers in TensorFlow and PyTorch, and quantum layers, which can be quantum circuits that perform transformations. You use hybrid networks to take advantage of quantum power while still benefiting from efficient classical computations. These networks can run on near-term quantum devices, known as the NISQ era, and potentially solve problems faster than classical deep learning.

A simple hybrid QNN typically follows this architecture: you start with classical processing, which normalizes and encodes classical data. Next is the quantum feature encoding, which maps the data to quantum states. Then comes the quantum processing layer, which is the **variational quantum circuit** (**VQC**) that learns from data, as you have seen in the previous section. Finally, you have classical post-processing, which measures quantum outputs and passes them into classical layers.

Quantum Boltzmann machines (QBMs)

A **Boltzmann machine (BM)** is a stochastic neural network that learns probability distributions over data. It consists of visible units that represent data features and hidden units that learn latent representations, and it uses energy-based learning to minimize an energy function for pattern recognition.

A **quantum Boltzmann machine** (**QBM**) differs by replacing classical probabilistic sampling with quantum sampling. By using quantum superposition, a QBM can explore multiple states at once and learn more expressive probability distributions than classical BMs.

We use QBMs because they can solve combinatorial optimization problems faster, such as portfolio optimization. They take advantage of quantum parallelism to explore many possible solutions and are potentially useful for deep generative models in machine learning.

Building a hybrid quantum-classical AI model

In this hands-on exercise, you will build a hybrid quantum-classical neural network that combines classical layers in PyTorch with a quantum layer built in Qiskit. You will train it on the two-moons dataset, a nonlinearly separable problem, using a standard PyTorch training loop. This is where the quantum circuits you have used so far become a layer in a model that you train like any other neural network.

We begin by importing the required libraries:

```
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from qiskit.primitives import StatevectorEstimator
from qiskit.circuit.library import RealAmplitudes
from qiskit_machine_learning.neural_networks import EstimatorQNN
from qiskit_machine_learning.connectors import TorchConnector
```

The following libraries are introduced for the first time in this code:

- `torch` is the PyTorch deep learning framework.
- `torch.nn` provides neural network layers.
- `torch.optim` supplies optimizers for training.
- `TorchConnector` integrates QNNs with PyTorch layers.

We are using a hybrid model because classical models struggle with some datasets, and quantum models can process high-dimensional data differently.

Let's start by generating and preprocessing the data for your hybrid quantum-classical neural network using the two-moons dataset, which is nonlinearly separable and ideal for hybrid models. The following code generates 100 samples from the two-moons dataset with a small amount of noise and a fixed random state for reproducibility:

```
X, y = make_moons(n_samples=100, noise=0.1, random_state=42)
```

Next, normalize and split the data:

```
# Normalize and split data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)
```

We've also split the dataset into training (80 percent) and testing (20 percent) sets.

> **NOTE**
>
> Normalization ensures the feature values have a mean of 0 and a variance of 1, which is important because quantum circuits process angles, and normalized values improve training stability.

Finally, convert the NumPy arrays to PyTorch tensors. The input features are converted to `torch.float32`, and the classification labels are converted to `torch.long`:

```
# Convert to PyTorch tensors
X_train_torch = torch.tensor(X_train, dtype=torch.float32)
X_test_torch = torch.tensor(X_test, dtype=torch.float32)
y_train_torch = torch.tensor(y_train, dtype=torch.long)
y_test_torch = torch.tensor(y_test, dtype=torch.long)
```

You now define the quantum circuit using the `RealAmplitudes` ansatz, which provides a trainable quantum feature map for your hybrid quantum-classical neural network:

```
num_qubits = 2
quantum_circuit = RealAmplitudes(num_qubits, reps=3)
```

Here, you set `num_qubits` to 2 and create a `quantum_circuit` with three repetitions (layers) of parameterized gates. Using `RealAmplitudes` allows you to build a variational quantum circuit that is suitable for encoding input data and training.

Next, you encapsulate the quantum circuit inside an `EstimatorQNN`, which converts the quantum circuit into a quantum neural network layer using the following code:

```
estimator = StatevectorEstimator()
qnn = EstimatorQNN(
    circuit=quantum_circuit,
    estimator=estimator,
    input_params=quantum_circuit.parameters[:2],
    weight_params=quantum_circuit.parameters[2:]
)
```

This code uses the `StatevectorEstimator` to simulate quantum behavior. The parameters of the quantum circuit are split into input parameters (the first two, which map input data) and weight parameters (the remaining parameters, which are trainable). This setup enables the quantum circuit to function as a neural network layer within your hybrid model.

Next, let's go ahead and convert the quantum circuit to PyTorch-compatible layers using `TorchConnector`:

```
quantum_layer = TorchConnector(qnn)
```

Here it integrates the estimator `qnn` into PyTorch and it converts the `qnn` into a PyTorch layer. `TorchConnector` allows training quantum circuits using PyTorch optimizers.

You now define your hybrid quantum-classical neural network by creating a custom PyTorch module. Follow these steps:

1. Start by declaring a class called `HybridQuantumNN` that inherits from `nn.Module`.
2. In the initializer, call the superclass initializer, then define the first classical layer as `self.fc1 = nn.Linear(2, 2)`, which takes two input features and outputs two features.
3. Next, assign the quantum layer you created earlier to `self.quantum = quantum_layer`, which expects two inputs.
4. Finally, define the output classical layer as `self.fc2 = nn.Linear(1, 2)`, since the quantum neural network outputs a single feature that you map to two classes.

The forward method of your model first applies a hyperbolic tangent activation to the output of the first classical layer, then passes the result through the quantum layer, and finally through the last classical layer. The output is returned as the model's prediction.

The following code defines the hybrid quantum-classical neural network architecture:

```
class HybridQuantumNN(nn.Module):
    def __init__(self):
        super(HybridQuantumNN, self).__init__()
        self.fc1 = nn.Linear(2, 2)
        self.quantum = quantum_layer
        self.fc2 = nn.Linear(1, 2)

    def forward(self, x):
        x = torch.tanh(self.fc1(x))
        x = self.quantum(x)
        x = self.fc2(x)
        return x
```

Next, initialize your model, loss function, and optimizer. Instantiate the model with `model = HybridQuantumNN()`. For the loss function, use cross-entropy loss by setting `criterion = nn.CrossEntropyLoss()`. Choose the Adam optimizer for training, as it works well with quantum-classical models, and set the learning rate to 0.01.

The following code initializes the model, loss function, and optimizer:

```
model = HybridQuantumNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

This setup prepares your hybrid quantum-classical neural network for training and evaluation.

To train your hybrid quantum-classical neural network, you need to define the number of epochs and implement the training loop. The following code sets the number of epochs to 50 and iterates through each epoch, performing the standard steps of zeroing the gradients, computing the outputs, calculating the loss, backpropagating the gradients, and updating the model weights:

```
epochs = 50
for epoch in range(epochs):
    optimizer.zero_grad()
    outputs = model(X_train_torch)
    loss = criterion(outputs, y_train_torch)
    loss.backward()
    optimizer.step()
```

```
    if (epoch + 1) % 10 == 0:
        print(f"Epoch {epoch+1}/{epochs}, Loss: {loss.item():.4f}")
```

This code prints the loss every 10 epochs so you can monitor the training progress.

After training, you can evaluate the model's performance on the test set. The following code disables gradient computation, makes predictions on the test data, calculates the accuracy, and prints the result:

```
with torch.no_grad():
    y_pred = model(X_test_torch).argmax(dim=1)
    accuracy = (y_pred == y_test_torch).float().mean()
    print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

When you run this code, you may see a message such as "No gradient function provided, creating a gradient function. If your Estimator requires transpilation, please provide a pass manager." This is an expected runtime message from Qiskit indicating that a gradient function is being created automatically for the quantum neural network layer. If your Estimator requires transpilation, you can provide a pass manager to customize the transpilation process. Otherwise, you can generally ignore this warning unless you need to optimize circuit transpilation for your use case.

The output produced is:

```
No gradient function provided, creating a gradient function. If your Estimator requires
transpilation, please provide a pass manager.
Epoch 10/50, Loss: 0.6116
Epoch 20/50, Loss: 0.5737
Epoch 30/50, Loss: 0.5415
Epoch 40/50, Loss: 0.5165
Epoch 50/50, Loss: 0.4928
Test Accuracy: 100.00%
```

During training, you should observe that the loss decreases as the epochs progress. In some runs, you might achieve a test accuracy of 100 percent, but this will vary depending on the generated data and the random initialization of the model. If you rerun the training, the accuracy may change, reflecting the stochastic nature of the process.

You may see different results depending on how your data is generated and used in the simulation. For example, in one run, the test accuracy might be as

low as 30 percent, reflecting the variability in synthetic data generation. You are encouraged to use your own dataset or download one from sources such as Kaggle or GitHub to experiment further.

The variation in results is a reminder to treat a small simulated dataset as a demonstration of the workflow, not as proof of quantum advantage. What matters here is that a quantum layer can sit alongside classical layers and be trained with the same tools you already know.

Summary

Quantum machine learning works best when you treat it as a partnership. A quantum circuit handles the part it is suited for: encoding data into a richer quantum feature space. A classical optimizer or neural network handles the training. This idea of feature mapping underlies both approaches you built: a variational quantum classifier that learns through a trainable circuit, and a quantum kernel that measures similarity between data points for a classical SVM. The same partnership extends to hybrid networks, where a quantum layer trains inside PyTorch like any other layer.

The results also teach a caution. Accuracy on these small simulated datasets varied from run to run, and your models did not clearly beat classical ones. Quantum advantages in machine learning are specific to the problem, the data, and the hardware, and they are not guaranteed. The habit to take away is to evaluate every quantum model on held-out data and compare it honestly with a classical baseline.

In the next chapter, we will explore real-world use cases by simulating a quantum chemistry problem in Qiskit, defining a molecular system and its electronic structure, and exploring quantum optimization techniques, including quantum annealing and D-Wave for combinatorial optimization.

7

Exploring AI-Quantum Use Cases and Optimization

In the previous chapter, you built quantum machine learning models and saw that their results on small datasets depend heavily on the data and the method. The harder question is where this combination is genuinely useful. Many valuable problems, from predicting how molecules behave to choosing the best mix of investments, involve many interacting parts, and the work grows quickly as parts are added.

This chapter moves from models to applications. You will see how AI and quantum computing are being applied in chemistry, finance, and drug discovery, and then work through two hands-on exercises on a simulator: estimating the energy of a molecule and selecting assets for a portfolio. Along the way, you will learn how quantum annealing and D-Wave approach optimization differently from gate-based machines, and how to read the results of small simulations with appropriate caution. In this chapter, we will cover the following topics:

- Understanding AI-quantum applications across different fields
- Simulating a quantum chemistry problem with Qiskit
- Exploring quantum annealing for combinatorial optimization
- Implementing hybrid AI and quantum optimization for portfolio management

Technical requirements

To follow the examples in this chapter, you need the following installed and working:

- Python 3.12 or later
- Jupyter Notebook or JupyterLab
- Qiskit, Qiskit Nature, Qiskit Algorithms, Qiskit Finance, Qiskit Optimization, PySCF, NumPy, pandas, and Matplotlib

You can install all the libraries with a single command:

```
pip install qiskit qiskit-nature qiskit-algorithms qiskit-finance qiskit-optimization pyscf numpy
pandas matplotlib
```

The code files for this chapter are available at the book's GitHub repository: https://github.com/PacktPublishing/Quantum-Computing-for-AI-with-Python/blob/main/Chapter7.ipynb

Understanding AI-quantum applications across different fields

Artificial intelligence and quantum computing (AIQC) open new frontiers in scientific research, financial modeling, and drug discovery. In this section, we will see how quantum computing (QC) is used for quantum chemistry simulations, alongside applications in finance and drug discovery.

Explaining AI-assisted quantum chemistry simulations

Quantum mechanics governs the behavior of molecules, making QC a natural tool for simulating them. Classical computers struggle with electron interactions in large molecules, the exponential complexity of Schrödinger's equation, and accurate quantum-state prediction for materials and drug discovery.

Artificial intelligence (AI) models approximate wave functions, speeding up calculations. Quantum neural networks (QNNs) help predict molecular properties, and AI-driven optimizers improve quantum simulations.

Applications of AI and QC in chemistry include materials science, where they can help design better superconductors; battery research, where they can simulate new energy storage materials; and drug discovery, where they can predict molecule-protein interactions.

These chemistry applications lead into the use of AI and quantum computing for financial modeling.

Applying AI and quantum computing to financial modeling

Financial markets involve high-dimensional optimization problems such as portfolio optimization, fraud detection, and risk analysis. Portfolio optimization seeks to minimize risk while maximizing returns; fraud detection identifies anomalous transactions; and risk analysis simulates future market scenarios.

AI-driven prediction models analyze stock trends, quantum annealing can optimize large portfolios, and quantum-enhanced Monte Carlo simulations can improve risk modeling.

In finance, portfolio optimization aims to find the best stock combinations, AI models can detect unusual transactions, and quantum-enhanced predictions can help with market forecasting.

These applications are followed by quantum computing approaches to drug discovery.

Describing quantum-enabled drug discovery workflows

Drug discovery requires simulating protein-ligand interactions, which classical computers struggle with. Quantum computing offers more accurate quantum-state representations for molecule simulation, quantum machine learning (QML) to predict drug efficacy, and optimization algorithms for screening chemical compounds.

AI models can generate and refine molecular structures. Quantum ML classifiers predict drug-protein binding affinity, while quantum-enhanced search can identify drug candidates faster.

In healthcare and pharma, these methods can support personalized medication by simulating DNA-protein interactions, cancer research by helping discover quantum-inspired treatment plans, and antiviral drug development by identifying new antivirals faster.

A Qiskit simulation of the hydrogen molecule covered in the next section will provide a concrete chemistry example.

Simulating a quantum chemistry problem with Qiskit

In this hands-on exercise, you will estimate the ground-state energy of the hydrogen molecule, the simplest molecule in chemistry, using Qiskit Nature and the **variational quantum eigensolver** (**VQE**). The ground-state energy is the lowest energy the molecule can have, and it determines properties such as how stable the molecule is. Hydrogen is small enough to simulate on your own computer, yet the same approach carries over to larger molecules.

We begin by importing the libraries and defining the molecule.

Importing libraries and defining the H_2 geometry

Create a notebook for the simulation and make sure the required packages are available. In this notebook, start importing the required libraries as follows:

```
import numpy as np
from qiskit_nature.second_q.problems import ElectronicStructureProblem
from qiskit_nature.second_q.drivers import PySCFDriver
from qiskit_nature.second_q.transformers import FreezeCoreTransformer
from qiskit_nature.second_q.mappers import JordanWignerMapper
from qiskit_algorithms import VQE
from qiskit.circuit.library import TwoLocal
from qiskit_algorithms.optimizers import COBYLA
from qiskit.primitives import StatevectorEstimator as Estimator
from qiskit.circuit.library import n_local
```

The libraries imported are:

- NumPy as `np` for numerical operations.
- `ElectronicStructureProblem` defines the quantum chemistry problem.problem
- `PySCFDriver` computes molecular orbitals.
- `FreezeCoreTransformer` simplifies the problem by freezing core electrons.
- `JordanWignerMapper` converts electronic Hamiltonians into qubit operators.

- `VQE` for finding the ground-state energy.
- `TwoLocal` for a variational quantum circuit.
- `COBYLA` for tuning circuit parameters.
- `Estimator` for computing expectation values.

Qiskit Nature bridges quantum computing and chemistry by solving Schrödinger's equation and providing the functions used in this simulation.

Next, the molecule is defined by its atomic positions. Each entry gives an element followed by its XYZ coordinates, with the distance expressed in angstrom units. Here, the H_2 bond length is 0.735 angstroms, which is close to equilibrium.

The following assignment defines the two hydrogen atoms and their positions:

```
# Define molecule (H2) geometry
molecule_geometry = "H 0 0 0; H 0 0 0.735"
```

Building the qubit Hamiltonian and configuring VQE

Use `PySCFDriver` to compute molecular orbitals, then create an electronic structure problem and map it into qubit space with the **Jordan-Wigner transformation**.

The driver uses PySCF, a Python-based quantum chemistry package, to compute the molecular orbitals for H_2. The STO-3G basis provides simplified hydrogen orbitals; it is computationally cheap and is used here for the small H_2 molecule.

The following code computes the molecular orbitals and creates an electronic structure problem from the driver results:

```
# Use PySCFDriver to compute molecular orbitals
driver = PySCFDriver(atom=molecule_geometry, basis="sto3g")

# Define electronic structure problem
es_problem = driver.run()
```

The electronic structure problem represents the molecular Hamiltonian in second quantization and lets Qiskit Nature interface with the quantum

chemistry data.

The next step is to apply the **freeze-core approximation**. Applying the freeze-core approximation is optional, but it is recommended. It removes core electrons that are not needed for a simple molecule such as H_2, reducing the computational cost. The approximation is used because only valence electrons matter in most chemical reactions.

The following code applies the optional freeze-core transformation and initializes the Jordan-Wigner mapper:

```
# Apply freeze-core approximation (optional)
transformer = FreezeCoreTransformer()
es_problem = transformer.transform(es_problem)

# Map to qubit space using Jordan-Wigner transformation
mapper = JordanWignerMapper()
```

The Jordan-Wigner mapper converts fermionic operators to qubit operators. It directly maps electrons to qubits, making it easier to use for small molecules.

Estimating H_2 ground-state energy with VQE

You pass the estimator, ansatz, and optimizer to VQE to configure the solver. The following four steps configure VQE:

1. Define a TwoLocal variational ansatz. Its RY and RZ rotation blocks rotate the qubits, while the CZ entanglement block adds quantum correlations. This parameterized circuit is efficient for VQE optimization.
   The following code creates the ansatz with the rotation and entanglement blocks:

   ```
   # Define a variational ansatz (Two-Local circuit)
   ansatz = n_local(4, rotation_blocks=["ry", "rz"], entanglement_blocks="cz")
   ```

2. Configure COBYLA, which stands for constrained optimization by linear approximation. Set `maxiter=100` to run up to 100 optimization steps. COBYLA works well for small variational circuits.
   The following code sets the optimizer and its iteration limit:

   ```
   # Define optimizer
   optimizer = COBYLA(maxiter=100)
   ```

3. Initialize the Estimator primitive to compute expectation values of quantum circuits. This version of the code uses the primitive updated for Qiskit 1.2 or later; upgrade Qiskit if you have an older version. The following code initializes the estimator:

```
# Initialize estimator primitive (Updated for Qiskit 1.2+)
estimator = Estimator()
```

4. Set up VQE with the estimator, ansatz, and optimizer. The `estimator` computes quantum expectation values, the `ansatz` supplies the variational circuit, and the `optimizer` chooses its parameters. The following code passes these three components to VQE:

```
# Set up VQE algorithm
vqe = VQE(estimator, ansatz, optimizer)
```

VQE finds the lowest-energy state of H_2 faster than classical methods.

Next, let us map the Hamiltonians to qubits. The following cell maps the electronic-structure Hamiltonian, runs VQE, and prints the result; the optimizer and estimator lines retain the configuration already set up:

```
# Compute ground state energy
hamiltonian = mapper.map(es_problem.hamiltonian.second_q_op())
result = vqe.compute_minimum_eigenvalue(hamiltonian)
print(f"Estimated electronic ground state energy: {result.optimal_value:.4f} Hartree")
```

In this code, we are:

- Using the Jordan Wigner mapper to map the molecular Hamiltonian to qubits
- Running VQE on the mapped Hamiltonian to estimate the molecule's ground-state energy

The printed estimate indicates the molecular stability of H_2. In this run, the output is:

```
Estimated electronic ground state energy: -1.1373 Hartree
```

The simulation demonstrates that a quantum algorithm can estimate H_2's ground-state energy. It also illustrates a hybrid quantum-classical approach and the feasibility of quantum-enhanced AI and drug discovery.

AI-assisted quantum chemistry simulations, financial modeling, and drug discovery are application areas for AI and quantum computing.

Financial modeling can improve portfolio optimization and risk analysis. AI-powered drug discovery can accelerate new drug formulation, and the Qiskit exercise demonstrates real-world QC applications through quantum chemistry simulation.

Exploring quantum annealing for combinatorial optimization

Quantum computing offers approaches to solving optimization problems more efficiently than classical methods. In this section, we will see how quantum annealing and D-Wave hardware address combinatorial problems.

Explaining quantum annealing and D-Wave QUBO hardware

Quantum annealing is a specialized quantum optimization technique that uses quantum fluctuations to find global optima for combinatorial problems. It is useful for finding a system's minimum energy state, solving NP-hard problems such as portfolio optimization, logistics, and scheduling, and performing quantum-enhanced combinatorial searches.

The mathematical goal is to minimize $f(x)$ over x, where x is an element of $\{0,1\}^n$, by encoding the problem in a Hamiltonian function. The equation for this is given as:

$$\min_x f(x), \quad \text{where} \quad x \in \{0,1\}^n$$

The following steps highlight how quantum annealing works:

- The system begins in a uniform quantum superposition that contains all possible states.
- A time-dependent Hamiltonian gradually transitions from an initial, simple Hamiltonian to the problem Hamiltonian.
- Finally, quantum fluctuations guide the system toward the lowest-energy state, which corresponds to the optimal solution.

The Schrödinger equation describes this process as:

$$H(t) = A(t)H_{\text{initial}} + B(t)H_{\text{problem}}$$

Here:

- $H_{initial}$ is an easy-to-solve Hamiltonian.
- $H_{problem}$ represents the optimization problem.

These equations describe annealing in principle. To see how it runs on real machines, we turn to D-Wave, the company that builds quantum annealing hardware.

D-Wave is a company that builds quantum annealing hardware for combinatorial optimization problems. Unlike gate-based quantum computers from companies such as IBM, Google, and Rigetti, D-Wave uses quantum annealing hardware to find optimal solutions for **quadratic unconstrained binary optimization** (**QUBO**) problems. Its use cases include logistics, where it can optimize delivery routes for supply chains; finance, where it can optimize portfolios for investment management; and manufacturing, where it can find efficient factory layouts. D-Wave machines naturally solve QUBO problems using quantum annealing.

Portfolio optimization is one application of these methods.

Combining AI predictions with quantum portfolio allocation

Financial portfolio optimization requires choosing an asset allocation based on expected returns and risk constraints. Solving this classically requires exponential time, but quantum computing can speed up the process.

Here is how AI and quantum computing optimize portfolios:

- AI predicts stock trends, while machine learning models forecast expected returns and risk factors.
- Quantum computing then selects the best portfolio by optimizing the allocations subject to constraints.
- In a hybrid model, AI handles predictions and quantum computing handles allocation.

You can apply this approach in a portfolio-management exercise.

Implementing AI-QC optimization for portfolio management

The hands-on exercise applies AI-quantum computing optimization to portfolio management using synthetic financial data and stock options. Here, AI handles predictions while quantum computing optimizes allocations subject to constraints.

Importing the necessary libraries

Begin by importing the libraries:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from datetime import datetime
from qiskit_finance.data_providers import RandomDataProvider
from qiskit_finance.applications.optimization import PortfolioOptimization
from qiskit_optimization.algorithms import MinimumEigenOptimizer
from qiskit.primitives import StatevectorSampler as Sampler
from qiskit.circuit.library import TwoLocal
from qiskit_algorithms import SamplingVQE
from qiskit_algorithms.optimizers import COBYLA
```

The following libraries are introduced for the first time in this chapter:

- `pandas` as `pd` to store financial data in a table.
- `matplotlib.pyplot` as `plt` to create graphs such as bar charts.
- `datetime` to define the start and end dates for the stock data.
- `RandomDataProvider` creates synthetic stock data.
- `PortfolioOptimization` sets up the stock-selection problem.
- `MinimumEigenOptimizer` finds a stock selection using quantum computing.
- `Sampler` runs quantum algorithms.
- `SamplingVQE` is used to solve stock-selection problems.

Preparing synthetic stock data and defining portfolio constraints

Next, generate the synthetic stock data by setting the number of assets to four and defining the date range from January 1, 2022, to January 1, 2023. The

following code sets these parameters:

```
num_assets = 4

# Define the date range correctly
start = datetime(2022, 1, 1)
end = datetime(2023, 1, 1)
```

It generates synthetic stock movements rather than using live market data; you can use APIs to obtain real data instead.

Initialize the random data provider next. Run the `RandomDataProvider` with the four stock tickers and the start and end dates you defined. Setting `seed=42` makes the synthetic data the same each time you run the program.

The following code runs the provider, retrieves the covariance matrix, calculates expected returns, and prints the matrix as a Pandas data frame:

```
# Initialize Random Data Provider
data_provider = RandomDataProvider(
    tickers=['AAPL', 'GOOGL', 'MSFT', 'TSLA'],
    start=start,
    end=end,
    seed=42
)

data_provider.run()
stock_data = data_provider.get_covariance_matrix()
expected_returns = np.mean(stock_data, axis = 0)

# Convert to Pandas DataFrame for visualization
df = pd.DataFrame(stock_data, columns=['AAPL', 'GOOGL', 'MSFT', 'TSLA'])
print("Stock Covariance Matrix:")
print(df)
```

The covariance matrix measures how much the stocks move together. The expected returns represent the average profit for each stock. A portfolio balances returns against risk, represented here by covariance. Displaying the matrix in a data frame makes the stock data easier to inspect and helps you decide which stocks to select together.

Formulating the portfolio problem and configuring sampling VQE

Set a risk factor and a budget to define the portfolio-selection problem. The risk factor is `0.5`, and you can adjust it to change how much risk you are willing to take. The budget specifies the number of assets to select.

The following code sets the risk tolerance to `0.5` and limits the portfolio to two assets:

```
# Define Portfolio Optimization Problem
risk_factor = 0.5  # Adjust risk tolerance
budget = 2  # Number of assets to select
```

The budget means you select two stocks from the four available. Since investors cannot buy every stock, the budget limits the selection.

Create the portfolio optimization object using the expected returns, covariance matrix, risk factor, and budget. This defines the problem of choosing a stock mix with high returns and low risk while staying within the budget.

The following code passes those values to `PortfolioOptimization`:

```
portfolio = PortfolioOptimization(
    expected_returns=expected_returns,
    covariances=stock_data,
    risk_factor=risk_factor,
    budget=budget
)
```

Convert the portfolio-selection problem into a quadratic program that a quantum computer can solve, then print the problem to inspect it.

The following code creates and prints the quadratic program:

```
quadratic_program = portfolio.to_quadratic_program()
print("\nPortfolio Optimization Problem:")
print(quadratic_program)
```

Define a variational ansatz using a `TwoLocal` circuit. The RY and RZ rotation gates rotate the qubits, and the CZ entanglement block adds entanglement. This circuit provides the structure for the quantum optimization.

The following code creates the ansatz:

```
ansatz = TwoLocal(rotation_blocks=["ry", "rz"], entanglement_blocks="cz")
```

Configure the optimizer and sampler, then use them with the ansatz to set up `SamplingVQE`. Set the optimizer limit to 100 iterations. COBYLA tries different stock selections, the sampler runs quantum computations, and `SamplingVQE` uses these components to search for the stock combination with the best returns and least risk.

The following code configures the optimizer and sampler and initializes `SamplingVQE`:

```
optimizer = COBYLA(maxiter=100)

sampler = Sampler()

sampling_vqe = SamplingVQE(sampler=sampler, ansatz=ansatz, optimizer=optimizer)
```

Solving the portfolio problem and interpreting the selected assets

Use `MinimumEigenOptimizer` to solve the quadratic program, then identify and visualize the selected stocks.

Create the optimizer with `SamplingVQE` and solve the portfolio-selection problem. This code runs the optimization and stores the result:

```
# Solve Portfolio Optimization Problem using Minimum Eigen Optimizer
qaoa = MinimumEigenOptimizer(sampling_vqe)
result = qaoa.solve(quadratic_program)
```

The optimizer uses quantum computing to find the stock combination with the best profit and least risk.

Interpret the result and ensure the portfolio is a one-dimensional array. The following code finds the selected asset indices and uses them to retrieve the corresponding ticker names:

```
# Interpret Results
optimal_portfolio = portfolio.interpret(result)
optimal_portfolio = np.atleast_1d(optimal_portfolio)  # Ensure 1D array

# Select indices of chosen assets
selected_assets = np.where(optimal_portfolio == 1)[0]
selected_tickers = [df.columns[i] for i in selected_assets]
```

Print the selected tickers:

```
# Print selected assets
print(f"\nOptimal Portfolio Allocation: {selected_tickers}")
```

The print statements produce the following output:

```
Stock Covariance Matrix:
        AAPL      GOOGL       MSFT       TSLA
0  32.929644  35.190928  10.995452  26.686611
1  35.190928  75.301763   2.828439  28.476003
2  10.995452   2.828439  20.612460   8.274859
3  26.686611  28.476003   8.274859  32.463091

Portfolio Optimization Problem:
minimize 16.464822152938986*x_0^2 + 35.19092772257926*x_0*x_1 + 10.995452117508314*x_0*x_2 +
26.686610767152448*x_0*x_3 + 37.65088127684887*x_1^2 + 2.8284390279535727*x_1*x_2 +
28.47600271635903*x_1*x_3 + 10.30623019377264*x_2^2 + 8.274859436940233*x_2*x_3 +
16.231545395904455*x_3^2 - 26.4506587282795*x_0 - 35.4492830051474*x_1 - 10.67780274248685*x_2 -
23.975140928065155*x_3 (4 variables, 1 constraints, 'Portfolio optimization')

Optimal Portfolio Allocation: []
```

Let's create a bar chart of expected returns. The chart marks selected assets in green and the other assets in gray:

```
# Visualize optimal portfolio selection
plt.figure(figsize=(8, 5))
plt.bar(df.columns, expected_returns, color=['green' if i in selected_assets else 'gray' for i in
range(len(df.columns))])
plt.xlabel("Assets")
plt.ylabel("Expected Returns")
plt.title("Optimal Portfolio Allocation")
plt.show()
```

The plot produced by this code is shown in the following figure:

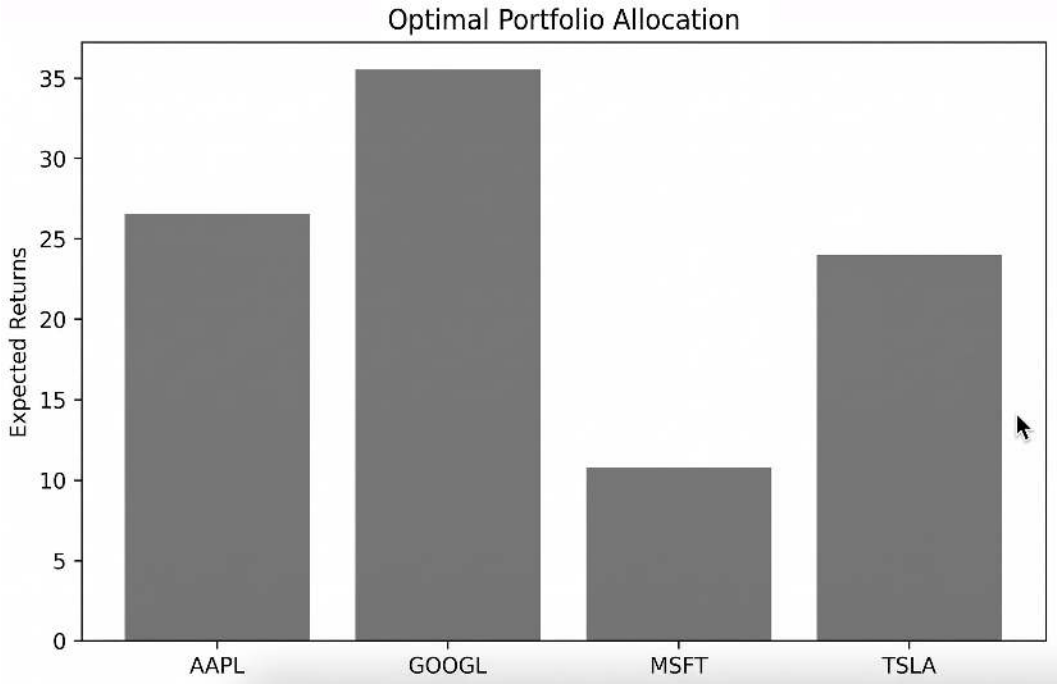


Figure 7.1: Bar chart of expected returns for AAPL, GOOGL, MSFT, and TSLA in the optimized portfolio allocation

You can also use real stock data for this exercise instead of synthetic data.

The displayed results suggest that Apple and Google are among the leading choices, with Tesla close to them. However, the printed portfolio allocation did not identify the selected assets as expected. The sampler may have had an issue, so use the graph to inspect the returns and select the two leading stocks rather than treating the printed allocation as a confirmed result.

This exercise provides a basic simulation of selecting two stocks from synthetic data. It demonstrates how quantum computing could support portfolio selection, though the result shown here did not clearly report the selected tickers. The same approach can be applied to stock data and presented to stakeholders; the exercise also illustrates the potential role of hybrid AI and quantum computing in financial optimization.

Summary

AI and quantum computing are most credible today as partners on well-defined problems, where a quantum routine works alongside a classical optimizer. You saw this in chemistry, where a molecule is converted into qubit operators and VQE searches for its lowest energy, and in finance, where a

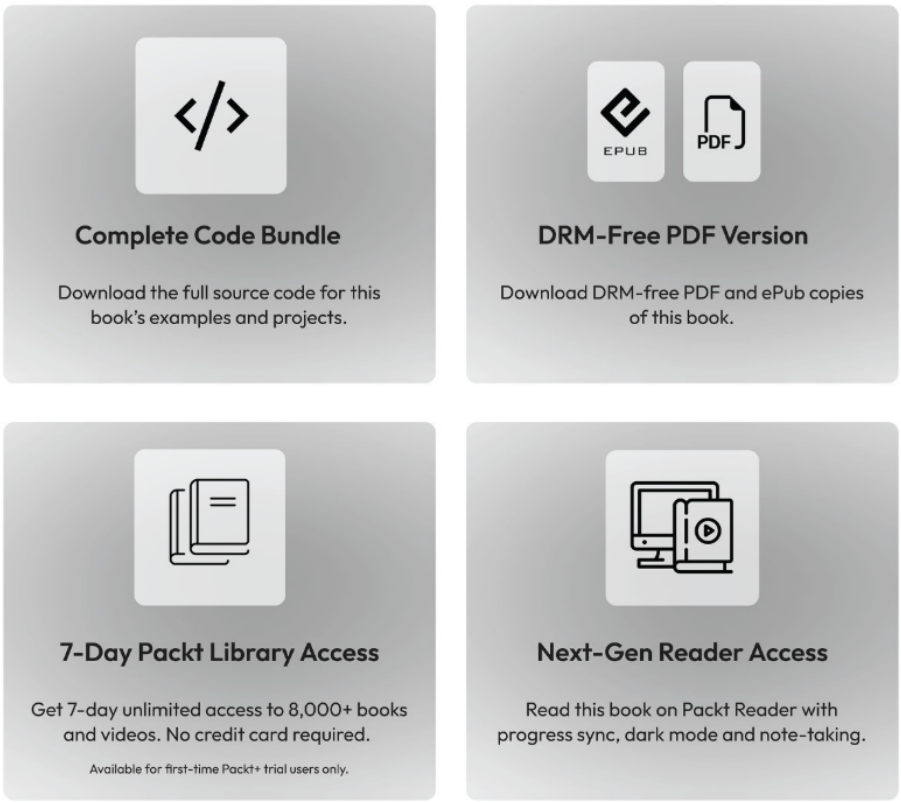
portfolio choice is written as a binary optimization problem. Quantum annealing and D-Wave target the same kind of binary problems, known as QUBO, with a different kind of hardware.

The exercises also show why results need scrutiny. Small simulations on synthetic data demonstrate a workflow, not a quantum advantage, and a result is only as reliable as the problem formulation and code behind it. Before trusting any quantum or AI result, check how the problem was formulated, compare it with a classical baseline, and confirm the output against the problem's constraints.

8

Unlock the Code Bundle and the PDF Version

Your copy of this book includes the following exclusive benefits:



Follow this guide to unlock them. The process takes only a few minutes and only needs to be completed once.

Unlock this book's free benefits in three easy steps

Step 1

Have your purchase invoice ready for *step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

For more help on finding your invoice, visit `https://www.packtpub.com/unlock-benefits/help`.

NOTE

Note: If you bought this book directly from the Packt website, no invoice is required. After *step 2*, you can access your exclusive content right away.

Step 2

Scan the QR code or go to `packtpub.com/unlock`.

On the page that opens (similar to *Figure 8.1* on desktop), search for this book by name and select the correct edition.

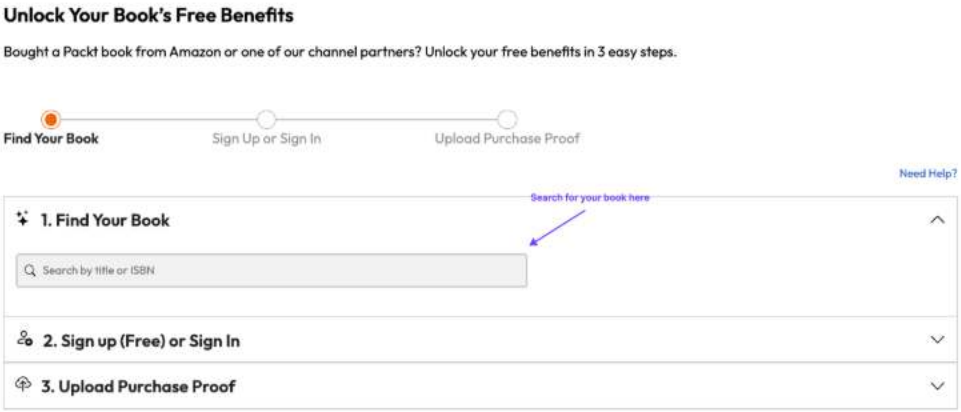


Figure 8.1: Packt unlock landing page on desktop

Step 3

After selecting the book, sign in to your Packt account or create one for free. Then, upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need help?

If you get stuck and need help, visit `https://www.packtpub.com/unlock-benefits/help` for a detailed FAQ on how to find your invoice and more. This QR code will take you to the help page.



NOTE

If you are still facing issues, reach out to `customercare@packt.com`.



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.

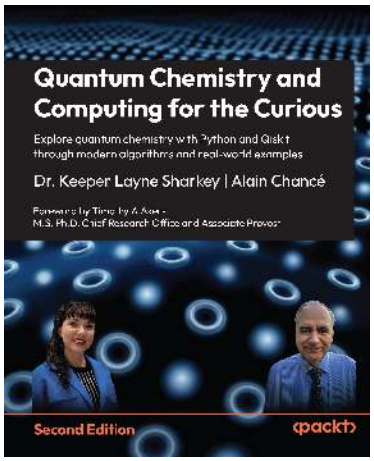
Why subscribe?

- 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:

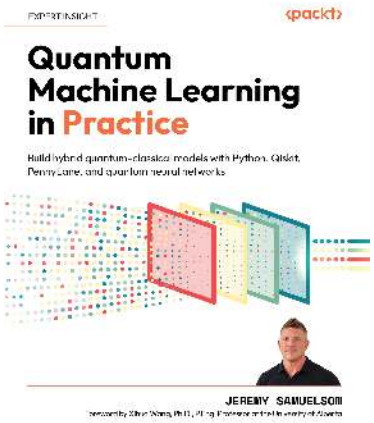


Quantum Chemistry and Computing for the Curious

Dr. Keeper Layne Sharkey, Alain Chancé

ISBN: 978-1-80730-467-6

- Understand quantum mechanics and molecular systems
- Build quantum circuits using Qiskit and Python
- Implement VQE for molecular energy estimation
- Apply error mitigation in noisy quantum systems
- Use optimizers for stable hybrid quantum workflows
- Develop quantum machine learning models for molecules
- Explore advanced algorithms beyond VQE



Quantum Machine Learning in Practice

Jeremy Samuelson

ISBN: 978-1-80760-083-9

- Understand core quantum computing concepts for ML
- Encode classical data into quantum circuits
- Build variational quantum classifiers and regressors
- Implement QML workflows in PennyLane and Qiskit
- Integrate quantum layers in deep learning models
- Design fair benchmarks against classical ML
- Develop end-to-end hybrid quantum ML projects

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packt.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 *Quantum Computing for AI with Python*, we'd love to hear your thoughts! Scan the QR code below 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.



https://packt.link/r/1808088336

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

Index

A

AI predictions
combining, with quantum portfolio allocation 127

AI-QC optimization
implementing, for portfolio management 127

AI-QC optimization, for portfolio management
constraints 128, 129
necessary libraries, importing 127
problem, formulating 129, 130
problem, solving 130–133
sampling VQE, configuring 129, 130
selected assets, interpreting 130–133
synthetic stock data, preparing 128, 129

AI-quantum applications
applying, to financial modeling 121
chemistry simulations, explaining 120
quantum-enabled drug discovery workflows, describing 121

Artificial intelligence and quantum computing (AIQC) **120**

array
reshaping 21

array operations
applying 22

artificial intelligence (AI) **2**
fundamentals 2
key concepts 2
libraries and frameworks, installing 7, 8

B

Bell state circuit
implementing 91, 92

Bidirectional Encoder Representations from Transformers (BERT) **42**

Bloch sphere
qubit, representing on 77, 78

Boltzmann machine (BM) **113**

backpropagation
in deep learning 41

bits **68**

C

CIFAR-10 dataset
loading, for image classification 44

CNOT gate **88**

circuit **71**

classical computers **73**

classification model
building, with scikit-learn 34
data model, evaluating 37
data predictions, making 37
evaluating, with scikit-learn 34
KNN classifier, scaling 36
KNN classifier, splitting 36
KNN classifier, training 36
libraries, importing 34, 35
sample dataset, creating 35

combinatorial optimization
quantum annealing, exploring for 125

convolutional neural network (CNN) model
architecture, defining for digit classification in PyTorch 49
architecture, defining with Keras Sequential API 45
architecture, layers 41
compiling and training 46, 47

for image processing 41
initializing, in PyTorch 50
training, on MNIST data over multiple epochs in PyTorch 50
used, for building image classifier 43

D

D-Wave QUBO hardware
explaining 126

DataFrame **23**
rows, displaying 24

Dirac notation **74**

data
reading, from CSV file into Pandas DataFrame 24

data loaders
creating 49

data preprocessing **30**
categorical values, encoding 32
final processed data, concatenating 33
implementing, with Pandas 31
implementing, with scikit-learn 31
missing values, handling 32
numerical features, scaling 33

deep Q-network (DQN) **53**
used, for training agent 59 – 64
used, for training RL 52

deep learning
backpropagation 41
concepts 12, 13

digit classification
CNN architecture, defining in PyTorch for 49

E

entanglement **69**

F

FrozenLake environment

creating 54 – 56

feature engineering **30**

feature mapping **106**

financial modeling
AI and quantum computing, applying to 121

foundational machine learning
exploring 12 , 13

freeze-core approximation **123**

G

Generative Pre-trained Transformer (GPT) **42**

Grover's search algorithm **93**
implementing, with Qiskit 94 – 98

H

H? geometry **122**

H? ground-state energy
estimating, with VQE 124 , 125

Hadamard gate **87**

hybrid quantum-classical AI model
building 113 – 118

hybrid quantum-classical neural network (HQCNN) **112**

I

image classification
CIFAR-10 dataset, loading for 44

image classifier
building, with CNNs 43
creating, with PyTorch 47 – 52
creating, with TensorFlow 43 – 47

image data
normalizing 44

image processing
convolutional neural network (CNN) for 41

J

Jordan-Wigner transformation **123**

Jupyter Notebook
installing 6, 7

K

Kaggle
reference link 24

Keras Sequential API
used, for defining CNN architecture 45

k-nearest neighbors (KNN) **35**

L

linear regression model **9**
AI model, building 9, 10
AI model, testing 9, 10
quantum circuit, creating in Qiskit 10, 11
quantum circuit, simulating in Qiskit 10, 11

loss function
initializing, in PyTorch 50

M

MNIST dataset
loading 49

N

NumPy **5**
using, for array operations 20
using, for numerical computing 20

NumPy array

creating 21

natural language processing (NLP)
neural networks, applying with RNNs and transformers to 42

neural networks
applying, to NLP with RNNs and transformers 42
exploring 40
structure, layers 40

noisy intermediate-scale quantum (NISQ) **108**

O

OpenCV **5**

one-qubit circuit
building 70, 71

P

Pandas **5**

Pandas DataFrame
creating, from Python dictionary 23
data, reading from CSV file 24

Pandas, for data manipulation and analysis
data, reading from CSV file into Pandas DataFrame 24
DataFrame rows, displaying 24
descriptive statistics for numerical columns, generating 25
missing values, handling in DataFrame 26
Pandas DataFrame, creating from Python dictionary 23
rows, filtering in DataFrame 26
using 23

Pauli gates **86, 87**

PyTorch **5**
CNN architecture, defining for digit classification 49
CNN model, initializing 50
CNN model, training on MNIST data over multiple epochs in 50

loss function, initializing 50
necessary libraries, installing and importing 47, 48
optimizer, initializing 50
used, for creating image classifier 47–52

Python
installing 5
reference link 5

Python dictionary
Pandas DataFrame, creating 23

Python libraries, for machine learning
exploring 20
NumPy for array operations, using 20
NumPy for numerical computing, using 20
Pandas for data manipulation and analysis, using 23
scikit-learn for machine learning workflows, applying 27

portfolio management
AI-QC optimization, implementing for 127

probability 76

Q

Q-learning algorithm 53
used, for training agent 56–58
used, for training RL 52

Qiskit 70

Qiskit for quantum computing
installing 8
verifying 8

quadratic unconstrained binary optimization (QUBO) 126

quantum Boltzmann machine (QBM) 113

quantum algorithms 92
Grover's search algorithm 93
Shor's factoring algorithm 94

quantum annealing
explaining 126
exploring, for combinatorial optimization 125

quantum approximate optimization algorithm (QAOA) **100**

quantum chemistry problem, with Qiskit
H? geometry 122
H? ground-state energy, 124 ,
estimating with VQE 125
libraries, importing 122
qubit Hamiltonian, building 123
simulating 121
VQE, configuring 123

quantum circuit **9**
composing, from gates 88
implementing 89

quantum circuit simulation
measurement results, extracting 91

quantum computing **3** **, ,** **68** **74**
AI capabilities, enhancing with 100
key concepts 3
quantum machine learning 101

quantum experiment **70**
simulating, with shots 71

quantum gates **86**
applying 89
CNOT 88
Hadamard 87
Pauli 86 , 87

quantum kernel methods **101** **,** **107**

quantum kernel-based classifier
training 108 – 112

quantum machine learning (QML) **4** **,** **101**
approaches 101
techniques, identifying 101

quantum measurement **70**
repeating, with more shots 79 , 80
verifying 79

quantum neural networks (QNNs) **4** , **101**
training 112
quantum parallelism **69**
quantum portfolio allocation
AI predictions, combining with 127
quantum speedup **93**
quantum state
inspecting, before measurement 81 , 82
measuring 78
writing, in ket notation 74
quantum-enabled drug discovery workflows **121**
qubit Hamiltonian
building 123
qubits **3** , , **68** **75**
manipulating, with quantum gates 75
measuring 90
representing, on Bloch sphere 77 , 78
state 75

R

Recurrent neural networks (RNNs)
used, for applying neural networks to NLP 42
random matrix
generating 22
reinforcement learning (RL)
components 52
training, with Q-learning and deep Q-network 52
workflow 52
reinforcement learning agents, in Gym
dependencies, installing 54
FrozenLake environment, creating 54 – 56
training, with deep Q-network 59 – 64

training, with Q-learning 56 – 58

S

Series **23**

Shor's factoring algorithm **94**

scikit-learn **5**

scikit-learn, for machine learning workflows
accuracy results, interpreting by 29 ,
modifying input data 30
applying 27
data, splitting into training and testing sets 28
logistic regression model, training on prepared data 28
model accuracy, evaluating 29
predictions, making 29
sample dataset for classification, creating 27 , 28
scikit-learn modules, importing 27

superposition **68**

supervised learning model
versus unsupervised learning 13 , 14
building, for house price prediction 14 – 17

support vector machines (SVMs) **107**

T

TensorFlow **5**
required libraries, installing and importing 43
used, for creating image classifier 43 – 47

Text-to-Text Transfer Transformer (T5) **42**

tech stack **5**

tools and libraries
AI libraries and frameworks, installing 7
Jupyter Notebook, installing 6 , 7
preparing, for AI and quantum computing development 5

Python, installing 5
Qiskit for quantum computing, installing 8
Qiskit for quantum computing, verifying 8
tech stack 5

trained CNN model
accuracy, interpreting 51
evaluating 51

transformers
used, for applying neural networks to NLP 42

U

unsupervised learning
versus supervised learning 13, 14

V

variational quantum circuit (VQC) **112**

variational quantum classifier (VQC) **101**
examining 102
implementing, in Qiskit 102–105

variational quantum eigensolver (VQE) **121**
configuring 123
used, for estimating H? ground-state energy 124, 125

W

wave functions **76**

Untitled

1. Preface
    1. Free benefits with your book
2. Chapter 1: Introduction to Artificial Intelligence and Quantum Computing
    1. Technical requirements
    2. Understanding the fundamentals of artificial intelligence
    3. Understanding quantum computing
    4. Preparing tools and libraries for AI and quantum computing development
        1. Choosing the tech stack
        2. Installing Python and Jupyter Notebook
        3. Installing essential AI libraries and frameworks
        4. Installing and verifying Qiskit for quantum computing
    5. Running hands-on demonstrations with a linear regression model and a quantum circuit
        1. Building and testing a simple AI model using linear regression
        2. Creating and simulating a basic quantum circuit in Qiskit
    6. Exploring foundational machine learning and deep learning concepts
    7. Comparing supervised and unsupervised learning approaches
    8. Building a supervised learning model for house price prediction
    9. Summary
3. Chapter 2: Building Models with Machine Learning and AI Frameworks

1. Technical requirements
2. Exploring foundational Python libraries for machine learning
    1. Using NumPy for numerical computing and array operations
        1. Creating a NumPy array
        2. Reshaping an array
        3. Applying operations on arrays
        4. Generating a random matrix
    2. Using Pandas for data manipulation and analysis
        1. Creating a Pandas DataFrame from a Python dictionary
        2. Reading data from a CSV file into a Pandas DataFrame
        3. Displaying the first rows of a DataFrame
        4. Generating descriptive statistics for numerical columns
        5. Handling missing values in a DataFrame
        6. Filtering rows in a DataFrame based on column values
    3. Applying scikit-learn for machine learning workflows
        1. Importing essential scikit-learn modules
        2. Creating a sample dataset for classification
        3. Splitting the data into training and testing sets
        4. Training a logistic regression model on the prepared data
        5. Making predictions and evaluating model accuracy
        6. Interpreting accuracy results by modifying input data
3. Introducing feature engineering and data preprocessing

        1. Implementing data preprocessing with Pandas and scikit-learn
            1. Handling missing values
            2. Encoding categorical values
            3. Scaling the numerical features
            4. Concatenating the final processed data
    4. Building and evaluating a classification model with scikit-learn
        1. Importing the necessary libraries
        2. Creating a sample dataset
        3. Splitting, scaling, and training with the KNN classifier
        4. Making predictions on data and evaluating the model
    5. Summary
4. Chapter 3: Training Deep Learning and Reinforcement Learning Models
    1. Technical requirements
    2. Exploring neural networks and backpropagation in deep learning
    3. Understanding convolutional neural networks for image processing
    4. Applying neural networks to natural language processing with RNNs and transformers
    5. Building an image classifier using CNNs
        1. Using TensorFlow to create the image classifier
            1. Installing and importing required libraries in TensorFlow
            2. Loading the CIFAR-10 dataset for image classification
            3. Normalizing image data
            4. Defining a CNN architecture using Keras Sequential API

        5. Compiling and training the CNN
    2. Using PyTorch to create the image classifier
        1. Installing and importing the necessary libraries
        2. Loading the MNIST dataset
        3. Creating the data loaders
        4. Defining a CNN architecture for digit classification in PyTorch
        5. Initializing the CNN model, loss function, and optimizer in PyTorch
        6. Training the CNN model on MNIST data over multiple epochs in PyTorch
        7. Evaluating the trained CNN model and interpreting accuracy
    6. Training reinforcement learning with Q-learning and deep Q-networks
    7. Understanding how Q-learning and deep Q-networks learn
    8. Implementing reinforcement learning agents in Gym
        1. Installing dependencies
        2. Creating the FrozenLake environment
        3. Training the agent with Q-learning
        4. Training the agent with a deep Q-network
    9. Summary
5. Chapter 4: Understanding the Foundations of Quantum Computing
    1. Technical requirements
    2. Understanding quantum computing through analogies
        1. Comparing bits and qubits
        2. Explaining superposition
        3. Visualizing entanglement
        4. Exploring quantum parallelism
        5. Describing quantum measurement
    3. Running your first quantum experiment in Qiskit

        1. Getting started with Qiskit
        2. Building a one-qubit circuit
        3. Simulating the experiment with shots
        4. Visualizing the flavor counts
    4. Describing qubits, ket notation, and quantum gates
        1. Contrasting classical and quantum information
        2. Writing quantum states in ket notation
        3. Defining a qubit and its state
        4. Manipulating qubits with quantum gates
    5. Exploring probability, wave functions, and the Bloch sphere
        1. Understanding probability and wave functions
        2. Representing a qubit on the Bloch sphere
        3. Measuring a quantum state
    6. Verifying quantum measurement in Python
        1. Repeating the measurement with more shots
        2. Inspecting the quantum state before measurement
    7. Summary
6. Chapter 5: Building Quantum Circuits and Algorithms
    1. Technical requirements
    2. Exploring fundamental quantum gates and their operations
        1. Understanding Pauli gates: X, Y, and Z
        2. Explaining the Hadamard gate and superposition
        3. Describing the CNOT gate and quantum entanglement
    3. Composing quantum circuits from gates
    4. Implementing basic quantum circuits in Qiskit
        1. Creating and simulating a basic quantum circuit
        2. Applying gates and measuring qubits in Qiskit
        3. Extracting and interpreting measurement results
        4. Implementing a Bell state circuit
    5. Exploring foundational quantum algorithms

        1. Explaining Grover's search algorithm and quantum speedup
        2. Describing Shor's factoring algorithm and its impact
    6. Implementing Grover's algorithm with Qiskit
    7. Summary
7. Chapter 6: Applying Quantum Computing to AI
    1. Technical requirements
    2. Exploring how quantum computing enhances AI capabilities
        1. Comparing classical and quantum machine learning approaches
        2. Identifying key quantum machine learning techniques
        3. Examining variational quantum classifiers
    3. Implementing a VQC model in Qiskit
    4. Processing data with quantum feature maps and kernel methods
        1. Introducing feature mapping
        2. Quantum kernel methods
    5. Training a quantum kernel-based classifier
    6. Advancing AI with hybrid quantum neural networks and quantum Boltzmann machines
        1. Hybrid quantum-classical neural networks (HQCNNs)
        2. Quantum Boltzmann machines (QBMs)
    7. Building a hybrid quantum-classical AI model
    8. Summary
8. Chapter 7: Exploring AI-Quantum Use Cases and Optimization
    1. Technical requirements
    2. Understanding AI-quantum applications across different fields

    1. Explaining AI-assisted quantum chemistry simulations
    2. Applying AI and quantum computing to financial modeling
    3. Describing quantum-enabled drug discovery workflows
3. Simulating a quantum chemistry problem with Qiskit
    1. Importing libraries and defining the H_2 geometry
    2. Building the qubit Hamiltonian and configuring VQE
    3. Estimating H_2 ground-state energy with VQE
4. Exploring quantum annealing for combinatorial optimization
    1. Explaining quantum annealing and D-Wave QUBO hardware
    2. Combining AI predictions with quantum portfolio allocation
5. Implementing AI-QC optimization for portfolio management
    1. Importing the necessary libraries
    2. Preparing synthetic stock data and defining portfolio constraints
    3. Formulating the portfolio problem and configuring sampling VQE
    4. Solving the portfolio problem and interpreting the selected assets
6. Summary

9. Chapter 8: Unlock the Code Bundle and the PDF Version
    1. Unlock this book's free benefits in three easy steps
10. Other Books You May Enjoy
11. Index