INTRODUCTION TO DEEP LEARNING

Mauricio Alberto Ortega Ruiz

Toronto Academic Press

INTRODUCTION TO DEEP LEARNING

Mauricio Alberto Ortega Ruiz

Toronto Academic Press
4164 Lakeshore Road
Burlington ON L7L 1A4
Canada
www.tap-books.com
Email: orders@arclereducation.com

ISBN: 978-1-77956-716-1 (e-book)

ISBN: 978-1-77956-299-9

Toronto Academic Press publishes wide variety of books and eBooks. For more information about Toronto Academic Press and its products, visit our website at www.tap-books.com.

ABOUT THE AUTHOR

Mauricio Alberto Ortega-Ruíz is an Electrical Engineering graduated from UNAM at Mexico, with experience in technical support for electronics equipment, field service and training services. This experience accomplishes instrumentation equipment, photo-lab and automation industry. Mauricio's academic journey includes a M. Sc. in signal processing at the Imperial College of Science Technology and Medicine and a PhD at City University of London, both are UK Universities. His main Research interest is in AI applications for medical imaging analysis and particularly in digital histopathology for breast cancer grading, he has published Scientific papers on this topic and participated in the Automated Gleason Grand Challenge 2022 in which he developed Deep Learning methods for Prostate cancer image grading and obtained the 10th place in the final ranking. He is cofounder of DigPatho, a research group for the LATAM region. Besides his passion for Research in the medical image field he has also demonstrated interest in signal processing, and other imaging applications. He dedicates time to culture and music, and as an amateur violist, he was member of the Imperial College Chamber orchestra during his masters.

Table of Contents

List of Figures

PREFACE

Artificial intelligence has been completely revolutionized by deep learning, which is now present in nearly all business applications. Machine learning algorithms have access to a tremendous amount of data for research since nearly all information and transactions are now recorded digitally. But it's difficult for conventional machine learning methods to investigate the complex correlations found in this so-called Big Data. This is especially true for unstructured data like text, speech, and image.

This book serves as an introductory guide to the field of deep learning, aiming to provide a comprehensive understanding of its principles, methodologies, and applications. Deep learning, a subset of machine learning, has gained prominence due to its ability to automatically learn and extract intricate patterns from data, thereby enabling sophisticated tasks such as image and speech recognition, natural language processing, and autonomous decision-making.

This book comprises eight chapters; the first chapter discusses the fundamental concepts and historical evolution of deep learning, as well as the framework and application of deep learning. The second chapter expands on deep neural network models, exploring multilayer perceptrons, convolutional neural networks, recurrent neural networks, and other advanced architectures like Boltzmann machines and deep autoencoders.

Chapter 3 shifts focus to deep reinforcement learning techniques, elucidating algorithms such as Q-learning, deep Q-networks, and policy gradient methods. The fourth chapter specializes in convolutional neural networks (CNNs), offering a detailed examination of their components such as filters, pooling, and padding. It also discusses the practical implementation of CNNs using TensorFlow.

Chapter 5 introduces the PyTorch framework, focusing on tensors, gradients, and the construction of neural networks using its powerful APIs. The sixth chapter deals with generative deep learning techniques, including text generation, neural style transfer, and variational autoencoders. These methods enable the creation of new content such as images and text, showcasing the creative potential of deep learning models beyond traditional classification tasks. The seventh chapter is about advanced deep learning techniques, such as attention mechanisms and generative adversarial networks (GANs). These techniques enhance model performance by improving focus and generating realistic data, which is critical for tasks in natural language processing, computer vision, and creativity-driven applications. Chapter 8 concludes by examining the practical applications of deep learning

in natural language processing and speech recognition. It covers topics like parsing, distributed representations, knowledge graphs, and multimodal learning, showcasing how deep learning transforms these domains by enabling accurate understanding and generation of human language.

This book has been written specifically for students and scholars to meet their needs in terms of knowledge and to provide them with a broad understanding of business information systems.

—Author

CHAPTER

1 Introduction to Deep Learning

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

- Understand the structure and function of biological neurons and their artificial counterparts.
- Know the historical development of deep learning and its key milestones.
- Understand the architecture and function of feed-forward neural networks, including their layers and activation processes.
- Explore and classify different types of deep learning networks based on their architectural characteristics and applications.
- Understand the functionalities and applications of supervised and unsupervised learning networks.
- Explore the key components and operations of popular deep learning architectures such as CNNs, RNNs, and autoencoders.
- Understand the functionalities and applications of leading deep learning frameworks like TensorFlow, Caffe, and Keras.
- Explore deep learning applications in diverse fields such as healthcare and natural language processing.

KEY TERMS FROM THIS CHAPTER

Autoencoder
Convolutional deep neural network
Deep neural network
Recurrent neural network (RNN)
Tensor Flow
Cognitive toolkit (CNTK)
Deep learning
Natural language processing (NLP)
Supervised learning
Unsupervised learning

UNIT INTRODUCTION

Human brain is a remarkable organ that processes signals from our senses, it possesses an immense capacity to retain a diverse range of experiences, emotions, memories, and possibly dreams. The human brain has a rare capacity to make decisions or solve intricate issues, surpassing even the abilities of the latest supercomputers. Given this, researchers have long aspired to create machines with intelligence like the human brain (Choi et al., 2020). In later research, scientists create robots to aid in human activities, the development of microscopes capable of automatically detecting diseases, and the innovation of self-driving cars. It can autonomously learn and solve increasingly intricate problems at a pace comparable to the brain of a person. These needs contribute to the most vibrant field of artificial intelligence (AI), often referred to as deep learning (Maier et al., 2019).

In this chapter, we introduce foundational principles of deep learning. Beginning with an exploration of neurons and their role in artificial neural networks, we trace the historical evolution of deep learning, highlighting pivotal advancements. The unit explains feed-forward neural networks and their operational mechanisms, alongside the crucial backpropagation algorithm used for network training. We categorize deep learning networks by their specific applications, emphasizing supervised and unsupervised learning paradigms. Additionally, we compare various deep learning frameworks such as TensorFlow, PyTorch, Keras, and Caffe, discussing their features and typical use cases. Finally, the unit examines prominent applications of deep learning in domains like speech recognition, healthcare diagnostics, and natural language processing, demonstrating its transformative impact across diverse fields.

1.1. NEURONS

Neurons are the fundamental building blocks of the human brain. There are tiny sections of the brain, roughly the size of wheat, that contain an impressive number of neurons (over 10,000) each forming over 6,000 connections to other neurons. The brain captures information through neurons, which then process and transmit the result to other cells (Vincent & Hope, 1992). The illustration can be seen in Figure 1.1. Dendrites serve as receptors for the inputs in neurons, resembling an antenna-like structure. The inputs are categorized into strengthened or weakened based on their frequency of usage. The power of a connection defines the extent to which the input influences the output of a neuron. The input signals undergo a process during which multiplication by their respective connection strengths occurs and then combine within the cell body. The outcome from these calculations is transformed into a new signal that travels along the cell's axon until it reaches the neurons at its designated destination (Kempermann, 2012).

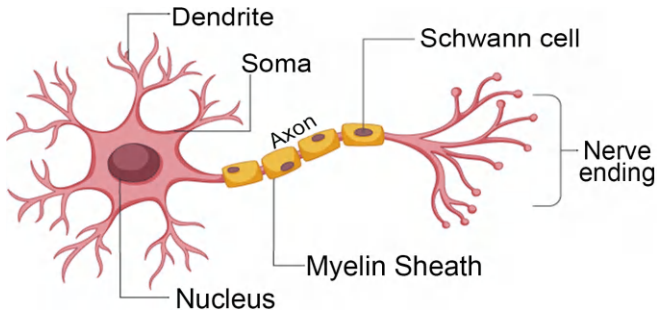


Figure 1.1. Illustration of biological neuron's structure.

Source: Ivilin Stoianov, Creative Commons License.

A research journey to unravel the complex mechanisms of neurons within the human brain was embarked on by Walter H. Pitts and S.Warren McCulloch in 1943. The computer-based artificial design was built, as depicted in Figure 1.2.

In the realm of artificial intelligence, the artificial neuron operates much like its biological counterpart. It takes in a series of inputs, denoted as x_1, x_2, x_3....x_n, and each input is assigned a specific weight, represented as w_1, w_2, w_3....w_n. The neuron then calculates the sum of these weighted inputs, which is used to determine the logit for the neuron (Chinta & Andersen, 2005).

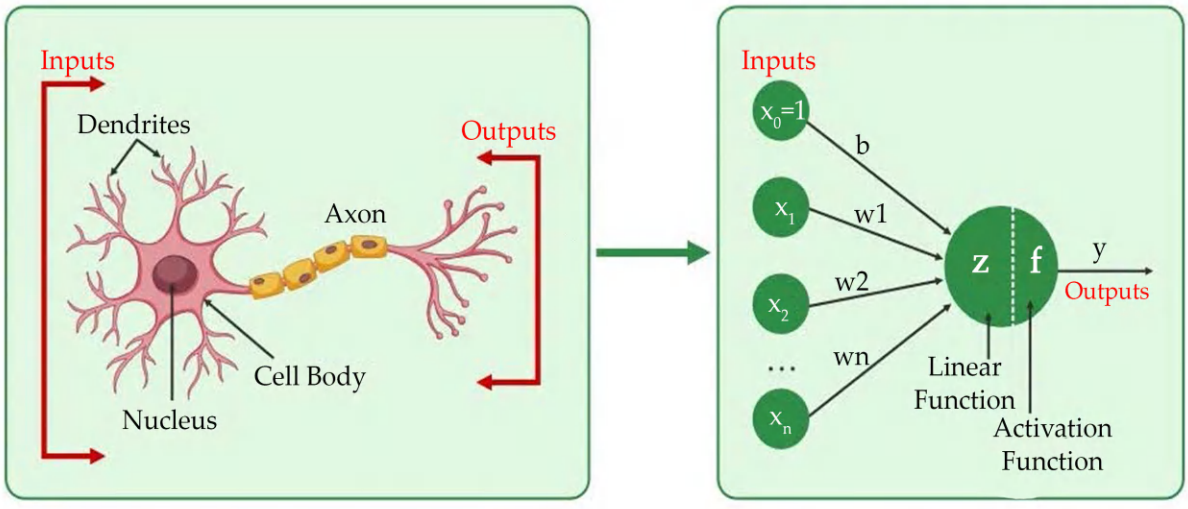


Figure 1.2. Illustration of. neuron in an artificial neural net.

Source: geeksforgeeks.org, Creative Commons License.

$$Z = \sum_{i=0}^{n} W_i \tag{1.1}$$

It is worth noting that certain logit models may incorporate a constant value known as the bias. Eventually, the logit transforms by applying function f to get the desired result y = f(z). A nonlinear function named activation function is applied to the output of the lineal function Z from Eq. (1.1).

1.2. HISTORY OF DEEP LEARNING

The origins of deep learning can be traced back to the early 1940s when Warren McCulloch and Walter Pitts pioneered a computer model that aimed to mimic the complexities of the human neural system. They utilized mathematical principles and algorithms to simulate the cognitive process, which was referred to as "threshold logic." Multiple algorithms are utilized to analyze data and identify objects along with human speech. The output for one layer is fed into the next layer as an input (Schmidhuber, 2015).

In 1960, Henry J. Kelley embarked on the development of the Backpropagation Model, which was later expanded upon from Stuart Dreyfus in 1962. The initial iteration of Backpropagation proved to be particularly efficient or elegant. In the year 1965, Valentin Grigor-evich Lapa published a book on cybernetics and methods for forecasting, whereas Alexey Grigoryevich Ivakhnenko devised a data handling method that incorporated polynomial activation functions (Zhao et al., 2019).

Kunihiko Fukushima made significant contributions to the field of neural networks by developing convolutional networks that integrate pooling and many convolutional layers. In 1979 Neocognitron was developed, an advanced artificial neural network architecture capable of recognizing visual patterns. The Neocognitron was considered the most advanced model of its time due to its utilization of innovative learning techniques and the incorporation of top-down connections. The selective Attention Model can identify distinct patterns. Inference is a concept utilized by Neocognitron to uncover unidentified and missing information (Boroumand & Fridrich, 2018).

Seppo Linnainmaa developed a FORTRAN code for backpropagation in the late 1970s. In the year 1985, Williams and Hinton undertook a study to explore the capacity of backpropagation in generating captivating distribution forms. In 1989, Yann LeCun made a significant breakthrough at Bell Labs by successfully demonstrating the ability of convolutional neural networks to accurately read "handwritten" digits. This achievement was made possible by combining backpropagation with these networks. In the past, numerous researchers expressed great enthusiasm for Artificial Intelligence. One notable example is the work of Dana Cortes and Vladimir Vapnik in 1995, a model known as the support vector machine was introduced. The goal of this model is to accurately analyze and categorize comparable information. In 1997, Sepp Hochreiter and Juergen Schmidhuber presented Long short-term memory (LSTM) as a solution to recurrent neural networks (Smith & Colby, 2007).

A significant turning point in the field of Deep Learning emerged in 1999, with the advent of Graphics Processing Units (GPU) that revolutionized the way one approaches this area of study. Subsequently, in 2000, researchers identified the Vanishing Gradient Problem, which played a crucial role in the ensuing growth of long short-term memory. In 2011 and 2012, AlexNet, a network for convolutional neural networks, attained remarkable success in various international competitions. In the year 2012, Google Brain introduced an innovative project called The Cat Experiment, that effectively addressed the difficulties related to unsupervised learning (Figure 1.3) (Briot, 2021).

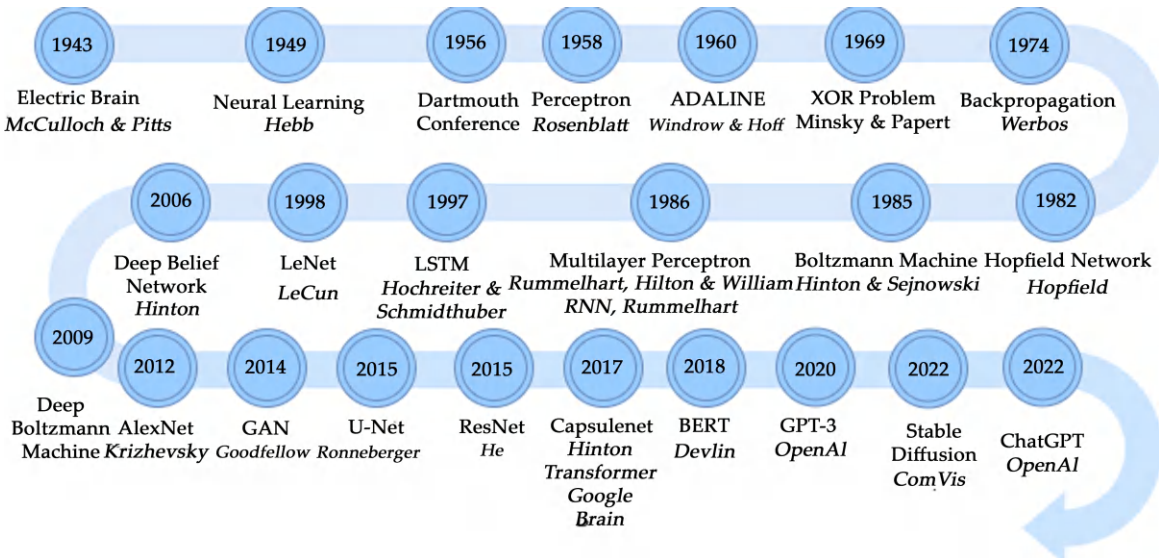


Figure 1.3. Illustration of roadmap of deep learning history.

Source: Anupama Kandala, Creative Commons License.

Remember

Understanding the foundational concepts of neurons, neural networks, and their training algorithms like backpropagation is crucial for effectively implementing and optimizing deep learning models in various applications.

1.3. FEED-FORWARD NEURAL NETWORKS

The arrangement of neurons within the human brain demonstrates an ordered pattern, the cerebral cortex, a crucial component of human intelligence, comprises six distinct cellular layers. Information is transferred between different layers, gradually building up a comprehensive understanding based on the input received from the senses (Svozil et al., 1997). In the same fashion an artificial neural network comprises several layers, that process some input data.

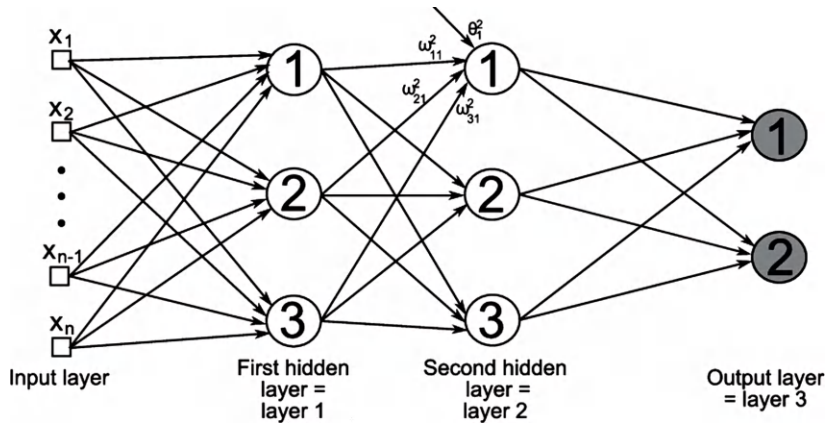


Figure 1.4. Illustration of a neural network at three layers, including two hidden layers, ongoing inputs, and two outputs.

Source: Kai Zhang, Creative Commons License.

Figure 1.4 illustrates a three-layer perceptron, which consists of a hidden layer containing neurons that utilize nonlinear processes of activation. The three-layer perceptron is capable of efficiently handling the computation and decision-making processes for any likelihood function, no matter how complex it might be (Sorin et al., 2020). Also, Figure 1.4 shows the progression of the connection from the lower-level layer up to the higher-level layer. Furthermore, there is a lack of inter-neuronal communication at the same level, and also a lack of communication from upper to lower levels. Thus, this arrangement is commonly known as a feed-forward network. The middle layer in Figure 1.4 depicts the hidden layer, where a neural network operates to solve complex problems (Ilonen et al., 2003).

1.3.1. Backpropagation

Backpropagation is a crucial step in training neural networks and is used to adjust the weights of the network based on the results from the previous epoch. The development of this technique took place in 1970, but it was not until 1986 that researchers truly recognized its potential. This breakthrough came with the publication of a paper by Ronald Williams, David Rumelhart, and Geoffrey Hinton. In their work, it was revealed that backpropagation not only operates at a faster pace but also offers solutions to problems that had previously remained unsolved. Backpropagation is a widely used technique in the field of Artificial Neural Networks (ANNs) that enables supervised learning. It has been widely used in different fields, including categorization pattern recognition, and medical diagnostics. The Backpropagation technique has facilitated the integration of multilayer perceptron networks into neural network research, making them an indispensable tool (Lillicrap et al., 2020).

The backpropagation algorithm calculates the partial derivatives over the result parameter. These derivatives are denoted as $\frac{\partial f}{\partial w_i}$, where w_i represents the ith parameter along with f represents the output. Take a look at a multiple-layer feedforward neural network depicted in Figure 1.2. Suppose there is a neuron i within the output layer while the Equation 1.2 represents the error signal generated during the m^{th} iteration (Werbos, 1990).

$$e_i(m) = d_i - y_i(m) \tag{1.2}$$

where d_i represents the desired output over neuron i and $y_j(m)$ represents the actual output over neuron i. The actual output is computed utilizing the current weights for a network at iteration m.

The equation represents the instantaneous error energy value for neuron i.

$$\varepsilon_i(m) = \frac{1}{2} e_i^2(m) \tag{1.3}$$

Equation 1.3 represents the total of all $\mathring{a}_i(m)$ values for every neuron in the output layer, resulting in the current value $\mathring{a}_i(m)$.

$$\varepsilon_i(m) = \frac{1}{2} e_i^2(m) \tag{1.4}$$

where S represents the collection of neurons that comprise the output layer.

Let's consider a scenario where one has a training set with N patterns. Equation 4 provides us with an average square energy used by the network (LeCun et al., 1988).

$$\varepsilon_{avg} = \frac{1}{N}\sum_{n=1}^{N}\varepsilon(m) \tag{1.5}$$

The backpropagation technique is implemented in two ways: batch mode and sequential mode. Weight updates are performed in the batch mode once an epoch is finished. However, updates for either sequential mode or stochastic mode occur after each training example is given (Wang et al., 2020). Here is the Equation 1.6 that provides the output expression over neuron i.

$$y_i(m) = f\sum_{i=0}^{n} w_{ij}(m)\, y_i(m) \tag{1.6}$$

In this context, 'm' denotes the overall amount of inputs on the neuron 'i' from the layer before it, while 'f' represents the activation function used in neuron 'i.'

The weight update for neuron i is determined by a partial derivative for the error energy ε_{η} concerning the corresponding weight. This update is proportional to the derivative.

$$\frac{\partial_{\varepsilon}(m)}{\partial_{wij}(m)} \tag{1.7}$$

Expressed through a chain rule of calculus

$$\frac{\partial_{\varepsilon}(m)}{\partial_{wij}(m)} = \frac{\partial_{\varepsilon}(m)}{\partial_{ei}(m)}\frac{\partial_{ei}(m)}{\partial_{yi}(m)}\frac{\partial_{yi}(m)}{\partial_{wij}(m)} \tag{1.8}$$

Equation 1.9 is derived from Equations 1.2, 1.1, and 1.4

$$\frac{\partial_{\varepsilon}(m)}{\partial_{ei}(m)} = e_i(m) \tag{1.9}$$

$$\frac{\partial_{ei}(m)}{\partial_{yi}(m)} = -1 \tag{1.10}$$

$$\frac{\partial_{yi}(m)}{\partial_{wij}(m)} = f\sum_{i=0}^{n} w_{ij}(m)\, y_i(m)\frac{\partial\sum_{i=0}^{n} w_{ij}(m)\, y_i(m)}{\partial_{wij}(m)}$$

$$f\sum_{i=0}^{n} w_{ij}(m)\, y_i(m)\, y_i(m) \tag{1.11}$$

Where

$$f'\sum_{i=0}^{n} w_{ü}(m)\, y\ (m) = \frac{\partial f\sum_{i=0}^{m} w_{ü}(m)\, y\ (m)}{\partial\sum_{i=0}^{m} w_{ü}(m)\, y\ (m)}$$

By substituting the Equations (1.7), (1.8), and (1.9) into Eq. (1.8), the following expression is obtained.

$$\frac{\partial_{\varepsilon}(m)}{\partial_{wij}(m)} = e_j(m)\, f'\partial\sum_{i=0}^{m} w_{ij}(m)\, y_i(m)\, y_i(m) \tag{1.12}$$

The Delta rule is utilized to calculate the correction $w_{ij}(m)$ while is expressed as

$$_{wij}(m) = -\eta\frac{\partial_{\varepsilon}(m)}{\partial_{wij}(m)} \tag{1.13}$$

where η represents an immutable parameter utilized to ascertain the pace of learning in the backpropagation process (Leung & Haykin, 1991).

1.4. CLASSIFICATION OF DEEP LEARNING NETWORKS

There are three different classes of deep learning networks, each tailored to specific applications such as synthesis, classification, and recognition. These classes are determined by the techniques and architectures employed.

- Unsupervised Deep Learning Network;
- Supervised Deep Learning Network;
- Hybrid Deep Learning Networks (Li et al., 2019).

An unsupervised deep learning network is capable of capturing higher-order correlation facts for synthesis purposes even in cases where a clear target class is not defined. The effectiveness of pattern classification in supervised training of deep networks relies on accurately representing a distribution of classes based on the available data. Hybrid networks combine the two discriminative and generative components. In addition, a combination model is formed by merging similar components, and some authors consider reinforcement as one category of deep learning in which the learning process is performed by a obtaining a maximal reward (Simon Prince, 2024).

1.5. DEEP LEARNING ARCHITECTURE

One delves into the architecture for deep learning and examines the different approaches that are commonly used in this field. Representation of the input data plays a vital role in the realm of deep learning. In the conventional approach, the input features are derived from the original dataset and incorporated into machine learning algorithms. The various stages in the process of engineering involve creating, analyzing, selecting, and evaluating the necessary features, which can be quite time-consuming and labor-intensive. This enables the identification of latent connections among the data that may otherwise remain concealed or unfamiliar (Sewak et al., 2020).

Complex data representation in deep learning often involves expressing it as compositions of simpler forms and usually represented as a vector input through the initial layer. The majority of deep learning algorithms are built upon the conceptual framework of Artificial Neural Networks (ANN). These networks consist of interconnected nodes referred to as "neurons," which are organized in layers as depicted in Figure 1.5. Hidden units, which are not present in these two layers, store the set of weights (Li et al., 2016) and in Figure 1.6 are examples of some popular deep network architectures.

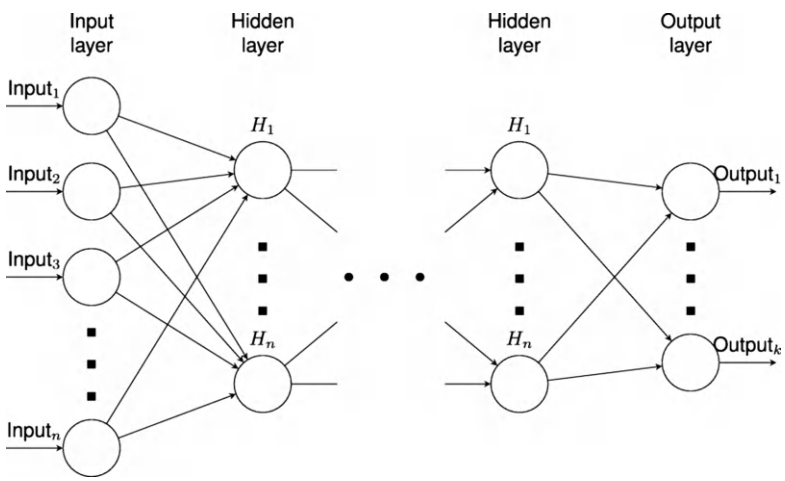


Figure 1.5. Illustration of neural network with hidden or output layers and 1, 2, 1 input.

Source: Weitian Tong, Creative Commons License.

Performance of the network is computed by the loss function, and it measures how well the network models the feature data. For a given set of observations, the negative log-likelihood can be defined as

$$NLL = -\sum_{i=1}^{N} \log p(y_i | x_i; \theta | \tag{1.14}$$

Did you know?

Deep learning models are often characterized by their ability to automatically learn representations from data, which allows them to perform tasks such as image recognition, natural language processing, and even autonomous driving with impressive accuracy.

Where N is the number of observations; yi is the true label for observation i; xi is the input features for observation i. To improve the weights of an ANN, the loss function is minimized, including the negative log-likelihood, shown in Eq. (1.14b).

$$E(\theta, D) == -\sum_{i=1}^{N} \left[\log p(y_i | x_i; \theta | \right] + \lambda \| \theta \| p \tag{1.14b}$$

The first term aims to minimize the overall log loss across the entire training dataset D.

The second term aims to minimize the p-norm of the learned parameter θi, and its impact can be adjusted by tuning the parameter λ. Regularization is a technique that helps prevent a model from overfitting, ensuring its generalizability. Navigating in reverse towards the last layer across the network is an essential step in this process (Ahn, 2016). Numerous open-source tools are at your disposal for the implementation of deep learning models, such tools are Keras3, Deeplearning4j8, –TensorFlow1, PyTorch5, CNTK7, Theano2, Caffe6, and Torch4.

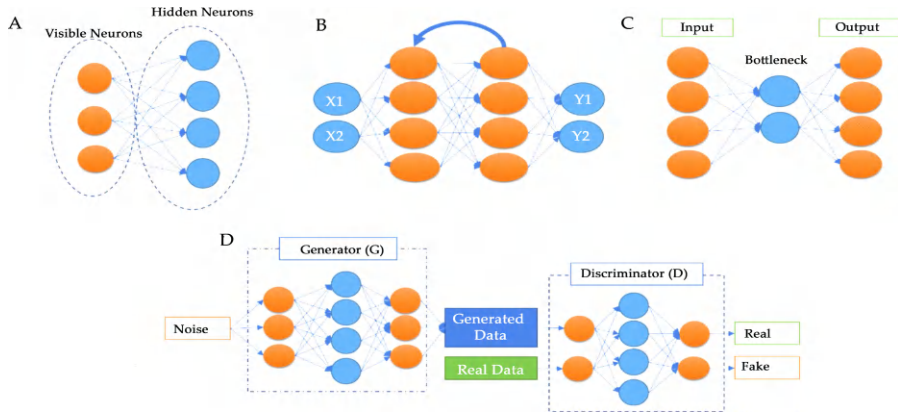


Figure 1.6. Illustration of a brief overview of popular deep learning architectures. (A) Restricted Boltzmann machine. (B) Recurrent neural network (RNN). (C) Autoencoders. (D) Generative adversarial networks.

Source: Manju Khari, Creative Commons License.

1.5.1. Supervised Learning

1.5.1.1. Multilayer Perceptron (MLP)

The multilayer perceptron is composed of several hidden layers, with each layer's neurons being fully connected with the neurons in the subsequent layer. This type of network has a restricted capacity for hidden layers but only permits data to flow in a single direction.

Eq. (1.15) illustrates the utilization of an activation function that is non-linear σ to process the computed sum.

At this stage, the value of d represents the number of units within the previous layer, while xj represents the output received through the j_{th} node of the previous layer. The terms b_{ij} or w_{ij} are commonly referred to as bias or weight terms, respectively, which are linked to each x_{ij}. In traditional networks, the nonlinear functions for activation typically include tanh or sigmoid. However, in more contemporary networks, (ReLU) Rectified Linear Units have become a popular choice (Desai & Shah, 2021).

A multilayer perceptron consists of several hidden layers, in which

$$hi = \sigma\left(\sum_{j=1}^{d} x_j w_{ij} + b_{ij}\right) \quad (1.15)$$

By fine-tuning the weights in the hidden layers while training, a robust relationship between the input x and the output y is established. Despite being one of the more straightforward models compared to other learning architectures that utilize fully connected neurons within the final layer (Mia et al., 2015).

1.5.1.2. Recurrent Neural Network (RNN)

A Recurrent Neural Network (RNN) seems a specific type of Neural Network that utilizes the output of the previous step as input for the current step. In conventional neural networks, both inputs and outputs are regarded as separate entities, with no interdependence. It has an important application in natural language processing. However, when it is necessary to anticipate the next word in a sentence, it becomes essential to retain the preceding words to make accurate predictions (Banerjee et al., 2019). As a result, RNN was developed to address this problem by incorporating a Hidden Layer. One of the key aspects of RNN is the Hidden state, which retains important information for a sequence. Uniform parameters are employed for each input, enabling the consistent execution of tasks across any inputs or just hidden layers, ultimately generating the output (Tarwani & Edem, 2017). This simplifies the parameter complexity, contrary to other neural networks. Recurrent Neural Networks encompass (Figure 1.7):

- Long Short-Term Memory (LSTM);
- Gated Recurrent Units (GRU).

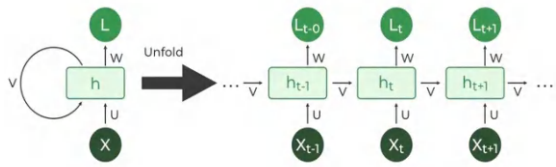


Figure 1.7. Illustration of recurrent neural network.

Source: geeksforgeeks.org, Creative Commons License.

1.5.1.3. Convolutional Neural Network (CNN)

Deep learning has gained significant popularity in recent years, especially in the

field of image processing. It has been inspired by the structure and function of the human and some animal visual systems, particularly the visual cortex. Raw data on CNN is subject to local connectivity.

For instance, a more comprehensive understanding of the image can be Attained by perceiving it to be a compilation of local pixel patches, that are processed to a convolution operation When examining a 1-dimensional time series, it can be viewed as a compilation of the local signal segments (Kattenborn et al., 2021). Here, one presents an Eq. (1.16) over 1-dimensional convolution,

$$C_{1d} = \sum_{a=-\neg}^{\neg} x(a).w(t-a) \quad . \tag{1.16}$$

where, the input signal is denoted as x, while the weight function or convolution filter is represented by w.

Here is the Eq. (1.17) in 2-dimensional convolution, with k representing the kernel and X representing a 2-D grid.

$$C_{2d} = \sum_{m}\sum_{n} X(m,n)K(i-m,j-n) \tag{1.17}$$

Acquiring feature maps requires calculating weights within the input using a filter or kernel. CNN utilizes sparse interactions, Filters in deep learning academic writing are often smaller than the input.

Encouraging parameter sharing in CNN is a natural choice, as each filter serves the entire input. The CNN architecture includes two convolutional layers that are accompanied by a pooling layer, as shown in Figure 1.8. Convolutional neural networks (CNNs) demonstrate optimal performance when utilized for computer vision tasks (Wu, 2017).

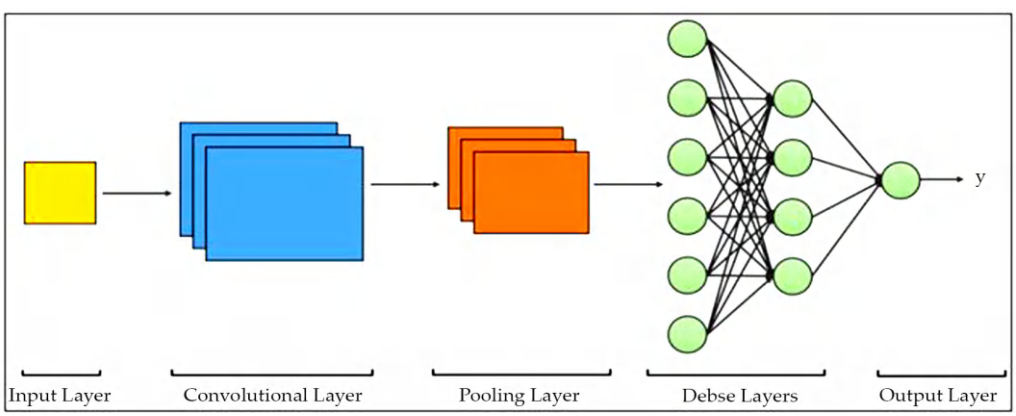


Figure 1.8. Illustration of CONVNET and output layers.

Source: Ashley, Creative Commons License.

1.5.2. Unsupervised Learning

1.5.2.1. Autoencoder (AE)

An autoencoder (AE) has emerged as a prominent model in the field of deep learning, illustrating the concept of unsupervised learning through visual representations. At first, it was primarily used for supervised learning models when labeled data were scarce. In AE, the input undergoes an encoding process that transforms it into a lower-dimensional z space (Guo et al., 2020). This encoded representation is then decoded to reconstruct the original input x. Thus, the process of encoding and decoding in an encoder can be represented by an Equation 1.18 involving a single hidden layer. The weights that have been encoded and decoded are denoted as W and W0, respectively. The main goal is to reduce the reconstruction error. Z is a dependable and encoded form (Zou et al., 2021).

$$z = \sigma\left(Wx + b\right) \tag{1.18}$$

$$\bar{x} = \sigma\left(W'x + b'\right) \tag{1.19}$$

Once an AE is properly trained, a single input is fed into the network, activating the innermost layer that is hidden for serving as the input of the encoded representation (Figure 1.9) (Xu et al., 2021).

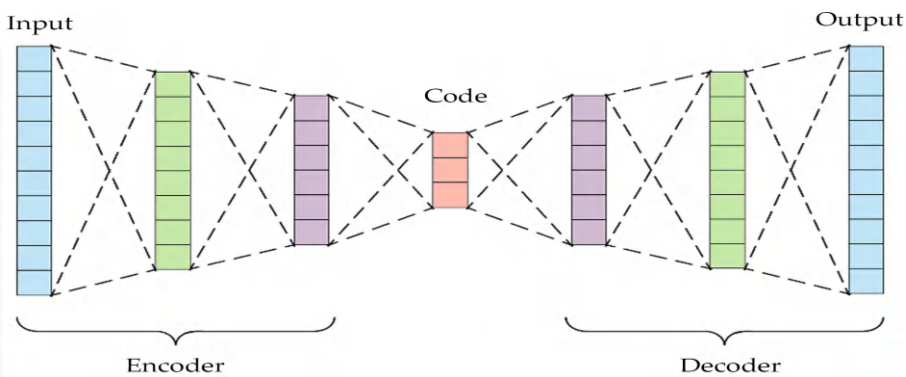


Figure 1.9. Illustration of autoencoder.

Source: Arden Dertat, Creative Commons License.

The input data is transformed into a format enabling AE to record the essential derived dimensions. It shares similarities with conventional dimensionality reduction techniques including (PCA) Principal Component Analysis and (SVD) Singular Value Decomposition. The stacking approach is a technique used for learning Deep Autoencoder networks (Fan et al., 2019).

- Sparse Autoencoder (SAE);
- Variational Autoencoder (VAE);
- Denoising Autoencoder (DAE).

1.5.2.2. RBM Restricted Boltzmann Machine

The RBM is a highly effective framework for acquiring a model of input data through unsupervised learning. It shares similarities with AE, however RBMs calculate the probability distribution derived from the input data provided. Ss a type of generative stochastic artificial neural network that can learn a probability distribution over its set of inputs. The canonical (RBM) seems a model that includes hidden units h or binary visible units v. It also includes an energy function, which is represented by an Equation 1.20 (Larochelle et al., 2012).

$$E\left(v,h\right)=-b^{T}v-c^{T}v-Wv^{T}h \qquad (1.20)$$

In the Boltzmann Machine (BM), all units are fully connected. The training of a Restricted Boltzmann machine usually involves a stochastic optimization technique such as Gibbs sampling. The process results in a comprehensive representation of the input data, which is considered the final expression of h. In addition, RBMs can be arranged hierarchically, allowing for the creation of a Deep Belief Network (DBN) that is especially useful for supervised learning (Zhang et al., 2018).

1.6. DEEP LEARNING FRAMEWORKS

There are many software packages available to help researchers streamline the development of deep learning architectures. There are several software packages available for deep learning, such as Microsoft Cognitive Toolkit (known as CNTK), Torch, Keras, and Deep learning4j, Tensorflow, Caffe, Neural Designer, H2o.ai, and DistBelief had a major effect on the business (Figure 1.10) (Nguyen et al., 2019).

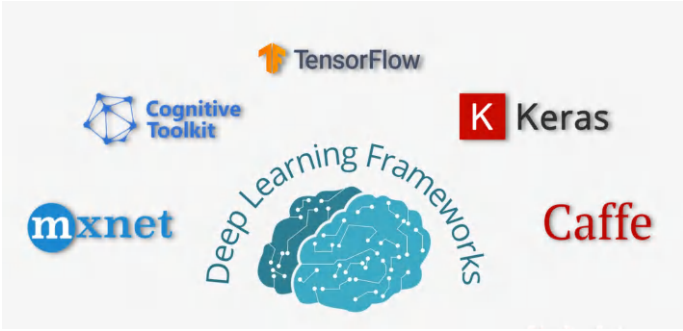


Figure 1.10. Illustration of deep learning framework.

Source: Mahesh, Creative Commons License.

1.6.1. Tensorflow

The concept of Tensors is closely linked to the mathematical principles used in engineering and physics. Over time, Tensors have also become relevant in computer science, particularly in the realm of logic and discrete mathematics. TensorFlow, an open-source library, provides a comprehensive platform for both production and research in machine learning. It provides APIs for both experienced and novice learners to create applications for various platforms such as cloud, mobile, web, and desktop. TensorFlow suggests using the Keras API for developing and training deep learning models, especially for beginners. When it comes to more complex tasks. The structure of TensorFlow is organized into three main components: data processing, model creation, and model training and evaluation (Le Glaz et al., 2021). The inputs are accepted as tensors or multi-dimensional arrays. A comprehensive operation flowchart is constructed to illustrate the various operations applied to the input. Eventually, the desired output is obtained. Thus, the name TensorFlow is derived from the concept of tensors flowing through a series of operations to generate the desired output. TensorFlow utilizes static graph computation

to facilitate the visualization of the neural network structure using the Tensorboard. TensorFlow offers a range of algorithms for various tasks, including boosted tree regression, classification, boosted tree classification, and linear regression. These algorithms provide powerful tools for data analysis and modeling (Mandal & Vipparthi, 2021).

1.6.2. Microsoft Cognitive Toolkit

Microsoft Cognitive Toolkit represents an invaluable resource for academic writing in the realm of deep learning. The concept of neural networks is illustrated as a series of computational steps, represented by a directed graph. The previous iteration of this toolkit is known to be the Computational Network Toolkit CNTK. The most recent release is CNTK v.2.0 Beta 1, which offers enhanced C++ and Python APIs and employs BrainScript as its specialized language. The libraries within the Computational Network Toolkit have been built using C++. The Python APIs offer a sophisticated interface for defining models, implementing learning algorithms, reading data, and performing distributed training. CNTK 2 is considered a valuable addition to Python API, offering the protocol buffers serialization feature created by Google. CNTK 2 now has the Fast R-CNN algorithm, which enables efficient object detection. The Fast R-CNN algorithm utilizes the concept of reusability by incorporating a return on investment (ROI) pooling scheme to reuse computations using the convolution layers (Etaati & Etaati, 2019). The Microsoft Cognitive Toolkit is a powerful tool for training neural networks to categorize and recognize images, text, and speech. It is open-source and designed to work efficiently across multiple GPUs and machines. Microsoft Cognitive Toolkit serves as the foundation for various Microsoft products, including Xbox, Skype live translation, Bing, and Cortana. The platform being used offers a variety of neural network types, including Feed Forward Network (FFN), Convolutional Neural Network (CNN), and Sequence-to-Sequence with attention, Recurrent/Long short-term memory (RNN/LSTM). The Microsoft Cognitive Toolkit provides a variety of functionalities, such as assistance for automatic hyper-parameter tuning, reinforcement learning, unsupervised learning, and generative adversarial networks, for that are generative (Rajendran, 2023).

1.6.3. Caffe

The convolution Architecture for Feature Extraction (Caffe) has a component of a deep learning framework created by Berkeley AI Research. The speed, capabilities, and design of this are truly remarkable. Caffe's impressive speed makes it a valuable tool both for industry development and academic research. With only one NVIDIA K40 GPU, it has the impressive capability to process almost 60M images per day. The flexible code feature encourages ongoing development. Embracing a dynamic architectural approach fosters creativity and practical implementation. There is a deliberate effort to reduce the reliance on hard coding during the development of models (Garea et al., 2019).

1.6.4. Deep Learning 4j (DL4J)

Deep Learning4j acts as a freely accessible open-source library that provides an extensive answer for deep learning across various applications. The software is built on the Java language and enables advanced predictive analysis and knowledge exploration on both CPU and GPU (Graphics Processing Units). DeepLearning4j combines

the power of artificial intelligence (AI) algorithms with a range of practical applications, including cyber forensics, predictive analysis, business intelligence, network intrusion detection and prevention, robotic process automation (RBA), recommender systems, Face recognition, regression, and other topics are covered. DeepLearning4j can import models from well-known deep learning structures like Keras, Theano, Caffe, and TensorFlow. It seamlessly supports both Python and Java programming languages, ensuring smooth compatibility. DeepLearning4j offers a range of features. It is built on a microservices architecture, allowing for scalability on Hadoop (an open-source framework designed for distributed storage and processing of large datasets across clusters of computers) for handling large amounts of data. Additionally, it can utilize GPUs for enhanced scalability upon the Amazon Web Services (AWS) cloud. It provides APIs over Java, Python, and Scala, and is compatible with both CPUs and GPUs. DeepLearning4j is capable of processing massive amounts of data by utilizing clusters (Deng & Yu, 2014). The DeepLearning4j framework includes the following libraries and components: ND4j, JavaCPP, DataVec, and RL4J. ND4j is a powerful tool that combines the functionality of NumPy with the (JVM). ND4j is an impressive library that provides efficient processing of matrix data and numerical computations. It demonstrates exceptional proficiency in handling complex concepts and tasks, including various mathematical operations, data manipulation, and algorithmic optimization, with a focus on performance. JavaCPP is a versatile tool that allows seamless communication between C++ and Java, eliminating the need for any additional third-party or intermediate applications (Mathew et al., 2021). DataVec is a powerful tool that simplifies the process of transforming raw data into a vector format. It also includes preprocessing capabilities to ensure that the data is ready for training in Machine Learning applications. It can handle a wide range of data types, including binary files, videos, images, text documents, and CSV files. RL4J is a powerful tool for Java platforms that incorporates cutting-edge techniques such as Deep Q Learning and Asynchronous Actor-Critic Agents (A3C) (Min et al., 2017).

1.6.5. Keras

Keras, an open Source Neural Network library developed by François Chollet, offers a range of efficient, adaptable, and intuitive capabilities on the Python platform. It seamlessly integrates with Tensorflow or Theano, making it a powerful tool for deep learning. Backend is a powerful library in Keras that efficiently manages the intricate computations required for advanced tasks. Keras, on the other hand, is focused on providing an advanced API wrapper. The backend carries out the necessary computations at a lower level, utilizing tools such as Theano or Tensorflow (Lee & Song, 2019). It is compatible with Theano, CNTK, or TensorFlow. The Keras High-Level API handles the compilation of the model, integration of the optimizer along with loss functions, and management of the training process using the fit function. Keras offers extensive platform and device support, allowing for deployment on various platforms such as Web browsers with iOS with CoreML, Android with Tensorflow Android, js support, Raspberry Pi, and Cloud engine. Keras is capable of efficiently processing large amounts of data, allowing for parallel data processing. This feature greatly accelerates the training process (Chicho & Sallow, 2021).

1.6.6. Neural Designer

Neural Designer has become a powerful tool that simplifies the implementation of analytic algorithms, making them more manageable. The software is equipped with a user-friendly interface that effectively guides the workflow and produces precise outcomes. It is straightforward to manage without the need for programming or block diagrams.

The user interface provides clear instructions and guides the user through the process (Almási et al., 2016). The Neural Viewer is an exceptional visualization tool that presents accurate results through exportable charts, tables, and pictures. By utilizing advanced data pre-processing techniques, the process of calculating principal components and removing outliers becomes more streamlined. Neural Designer offers users the ability to create highly effective predictive models using a range of error and regularization techniques. Furthermore, advanced techniques such as Levenberg-Marquardt and quasi-Newton methods are incorporated to enhance the accuracy of calculations. It offers faster processing speed and improved memory management through CPU parallelization, thanks to GPU acceleration using CUDA and OpenMP (Koitmäe et al., 2018).

1.6.7. Torch

Torch, written in Lua, is a programming environment similar to Matlab that is used for both deep and shallow machine learning solutions. It offers a versatile way to build tensors, which are mathematical objects used in machine learning. The software possesses several characteristics, including the ability to automatically calculate gradients, the capacity to store numerous backend tensors for enhanced CPU/GPU calculation, and support for high-level language, enabling rapid prototyping (de Jong et al., 2013).

1.7. DEEP LEARNING APPLICATION

1.7.1. Speech Recognition

Speech recognition harnesses the principles of deep learning and emerges as a leading application in the field, leveraging its immense potential with careful precision. In 2010, the emergence of deep learning left a lasting impact on the field of speech recognition. The traditional recognition of speech systems is Gaussian Mixture Models (GMMs), which rely on hidden Markov models (HMMs). In this context, the speech signal is viewed as a short-time stationary signal or a piecewise stationary signal. As a result, the Markov model is well-suited for this particular application. One drawback of this approach is its inefficiency in modeling non-linear functions. Unlike HMMs, neural networks have demonstrated their effectiveness in discriminative training. In 2012, Microsoft unveiled an innovative iteration of their Microsoft Audio Video Indexing Service (MAVIS) that utilized deep learning technology. The findings presented by Microsoft demonstrate the effectiveness of deep learning in reducing word error rate (WER) compared to Gaussian mixtures (Gaikwad et al., 2010).

1.7.2. Deep Learning in HealthCare

The healthcare system is entering a new era, harnessing cutting-edge technologies to deliver timely and appropriate treatment to patients. A recent study discovered that utilizing multi-layer neural networks to analyze pharmaceutical data leads to highly precise predictive outcomes in a range of clinical contexts. The architecture of deep learning is built upon a hierarchical learning structure, allowing it to effectively integrate diverse data sources and generate more comprehensive generalizations (Amin et al., 2021). Several studies have demonstrated the potential of deep learning in revolutionizing the healthcare system of the future. By analyzing vast amounts of patient data, deep learning algorithms can predict diseases, create personalized prescriptions, suggest clinical trials, and provide treatment recommendations. Wang et al. achieved a remarkable feat by utilizing temporal deep learning conduct to emerge victorious in the Parkinson's Progression Marker's Initiative data challenge (Miotto et al., 2018). Their groundbreaking work involved identifying the various subtypes of Parkinson's disease. Conventional methods, which rely on matrices or vectors, are not regarded as optimal due to the progressive nature of Parkinson's disease and the challenges in identifying patterns of

disease progression. In addition, Wang et al. discovered three additional subtypes of Parkinson's disease by utilizing an LSTM RNN model. This finding highlights the power and promise of models based on deep learning in addressing healthcare challenges (Malik et al., 2020).

The deep learning architecture commonly used in healthcare systems primarily consists of Recurrent Neural Networks (RNNs), Autoencoders (AEs), convolutional neural networks (CNNs), and Restricted Boltzmann Machines (RBMs). A notable application is the use of deep learning algorithms to predict Alzheimer's disease through the analysis of Magnetic Resonance Imaging (MRI) scans. Low-field knee MRI can be hierarchically represented using CNNs to automatically perform cartilage region segmentation, enabling the detection of osteoarthritis risk. Deep learning is employed to partition multiple sclerosis lesions within multi-channel 3D MRI scans to forecast malignant and benign breast lumps. Deep learning is applied to analyze data from Electronic Health Records (EHRs) in many areas such as laboratory tests, diagnoses, prescriptions, and clinical notes written in free-text format (Almutairi et al., 2022). The utilization of the Area under the Receiver Operating Characteristic Curve and F-score approach demonstrates that the accuracy of deep learning surpasses that of the classical machine learning process. Choi et al have created a remarkable model called Doctor AI, which utilizes advanced RNNs with gated recurrent unit (GRU) to accurately predict diagnoses and recommend appropriate medications. Miotto et al utilized a three-layer Stacked Denoising Autoencoder (SDA) and put forth a profound patient representation from the EHRs to forecast risks using random forest classifiers (Abdullah et al., 2022).

1.7.3. Deep Learning in Natural Language Processing

Natural language processing (NLP) involves using computational methods to analyze human language automatically. Research initiatives involving document text and language have been gaining momentum within the signal-processing community. The significant impact of deep learning lies in its application to language modeling, where it assigns probabilities to sequences of linguistic symbols or words. The early days of natural language processing (NLP) were characterized by batch processing and punch cards, with analysis taking up to 7 minutes (Otter et al., 2020). Groundbreaking progress is being made in the realm of pattern recognition thanks to the remarkable advancements in deep learning algorithms along with architectures. According to Collobert, a deep learning framework has proven to be highly effective in various natural language processing tasks, including semantic role labeling (SRL), POS tagging, and named-entity recognition (NER). Over time, numerous iterations of sophisticated

algorithms rooted in deep learning have emerged to tackle a wide range of challenging tasks in natural language processing (Young et al., 2018).

ACTIVITY 1.1.

Objective: To gain hands-on experience in constructing and training a basic feed-forward neural network using a deep learning framework.

Materials Needed:

- A computer with Python installed
- Deep learning framework (e.g., TensorFlow, PyTorch, Keras)
- Sample dataset (e.g., MNIST for image classification)

Instructions:

1. Setup Environment:
   - Install the chosen deep learning framework (e.g., TensorFlow, PyTorch) and required dependencies on your computer.
   - Prepare a development environment with a text editor or an integrated development environment (IDE) like Jupyter Notebook.
2. **Define Neural Network Archi**tecture:
   - Create a simple feed-forward neural network architecture.
   - Define the number of layers, neurons per layer, activation functions, and output layer for the specific task.

SUMMARY

- The chapter covers foundational concepts and applications of deep learning. It starts with an explanation of neurons and their role in artificial neural networks. The historical development of deep learning is discussed, tracing its evolution and milestones. Feed-forward neural networks are introduced as fundamental architectures for processing data. Backpropagation, a key algorithm for training neural networks, is explained in detail.
- The chapter categorizes deep learning networks based on their functionality and structure. It outlines various deep learning architectures and their specific applications. Supervised and unsupervised learning paradigms are contrasted, highlighting their respective uses in training deep learning models. Popular deep learning frameworks such as TensorFlow, Microsoft Cognitive Toolkit (CNTK), Caffe, Keras, Neural Designer, Torch, and their roles in facilitating deep learning development are examined.
- Deep learning applications in speech recognition, healthcare, and natural language processing are explored, emphasizing the impact and advancements enabled by deep learning techniques in these domains.

REVIEW QUESTIONS

1. Define a neuron in the context of deep learning. How does it contribute to artificial neural networks?
2. Discuss the historical development milestones of deep learning. What were some key advancements that paved the way for modern deep learning techniques?
3. Explain the architecture and working principle of feed-forward neural networks. How do they process input data to produce output?
4. What is backpropagation and why is it important in training neural networks? Describe its role in adjusting network weights.
5. Differentiate between supervised and unsupervised learning in the context of deep learning. Provide examples of each and discuss their respective advantages and applications.
6. Compare and contrast the following deep learning frameworks: TensorFlow, PyTorch, Keras, and Caffe. What are their key features, strengths, and typical use cases?
7. Describe the typical structure of a convolutional neural network (CNN). How is it different from other types of deep learning architectures, and what applications benefit most from CNNs?
8. Explore the role of deep learning in speech recognition. What specific techniques and models are commonly used, and what are the challenges in this application domain?

9. Discuss the applications of deep learning in healthcare. How is it used to analyze medical images, predict outcomes, or improve diagnostic accuracy?
10. Examine the role of deep learning in natural language processing (NLP). What are some common tasks where deep learning is applied, and how does it compare to traditional NLP techniques?

MULTIPLE CHOICE QUESTIONS

1. Which algorithm is commonly used for training feed-forward neural networks?

a. Gradient Descent
b. Random Forest
c. K-Means
d. Apriori

2. What is a primary characteristic of supervised learning in deep learning?

a. It does not require labeled data
b. It learns from unlabeled data
c. It requires a predefined set of input-output pairs
d. It is always used for unsupervised tasks

3. Which deep learning framework is known for its ease of use and high-level API design?

a. TensorFlow
b. Keras
c. Caffe
d. Torch

4. Which application area typically benefits from deep learning techniques like convolutional neural networks (CNNs)?

a. Algorithmic trading
b. Social media analysis
c. Speech recognition
d. Statistical analysis

5. What is the main advantage of using unsupervised learning in deep learning?

a. It requires less computational resources
b. It does not need a training phase
c. It can discover patterns without labeled data
d. It is more accurate than supervised learning

6. **Which deep learning framework is developed and maintained by Facebook's AI Research lab (FAIR)?**
   a. Tensor Flow
   b. Keras
   c. Caffe
   d. PyTorch

Answers to Multiple Questions

1. (a); 2. (c); 3. (b); 4. (c); 5. (c); 6. (d).

REFERENCES

1. Abdullah, A. A., Hassan, M. M., & Mustafa, Y. T. (2022). A review on Bayesian deep learning in healthcare: Applications and challenges. IEEE Access, 10, 36538–36562. https://doi.org/10.1109/ACCESS.2022.3163726.
2. Aggarwal, H. K., Mani, M. P., & Jacob, M. (2018). MoDL: Model-based deep learning architecture for inverse problems. IEEE Transactions on Medical Imaging, 38(2), 394–405. https://doi.org/10.1109/TMI.2018.2876633.
3. Ahn, S. (2016). Deep learning architectures and applications. Journal of Intelligence and Information Systems, 22(2), 127–142. https://doi.org/10.13088/jiis.2016.22.2.127.
4. Almási, A. D., Woźniak, S., Cristea, V., Leblebici, Y., & Engbersen, T. (2016). Review of advances in neural networks: Neural design technology stack. Neurocomputing, 174, 31–41. https://doi.org/10.1016/j.neucom.2015.06.088.
5. Almutairi, M., Gabralla, L. A., Abubakar, S., & Chiroma, H. (2022). Detecting elderly behaviors based on deep learning for healthcare: Recent advances, methods, real-world applications, and challenges. IEEE Access, 10, 69802–69821. https://doi.org/10.1109/ACCESS.2022.3177878.
6. Amin, R., Al Ghamdi, M. A., Almotiri, S. H., & Alruily, M. (2021). Healthcare techniques through deep learning: Issues, challenges, and opportunities. IEEE Access, 9, 98523–98541. https://doi.org/10.1109/ACCESS.2021.3095126.
7. Banerjee, I., Ling, Y., Chen, M. C., Hasan, S. A., Langlotz, C. P., Moradzadeh, N., & Lungren, M. P. (2019). Comparative effectiveness of convolutional neural network (CNN) and recurrent neural network (RNN) architectures for radiology text report classification. Artificial Intelligence in Medicine, 97, 79–88. https://doi.org/10.1016/j.artmed.2018.12.002.
8. Boroumand, M., & Fridrich, J. (2018). Deep learning for detecting the processing history of images. Electronic Imaging, 2018(7), 1–9. https://doi.org/10.2352/ISSN.2470-1173.2018.07.MWSF-158.
9. Briot, J. P. (2021). From artificial neural networks to deep learning for music generation: History, concepts, and trends. Neural Computing and Applications, 33(1), 39–65. https://doi.org/10.1007/s00521-020-05111-8.

10. Chicho, B. T., & Sallow, A. B. (2021). A comprehensive survey of deep learning models based on Keras framework. Journal of Soft Computing and Data Mining, 2(2), 49–62. https://doi.org/10.30880/jscdm.2021.02.02.006.

11. Chinta, S. J., & Andersen, J. K. (2005). Dopaminergic neurons. The International Journal of Biochemistry & Cell Biology, 37(5), 942–946. https://doi.org/10.1016/j.biocel.2004.09.009.

12. Choi, R. Y., Coyner, A. S., Kalpathy-Cramer, J., Chiang, M. F., & Campbell, J. P. (2020). Introduction to machine learning, neural networks, and deep learning. Translational Vision Science & Technology, 9(2), 14–14. https://doi.org/10.1167/tvst.9.2.14.

13. de Jong, E. P., Vossen, A. C., Walther, F. J., & Lopriore, E. (2013). How to use... neonatal TORCH testing. Archives of Disease in Childhood–Education and Practice, 98(3), 93–98. https://doi.org/10.1136/archdischild-2012-302540.

14. Deng, L., & Yu, D. (2014). Deep learning: Methods and applications. Foundations and Trends® in Signal Processing, 7(3–4), 197–387. https://doi.org/10.1561/2000000039.

15. Desai, M., & Shah, M. (2021). An anatomization of breast cancer detection and diagnosis employing multilayer perceptron neural network (MLP) and convolutional neural network (CNN). Clinical eHealth, 4, 1–11. https://doi.org/10.1016/j.ceh.2021.07.002.

16. Etaati, L., & Etaati, L. (2019). Deep learning tools with cognitive toolkit (CNTK). In Machine Learning with Microsoft Technologies: Selecting the Right Architecture and Tools for Your Project (pp. 287–302). Apress. https://doi.org/10.1007/978-1-4842-3828-7_13.

17. Fan, F., Shan, H., Kalra, M. K., Singh, R., Qian, G., Getzin, M., & Wang, G. (2019). Quadratic autoencoder (Q-AE) for low-dose CT denoising. IEEE Transactions on Medical Imaging, 39(6), 2035–2050. https://doi.org/10.1109/TMI.2020.2976798.

18. Gaikwad, S. K., Gawali, B. W., & Yannawar, P. (2010). A review on speech recognition technique. International Journal of Computer Applications, 10(3), 16–24. https://doi.org/10.5120/1453-1968.

19. Garea, A. S., Heras, D. B., & Argüello, F. (2019). Caffe CNN-based classification of hyperspectral images on GPU. The Journal of Supercomputing, 75(2), 1065–1077. https://doi.org/10.1007/s11227-018-2225-y.

20. Guo, J., He, H., He, T., Lausen, L., Li, M., Lin, H., & Zhu, Y. (2020). GluonCV and GluonNLP: Deep learning in computer vision and natural language processing. Journal of Machine Learning Research, 21(23), 1–7. https://www.jmlr.org/papers/v21/19-429.html.

21. Ilonen, J., Kamarainen, J. K., & Lampinen, J. (2003). Differential evolution training algorithm for feed-forward neural networks. Neural Processing Letters, 17(1), 93–105. https://doi.org/10.1023/A:1022993527446.

22. Kattenborn, T., Leitloff, J., Schiefer, F., & Hinz, S. (2021). Review on convolutional neural networks (CNN) in vegetation remote sensing. ISPRS Journal of Photogrammetry and Remote Sensing, 173, 24–49. https://doi.org/10.1016/j.isprsjprs.2020.12.010.

23. Kempermann, G. (2012). New neurons for 'survival of the fittest'. Nature Reviews Neuroscience, 13(10), 727–736. https://doi.org/10.1038/nrn3319.

24. Koitmäe, A., Müller, M., Bausch, C. S., Harberts, J., Hansen, W., Loers, G., & Blick, R. H. (2018). Designer neural networks with embedded semiconductor microtube arrays. Langmuir, 34(4), 1528–1534. https://doi.org/10.1021/acs.langmuir.7b03703.

25. Larochelle, H., Mandel, M., Pascanu, R., & Bengio, Y. (2012). Learning algorithms for the classification restricted Boltzmann machine. Journal of Machine Learning Research, 13(1), 643–669. https://jmlr.org/papers/v13/larochelle12a.html (accessed on 5 September 2024).

26. Le Glaz, A., Haralambous, Y., Kim-Dufor, D. H., Lenca, P., Billot, R., Ryan, T. C., & Lemey, C. (2021). Machine learning and natural language processing in mental health: Systematic review. Journal of Medical Internet Research, 23(5), e15708. https://doi.org/10.2196/15708.

27. LeCun, Y., Touresky, D., Hinton, G., & Sejnowski, T. (1988). A theoretical framework for back-propagation. In Proceedings of the 1988 Connectionist Models Summer School (Vol. 1, pp. 21–28). Morgan Kaufmann. https://doi.org/10.1016/B978-1-4832-1446-7.50005-3.

28. Lee, H., & Song, J. (2019). Introduction to convolutional neural network using Keras; An understanding from a statistician. Communications for Statistical Applications and Methods, 26(6), 591–610. https://doi.org/10.29220/CSAM.2019.26.6.591.

29. Leung, H., & Haykin, S. (1991). The complex backpropagation algorithm. IEEE Transactions on Signal Processing, 39(9), 2101–2104. https://doi.org/10.1109/78.83961.

30. Li, S., Song, W., Fang, L., Chen, Y., Ghamisi, P., & Benediktsson, J. A. (2019). Deep learning for hyperspectral image classification: An overview. IEEE Transactions on Geoscience and Remote Sensing, 57(9), 6690–6709. https://doi.org/10.1109/TGRS.2019.2907262.

31. Li, X., Peng, L., Hu, Y., Shao, J., & Chi, T. (2016). Deep learning architecture for air quality predictions. Environmental Science and Pollution Research, 23, 22408–22417. https://doi.org/10.1007/s11356-016-7812-0.

32. Lillicrap, T. P., Santoro, A., Marris, L., Akerman, C. J., & Hinton, G. (2020). Backpropagation and the brain. Nature Reviews Neuroscience, 21(6), 335–346. https://doi.org/10.1038/s41583-020-0277-3.

33. Maier, A., Syben, C., Lasser, T., & Riess, C. (2019). A gentle introduction to deep learning in medical image processing. Zeitschrift für Medizinische Physik, 29(2), 86–101. https://doi.org/10.1016/j.zemedi.2018.12.003.

34. Malik, M., Singh, Y., Garg, P., & Gupta, S. (2020). Deep learning in healthcare system. International Journal of Grid and Distributed Computing, 13(2), 469–468. https://doi.org/10.21742/ijgdc.2020.13.2.39.

35. Mandal, M., & Vipparthi, S. K. (2021). An empirical review of deep learning frameworks for change detection: Model design, experimental frameworks, challenges and research needs. IEEE Transactions on Intelligent Transportation Systems, 23(7), 6101–6122. https://doi.org/10.1109/TITS.2021.3092001.

36. Mathew, A., Amudha, P., & Sivakumari, S. (2021). Deep learning techniques: An overview. In Advanced Machine Learning Technologies and Applications: Proceedings of AMLTA 2020 (pp. 599–608). Springer. https://doi.org/10.1007/978-3-030-63128-4_60.

37. Mia, M. M. A., Biswas, S. K., Urmi, M. C., & Siddique, A. (2015). An algorithm for training multilayer perceptron (MLP) for image reconstruction using neural network without overfitting. International Journal of Scientific & Technology Research, 4(02), 271–275. https://www.ijstr.org/research-paper-publishing.php?month=feb2015 (accessed on 5 September 2024).

38. Min, S., Lee, B., & Yoon, S. (2017). Deep learning in bioinformatics. Briefings in Bioinformatics, 18(5), 851–869. https://doi.org/10.1093/bib/bbw068.

39. Miotto, R., Wang, F., Wang, S., Jiang, X., & Dudley, J. T. (2018). Deep learning for healthcare: Review, opportunities and challenges. Briefings in Bioinformatics, 19(6), 1236–1246. https://doi.org/10.1093/bib/bbx044.

40. Nguyen, G., Dlugolinsky, S., Bobák, M., Tran, V., López García, Á., Heredia, I., & Hluchý, L. (2019). Machine learning and deep learning frameworks and libraries for large-scale data mining: A survey. Artificial Intelligence Review, 52, 77–124. https://doi.org/10.1007/s10462-018-09679-z.

41. Otter, D. W., Medina, J. R., & Kalita, J. K. (2020). A survey of the usages of deep learning for natural language processing. IEEE Transactions on Neural Networks and Learning Systems, 32(2), 604–624. https://doi.org/10.1109/TNNLS.2020.2979670.

42. Prince, S. (2024). Understanding deep learning. MIT Press, 1(1), 1–16.

43. Rajendran, R. M. (2023). Code-driven cognitive enhancement: Customization and extension of Azure Cognitive Services in .NET. Journal of Science & Technology, 4, 45–54. https://doi.org/10.1234/jst.2023.4.45.

44. Schmidhuber, J. (2015). Deep learning. Scholarpedia, 10(11), 32832. https://doi.org/10.4249/scholarpedia.32832.

45. Shawahna, A., Sait, S. M., & El-Maleh, A. (2018). FPGA-based accelerators of deep learning networks for learning and classification: A review. IEEE Access, 7, 7823–7859. https://doi.org/10.1109/ACCESS.2018.2883480.

46. Smith, T. W., & Colby, S. A. (2007). Teaching for deep learning. The Clearing House: A Journal of Educational Strategies, Issues and Ideas, 80(5), 205–210. https://doi.org/10.3200/TCHS.80.5.205-210.

47. Soak, M., Sahay, S. K., & Rathore, H. (2020). An overview of deep learning architecture of deep neural networks and autoencoders. Journal of Computational and Theoretical Nanoscience, 17(1), 182–188. https://doi.org/10.1166/jctn.2020.8700.

48. Sorin, V., Barash, Y., Konen, E., & Klang, E. (2020). Deep learning for natural language processing in radiology—Fundamentals and a systematic review. Journal of the American College of Radiology, 17(5), 639–648. https://doi.org/10.1016/j.jacr.2020.01.024.

49. Svozil, D., Kvasnicka, V., & Pospichal, J. (1997). Introduction to multi-layer feed-forward neural networks. Chemometrics and Intelligent Laboratory Systems, 39(1), 43–62. https://doi.org/10.1016/S0169-7439(97)00061-8.

50. Tarwani, K. M., & Edem, S. (2017). Survey on recurrent neural network in natural language processing. International Journal of Engineering Trends and Technology, 48(6), 301–304. https://doi.org/10.14445/22315381/IJETT-V48P254.

51. Vincent, S. R., & Hope, B. T. (1992). Neurons that say NO. Trends in Neurosciences, 15(3), 108–113. https://doi.org/10.1016/0166-2236(92)90331-B.

52. Wang, D., Su, J., & Yu, H. (2020). Feature extraction and analysis of natural language processing for deep learning English language. IEEE Access, 8, 46335–46345. https://doi.org/10.1109/ACCESS.2020.2978571.

53. Werbos, P. J. (1990). Backpropagation through time: What it does and how to do it. Proceedings of the IEEE, 78(10), 1550–1560. https://doi.org/10.1109/5.58337.

54. Wu, J. (2017). Introduction to convolutional neural networks. National Key Lab for Novel Software Technology, Nanjing University, 5(23), 495. https://doi.org/10.1007/978-1-4419-1428-6_14.

55. Xu, W., Jang-Jaccard, J., Singh, A., Wei, Y., & Sabrina, F. (2021). Improving performance of autoencoder-based network anomaly detection on NSL-KDD dataset. IEEE Access, 9, 140136–140146. https://doi.org/10.1109/ACCESS.2021.3114712.

56. Young, T., Hazarika, D., Poria, S., & Cambria, E. (2018). Recent trends in deep learning based natural language processing. IEEE Computational Intelligence Magazine, 13(3), 55–75. https://doi.org/10.1109/MCI.2018.2840738.

57. Zhang, N., Ding, S., Zhang, J., & Xue, Y. (2018). An overview on restricted Boltzmann machines. Neurocomputing, 275, 1186–1199. https://doi.org/10.1016/j.neucom.2017.09.043.

58. Zhao, Z. Q., Zheng, P., Xu, S. T., & Wu, X. (2019). Object detection with deep learning: A review. IEEE Transactions on Neural Networks and Learning Systems, 30(11), 3212–3232. https://doi.org/10.1109/TNNLS.2018.2876865.

59. Zou, C., Yang, F., Song, J., & Han, Z. (2021). Channel autoencoder for wireless communication: State of the art, challenges, and trends. IEEE Communications Magazine, 59(5), 136–142. https://doi.org/10.1109/MCOM.001.2000664.

CHAPTER

2 Deep Neural Network Models

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

- Understand the historical development and limitations of early artificial neural networks.
- Know the architecture and training methods of Multilayer Perceptrons (MLPs).
- Explore advanced deep learning techniques including Boltzmann Machines and Deep Belief Networks (DBNs).
- Understand the architecture and components of Convolutional Neural Networks (CNNs).
- Implement deep attention selective networks (dasNet) for enhancing CNN performance.
- Comprehend the structure and training process of Recurrent Neural Networks (RNNs).

KEY TERMS FROM THIS CHAPTER

Backpropagation
Boltzmann machines
Convolutional neural networks (CNNs)
Deep auto-encoders
Deep belief networks
Long short-term memory (LSTM)
Perceptron
Recurrent neural networks (RNNs)

UNIT INTRODUCTION

The idea of deep learning emerged from the study of artificial neural networks. In this context, neural networks with numerous hidden layers are widely recognized as deep neural networks (DNNs).

MLP DL networks can be now implemented with the backpropagation algorithm, gradient descent. The underlying concept of Backpropagation (BP) is quite easy: The process entails evaluating the output signal of the neural network's output layer against the actual output via the data for each input/output pair, enabling the detection of any errors. Given the ability to calculate the signal flow within the network, it becomes possible to adjust the weights between neurons in the various layers. This adjustment aims to minimize the error in subsequent iterations. To accomplish this, the weights are adjusted based on the magnitude of the error (Srinidhi et al., 2021).

When training deep networks, using only BP can present several challenges. Certain challenges arise in the field of deep neural network models. An obstacle that arises is the existence of local optima traps within the objective associated with the nonconvex function. Another obstacle is the occurrence of vanishing gradients, where the output signal diminishes fast as data is carried backward through the layers. To grasp the solution to this problem, it is necessary to delve into the historical background of artificial neuron networks (ANNs) (Mehrer et al., 2020).

Within this unit, the fundamental principles and sophisticated structures are explained that form the basis of contemporary machine learning. Starting with the historical progression from early neural networks to the advanced deep neural networks of the present, one will discuss important models including Recurrent Neural Networks (RNNs), Multilayer Perceptrons (MLPs), Convolutional Neural Networks (CNNs), and Generative Models such as Generative Adversarial Networks (GANs) and Variational Auto-encoders (VAEs) (Kriegeskorte & Golan, 2019).

2.1. A CONCISE OVERVIEW OF NEURAL NETWORK DEVELOPMENT

Artificial Neural Networks originated from the groundbreaking research conducted by McCullogh and Pitts. Their work demonstrated basic units, known as artificial neurons, possessed the remarkable ability to execute any logical operation, making them capable of achieving universal computation. This research was conducted alongside the pioneering work of Von Neumann and Turing and was among the first to explore the statistical aspects of brain information processing and the development of a machine able to replicate it. Frank Rosembalt pioneered the development of the perceptron machine, which revolutionized the field of pattern classification. However, this innovative learning machine demonstrated its limitations when attempting to solve fundamental problems, such as the logic XOR. Minsky and Papert (1969) presented findings that highlighted the inherent limitations of perceptrons. As a result, enthusiasm for artificial neural networks began to wane (Sze et al., 2017).

In 1983, John Hopfield introduced a unique class of artificial neural networks known as the Hopfield networks and demonstrated their remarkable capabilities in pattern completion and memory retention. In 1970, Linnainmaa introduced the backpropagation algorithm, which discusses the cumulative rounding error in an algorithm. It is worth noting that this algorithm was described without any mention of neural networks. In the year 1985, Rumelhart, McClelland, and Hinton made a significant breakthrough by rediscovering a potent learning rule. This rule enabled them to train artificial neural networks with multiple hidden units, effectively addressing the criticism raised by Minsk (Samek et al., 2021).

2.1.1. The Multilayer Perceptron

An approach was suggested to tackle issues that might not be readily distinguished linearly. Simply put, categorization cannot be achieved by drawing straight lines. Shown in Figure 2.1 is an example of a multifaceted perceptron (Popescu et al., 2009). A neural network consists of input units which are linked to hidden units with weights, w. The hidden units connect to the output through weights, v. Initially, random values are assigned to each weight and bias term (Ruck et al., 1990). The transmission of information within the network occurs through the weights connecting the input layer to its hidden layer, in which a function is computed to describe the net activation. Activation functions commonly employed in deep neural network models include sigmoid, tanh, and the current (ReLU) rectified linear unities. Next, the process is continued by utilizing more weight for the output neurons (Tang et al., 2015).

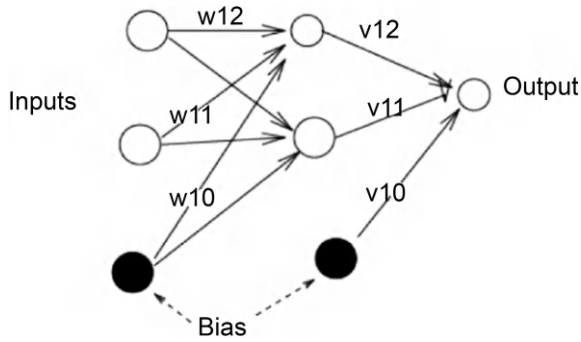


Figure 2.1. Illustration of MLP includes input nodes, hidden nodes, and output nodes. Training involves identifying the optimal weights, w or v, as well as the bias.

Source: Armando Vieira, Creative Commons License.

Did you know?

Neural networks have a rich history that can be traced back to the 1940s when the intricate workings of the human brain influenced early models. Nevertheless, the practical implementation of these models was constrained until a significant increase in computational power and the availability of large datasets in the past few decades.

Basic Perceptron example

Given the neural network of the figure determine the output.

2.1.1.1. Training a DL network

Updating the weights involves two distinct sets: the weights connecting the hidden and output layers, along with the weights connecting the input and hidden layers. The inaccuracy resulting from the initial set of weights can be determined using the minimum mean square principle. The algorithm for backpropagation is commonly employed to propagate the error caused by the errors within the second set of weights (W) in a backward manner. This indicates that the errors should be in proportion to the weight contribution. The algorithm has two primary parameters: the learning rate and momentum, which are used to prevent getting stuck in local minima. Additionally, the number of units in the hidden layer is a crucial input. Increasing the number of hidden units can enhance computational power, however, it may also impact the model's generalization abilities (Gardner & Dorling, 1998).

The selection of network parameters is typically done through k-fold cross-validation. This involves dividing the training data into k segments, using k–1 segments for training and the extra segment for testing. These segments are then swapped to ensure a comprehensive evaluation.

The training process of a neural network can be expedited by employing the stochastic gradient descent (SGD) algorithm. However, gradient descent requires all training samples for optimization, SGD just utilizes a portion of the training sample. SGD accelerates

convergence by utilizing a subset of training samples for every epoch (Attali & Pagès, 1997).

In 2006, Hinton presented an algorithm for autonomous learning that utilizes a method known as contrastive divergence (CD). An effective method for deep generative training models is through the use of deep belief networks (DBNs). The CD method is an algorithm that follows a sequential learning approach" that operates layer upon layer, as depicted in Figure 2.2. Usually employed for unsupervised tasks, the model may be modified for learning under supervision by incorporating a softmax layer as the final layer (Zhang et al., 2021).

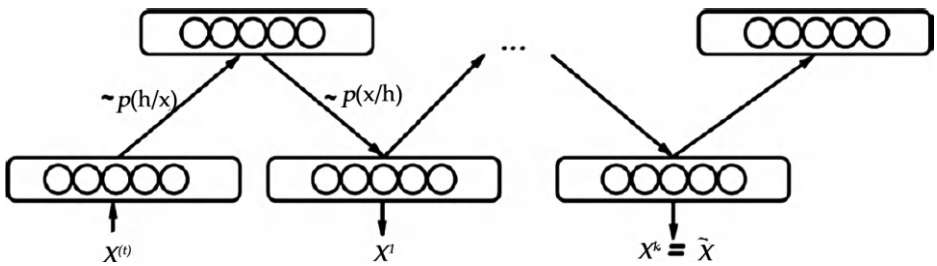


Figure 2.2. Representation of (CD) simulating contrastive divergence involves employing an MCMC process with k steps. CD–1 stops at stage 1 and ignores any further iterations if the input x is successfully reconstructed as x^1.

Source: Bernardete Ribeiro, Creative Commons License.

2.2. UNDERSTANDING DEEP NEURAL NETWORKS

It has been widely recognized that artificial neural networks with increased hidden layers (thus the concept of Deep neural network) possess greater capabilities at addressing classification or regression tasks. Once the weights are computed and train these models, specifically, to acquire knowledge of the weights or connections that establish links between different layers of neurons. The backpropagation algorithm has proven to be successful for artificial neural networks with a single hidden layer. Nevertheless, deeper architectures pose challenges for this approach, mainly due to the issue of vanishing gradients. Put simply, the corrective signal becomes weaker as it progresses towards the lower layers (Montavon et al., 2018).

There exists a diverse array of DL approaches and architectures to choose from, however, the majority of DNNs can be classified into five primary groups which includes:

- Models for unsupervised learning, are aimed at capturing complex relationships within data by simultaneously modeling statistical distributions alongside their corresponding classes, if provided. In subsequent stages, the application of the Bayes rule can be employed to construct a learning machine focusing on discrimination (Fong et al., 2019).
- Supervised learning models are designed to maximize their ability to accurately classify data by utilizing labeled examples during training. Assigning tags to each of the outputs is crucial.
- Hybrid or semi-supervised networks focus on classifying data by leveraging the outputs of a generative model that operates in an unsupervised manner. Using information to pretrain the network weights is a widely adopted practice. This accelerates the learning process before the stage of supervision.
- As depicted in Figure 2.2, understanding the structure of data without labels x, or the distribution P(x) in statistical terms, can prove to be more effective than relying solely on labeled data with supervised learning (Bau et al., 2020)., Reinforcement learning entails the agent engaging with surroundings and obtaining feedback upon completion of a series of actions. This type of learning is commonly used in the fields of robotics and gaming.
- Deep generative models tend to be highly effective in unsupervised and semi-supervised learning. Their purpose is to uncover the underlying patterns in data without the need for labels. These models offer an effective means of understanding the hidden structure in datasets (Saleem et al., 2022). Due to their generative nature, these models can create a vivid representation of the environment to which they are applied. Imagination can be utilized to delve into different aspects of data, analyze the structure and dynamics of the world, and ultimately, facilitate decision-making. One notable benefit of such models is that can learn the structure of the data on their own, eliminating the need for an external loss function.

Despite the current buzz surrounding deep learning, conventional approaches continue to hold significance in addressing machine learning problems, particularly in cases where the data volume is limited and input characteristics are relatively uncomplicated. Moreover, when confronted with a substantial number of variables about the number of training examples, alternative approaches Alternative machine learning methods also known as traditional methods, like support vector machines (SVMs) or ensemble methods like random forest and extreme gradient boost trees (the XGBoost), can provide more straightforward, efficient, and advanced choices (Samek et al., 2016). Some commonly utilized architectures in the field of deep neural network models are stacked denoising auto-encoders (SdAE), convolutional neural networks (CNNs), deep belief networks, and (RNNs) recurrent neural networks. Significant progress in machine vision has been made through the use of CNNs, establishing this type of DNN as the benchmark for image processing. Nevertheless, a wide range of DNN variations can be employed in diverse business contexts, contingent upon factors such as initialization, architecture, connectivity, training method, and loss functions (Golovko et al., 2016).

Figure 2.3 provides a concise overview of these widely used DNN architectures. Here are some helpful guidelines for gaining an in-depth grasp of the terms and prevalent kinds of deep neural networks.

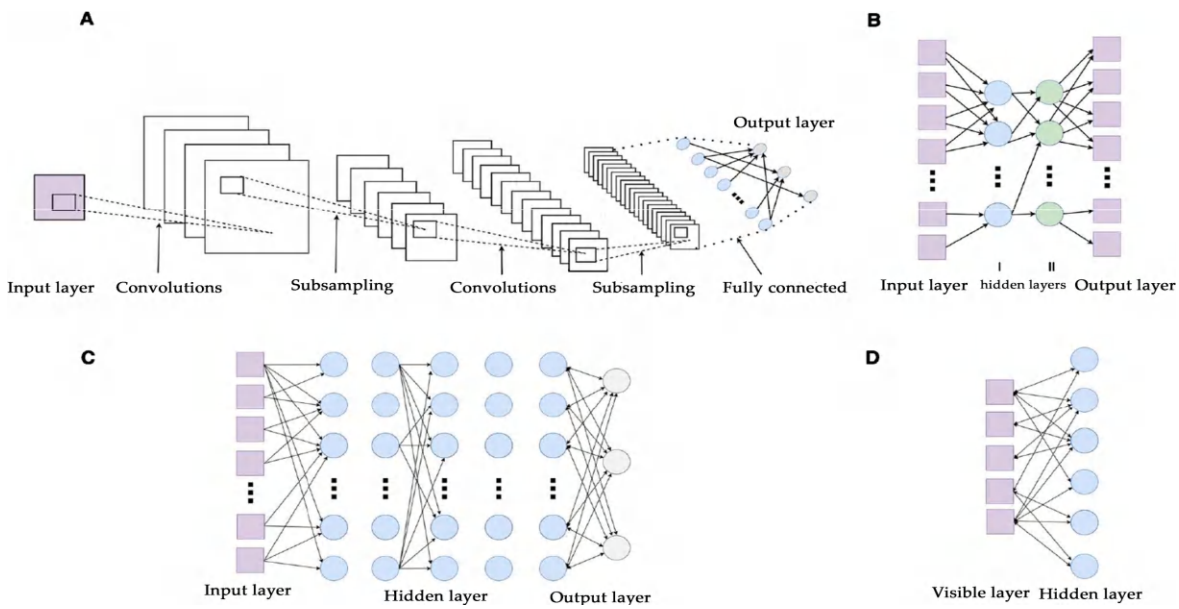


Figure 2.3. Illustration of Four widely used classes for deep learning architectures at data analysis. (A). A Convolutional neural network (CNN) consists of multiple convolutional and subsampling layers, which can be further enhanced with fully connected layers to create a deep architecture. (B). the stacked auto-encoder is composed of numerous sparse auto-encoders. (C). One way to train a DBN is by sequentially freezing the weights of each layer and passing the output to the next layer. (D). The RBM architecture consists of a visible layer along with an additional layer of hidden units.

Source: Bernardete Ribeiro, Creative Commons License.

2.3. BOLTZMANN MACHINES

The Boltzmann machine represents a probabilistic variation of the Hopfield network, incorporating hidden units. It is named after the Boltzmann distribution (Ackley et al., 1985).

The energy function of a Boltzmann device can be defined similarly as in the Hopfield network, with the sole distinction being the assignment of separate labels for hidden units, h, and visible units, v.

$$E(v,h) = -\sum_i v_i b_i - \sum_k h_k b_k - \sum_{i,j} v_i v_j w_{ij} - \sum_{i,k} v_i h_k w_{ik} - \sum_{k,l} h_k h_l w_{k,l} \quad (2.1)$$

In this context, the notation used includes v for visible units, h for hidden units, b for bias, and w_{ij} for the weights connecting units i and j.

Based on this energy function, the likelihood of a combined configuration involving both the visible unit and the hidden unit can be determined as follows:

$$p(v,h) = \frac{e^{-E(v,h)}}{\sum_{m,n} e^{-E(m,n)}} \quad (2.2)$$

The determination of the probability for visible/hidden units occurs through the process of marginalization of the joint probability. For instance, when considering the hidden units as a whole, it is possible to derive the probability distribution of the visible units (Zhang et al., 2018).

$$p(v) = \frac{\sum_h e^{-E(v,h)}}{\sum_{m,n} e^{-E(m,n)}} \quad (2.3)$$

Now, this technique can be employed for testing visible units.

After the training of a Boltzmann machine is finished and it reaches thermal equilibrium, the probability distribution, p (v, h), keeps constant just like its energy distribution also becomes stable. Nevertheless, the likelihood of every visible and hidden unit can fluctuate, and its energy might not be at its lowest level (Fischer & Igel, 2014).

The training procedure of a Boltzmann machine entails optimizing the parameters to increase the likelihood of the data that is observed. The conventional goal function involves utilizing gradient descent on the logarithm for the likelihood function (Upadhya & Sastry, 2019).

The algorithm operates as detailed. Initially, a log-likelihood function of visible units is calculated.

$$l(v;w) = \log p(v;w) = \log \sum_{h} e^{-E_{v,h}} - \log \sum_{m,n} e^{-E_{m,n}} \tag{2.4}$$

Now, the next step is to calculate the derivative of the log-likelihood function for w and then simplify the expression.

$$= -\mathbb{E}_{p(h|v)} \frac{\partial E(v,h)}{\partial w} + \mathbb{E}_{p(m,n)} \frac{\partial E(m,n)}{\partial w} \tag{2.5}$$

$$\frac{\partial l(v;w)}{\partial w} = -\sum_{h} p(h|v) \frac{\partial E(v,h)}{\partial w} + \sum_{m,n} p(m,n) \frac{\partial E(m,n)}{\partial w} \tag{2.6}$$

In this context, the symbol 'd' represents the concept of expectation. Gradient consists of two components. The first part of the analysis focuses on evaluating the slope of an energy function concerning the conditionally concerning distribution p (h |v). The next step involves calculating the anticipated gradient of the energy functions for the general distribution over all states.

Calculating these expectations is typically a challenging task since it requires summing over a vast array of potential states or configurations. One common method for addressing this issue involves utilizing Markov chain Monte Carlo (MCMC) to estimate these values (Larochelle & Bengio, 2008).

$$\frac{\partial l(v;w)}{\partial w} = - < s_i, s_j >_{p(h_{data}|v_{data})} + < s_i, s_j >_{p(h_{model}|v_{model})} \tag{2.7}$$

Here, <× > represents expectation.

Equation (2.7) represents the contrast between the expected value of state products when data is input into visible states along the expected value of state products when no data is input. The initial term is computed by averaging the gradient of the energy function while the visible or hidden units are influenced by observed data samples.

Calculating the first term is straightforward, while the second one poses a greater challenge. It requires running a series of Markov chains across all potential states until get converge to the equilibrium distribution of the current model. Only then one can compute the average gradient of the energy function. The invention of the limited Boltzmann machine was a result to improve upon the limitations of traditional Boltzmann Machines and enhance the capabilities of neural networks in learning complex data distributions. (Hjelm et al., 2014).

2.3.1. Restricted Boltzmann Machines

Hinton and Sejnowski are credited with inventing the Boltzmann machine (RBM). The Restricted Boltzman Machine is a specific type of Boltzmann Machine with a restricted

architecture that simplifies the learning process. Boltzmann Machines have connections between all pairs of nodes, including within the same layer. Restricted Boltzmann Machines (RBMs) does not have connections between nodes within the same layer. RBM's can be trained more efficiently.

Figure 2.4 depicts the application of the constrained Boltzmann machine, a variant of the Boltzmann machine. The link between visible units and hidden units is eliminated, resulting in a bipartite graph. By imposing this limitation, the energy function in the RBM becomes significantly more straightforward (Jin et al., 2019).

$$E(v,h) = -\sum_i v_i b_i - \sum_k h_k b_k - \sum_{i,k} v_i h_k w_{ik} \tag{2.8}$$

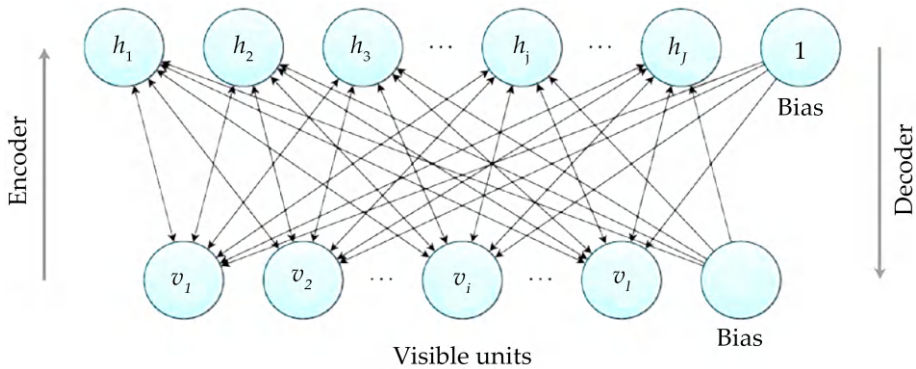


Figure 2.4. Illustration of restricted Boltzmann machine. With the restriction that there are no connections between hidden units (h_j = 1, …, J nodes) and no connections between visible units (v_i = 1, …, I nodes), the Boltzmann machine turns into a restricted Boltzmann machine. The model now is a bipartite graph.

Source: Bryan Catanzaro, Creative Commons License.

2.3.2. Contrastive Divergence

The RBM can be trained using an identical methodology to that of a Boltzmann machine. Due to the straightforwardness of the energy function used by the Restricted Boltzmann Machine (RBM), the method of sampling for inferring the next term of Eq. (2.7) becomes simpler. Although this learning technique may seem simple, it nevertheless requires a substantial number of sample steps to accurately estimate the distribution of the model (Hinton, 2002).

To highlight the challenges associated with this sampling mechanism and to streamline the subsequent introduction, it is possible to rephrase Eq. (2.7) using alternative notations:

$$\frac{\partial l(v;w)}{\partial w} = -\langle s_i, s_j \rangle_{p_0} + \langle s_i, s_j \rangle_{p_\infty} \tag{2.8}$$

In this context, p_0 is used to represent the data distribution, while p_∞ is used to represent the model distribution. The remaining notations remain the same. Thus, the challenge with the mentioned methods lies in the need for a potentially endless number of sampling steps to calculate the model distribution (Bengio & Delalleau, 2009).

Hinton successfully addressed this problem by introducing a technique called contrastive divergence. Based on empirical evidence, it has been discovered that achieving convergence to a model distribution does not require an infinite number of sampling steps. A finite number of k steps of sampling seems sufficient. Thus, Equation 2.9 can be reformulated as follows:

$$\frac{\partial l(v;w)}{\partial w} = -<s_i, s_j>_{p_0} + <s_i, s_j>_{p_k} \tag{2.9}$$

It has been demonstrated by Hinton that utilizing a value of k = 1 is adequate for the algorithm used for learning to reach convergence. This algorithm is commonly referred to as CD1 (Yu et al., 2021).

2.3.3. Deep Belief Nets

DBN is a type of deep neural network that is composed of multiple layers of RBMs, stacked on top of each other. A groundbreaking discovery was made, demonstrating the ability to stack and train (RBMs) in a layer-wise manner. Figure 2.5 depicts the configuration for a three-layer profound belief network. Unlike the Restricted Boltzmann Machine (RBM), the Deep Belief Network (DBN) allows for bidirectional connections only on the top layer, enabling information flow from both the top-down and bottom-up. The lowest tiers that follow are composed exclusively of directional connections. A Deep Belief Network (DBN) may be understood as a complex generative model, in which each neuron functions as a probabilistic unit (Hinton et al., 2006).

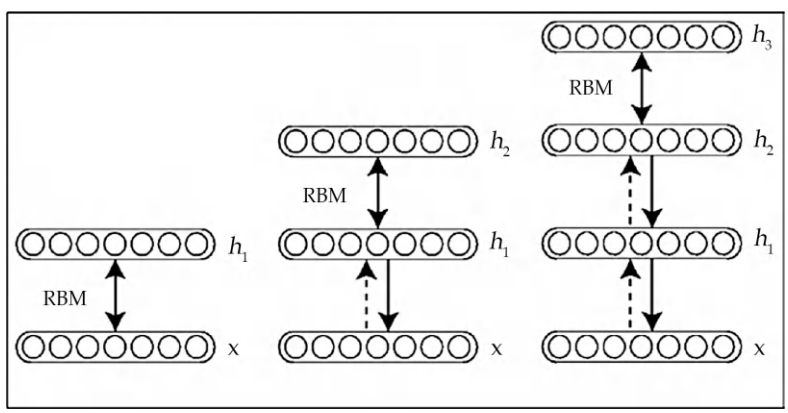


Figure 2.5. Illustration of deep belief networks. Except the top layer, the lower levels do not have bidirectional connections. They only have connections that go across the top to the bottom.

Source: Jesse Engel, Creative Commons License.

Thus, the model only requires a sample to reach thermal equilibrium in the upper layer before transferring the data to the visible states (Salakhutdinov & Murray, 2008). DBNs are trained by a two-step procedure: layer-wise pre-training followed by parameter fine-tuning. The process of layer-wise pre-training involves training individual layers sequentially. Once the initial layer has been trained, the connections are then locked in place, allowing for the addition of a new layer on top. The next layer is trained using the same methodology as the initial layer, and this process is repeated for multiple layers as necessary. This pre-training may be viewed as a highly effective method for initializing the weights (Mohamed et al., 2011).

Refinement is conducted to further improve the network through the utilization of one of two distinct refinement strategies which includes:

- Improving a Generative Model: Refining a generative model involves utilizing a contrastive variant of the wake-sleep algorithm, a method that draws inspiration from neuroscience. During the wake phase, there is a flow of information from the lower layer to the upper layer, which helps in adjusting the weights in a downward-upward manner. This adjustment is crucial for creating a representation on the upper layer. During the sleep phase, there is a reversal of the process where the information is transmitted downwards to modify the connections from top to bottom (Le Roux & Bengio, 2010).
- Fine-Tuning For a Discriminative Model: In this scenario, the process of fine-tuning a DBN is to apply standard backpropagation on a pre-trained network, utilizing the labels for the data upon the higher layer.

In addition to offering a solid network initialization, the DBN possesses several other noteworthy properties. Firstly, all types of data can be utilized, including unlabeled data sets. Furthermore, it can be regarded to be a probabilistic generative method that proves to be valuable in the Bayesian framework. Furthermore, the issue of over-fitting can be successfully mitigated through the utilization of pre-training and the implementation of robust regularizes, such as dropout.

Nevertheless, a DBN encounters the subsequent challenges:

- DBNs pose a challenge when it comes to inference due to the presence of the "explaining away" effect.
- Traditional DBN models are limited to greedy retraining, lack the capability for joint optimization across all layers.
- Approximate inference operates in a feed-forward manner, with no exchange of information between lower and higher levels (Movahedi et al., 2017).

2.3.4. Deep Boltzmann Machines

Salakhutdinov introduced the deep Boltzmann machine. Figure 2.6 illustrates a three-layer Boltzmann machine with multiple layers. In the previous section, it was mentioned that there is a difference between DBM and DBN. This distinction lies in the way data circulates on bidirectional connections within the bottom layers of DBM (Taherkhani et al., 2018).

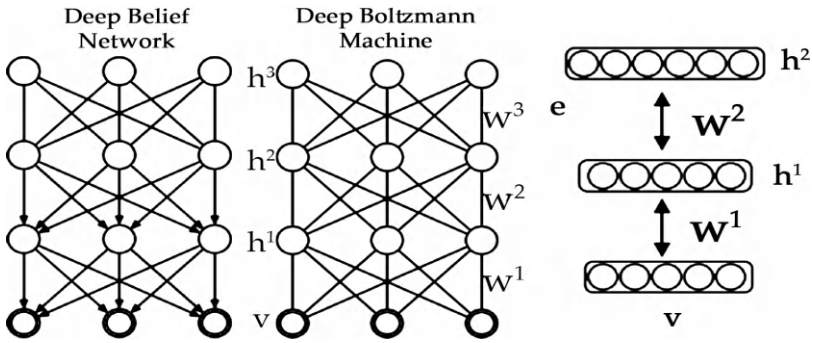


Figure 2.6. Illustration deep Boltzmann machine (DBM). DBM has a composite structure consisting of many restricted Boltzmann machines.

Source: David H. Ackley, Creative Commons License.

The power function of a (DBM) with numerous hidden layers can be characterized as an extension of the energy function of a (RBM). Equation (2.10) demonstrates this concept for a (DBM) with N hidden layers:

$$E(v,h) = -\sum_i v_i b_i - \sum_{n=1}^{N}\sum_k h_{n,k} b_{n,k} - \sum_{i,k} v_i w_{ik} h_k - \sum_{n=1}^{N-1}\sum_{k,1} h_{n,k} w_{n,k,l} h_{n+1,l} \quad (2.10)$$

Due to similarities of energy functions, training a DBM utilizing contrastive divergence (CD1) is also possible. There are certain similarities between DBN and DBM as Both of these neural networks are heavily influenced by the restricted Boltzmann machine. The bidirectional structure of a DBM allows for the detection and analysis of intricate patterns within datasets (Passos & Papa, 2020).

2.4. CONVOLUTIONAL NEURAL NETWORKS

The CNN consists of multiple blocks that contain stacked layers of various types. Every block is comprised of convolutional and pooling layers, typically utilizing max pooling. These modules are typically organized vertically, either layered on top of one another or with the addition of a softmax logistic layer to form a deep model. CNN utilizes a range of techniques that render them exceptionally proficient in the realm of image processing (Li et al., 2021).

These techniques involve the use of pooling, weight sharing, or adaptive filters. Subsampling is used in the layer of convolution to select representative samples for the next layer, which helps to improve the overall performance of the model. Weight sharing or pooling schemes, often utilizing max pooling, enable CNNs to achieve desirable conservation properties such as translation invariance. CNNs have shown impressive effectiveness and are widely used in the field of machine vision and image recognition (Gu et al., 2018). Convolutional Neural Networks (CNNs) operate on a sequence of signals rather than a single feature vector. In completely linked neural networks, each activation unit is coupled to every input of the characteristics vector. Weights are assigned to each unit based on the features in the input. Convolutional layers, in contrast to other layer types, employ weight share by sliding a small filter of weights through the input vector and 2D input map, commonly used for images. This procedure applies convolution to each overlapping region in the input map using the filter (Yamashita et al., 2018).

Convolutional neural networks (CNNs) that include max pooling have the potential to accurately replicate the early stages of a primate's visual cortex. These networks can produce feature detectors that align with biological mechanisms, like Gabor filters. Nevertheless, after being trained, the CNN functions as a straightforward feed-forward mechanism with unchanging weights. In a recent study, Stollenga introduced a novel approach to CNNs known as deep attention selective networks (dasNet). This iteration of CNNs incorporates post-processing behavior, resulting in improved performance (Lavin & Gray, 2016). This architecture showcases the capacity to incorporate selective attention in CNNs by permitting every layer to influence all other layers through multiple iterations into an image. This is accomplished through distinct connections, coming from the lower levels and higher levels, which govern the behavior within the convolutional filters. The weights of these distinct connections represent a control policy that is acquired by reinforcement learning after the CNN has completed standard supervised learning training. The attentional policy may alter the properties of an input image across several iterations to enhance the classification of challenging cases that were not effectively addressed during the initial supervised training. The dasNet architecture enables automated inspection of internal CNN filters, eliminating the need for manual verification (Bilal et al., 2017).

Remember

Convolutional Neural Networks (CNNs) were created for image processing tasks, have demonstrated their adaptability by effectively addressing other fields like natural language processing and speech recognition.

2.5. DEEP AUTO-ENCODERS

Auto-encoder describes a type of deep neural network that produces the input data as its output. When these structures are trained with extra noise, they can serve as generative models and are commonly known as denoising auto-encoders. A possible training method for an auto-encoder involves a step-by-step approach, resembling the technique used for Deep Belief Networks (DBNs), to build a complex model (Karim et al., 2020).

Stacking auto-encoders allows for the creation of a network with multiple layers, resulting in a more complex and powerful model. This is achieved by extracting the results of the auto-encoder from one layer and using them as input for the layer above. During the unsupervised pre-training process, each layer is trained individually to minimize mistakes when reconstructing its input. After the pre-training phase, the network can be fine-tuned by adding a softmax layer and applying supervised backpropagation. This process is similar to what is done for multilayer perceptrons (Basati & Faghih, 2022).

A stacked denoising auto-encoder (SdAE) is a variant of an auto-encoder (AE) that adds randomness by injecting noise to the input. This aids in preventing the model from merely duplicating the input. When attempting to encode the input and reverse the impact of corruption, it is important to capture the statistical dependency in the inputs (Tong et al., 2018).

2.6. RECURRENT NEURAL NETWORKS

Conventional machine learning techniques, such as support vector machines, feed-forward networks, and logistic regression, are known for their utility in capturing temporal processes without explicitly incorporating time as a spatial concept. Unfortunately, this assumption fails to capture long-range dependencies and is of limited use in intricate temporal patterns (Schuster & Paliwal, 1997). Recurrent neural networks are now a diverse set of models that can be differentiated from end to end, making them suitable for gradient-based training.

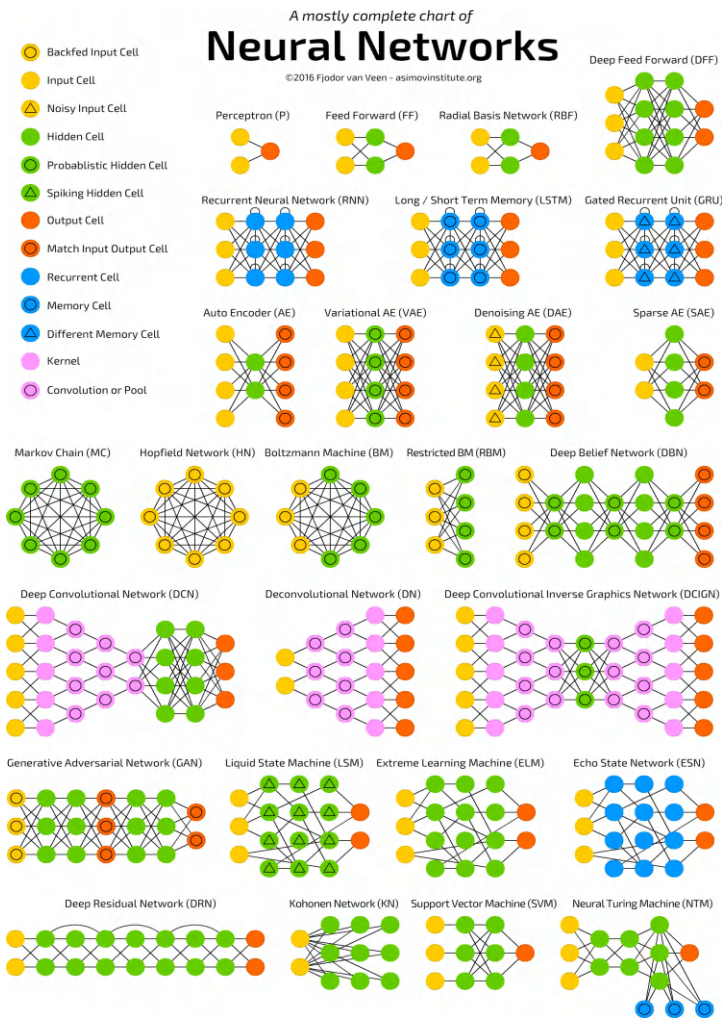


Figure 2.7. Representation of various network architectures.

Source: Geoffrey E. Hinton, Creative Commons License.

And can also be regularized using common techniques like dropout or noise injection. Recurrence plays a crucial role in tackling complex problems, such as language, as it appears to be a fundamental aspect of various brain mechanisms. Figure 2.7 provides a visual representation of various neural networks, such as recurrent networks (Graves et al., 2007).

Jordan introduced the fundamental structure of Recurrent Neural Networks (RNNs) as feed-forward networks that consist of a solitary hidden layer, enhanced with separate units. The output node data are sent by special units, which subsequently relay these values to any hidden nodes in the subsequent time step. Through the utilization of specialized units, the network can retain information regarding previous activities, provided that the output values align with those specific actions. In addition, the nodes within the Jordan networks have self-connections (Dernoncourt et al., 2017).

The architecture recommended by Elman is simpler. Every unit on a hidden layer follows with a context unit. Every unit in the network receives input from the previously executed step through a fixed-weighted edge. The value is then fed to an identical hidden node j through a standard edge. This architecture can be seen as a basic RNN, where each hidden node is connected to itself through a recurrent edge. The concept of self-connected hidden nodes through fixed-weight recurrent edges is crucial in the later research on LSTM networks (Hammer, 2000).

RNNs are a type of architecture that can be used to learn patterns that occur over time or in a specific order. A recurrent neural network (RNN) can anticipate the subsequent data point within a sequence by leveraging the information from the preceding data samples. As an example, in text analysis, a technique involves using a sliding window to examine preceding words and make predictions about the subsequent word or group of words within the sentence. Recurrent neural networks (RNNs) are commonly trained using the long short-term memory (LSTM) algorithm developed by Schmidhuber or gated recurrent units (GRUs). On the other hand, training them to capture long-term dependencies can be quite challenging due to the notorious issues of gradient vanishing and gradient explosion, as well as the need for meticulous optimization of hyperparameters (Schäfer & Zimmermann, 2007).

Deep neural network models have gained significant popularity in recent years, particularly due to the emergence of various techniques. These include bidirectional learning, which involves predicting sequences in both forward and backward directions, as well as attentive mechanisms that enable the utilization of dynamic-size window sliding. These advancements have proven especially valuable in constructing language models (Rodriguez et al., 1999).

Figure 2.8 illustrates the interplay between multiple RNNs and patterns of vectors for the input and output. Each rectangle symbolizes a single vector. Input vectors are located at the bottom, whereas the output vectors are placed at the top. The middle rectangles are used to store the state of the RNN.

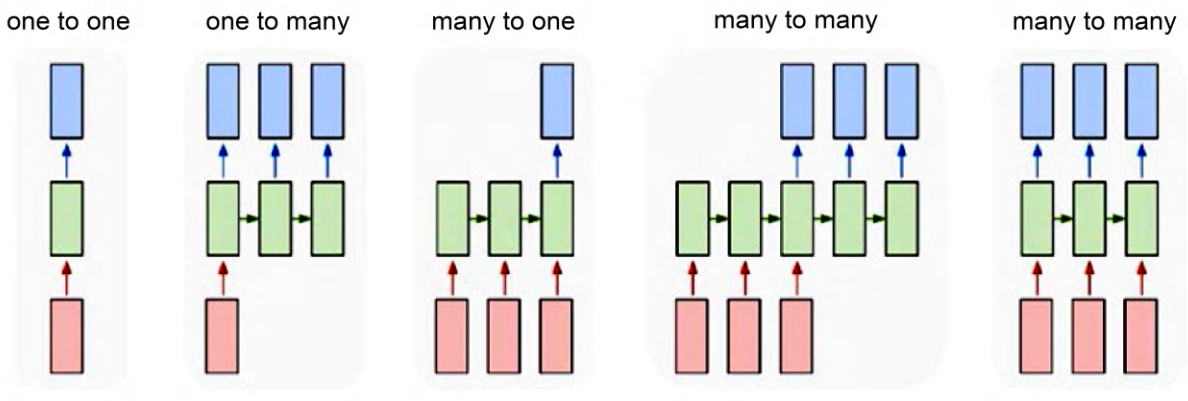


Figure 2.8. Illustration of Topologies of recurrent networks.

Source: Chris Olah, Creative Commons License.

2.6.1. RNNs for Reinforcement Learning

Reinforcement learning (RL) involves the use of reward signals that are delayed refining the parameters from the learning machine. One of the most difficult obstacles in RL involves tasks where the environment's state is just partially observable, requiring consideration of hidden states. These tasks are often called non-Markovian tasks and partially observable Markov decision processes (POMDP). Several pragmatic problems belong to this group, including maze navigation challenges. Hidden states, on the other hand, add complexity to the problem as the agents must not only learn how to map environmental states to actions but also accurately identify their current environmental state at every location (Perrusquía & Yu, 2021).

RNNs trained with LSTM are well-suited for tackling intricate tasks, especially in cases where there is no pre-existing model of the environment. It is possible to construct an online model that can learn to make predictions based on observations and rewards. This enables the model to comprehend the surroundings and decompose it into a sequence of Markovian subtasks. Subsequently, each subtask can be resolved by an active controller that associates observations with suitable actions. An alternate approach involves addressing the secret condition by integrating the selected action with both the present observation and an illustration of past observations and acts. The current observing, when paired with an illustration of the past, is thought to be capable of generating a Markovian state signal (Mousavi et al., 2018).

When dealing with a maze navigation task, it can be challenging for various methods to handle long-term dependencies between events. This is particularly true when T-junctions appear identical and the only way to differentiate them is by considering the previous sequence of events. In these instances, finding a simple solution to break down the task into smaller, sequential steps is not possible. Instead, the agent is required to retain and recall the pertinent information. Schmidhuber proposed LSTM units as a solution to address this issue. These units incorporate a memory state or a forgetting term which are learned from data, as shown in Figure 2.9.

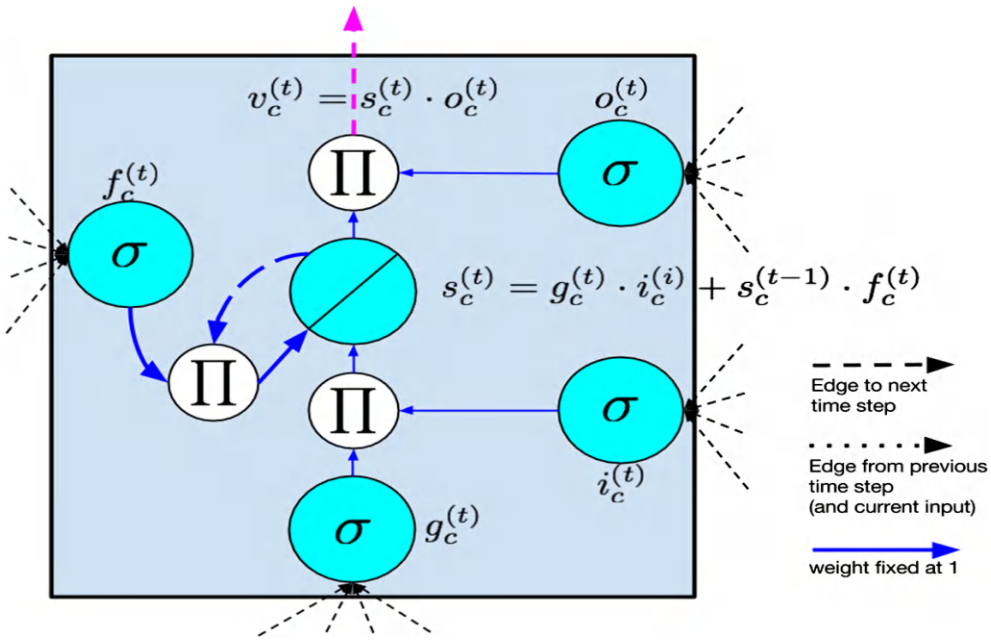


Figure 2.9. Illustration of a memory gate that allows for forgetting in an LSTM cell.

Source: Jacob Andreas, Creative Commons License.

Significant advancements have been made in reinforcement learning, which involves training an agent to make optimal decisions in a specific environment to maximize its overall reward. These advancements have been achieved by harnessing the power of deep learning to create effective feature representations (Geravanchizadeh & Roushan, 2021).

2.6.2. LSTMs

RNNs can connect past information to solve the current task. As an illustration, one can utilize preceding words to anticipate the subsequent word within a sentence. LSTM networks have a unique form of RNN that can grasp long-term connections. These models were first introduced in 1997 by Hochreiter and Schmidhuber. Over time, it has been refined and has gained popularity in various fields, including language translation and video processing.

LSTMs were specifically developed to address issues related to long-term dependencies and the challenges of vanishing and exploding gradients. The LSTM's repeating module consists of four interconnected layers: cell state, input, output, and forget gate. The LSTM architecture possesses the capability to selectively modify or incorporate information into the cell state, which is governed by gates that effectively regulate the flow of information. Gate structures consist of a neural neuron with a sigmoid or tanh activation function, combined with a point-wise multiplication operation (Greff et al., 2016).

In LSTM, every memory cell is equipped with a node that has a recurrent edge of weight 1, allowing the gradient to propagate through multiple time steps without diminishing or amplifying excessively (Eck & Schmidhuber, 2002).

Recurrent neural networks possess the ability to retain information over long periods, thanks to the slowly changing weights during training. Additionally, also have short-term memory in the form of activations that are passed from one node to the next. The LSTM architecture includes a memory cell that serves as an intermediate storage component. A memory cell consists of multiple components (Gers et al., 2002).

- Input Node: This unit functions as a node that gets activation from either the input layers from the current time step or the hidden layer from the prior time step (t–1). The whole weighted input is calculated using a hyperbolic tangent function of activation.
- Input Gate: A gate is a component that receives activating signals on the current data x (t) and the hidden layer from the previous time step. Nevertheless, its importance lies in its ability to multiply the value for a different node, rather than simply adding it. If the current value is 0 suggests that the input should be ignored, a value equals 1 indicates that the input should be fully retained in the memory. The flow of information from every other node becomes disconnected (Sundermeyer et al., 2015).
- Internal State: This is a recurrent edge that is self-connected and has a constant unit weight. Due to the constant weight of this edge, faults can propagate across neighboring time steps without diminishing or amplifying.
- Forget Gates: Understanding the importance of these models is crucial in the network to effectively release the information stored in its internal state (Camargo et al., 2019).
- Output Gate: The value kept within a cell of memory is chosen by the interaction between the internal state and the output gate. The starting internal state is processed using a tanh-activated work to ensure that the output over each cell keeps a similar dynamic range as the typical tanh hidden unit (Vaswani et al., 2016).

LSTM has proven to be highly effective in handling classification and prediction tasks for evolving time series. And have consistently outperformed traditional methods including hidden Markov models and various sequence learning approaches in a diverse array of applications. However, it can be quite demanding in terms of computational resources (Li et al., 2015).

GRUs, or Gated Recurrent Units, are a type of recurrent neural network (RNN), were first developed by Felix Greff, who initially referred to them as forget gates The forget and input gates are merged to form a single "update gate." In addition, the cell state and hidden state are merged along with other adjustments. The final model has seen a rise in popularity due to its reduced complexity in comparison to standard LSTM models. Greff conducted a thorough analysis of various popular variants and discovered that it is nearly impossible to differentiate (Soutner & Müller, 2013).

ACTIVITY 2.1.

Objective:

To gain hands-on experience in constructing, training, and evaluating a CNN using the MNIST dataset.

Materials Needed:

- A computer with Python installed
- TensorFlow or PyTorch library
- MNIST dataset

Steps:

1. Setup:
   - Install TensorFlow or PyTorch.
   - Import necessary libraries and load the MNIST dataset.
2. Model Construction and Compilation:
   - Define a simple CNN architecture with convolutional, pooling, and fully connected layers.
   - Compile the model with a loss function (e.g., categorical cross-entropy) and an optimizer (e.g., Adam).
3. **Model Training and Evalua**tion:
   - Train the CNN on the training data.
   - Evaluate the trained model on the test data.

SUMMARY

- The chapter provides an overview of various neural network architectures and models. It begins with a historical perspective on neural networks, highlighting their evolution. The Multilayer Perceptron (MLP) is introduced as a fundamental neural network model. Deep Neural Networks (DNNs) are explored next, emphasizing their increased depth and complexity compared to traditional models.
- Boltzmann Machines and Restricted Boltzmann Machines (RBMs) are discussed in the context of probabilistic models. Contrastive Divergence is presented as a training method for RBMs. Deep Belief Nets (DBNs) and Deep Boltzmann Machines (DBMs) are covered as hierarchical probabilistic models.
- Convolutional Neural Networks (CNNs) are detailed, focusing on their architecture suited for processing grid-like data such as images. Deep Auto-encoders are introduced as neural networks used for unsupervised learning tasks. Recurrent Neural Networks (RNNs) and their applications in tasks like reinforcement learning are discussed, with specific attention to Long Short-Term Memory networks (LSTMs).

REVIEW QUESTIONS

1. Briefly explain the historical progression from the Multilayer Perceptron to Deep Neural Networks.
2. What are the key differences between Boltzmann Machines and Restricted Boltzmann Machines?
3. Describe the Contrastive Divergence algorithm and its role in training Boltzmann Machines.
4. Summarize the primary characteristics and applications of Convolutional Neural Networks (CNNs).
5. What are Deep Auto-encoders and how do they differ from traditional auto-encoders?
6. Explain the significance of Long Short-Term Memory (LSTM) networks in handling sequential data.

MULTIPLE CHOICE QUESTIONS

1. What is the primary function of Contrastive Divergence in training Boltzmann Machines?

a. Regularization

b. Feature extraction

c. Error minimization

d. Parameter estimation

2. **Which type of neural network architecture is specifically designed for processing sequential data?**
   a. Recurrent Neural Networks
   b. Convolutional Neural Networks
   c. Deep Auto-encoders
   d. Deep Belief Nets
3. **`Which neural network model is well-suited for image recognition tasks due to its ability to capture spatial hierarchies?**
   a. Restricted Boltzmann Machines
   b. Deep Belief Nets
   c. Convolutional Neural Networks
   d. Deep Boltzmann Machines
4. **What distinguishes Long Short-Term Memory (LSTM) networks from traditional Recurrent Neural Networks (RNNs)?**
   a. LSTMs have deeper layers
   b. LSTMs have an internal memory mechanism
   c. LSTMs use convolutional filters
   d. LSTMs are faster to train
5. **Which generative model framework leverages a competition between a generator and a discriminator to generate realistic data samples?**
   a. Variational Auto-encoders
   b. Deep Boltzmann Machines
   c. Generative Adversarial Networks
   d. Deep Belief Nets
6. **What is the primary objective of Variational Auto-encoders (VAEs) in neural network applications?**
   a. Discrimination of data classes
   b. Feature extraction from raw data
   c. Generation of new data samples
   d. Compression of input data

Answers to Multiple Questions

1. (d); 2. (a); 3. (c); 4. (b); 5. (c); 6. (c).

REFERENCES

1. Ackley, D. H., Hinton, G. E., & Sejnowski, T. J. (1985). A learning algorithm for Boltzmann machines. Cognitive Science, 9(1), 147–169. https://doi.org/10.1016/S0364-0213(85)80012-4.
2. Attali, J. G., & Pagès, G. (1997). Approximations of functions by a multilayer perceptron: A new approach. Neural Networks, 10(6), 1069–1081. https://doi.org/10.1016/S0893-6080(97)00025-3.
3. Basati, A., & Faghih, M. M. (2022). PDAE: Efficient network intrusion detection in IoT using parallel deep auto-encoders. Information Sciences, 598, 57–74. https://doi.org/10.1016/j.ins.2022.03.016.
4. Bau, D., Zhu, J. Y., Strobelt, H., Lapedriza, A., Zhou, B., & Torralba, A. (2020). Understanding the role of individual units in a deep neural network. Proceedings of the National Academy of Sciences, 117(48), 30071–30078. https://doi.org/10.1073/pnas.1907375117.
5. Bengio, Y., & Delalleau, O. (2009). Justifying and generalizing contrastive divergence. Neural Computation, 21(6), 1601–1621. https://doi.org/10.1162/neco.2009.06-08-804.
6. Bilal, A., Jourabloo, A., Ye, M., Liu, X., & Ren, L. (2017). Do convolutional neural networks learn class hierarchy? IEEE Transactions on Visualization and Computer Graphics, 24(1), 152–162. https://doi.org/10.1109/TVCG.2017.2743472.
7. Camargo, M., Dumas, M., & González-Rojas, O. (2019). Learning accurate LSTM models of business processes. In Business Process Management: 17th International Conference, BPM 2019, Vienna, Austria, September 1–6, 2019, Proceedings (pp. 286–302). Springer. https://doi.org/10.1007/978-3-030-26619-6_18.
8. Dernoncourt, F., Lee, J. Y., Uzuner, O., & Szolovits, P. (2017). De-identification of patient notes with recurrent neural networks. Journal of the American Medical Informatics Association, 24(3), 596–606. https://doi.org/10.1093/jamia/ocw156.
9. Eck, D., & Schmidhuber, J. (2002). A first look at music composition using LSTM recurrent neural networks. Istituto Dalle Molle Di Studi Sull'Intelligenza Artificiale, 103(4), 48.
10. Fischer, A., & Igel, C. (2014). Training restricted Boltzmann machines: An introduction. Pattern Recognition, 47(1), 25–39. https://doi.org/10.1016/j.patcog.2013.05.025.
11. Fong, R., Patrick, M., & Vedaldi, A. (2019). Understanding deep networks via extremal perturbations and smooth masks. In Proceedings of the IEEE/CVF International Conference on Computer Vision (pp. 2950–2958). https://doi.org/10.1109/ICCV.2019.00304.
12. Gardner, M. W., & Dorling, S. R. (1998). Artificial neural networks (the multilayer perceptron)—A review of applications in the atmospheric sciences. Atmospheric Environment, 32(14–15), 2627–2636. https://doi.org/10.1016/S1352-2310(97)00447-0.
13. Geravanchizadeh, M., & Roushan, H. (2021). Dynamic selective auditory attention detection using RNN and reinforcement learning. Scientific Reports, 11(1), 15497. https://doi.org/10.1038/s41598-021-94632-1.

14. Gers, F. A., Schraudolph, N. N., & Schmidhuber, J. (2002). Learning precise timing with LSTM recurrent networks. Journal of Machine Learning Research, 3, 115–143. https://doi.org/10.1162/jmlr.2002.3.4-5.115.

15. Golovko, V., Kroshchanka, A., & Treadwell, D. (2016). The nature of unsupervised learning in deep neural networks: A new understanding and novel approach. Optical Memory and Neural Networks, 25(3), 127–141. https://doi.org/10.3103/S1060992X16030038.

16. Graves, A., Fernández, S., & Schmidhuber, J. (2007). Multi-dimensional recurrent neural networks. In International Conference on Artificial Neural Networks (pp. 549–558). Springer. https://doi.org/10.1007/978-3-540-74690-4_55.

17. Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., & Schmidhuber, J. (2016). LSTM: A search space odyssey. IEEE Transactions on Neural Networks and Learning Systems, 28(10), 2222–2232. https://doi.org/10.1109/TNNLS.2016.2582924.

18. Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., & Chen, T. (2018). Recent advances in convolutional neural networks. Pattern Recognition, 77, 354–377. https://doi.org/10.1016/j.patcog.2017.10.013.

19. Hammer, B. (2000). On the approximation capability of recurrent neural networks. Neurocomputing, 31(1–4), 107–123. https://doi.org/10.1016/S0925-2312(99)00182-1.

20. Hinton, G. E. (2002). Training products of experts by minimizing contrastive divergence. Neural Computation, 14(8), 1771–1800. https://doi.org/10.1162/089976602760128018.

21. Hinton, G. E., Osindero, S., & Teh, Y. W. (2006). A fast-learning algorithm for deep belief nets. Neural Computation, 18(7), 1527–1554. https://doi.org/10.1162/neco.2006.18.7.1527.

22. Hjelm, R. D., Calhoun, V. D., Salakhutdinov, R., Allen, E. A., Adali, T., & Plis, S. M. (2014). Restricted Boltzmann machines for neuroimaging: An application in identifying intrinsic networks. NeuroImage, 96, 245–260. https://doi.org/10.1016/j.neuroimage.2014.03.048.

23. Jin, J., Zhang, C., Feng, F., Na, W., Ma, J., & Zhang, Q. J. (2019). Deep neural network technique for high-dimensional microwave modeling and applications to parameter extraction of microwave filters. IEEE Transactions on Microwave Theory and Techniques, 67(10), 4140–4155. https://doi.org/10.1109/TMTT.2019.2933310.

24. Karim, A. M., Kaya, H., Güzel, M. S., Tolun, M. R., Çelebi, F. V., & Mishra, A. (2020). A novel framework using deep auto-encoders based linear model for data classification. Sensors, 20(21), 6378. https://doi.org/10.3390/s20216378.

25. Kriegeskorte, N., & Golan, T. (2019). Neural network models and deep learning. Current Biology, 29(7), R231–R236. https://doi.org/10.1016/j.cub.2019.02.034.

26. Larochelle, H., & Bengio, Y. (2008). Classification using discriminative restricted Boltzmann machines. In Proceedings of the 25th International Conference on Machine Learning (pp. 536–543). https://doi.org/10.1145/1390156.1390224.

27. Lavin, A., & Gray, S. (2016). Fast algorithms for convolutional neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 4013–4021). https://doi.org/10.1109/CVPR.2016.435.

28. Le Roux, N., & Bengio, Y. (2010). Deep belief networks are compact universal approximators. Neural Computation, 22(8), 2192–2207. https://doi.org/10.1162/neco.2010.06-09-1047.

29. Li, J., Mohamed, A., Zweig, G., & Gong, Y. (2015). LSTM time and frequency recurrence for automatic speech recognition. In 2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU) (Vol. 3, No. 2, pp. 187–191). IEEE. https://doi.org/10.1109/ASRU.2015.7404827.

30. Li, Z., Liu, F., Yang, W., Peng, S., & Zhou, J. (2021). A survey of convolutional neural networks: Analysis, applications, and prospects. IEEE Transactions on Neural Networks and Learning Systems, 33(12), 6999–7019. https://doi.org/10.1109/TNNLS.2021.3084827.

31. Mehrer, J., Spoerer, C. J., Kriegeskorte, N., & Kietzmann, T. C. (2020). Individual differences among deep neural network models. Nature Communications, 11(1), 5725. https://doi.org/10.1038/s41467-020-19632-w.

32. Mohamed, A. R., Dahl, G. E., & Hinton, G. (2011). Acoustic modeling using deep belief networks. IEEE Transactions on Audio, Speech, and Language Processing, 20(1), 14–22. https://doi.org/10.1109/TASL.2011.2109382.

33. Montavon, G., Samek, W., & Müller, K. R. (2018). Methods for interpreting and understanding deep neural networks. Digital Signal Processing, 73, 1–15. https://doi.org/10.1016/j.dsp.2017.10.011.

34. Mousavi, S. S., Schukat, M., & Howley, E. (2018). Deep reinforcement learning: An overview. In Proceedings of SAI Intelligent Systems Conference (IntelliSys) 2016 (Vol. 2, pp. 426–440). Springer International Publishing. https://doi.org/10.1007/978-3-319-56991-8_33.

35. Movahedi, F., Coyle, J. L., & Sejdiź, E. (2017). Deep belief networks for electroencephalography: A review of recent contributions and future outlooks. IEEE Journal of Biomedical and Health Informatics, 22(3), 642–652. https://doi.org/10.1109/JBHI.2017.2738481.

36. Passos, L. A., & Papa, J. P. (2020). A metaheuristic-driven approach to fine-tune deep Boltzmann machines. Applied Soft Computing, 97, 105717. https://doi.org/10.1016/j.asoc.2020.105717.

37. Perrusquía, A., & Yu, W. (2021). Identification and optimal control of nonlinear systems using recurrent neural networks and reinforcement learning: An overview. Neurocomputing, 438, 145–154. https://doi.org/10.1016/j.neucom.2021.01.118.

38. Popescu, M. C., Balas, V. E., Perescu-Popescu, L., & Mastorakis, N. (2009). Multilayer perceptron and neural networks. WSEAS Transactions on Circuits and Systems, 8(7), 579–588.

39. Rodriguez, P., Wiles, J., & Elman, J. L. (1999). A recurrent neural network that learns to count. Connection Science, 11(1), 5–40. https://doi.org/10.1080/095400999116340.

40. Ruck, D. W., Rogers, S. K., & Kabrisky, M. (1990). Feature selection using a multilayer perceptron. Journal of Neural Network Computing, 2(2), 40–48.

41. Salakhutdinov, R., & Murray, I. (2008). On the quantitative analysis of deep belief

networks. In Proceedings of the 25th International Conference on Machine Learning (pp. 872–879). ACM. https://doi.org/10.1145/1390156.1390265.

42. Saleem, R., Yuan, B., Kurugollu, F., Anjum, A., & Liu, L. (2022). Explaining deep neural networks: A survey on the global interpretation methods. Neurocomputing, 513, 165–180. https://doi.org/10.1016/j.neucom.2022.09.013.

43. Samek, W., Binder, A., Montavon, G., Lapuschkin, S., & Müller, K. R. (2016). Evaluating the visualization of what a deep neural network has learned. IEEE Transactions on Neural Networks and Learning Systems, 28(11), 2660–2673. https://doi.org/10.1109/TNNLS.2016.2599820.

44. Samek, W., Montavon, G., Lapuschkin, S., Anders, C. J., & Müller, K. R. (2021). Explaining deep neural networks and beyond: A review of methods and applications. Proceedings of the IEEE, 109(3), 247–278. https://doi.org/10.1109/JPROC.2021.3056294.

45. Schäfer, A. M., & Zimmermann, H. G. (2007). Recurrent neural networks are universal approximators. International Journal of Neural Systems, 17(04), 253–263. https://doi.org/10.1142/S0129065707001059.

46. Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, 45(11), 2673–2681. https://doi.org/10.1109/78.650093.

47. Soutner, D., & Müller, L. (2013). Application of LSTM neural networks in language modelling. In Text, Speech, and Dialogue: 16th International Conference, TSD 2013, Pilsen, Czech Republic, September 1–5, 2013, Proceedings (pp. 105–112). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-40585-3_14.

48. Srinidhi, C. L., Ciga, O., & Martel, A. L. (2021). Deep neural network models for computational histopathology: A survey. Medical Image Analysis, 67, 101813. https://doi.org/10.1016/j.media.2020.101813.

49. Sundermeyer, M., Ney, H., & Schlüter, R. (2015). From feedforward to recurrent LSTM neural networks for language modeling. IEEE/ACM Transactions on Audio, Speech, and Language Processing, 23(3), 517–529. https://doi.org/10.1109/TASLP.2015.2400218.

50. Sze, V., Chen, Y. H., Yang, T. J., & Emer, J. S. (2017). Efficient processing of deep neural networks: A tutorial and survey. Proceedings of the IEEE, 105(12), 2295–2329. https://doi.org/10.1109/JPROC.2017.2761740.

51. Taherkhani, A., Cosma, G., & McGinnity, T. M. (2018). Deep-FS: A feature selection algorithm for deep Boltzmann machines. Neurocomputing, 322, 22–37. https://doi.org/10.1016/j.neucom.2018.09.047.

52. Tang, J., Deng, C., & Huang, G. B. (2015). Extreme learning machine for multilayer perceptron. IEEE Transactions on Neural Networks and Learning Systems, 27(4), 809–821. https://doi.org/10.1109/TNNLS.2015.2424995.

53. Tong, C., Li, J., Lang, C., Kong, F., Niu, J., & Rodrigues, J. J. (2018). An efficient deep model for day-ahead electricity load forecasting with stacked denoising auto-encoders. Journal of Parallel and Distributed Computing, 117, 267–273. https://doi.org/10.1016/j.jpdc.2017.09.011.

54. Upadhya, V., & Sastry, P. S. (2019). An overview of restricted Boltzmann machines. Journal of the Indian Institute of Science, 99, 225–236. https://doi.org/10.1007/s41745-019-0111-4.

55. Vaswani, A., Bisk, Y., Sagae, K., & Musa, R. (2016). Super tagging with LSTMs. In Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Vol. 5, No. 3, pp. 232–37). https://doi.org/10.18653/v1/N16-1027.

56. Yamashita, R., Nishio, M., Do, R. K. G., & Togashi, K. (2018). Convolutional neural networks: An overview and application in radiology. Insights into Imaging, 9, 611–629. https://doi.org/10.1007/s13244-018-0639-9.

57. Yu, L., Song, J., Song, Y., & Ermon, S. (2021). Pseudo-spherical contrastive divergence. In Advances in Neural Information Processing Systems, 34, 22348–22362. https://doi.org/10.5555/3546258.3546312.

58. Zhang, C., Bengio, S., Hardt, M., Recht, B., & Vinyals, O. (2021). Understanding deep learning (still) requires rethinking generalization. Communications of the ACM, 64(3), 107–115. https://doi.org/10.1145/3446776.

59. Zhang, N., Ding, S., Zhang, J., & Xue, Y. (2018). An overview on restricted Boltzmann machines. Neurocomputing, 275, 1186–1199. https://doi.org/10.1016/j.neucom.2017.09.065.

CHAPTER

3 Deep Reinforcement Learning

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

- Understand Markov Decision Processes (MDPs) and value iteration in RL.
- Explore Q-learning and deep-Q learning for complex environment solutions.
- Understand policy gradient methods for learning optimal policies in RL.
- Know the application of neural networks in continuous state space problems.
- Grasp the benefits of actor-critic methods in RL efficiency and stability.

KEY TERMS FROM THIS CHAPTER

Actor-critic methods
Deep-Q learning
Neural networks
Q-learning
Value iteration
Continuous state space
Markov decision processes (MDPs)
Policy gradient methods
Reinforcement learning

UNIT INTRODUCTION

Reinforcement learning, or RL for short, is a fascinating field within machine learning that focuses on teaching an agent how to make decisions in an environment to achieve the highest possible reward. It is natural in deep reinforcement learning to limit the learning approach to deep learning.

In academic writing, it is common to define the environment mathematically to be a Markov decision process (MDP). MDPs involve a collection of states, denoted as s ∈ S, which represent various locations on a map. The agent has a finite set of actions, denoted as a ∈ A. The function T (s, a, s') = P ($\mathbf{S}_{t+1}$ = s' | $\mathbf{S}_t$ = s, A = a) allows the agent to transition from one state to another, A reward function is defined as a mapping from a state, action, and subsequent state to real numbers R (s, a, s'). Additionally, one introduces a discount factor γ ∈ [0, 1] which will be explained shortly (Mousavi et al., 2018). Typically, actions have a probabilistic nature, meaning that T represents a distribution that determines the likely resulting state when an action is taken in a given state. The models are referred to as Markov decision processes due to their adherence to the Markov assumption, which states that the history of how one arrives at the current state is irrelevant as long as one knows the present state (Arulkumaran et al., 2017).

MDPs operate on a discrete time scale. During the agent's operation, it is situated in a particular state, where it selects an action that results in a transition to a different state, and subsequently receives a reward, typically with a value of zero (Gronauer & Diepold, 2022).

The objective is to optimize the discounted future reward, as specified by

$$\sum_{t=0}^{t=\infty} \gamma^t R(s_t, a_t, s_{t+1}) \tag{3.1}$$

If γ < 1, then the sum mentioned is finite. If the value of γ is absent or equal to 1, the sum has the potential to increase indefinitely, leading to a more intricate analysis in mathematics. A common value for γ is .9. The term γ^t in Equation 3.1 is referred to as discounted future reward as the iterative multiplication by a value less than one results in the model assigning less importance to rewards in the future in comparison to immediate rewards. This approach is sensible considering the finite lifespan of individuals (François et al., 2018).

To compute the optimal policy, which is a strategy that maximizes the expected cumulative reward, the Value Iteration algorithm is a fundamental method in a Markov Decision Process MDP. A policy refers to a function π(s) = a that determines the action an agent should take for each specified state. An optimal policy, represented as $\eth^*$(s), results in the highest hoped-for discounted future reward when specific actions are taken (Ladosz et al., 2022).

1. For all s set $V(s) = 0$
2. Repeat until convergence:
   (a) For all s:
      i. For all a, set $Q(s,a) = \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V(s'))$
      ii. $V(s) = max_a Q(s,a)$
3. return Q

Figure 3.1. Illustration of the value iteration algorithm.

Source: Eugene Charniak, Creative Commons License.

This chapter explains integrating reinforcement learning alongside deep learning techniques. This comprehensive guide explores the core principles and algorithms. It delves into essential concepts such as Actor-Critic Methods, Q-learning, Value Iteration, Policy Gradient Methods, and Basic Deep-Q Learning (DQN). The main emphasis is placed on comprehending the utilization of artificial neural networks to estimate value functions and policies to tackle intricate decision-making problems in spaces with a high number of dimensions. This unit offers an in-depth comprehension of the theoretical and practical elements of deep reinforcement teaching (Wang et al., 2022).

3.1. VALUE ITERATION

Before delving into the intricacies of solving MDPs, it is crucial to address a fundamental question: does the agent possess prior knowledge of the functions T and R, or does it need to explore the environment and learn these functions while formulating its policy? It becomes much simpler when one knows T or R, so begin with that scenario (Spaan & Vlassis, 2005). Within this section, it is assumed that there exists a limited quantity of states, denoted as "s".

Value iteration is the simplest technique for policy learning in MDPs. (In fact, it can be argued that this algorithm is not a traditional learning algorithm, as it does not require training examples or direct interaction with the environment.) The algorithm can be found in Figure 3.1. V represents a value function that consists of a vector with a size of |s|. Each entry in V(s) relates to the optimal expected discounted reward that can be achieved when starting in different states. The Q function, also known as Q, is a table that holds the current estimation of the discounted reward for taking action in states. It has a size of |s| by |a|. The value function V assigns a real-number value to each state, indicating the desirability of reaching that state. Higher values correspond to more favorable states. The Q value provides a more detailed analysis by estimating the expected values for every state-action pair. When values in V are accurate, then line 2(a)i may appropriately set Q (s, a). The value for Q (s, a) is determined by adding the immediate reward R (s, a, s') to the value for the resulting state, as indicated by V. Given the non-deterministic nature of actions, it becomes necessary to consider the summation of all potential states one can expect (Dai et al., 2011).

0:S	1:F	2:F	3:F
4:F	5:H	6:F	7:H
8:F	9:F	10:F	11:H
12:H	13:F	14:F	15:G

S	starting location
F	frozen location
H	hole
G	goal location

Figure 3.2. Illustration of the frozen-lake problem.

Source: Yoshua Bengio, Creative Commons License.

After obtaining the correct Q value, one can establish the optimal policy π by consistently selecting the action a = arg $\mathbf{max}_{a'}$ Q (s, $\mathbf{a}'$). The function arg $\mathbf{max}_x$ g(x) returns the value of x that maximizes g(x).

Now examine a straightforward MDP known as the frozen-lake problem. This particular game is just one of several included in the Open AI Gym, which is a collection of

computer games designed for experimentation with reinforcement learning. The games all have consistent APIs, making them ideal for academic research in this field. Figure 3.2 displays a 4 * 4 grid, which represents the lake. The game aims to navigate from the initial position (state 0 at the upper left) towards the destination (lower right) while avoiding any pitfalls in the ice. Whenever an action leads us to the goal state, may get rewarded with a value of 1. Each of the other state-action-state triples receives a reward of zero. When encountering a hole state or reaching the goal state, the game comes to a halt. Upon restarting, get returned to the initial state (Luo et al., 2019). Alternatively, the down (d), left (l), right (r), or up (u) directions (represented by the numbers zero to three) may be taken, although there is a chance of deviating from the intended path. Indeed, the open-ended AI Gym game is designed in such a way that when one takes an action, such as moving right, there is an equal probability of transitioning to any one of the nearest states, except for the exact opposite direction (e.g., left). This design makes the game quite slippery and unpredictable. If a particular action would cause us to move away from the lake, it rather maintain a team in the same state where one began (Mann et al., 2015).

0	0	0	0
0	0	0	0
0	0	0	0
0	0	.33	0

0	0	0	0
0	0	0	0
0	0	.1	0
0	.1	.46	0

Figure 3.3. Illustration of state values after the first and second iterations of value iteration.

Source: Peter Brown, Creative Commons License.

To calculate V and Q to the frozen lake, it is necessary to iterate with all states s or recalculate V (s) multiple times. Now examine state 1. To determine the optimal action for a given state, it is necessary to calculate the Q (1, a) value for each of the available actions. The maximum Q value is selected and assigned to the corresponding state value, V (1). Alright, now begin by calculating the outcome of moving to the left, Q (1, l). To accomplish this, it is necessary to calculate the sum of all game states, denoted as s'. In the game, there are a total of 16 states. However, when starting from state 1, one can reach three among them with a probability greater than zero. These states include states 0, 5, or 1 itself (Goyal & Grand, 2023). This limitation occurs when attempting to move up, but being blocked through the lake boundary, resulting in no movement at all:

$$Q(1, l) = .33 \bullet (0 + .9 \bullet 0) + ..33 \bullet (0 + .9 \bullet 0) + .33 \bullet (0 + .9 \bullet 0) \quad (3.2)$$

$$= 0 + 0 + 0 \quad (3.3)$$

$$= 0 \quad (3.4)$$

One of the terms in the equation indicates that there is a 33% chance of transitioning to state 0 when trying to move left. There is no reward for this action, and the expected

future reward is calculated as 0.9 multiplied by 0. The value is zero, which occurs when one chooses to go left, slip down to state 5, or stay in state 1. Therefore, Q (1, l) = zero. Since the V values of the three states that can be reached to state 1 are all 0, it follows that Q (1, d) and Q (1, u) are also 0. Consequently, line 2(a)ii establishes V (1) as 0.

Interestingly, during the initial iteration, V remains at zero until one reaches state 14. It is at this point that one observes non-zero values of Q(14, d), Q(14, r), and Q(14, u):

$$Q\ (14,\ d) = .33 \cdot (0 + .9 \cdot 0) + .33 \cdot (0 + .9 \cdot 0) + .33 \cdot (1 + .9 \cdot 0) = .33$$

$$Q\ (14,\ r) = .33 \cdot (0 + .9 \cdot 0) + .33 \cdot (0 + .9 \cdot 0) + .33 \cdot (1 + .9 \cdot 0) = .33$$

$$Q\ (14,\ u) = .33 \cdot (0 + .9 \cdot 0) + .33 \cdot (0 + .9 \cdot 0) + .33 \cdot (1 + .9 \cdot 0) = .33$$

and V (14) = .33.

Displayed in Figure 3.3 is a table for V values following the initial iteration. Value iteration is a prominent algorithm used to achieve an optimal policy by maintaining tables of the most accurate function value estimates. Thus, the term "tabular methods" is derived (Chen & Meyn, 1999).

```
0  import gym
1  game = gym.make('FrozenLake-v0')
2  for i in range(1000):
3     st = game.reset()
4     for stps in range(99):
5          act=np.random.randint(0,4)
6          nst,rwd,dn,_=game.step(act)
7          # update T and R
9          if dn: break
```

Figure 3.4. Illustration of collecting statistics for an open AI gym game.

Source: John Cocke, Creative Commons License.

In the second iteration, like before, most values remain at 0. However, this time, states 10 and 13 contain non-zero Q and V entries. This is because one can now move from these states to state 14, and has just observed, that V (14) is now equal to 0.33. The V values after the second iteration are displayed on the right-hand side of Figure 3.3. Another perspective on value iteration suggests that each update to V (and Q) integrates precise knowledge about the immediate future outcome (reward R), only to revert toward the initially imprecise information already embedded in these functions. Over time, the functions gradually incorporate additional details about states that have not been encountered yet (Wei et al., 2015).

3.2. Q-LEARNING

Value iteration assumes that the learner possesses comprehensive knowledge of the model environment. Now explore the contrasting scenario of model-free learning, this is the Q-learning algorithm. The agent can navigate the environment by taking actions, and receiving feedback on the reward and the subsequent state. However, it lacks knowledge of the precise probabilities of movement or the reward function T, R (Watkins & Dayan, 1992).

Given the assumption that our environment follows a Markov decision procedure, a straightforward approach to planning in a model-free setting would involve exploring the environment randomly, gathering data on T and R, and subsequently constructing a policy using the Q table, as explained in the previous section. Figure 3.4 presents the key aspects of a program designed to accomplish this task. A frozen-lake game is created in line 1. At the beginning of a game, one initiates the process by calling the reset () function. One iteration in the frozen-lake game concludes when one encounters a hole or successfully reaches the goal state. The outermost loop (line 2) indicates that the game will be executed 1000 times. In the inner loop of the code (line 4), it is specified that the game is terminated after 99 steps. In practice, it is unlikely to encounter such a scenario where one either gets stuck or achieves the goal before that point (Clifton & Laber, 2020). According to Line 5, the next action is randomly generated at each step. There are four available actions: left, right, and up, down, represented by the numbers 0 to 3 accordingly Step 6 is of utmost importance. The step function accepts a single argument, the action that should be executed, and then provides four values as its output. One aspect to consider is the state of the game after the action is taken, represented by an integer ranging from 0 to 15. Another factor is the reward one obtains, usually 0 but occasionally 1 in FL. In the given figure, the third state, referred to as "dn," serves as an indicator to determine whether the game has ended. It is a binary value, representing either true or false. This state will be true if the agent falls into a hole or reaches the goal. Information regarding the true probability of transition is disregarded in model-free learning (Dayan & Watkins, 1992).

In 2015, DeepMind achieved a breakthrough in AI research with the development of DQN. This groundbreaking achievement paved the way for the development of AlphaGo, an AI that made history by defeating a world champion in the game of Go. Engaging in simultaneous learning and exploration would be more advantageous, allowing the knowledge gained to shape our path. A deep understanding of this game allows one to gain valuable insights. One uncovers new knowledge about a wider range of states with each step forward. This is done by making a decision based on the probability ϵ. There are two options: randomly selecting a move (with probability ϵ) or choosing a

move based on a different probability (1 – ϵ). Make decisions based on what is acquired thus far. When ϵ is held constant, this strategy is called an epsilon-greedy approach (Jang et al., 2019).

It is a commonly observed practice for ϵ to decrease gradually over time, which is known as an epsilon-decreasing strategy. A straightforward approach is to introduce a corresponding hyper parameter E and define ϵ=E/(E+1), where i represents the number of gameplays. (The value of E represents the number of games during which one transitions from a state of mostly random actions to a state of mostly learned actions) As one might anticipate, the decision of whether to explore or rely on existing knowledge of the game can significantly impact the speed at which one acquires new skills. This phenomenon is commonly referred to as the exploration-exploitation tradeoff, where utilizing our game knowledge is considered exploiting the information one has already acquired (Millán et al., 2002).

Fun Fact

The development of DQN by DeepMind in 2015 was a significant milestone in AI research, leading to the creation of AlphaGo, the first AI to defeat a world champion in the game of Go.

One effective approach to balancing exploration and exploitation involves converting the values provided by the Q function into a probability distribution. Instead of always selecting the action using the highest value, an action is chosen based on this distribution. (The second method is referred to as the greedy algorithm.) Therefore, in the given scenario where one has three available actions and their corresponding Q values are [4, 1, 1], it would be reasonable to select the first two actions approximately two-thirds for the time, and so on (Guo et al., 2004).

Q-learning is widely recognized as one of the pioneering and highly regarded algorithms for model-free learning, effectively balancing the crucial aspects of exploration and exploitation. The fundamental concept revolves around the direct learning of the Q or V tables, rather than focusing on learning R and T. In Figure 3.4, there are a couple of modifications that need to be made. Firstly, on line 5, we will no more act completely randomly. Secondly, on line 7, one must change Q and V, not R and T (Kumar et al., 2020).

After providing a thorough explanation at line 5, now shift focus to line 7.

The update equations one uses for Q-learning are

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(R(s,a,n) + \gamma V(n)) \quad (3.5)$$

$$V(s) = \max_{a'} Q(s,a'), \quad (3.6)$$

On line 6 of the figure, one finds in states, takes action a, and transitions to state a' after taking a step within the game.

The updated value of Q (s, a) seems a combination influenced by α, which determines the weight given to the previous value or the new information. α can be thought of as a learning rate. Usually, α tends to be on the smaller side. To clarify the necessity of these equations, it is beneficial to compare them with lines 2(a)i or 2(a)ii via the value iteration algorithm shown in Figure 3.1. There, with the algorithm provided with R and T, one could calculate the sum of all potential outcomes resulting from the action taken. Unfortunately, in Q-learning, achieving this is not possible (Ahmadabadi & Asadpour, 2002). One gets left with only the final result of the action. And findings are derived from a single move in the investigation of the environment. Let's consider the scenario where one finds himself in a state 14 of Figure 3.2. However, what is not known is that there exists a minuscule probability (.0001) that, if one decides to move downwards from this state, thus will meet with a rather unfavorable "reward" of -10. It is highly unlikely that this scenario will occur, but if it does, it will significantly disrupt the current state of affairs. It is important for the algorithm to avoid placing excessive importance on any individual move. Value iteration involves considering both the transition probabilities (T) and the rewards (R). This algorithm takes into account the potential for negative rewards and the low likelihood of their occurrence (Goldberg & Kosorok, 2012).

3.3. BASIC DEEP-Q LEARNING

After mastering tabular Q learning, one is now ready to delve into the realm of deep Q learning. Just like with the tabular version, one begins with the schema depicted in Figure 3.4. This time, the approach is significantly shifting as one no longer relies on a table to represent the Q function. Instead, utilize a neural network model (Ramaswamy & Hüllermeier, 2021). As previously mentioned, machine learning involves the task of approximating a function that closely resembles a target function. For example, the target function could map pixels from an image to a value of ten integers, representing the corresponding digit. One gets provided with the value for the function on specific inputs, and the objective is to develop a function that accurately approximates its output to all those values. In doing so, one aims to determine the results of the function in locations where its value was not provided, when it comes to deep-Q learning, the analogy of function approximation is highly relevant. The main goal is to approximate the Q function, which is unknown, using neural networks. One can achieve this by exploring the Markov decision-making procedure and continuously learning from our experiences (Chen et al., 2020).

It is important to note that the transition from tabular towards deep-learning models cannot be explained by the frozen-lake example. This particular problem is well-suited for tabular Q learning. Deep-Q learning becomes necessary in situations where the number of states is too vast to be feasibly represented in a table format (Kumar et al., 2019).

An important milestone in the resurgence of neural networks was the development of a single model capable of utilizing deep-Q learning across multiple Atari games. This program was developed from DeepMind, a startup that was acquired by Google in 2014. DeepMind successfully trained one program to master various games by utilizing the visual representation of the games through pixel images (Tan et al., 2020). Every combination of pixels represents a state. Without much thought, one cannot recall the exact image size utilized in the study, but if it were as tiny as those 28 ź 28 photos one employed for MNIST, alongside each pixel was binary (on or off), it would result in a staggering 2784 potential combinations of pixel values. Consequently, in theory, this multitude of states should be required in the Q table. In any case, the number of items is far too extensive to be accepted by a tabular scheme (It was discovered that the dimensions of an Atari game window are 210 ź 160 RGB). Yet the DeepMind program effectively reduced this to a smaller size of 84 ź 84 in black and white. It will revisit more complex scenarios beyond the frozen lake environment in subsequent discussions (Zhang et al., 2018).

When it comes to obtaining a movement recommendation, the approach of replacing the Q table with a neural network function simplifies the process. Instead of referring to the Q table directly, one utilizes a one-layer neural network by inputting the state, as depicted in Figure 3.5. The code snippet for generating the Q function's model parameters can be found in Figure 3.6. The current state, represented as the scalar input, is processed through a transformation process. This involves converting it into a one-hot vector, denoted as oneH, which is then passed through a layer of linear units. The shape of Q is 16 * 4, with 16 representing the size of a one-hot vector for states along with 4 indicating the total amount of possible actions. The qVals represent the values in Q(s), while outAct, which is the highest value in the Q table, serves as the recommended policy (Qiao et al., 2018).

Did you know?

The emergence of Deep-Q Learning (DQN) marked a significant milestone in the field of AI. It showcased the remarkable potential of integrating reinforcement learning in deep neural networks by enabling AI to surpass human performance on Atari games.

Figure 3.6 implies that one gets focused on playing a single game at a time. This means that when inputting a state and receiving a policy recommendation, there is only one game being considered. In the typical approach to neural networks, this represents an entire batch size of a single one. As an illustration, an input state, inptSt, represents the numerical value of the current state that the actor is in. It can be inferred which oneH is a vector (Camci et al., 2022).

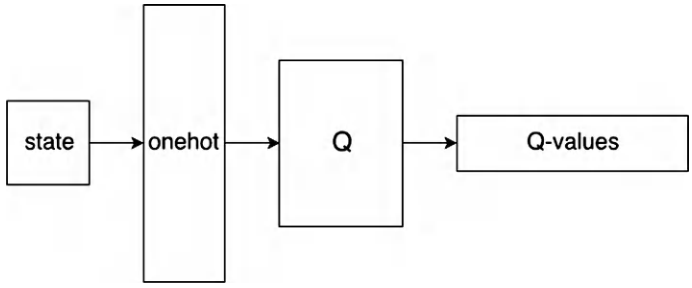


Figure 3.5. Illustration of Frozen-lake deep-Q-learning NN.

Source: Yoshua Bengio, Creative Commons License.

```
inptSt = tf.placeholder(dtype=tf.int32)
oneH=tf.one_hot(inptSt,16)
Q= tf.Variable(tf.random_uniform([16,4],0,0.01))
qVals= tf.matmul([oneH],Q)
outAct= tf.argmax(qVals,1)
```

Figure 3.6. Illustration of TF model parameters for the Q learning function.

Source: Pascal Vincent, Creative Commons License.

Next, as matmul function requires two matrices, one invokes it using [oneH]. Consequently, the matrix qVals will have a shape of [1, 4], containing solely the Q values for a single action (up, down, etc.). Finally, the shape of outAct is [1], indicating that the recommended action is outAct [0]. It is important to understand the level of detail one provides when presenting the remaining code in deep-Q learning (Ge et al., 2019) as shown in Figure 3.7.

It is worth noting that the algorithm makes a decision on which action to choose. This decision can either be random, particularly at the start of the learning process, or it can be based on the recommendation provided by the Q-table, especially as the learning process nears its conclusion. When utilizing deep-Q learning, one obtains the recommendation from the Q-table by inputting the present state s to the neural network depicted in Figure 3.5. Based on the highest value among the four possible actions (up, down, right, or left), one selects the corresponding action. After obtaining the action, one proceeds to execute it and observe the outcome, which is then used as a learning opportunity. Of course, in the realm of deep learning, a loss function is essential for this purpose (Gupta et al., 2019).

However, it is worth discussing the loss function used in deep-Q learning. This question is of utmost importance since, as one has observed throughout, during the initial stages of learning, and remain uncertain about the quality of our actions. Nevertheless, there are certain facts that one is aware of: On average, the estimation of Q (s, a) is more accurate when considering the state one arrives in after taking action in state s. This is because one looks one move ahead (Stember & Shalu, 2022).

$$R(s,a) + \gamma \max_{a'} Q(s',a') \tag{3.7}$$

Therefore, the loss is calculated as the squared difference between the observed outcome after taking a step and the predicted values obtained from the Q table or function. This is commonly referred to as the squared-error loss or quadratic loss (Ning et al., 2021).

$$(Q(s,a) - (R(s,a) + \gamma \max_{a'} Q(s',a')))^2, \tag{3.8}$$

The discrepancy between the Q value calculated from the network (the first term) and the Q value that one can determine by noticing the actual reward over the next action plus a Q value one step within the future (the second term) is referred to as the temporal difference error, or TD (0). If one were to consider the potential outcomes two steps ahead, one would arrive at TD (1) (Sumanas et al., 2022).

Continuing from the code in Figure 3.6, Figure 3.7 provides the remaining TF code. The subsequent lines construct the rest of the TensorFlow graph. Now look at the remaining code, paying particular attention to lines 7, 11,13, 14, 19, or 25. It applies the fundamental AI Gym "wandering" technique. That is, get aligned with the entire Figure 3.4. One generates the game (line 7) and engages in 2000 individual games (line

11), with each one commencing with the game (Galkin et al., 2021). Reset the function on line 13. There is a limit of 99 moves per episode (line 14). The move is executed in line 19. The game has concluded, as evidenced by the flag that one has designated as "dn" on line 25.

Two gaps need to be addressed, specifically lines 15–17 and 20–22. In the code, line 15 represents the forward pass. Here, the neural network is provided with its current state and returns a vector for length 1. The subsequent line then converts this vector into a scalar representing the action number. In line 18, one also incorporates a small probability for the program to choose a random action. By exploring the entire game space, one can guarantee comprehensive coverage. Lines 20–22 involve the computation of the loss and the subsequent backward pass to modify the model parameters. The purpose of lines 1–5 is to establish the TensorFlow graph for calculating and updating the loss (Jeong & Kim, 2019).

The program's performance falls short compared to tabular Q learning. However, as previously mentioned, tabular methods are well-suited for the frozen-lake MDP (Wu et al., 2018).

```
1  nextQ = tf.placeholder(shape=[1,4],dtype=tf.float32)
2  loss = tf.reduce_sum(tf.square(nextQ - qVals))
3  trainer = tf.train.GradientDescentOptimizer(learning_rate=0.1)
4  updateMod = trainer.minimize(loss)
5  init = tf.global_variables_initializer()

6  gamma = .99
7  game=gym.make('FrozenLake-v0')
8  rTot=0
9  with tf.Session() as sess:
10  sess.run(init)
11  for i in range(2000):
12    e = 50.0/(i + 50)
13    s=game.reset()
14    for j in range(99):
15     nActs,nxtQ=sess.run([outAct,qVals],feed_dict={inptSt: s})
16     nAct=nActs[0]
17     if np.random.rand(1)<e: nAct= game.action_space.sample()
19     s1,rwd,dn,_ = game.step(nAct)
20     Q1 = sess.run(qVals,feed_dict={inptSt: s1})
21     nxtQ[0,nAct] = rwd + gamma*(np.max(Q1))
22     sess.run(updateMod,feed_dict={inptSt:s, nextQ:nxtQ})
23     rTot+=rwd
24     if dn: break
25     s = s1
26 print "Percent games succesful: ", rTot/2000
```

Figure 3.7. Illustration of the remainder of deep-Q-learning code.

Source: Chris Colah, Creative Commons License.

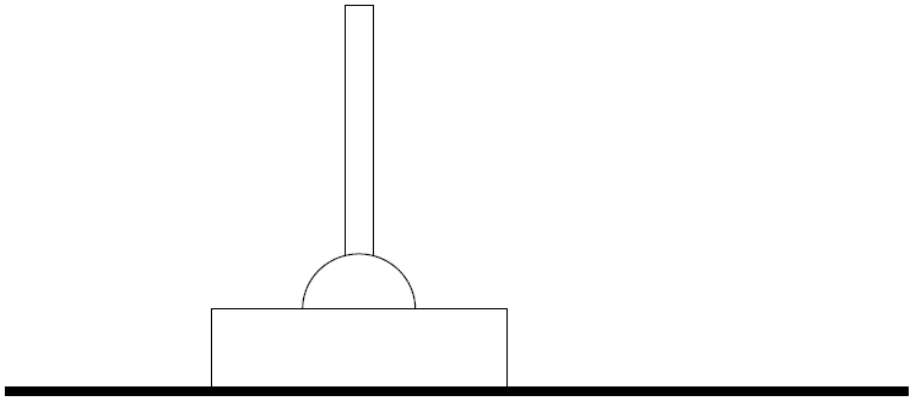


Figure 3.8. Illustration of a cart pole.

Source: Ian Goodfellow, Creative Commons License.

3.4. POLICY GRADIENT METHODS

Next, one explores a challenging problem in Open AI Gym that goes beyond the capabilities of traditional tabular methods. Specifically, the focus on the cart pole problem and introducing a cutting-edge technique in deep reinforcement learning called policy gradients. A "cart pole," depicted in Figure 3.8, consists of a cart positioned on a one-dimensional track. A state comprises four values: the cart's position and the pole's angle following the previous and current move. Values are assigned at successive time intervals to facilitate the program in determining the path of motion. The player can take two possible actions: Move the cart either to the right or towards the left. The magnitude of the impulse remains constant (Agarwal et al., 2021). If the cart moves too far to the right or left, or if the top of the pole moves too far from the perpendicular, the step signals that the current game is over, and one needs to reset it to start a new one. Each action one takes before reaching a point of failure results in a single unit of reward. Of course, the objective is to maintain optimal positioning of the cart or pole for as long as feasible. Given that the state is represented by a four-tuple containing actual numbers, it is important to note that the number of potential states is infinite. Consequently, tabular methods are not applicable in this scenario (Zhang et al., 2020).

Up until now, the neural network models have been employed to estimate a Q function for the Markov Decision Process (MDP). In this section, one presents a method that involves the direct modeling of the policy function by the neural network. Once more, our focus lies on model-free learning. It embraces the approach of exploring the gaming environment, initially making random choices for actions but eventually transitioning to relying on the recommendations of the neural network. Throughout this chapter, a significant challenge arises in determining a suitable loss function due to the lack of knowledge regarding the optimal actions to be taken (Ghosh et al., 2020).

When utilizing deep-Q learning, one takes a step forward with each move, relying on the understanding that by making a move, receiving a reward, and transitioning to a different state, the comprehension of the immediate surroundings becomes more refined. The loss was calculated by comparing the predicted outcome, based on previous knowledge (such as the Q function), with the actual outcome (Daskalakis et al., 2020).

Here one explores a unique approach. Now consider a scenario where one goes through a complete iteration of the game with no making any adjustments to our network. For instance, it makes a series of 20 moves (providing directions for the cart) before

the pole eventually tip over. In this case, exploration and exploitation are managed by selecting actions based on a probability distribution that is derived by the Q function, instead of simply choosing the action with the highest Q value (Liu et al., 2020).

In this particular scenario, the discounted reward to the initial state (D0(s, a)) can be calculated by considering all the subsequent states and actions that one has previously experimented with:

$$D_0(\mathbf{s}, \mathbf{a}) = \sum_{t=0}^{n-1} \gamma^t R(s_t, a_t, s_{t+1}) \tag{3.9}$$

After taking a certain number of steps, it becomes possible to calculate a future discounted reward of every state-action combination $\mathbf{s_i}$, $\mathbf{a_i}$ using a recurrence relation (Perdomo et al., 2021).

$$D_n(\mathbf{s}, \mathbf{a}) = 0 \tag{3.10}$$

$$D_i(\mathbf{s}, \mathbf{a}) = R(s_i, a_i, s_{i+1}) + \gamma D_{i+1}(\mathbf{s}, \mathbf{a}) \tag{3.11}$$

In the order of states one gets through, the reduced future reward for the fourth state (such as when implementing action, a) is represented as D4. Once again, it is important to acknowledge that it has acquired new knowledge in this instance. As an illustration, before attempting the initial random sequence of moves, one must understand the potential reward is completely unknown Later on, it becomes evident that a value of 10 is achievable (Hambly & Yang, 2021) and in fact, a sensible choice for a random sequence of actions). Alternatively, it has been established that in the event of a fall occurring on the move 10, the value of Q (s_9, a_9) = 0 is determined to be 0.

An effective loss function that encompasses these principles and numerous others is:

$$L(\mathbf{s}, \mathbf{a}) = \sum_{t=0}^{n-1} D_t(\mathbf{s}, \mathbf{a})(-\log \Pr(a_t \mid s))) \tag{3.12}$$

Let's break this down. It's important to highlight that the term on the right is the cross-entropy loss. On its own, it encourages the network to take action when it is in state s_t. Naturally, on its own, this holds little value as, especially during the initial stages of learning, one opts for random action selection (Matsubara et al., 2006).

Now let us explore the impact of the $\mathbf{D_t}$ values on this. Specifically, let us say that $\mathbf{a_0}$ was an unfavorable response to $\mathbf{s_0}$. As an illustration, let's consider a scenario where the cart is in the center or the pole is initially leaning towards the right. In this case, if one decides to move the cart to the left, it will cause the pole to lean even more towards the right (Ammar et al., 2014).

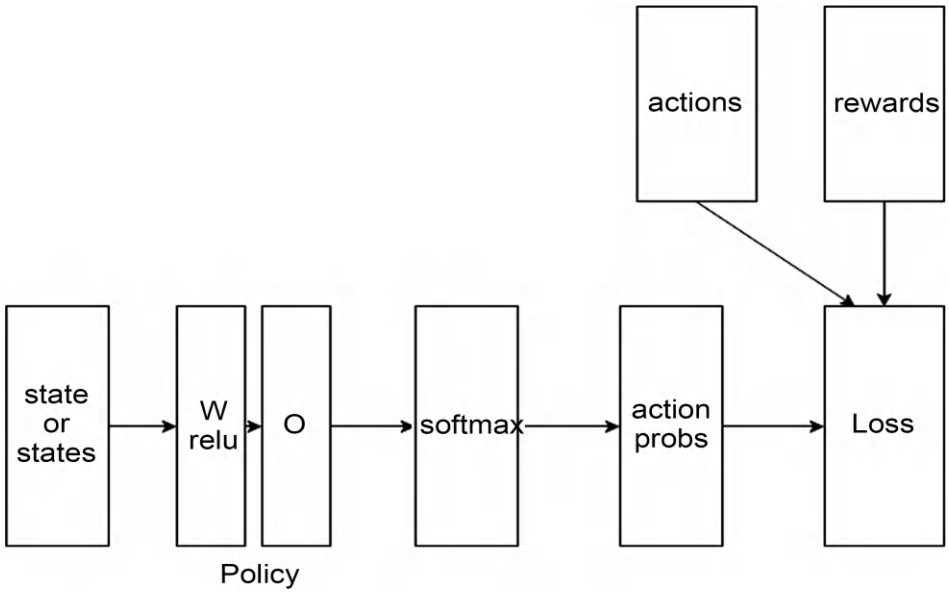


Figure 3.9. Illustration of deep learning architecture for REINFORCE.

Source: Aaron Courville, Creative Commons License.

Remember

Experience relive in DQN is a valuable technique that contributes to the stability of training. It involves the storage of past experiences and their random sampling during training. This process effectively reduces the correlation among consecutive updates, leading to improved training outcomes.

It is evident that, under the same conditions, the value for D0 is comparatively smaller in this scenario compared to a right movement choice. The rationale behind this is that, assuming all other factors remain constant, if the initial move is favorable, the pole or cart will stay within the boundaries for a longer duration (n is larger), resulting in larger D values. Equation 3.12 assigns a greater loss to a subpar a_0 compared to a superior one, effectively training the neural network to favor the latter (Russo, 2023).

The combination of this architecture and loss function is commonly referred to as REINFORCE. Figure 3.9 depicts the fundamental structure. It is worth noting that the neural network is utilized in two distinct manners. First, examining the left-hand side, one provides the neural network with a solitary state, which, as previously stated, consists of a four-tuple of real numbers representing the cart's position and velocity, as well as the pole's position and velocity. In this mode, one obtains probabilities for selecting the two available actions, as shown in the middle-right section of the figure. In this mode, the rewards and actions are not provided with values. This is because one does not know them and thus does not need them for computing loss on this point. Once all the moves to a complete game have been executed, the neural network is employed in a different mode (Gargiani et al., 2022). Now, provide an order of actions and rewards, and then one instructs it to calculate the loss and carry out backpropagation. During the training phase, it is essential to

calculate actions using two distinct methods. Initially, the neural network is provided with the states one encounters. Subsequently, the policy calculation layers calculate the probabilities of different actions for each state. Additionally, the actions taken are directly inputted as a placeholder. When making decisions in game-playing mode, one does not always select the action at the highest probability. Instead, of making random choices based on the action probabilities (Russo, 2023).

```
state= tf.placeholder(shape=[None,4],dtype=tf.float32)
W =tf.Variable(tf.random_uniform([4,8],dtype=tf.float32))
hidden= tf.nn.relu(tf.matmul(state,W))
O= tf.Variable(tf.random_uniform([8,2],dtype=tf.float32))
output= tf.nn.softmax(tf.matmul(hidden,O))

rewards = tf.placeholder(shape=[None],dtype=tf.float32)
actions = tf.placeholder(shape=[None],dtype=tf.int32)
indices = tf.range(0, tf.shape(output)[0]) * 2 + actions
actProbs = tf.gather(tf.reshape(output, [-1]), indices)
aloss = -tf.reduce_mean(tf.log(actProbs)*rewards)
trainOp= tf.train.AdamOptimizer(.01).minimize(aloss)
```

Figure 3.10. Illustration of TF graph instructions for cart-pole policy gradient NN.

Source: Xavier Glorot, Creative Commons License.

To calculate the loss based on Equation 3.12, one requires both components. Figure 3.10 provides the TensorFlow code for constructing the policy gradient neural network, utilizing the loss function described in Equation 3.12. Additionally, Figure 3.11 presents pseudocode outlining the process of utilizing the neural network to learn a policy and make decisions within the game environment. Let's begin by examining the pseudocode. It is worth noting that this outermost loop (line 2) involves playing 3001 sessions for the game. In the inner loop (line 2b), we engage in the game session until the step indicates completion (line D) or until we have made 999 moves. One selects a random action based on probabilities generated by our neural network (lines i, ii) or subsequently acts within the game. One stores the outcomes in the list to maintain a comprehensive record of the events. Once the action results in a final state, the model parameters are updated accordingly (Wierstra et al., 2010).

As depicted in Figure 3.10, the output is determined by processing the current-state values through a two-layer neural network. This network consists of linear units W and O, which are conveniently divided by a tf.relu activation function. The resulting values are then passed through a softmax function to convert the logits to probabilities. Recalling from previous applications of multilayer neural networks, the dimensions of the first layer are [input-size, hidden-size], or the dimensions of the second layer are [hidden-size, output-size]. In this case, one has selected a value of 8 for the hidden-size hyperparameter (Hambly et al., 2023).

As we have developed a novel loss function in this study, it did not rely on a standard one found in the TF library. Consequently, the computation of the loss had to

be constructed using fundamental TF functions, as depicted in the latter part of Figure 3.10. As an illustration, in all of our previous neural networks, the process of moving forward and backward was closely intertwined, with no computations from external TensorFlow sources being utilized (Ding et al., 2020).

1. totRs=[ ]
2. for i in range(3001):
   (a) st=reset game
   (b) for j in range(999):
      i. actDist = sess.run(output, feed_dict=state:[st])
      ii. select act randomly according to actDist
      iii. st1,r,dn,_=game.step(act)
      iv. collect st,a,r in hist
      v. st=st1
      vi. if dn:
         A. disRs = [D_i(states, actions from hst) | i = 0 to j - 1]
         B. create feed_dict with state=st, actions, from hist and rewards=disRs.
         C. sess.run(trainOp,feed_dict=feed_dict)
         D. add j to end of totRs
         E. break
      vii. if i%100=0: print out average of last 100 entries in totRs

Figure 3.11. Illustration of pseudocode for a policy-gradient-training NN for cart pole.

Source: John Glover, Creative Commons License.

$$\begin{matrix} \Pr(\mathbf{l} \mid s_1) & \Pr(\mathbf{r} \mid s_1) \\ \Pr(\mathbf{l} \mid s_2) & \Pr(\mathbf{r} \mid s_2) \\ \\ \Pr(\mathbf{l} \mid s_n) & \Pr(\mathbf{r} \mid s_n) \end{matrix} \quad \rightarrow \quad \begin{matrix} Pr(a_1 \mid s_1) \\ Pr(a_2 \mid s_2) \\ \\ Pr(a_n \mid s_n) \end{matrix}$$

Figure 3.12. Extracting action probabilities from the tensor of all probabilities.

Source: J□urgen Schmidhuber, Creative Commons License.

In this context, the values of reward are obtained from an external source. The reward serves as a temporary placeholder and is provided based on the instructions outlined in lines A, B, and C in Figure 3.11. Likewise, actions serve as a temporary substitute (Zhao et al., 2012).

The final three lines for Figure 3.10 present a more recognizable scenario, where the loss is simply calculated based on the quantities derived from Equation 3.12. One utilizes the Adam optimizer for our optimization process. The acquainted gradient-descent optimizer could have been utilized by simply substituting it in and increasing the learning rate. This would have resulted in achieving nearly comparable performance, although not quite at the same level. The Adam optimizer is a bit more intricate and

is widely regarded as superior. There are several notable distinctions between it and gradient descent, with the most significant one being the incorporation of momentum. An optimizer that incorporates momentum is designed to maintain the movement of a parameter value in the same direction if it has been consistently moving in that direction recently. This behavior is more pronounced compared to the standard gradient descent algorithm (Agarwal et al., 2020).

These two lines in Figure 3.10, which establish indices and actProbs, remain. Firstly, shift the focus away from the technical details and instead direct our attention towards the objectives at hand. It is essential to implement the transformation depicted in Figure 3.12. Displayed on the left is the result of a forward pass, which calculates the likelihood of each potential action (r and l) being the optimal choice (Lincoln et al., 2011).

To achieve this transformation, one relies on the gather function. This function takes two arguments and extracts each element of the tensor-based on the given numeric indices. It then combines these elements to create a new tensor.

tf.gather (tensor, indices)

As an illustration, given a tensor with values ((1,3), (4,6), (2,1), (3,3)), or a set of indices (3,1,3), the resulting output would be ((3,3), (4,6), (3,3)). In our case, one converts the action's probability matrix shown on the left for Figure 3.12 into a probability vector. It depends upon the line before it to assign indices to the appropriate list. This allows tf.gather to collect the probabilities of only the actions stated by the vector actions (Grondman et al., 2012).

It is beneficial to revisit and thoroughly examine the connection between Q-learning and REINFORCE. Firstly, there are variations in the methods used to gather information about the environment to provide input to the neural network. Q-learning takes a single step and then evaluates the accuracy of the neural network's prediction about the actual outcome. Upon reviewing Equation 3.8, which represents the Q-learning function for loss, it becomes evident that when the prediction or outcome aligns, no further updates are required (Fenjiro & Benbrahim, 2018). Using the REINFORCE method, one adopts a different approach. Instead of updating neural network parameters after each step and waiting until an entire episode is completed. An episode represents a full game, beginning with the initial state and continuing until the game indicates its completion. It is worth noting that an alternative approach could have been employed, such as utilizing the REINFORCE parameter alteration schedule instead of Q learning. This hinders the learning process by reducing the frequency of parameter changes. However, it is offset by the fact that it can make more effective changes as one calculates the actual discounted reward (Ishihara & Igarashi, 2006).

3.5. ACTOR-CRITIC METHODS

After examining the distinctions between Q learning and REINFORCE, the main focus now shifts to their commonalities. The neural network is responsible for computing in two ways a policy or, in the case of Q-learning, a function that can be easily utilized to generate a policy. This policy dictates that for any given state, the recommended action is to choose the action that maximizes the Q-value (Q(s, a)). In this section, one will explore programs that consist of two neural network subcomponents, each having its loss functions (Parisi et al., 2019).

The first subcomponent is an actor program, while the second subcomponent is a critic program. This particular type of RL is commonly referred to as the actor-critic method. In this section, one will explore the benefits of the actor-critic technique, commonly referred to as a2c. Choosing this option is advantageous for several reasons. Firstly, it has proven to be highly effective in practice. Additionally, one has the flexibility to adopt an incremental approach, beginning with the REINFORCE algorithm. The initial iteration is referred to as a2c. Once more, one put it to the test in the cart-pole game (Dutta & Upreti, 2022).

The technique is referred to as advantage actor-critic due to its utilization of the concept of "advantage." An important benefit of a state-action pair lies in the disparity between the Q value associated with the state-action pair and the value of the state itself:

$$A(s, a) = Q(s, a) - V(s) \tag{3.13}$$

It is commonly anticipated that the advantage would be a negative value due to the computation of V(s) in value iteration, where an arg $\mathbf{max}_{a}$ is performed over the available actions (Demir et al., 2022). Nevertheless, when it comes to favorable actions, A is significantly large in the realm of negative numbers, thereby serving as a metric to gauge the quality of an action within a specific state about the state as a whole (Zanette et al., 2021).

Next, the loss incurred by a2c from examining a sequence of acts to a start state towards the final stage of a game is defined as follows:

$$L_A(\mathbf{s}, \mathbf{a}) = \sum_{t=0}^{n-1} A(s_t, a_t)(-\log \Pr(a_t \mid s_t))) \tag{3.14}$$

The approach closely resembles the REINFORCE loss of Equation 3.12, with the only difference being the replacement of D_t(s, a), the discounted reward, with A_t(s, a) One has named the loss L_A to distinguish it from the overall loss for a2c, which, as shown below, includes a separate loss L_C related to the critic (Paschalidis et al., 2009).

It is important to note that the loss function of REINFORCE is designed to incentivize actions that result in greater rewards. Currently, there is a focus on promoting actions that outperform other potential actions in a given state. Although there is some merit to this argument, one must question why it would be superior to simply promoting high-reward actions directly (Nakamura et al., 2007).

The explanation lies in the variance of A (s, a). Variance of a function refers to the expected square difference between the function's value and its mean value. It can be understood that functions with significant variations exhibit high variance, in comparison to Q, A should demonstrate considerably lower variance (Rosenstein et al., 2004). Take a closer look at the cart pole. Given the game's responsiveness to our actions and the pole's movement, the distinction between moving left and moving right is minimal. Consequently, A remains small across the entire state space. On the other hand, let's compare this with Q. After learning about 100 games, the cart-pole game is only able to last for an average of 20 moves before failing. However, a policy that is considered moderately well can achieve 200 or more moves (Holzleitner et al., 2021).

When all other factors are held constant, it is simpler to estimate a function that has low variance compared to one that has high variance. The simplest function of all is one that remains constant with no variation. Therefore, if A is significantly more manageable to estimate, it has the potential to compensate for the drawback of maximizing A instead of Q directly. It appears this is the case. The computation of A remains uncertain at this stage. Now, let's move on to our next item of discussion (Peters et al., 2005).

Recalling, in the REINFORCE algorithm, one traverses a path according to our current policy until the end of a game. And then utilize the discounted reward D_t(s, a) via Equation 3.11 to estimate Q (s, a). Now, one can utilize this information to serve a dual purpose by using it as our estimation of Q while calculating A (Equation 3.13). Regarding V (s),

```
V1 =tf.Variable(tf.random_normal([4,8],dtype=tf.float32,stddev=.1))
v1Out= tf.nn.relu(tf.matmul(state,V1))
V2 =tf.Variable(tf.random_normal([8,1],dtype=tf.float32,stddev=.1))
v2Out= tf.matmul(v1Out,V2)
advantage = rewards-v2Out
aLoss = -tr.reduce_mean(tf.log(actProbs) * advantage)
cLoss=tf.reduce_mean(tf.square(rewards-vOut))
loss=aLoss + cLoss
```

Figure 3.13. Illustration of TF code added to Figures 3.10 and 3.11 for a2c build into our NN a sub network just to compute it.

Source: Arthur Juliani, Creative Commons License.

Figure 3.13 provides additional code for building the TF network, going beyond what is necessary for REINFORCE (Figure 3.10). One has developed two-layer fully connected neural networks, v10ut and v20ut, to calculate V, which represents the value function of the critic. The model is trained to generate accurate estimates of V by minimizing the difference between the actual rewards and the predictions made by the neural network (cLoss). The actor loss in this case is derived from Equation 3.14 and therefore incorporates the advantage function (Aslani et al., 2017). By making these modest adjustments, our REINFORCE algorithm is transformed into a2c–.

Advancing beyond a2c–, actual a2c introduces two additional enhancements. An issue with REINFORCE (and its derivative a2c–) is that learning only occurs after playing a complete game. In the early stages of cart-pole, where each game is relatively short, this doesn't pose much of a constraint (Veeriah et al., 2017). However, games utilizing REINFORCE tend to have a considerable length of a few hundred moves, while a2c– games tend to be even longer. A2c has the potential to enhance this aspect by updating the model's parameters at an earlier stage and with greater frequency (Nguyen et al., 2021).

A key strategy is to halt game execution at regular intervals, such as every 50 actions, to update the parameters of the model. Unfortunately, this was not achievable in REINFORCE. The purpose of observing a complete game's sequence of actions had been to obtain an accurate estimation of the Q values associated with the actions one as a species executed. However, a2c enables us to approximate the estimate by combining the total rewards obtained in the previous 50 moves with the V value for the final state. Next, one reset the list variable to its initial state at the 51st move and then repeat this process once more after 50 moves (Nam et al., 2021) (Pushed to the extreme, this may also free a2c from the constraint of REINFORCE's limitation to games in explicit game restarts.)

Another enhancement in full a2c involves utilizing multiple environments. It was observed from the beginning that conducting a batch of instruction examples provides an advantage by making better use of efficient matrix multiplication capabilities. When considering the computation of the next game action, it is not feasible to play only one game at a time. Engaging in multiple games is akin to consolidating examples in this context (Su et al., 2021).

ACTIVITY 3.1.

Objective

To understand the key concepts and relationships in Deep-Q Learning (DQN).

Materials Needed

- Large paper or whiteboard
- Markers or pens
- Post-it notes or index cards

Steps

1. Identify and Arrange Key Concepts:
   - Write down terms related to DQN (e.g., Deep Reinforcement Learning, Q-learning, Neural Networks, Experience Replay, Target Networks, Bellman Equation, and Exploration vs. Exploitation, Epsilon-Greedy Strategy, State, Action, Reward, and Policy) on post-it notes or index cards.
   - Arrange these terms on a large sheet of paper or whiteboard.
2. **Connect and Discuss**:
   - Draw lines to connect related concepts, annotating the connections.
   - Present and discuss the concept map with the class, explaining the relationships and receiving feedback.

SUMMARY

- The chapter explores reinforcement learning (RL) methodologies starting with Markov Decision Processes (MDPs) and value iteration, where an agent learns optimal policies by iteratively updating expected rewards for state-action pairs. It contrasts this with Q-learning, a model-free approach that allows agents to learn directly from interactions with an environment, balancing exploration and exploitation through strategies like epsilon-greedy methods.
- The discussion advances into deep-Q learning, utilizing neural networks to approximate the Q-function, which is essential for handling complex environments such as Atari games where tabular methods are impractical due to large state spaces.
- Policy gradient methods, such as REINFORCE, use neural networks to directly learn optimal policies in reinforcement learning. They eschew traditional Q-tables, making them suitable for continuous state spaces like the cart-pole problem.
- Actor-Critic methods like A2C enhance learning efficiency by combining actor networks for policy determination with critic networks for state-action value estimation. This approach, leveraging advantages to quantify action effectiveness, improves stability and learning speed compared to pure policy gradients.

REVIEW QUESTIONS

1. What is Deep Reinforcement Learning, and how does it differ from traditional reinforcement learning?
2. Explain the concept of Value Iteration and describe its process in reinforcement learning.
3. What is Q-learning, and how does it function without a model of the environment?
4. Describe the Basic Deep-Q Learning (DQN) algorithm and its key techniques for stabilizing training.
5. What are Policy Gradient Methods, and why are they useful for high-dimensional action spaces and continuous control problems?
6. `Explain the Actor-Critic Methods and the roles of the actor and critic in these methods. How do they work together to improve policy learning?

MULTIPLE CHOICE QUESTIONS

1. Which of the following best describes Deep Reinforcement Learning?

a. Combining reinforcement learning with deep learning.

b. Combining supervised learning with clustering techniques.

c. Using neural networks to approximate supervised learning tasks.

d. Implementing unsupervised learning in high-dimensional spaces.

2. **What is the primary objective of Value Iteration in reinforcement learning?**
   a. To directly optimize the policy by gradient ascent.
   b. To compute optimal policies by iteratively updating value functions.
   c. To approximate Q-functions using neural networks.
   4. To evaluate the current policy using Monte Carlo simulations.
3. **In Q-learning, what is the purpose of the Q-function?**
   a. To estimate the value of taking a specific action in a given state.
   b. To model the transition probabilities of the environment.
   c. To represent the policy directly.
   d. To calculate the gradient of the reward function.
4. **Which technique is employed in Basic Deep-Q Learning (DQN) to stabilize training?**
   a. Temporal Difference Learning
   b. Policy Gradient
   c. Experience Replay and Target Networks
   d. Monte Carlo Methods
5. **`Policy Gradient Methods are particularly suited for which type of problems?**
   a. Problems with discrete action spaces.
   b. Supervised learning problems.
   c. High-dimensional action spaces and continuous control problems.
   d. Clustering and classification problems.
6. **What is the role of the 'critic' in Actor-Critic Methods?**
   a. To directly optimize the policy.
   b. To estimate value functions and evaluate actions taken by the actor.
   c. To store and replay experiences for training.
   d. To generate random actions for exploration.

Answers to Multiple Questions

1. (a); 2. (b); 3. (a); 4. (c); 5. (c); 6. (b).

REFERENCES

1. Agarwal, A., Henaff, M., Kakade, S., & Sun, W. (2020). Pc-pg: Policy covers directed exploration for provable policy gradient learning. Advances in Neural Information Processing Systems, 33, 13399–13412.
2. Agarwal, A., Kakade, S. M., Lee, J. D., & Mahajan, G. (2021). On the theory of policy gradient methods: Optimality, approximation, and distribution shift. Journal of Machine Learning Research, 22(98), 1–76.

3. Ahmadabadi, M. N., & Asadpour, M. (2002). Expertness based cooperative Q-learning. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 32(1), 66–76.

4. Ammar, H. B., Eaton, E., Ruvolo, P., & Taylor, M. (2014). Online multi-task learning for policy gradient methods. In International Conference on Machine Learning, 23(3), 1206–1214.

5. Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). Deep reinforcement learning: A brief survey. IEEE Signal Processing Magazine, 34(6), 26–38.

6. Aslani, M., Mesgari, M. S., & Wiering, M. (2017). Adaptive traffic signal control with actor-critic methods in a real-world traffic network with different traffic disruption events. Transportation Research Part C: Emerging Technologies, 85, 732–752.

7. Camci, E., Gupta, M., Wu, M., & Lin, J. (2022). Qlp: Deep Q-learning for pruning deep neural networks. IEEE Transactions on Circuits and Systems for Video Technology, 32(10), 6488–6501.

8. Chen, R. R., & Meyn, S. (1999). Value iteration and optimization of multiclass queueing networks. Queueing Systems, 32, 65–97.

9. Chen, S. A., Tangkaratt, V., Lin, H. T., & Sugiyama, M. (2020). Active deep Q-learning with demonstration. Machine Learning, 109(9), 1699–1725.

10. Clifton, J., & Laber, E. (2020). Q-learning: Theory and applications. Annual Review of Statistics and Its Application, 7(1), 279–301.

11. Dai, P., Weld, D. S., & Goldsmith, J. (2011). Topological value iteration algorithms. Journal of Artificial Intelligence Research, 42, 181–209.

12. Daskalakis, C., Foster, D. J., & Golowich, N. (2020). Independent policy gradient methods for competitive reinforcement learning. Advances in Neural Information Processing Systems, 33, 5527–5540.

13. Dayan, P., & Watkins, C. J. C. H. (1992). Q-learning. Machine Learning, 8(3), 279–292.

14. Demir, S., Stappers, B., Kok, K., & Paterakis, N. G. (2022). Statistical arbitrage trading on the intraday market using the asynchronous advantage actor–Critic method. Applied Energy, 314, 118912.

15. Ding, D., Zhang, K., Basar, T., & Jovanovic, M. (2020). Natural policy gradient primal-dual method for constrained markov decision processes. Advances in Neural Information Processing Systems, 33, 8378–8390.

16. Dutta, D., & Upreti, S. R. (2022). A survey and comparative evaluation of actor-critic methods in process control. The Canadian Journal of Chemical Engineering, 100(9), 2028–2056.

17. Fenjiro, Y., & Benbrahim, H. (2018). Deep reinforcement learning overview of the state of the art. Journal of Automation Mobile Robotics and Intelligent Systems, 12(3), 20–39.

18. François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., & Pineau, J. (2018). An introduction to deep reinforcement learning. Foundations and Trends® in Machine Learning, 11(3–4), 219–354.

19. Galkin, B., Fonseca, E., Amer, R., DaSilva, L. A., & Dusparic, I. (2021). REQIBA: Regression and deep Q-learning for intelligent UAV cellular user to base station association. IEEE Transactions on Vehicular Technology, 71(1), 5–20.

20. Gargiani, M., Zanelli, A., Martinelli, A., Summers, T., & Lygeros, J. (2022). PAGE-PG: A simple and loopless variance-reduced policy gradient method with probabilistic gradient estimation. In International Conference on Machine Learning, 33(12), 7223–7240.

21. Ge, H., Song, Y., Wu, C., Ren, J., & Tan, G. (2019). Cooperative deep Q-learning with Q-value transfer for multi-intersection signal control. IEEE Access, 7, 40797–40809.

22. Ghosh, D., Machado, M. C., & Le Roux, N. (2020). An operator views of policy gradient methods. Advances in Neural Information Processing Systems, 33, 3397–3406.

23. Goldberg, Y., & Kosorok, M. R. (2012). Q-learning with censored data. Annals of Statistics, 40(1), 529.

24. Goyal, V., & Grand-Clement, J. (2023). A first-order approach to accelerated value iteration. Operations Research, 71(2), 517–535.

25. Gronauer, S., & Diepold, K. (2022). Multi-agent deep reinforcement learning: A survey. Artificial Intelligence Review, 55(2), 895–943.

26. Grondman, I., Busoniu, L., Lopes, G. A., & Babuska, R. (2012). A survey of actor-critic reinforcement learning: Standard and natural policy gradients. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 42(6), 1291–1307.

27. Guo, M., Liu, Y., & Malec, J. (2004). A new Q-learning algorithm based on the Metropolis criterion. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 34(5), 2140–2143.

28. Gupta, U., Mandal, S. K., Mao, M., Chakrabarti, C., & Ogras, U. Y. (2019). A deep Q-learning approach for dynamic management of heterogeneous processors. IEEE Computer Architecture Letters, 18(1), 14–17.

29. Hambly, B., Xu, R., & Yang, H. (2021). Policy gradient methods for the noisy linear quadratic regulator over a finite horizon. SIAM Journal on Control and Optimization, 59(5), 3359–3391.

30. Hambly, B., Xu, R., & Yang, H. (2023). Policy gradient methods find the Nash equilibrium in n-player general-sum linear-quadratic games. Journal of Machine Learning Research, 24(139), 1–56.

31. Holzleitner, M., Gruber, L., Arjona-Medina, J., Brandstetter, J., & Hochreiter, S. (2021). Convergence proof for actor-critic methods applied to PPO and RUDDER. In Transactions on Large-Scale Data-and Knowledge-Centered Systems XLVIII: Special Issue in Memory of Univ. Prof. Dr. Roland Wagner, 4(2), 105–130.

32. Ishihara, S., & Igarashi, H. (2006). Applying the policy gradient method to behavior learning in multiagent systems: The pursuit problem. Systems and Computers in Japan, 37(10), 101–109.

33. Jang, B., Kim, M., Harerimana, G., & Kim, J. W. (2019). Q-learning algorithms: A

comprehensive classification and applications. IEEE Access, 7, 133653–133667.

34. Jeong, G., & Kim, H. Y. (2019). Improving financial trading decisions using deep Q-learning: Predicting the number of shares, action strategies, and transfer learning. Expert Systems with Applications, 117, 125–138.
35. Kumar, A., Zhou, A., Tucker, G., & Levine, S. (2020). Conservative Q-learning for offline reinforcement learning. Advances in Neural Information Processing Systems, 33, 1179–1191.
36. Kumar, N., Kumar, P., Sandeep, C., Thirupathi, V., & Shwetha, S. (2019). A study on deep Q-learning and single stream Q-network architecture. International Journal of Advanced Science and Technology, 28(20), 586–592.
37. Ladosz, P., Weng, L., Kim, M., & Oh, H. (2022). Exploration in deep reinforcement learning: A survey. Information Fusion, 85, 1–22.
38. Lincoln, R., Galloway, S., Stephen, B., & Burt, G. (2011). Comparing policy gradient and value function-based reinforcement learning methods in simulated electrical power trade. IEEE Transactions on Power Systems, 27(1), 373–380.
39. Liu, L., & Hodgins, J. (2017). Learning to schedule control fragments for physics-based characters using deep Q-learning. ACM Transactions on Graphics (TOG), 36(3), 1–14.
40. Liu, Y., Zhang, K., Basar, T., & Yin, W. (2020). An improved analysis of (variance-reduced) policy gradient and natural policy gradient methods. Advances in Neural Information Processing Systems, 33, 7624–7636.
41. Luo, B., Yang, Y., Wu, H. N., & Huang, T. (2019). Balancing value iteration and policy iteration for discrete-time control. IEEE Transactions on Systems, Man, and Cybernetics: Systems, 50(11), 3948–3958.
42. Mann, T. A., Mannor, S., & Precup, D. (2015). Approximate value iteration with temporally extended actions. Journal of Artificial Intelligence Research, 53, 375–438.
43. Matsubara, T., Morimoto, J., Nakanishi, J., Sato, M. A., & Doya, K. (2006). Learning CPG-based biped locomotion with a policy gradient method. Robotics and Autonomous Systems, 54(11), 911–920.
44. Millán, J. D. R., Posenato, D., & Dedieu, E. (2002). Continuous-action Q-learning. Machine Learning, 49, 247–265.
45. Mousavi, S. S., Schukat, M., & Howley, E. (2018). Deep reinforcement learning: An overview. In Proceedings of SAI Intelligent Systems Conference (IntelliSys) 2016, 2(1), 426–440.
46. Nakamura, Y., Mori, T., Sato, M. A., & Ishii, S. (2007). Reinforcement learning for a biped robot based on a CPG-actor-critic method. Neural Networks, 20(6), 723–735.
47. Nam, D. W., Kim, Y., & Park, C. Y. (2021). GMAC: A distributional perspective on actor-critic framework. In International Conference on Machine Learning, 10, 7927–7936.
48. Nguyen, N. D., Nguyen, T. T., Vamplew, P., Dazeley, R., & Nahavandi, S. (2021). A prioritized objective actor-critic method for deep reinforcement learning. Neural Computing and Applications, 33, 10335–10349.

49. Ning, B., Lin, F. H. T., & Jaimungal, S. (2021). Double deep Q-learning for optimal execution. Applied Mathematical Finance, 28(4), 361–380.

50. Parisi, S., Tangkaratt, V., Peters, J., & Khan, M. E. (2019). TD-regularized actor-critic methods. Machine Learning, 108, 1467–1501.

51. Paschalidis, I. C., Li, K., & Estanjini, R. M. (2009). An actor-critic method using least squares temporal difference learning. In Proceedings of the 48th IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference, 9(5), 2564–2569.

52. Perdomo, J., Umenberger, J., & Simchowitz, M. (2021). Stabilizing dynamical systems via policy gradient methods. Advances in Neural Information Processing Systems, 34, 29274–29286.

53. Peters, J., Vijayakumar, S., & Schaal, S. (2005). Natural actor-critic. In Machine Learning: ECML 2005: 16th European Conference on Machine Learning, Porto, Portugal, October 3–7, 2005. Proceedings, 16, 280–291.

54. Qiao, J., Wang, G., Li, W., & Chen, M. (2018). An adaptive deep Q-learning strategy for handwritten digit recognition. Neural Networks, 107, 61–71.

55. Ramaswamy, A., & Hüllermeier, E. (2021). Deep Q-learning: Theoretical insights from an asymptotic analysis. IEEE Transactions on Artificial Intelligence, 3(2), 139–151.

56. Rosenstein, M. T., Barto, A. G., Si, J., Barto, A., Powell, W., & Wunsch, D. (2004). Supervised actor-critic reinforcement learning. In Learning and Approximate Dynamic Programming: Scaling Up to the Real World (pp. 359–380).

57. Russo, D. (2023). Approximation benefits of policy gradient methods with aggregated states. Management Science, 69(11), 6898–6911.

58. Spaan, M. T., & Vlassis, N. (2005). Perseus: Randomized point-based value iteration for POMDPs. Journal of Artificial Intelligence Research, 24, 195–220.

59. Stember, J. N., & Shalu, H. (2022). Reinforcement learning using deep Q-networks and Q-learning accurately localizes brain tumors on MRI with very small training sets. BMC Medical Imaging, 22(1), 224.

60. Su, J., Adams, S., & Beling, P. (2021). Value-decomposition multi-agent actor-critics. In Proceedings of the AAAI Conference on Artificial Intelligence, 35(13), 11352–11360.

61. Sumanas, M., Petronis, A., Bucinskas, V., Dzedzickis, A., Virzonis, D., & Morkvenaite-Vilkonciene, I. (2022). Deep Q-learning in robotics: Improvement of accuracy and repeatability. Sensors, 22(10), 3911.

62. Tan, C., Han, R., Ye, R., & Chen, K. (2020). Adaptive learning recommendation strategy based on deep Q-learning. Applied Psychological Measurement, 44(4), 251–266.

63. Veeriah, V., van Seijen, H., & Sutton, R. S. (2017). Forward actor-critic for nonlinear function approximation in reinforcement learning. In Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, 8(4), 556–564.

64. Wang, X., Wang, S., Liang, X., Zhao, D., Huang, J., Xu, X., & Miao, Q. (2022). Deep reinforcement learning: A survey. IEEE Transactions on Neural Networks and Learning Systems, 35(4), 5064–5078.

65. Watkins, C. J., & Dayan, P. (1992). Q-learning. Machine Learning, 8, 279–292.

66. Wei, Q., Liu, D., & Lin, H. (2015). Value iteration adaptive dynamic programming for optimal control of discrete-time nonlinear systems. IEEE Transactions on Cybernetics, 46(3), 840–853.

67. Wierstra, D., Förster, A., Peters, J., & Schmidhuber, J. (2010). Recurrent policy gradients. Logic Journal of IGPL, 18(5), 620–634.

68. Wu, J., He, H., Peng, J., Li, Y., & Li, Z. (2018). Continuous reinforcement learning of energy management with deep Q network for a power split hybrid electric bus. Applied Energy, 222, 799–811.

69. Zanette, A., Wainwright, M. J., & Brunskill, E. (2021). Provable benefits of actor-critic methods for offline reinforcement learning. Advances in Neural Information Processing Systems, 34, 13626–13640.

70. Zhang, J., Koppel, A., Bedi, A. S., Szepesvari, C., & Wang, M. (2020). Variational policy gradient method for reinforcement learning with general utilities. Advances in Neural Information Processing Systems, 33, 4572–4583.

71. Zhang, Q., Lin, M., Yang, L. T., Chen, Z., Khan, S. U., & Li, P. (2018). A double deep Q-learning model for energy-efficient edge scheduling. IEEE Transactions on Services Computing, 12(5), 739–749.

72. Zhao, T., Hachiya, H., Niu, G., & Sugiyama, M. (2012). Analysis and improvement of policy gradient estimation. Neural Networks, 26, 118–129.

CHAPTER

4 Convolutional Neural Networks

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

- Understand the role of filters, strides, and padding in convolutional neural networks.
- Understand convolutional filters and their application in image processing.
- Illustrate the role of convolutional filters in feature extraction from images.
- Understand the multilevel convolution concept.
- Implement a two-layer convolutional neural network model in TensorFlow for enhanced image recognition accuracy.
- Understand bias addition and pooling techniques to enhance convolutional neural network performance in TensorFlow.

KEY TERMS FROM THIS CHAPTER

Convolution
Filters
Image processing
Max pooling
Neural network (NN)
Padding
Strides
TensorFlow (TF)

UNIT INTRODUCTION

All of the neural networks that have been discussed thus far have been fully connected. In other words, the linear units at each layer are interconnected with the linear units at the subsequent layer. Nevertheless, there is no obligation for neural networks to adhere to this specific structure. It is possible to develop a scenario in which a linear unit selectively transmits its output to a subset of the units in the subsequent layer. It is slightly more challenging, but not excessively difficult, to observe that training can accurately calculate the weight derivatives during the backward pass when it knows the connected units (Li et al., 2021). Convolutional neural networks are a specific type of partially connected neural networks. Continuing our discussion on the Mnist data set, one delves into the practical applications of Convolutional Neural Networks in the field of computer vision (Gu et al., 2018).

After an inspection of the dataset, it can be seen particular light intensities at particular positions in the image with specific digits. Therefore, it is possible to establish a correlation between the number 1 and the position (8, 14) by assigning high values. However, this is not how individuals typically operate. Taking pictures of each digit in a bright room may increase the pixel values by 10, but it is unlikely to significantly impact the categorization. In scene recognition, the focus lies on the disparities in pixel values rather than their exact values. Also, the distinctions hold significance solely among values that are near each other (Yamashita et al., 2018). Imagine yourself in a compact space, illuminated by a solitary light bulb positioned in one corner. The perception of a light patch in a distant wallpaper may involve the reflection of fewer photons compared to a nearby "dark" patch close to the bulb. The key factor in understanding the dynamics of a scene lies in the analysis of local variations in light intensity, particularly focusing on the concept of "local" and "differences." Computer vision investigators are well aware of this and have widely embraced convolutional techniques as the standard responses to such facts (Krichen, 2023).

$$\begin{matrix} 1.0 & 1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 & 1.0 \\ -1.0 & -1.0 & -1.0 & -1.0 \\ -1.0 & -1.0 & -1.0 & -1.0 \end{matrix}$$

Figure 4.1. A basic filter for detecting horizontal lines.

Source: Tomas Mikolov, Creative Commons License.

This unit delves into Convolutional Neural Networks (CNNs), examining essential concepts like filters, strides, and pooling. These concepts are vital for comprehending how CNNs identify and interpret trends in images. In addition, the unit examines the concepts of multilevel convolution, TF convolution, and the architecture of CNNs. It also delves into the wide range of applications of CNNs, from recognition of images to medical imaging (Nebauer, 1998).

4.1. FILTERS, STRIDES, AND PADDING

A convolutional filter, often referred to as the convolutional kernel, is a small array of numbers. When analyzing a black-and-white image, it can be represented as a two-dimensional array. The Mnist dataset comprises black-and-white images, which simplifies our needs. To include color, it would be essential to use an array using three dimensions and three two-dimensional arrays representing the red, blue, and green (RGB) wavelengths of light. This arrangement allows for the reconstruction of all colors. Currently, the disregard of complexities associated with color is ignored. One will revisit them at a later point (Naseri & Mehrdad, 2023).

Examine the convolution filter shown in Figure 4.1. When applying the filter to a section of an image, the product of dots is calculated from the filter and a corresponding portion of the picture that has the same dimensions. It is important to keep in mind that when calculating the dot product for two vectors, the corresponding elements are multiplied together and then summed to obtain a single value. In this context, one extends this concept to arrays with two or more dimensions. The process involves multiplying every element of the arrays and subsequently summing all the products (Hashemi, 2019).

In a more formal context, the convolution kernel is regarded as a function, often known as a kernel function. The value of V for that function is determined by evaluating it at the coordinates (x, y) on image I:

$$V(x,y) = (I \cdot K)(x,y) = \sum_m \sum_n I(x+m, y+n) K(m,n) \tag{4.1}$$

In a formal sense, convolution involves the operation of two functions, I or K, resulting in a third function that executes the operation on the right side (Liu et al., 2022).

0.0	0.0	0.0	0.0	0.0	0.0
0.0	2.0	2.0	2..0	0.0	0.0
0.0	2.0	2.0	2..0	0.0	0.0
0.0	2.0	2.0	2..0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0

Figure 4.2. Illustration of Image of a small square.

Source: Suriyadeepan Ram, Creative Commons License.

For typical purposes, one can bypass the official definition and proceed directly to the operations on the right-hand side. In typical scenarios, the points x, and y will be considered to be located in or close to the center of the patch being analyzed (Lee et al., 2018). Thus, in the case of the previously given 4 * 4 kernel, the two m and n can vary from -2 to +1. If we utilize the filter displayed in Figure 4.1 to analyze a particular portion of the square image illustrated in Figure 4.2. The lower two rows for this filter are positioned over the zeros in the image, while the uppermost four elements in this filter are positioned over the 2.0s in the square. Consequently, the filter's value in this patch is number 6. It is important to mention that in the case where all the pixels have a value of zero, the resulting filter value would also be zero. However, even if every single patch consisted of only the number 10, the resulting value remains a zero. This filter prioritizes patches to a horizontal line running by the middle, with higher values upon the top and fewer ones below. It is worth noting that filters can be designed to detect variations in light intensities instead of focusing on their exact values. In addition, filters typically concentrate on local changes as they are generally smaller than entire images. A filter kernel can be designed to prioritize picture patches containing straight lines in different directions, like those from upper left to lower right (Kamath et al., 2019).

Did you know?

CNNs draw inspiration from the intricate structure of the animal visual cortex. The hierarchical arrangement of CNNs, alongside layers detecting fundamental traits like edges and textures advancing to more intricate features, mirrors the way visual data is processed in the brain.

In the previous discussion, one has described the filter in the way that it was intentionally created by the programmer to identify specific features in the image. This approach was commonly used before the emergence of deep convolutional neural network filtering. One notable characteristic of deep-learning strategies is the way the filter values serve as parameters for the neural network, which are acquired through the backward pass. In our ongoing conversation about the operation of convolution, one will conveniently overlook this aspect and proceed with the presentation of our pre-designed filters till the next section (Salomon et al., 2017). Furthermore, the process involves the application of a filter not only to the image patch but also to the entire image. Applying the filter to multiple patches in the image is a vital step in the process. Typically, a wide range of filters are employed, each to detect a particular characteristic within the image (Reinel et al., 2021). Upon finishing, the feature values can be passed to any number of fully connected layers, followed by softmax and ultimately the loss function. The architecture is depicted in Figure 4.3. In our representation, a convolution filter layer is depicted as a three-dimensional box. This is due to a bank of filters can be visualized to be a three-dimensional tensor, with dimensions of height, width, and the number of different filters (Romanuke, 2018).

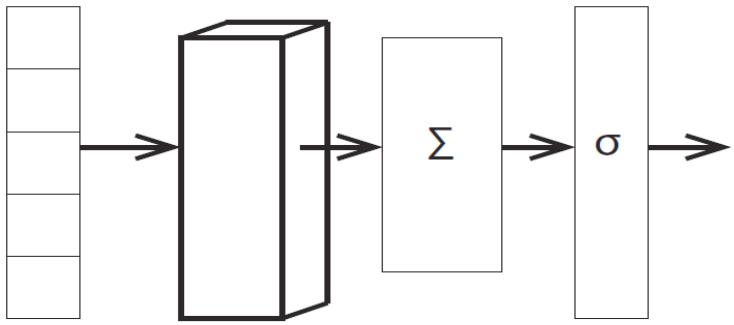


Figure 4.3. Illustration of An image recognition architecture with convolution filters.

Source: David E Rumelhart, Creative Commons License.

Observe the intentional lack of specificity in the previous statement about the convolution for a filter with a "large number" of patches within an image. To improve clarity, first establish the concept of stride, which refers to the distance between two instances of applying a filter (Dorj et al., 2018). Applying a filter at each extra pixel can be achieved by using a stride of two. To further specify, one refers to both the horizontal stride s_h or the vertical stride s_v. As one traverses the image, the filter is applied at regular intervals of s_h pixels. Upon reaching the end of a line, one proceeds to descend vertically by s_v lines and then repeats the procedure. Even when a stride for two is used, the filter is still applied to all pixels within the region, rather than just every other one. The stride only impacts the location of the filter's next application (Chou et al., 2019).

Now, establish the understanding of the "end of a line" when utilizing a filter. Padding on the convolution is specified to achieve this. TensorFlow provides two options for padding: Valid and Same. Once the filters have been convolved with a specific patch of the image, one proceeds to shift s_h to the right (Batchuluun et al., 2018).

0	1		23	24	25	26	27
.	.	3.2	3.1	2.5	2..0	0	0
.	.	3.2	3.1	2.5	2.0	0	0
.	.	3.2	3.1	2.5	2.0	0	0
.	.	3.2	3.1	2.5	2..0	0	0
.	.	3.2	3.1	2.5	2.0	0	0

Figure 4.4. Illustration of End of line with valid and same padding.

Source: Eugene Charnak, Creative Commons License.

There are three possible scenarios: (a) when one is not near the image boundary, they can continue working upon this line, (b) when the leftmost pixel of the following convolution patch grows beyond the image edge, or (c) if a leftmost pixel which the filters look into is in the rightmost part of the image, yet the pixel is over the end of the image. In case (b), the process of same padding comes to a halt, while in case (c), it is the process of Valid that stops (Adem, 2022).

As shown in Figure 4.4, the image width is 28 pixels, the filter dimensions are 4 pixels wide by 2 pixels high, and the stride value is set to 1. When using valid padding, the process stops at pixel 24, following zero-based counting. This occurs because the stride would lead to pixel 25 or accommodating a filter with a width of 4 would necessitate a 29th pixel that is not present. The convolution would continue until reaching pixel 27, using the same padding. It is common practice to make the same choice within the vertical direction once one gets to the bottom of the image (Pang et al., 2020).

The choice of when to stop is referred to as padding, as it involves the utilization of "imaginary" pixels when moving horizontally with the same padding. The left side of a filter falls for the image boundary, while the right side does not. Within TensorFlow, the imaginary pixels are at a value of zero (Traore et al., 2018). When using the Same padding, it becomes necessary to extend the image's boundaries with fictitious pixels. In most cases, actual padding is not necessary when using valid padding. This occurs because one terminates the process of convolving before any portion of the filter shifts adjacent to the edge of the image. When padding is required, it is distributed evenly over all edges (Prusa & Khoshgoftaar, 2017).

Due to the significance of this data for future utilization, an individual supplies the count of patch convolutions used in a horizontal direction with identical padding:

$$\lceil i_h / s_h \rceil \tag{4.2}$$

The ratio x in Equation (4.2) is rounded up to the nearest integer. The function returns a small integer that is ≥ x. To understand the necessity of the ceiling function, consider a situation in which the width of the image is an odd number of pixels, like five, and the stride is two. Initially, the filter is applied to the patch 0–2 in the horizontal direction (Pang et al., 2017). After that, it shifts to the right by two positions and is utilized in the range of 2–4. Once position 4 is reached, it needs to be applied to 4–6. Given the width of 5, it is important to note that there is not a position 6. However, when using the same padding, an additional zero is added after a line to ensure that the filter can process positions 4-6. As a result, the total number of applications increases to 4. Without the addition of extra zeros, the equation mentioned would utilize the floor function instead of the ceiling (Kubanek et al., 2019). Similarly, the vertical direction follows the same logic, resulting in i_v / s_v.

The integer is horizontally aligned for valid padding

$$\lfloor (i_h - f_h + 1) / s_h \rfloor \tag{4.3}$$

If the final equation is not identified, ensure that one understands that $i_h - f_h$ is the frequency at which one can shift (without exceeding the available space) when the stride is set to one. The number of applications is equal to the sum of one and the number of shifts (Pamula, 2018).

Although it utilizes fictional pixels, the same padding is popular due to its ability to maintain the same output size as the original image if combined with a stride of one. It is common for one to merge multiple layers of convolution, with each output serving as the input for the subsequent layer. Regardless of the length of the stride, appropriate padding always results in an output that is less than the input. Through successive convolution layers, the outcome gradually diminishes starting from the outermost regions (Zhang et al., 2018).

Before delving into the implementation, it is essential to explore the impact of convolution on image representation. The core component of a convolutional neural network in TensorFlow is the two-dimensional convolution function, together with some additional optional named arguments that one will disregard for now (Fotouhi et al., 2021).

tf.nn.conv2d(input, filters, strides, padding)

The "2d" in the name indicates that the process uses convolving an image. (Also, there are variations of convolutional neural networks that are designed to process one-dimensional items, such as audio signals, or even three-dimensional objects like video clips.)

As one would anticipate, the initial parameter pertains to the batch size for each of the images. Until now, the concept of an individual picture has been seen as a 2D array of numbers, with each number representing a specific light intensity. Considering the inclusion of the batch size, it is important to note that the input takes the form of a three-dimensional tensor (Shalbaf et al., 2020).

However, the tt.nn.conv2d function requires that every image be expressed as a three-dimensional object, where the last dimension is a vector representing the channels. As mentioned before, standard color images are composed of three channels: green, blue, and red. When analyzing images, it is crucial to understand that they are depicted as a two-dimensional array of pixels, where each pixel has a set of intensity values. The list includes a solitary value to represent black-and-white photographs, however, there are three values for colored images (Hossain & Sajib, 2019).

(1, –1, –1)	(1, –1, –1)	(1, –1, –1)	(1, –1, –1)
(–1 , 1, 1)	(–1, 1, 1)	(–1, 1, 1)	(–1, 1, 1)
(–1 , 1, 1)	(–1, 1, 1)	(–1, 1, 1)	(–1, 1, 1)

Figure 4.5. Illustration of a simple filter for horizontal ketchup line detection.

Source: James L McClelland, Creative Commons License.

Convolution filters follow the same principle. A filter of size m by n is capable of matching with a corresponding number of pixels. It is worth mentioning that each pixel or filter possesses multiple channels. An interesting example is a filter specifically designed to identify the horizontal edges of ketchup bottles, as depicted in Figure 4.5. While the input light seems predominantly red, the filter's topmost row is strongly activated, whereas it is less activated in the blue and green channels. The following two rows require a reduction in the amount of red color, to create a contrast, while increasing the presence of blue and green colors (Ghazal, 2022).

Figure 4.6 depicts an easy TF example that shows the application of a small convolutions feature to a simple image artificially created artificially. As previously mentioned, the initial input for conv2D involves a 4D tensor, specifically referred to as the constant I. In the preceding comment,

one illustrates the model as a basic 2D array, excluding the additional dimensions introduced by batch size (in this case, one) or channel size (also one). The second parameter is a 4D tensor of filters, marked as W. A comment is included to illustrate a 2D version of the tensor, without the additional dimensions representing the number of channels and number of filters (one each) (Ardakani et al., 2017). Next, one demonstrates the implementation of the conv2D function using horizontal and vertical strides of one, along with valid padding. Upon examining the outcome, it becomes apparent that it is in a 4D format, consisting of batch size (1), height (2), width (2), and channels (1). Both the width and height for the image are significantly smaller due to the use of valid padding. Additionally, the filter has a high activity level (6), which is expected as it is specifically designed to detect vertical lines that are present in the image (Ma et al., 2017).

4.2. A SIMPLE TF CONVOLUTION

Next, an example that demonstrates the process of transforming the feed-forward TF Mnist program to a model using convolutional neural networks is presented in Figure 4.7. As previously mentioned, the crucial call is TF function tf.nn.conv2d (Bi et al., 2021). Figure 4.7 displays the information on line 5.

```
ii = [[ [[0],[0],[2],[2]],
        [[0],[0],[2],[2]],
        [[0],[0],[2],[2]],
        [[0],[0],[2],[2]] ]]
''' ((0 0 2 2)
     (0 0 2 2)
     (0 0 2 2)
     (0 0 2 2))'''
I = tf.constant(ii, tf.float32)

ww = [ [[[-1]],[[-1]],[[1]]],
       [[[-1]],[[-1]],[[1]]],
       [[[-1]],[[-1]],[[1]]] ]
'''((-1 -1 1)
    (-1 -1 1)
    (-1 -1 1))'''
W = tf.constant(ww, tf.float32)

C =  tf.nn.conv2d( I, W, strides=[1, 1, 1, 1], padding='VALID')
sess = tf.Session()
print sess.run(C)
'''[ [[ 6.]  [ 0.]]
     [[ 6.]  [ 0.]]]]'''
```

Figure 4.6. Illustration of A basic TensorFlow (TF) exercise utilizing conv2D

Source: Richard S Sutton, Creative Commons License.

```
1  image = tf.reshape(img, [100, 28, 28, 1])
2     #Turns img into 4d Tensor
3  flts=tf.Variable(tf.truncated_normal([4,4,1,4],stddev=0.1))
4     #Create parameters for the filters
5  convOut = tf.nn.conv2d(image, flts, [1,  2,  2,  1], "SAME")
6     #Create graph to do convolution
7  convOut= tf.nn.relu(convOut)
8     #Don't forget to add nonlinearity
9  convOut=tf.reshape(convOut,[100, 784])
10    #Back to  100 1d image vectors
11 prbs = tf.nn.softmax(tf.matmul(convOut, W) + b)
```

Figure 4.7. Illustration of primary code needed to turn into a convolutional NN.

Source: Amey Varangaonkar., Creative Commons License.

image = tf.reshape(img,[100, 28, 28, 1])

One will examine each code line individually starting with line 1. The image can be represented as a four-dimensional tensor, specifically a vector of three-dimensional images. To meet the requirements of tf.nn.conv2d, a batch size of 100 was selected, which requires the presence of 100 3D images. The reshape function processes the input data to transform the input to a shape of 28x28. The final value represents that there is only one input channel. TF reshape function acts similarly to the reshape function found in Numpy (Tajmirriahi et al., 2022).

[height, width, channels, number]

The filter parameters for the neural network model are generated in line 3 with an initial mean of zero and a standard deviation of 0.1, consequently, the model can learn these values (Park et al., 2020). Filters of size 4 by 4 are selected, with pixels having one channel. Additionally, four filters in total are chosen. It is important to note that all these parameters height and width of the filter, as well as the number of filters created, are called hyperparameters. The total number of channels (in this instance, 1) is set by the number of channels present in the image and remains constant.

convOut = tf.nn.conv2d(image, flts, [1, 2, 2, 1], "SAME")

In line 5, the argument passed to tf.nn.conv2d relates to the filters that will be utilized. This tensor has a unique shape, consisting of four dimensions. The strides parameter for tf.nn.conv2d requires a list consisting of four integers to specify the stride size within each dimension of the input. Upon examination of line 5, it is evident that strides 1, 2, 2, and 1 have been selected. Typically, the values of the first and final elements are set to 1 in most cases. It is difficult to envision a scenario that could not be 1. Indeed, the initial dimension represents the distinct 3D images within the batch (Wang et al., 2018). By setting the stride in this dimension to two, it would effectively be skipping each additional image. Interestingly, in the case where the final stride is greater than one, such as two, and there are three color channels, the blue and red light would selectively be focused on, while disregarding the green channel. Stride values are often chosen such as (1, 1, 1, 1) for typical cases. However, if the intention is to convolve each additional image patch within both vertical and horizontal directions, a stride of (1, 2, 2, 1) can be used. It is common to come across discussions about tf.nn.conv2d where instructions emphasize the requirement for the initial and final strides to be one (Kondylidis et al., 2018).

The last parameter, padding, should be set to one from the recognized padding types in TF, such as SAME. The output of the conv2d function bears a striking resemblance to its input. Similarly, the output is also a 4D tensor, and just like the input, the first dimension in the output represents the batch size. The result is a collection of convolution outputs, with each output corresponding to an input image (Ackroyd, 1971). The two dimensions provided show the quantity of filter applications in both the horizontal and vertical directions. The last dimension of the output tensor indicates the quantity of filters being applied to the image during convolution (Saia et al., 2015). Put simply, the output's form is

[batch-size, horizontal-size, vertical-size, number-filters]

In line 11, the 784 values are passed into an entirely connected layer that generates logits for each image. These logits are then inputted into softmax, and the cross-

entropy loss is computed (not depicted in Figure 4.7). This results in a straightforward convolutional neural network for Mnist. In addition, it is worth noting that in line 7 mentioned earlier, a nonlinearity is introduced within the output from the convolution or the input for the layer that is completely connected. This holds significant importance. As previously observed, the absence of nonlinear activation functions within linear units leads to a lack of improvement (Xu et al., 2017).

4.3. MULTILEVEL CONVOLUTION

As previously mentioned, the accuracy can be enhanced by transitioning from a single layer of convolution to multiple layers. Within this section, one will proceed to develop a model consisting of two layers (Yang et al., 2019).

An important aspect of multilevel convolution is to maintain the same format of the input image input Both consist of batch size vectors containing 3D images, where the images are 2D with an additional dimension representing the total number of channels. Therefore, the result obtained from one layer for convolution can serve as the input for a subsequent layer, which is precisely the approach taken. When discussing the image produced by the data, the final dimension represents the quantity of color channels (Amin et al., 2019). When referring to the conv2d output, one describes the last dimension as the number of distinct filters within the convolution layer. The term "filter" used in this context is quite appropriate. It can be compared to a light color filter to selectively allow just blue light to pass through a lens, after a colored filter is placed directly in front of it. Three filters allow us to obtain images in the RGB spectrum. The hypothetical image generated by the filter is like the example shown in Figure 4.1. In addition, similar to weights assigned to filters for images with RGB, the second convolution layer also has weights for every channel of output from the first (Mohanty & Meher, 2012).

The code for transforming the feed-forward Minst NN to a two-layer convolution model is provided in Figure 4.8. Lines 1–4 replicate the initial lines of Figure 4.7, with the exception that in line 2, the initial convolution layer is enhanced to 16, as opposed to the previous version which had 4 filters. The creation of both second convolution layer filters, flts2, is attributed to Line 2. It should be noted that there are a total of 32 of these items (Tian et al., 2021). This is demonstrated in Line 5 when the values for all 32 filters are used as the input channel values for the second convolution layer.

In line 7, the image is represented linearly with 1568 scalar values. These values correspond to the height for a weight matrix (W) which is responsible for transforming the image values in 10 logits.

```
image = tf.reshape(img, [100, 28, 28, 1])
flts=tf.Variable(tf.normal([4, 4, 1, 16], stddev=0.1))
convOut = tf.nn.conv2d(image, flts, [1,  2,  2,  1], "SAME")
convOut= tf.nn.relu(convOut)
flts2=tf.Variable(tf.normal([2, 2, 16, 32], stddev=0.1))
convOut2 = tf.nn.conv2d(convOut, flts2, [1,  2,  2,  1], "SAME")
convOut2 = tf.reshape(convOut2, [100, 1568])
W = tf.Variable(tf.normal([1568,10],stddev=0.1))
prbs = tf.nn.softmax(tf.matmul(convOut2, W) + b)
```

Figure 4.8. Illustration of primary code needed to turn into a two-layer convolutional NN.

Source: Rui Zhao, Creative Commons License.

Did you know?

Convolutional Neural Networks (CNNs) have transformed computer vision tasks, such as identifying and detecting objects in images, in a groundbreaking manner. Their capacity to autonomously acquire structured models of data has resulted in notable progress in various domains, including medical imaging and autonomous driving.

When considering the model as a whole, it is important to see the overall progression. Initially, one gets an image with dimensions of 28 ∗ 28. Eventually, we obtain a "picture" with dimensions of 7 ∗ 7. Additionally, at each position in the 2D array, there are 32 distinct filter values. In other words, one has divided the image in 49 patches, with each patch initially consisting of 4 * 4 pixels and now being represented with 32 filter values. Given that this enhances efficiency, one may infer that these figures are indicative of significant aspects of the activities occurring within their respective 4 ∗ 4 sections (Du & Ward, 2009).

Indeed, this appears to be true. Although the values within the filters may initially seem confusing, studying them at the introductory levels might uncover some underlying rationale in their design. Figure 4.9 displays the 4 ∗ 4 weights for four out of all eighth first-level convolution filter coefficients that were acquired during a single execution of the code depicted in Figure 4.7. These all numbers are difficult to understand however, referring to Figure 4.10 should assist. Figure 4.10 was generated by printing the filter that had the highest value overall of 14 for 14 points within the image following the initial convolution layer. This filter was applied to our typical image of a 7. Soon enough, the image of a 7 becomes apparent amidst a sea of zeros, indicating that filter 0 corresponds to a section consisting entirely of zeros (Kiranyaz et al., 2021).

```
-0.152168 -0.366335 -0.464648 -0.531652
 0.0182653 -0.00621072 -0.306908 -0.377731
 0.482902  0.581139  0.284986  0.0330535
 0.193956  0.407183  0.325831  0.284819

 0.0407645  0.279199  0.515349  0.494845
 0.140978  0.65135  0.877393  0.762161
 0.131708  0.638992  0.413673  0.375259
 0.142061  0.293672  0.166572 -0.113099

 0.0243751  0.206352  0.0310258 -0.339092
 0.633558  0.756878  0.681229  0.243193
 0.894955  0.91901  0.745439  0.452919
 0.543136  0.519047  0.203468  0.0879601

 0.334673  0.252503 -0.339239 -0.646544
 0.360862  0.405571 -0.117221 -0.498999
 0.520955  0.532992  0.220457  0.000427301
 0.464468  0.486983  0.233783  0.101901
```

Figure 4.9. Illustration of Four of the eight filters generated in a single iteration of the two-layer convolutional NN labeled as 0, 1, 2, and 7.

Source: Steven Miller, Creative Commons License.

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 5 2 2 2 2 2 2 2 2 2 0 0
0 0 1 1 4 4 4 4 4 2 2 2 0 0
0 0 1 1 1 1 1 1 1 1 2 7 0 0
0 0 0 0 0 0 0 5 1 4 2 7 0 0
0 0 0 0 0 0 0 5 1 2 7 0 0 0
0 0 0 0 0 0 5 1 4 2 7 0 0 0
0 0 0 0 0 5 2 1 2 7 0 0 0 0
0 0 0 0 0 5 1 4 2 0 0 0 0 0
0 0 0 0 5 1 4 2 7 0 0 0 0 0
0 0 0 0 2 1 2 2 0 0 0 0 0 0
0 0 0 0 1 1 1 7 0 0 0 0 0 0
```

Figure 4.10. Illustration of The most prominent feature is identified for each of the 14 of 14 points at layer 1.

Source: Felix Mohr, Creative Commons License.

It is worth mentioning that the diagonal edge of the number 7 on the right side consists mostly of 7s, while the lower part of the image represents the number 1. The arg-max function in Numpy is used to determine the index for the highest value in a provided list of numbers. The pixel values within the empty zones have been assigned a value of zero, causing all the filters to provide a result of zero. This is the expected behavior when every number is equal and the arg-max function comes back with the first value (Kuo, 2016). Figure 4.11 bears resemblance to Figure 4.10, with the distinction being that it illustrates the filters that exhibit the highest level of activity in the second layer of the model.

4.4. CONVOLUTION DETAILS

4.4.1. Biases

Biases can also be incorporated into the convolution kernels. This point has not been discussed yet. It is worth noting that in the most recent example, multiple filters were applied to each patch. Specifically, within line 2 of Figure 4.8, it has specified the usage of 16 distinct filters (Greenland, 1996).

0	0	0	0	0	0	0
17	11	31	17	17	16	16
6	16	12	6	6	5	5
17	17	17	5	24	5	10
0	0	11	26	3	5	0
0	17	11	24	5	10	0
0	6	24	8	5	0	0

Figure 4.11. Illustration of most active features for all 7 by 7 points in layer 2 after processing.

Source: Andrew W, Creative Commons License.

Introducing a bias into the program can lead to an imbalance in the weight assigned to different filter channels. This is achieved by incorporating a distinct value into the convolution output of the channel. Thus, the number of bias variables within a given convolution layer is equal to the total number of output channels. In Figure 4.8, additional biases may be added to the first convolution layer by inserting a code snippet between lines 3 and 4:

bias = tf.Variable(tf.zeros [16]) convOut += bias

Implicit broadcasting is a fundamental concept in certain computational processes. The shape of convOut is [100, 14, 14, 16], while the shape of bias is [16]. Therefore, when they are added together, it results in the creation of [100, 14, 14] versions of bias (MacCoun, 1998).

4.4.2. Pooling

For larger pictures, such as those with dimensions of 1000 ∗ 1000 pixels, the difference in image size in the original image along with the values inputted into a layer that is entirely linked followed by softmax on the end becomes significantly more pronounced. There are TensorFlow functions available to assist with this reduction (Wischik et al., 2008). It is important to observe the program in which the reduction occurred because the strides within convolution were limited to every other patch. An alternative approach is the next:

convOut = tf.nn.conv2d(image, flts, [1,1,1,1], "SAME")

convOut = tf.nn.max pool(convOut, [1,2,2,1], [1,2,2,1], "SAME").

This set of line serves as a replacement for line 3 in Figure 4.8. Instead of employing the convolutional neural network having a stride of two, first it is employed the convolutional neural network having a stride of one. Therefore, convOut has a shape of [batchSz, 28, 28, 1] with no reduction within image size. The subsequent line provides a decrease in image size that precisely matches the stride for two that was initially employed (Zafar et al., 2022).

One important function in this context is the max pool, which identifies the highest value within a filter's region at the image, like the conv2d, it has three identical arguments. One aspect to consider is the 4D tensor for images one uses as a standard, and the strides and padding need to be considered. In the given example, the max pool operation is applied to the convOut, which is the 4D output from the initial convolution. It is accomplishing this stride in [1, 2, 2, 1]. One aspect to consider is examining each image on the batch size, while another important aspect is examining each channel. The two 2s indicate a horizontal and vertical shift of two units before repeating the operation. The size of the region in which the maximum is to be found can be specified as the second argument. In the typical scenario, the initial and final 1s are essentially predetermined, while the two 2s in the middle indicate that one needs to calculate the maximum value within a 2 * 2 patch of convOut (Sun et al., 2017).

Figure 4.12 illustrates the comparison between two methods of achieving a four-fold drop in dimensionality using the Mnist program, and the reduction is applied to a 4 * 4 image (the numbers are randomly selected). Applying the filter with a stride of two (Same padding) in the top row resulted in an array of filter values with dimensions of 2 x 2. In the second row, the filter was applied with a stride of one, resulting in the creation of a 4 * 4 array for values. Next, for every individual 2 ∗ 2 patch, the highest value is extracted to generate the final array located in the lower right part of the diagram. It is worth mentioning that there is another type of pooling called average pool. This type of pooling functions similarly to the max pool, with the only difference being that the value over a pool is calculated as the average for the individual values, rather than the maximum (Cui et al., 2017).

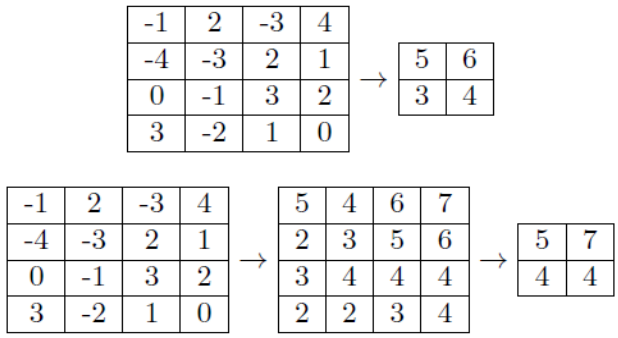


Figure 4.12. Illustration of factor of 4 dimensionality reduction, with and without max pool.

Source: Georey Hinton, Creative Commons License.

ACTIVITY 4.1.

Objective: To visually explore how convolutional filters operate on images to extract features.

Materials Needed:

- Python environment with TensorFlow or similar library installed
- Sample image(s)
- Jupyter Notebook or similar IDE

Steps:

1. Setup:
   - Set up the programming environment with TensorFlow or another library installed.
   - Load and display a sample image.
2. Apply Convolutional Filter:
   - Define a simple convolutional filter (e.g., edge detector, blur filter).
   - Apply the filter to the image and visualize the resulting feature map.
3. **Experiment and Discuss**:
   - Modify the filter parameters (e.g., size, weights).
   - Discuss how different filters affect feature extraction and image representation.

SUMMARY

- The chapter on Convolutional Neural Networks covers essential concepts including filters, strides, padding, and pooling. Filters are used to extract features from input data, with strides determining the step size of the filter's movement across the input. Padding adjusts input dimensions to ensure consistent output size after convolution.
- TensorFlow provides a straightforward method for convolution operations. Multilevel convolution involves stacking multiple layers of convolutions to learn hierarchical features. Details of convolution include weight sharing and feature map dimensions. Biases are additional learnable parameters that adjust output along with weights. Pooling reduces feature map size by aggregating neighboring values.

REVIEW QUESTIONS

1. Describe the role of filters in Convolutional Neural Networks (CNNs). How do they contribute to feature extraction?
2. `Explain how padding, strides, and filter size influence the output dimensions of convolutional layers.
3. Compare and contrast max pooling and average pooling. In what scenarios might one be preferred over the other?
4. What is the purpose of biases in convolutional layers? How do they impact the learning process in CNNs?
5. Discuss the concept of multilevel convolution in CNNs. How does stacking multiple convolutional layers contribute to learning hierarchical features?
6. Explain the term "local connectivity" in the context of convolutional operations. How does it relate to the efficiency of CNNs compared to fully connected networks?
7. How does the depth of a CNN (number of convolutional layers) affect its performance and ability to learn complex representations?

MULTIPLE CHOICE QUESTIONS

1. What role do filters play in Convolutional Neural Networks (CNNs)?

a. They determine the size of the input data.

b. They extract features from input data.

c. They control the learning rate.

d. They define the activation function.

2. What does "padding" refer to in CNNs?

a. Adding extra layers to the network.

b. Adjusting the learning rate dynamically.

c. Adding zeros around the input data.

d. Increasing the size of filters.

3. How does increasing stride affect the output size in CNNs?

a. It increases the size of the output.

b. It decreases the size of the output.

c. It has no effect on the output size.

d. It reduces the number of filters.

4. What does a pooling layer in a CNN do?

a. Increases the size of feature maps.

b. Adds more filters to the network.

c. Reduces the dimensionality of feature maps.

d. Introduces non-linearities in the network.

5. Why are biases used in convolutional layers?

a. To prevent over fitting.

b. To add non-linearity to the model.

c. To normalize the input data.

d. To shift the activation function.

Answers to Multiple Questions

1. (b); 2. (c); 3. (b); 4. (c); 5. (d).

REFERENCES

1. Ackroyd, M. H. (1971). Short-time spectra and time-frequency energy distributions. The Journal of the Acoustical Society of America, 50(5A), 1229–1231.
2. Adem, K. (2022). Impact of activation functions and number of layers on detection of exudates using circular Hough transform and convolutional neural networks. Expert Systems with Applications, 203, 117583.
3. Amin, S. U., Alsulaiman, M., Muhammad, G., Bencherif, M. A., & Hossain, M. S. (2019). Multilevel weighted feature fusion using convolutional neural networks for EEG motor imagery classification. IEEE Access, 7, 18940–18950.
4. Ardakani, A., Condo, C., Ahmadi, M., & Gross, W. J. (2017). An architecture to accelerate convolution in deep neural networks. IEEE Transactions on Circuits and Systems I: Regular Papers, 65(4), 1349–1362.

5. Batchuluun, G., Naqvi, R. A., Kim, W., & Park, K. R. (2018). Body-movement-based human identification using convolutional neural network. Expert Systems with Applications, 101, 56–77.
6. Bi, X., Zhang, C., He, Y., Zhao, X., Sun, Y., & Ma, Y. (2021). Explainable time–Frequency convolutional neural network for micro seismic waveform classification. Information Sciences, 546, 883–896.
7. Chou, F. I., Tsai, Y. K., Chen, Y. M., Tsai, J. T., & Kuo, C. C. (2019). Optimizing parameters of multi-layer convolutional neural network by modeling and optimization method. IEEE Access, 7, 68316–68330.
8. Cui, Y., Zhou, F., Wang, J., Liu, X., Lin, Y., & Belongie, S. (2017). Kernel pooling for convolutional neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 2921–2930).
9. Dorj, U. O., Lee, K. K., Choi, J. Y., & Lee, M. (2018). The skin cancer classification using deep convolutional neural network. Multimedia Tools and Applications, 77, 9909–9924.
10. Du, S., & Ward, R. K. (2009). Improved face representation by nonuniform multilevel selection of Gabor convolution features. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 39(6), 1408–1419.
11. Fotouhi, S., Pashmforoush, F., Bodaghi, M., & Fotouhi, M. (2021). Autonomous damage recognition in visual inspection of laminated composite structures using deep learning. Composite Structures, 268, 113960.
12. Ghazal, T. M. (2022). Convolutional neural network based intelligent handwritten document recognition. Computers, Materials & Continua, 70(3), 4563–4581.
13. Gonzalez, R. C. (2018). Deep convolutional neural networks [Lecture notes]. IEEE Signal Processing Magazine, 35(6), 79–87.
14. Greenland, S. (1996). Basic methods for sensitivity analysis of biases. International Journal of Epidemiology, 25(6), 1107–1116.
15. Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., & Chen, T. (2018). Recent advances in convolutional neural networks. Pattern Recognition, 77, 354–377.
16. Hashemi, M. (2019). Enlarging smaller images before inputting into convolutional neural network: Zero-padding vs. interpolation. Journal of Big Data, 6(1), 1–13.
17. Helmy, A. A., Omar, Y. M., & Hodhod, R. (2018). An innovative word encoding method for text classification using convolutional neural network. In 2018 14th International Computer Engineering Conference (ICENCO) (pp. 42–47).
18. Hossain, M. A., & Sajib, M. S. A. (2019). Classification of image using convolutional neural network (CNN). Global Journal of Computer Science and Technology, 19(2), 13–14.
19. Kamath, U., Liu, J., Whitaker, J., Kamath, U., Liu, J., & Whitaker, J. (2019). Convolutional neural networks. Deep Learning for NLP and Speech Recognition, 4(3), 263–314.

20. Kiranyaz, S., Avci, O., Abdeljaber, O., Ince, T., Gabbouj, M., & Inman, D. J. (2021). 1D convolutional neural networks and applications: A survey. Mechanical Systems and Signal Processing, 151, 107398.

21. Kondylidis, N., Tzelepi, M., & Tefas, A. (2018). Exploiting tf-idf in deep convolutional neural networks for content-based image retrieval. Multimedia Tools and Applications, 77(23), 30729–30748.

22. Krichen, M. (2023). Convolutional neural networks: A survey. Computers, 12(8), 151.

23. Kubanek, M., Bobulski, J., & Kulawik, J. (2019). A method of speech coding for speech recognition using a convolutional neural network. Symmetry, 11(9), 1185.

24. Kuo, C. C. J. (2016). Understanding convolutional neural networks with a mathematical model. Journal of Visual Communication and Image Representation, 41, 406–413.

25. Lee, J., Park, J., Kim, K. L., & Nam, J. (2018). SampleCNN: End-to-end deep convolutional neural networks using very small filters for music classification. Applied Sciences, 8(1), 150.

26. Li, Z., Liu, F., Yang, W., Peng, S., & Zhou, J. (2021). A survey of convolutional neural networks: Analysis, applications, and prospects. IEEE Transactions on Neural Networks and Learning Systems, 33(12), 6999–7019.

27. Liu, G., Dundar, A., Shih, K. J., Wang, T. C., Reda, F. A., Sapra, K., & Catanzaro, B. (2022). Partial convolution for padding, inpainting, and image synthesis. IEEE Transactions on Pattern Analysis and Machine Intelligence, 45(5), 6096–6110.

28. Ma, J., Wu, F., Jiang, T. A., Zhu, J., & Kong, D. (2017). Cascade convolutional neural networks for automatic detection of thyroid nodules in ultrasound images. Medical Physics, 44(5), 1678–1691.

29. MacCoun, R. J. (1998). Biases in the interpretation and use of research results. Annual Review of Psychology, 49(1), 259–287.

30. Mohanty, B. K., & Meher, P. K. (2012). Memory-efficient high-speed convolution-based generic structure for multilevel 2-D DWT. IEEE Transactions on Circuits and Systems for Video Technology, 23(2), 353–363.

31. Naseri, H., & Mehrdad, V. (2023). Novel CNN with investigation on accuracy by modifying stride, padding, kernel size and filter numbers. Multimedia Tools and Applications, 82(15), 23673–23691.

32. Nebauer, C. (1998). Evaluation of convolutional neural networks for visual recognition. IEEE Transactions on Neural Networks, 9(4), 685–696.

33. Pamula, T. (2018). Road traffic conditions classification based on multilevel filtering of image content using convolutional neural networks. IEEE Intelligent Transportation Systems Magazine, 10(3), 11–21.

34. Pang, B., Nijkamp, E., & Wu, Y. N. (2020). Deep learning with TensorFlow: A review. Journal of Educational and Behavioral Statistics, 45(2), 227–248.

35. Pang, Y., Sun, M., Jiang, X., & Li, X. (2017). Convolution in convolution for network in network. IEEE Transactions on Neural Networks and Learning Systems, 29(5), 1587–1597.

36. Park, S., Koh, Y., Jeon, H., Kim, H., Yeo, Y., & Kang, J. (2020). Enhancing the interpretability of transcription factor binding site prediction using attention mechanism. Scientific Reports, 10(1), 13413.

37. Prusa, J. D., & Khoshgoftaar, T. M. (2017). Improving deep neural network design with new text data representations. Journal of Big Data, 4, 1–16.

38. Reinel, T. S., Brayan, A. A. H., Alejandro, B. O. M., Alejandro, M. R., Daniel, A. G., Alejandro, A. G. J., & Raul, R. P. (2021). GBRAS-Net: A convolutional neural network architecture for spatial image steganalysis. IEEE Access, 9, 14340–14350.

39. Romanuke, V. V. (2018). Smooth non-increasing square spatial extents of filters in convolutional layers of CNNs for image classification problems. Applied Computer Systems, 23(1), 52–62.

40. Saia, R., Boratto, L., & Carta, S. (2015). A proactive time-frame convolution vector (TFCV) technique to detect fraud attempts in e-commerce transactions. International Journal of e-Education, e-Business, e-Management and e-Learning, 5(4), 229.

41. Salomon, M., Couturier, R., Guyeux, C., Couchot, J. F., & Bahi, J. M. (2017). Steganalysis via a convolutional neural network using large convolution filters for embedding process with same stego key: A deep learning approach for telemedicine. European Research in Telemedicine/La Recherche Européenne en Télémédecine, 6(2), 79–92.

42. Shalbaf, A., Bagherzadeh, S., & Maghsoudi, A. (2020). Transfer learning with deep convolutional neural network for automated detection of schizophrenia from EEG signals. Physical and Engineering Sciences in Medicine, 43, 1229–1239.

43. Sun, M., Song, Z., Jiang, X., Pan, J., & Pang, Y. (2017). Learning pooling for convolutional neural network. Neurocomputing, 224, 96–104.

44. Tajmirriahi, M., Amini, Z., Rabbani, H., & Kafieh, R. (2022). An interpretable convolutional neural network for P300 detection: Analysis of time frequency features for limited data. IEEE Sensors Journal, 22(9), 8685–8692.

45. Tian, W., Cheng, X., Li, G., Shi, F., Chen, S., & Zhang, H. (2021). A multilevel convolutional recurrent neural network for blade icing detection of wind turbine. IEEE Sensors Journal, 21(18), 20311–20323.

46. Traore, B. B., Kamsu-Foguem, B., & Tangara, F. (2018). Deep convolution neural network for image recognition. Ecological Informatics, 48, 257–268.

47. Wang, M., Tai, C., E, W., & Wei, L. (2018). DeFine: Deep convolutional neural networks accurately quantify intensities of transcription factor-DNA binding and facilitate evaluation of functional non-coding variants. Nucleic Acids Research, 46(11), e69–e69.

48. Wischik, D., Handley, M., & Braun, M. B. (2008). The resource pooling principle. ACM SIGCOMM Computer Communication Review, 38(5), 47–52.

49. Xu, J., Xu, B., Wang, P., Zheng, S., Tian, G., & Zhao, J. (2017). Self-taught convolutional neural networks for short text clustering. Neural Networks, 88, 22–31.

50. Xu, S., Rao, H., Peng, H., Jiang, X., Guo, Y., Hu, X., & Hu, B. (2020). Attention-based multilevel co-occurrence graph convolutional LSTM for 3-D action recognition. IEEE Internet of Things Journal, 8(21), 15990–16001.

51. Yamashita, R., Nishio, M., Do, R. K. G., & Togashi, K. (2018). Convolutional neural networks: An overview and application in radiology. Insights into Imaging, 9, 611–629.

52. Yang, Y., Nie, Z., Huang, S., Lin, P., & Wu, J. (2019). Multilevel features convolutional neural network for multifocus image fusion. IEEE Transactions on Computational Imaging, 5(2), 262–273.

53. Zafar, A., Aamir, M., Mohd Nawi, N., Arshad, A., Riaz, S., Alruban, A., & Almotairi, S. (2022). A comparison of pooling methods for convolutional neural networks. Applied Sciences, 12(17), 8643.

54. Zhai, H., Zhao, J., & Zhang, H. (2022). Double attention based multilevel one-dimensional convolution neural network for hyperspectral image classification. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 15, 3771–3787.

55. Zhang, Y. D., Muhammad, K., & Tang, C. (2018). Twelve-layer deep convolutional neural network with stochastic pooling for tea category classification on GPU platform. Multimedia Tools and Applications, 77, 22821–22839.

CHAPTER

5

Deep Learning with PyTorch

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

- Understand the basics of tensors in PyTorch and their role in deep learning.
- Learn tensor creation, types, and operations in PyTorch.
- Learn tensor creation methods and operations in PyTorch.
- Understand automatic gradient computation in deep learning frameworks.
- Differentiate between static and dynamic computation graphs in DL.
- Discover how to build and customize neural network architectures using PyTorch's nn.Module

KEY TERMS FROM THIS CHAPTER

Automatic differentiation
Deep learning
GPU acceleration
Gradient computation
Loss functions
Neural networks
nn.Module
PyTorch
Tensors

UNIT INTRODUCTION

Advancements in Reinforcement Learning (RL), particularly when combined with DL, have opened up new possibilities for tackling increasingly complex problems. One reason for this is the advancement of deep learning techniques and resources. This chapter focuses on PyTorch, a powerful tool that allows us to effortlessly implement complex deep learning models using Python code (Gao et al., 2020).

This chapter does not claim to be a comprehensive guide to DL, as the field is vast and constantly evolving. However, it will address that assuming you already have a solid understanding of deep learning fundamentals, let's dive into the specifics and implementation details of the PyTorch library. Libraries built on PyTorch that aim to simplify common problems in deep learning The PyTorch ignite library will be utilized in a few of the examples (Virtsionis et al., 2022).

This module delves into fundamental concepts in deep learning using PyTorch, a highly adaptable framework that has gained widespread popularity due to its versatility and computational efficiency. The unit starts with an introduction to tensors, which are the core data structure in PyTorch. It covers how to create, manipulate, and perform operations on tensors (Laporte et al., 2019). The content explores the utilization of GPU acceleration to improve performance and covers the concept of gradients for automatic differentiation, which is crucial for optimizing neural networks. The unit also explores the fundamental components of neural networks and the process of designing custom layers, offering practical insights into the construction and customization of deep learning architectures (Novac et al., 2022).

5.1. TENSORS

Tensors serve as the foundational component of all deep learning frameworks. The concept may seem unknown at first, but in reality, a tensor is simply a multi-dimensional array. Comparing to school math, a solitary number can be likened to a point, which lacks dimensionality. On the other hand, a vector possesses one dimension, similar to a line segment, and a matrix is a two-dimensional entity. Parallelepiped of numbers can be used to represent three-dimensional number collections, although they lack a distinct name like a matrix. It is possible to retain the term "tensor" when referring to collections with higher dimensions (Bi et al., 2021).

It is important to mention that tensors used in deep learning have only a minor connection to tensors used in tensor calculus or tensor algebra. A multi-dimensional array is referred to as a tensor in deep learning terminology. On the other hand, a tensor in mathematics is a mapping between vector spaces. Although in some circumstances it can be expressed as a multi-dimensional array, its semantic value is far richer. Mathematicians typically disapprove of individuals who assign well-established mathematical words to label unrelated concepts, so exercise caution (Figure 5.1) (Zhang et al., 2014).

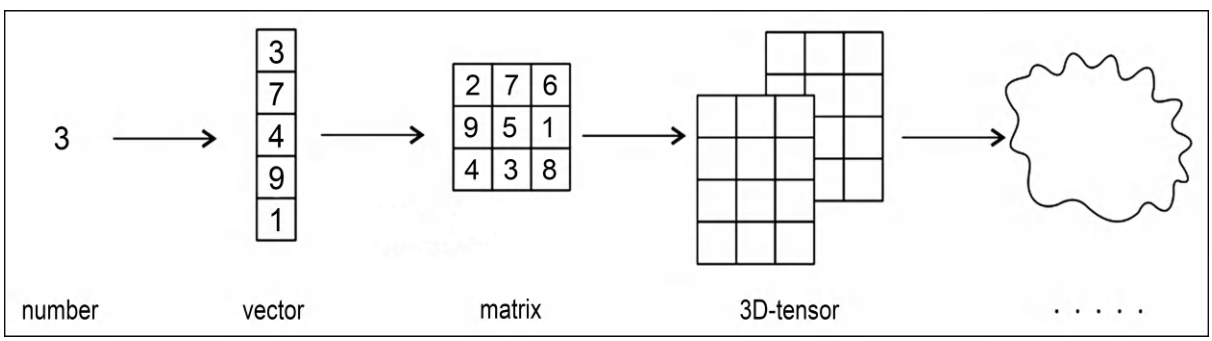


Figure 5.1. Illustration of the going from a single number to an n-dimensional tensor.

Source: Maxim Lapan, Creative Commons License.

5.1.1. The Creation of Tensors

Those who have become familiar with the NumPy library (as suggested) are already aware that its main purpose is to efficiently manage multi-dimensional arrays. Although not referred to as tensors in NumPy, these arrays are indeed tensors. Tensors are extensively utilized in scientific computations as a versatile means of storing data. For example, a color image can be represented as a three-dimensional tensor, incorporating the dimensions of width, height and color plane (McCrea & Mikhail, 1956).

In addition to dimensions, a tensor is defined by the nature of its elements. PyTorch supports a total of eight types, including three float types and five integer types. The float types consist of 16-bit, 32-bit, and 64-bit, whereas the integer types encompass 8-bit signed, 8-bit unsigned, 16-bit, 32-bit, and 64-bit. Torch has distinct classes to

represent tensors of various sorts, with torch being the most frequently utilized. A FloatTensor, which represents a 32-bit float, is used in PyTorch. ByteTensor is an 8-bit unsigned integer, and torch.LongTensor is a data type in PyTorch that represents a 64-bit signed integer. One can find the remaining information in the PyTorch documentation (Pirani,1955).

PyTorch offers three different methods for creating tensors:

- To call a constructor of the required type, one can simply use the appropriate constructor.
- Converting a NumPy array or a Python list into a tensor is a common practice. The type will be determined based on the array's type in this scenario.
- One can instruct PyTorch to generate a tensor with the desired data. As an illustration, the torch.zeros() function can be utilized to generate a tensor that is populated with zero values (Zhao et al., 2021).

Let's explore some examples of these methods by examining a simple session:

```
>>> import torch
>>> import numpy as np
>>> a = torch.FloatTensor(3, 2)
>>> a
tensor([[4.1521e+09,  4.5796e-41],
        [ 1.9949e-20, 3.0774e-41],
        [ 4.4842e-44, 0.0000e+00]])
```

Here, one included both PyTorch and NumPy libraries and initialized a tensor of size 3×2 without assigning any values to it. PyTorch, by default, assigns memory for the tensor without any initialization (Zhang et al., 2006). In order to clear the content of the tensor, it is necessary to utilize its operation:

```
>>> a.zero_()
tensor([[ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.]])
```

There are two different types of operations that can be performed on tensors: in place operations and functional operations. Operations that modify the tensor's content have an underscore appended to their name. Following this, the object is then returned. The accomplished alteration creates a copy of the tensor, leaving the original tensor unaltered. In general, doing processes in place results in better performance and less memory utilization (McRae et al., 2016).

A possible method for creating a tensor is to use its constructor and provide a Python iterable, such as a tuple or list, to serve as the contents of the newly created tensor:

```
>>> torch.FloatTensor([[1,2,3],[3,2,1]])
tensor([[ 1., 2., 3.],
        [ 3., 2., 1.]])
```

Here, an identical tensor using NumPy, initializing all its values to zero will be generated:

```
>>> n = np.zeros(shape=(3, 2))
>>> n
array([[ 0., 0.],
       [ 0., 0.],
       [ 0., 0.]])
>>> b = torch.tensor(n)
>>> b
tensor([[ 0., 0.],
        [ 0., 0.],
        [ 0., 0.]], dtype=torch.float64)
```

A NumPy array can be used as the input for the torch tensor method, which will produce a tensor with the appropriate shape. In the example above, a NumPy array was created with an initial value of zero. By default, this resulted in the creation of a double (64-bit float) array (Segal, 1956).

Therefore, the tensor obtained has a Double Tensor type, as demonstrated in the previous example by the dtype value. Typically, in deep learning, there is no need for double precision as it can lead to unnecessary memory usage and performance issues. It is generally recommended to utilize the 32-bit float type, or possibly the 16-bit float type, as they provide enough precision. In order to create a tensor, it is necessary to explicitly specify the type of NumPy array:

```
>>> n = np.zeros(shape=(3, 2), dtype=np.float32)
>>> torch.tensor(n)
tensor([[ 0., 0.],
        [ 0., 0.],
        [ 0., 0.]])
```

One possible approach is to specify the desired tensor type using the dtype argument in the torch.tensor function. However, it is important to note that this argument requires a PyTorch type specification, not a NumPy one. The torch package contains various types in PyTorch, such as float32 and uint8.

```
>>> n = np.zeros(shape=(3,2))
>>> torch.tensor(n, dtype=torch.float32)
tensor([[ 0., 0.],
        [ 0., 0.],
        [ 0., 0.]])
```

5.1.2. Scalar Tensors

Starting from the 0.4.0 release, PyTorch has included support for zero-dimensional tensors, which represent scalar values. These tensors can be generated through various operations, such as calculating the sum of all values within a tensor. In the past, the solution involved creating a tensor with a single dimension equal to one (Bergmann, 1968).

The solution proved effective, although it required additional indexation to access the desired value. Currently, the torch.tensor() function allows for the creation and support of zero-dimensional tensors, which are returned by the relevant functions (Horndeski, 1974). Using the special item () technique, one can retrieve a tensor's Python value:

```
>>> a = torch.tensor([1,2,3])
>>> a
tensor([ 1, 2, 3])
>>> s = a.sum()
>>> s
tensor(6)
>>> s.item()
6
>>> torch.tensor(1)
tensor(1)
```

5.1.3. Tensor Operations

There are numerous operations that can be performed on tensors, and the list is extensive. It is important to mention that besides the previously mentioned in-place and functional variations (i.e., with and without an underscore, like abs () and abs_ (), there are two sources to investigate for operations: the torch package and the tensor class. Typically, the function takes the tensor as an argument in the first case. It operates on the tensor that is being called (Lee & Cichocki, 2018)

Typically, tensor operations in PyTorch aim to mirror their counterparts in NumPy. Therefore, if there is a general function in NumPy, it is likely that PyTorch will offer a similar one. Some examples include the functions torch.transpose (), torch.stack(), and torch.cat () (Hackbusch & Kühn, 2009).

5.1.4. GPU Tensors

PyTorch effortlessly supports CUDA GPUs, allowing for automatic selection between CPU and GPU versions for all operations. The decision is determined by the type of tensors being manipulated (Lyakh, 2015). All the tensor types I mentioned have equivalents for both CPU and GPU. There is a little difference in the location of GPU tensors, as they are found in the torch.cuda package rather than just torch. As an example, a torch.FloatTensor is a tensor with a 32-bit floating-point data type that is stored in the memory of the CPU. However, torch.cuda.FloatTensor is the equivalent for GPU usage (Navarro et al., 2020).

To transition from CPU to GPU, one can utilize the tensor technique, to(device), which duplicates the tensor onto a designated device (which can be either GPU or CPU). If the tensor is already located on the device, no further action will be performed and the original tensor will be returned (Huang et al., 2022).

There are various methods to specify the device type. To begin with, a simple way to specify the device is by using a string name. For CPU memory, one can use "cpu", while for GPU, you can use "cuda". An optional device index can be added to a GPU device specification after the colon (Liu et al., 2022). Since the index starts at zero, the second GPU card in the system, for example, can be addressed as "cuda: 1".

An alternative and marginally more effective approach to indicate a device in the to() method is by utilizing the torch.device class, which allows for the inclusion of the device name and an optional index. To access the current device of your tensor, one can use the device property (Yang et al., 2019).

```
>>> a = torch.FloatTensor([2,3])
>>> a
tensor([ 2., 3.])
>>> ca = a.to('cuda'); ca
tensor([ 2.,3.], device='cuda:0')
```

Here, a tensor was created on the CPU and then transferred to the GPU memory. The computations can utilize both copies, and the user doesn't need to worry about any GPU-specific details:

```
>>> a = torch.FloatTensor([2,3])
>>> a
tensor([ 2., 3.])
>>> ca = a.to('cuda'); ca
tensor([ 2.,3.], device='cuda:0')
```

Did you know?

PyTorch tensors have the ability to handle several sorts of input, not simply numeric data. This versatility allows them to be used for a variety of deep learning applications that go beyond typical arithmetic calculations.

5.2. GRADIENTS

Despite the availability of transparent GPU support, engaging in all these tensor operations may not be worthwhile without the crucial "killer feature" of automatic gradient computation. This functionality was initially developed in the Caffe toolkit and later became widely adopted in deep learning libraries (Clarke, 1975).

Implementing and debugging the computation of gradients manually was a highly challenging task, even for the most basic neural network (NN). You had to perform calculations on your functions, apply the chain rule, and then implement the results, hoping that everything was done correctly. Engaging in this exercise can provide valuable insights into the inner workings of deep learning. However, it may not be adequate to repetition when implemented in various neural network architectures (Julesz, 1986).

Fortunately, those days are long gone, similar to the era of the early days of electronics with soldering irons and vacuum tubes! Now, constructing a neural network with numerous layers can be done by simply assembling it from preexisting components or, in more advanced cases, manually defining the transformation expression (Rivest et al., 1993).

Each gradient is carefully calculated, backpropagated, and implemented into the network. In order to accomplish this, it is necessary to specify the network architecture in terms of the deep learning library used. Although some parameters may vary, the overall structure must be consistent. This involves defining the sequence in which the network will process inputs and generate outputs (Figure 5.2) (Vergassola et al., 2007).

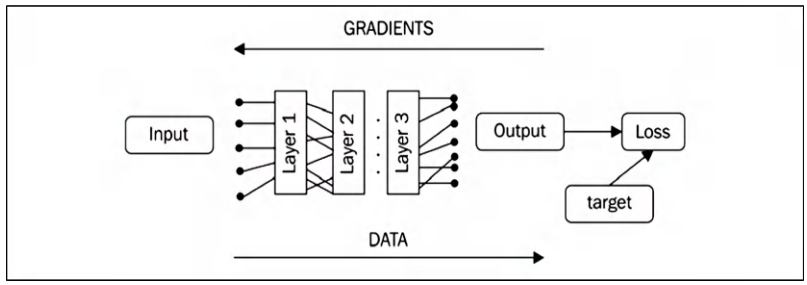


Figure 5.2. Illustration of data and gradients flowing through the NN.

Source: Rosenblatt, Creative Commons License.

The calculation method used for your gradients can have a significant impact. There are two methods or strategies:

- Static Graph: With this approach, it is crucial to specify your calculations, and they cannot be modified later on. Before any computation is made, the DL library will process and optimize the graph. This model is implemented in various deep learning toolkits, including TensorFlow (<2) and Theano.

- Dynamic Graph: There is no requirement to precisely specify your graph in advance before executing it. Instead, you only need to execute the operations that you wish to use for transforming your data. Throughout this process, the library will carefully document the sequence of operations executed. When instructed to calculate gradients, the system will methodically examine its past activities, combining the gradients of the network parameters. This technique is commonly referred to as notebook gradients and is utilized in Chainer, PyTorch,and other similar frameworks (Sehnke et al., 2010).

There are pros and cons to both approaches. As an example, utilizing a static graph typically results in faster performance since all computations can be efficiently executed on the GPU, thereby reducing the data transfer overhead. When working with a static graph, the library offers a higher level of flexibility when it comes to optimizing the order of computations or potentially removing certain parts of the graph (Ellis et al., 2012).

However, despite the increased computational overhead, dynamic graph provides developers with greater flexibility. As an example, one could state, "In the case of this particular dataset, it is possible to utilize this network twice, while for another dataset, a distinct model can be used with gradients clipped based on the batch mean." One of the notable advantages of the dynamic graph model is its ability to express transformations in a more natural and "Pythonic" manner, which adds to its appeal. Ultimately, it boils down to utilizing a Python library that offers an array of functions. Simply use these functions and witness the library's enchanting capabilities (Wierstra et al., 2010).

5.3. TENSORS AND GRADIENTS

Performing computations using tensor methods and functions provided by torch is made easier with PyTorch tensors, which come with a convenient built-in gradient calculation and tracking machinery. Simply convert your data into tensors and you're ready to go. Certainly, if you find yourself in need of delving into the underlying low-level complexities, you have the option to do so. However, in the majority of cases, PyTorch effortlessly fulfills your expectations (Meneveau, 2011).

Every tensor possesses various attributes associated with gradients:

- grad: A tensor property that stores the computed gradients, maintaining the same shape.
- is_leaf: This tensor's origin can be determined by its construction. If it was created by the user, it will be marked as True. However, if it is a result of a function transformation, it will be marked as False.
- requires_grad: Indicates whether or not gradients need to be calculated for this tensor. This value is obtained by leaf tensors during the construction process (using functions like torch.zeros() or torch.tensor()). It is then inherited by other tensors. Typically, the constructor has a default setting of requires_grad=False. Therefore, if you want gradients to be determined for your tensor, you must explicitly indicate this (Hasan et al., 2001).

To further understand all this gradient-leaf apparatus, let's look at this session:

```
>>> v1 = torch.tensor([1.0, 1.0], requires_grad=True)
>>> v2 = torch.tensor([2.0, 2.0])
```

In the code provided, two tensors were created. One of the tasks involves the calculation of gradients, while the other does not.

```
>>> v_sum = v1 + v2
>>> v_res = (v_sum*2).sum()
>>> v_res

tensor(12., grad_fn=<SumBackward0>)
```

Therefore, we have performed element-wise addition on both vectors (resulting in vector [3, 3]), multiplied each element by two, and then calculated their sum (Casotto & Fantino, 2009). The outcome is a tensor of zero dimensions, with a value of 12. This is basic arithmetic at this point. Now, let us examine the fundamental graph that has been generated by our expressions (Figure 5.3):

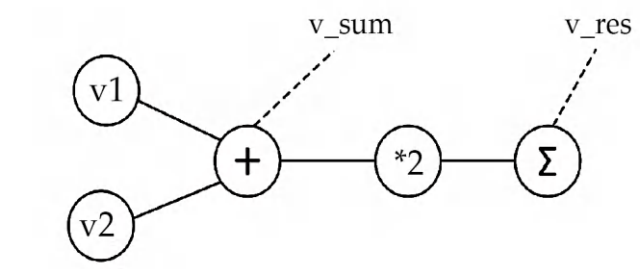


Figure 5.3. Graph representation of the expression.

Source: Ronald Fisher, Creative Commons License.

Upon inspecting the attributes of our tensors, it becomes evident that v1 and v2 stand as the sole leaf nodes. It is worth noting that every variable, with the exception of v2, necessitates the calculation of gradients:

```
>>> v1.is_leaf, v2.is_leaf
(True, True)
>>> v_sum.is_leaf, v_res.is_leaf
(False, False)
>>> v1.requires_grad

True
>>> v2.requires_grad
False
>>> v_sum.requires_grad
True
>>> v_res.requires_grad
True
```

Let's now instruct PyTorch to calculate our graph's gradients:

```
>>> v_res.backward()
>>> v1.grad

tensor([ 2., 2.])
```

By invoking the backward function, PyTorch is able to compute the numerical derivative of the v_res variable in relation to any variable present in our graph. Put simply, how do slight modifications to the v_res variable impact the remaining components of the graph? In this specific example, a gradient value of two indicates that increasing any element of v1 by one will cause the resulting value of v_res to increase by two (Zengerer, 2018).

As stated, PyTorch specifically computes gradients solely for leaf tensors that have the condition requires_grad=True. Unfortunately, when attempting to examine the gradients of v2, no results are obtained:

```
>>> v2.grad
```

Efficiency is a key factor when it comes to computations and memory. In practice, networks often consist of a vast number of optimized parameters, with numerous intermediate operations being executed on them. During gradient descent optimization, the focus lies only in adjusting the model's parameters (weights) based on the gradients of loss. Gradients of intermediate matrix multiplication are not of interest in this process (White et al., 1997).

Certainly, if you are interested in computing the gradients of input data (which can be valuable for generating adversarial examples to deceive existing neural networks or adjusting pre-trained word embeddings), you can easily achieve this by setting requires_grad=True when creating the tensor.

Essentially, there are all the necessary components to optimally execute a specific neural network. This chapter will cover additional functions that serve as higher-level components for constructing neural network designs. They also include popular optimization algorithms and commonly used loss functions. Nevertheless, it is important to remember that these additional features can be effortlessly reintroduced in a DL design. PyTorch is highly favored among deep learning researchers because to its elegant and flexible nature (Lazar & Po, 2015).

5.4. NN BUILDING BLOCKS

The torch.nn package contains numerous pre-defined classes that offer fundamental functional blocks. All of them are specifically designed to facilitate practice, incorporating features such as support for mini-batches, sensible default settings, and suitable weight initialization (Amato et al., 2023). Every module adheres to the convention of being callable, which implies that each instance of a class can be a function when supplied with its parameters.

As an example, the linear class incorporates a feed-forward layer that can include a bias if desired:

```
>>> import torch.nn as nn
>>> l = nn.Linear(2, 5)
>>> v = torch.FloatTensor([1, 2])
>>> l(v)
tensor([ 1.0532,  0.6573, -0.3134,  1.1104, -0.4065], grad_
fn=<AddBackward0>)
```

In this study, a feed-forward layer was created with two inputs and five outputs. The layer was randomly initialized and then applied to a float tensor. You can utilize the nn.module base class, which is the foundation class for all classes in the torch. nn packages, to create your own higher-level NN blocks (Nguyen et al., 2016). In the following part, you will discover how to accomplish this; in the meantime, let us examine practical techniques that every nn.Module child offers. They are listed below:

- The parameters () provide an iterator of all the variables that necessitate gradient computation, specifically the weights of the module.
- The zero_grad () method is in charge of zeroing out every parameter's gradient.
- The function to(device) is used to transfer all module parameters to a specified device, whether it is the CPU or GPU.
- The state_dict () function returns a dictionary containing all the module parameters, making it a valuable tool for model serialization (Tama et al., 2000).
- The load_state_dict () function is used to initialize the module using the state dictionary.

There is a highly convenient class available that enables you to easily combine various layers into a pipeline: Sequential. An effective approach to illustrate Sequential is by providing an example:

```
>>> s = nn.Sequential(
... nn.Linear(2, 5),
... nn.ReLU(),
... nn.Linear(5, 20),
... nn.ReLU(),
... nn.Linear(20, 10),
... nn.Dropout(p=0.3),
... nn.Softmax(dim=1))
>>> s
Sequential(
  (0): Linear(in_features=2, out_features=5, bias=True)
  (1): ReLU()
  (2): Linear(in_features=5, out_features=20, bias=True)
  (3): ReLU()
  (4): Linear(in_features=20, out_features=10, bias=True)
  (5): Dropout(p=0.3)
  (6): Softmax()
)
```

A three-layer neural network with softmax activation on the output is defined here. It is applied along dimension 1, with rectified linear unit (ReLU) dropout and nonlinearities. Let's push something through it:

```
>>> s(torch.FloatTensor([[1,2]]))
tensor([[0.1115, 0.0702, 0.1115, 0.0870, 0.1115, 0.1115, 0.0908,
0.0974, 0.0974, 0.1115]], grad_fn=<SoftmaxBackward>)
```

5.5. CUSTOM LAYERS

In this last section the nn.Module class is briefly described, which serves as the base parent for all neural network building components provided by PyTorch. It serves as more than just a unifying parent for the current layers; it has additional functions and purposes. By inheriting from the nn.Module class, you have the ability to construct custom components that may be combined, reused, and seamlessly integrated into the PyTorch framework (Titanto & Dirgahayu, 2014).

The nn.Module offers extensive functionality to its subclasses:

It keeps a record of all the submodules that are included in the current module. For example, a building block can consist of two feed-forward layers that are utilized to carry out the transformation of the block.

- Functions are available to handle all parameters of the registered submodules. There are several useful methods available for this module. You can access a complete list of parameters using the parameters () method. If you need to reset the gradients to zero, you can use the zero_grads() method. Additionally, you have the flexibility to move the module to either the CPU or GPU by using the to(device) method. If you want to serialize or deserialize the module, you can utilize the state_dict() and load_state_dict() methods. Lastly, if you need to apply any custom transformations, you can do so using the apply () method.
- The convention of applying Modules to data is established. Each module must carry out its data transformation within the forward () method by overriding it (Brahmakshatriya et al., 2024).

Additional functions are available for more advanced use cases, including the option to register a hook function for adjusting module transformation or gradients flow.

These features enable us to seamlessly incorporate our submodels into more advanced models, which proves to be highly valuable when tackling intricate problems. Whether it's a basic linear transformation or a complex ResNet with multiple layers, as long as they adhere to the conventions of nn.Module, both can be managed in a similar manner. This is extremely useful for simplifying code and making it reusable.In order to simplify our lives, the authors of PyTorch have made the building of modules easier by employing thoughtful design and utilizing Python's powerful features. To construct a custom module, one typically need to perform two tasks: registering submodules and implementing the forward() method (Lee et al., 2008).

Now, let's explore how to achieve this for Sequential example from the previous section, but in a manner that is more versatile and can be used repeatedly:

```
class OurModule(nn.Module):
    def __init__(self, num_inputs, num_classes, dropout_prob=0.3):
        super(OurModule, self).__init__()
        self.pipe = nn.Sequential(
            nn.Linear(num_inputs, 5),
            nn.ReLU(),
            nn.Linear(5, 20),
            nn.ReLU(),
            nn.Linear(20, num_classes),
            nn.Dropout(p=dropout_prob),
            nn.Softmax(dim=1)
        )
```

Our module class inherits from nn.Module. Three parameters are passed in the constructor: the input size, the output size, and an optional dropout probability. One of the initial steps is to invoke the constructor of the parent class in order to facilitate its initialization (Schmitt et al., 2008).

In order to proceed, a nn.Sequential object with several layers must be created and assigned to the "pipe" class field. This module will automatically register when we assign a Sequential instance to our field. Like all the other classes in the nn package, nn.Sequential also derives from nn.Module. It does not require any function calls to register. All that is left to do is link our submodules to the appropriate fields. Once the constructor is complete, all the fields will be automatically registered. If desired, there is a function in nn.Module that can be used to register submodules.

```
def forward(self, x):
    return self.pipe(x)
```

Here, our data transformation implementation must take precedence over the forward function. We only need to request that they alter the data because our module is essentially a very basic wrapper around other layers. It is important to mention that when applying a module to the data, we need to treat the module as callable by calling the module instance with the appropriate parameters, instead of using the forward() function of the nn.Module class. The reason for this behavior is that when we consider an instance as callable, nn.Module takes priority by overriding the call () method. This method invokes our forward () function and carries out some complex operations with the nn.Module. Calling forward () directly can disrupt the responsibility of nn.Module, leading to potential negative consequences (Niese et al., 2014).

So, that's the process we must follow to define our own module. Now, let's put it into practice:

```
if __name__ == "__main__":
    net = OurModule(num_inputs=2, num_classes=3)
    v = torch.FloatTensor([[2, 3]])
    out = net(v)
    print(net)
    print(out)
```

We set up our module by specifying the desired number of inputs and outputs. Next, we generate a tensor and instruct our module to apply a transformation to it, using the convention of treating it as a callable object. Next, we display the structure of our network using nn. To create a visually pleasing representation of the inner structure, the module replaces the functions str() and repr(). We finally show the result of the change of the network (Farahani & Safari, 2015). The output that our code should generate should look somewhat like this:

```
rl_book_samples/Chapter03$ python 01_modules.py
OurModule(
  (pipe): Sequential(
    (0): Linear(in_features=2, out_features=5, bias=True)
    (1): ReLU()
    (2): Linear(in_features=5, out_features=20, bias=True)
    (3): ReLU()
    (4): Linear(in_features=20, out_features=3, bias=True)
    (5): Dropout(p=0.3, inplace=False)
    (6): Softmax(dim=1)
  )
)
tensor([[0.5436, 0.3243, 0.1322]], grad_fn=<SoftmaxBackward>)
Cuda's availability is True
Data from cuda: tensor([[0.5436, 0.3243, 0.1322]], device='cuda:0', grad_
fn=<CopyBackwards>)
```

Indeed, all the information regarding the dynamic nature of PyTorch remains accurate. The forward() method is used for each batch of data, allowing you to perform complex data transformations such as hierarchical softmax or applying a random choice of network. There are no limitations on the complexity of the operations you can perform. There is no restriction on the number of arguments that can be passed to your module. If you prefer, you have the option to create a module that includes multiple required parameters and numerous optional arguments, and it will work perfectly (Bichri et al., 2023).

Remember

Understanding gradients in PyTorch is crucial as they enable automatic differentiation, which simplifies the process of optimizing neural network parameters during training by computing how each parameter affects the final output.

ACTIVITY 5.1.

Objective: Gain practical experience in constructing, training, and evaluating a neural network using PyTorch.

Steps:

1. Network Definition:
   - Define a neural network architecture using torch.nn.Module, specifying layers and activation functions.
2. Training and Optimization:
   - Prepare a dataset (e.g., synthetic data or MNIST), convert it into PyTorch tensors, and set up data loaders.
   - Implement a training loop with a chosen loss function (e.g., CrossEntropyLoss) and optimizer (e.g., SGD or Adam).
   - Iterate through batches, compute predictions, calculate loss, backpropagate gradients, and update model parameters.
3. Evaluation and Reflection:
   - Evaluate the trained model on a validation set to assess performance metrics such as accuracy or loss.
   - Reflect on challenges encountered, insights gained, and the practical applications of neural networks in PyTorch.

SUMMARY

- This chapter provides an overview of important concepts for utilizing PyTorch in deep learning applications. It begins with an introduction to tensors, which are the fundamental data structures in PyTorch. It provides a comprehensive overview of how to create, manipulate, and perform operations on tensors. One must grasp the concepts of scalar tensors, tensor operations, and leveraging GPU tensors to enhance performance and expedite computations.
- The chapter also explores gradients and explains their significance in automatic differentiation for optimizing neural network parameters during training. Moreover, it explores into the fundamental elements of neural networks and the development of personalized layers, providing a hands-on method for designing and expanding deep learning structures using PyTorch.

REVIEW QUESTIONS

1. What is a tensor in PyTorch and how is it used in deep learning?
2. Explain the significance of GPU tensors in PyTorch. How do they impact computational performance?
3. What role do gradients play in PyTorch, and how are they computed during neural network training?
4. How can custom layers be implemented and why are they useful in PyTorch?
5. Describe the process of creating and training a neural network using PyTorch.

MULTIPLE CHOICE QUESTIONS

1. What is a tensor in PyTorch?

a. A data structure for storing numbers arranged in multiple dimensions.
b. A type of neural network layer.
c. A deep learning framework.
d. A type of GPU processor.

2. Which of the following is true about GPU tensors in PyTorch?

a. They are used for storing non-numeric data.
b. They accelerate computations due to parallel processing capabilities.
c. They are slower than CPU tensors.
d. They cannot perform tensor operations.

3. What is the purpose of gradients in PyTorch?

a. They are used to create neural network layers.
b. They define the shape of tensor dimensions.
c. They optimize neural network parameters during training.

d. They are used for GPU computations.

4. Which component of PyTorch is responsible for automatic differentiation?

a. Gradients

b. Tensors

c. Neural network layers

d. Custom layers

5. What do custom layers enable in PyTorch?

a. They allow defining new types of GPUs.

b. They facilitate high-level operations like matrix multiplication.

c. They improve tensor creation efficiency.

d. They enable users to create unique neural network architectures.

Answers to Multiple Questions

1. (a); 2. (b); 3. (c); 4. (a); 5. (d).

REFERENCES

1. Amato, D., Lo Bosco, G., & Giancarlo, R. (2023). Neural networks as building blocks for the design of efficient learned indexes. Neural Computing and Applications, 35(29), 21399–21414.
2. Bergmann, P. G. (1968). Comments on the scalar-tensor theory. International Journal of Theoretical Physics, 1, 25–36.
3. Bi, X., Tang, X., Yuan, Y., Zhang, Y., & Qu, A. (2021). Tensors in statistics. Annual Review of Statistics and Its Application, 8(1), 345–368.
4. Bichri, H., Chergui, A., & Hain, M. (2023). Image classification with transfer learning using a custom dataset: Comparative study. Procedia Computer Science, 220, 48–54.
5. Brahmakshatriya, A., Rinard, C., Ghobadi, M., & Amarasinghe, S. (2024). NetBlocks: Staging layouts for high-performance custom host network stacks. Proceedings of the ACM on Programming Languages, 8(PLDI), 467–491.
6. Casotto, S., & Fantino, E. (2009). Gravitational gradients by tensor analysis with application to spherical coordinates. Journal of Geodesy, 83, 621–634.
7. Clarke, F. H. (1975). Generalized gradients and applications. Transactions of the American Mathematical Society, 205, 247–262.
8. Ellis, N., Smith, S. J., & Pitcher, C. R. (2012). Gradient forests: Calculating importance gradients on physical predictors. Ecology, 93(1), 156–168.
9. Farahani, B., & Safari, S. (2015). Cross-layer custom instruction selection to address PVTA variations and soft error. Microelectronics Reliability, 55(11), 2423–2438.
10. Gao, X., Ramezanghorbani, F., Isayev, O., Smith, J. S., & Roitberg, A. E. (2020).

TorchANI: A free and open source PyTorch-based deep learning implementation of the ANI neural network potentials. Journal of Chemical Information and Modeling, 60(7), 3408–3415.

11. Hackbusch, W., & Kühn, S. (2009). A new scheme for the tensor representation. Journal of Fourier Analysis and Applications, 15(5), 706–722.
12. Hasan, K. M., Parker, D. L., & Alexander, A. L. (2001). Comparison of gradient encoding schemes for diffusion-tensor MRI. Journal of Magnetic Resonance Imaging: An Official Journal of the International Society for Magnetic Resonance in Medicine, 13(5), 769–780.
13. Horndeski, G. W. (1974). Second-order scalar-tensor field equations in a four-dimensional space. International Journal of Theoretical Physics, 10, 363–384.
14. Huang, H., Liu, X. Y., Tong, W., Zhang, T., Walid, A., & Wang, X. (2022). High performance hierarchical Tucker tensor learning using GPU tensor cores. IEEE Transactions on Computers, 72(2), 452–465.
15. Julesz, B. (1986). Texton gradients: The texton theory revisited. Biological Cybernetics, 54(4), 245–251.
16. Laporte, F., Dambre, J., & Bienstman, P. (2019). Highly parallel simulation and optimization of photonic circuits in time and frequency domain based on the deep-learning framework PyTorch. Scientific Reports, 9(1), 5918.
17. Lazar, M., & Po, G. (2015). The non-singular green tensor of gradient anisotropic elasticity of Helmholtz type. European Journal of Mechanics-A/Solids, 50, 152–162.
18. Lee, M. Y., Chang, C. C., & Ku, Y. C. (2008). New layer-based imaging and rapid prototyping techniques for computer-aided design and manufacture of custom dental restoration. Journal of Medical Engineering & Technology, 32(1), 83–90.
19. Lee, N., & Cichocki, A. (2018). Fundamental tensor operations for large-scale data analysis using tensor network formats. Multidimensional Systems and Signal Processing, 29, 921–960.
20. Liu, X. Y., Zhang, Z., Wang, Z., Lu, H., Wang, X., & Walid, A. (2022). High-performance tensor learning primitives using GPU tensor cores. IEEE Transactions on Computers, 72(6), 1733–1746.
21. Lyakh, D. I. (2015). An efficient tensor transpose algorithm for multicore CPU, Intel Xeon Phi, and NVidia Tesla GPU. Computer Physics Communications, 189, 84–91.
22. McCrea, W. H., & Mikhail, F. I. (1956). Vector-tetrads and the creation of matter. Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences, 235(1200), 11–22.
23. McRae, A. T., Bercea, G. T., Mitchell, L., Ham, D. A., & Cotter, C. J. (2016). Automated generation and symbolic manipulation of tensor product finite elements. SIAM Journal on Scientific Computing, 38(5), S25–S47.
24. Meneveau, C. (2011). Lagrangian dynamics and models of the velocity gradient tensor in turbulent flows. Annual Review of Fluid Mechanics, 43(1), 219–245.

25. Navarro, C. A., Carrasco, R., Barrientos, R. J., Riquelme, J. A., & Vega, R. (2020). GPU tensor cores for fast arithmetic reductions. IEEE Transactions on Parallel and Distributed Systems, 32(1), 72–84.

26. Nguyen, H. H., Jegathesh, J. J., Takiden, A., Hauenstein, D., Pham, C. T., Le, C. D., & Abram, U. (2016). 2, 6-Dipicolinoylbis (N, N-dialkylthioureas) as versatile building blocks for oligo-and polynuclear architectures. Dalton Transactions, 45(26), 10771–10779.

27. Niese, S., Krüger, P., Kubec, A., Laas, R., Gawlitza, P., Melzer, K., & Zschech, E. (2014). Fabrication of customizable wedged multilayer Laue lenses by adding a stress layer. Thin Solid Films, 571, 321–324.

28. Novac, O. C., Chirodea, M. C., Novac, C. M., Bizon, N., Oproescu, M., Stan, O. P., & Gordan, C. E. (2022). Analysis of the application efficiency of TensorFlow and PyTorch in convolutional neural network. Sensors, 22(22), 8872.

29. Pirani, F. A. E. (1955). On the energy-momentum tensor and the creation of matter in relativistic cosmology. Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences, 228(1175), 455–462.

30. Rivest, J. F., Soille, P., & Beucher, S. (1993). Morphological gradients. Journal of Electronic Imaging, 2(4), 326–336.

31. Schmitt, H., Zeidler, M., Rommel, M., Bauer, A. J., & Ryssel, H. (2008). Custom-specific UV nanoimprint templates and life-time of antisticking layers. Microelectronic Engineering, 85(5–6), 897–901.

32. Segal, I. E. (1956). Tensor algebras over Hilbert spaces. I. Transactions of the American Mathematical Society, 81(1), 106–134.

33. Sehnke, F., Osendorfer, C., Rückstieß, T., Graves, A., Peters, J., & Schmidhuber, J. (2010). Parameter-exploring policy gradients. Neural Networks, 23(4), 551–559.

34. Tama, F., Gadea, F. X., Marques, O., & Sanejouand, Y. H. (2000). Building-block approach for determining low-frequency normal modes of macromolecules. Proteins: Structure, Function, and Bioinformatics, 41(1), 1–7.

35. Titanto, M. T., & Dirgahayu, T. (2014). Google Maps-based geospatial application framework with custom layers management. Applied Mechanics and Materials, 513, 822–826.

36. Vergassola, M., Villermaux, E., & Shraiman, B. I. (2007). 'Infotaxis' as a strategy for searching without gradients. Nature, 445(7126), 406–409.

37. Virtsionis Gkalinikis, N., Nalmpantis, C., & Vrakas, D. (2022). Torch-nilm: An effective deep learning toolkit for non-intrusive load monitoring in PyTorch. Energies, 15(7), 2647.

38. White, C. A., Maslen, P., Lee, M. S., & Head-Gordon, M. (1997). The tensor properties of energy gradients within a non-orthogonal basis. Chemical Physics Letters, 276(1–2), 133–138.

39. Wierstra, D., Förster, A., Peters, J., & Schmidhuber, J. (2010). Recurrent policy gradients. Logic Journal of IGPL, 18(5), 620–634.

40. Yang, W., Li, K., & Li, K. (2019). A pipeline computing method of SpTV for three-order tensors on CPU and GPU. ACM Transactions on Knowledge Discovery from Data (TKDD), 13(6), 1–27.

41. Zengerer, M. (2018). An overview of tensors, gradient and invariant products in imaging and qualitative interpretation. ASEG Extended Abstracts, 2018(1), 1–8.

42. Zhang, E., Hays, J., & Turk, G. (2006). Interactive tensor field design and visualization on surfaces. IEEE Transactions on Visualization and Computer Graphics, 13(1), 94–107.

43. Zhang, L., Qi, L., & Zhou, G. (2014). M-tensors and some applications. SIAM Journal on Matrix Analysis and Applications, 35(2), 437–452.

44. Zhao, M., Li, W., Li, L., Ma, P., Cai, Z., & Tao, R. (2021). Three-order tensor creation and Tucker decomposition for infrared small-target detection. IEEE Transactions on Geoscience and Remote Sensing, 60, 1–16.

CHAPTER

6 Generative Deep Learning

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

- Understand how LSTM networks generate sequence data and their applications.
- Know the importance of sampling strategy in text generation using LSTM models.
- Understand the principles and process of neural style transfer using convnets.
- Understand the roles of content and style loss in neural style transfer.
- Understand the principles of image generation using variational autoencoders (VAEs).
- Learn the concept of latent spaces and how they enable image editing.

KEY TERMS FROM THIS CHAPTER

Concept vector	Generative adversarial networks (GAN)
Language model	Latent space
LSTM (Long short-term memory)	Neural style transfer
Recurrent neural networks (RNN)	Sequence data generation
Softmax temperature	Variational Autoencoders (VAE)

UNIT INTRODUCTION

Beyond reactive skills like operating an automobile and passive ones like object recognition, artificial intelligence can mimic human mental processes and even human artistic endeavors. When it was firstly claimed that most of the cultural content in the not-too-distant future will be produced with significant assistance from AIs, even seasoned machine-learning practitioners expressed complete incredulity (Salakhutdinov, 2015). That was 2014, but later, the initial skepticism diminished rapidly. During the summer of 2015, Google's DeepDream algorithm provided us with a fascinating display of transforming images into a mesmerizing blend of dog eyes and pareidolic artifacts. The following year, in 2016, a fascinating application, the Prisma, helped to effortlessly convert our photos into stunning paintings, each showcasing a unique artistic style. In the summer of 2016, a unique short film called Sunspring was created. The film was directed using a script generated by the Long Short-Term Memory (LSTM) algorithm. The algorithm successfully generated dialogue for the characters in the movie. Besides film, also music has been generated by a neural network (Sousa et al., 2021).

Admittedly, the artistic creations generated by AI thus far have been of relatively low quality. Artificial intelligence is still far from being able to match the creativity and skill of human screenwriters, painters, and composers. However, the goal was never to replace humans. Artificial intelligence is not meant to replace our own intelligence or creativity, but rather to enhance our lives and work with a different kind of intelligence. In various domains, particularly in creative endeavors, humans will utilize AI as a tool to enhance their own skills, resulting in augmented intelligence rather than solely relying on artificial intelligence (Goodfellow et al., 2020).

An important aspect of artistic creation involves developing high technical skills, which is the part of the artistic creation process that many people find unattractive or even unnecessary. That is where artificial intelligence comes in. The statistical structure can be observed in our perceptual modalities, language, and artwork. Mastering this framework is what deep-learning algorithms succeed in. Machine-learning models have the ability to grasp the underlying patterns in images, music, and stories, enabling them to generate new artworks that possess similar characteristics to the ones they have been trained on (Bian & Xie, 2021).

This unit delves into the complex concepts of artificial intelligence, specifically in the fields of text and image generation. This resource explores various topics related to generating diverse sequence data, including Generative Recurrent Networks, Text Generation using LSTM models, and effective sampling strategies. In addition, the chapter covers topics such as Variational Autoencoders for image generation, Neural Style Transfer, and Concept Vectors for image editing (Kallioras & Lagaros, 2020).

6.1. TEXT GENERATION WITH LSTM

Let's start with the application of recurrent neural networks for sequence data generation. Text generation can be used as an example, but the methods can be applied to different kinds of sequence data. For example, one could utilize these techniques to generate new music by working with sequences of musical notes. Similarly, paintings stroke can be generated by analyzing time series of brushstroke data, such as those recorded while an artist paints on an iPad (Li & Zhang, 2021).

Sequence data production has diverse applications that extend beyond the creation of artistic content. It has achieved success in various applications, such as utilizing speech synthesis and generating discourse for chat bots. Google developed the Smart Reply option in 2016, which used similar methods to automatically generate a variety of short responses for emails or text messages (Islam et al., 2019).

6.1.1. Generative Recurrent Networks Brief History

Even in the machine-learning community, relatively few people knew the initials LSTM by the end of 2014. Successful uses of recurrent networks for sequence data production only began to be acknowledged until 2016. Still, these methods have a long history, dating back to the development of the LSTM algorithm in 1997. In its early phases, the new algorithm was used to produce text character by character (Wang et al., 2017).

In 2002, Douglas Eck, a researcher working at Schmidhuber's laboratory in Switzerland, conducted experiments with Long Short-Term Memory (LSTM) in the domain of music production. The results were really encouraging. Eck presently holds a position as a researcher at Google Brain, where he founded a research group named Magenta in the year 2016. The primary objective of Magenta is to utilize cutting-edge deep-learning techniques to create captivating music. Occasionally, it can take a significant amount of time for promising concepts to gain traction (Briot, 2021).

Alex Graves made notable advancements in the field of recurrent networks and their utilization in producing sequence data during the late 2000s and early 2010s. Some consider his 2013 research on utilizing recurrent mixture density networks to produce handwriting that resembles human penmanship using time series of pen positions as a significant breakthrough (Creswell et al., 2018).

At that moment, the use of neural networks sparked the idea of machines with the ability to dream, which greatly influenced the development of Keras. Graves discreetly included a remark in a 2013 arXiv preprint server: "Generating sequential data is the

closest computers come to experiencing dreams." Years later, many of these advancements are now widespread, but back then, it was hard not to be amazed by the potential after witnessing Graves's impressive demonstrations (Gui et al., 2021).

Over time, recurrent neural networks have proven to be highly effective in various domains such as dialogue generation, music generation, speech synthesis, image generation, and molecule design. They were even utilized to create a screenplay that was subsequently brought to life by a cast of live actors (Pater, 2019).

Did you know?

LSTM networks are often used in text generation tasks because of their capacity to capture long-range dependencies in sequences, enabling the production of coherent text paragraphs.

6.1.2. Process of Generating Sequence Data

When it comes to producing sequence data in deep learning, a commonly used method is to train a network, typically an RNN or a convnet, to forecast the subsequent token or tokens in a sequence. This is accomplished by utilizing the tokens that come before as input. When provided with the input "the cat is on the ma," the network is instructed to forecast the target t, which represents the subsequent character. In the context of textual data, tokens typically refer to words or characters. A language model is a type of neural network that can accurately estimate the probability of the next token based on the preceding tokens. A language model is a representation of the fundamental structure of language, particularly its statistical patterns (Yang et al., 2021).

After obtaining a well-trained language model, it becomes possible to generate new sequences by sampling from it. This is achieved by supplying a starting sequence of text, referred to as conditioning data, and directing the model to produce the subsequent character or word. Multiple tokens can be generated simultaneously, which is quite remarkable (Soibelman & Kim, 2002). The resulting output is then incorporated into the input data, and this procedure is repeated several times, as displayed in Figure 6.1.

This loop enables the generation of sequences of varying lengths that accurately capture the underlying structure of the data used to train the model. These sequences closely resemble sentences written by humans. In the example provided in this section, we will explore the usage of an LSTM layer. It will be trained to predict the following character, N + 1, by feeding it strings of N characters taken from a text corpus. The output of the model will be a probability distribution for the next character, which is a softmax over all possible characters. The name "character-level neural language model" refers to this LSTM (Nekrutenko & Taylor, 2012).

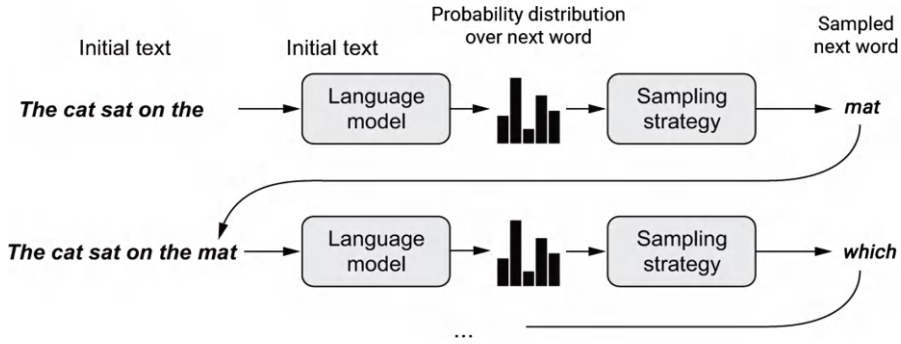


Figure 6.1. Illustration of generating text character by character using a language model.

Source: François Chollet, Creative Commons License.

6.1.3. The Importance of the Sampling Strategy

In terms of text generation, the character that is chosen next is quite important. Greedy sampling is a fundamental technique that involves repeatedly choosing the most likely character to show up next. However, this method leads to repetitive, foreseeable sequences that lack the appearance of cohesive language. A more interesting method takes slightly unexpected decisions: it incorporates randomness into the sampling process, by selecting characters based on their probability distribution for the next character (Hirzel & Guisan, 2002). This technique is referred to as stochastic sampling (remember that stochasticity refers to randomness in this domain).

In this scenario, if the probability of e being the next character is 0.3, as per the model, you will select it 30% of the time. It is important to mention that greedy sampling can also be viewed as sampling from a probability distribution. In this distribution, a single character is assigned a probability of 1, while the probabilities of all other characters are set to 0.

Sampling from the softmax output of the model is quite interesting. It enables the possibility of sampling even improbable characters, resulting in sentences that are more intriguing and occasionally showcasing creativity by generating new, authentic-sounding words that were not present in the training data. However, there is a drawback to this approach: it lacks a method to regulate the level of randomness in the sampling procedure (Cousens et al., 2002).

What is the reason for wanting more or less randomness? Let's explore a hypothetical situation: completely random sampling. Here, the next character is selected randomly from a uniform probability distribution, ensuring that each character has an equal chance of being chosen. This scheme demonstrates a significant degree of randomness; in simpler terms, the probability distribution linked to it possesses a high level of entropy. Obviously, it will not generate anything that grabs attention (Ludwig et al., 2022).

On the opposite end of the spectrum, greedy sampling fails to generate anything of interest. It lacks any element of randomness, resulting in a probability distribution with minimal entropy. Sampling from the probability distribution generated by the model's softmax function provides a balanced approach between the two extremes. However, there are many

intermediate points with different levels of entropy that you may find interesting to explore (Franco et al., 2005).

Reducing the level of randomness in the generated sequences can lead to a more structured outcome, making them appear more realistic. On the other hand, increasing the level of randomness can result in sequences that are more unexpected and imaginative. When experimenting with generative models, it is beneficial to vary the level of randomness in the generation process. As humans, we have the final decision on what we find interesting in the data generated. However, determining the level of interest is a subjective matter, and it is impossible to predict where the optimal point of entropy will be (Madrid & Zayas, 2007).

In order to control the degree of unpredictability in the sampling procedure, a softmax temperature parameter is incorporated. This parameter establishes the degree of surprise or predictability in choosing the subsequent character by calculating the entropy of the probability distribution used for sampling. Upon receiving a temperature value, a new probability distribution is computed using the original distribution, which is the softmax output of the model. This is accomplished by modifying the weights in the subsequent manner (Erbaź & Christie, 2007).

```
import numpy as np
def reweight_distribution(original_distribution, temperature=0.5):
    distribution = np.log(original_distribution) / temperature
    distribution = np.exp(distribution)
    return distribution / np.sum(distribution)
```

original_distribution is a 1D Numpy array of probability values that must sum to 1.
temperature is a factor quantifying the entropy of the output distribution.

Returns a reweighted version of the original distribution. The sum of the distribution may no longer be 1, so you divide it by its sum to obtain the new distribution.

High temperatures lead to sampling distributions with increased entropy, resulting in more unexpected and disorganized generated data. On the other hand, data generated at lower temperatures is more predictable and less random (see Figure 6.2).

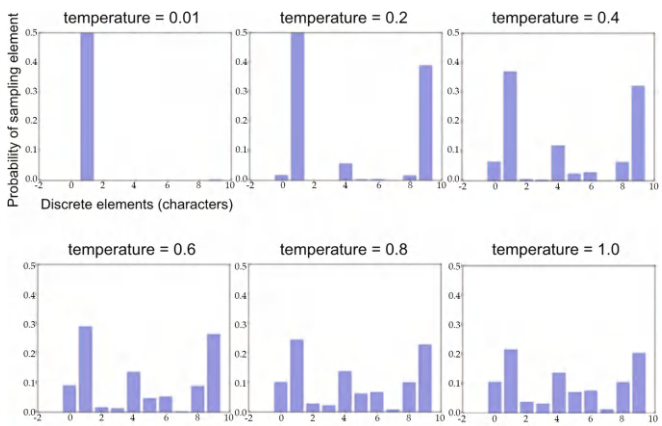


Figure 6.2. One probability distribution with a distinct reweighting. High temperature = more random; Low temperature = more deterministic.

Source: Bradley, D., Creative Commons License.

6.2. NEURAL STYLE TRANSFER

Neural style transfer, an important progress in picture alteration propelled by deep learning, was first presented by Leon Gatys et al. in the summer of 2015. The neural style transfer algorithm has seen significant advancements and led to a multitude of versions since its initial release. As a result, it has become a popular feature in various smartphone photo applications. To maintain simplicity, this section will focus exclusively on the formulation described in the original publication (Jing et al., 2019).

Neural style transfer is the process of applying the artistic style of a reference image to a target image, while maintaining the original content of the target image. This is an example shown in Figure 6.3.

Figure 6.3. Illustration of style transfer example.

Source: D. Koller, Creative Commons License.

Within this context, style refers to the complex details such as colors, textures, and visual patterns found in the image, observed at different spatial scales. On the other hand, content refers to the overall macrostructure of the image at a higher level. As an example, the artistic style shown in Figure 6.4 (featuring Starry Night by Vincent Van Gogh) is characterized by blue-and-yellow circular brushstrokes, while the Tübingen photograph showcases buildings as its subject matter (Singh et al., 2021).

The concept of style transfer, closely linked to texture generation, has been a topic of interest in the image-processing community for a long time before the born of neural style transfer in 2015. However, the recent advancements in deep learning have revolutionized style transfer, surpassing the capabilities of traditional computer-vision techniques. This has sparked a remarkable surge in creative applications of computer vision (Cai et al., 2023).

The fundamental concept behind implementing style transfer is the core principle of all deep-learning algorithms: defining a loss function to specify the desired outcome

and minimizing this loss. You have a clear goal in mind: preserving the essence of the original image while incorporating the artistic style of the reference image (Han et al., 2018). If content and style could be precisely defined mathematically, then the ideal loss function to minimize would be as follows:

```
loss = distance(style(reference_image) - style(generated_image)) +
       distance(content(original_image) - content(generated_image))
```

In this context, distance is considered a standard or typical expectation. The L2 norm is a mathematical function that measures the magnitude of a vector. In the context of image analysis, "content" refers to a function that examines an image and produces a representation of its visual information, while "style" refers to a function that examines an image and produces a representation of its artistic characteristics (Cheng et al., 2019).

By reducing this loss, the generated image adopts a style that closely resembles the reference image, while preserving the original content of the generated image. This enables us to accomplish the intended style transfer, in accordance with our criteria (Virtusio et al., 2021).

Gatys made a fundamental observation that deep convolutional neural networks provide a mathematical definition for style and content functions. Let's explore.

6.2.1. The Content Loss

It is well-known that activations from lower layers in a network provide specific information about the image, but activations from higher layers contain broader and more conceptual information. Put simply, convolutional neural networks (convnet's) layers break down the image into different spatial scales through their activations. Therefore, it is expected that the higher layers in a convnet would capture the broader and more conceptual aspects of a picture (Hayn, 1995).

Finding the L2 norm between a higher layer in a pretrained convolutional neural network's activations is one potential solution for content loss. This is done by comparing the activations of the target image with those of the generated image. Ensuring that, when observed from the higher layer, the produced image will bear resemblance to the initial target image. If we consider that the higher layers of a convnet accurately represent the content of the input images, then this method effectively preserves the image content (Lyzwinski, 2014).

6.2.2. The Style Loss

While the style loss, as defined by Gatys, makes use of many layers of a convolutional neural network, the content loss only requires one higher layer. The goal of the style loss is to accurately capture the visual attributes of the style reference image at various spatial scales extracted by the convolutional neural network, rather than only emphasizing one scale. Gatys uses the Gram matrix of a layer's activations to calculate the style loss. The Gram matrix is obtained by taking the dot product of the feature maps of a

specific layer (Jiang et al., 2021). The inner product serves as a means for showing the interrelationships among the features of the layer. The feature correlations in question capture statistical information about the patterns observed at a particular spatial scale. These patterns are known to correspond to the visual textures that are present at this scale, as observed through empirical evidence.

As a result, the style loss concentrates on preserving internal linkages that are consistent between the generated image and the style-reference image's activations of different layers. This guarantees that the textures that are present at different spatial scales appear consistently in the style-reference image and the generated image (Ye et al., 2020).

In short, a pre-trained convnet can be utilized to establish a loss function that achieves the following objectives:

- To maintain the integrity of the content, it is important to maintain constant high-level layer activations between the target content image and the generated image. It is crucial for the convolutional neural network (convnet) to accurately recognize that both the target picture and the produced image contain the same content.
- Maintain comparable correlations between activations for both lower-level and higher-level layers to ensure consistency. The generation of the image and the style-reference image must have comparable textures at different spatial scales to effectively capture feature correlations (Liu et al., 2021).

Now, let's analyze a Keras implementation of the neural style transfer technique from 2015. It is somewhat similar to the DeepDream implementation described in the previous section, as you will see.

6.2.2.1. Transfer of Neural Styles in Keras

Any pre-trained convnet can be used to implement neural style transfer. Here, you will utilize Gatys's VGG19 network.

This is the standard procedure:

- Develop a network that can efficiently compute VGG19 layer activations for the target image, style-reference image, and generated image simultaneously.
- Use the layer activations calculated from these three photos to define the previously described loss function. The objective is to decrease this function in order to achieve style transmission.
- Implement a gradient-descent algorithm to minimize the loss function (Ma et al., 2024).

First, let's define the paths to the target image and the style-reference image. You'll eventually resize all the processed photos to a shared height of 400 pixels to ensure that they are all roughly the same size (large size differences make style transfer more challenging).

```
from keras.preprocessing.image import load_img, img_to_array

target_image_path = 'img/portrait.jpg'
style_reference_image_path = 'img/transfer_style_reference.jpg'

width, height = load_img(target_image_path).size
img_height = 400
img_width = int(width * img_height / height)
```

Path to the image you want to transform

Path to the style image

Dimensions of the generated picture

For loading, preparing, and post processing the pictures that enter and exit the VGG19 convolutional neural network, you require a few auxiliary functions.

```
import numpy as np
from keras.applications import vgg19

def preprocess_image(image_path):
    img = load_img(image_path, target_size=(img_height, img_width))
    img = img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = vgg19.preprocess_input(img)
    return img
```

```
def deprocess_image(x):
    x[:, :, 0] += 103.939
    x[:, :, 1] += 116.779
    x[:, :, 2] += 123.68
    x = x[:, :, ::-1]
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

Zero-centering by removing the mean pixel value from ImageNet. This reverses a transformation done by vgg19.preprocess_input.

Converts images from 'BGR' to 'RGB'. This is also part of the reversal of vgg19.preprocess_input.

Let's configure the network for VGG19. It receives three input images: the style reference image, the target image, and a placeholder that contains the generated image. A placeholder is a symbolic tensor that uses Numpy arrays to receive values from outside sources. The created picture's placeholder variables will change over time, but the target image and style-reference stay constant and are therefore defined using K constant (Zhou et al., 2022).

```
from keras import backend as K

target_image = K.constant(preprocess_image(target_image_path))
style_reference_image = K.constant(preprocess_image(style_reference_image_path))
combination_image = K.placeholder((1, img_height, img_width, 3))

input_tensor = K.concatenate([target_image,
                              style_reference_image,
                              combination_image], axis=0)

model = vgg19.VGG19(input_tensor=input_tensor,
                    weights='imagenet',
                    include_top=False)
print('Model loaded.')
```

Placeholder that will contain the generated image

Combines the three images in a single batch

Builds the VGG19 network with the batch of three images as input. The model will be loaded with pretrained ImageNet weights.

Now let's determine the content loss, which will ensure that the generated image and the target image are seen similarly by the top layer of the VGG19 convolutional network.

```
def content_loss(base, combination):
    return K.sum(K.square(combination - base))
```

The lack of style comes next. The Gram matrix of an input matrix, which depicts the correlations discovered in the original feature matrix, is computed using an auxiliary function (Jackson et al., 2019).

```
def gram_matrix(x):
    features = K.batch_flatten(K.permute_dimensions(x, (2, 0, 1)))
    gram = K.dot(features, K.transpose(features))
    return gram


def style_loss(style, combination):
    S = gram_matrix(style)
    C = gram_matrix(combination)
    channels = 3
    size = img_height * img_width
    return K.sum(K.square(S - C)) / (4. * (channels ** 2) * (size ** 2))
```

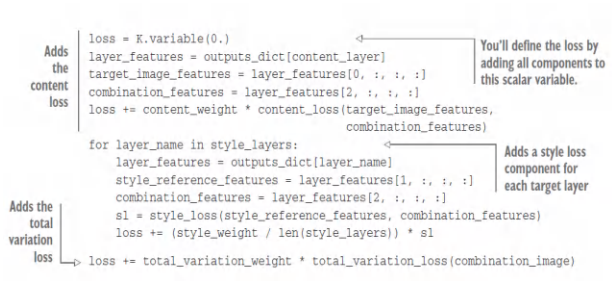
Furthermore, an additional loss component, known as the total variation loss, is included. This loss operates on the individual pixels of the resulting combination image. It enhances the spatial coherence in the resulting image, hence preventing highly pixelated results. One approach to understanding it is as a type of regularization loss.

```
def total_variation_loss(x):
    a = K.square(
        x[:, :img_height - 1, :img_width - 1, :] -
        x[:, 1:, :img_width - 1, :])
    b = K.square(
        x[:, :img_height - 1, :img_width - 1, :] -
        x[:, :img_height - 1, 1:, :])
    return K.sum(K.pow(a + b, 1.25))
```

These three losses are combined and given different weights to determine the overall loss to minimize. For calculating the content loss, only the block5_conv2 layer is utilized as the upper layer. However, when it comes to calculating the style loss, a range of layers, including both low-level and high-level levels, are used. Lastly, the total variation loss at the end is included (Barzilay et al., 2021).

To fine-tune the content loss's contribution to the overall loss, the content weight coefficient should be modified based on the style-reference image and content picture under use. The desired content will appear more prominently in the final image if the content is given a higher weight.

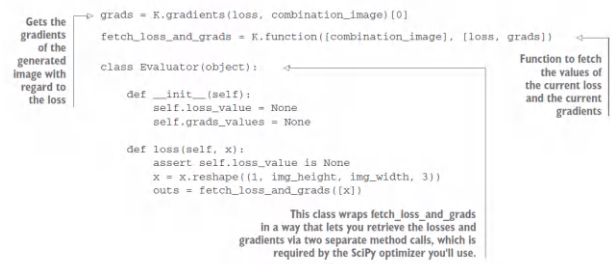
```
Dictionary that maps layer
names to activation tensors
outputs_dict = dict([(layer.name, layer.output) for layer in model.layers])
content_layer = 'block5_conv2'          Layer used for content loss
style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',         Layers used for style loss
                'block4_conv1',
                'block5_conv1']
total_variation_weight = 1e-4
style_weight = 1.                       Weights in the weighted average
content_weight = 0.025                  of the loss components
```



At last, the gradient-descent process will be established. In the original Gatys et al. paper, the optimization process is carried out using the L-BFGS algorithm, which will be utilized in this case as well. Although the L-BFGS algorithm is readily available in SciPy, it is worth noting that there are a couple of minor drawbacks associated with the implementation in SciPy:

- It is necessary to provide two different functions: one for the value of the loss function and another for the value of the gradients.
- This method can only be used with two-dimensional vectors, while you have a three-dimensional image array.

Computing the value of the loss function and the value of the gradients separately would be wasteful due to unnecessary computation. This would result in a nearly twofold decrease in speed compared to computing them together. To address this problem, a Python class called Evaluator is created that efficiently computes both the loss value and the gradients value. Upon the initial invocation of the class, it will provide the loss value and subsequently maintain the gradients for next invocations (Olson et al., 2021).




```
        loss_value = outs[0]
        grad_values = outs[1].flatten().astype('float64')
        self.loss_value = loss_value
        self.grad_values = grad_values
        return self.loss_value

    def grads(self, x):
        assert self.loss_value is not None
        grad_values = np.copy(self.grad_values)
        self.loss_value = None
        self.grad_values = None
        return grad_values

evaluator = Evaluator()
```

Finally, SciPy's L-BFGS algorithm can be used to carry out the gradient-ascent

process. After each algorithm iteration, there is a choice to save the currently generated image. It is worth noting that in this context, a single iteration corresponds to 20 steps of gradient ascent (Yoo et al., 2021).

```
from scipy.optimize import fmin_l_bfgs_b
from scipy.misc import imsave
import time

result_prefix = 'my_result'
iterations = 20

x = preprocess_image(target_image_path)
x = x.flatten()
for i in range(iterations):
    print('Start of iteration', i)
    start_time = time.time()
    x, min_val, info = fmin_l_bfgs_b(evaluator.loss,
                                     x,
                                     fprime=evaluator.grads,
                                     maxfun=20)
    print('Current loss value:', min_val)
    img = x.copy().reshape((img_height, img_width, 3))
    img = deprocess_image(img)
    fname = result_prefix + '_at_iteration_%d.png' % i
    imsave(fname, img)
    print('Image saved as', fname)
    end_time = time.time()
    print('Iteration %d completed in %ds' % (i, end_time - start_time))
```

This is the initial state: the target image.

You flatten the image because scipy.optimize.fmin_l_bfgs_b can only process flat vectors.

Runs L-BFGS optimizatio over the pixels of the generated image to minimize the neural style loss. Note that you have to pass the function that computes the loss and th function that computes the gradients as two separate arguments.

Saves the current generated image.

Figure 6.4 depicts the resultant image. It is important to note that this approach essentially accomplishes image retexturing, or the transfer of textures. It is most effective when used with style reference images that have distinct textures and strong self-similarity, and with content targets that do not need a lot of detail to be identifiable. It is generally unable to accomplish complex tasks such as translating the artistic style of one portrait to another. The method exhibits a greater resemblance to classical signal processing rather than artificial intelligence; hence it should not be anticipated to function miraculously.

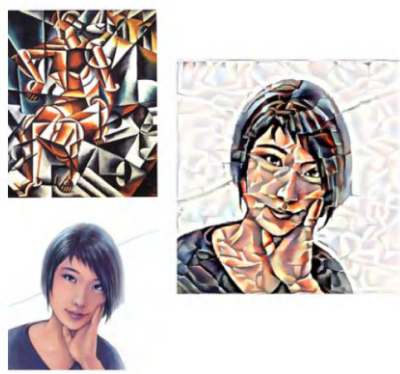


Figure 6.4. Illustration of iterative image.

Source: Behnke, S., Creative Commons License.

In addition, it is worth mentioning that the execution time of this style-transfer algorithm is quite long. However, the setup's transformation is simple and can be easily learned by a compact and efficient feed forward convnet, given the availability of suitable training data. Efficient style transfer can be accomplished by initially investing significant computational resources in generating training examples for a specific style-reference image, as described in the method outlined here. Subsequently, a simple convnet can be trained to acquire the ability to perform this style-specific transformation. After completing the necessary steps, the process of stylizing an image becomes incredibly fast. It simply involves performing a forward pass of a small convnet (Tripp et al., 2020).

6.3. GENERATING IMAGES WITH VARIATIONAL AUTOENCODERS

Exploring the vast potential of creative AI, one of the most popular and successful applications involves sampling from a latent space of images. This process allows for the creation of entirely new images or the editing of existing ones, opening up exciting possibilities for artistic expression. It is important to understand some fundamental concepts related to image generation. Implementation of Variational autoencoders (VAEs) and generative adversarial networks (GANs), are two main approaches in this subject. These methods can be used with text, music, and sound among other kinds of data. However, the primarily focus is on the application of these techniques to images, as they have yielded some interesting results in practice (Cristovao et al., 2020).

6.3.1. Sampling From Latent Spaces of Images

The main concept behind image generation involves creating a compact representation space where each point can be transformed into a visually convincing image.

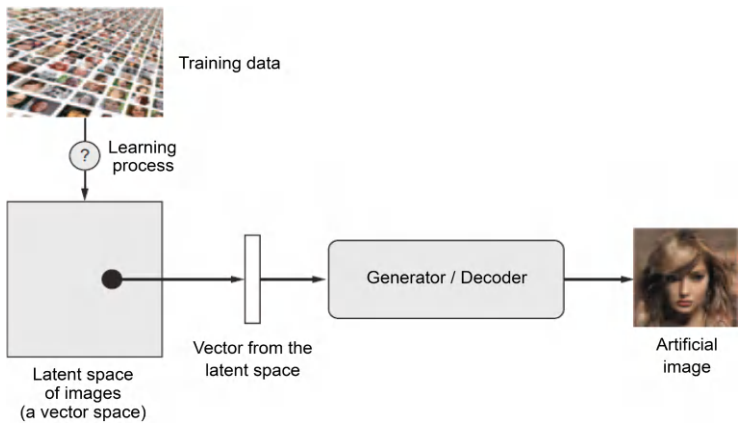


Figure 6.5. Illustration of learning of an image latent vector space and then sampling fresh images from it.

Source: Belkin, M., Creative Commons License.

For GANs, the module that can accomplish this mapping is called a generator; for VAEs, it is called a decoder. It takes a latent point as input and outputs an image, which is a grid of pixels. One can take intentional or random samples of points from a latent space once it has been established (Vahdat et al., 2021). Then original, never-before-seen visuals are created by mapping these points to image space (see Figure 6.5).

Figure 6.6. Illustration of tom white created a continuous space of faces using VAEs.

Source: Jaitly, N., Creative Commons License.

Remember

Effective sampling strategies in sequence generation are crucial for balancing exploration (generating novel sequences) and exploitation (choosing high-quality sequences), ensuring diverse and meaningful outputs.

6.3.2. Concept Vectors for Image Editing

The concept remains unchanged: when provided with a latent space of representations, or an embedding space, specific directions within the space can capture significant dimensions of variation in the original data. Within the context of image analysis, it is possible to identify a specific vector, let's call it "s," that represents a smile. This means that if we have a latent point "z" that represents a particular face, adding the smile vector "s" to "z" will result in a new latent point that represents the same face, but with a smile (Hu et al., 2013). After discovering a suitable vector, images can be altered by projecting them into a concealed space, making significant changes to their representation, and then projecting them back into image space through decoding. Concept vectors exist for a number of picture space variation dimensions. For example, when it comes to faces, there are vectors that can be used to change a male face into a feminine face, take off spectacles, add sunglasses, and more. Tom White, from Victoria University School of Design in New Zealand, discovered a concept vector called "smile" (Figure 6.7). This vector was obtained by training VAEs on a dataset of celebrity faces, specifically the CelebA dataset (Farbman et al., 2010).

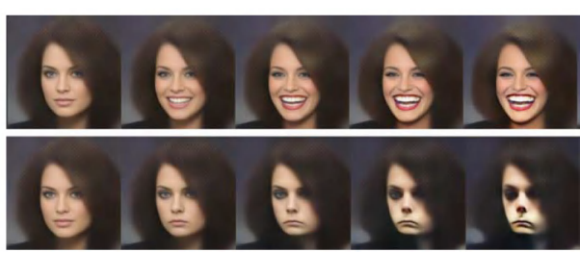


Figure 6.7. The smile vector.

Source: Dauphin, Y. Creative Commons License.

ACTIVITY 6.1.

Objective: To gain practical understanding of Generative Deep Learning techniques.

Materials Needed:

- Python environment with TensorFlow or PyTorch;
- Jupyter notebook or IDE for coding.

Steps:

1. `Implementing LSTM for Text Generation:
   - Use TensorFlow or PyTorch to build an LSTM model.
   - Train the model on a small dataset (e.g., text from a book or dataset).
   - Generate new text sequences and analyze the results.
2. Exploring Neural Style Transfer:
   - Implement a basic Neural Style Transfer algorithm.
   - Experiment with different content and style images.
   - Discuss the impact of content loss and style loss on the generated images.

SUMMARY

- The chapter explores different techniques and applications within the field. The exploration begins with delving into Text Generation using LSTM models and offers a historical perspective on Generative Recurrent Networks.
- The chapter highlights the significance of sampling strategies in generating sequence data and introduces Neural Style Transfer, explaining techniques like Content Loss and Style Loss. In addition, it explores the utilization of Variational Autoencoders to create images and explores different methods for extracting information from the concealed spaces of images.
- Finally, it concludes by exploring the use of concept vectors in image editing, providing valuable insights into the manipulation and editing of images using learned concepts.

REVIEW QUESTIONS

1. What is the role of LSTM models in text generation within Generative Deep Learning?
2. How has the development of Generative Recurrent Networks evolved over time?
3. Why sampling strategies are important in sequence generation, and what are some common approaches?
4. Explain Neural Style Transfer and the roles of content loss and style loss in image transformation.
5. How do Variational Autoencoders (VAEs) contribute to image generation in Generative Deep Learning?
6. What are concept vectors in image editing, and how are they used to manipulate image features?

MULTIPLE CHOICE QUESTIONS

1. Which architecture is commonly used for Text Generation in Generative Deep Learning?

a. LSTM
b. CNN
c. Transformer
d GAN

2. What are the primary components used in Neural Style Transfer?

a. Encoder and Decoder
b. Discriminator and Generator
c. Content Loss and Style Loss
d. Activation and Pooling Layers

3. **Variational Autoencoders (VAEs) are useful for:**
   a. Image classification
   b. Image generation
   c. Image segmentation
   d. Image enhancement
4. **Sampling strategies in sequence generation refer to:**
   a. Randomly selecting data points
   b. Selecting the most probable sequences
   c. Removing outliers from the dataset
   d. Balancing exploration and exploitation
5. **Concept vectors in image editing allow for:**
   a. Enhancing resolution of images
   b. Adding new objects to images
   c. Manipulating specific image features
   d. Filtering noise from images

Answers to Multiple Questions

1. (a); 2. (c); 3. (b); 4. (d); 5. (c).

REFERENCES

1. Barzilay, N., Shalev, T. B., & Giryes, R. (2021). MISS GAN: A multi-IlluStrator style generative adversarial network for image to illustration translation. Pattern Recognition Letters, 151, 140–147.
2. Bian, Y., & Xie, X. Q. (2021). Generative chemistry: Drug discovery with deep learning generative models. Journal of Molecular Modeling, 27, 1–18.
3. Briot, J. P. (2021). From artificial neural networks to deep learning for music generation: History, concepts and trends. Neural Computing and Applications, 33(1), 39–65.
4. Cai, Q., Ma, M., Wang, C., & Li, H. (2023). Image neural style transfer: A review. Computers and Electrical Engineering, 108, 108723.
5. Cheng, M. M., Liu, X. C., Wang, J., Lu, S. P., Lai, Y. K., & Rosin, P. L. (2019). Structure-preserving neural style transfer. IEEE Transactions on Image Processing, 29, 909–920.
6. Cousens, R. D., Brown, R. W., McBratney, A. B., Whelan, B., & Moerkerk, M. (2002). Sampling strategy is important for producing weed maps: A case study using kriging. Weed Science, 50(4), 542–546.

7. Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., & Bharath, A. A. (2018). Generative adversarial networks: An overview. IEEE Signal Processing Magazine, 35(1), 53–65.

8. Erbaş, D., & Christie, M. (2007). How does sampling strategy affect uncertainty estimations? Oil & Gas Science and Technology-Revue de l'IFP, 62(2), 155–167.

9. Farbman, Z., Fattal, R., & Lischinski, D. (2010). Diffusion maps for edge-aware image editing. ACM Transactions on Graphics (TOG), 29(6), 1–10.

10. Franco, J., Crossa, J., Taba, S., & Shands, H. (2005). A sampling strategy for conserving genetic diversity when forming core subsets. Crop Science, 45(3), 1035–1044.

11. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., & Bengio, Y. (2020). Generative adversarial networks. Communications of the ACM, 63(11), 139–144.

12. Gui, J., Sun, Z., Wen, Y., Tao, D., & Ye, J. (2021). A review on generative adversarial networks: Algorithms, theory, and applications. IEEE Transactions on Knowledge and Data Engineering, 35(4), 3313–3332.

13. Han, W., Feng, R., Wang, L., & Cheng, Y. (2018). A semi-supervised generative framework with deep learning features for high-resolution remote sensing image scene classification. ISPRS Journal of Photogrammetry and Remote Sensing, 145, 23–43.

14. Hayn, C. (1995). The information content of losses. Journal of Accounting and Economics, 20(2), 125–153.

15. Hirzel, A., & Guisan, A. (2002). Which is the optimal sampling strategy for habitat suitability modelling. Ecological Modelling, 157(2–3), 331–341.

16. Hu, S. M., Xu, K., Ma, L. Q., Liu, B., Jiang, B. Y., & Wang, J. (2013). Inverse image editing: Recovering a semantic editing history from a before-and-after image pair. ACM Transactions on Graphics (TOG), 32(6), 1–11.

17. Islam, M. S., Mousumi, S. S. S., Abujar, S., & Hossain, S. A. (2019). Sequence-to-sequence Bangla sentence generation with LSTM recurrent neural networks. Procedia Computer Science, 152, 51–58.

18. Jackson, P. T., Abarghouei, A. A., Bonner, S., Breckon, T. P., & Obara, B. (2019). Style augmentation: Data augmentation via style randomization. In CVPR Workshops (Vol. 6, pp. 10–11).

19. Jiang, S., Li, J., & Fu, Y. (2021). Deep learning for fashion style generation. IEEE Transactions on Neural Networks and Learning Systems, 33(9), 4538–4550.

20. Jing, Y., Yang, Y., Feng, Z., Ye, J., Yu, Y., & Song, M. (2019). Neural style transfer: A review. IEEE Transactions on Visualization and Computer Graphics, 26(11), 3365–3385.

21. Kallioras, N. A., & Lagaros, N. D. (2020). DzAIN: Deep learning based generative design. Procedia Manufacturing, 44, 591–598.

22. Li, L., & Zhang, T. (2021). Research on text generation based on LSTM. International Core Journal of Engineering, 7(5), 525–535.

23. Liu, R., Sisman, B., Gao, G., & Li, H. (2021). Expressive TTS training with frame and style reconstruction loss. IEEE/ACM Transactions on Audio, Speech, and Language Processing, 29, 1806–1818.

24. Ludwig, M., Enders, D., Basedow, F., Walker, J., & Jacob, J. (2022). Sampling strategy, characteristics and representativeness of the InGef research database. Public Health, 206, 57–62.

25. Lyzwinski, L. N. (2014). A systematic review and meta-analysis of mobile devices and weight loss with an intervention content analysis. Journal of Personalized Medicine, 4(3), 311–385.

26. Ma, C., Sun, Z., & Ruan, C. (2024). Style creation: Multiple styles transfer with incremental learning and distillation loss. Multimedia Tools and Applications, 83(10), 28341–28356.

27. Madrid, Y., & Zayas, Z. P. (2007). Water sampling: Traditional methods and new approaches in water sampling strategy. TrAC Trends in Analytical Chemistry, 26(4), 293–299.

28. Nekrutenko, A., & Taylor, J. (2012). Next-generation sequencing data interpretation: Enhancing reproducibility and accessibility. Nature Reviews Genetics, 13(9), 667–672.

29. Olson, M. L., Khanna, R., Neal, L., Li, F., & Wong, W. K. (2021). Counterfactual state explanations for reinforcement learning agents via generative deep learning. Artificial Intelligence, 295, 103455.

30. Pater, J. (2019). Generative linguistics and neural networks at 60: Foundation, friction, and fusion. Language, 95(1), e41–e74.

31. Ristovao, P., Nakada, H., Tanimura, Y., & Asoh, H. (2020). Generating in-between images through learned latent space representation using variational autoencoders. IEEE Access, 8, 149456–149467.

32. Salakhutdinov, R. (2015). Learning deep generative models. Annual Review of Statistics and Its Application, 2(1), 361–385.

33. Singh, A., Jaiswal, V., Joshi, G., Sanjeeve, A., Gite, S., & Kotecha, K. (2021). Neural style transfer: A critical review. IEEE Access, 9, 131583–131613.

34. Soibelman, L., & Kim, H. (2002). Data preparation process for construction knowledge generation through knowledge discovery in databases. Journal of Computing in Civil Engineering, 16(1), 39–48.

35. Sousa, T., Correia, J., Pereira, V., & Rocha, M. (2021). Generative deep learning for targeted compound design. Journal of Chemical Information and Modeling, 61(11), 5343–5361.

36. Tripp, A., Daxberger, E., & Hernández-Lobato, J. M. (2020). Sample-efficient optimization in the latent space of deep generative models via weighted retraining. Advances in Neural Information Processing Systems, 33, 11259–11272.

37. Vahdat, A., Kreis, K., & Kautz, J. (2021). Score-based generative modeling in latent space. Advances in Neural Information Processing Systems, 34, 11287–11302.

38. Virtusio, J. J., Ople, J. J. M., Tan, D. S., Tanveer, M., Kumar, N., & Hua, K. L. (2021). Neural style palette: A multimodal and interactive style transfer from a single style image. IEEE Transactions on Multimedia, 23, 2245–2258.

39. Wang, K., Gou, C., Duan, Y., Lin, Y., Zheng, X., & Wang, F. Y. (2017). Generative adversarial networks: Introduction and outlook. IEEE/CAA Journal of Automatica Sinica, 4(4), 588–598.

40. Yang, C., Chowdhury, D., Zhang, Z., Cheung, W. K., Lu, A., Bian, Z., & Zhang, L. (2021). A review of computational tools for generating metagenome-assembled genomes from metagenomic sequencing data. Computational and Structural Biotechnology Journal, 19, 6301–6314.

41. Ye, H., Liu, W., & Liu, Y. (2020). Image style transfer method based on improved style loss function. In 2020 IEEE 9th Joint International Information Technology and Artificial Intelligence Conference (ITAIC) (Vol. 9, pp. 410–413).

42. Yoo, S., Lee, S., Kim, S., Hwang, K. H., Park, J. H., & Kang, N. (2021). Integrating deep learning into CAD/CAE system: Generative design and evaluation of 3D conceptual wheel. Structural and Multidisciplinary Optimization, 64(4), 2725–2747.

43. Zhou, Z., Wu, Y., Yang, X., & Zhou, Y. (2022). Neural style transfer with adaptive auto-correlation alignment loss. IEEE Signal Processing Letters, 29, 1027–1031.

CHAPTER

7

Advanced Deep Learning Techniques

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

- Understand selective attention mechanisms in human vision and sensory processing.
- Understand the reinforcement learning for visual attention in computer vision tasks.
- Explore attention mechanisms in neural machine translation.
- Understand generative and discriminative models in machine learning.
- Learn the architecture and applications of GANs and DCGANs.
- Explore conditional GANs and their use in context-based image generation.
- Understand the concept and mechanism of competitive learning.

KEY TERMS FROM THIS CHAPTER

Attention mechanisms
Computer vision
Generative adversarial networks (GANs)
Natural language processing (NLP)
Reinforcement learning
Competitive learning
Context vector
Machine translation
Recurrent neural network (RNN)
Visual attention

UNIT INTRODUCTION

People do not consciously utilize all the available information from their surroundings at any given moment. Instead, the emphasis is placed on sections of the data that pertain to the current objective. This concept is known as attention, in the context of AI. Applications of artificial intelligence can also benefit from the integration (insertion) of new ideas. Attention-based models use various strategies, such as reinforcement learning, to concentrate on data segments that are pertinent to the assigned job. In recent decades, several methods have been applied to improve performance (Han et al., 2018).

These models have a strong connection to attention models, but they differ in that the attention is primarily directed towards specific parts of the stored data. One way to understand the process is by comparing it to the way humans access memory to complete specific tasks. Humans possess a vast reservoir of information stored within the memory cells of their brains. Nevertheless, only a fraction of the information is typically accessed at a time, focusing solely on the aspects that are important to the current task (Sadad et al., 2021).

In the same was as contemporary computers possess large memory units, computer programs are crafted to efficiently and systematically access this memory by utilizing variables as indirect addressing mechanisms. Hidden states are present in all neural networks, serving as a form of memory. Nevertheless, the close integration of data access and computations poses a challenge in terms of separation.

Through careful control of memory access and the incorporation of addressing mechanisms, the resulting neural network exhibits computational patterns that closely resemble human brain behavior. Typically, these networks exhibit superior generalization capabilities compared to conventional neural networks when making predictions on data that is not part of the training set (Zaidi & El Naqa, 2021).

One technique to pay attention to a neural network's memory internally is by selective memory access. The final design is referred to as a neural Turing machine or memory network.

Generative adversarial networks are specifically designed to generate models of data by using samples. These networks can generate highly realistic samples from data through the utilization of two adversarial networks. A network is used to create artificial samples, while another network acts as a classifier to determine if the samples are real or synthetic (Mutasa et al., 2020).

A competitive game leads to the gradual enhancement of the generator, until the discriminator becomes incapable of differentiating between genuine and fake samples. In addition, by concentrating on a certain kind of context, such an image caption, it is possible to guide the creation of particular kinds of desired samples.

Attention mechanisms frequently face the challenge of making difficult choices regarding which specific aspects of the data to focus on. This decision can be compared to the choices faced by an algorithm that uses reinforcement learning. While some attention-based model building strategies strongly rely on reinforcement learning, others do not (Jeyaraj & Nadar, 2020).

Memory networks are a class of topology closely related to neural Turing computers. Lately, there has been some potential in developing question-answering systems, although the outcomes are still in their early stages. The development of a neural Turing machine has the potential to unlock a multitude of unexplored possibilities in the field of artificial intelligence. Like in the past, the success of neural networks relies heavily on having access to large amounts of data and powerful computational resources (Abdullakutty et al., 2021).

This chapter explores advanced deep learning techniques that are essential for contemporary machine learning applications. The exploration starts with Attention Mechanisms, which have the ability to selectively enhance the performance of neural networks by focusing on inputs that are relevant. Next, Generative Adversarial Networks (GANs) will be discussed, which present an interesting framework where two networks engage in a competitive battle to produce real data, specifically in the context of image generation tasks. The unit concludes with Competitive Learning, a mechanism in which neurons engage in a competitive process to effectively represent patterns in data (Garg et al., 2021).

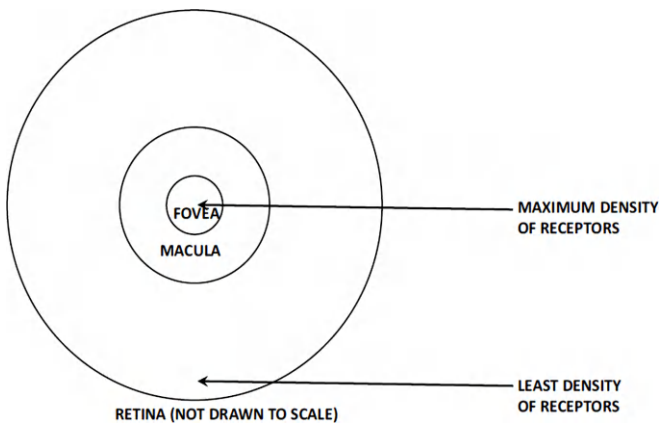


Figure 7.1. Illustration of resolutions in various ocular areas. Macula captures most of what we focus on.

Source: Charu C. Aggarwal, Creative Commons License.

7.1. ATTENTION MECHANISMS

It is uncommon for individuals to utilize all their sensory inputs to complete particular tasks. Let's explore the challenge of locating a specific address based on a house number and street name. Thus, a crucial aspect of the assignment is to determine the numerical value displayed on either the mailbox or the door of a residence. During this process, the retina frequently captures a wider view of the scene, even though our attention is rarely directed towards the entire image. The retina contains a small area called the macula, which includes a central fovea. This fovea has an exceptionally high resolution in comparison to the rest of the eye (Niu et al., 2021).

In this particular area, there is a significant number of cones (a photoreceptor cell) that are sensitive to color, while the rest of the eye, especially the outer parts, have lower resolution and are mainly composed of rods (a photoreceptor cell more sensitive to low light intensity) that are not sensitive to color. Figure 7.1 displays the various regions of the eye. The fovea concentrates on the numbers when searching a street number, and the general picture lands on a certain region of the retina that corresponds to the macula (and especially the fovea). While it is possible to recognize the presence of objects in our peripheral vision, relying on them for tasks that require attention to detail is extremely challenging. As an example, reading letters that are projected on the outer edges of the retina can be quite challenging (Guo et al., 2022).

Measuring about 1.5 mm in diameter, the foveal area is a minuscule portion of the overall retina. The eye efficiently transmits only a small fraction of the image's surface area that reaches the retina. This approach has a clear advantage from a biological standpoint. It selectively transmits a small portion of the image in high resolution, minimizing the internal processing needed for the task at hand (Kardakis et al., 2021).

While selective attention to visual stimuli is particularly easy to grasp because to the structure of the eye, selectivity is not limited to visual characteristics alone. Depending on the circumstance, the majority of the human senses such as hearing and smell are frequently extremely focused. Similarly, the concept of attention in relation to computer vision will be discussed initially, followed by an exploration of other domains such as text (Brauwers & Frasincar, 2021).

One noteworthy use of attention stems from the photos taken by Google Street View, an amazing tool created by Google to locate online street photos in many different nations. For this type of retrieval, a method is needed to establish a connection between houses and their street numbers. Even though the street number can be captured in the image, it must be extracted from the image itself.

Is it possible to methodically identify the street address numbers from a large image of the front of a house? It is crucial to adopt a systematic approach in order to

effectively concentrate on specific elements within an image and locate the desired information. One of the main challenges in this scenario is the difficulty in pinpointing the relevant section of the image based on the available information. Therefore, a methodical approach is necessary for searching specific sections of the image by leveraging knowledge acquired from previous iterations (LIU et al., 2021).

It is helpful to take inspiration from the functioning of biological organisms in this case. Biological organisms use the visual clues they are focusing on to quickly determine where to look next to obtain what they seek. For example, our eyes naturally move to the upper left or right to find the street number when our attention is directed to the doorknob. Our learnt brain circuits and our past experiences have shaped this habit.

The goal of reinforcement learning techniques, which is what this iterative process sounds like, is to continuously learn from past steps what to do to obtain rewards (i.e., complete a job like finding the street number). Several uses of attention are paired with reinforcement learning.

The idea of attention is particularly applicable to natural language processing, as important information can often be hidden within long passages of text. This problem commonly arises in various applications like question answering systems and machine translation. In these cases, the recurrent neural network needs to encode the entire sentence into a vector of fixed length (Niv et al., 2015).

Because of this, it is frequently difficult for the recurrent neural network to concentrate on the crucial elements of the source sentence while converting it into the target sentence. In situations like this, it can be helpful to translate the statement by aligning the target sentence with pertinent parts of the source sentence. When constructing a particular element of the target sentence, attention processes are helpful for identifying the relevant source phrase parts (Choi et al., 2018).

It is worth mentioning that attention mechanisms can be approached from different perspectives, not just limited to reinforcement learning. Indeed, most attention techniques used in natural language models do not depend on reinforcement learning. Instead, they use attention to allocate varied levels of significance to distinct components of the information in a versatile manner (Yan et al., 2019).

7.1.1. Recurrent Models of Visual Attention

Reinforcement learning plays a crucial role in the study of recurrent models of visual attention, enabling researchers to direct their attention towards significant features within an image. One approach is to utilize a straightforward neural network to enhance the clarity of specific areas in an image that are focused on a particular point. As one develops a deeper comprehension of the image and determines the pertinent sections to explore, the placement may change over time (Guo et al., 2023).

Identifying a particular spot at a specific moment is referred to as a glimpse. The controller, a recurrent neural network, is used to precisely calculate the location at every time-stamp. In making this selection, the feedback from the previous time-stamp's glance is taken into account. Research has demonstrated that when a convolutional neural network is used instead of a simple neural network, often known as a "glimpse network," superior classification results can

be achieved when image processing is combined with reinforcement-based training (Spratling & Johnson, 2004).

Next, a dynamic environment where the viewable portions of the image may fluctuate with time-stamp t, and the image may be partially observable is explored. Starting from a generic picture Xt fixed in time, and later on other specialized situations. By treating certain neural network components as "black boxes," the whole architecture can be explained in a modular fashion. Below is a description of these modular sections:

- Glimpse Sensor: When presented with an image, a glimpse sensor is able to generate a retina-like representation of the image, based on the given representation. The glimpse sensor is designed to work within the limitations of bandwidth constraints. It can only access a small, high-resolution portion of the image, specifically centered at . This is similar to how the eye perceives an image in the physical world. The resolution of an image specific location decreases as it moves away from the previous location . The image's reduced representation is indicated by ρ (,,.). The glimpse sensor, shown in Figure 7.2's top left corner, is an essential component of a broader glimpse network. Below is a detailed discussion of the network that follows (Osman & Samek, 2019).

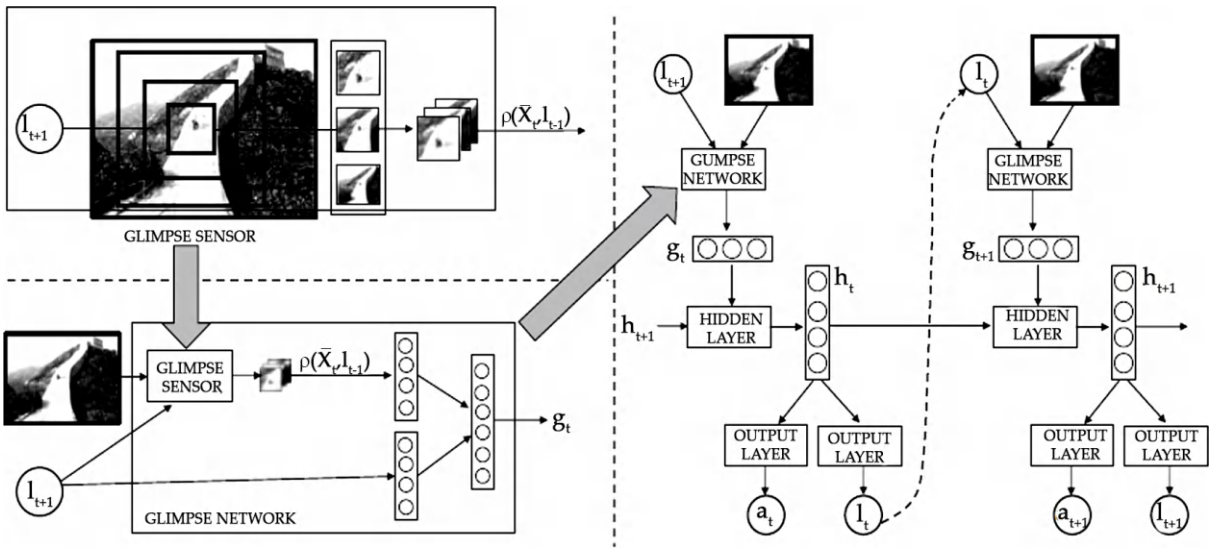


Figure 7.2. Illustration of the recurrent architecture for leveraging visual attention.

Source: D. Ackley, Creative Commons License.

- Glimpse Network: The glimpse representation ρ (,, .) and the glimpse location are encoded into hidden spaces by the glimpse network, which is made up of the glimpse sensor and linear layers. Following that, the two are merged into a single concealed representation by the utilization of an additional linear layer. The output, , serves as the input to the time-stamp of the hidden layer of the recurrent neural network. Figure 7.2 provides a partial view of the network, located in the lower-right corner (Le Meur et al., 2006).

- Recurrent Neural Network: At every time-stamp, the recurrent neural network is in charge of producing the action-driven outputs that eventually result in rewards. The glimpse network, which incorporates the glimpse sensor as a component of the glance network, is a subset of the recurrent neural network. The rewards are linked to the network's output activity at a specific time-stamp, referred to as "a_t". In a simple case, the reward could be the label of the object or a numerical value, as demonstrated in the example of Google Street view (Cox et al., 2022). In addition, it provides the location of a specific point in the image for the next time-stamp, instructing the glimpse network on where to direct its attention. The probability of acting at is computed as the output π (). Probability is often implemented using the softmax function, a widely used tool in policy networks. The recurrent network is trained using an objective function from the REINFORCE framework to maximize the expected reward over time. The reward for each action logπ() is calculated by multiplying the logarithm of the probability of taking that action with the advantage of that action (Sussner & Esmi, 2011). Thus, the general strategy involves using a reinforcement learning technique to simultaneously learn the attention locations and actionable outputs. It is interesting to observe that the recurrent network's hidden states h_t contain the encoded history of its actions. Figure 7.2 showcases the neural network's overall architecture on the right-hand side. It is important to note that the glimpse network is an integral component of the overall architecture. This is because the recurrent network relies on a glimpse of the image or current scene to carry out computations at each time-stamp (Deco & Rolls, 2004).

7.1.2. Attention Mechanisms for Machine Translation

Machine translation often relies on recurrent neural networks, specifically those that utilize long short-term memory (LSTM). In the following discussion, we utilize universal symbols that pertain to any type of recurrent neural network, although the LSTM is generally the favored method in such situations. In order to simplify the explanation, we utilize a neural network with only one layer. This choice is also reflected in all the visual representations of the neural structures (Choi et al., 2018).

In real-world applications, it is common to utilize multiple layers, making it easy to extend the simplified explanation to scenarios with multiple layers. There are multiple approaches to integrating attention into neural machine translation. In this discussion, our main focus is on a method introduced by Luong, which presents an enhancement to the initial mechanism proposed by Bahdanau (Aljohany et al., 2022).

It is important to understand that there are two recurrent neural networks involved in this process. One network takes the source sentence and converts it into a representation of a fixed length. The other network then takes this representation and converts it into the target sentence. Therefore, this serves as a simple example of sequence-to-sequence learning, a commonly used technique in neural machine translation (He et al., 2021).

The source and target networks have hidden states represented by and respectively. The hidden state corresponds to the t-th word in the source sentence, while corresponds to the t-th word in the target sentence.

Through the use of attention-based methods, the hidden states undergo a transformation to become enhanced states , with the assistance of an attention layer for additional processing. The attention layer plays a crucial role in enhancing the target hidden states by incorporating relevant information from the source hidden states. This integration process leads to a more refined and improved set of target hidden states (Zhang et al., 2018).

To effectively execute attention-based processing, the objective is to locate a source representation that closely aligns with the ongoing processing of the current target hidden state . One way to accomplish this is by creating a context vector through the similarity-weighted average of the source vectors (Li et al., 2021).

$$\overline{c}_t = \frac{\sum_{j=1}^{T_s} \exp(\overline{h}_j^{(1)} \cdot \overline{h}_t^{(2)}) \overline{h}_j^{(1)}}{\sum_{j=1}^{T_s} \exp(\overline{h}_j^{(1)} \cdot \overline{h}_t^{(2)})} \tag{7.1}$$

in this instance indicates the source sentence's length. This approach to creating the context vector is the simpler of the multiple versions that have been covered. There are, nevertheless, a number of additional options, some of which have restrictions. There is a way to understand this weighting by considering the concept of an attention variable a(t, s) (Lee et al., 2021). This variable helps determine the significance of the source word s in relation to the target word t.

$$a(t,s) = \frac{\exp(\overline{h}_s^{(1)} \cdot \overline{h}_t^{(2)})}{\sum_{j=1}^{T_s} \exp(\overline{h}_j^{(1)} \cdot \overline{h}_t^{(2)})} \tag{7.2}$$

The attention vector at is a specific vector that refers to [a (t, 1), a (t, 2),... a(t,)] It is specifically associated with the target word t. The vector can be seen as a collection of weights that represent probabilities, with a total sum of 1. Its size is determined by the length of the source sentence, Ts. It is apparent that the formula for Equation 7.1 is the sum of the source hidden vectors (Nath et al., 2024), where a(t, s) represents the target word t's attention weight toward the source word s. To put it differently, we can express Equation 7.1 in the following manner

$$\overline{c}_t = \sum_{j=1}^{T_s} a(t,j) \overline{h}_j^{(1)} \tag{7.3}$$

This approach aims to find the most relevant contextual representation of the source hidden states for the current target hidden state. The importance of relevance is determined by calculating the similarity between the hidden states of the source and target, which is then represented in the attention vector (Liu & Chen, 2022). As a result, a new target hidden state , is generated by combining the information from the context and the original target hidden state in the following manner

$$\overline{H}_t^{(2)} = \tanh\left(W_c \begin{bmatrix} \overline{c}_t \\ \overline{h}_t 2 \end{bmatrix}\right) \tag{7.4}$$

For the final prediction, this new hidden representation is used instead of the original hidden representation . Figure 7.4 shows the general design of the attention-sensitive system. Take note of the improvements from Figure 7.3 that include an attention mechanism. The global attention model is the name given to this model. Because all source words are given a probabilistic weight and no hard decisions are made regarding which word is more relevant to a target word, this model is known as a soft attention model (Mohamed et al., 2021).

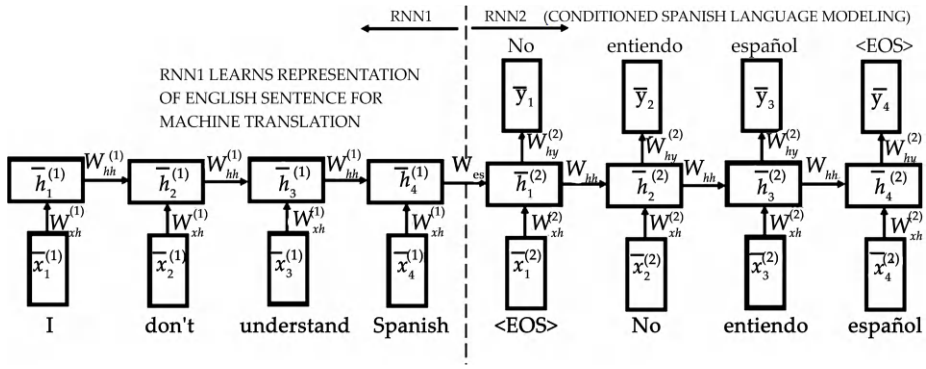


Figure 7.3. Illustration of machine translation without attention.

Source: G. Hinton, Creative Commons License.

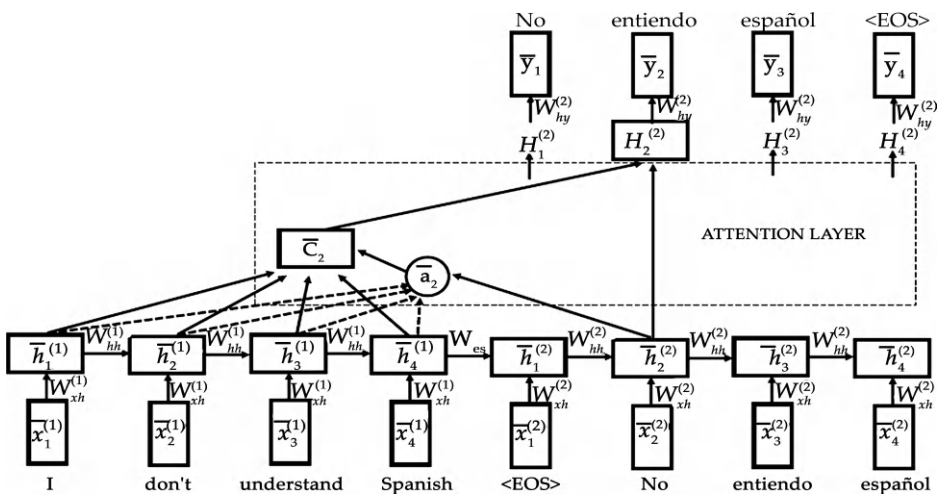


Figure 7.4. Illustration of machine translation with attention.

Source: G. Hinton, Creative Commons License.

7.2. GENERATIVE ADVERSARIAL NETWORKS (GANS)

Since both generative and discriminative models are utilized to create generative adversarial networks, we will first go over their concepts before moving on to generative adversarial networks. The following are these two categories of learning models:

- Discriminative Models: Given the feature values , discriminative models calculate the conditional probability P (of the label y. Logistic regression serves as an example of a discriminative model (Pan et al., 2019).
- Generative Models: The joint probability P (represents the generative probability of a data instance and can be evaluated using generative models. It is crucial to keep in mind that the joint probability can be used in the following manner to use the Bayes rule to estimate the conditional probability of y given X.

$$P\left(y \mid \bar{X}\right) = \frac{P\left(\bar{X}, y\right)}{P\left(\bar{X}\right)} = \frac{P\left(\bar{X}, y\right)}{\sum_z P\left(\bar{X}, z\right)} \tag{7.5}$$

The generative model exemplified by the naïve Bayes classifier.

Supervised settings exclusively utilize discriminative models, while generative models find application in both unsupervised and supervised settings. As an example, in a multiclass scenario, it is possible to construct a generative model for a specific class by establishing a suitable prior distribution for that class. Samples from the prior distribution can then be used to generate class examples (Saxena & Cao, 2021).

Similarly, using a probabilistic model with a preset prior, it is possible to generate every single data point in the dataset from a certain distribution. In the variational autoencoder, this technique is frequently employed. The procedure involves choosing data points to serve as a prior from a Gaussian distribution, which are then used as input for the decoder. The objective is to produce samples that closely mirror the given data (Creswell et al., 2018).

7.2.1. Generating Image Data by using GANs

GANs are often used to generate diverse image objects with different contextual backgrounds. Undoubtedly, the utilization of GANs for generating images is extremely prevalent. The generator used in the image setting is commonly known as a deconvolutional network. The following discusses the famous method for creating a deconvolutional network for the GAN. Consequently, the previously mentioned GAN is frequently referred to as a DCGAN. It is important to note that transposed convolution has mostly supplanted the phrase "de convolution" in recent years, as the former can be a little misleading (Wang et al., 2018).

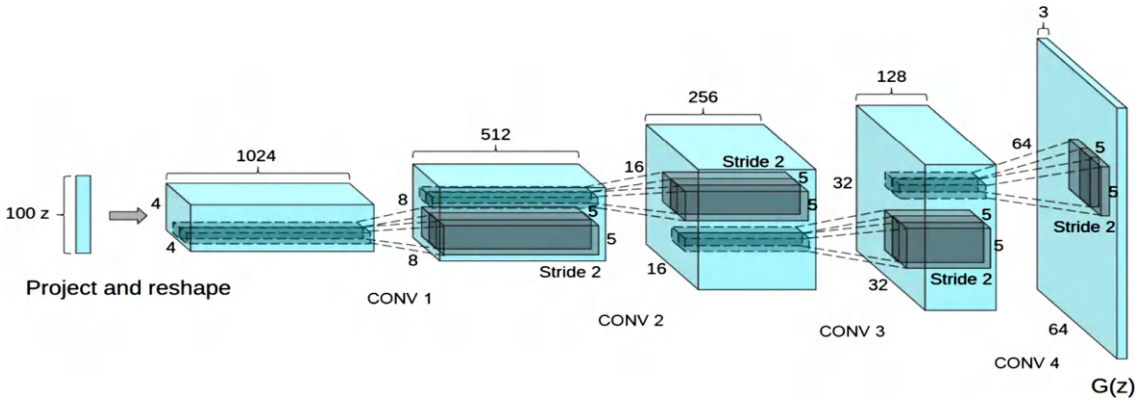


Figure 7.5. Illustration of DCGAN Convolution architecture.

Source: L. Bottou, Creative Commons License.

Figure 7.6. Illustration of the smooth transitions of the image in each row as a resulting of altering the input noise.

Source: M. Arjovsky, Creative Commons License.

The decoder's starting point is 100-dimensional Gaussian noise, which is used to initiate the process. Ten thousand four-by-four feature maps are created from the hundred-dimensional Gaussian noise. One way to accomplish this is by performing a matrix multiplication on the 100-dimensional input using a fully connected approach (Skandarani et al., 2023). The resulting output is then reshaped into a tensor. As a

result, the layers become shallower by half, but their lengths and widths double. For example, there are 512 feature maps in the second layer and 256 feature maps in the third layer (Lei et al., 2012).

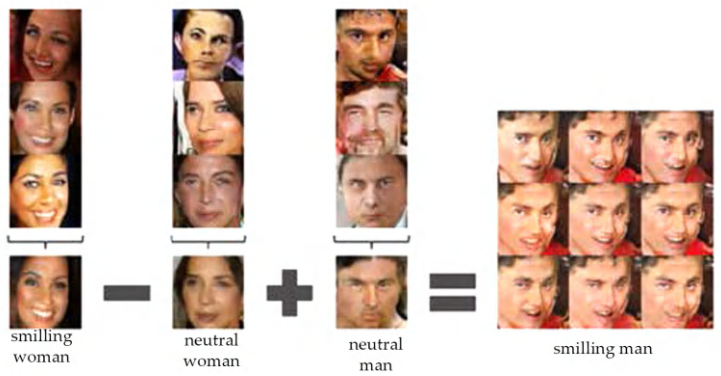


Figure 7.7. Illustration of semantic importance of arithmetic operations on input noise.

Source: M.Arjovsky, Creative Commons License.

Nevertheless, it may seem odd to increase the length and width through convolution, as convolutions typically reduce the spatial map size, unless extra zero padding is used. Using transposed convolutions with a fractional value of 0.5 or fractionally strided convolutions is one method to do this. The scenario is very similar to unit strides when it comes to fractional strides. One way to think about it is as a convolution that takes place after stretching the input volume spatially. Either interpolated values or zeros can be inserted between rows and columns to accomplish this stretching (Han et al., 2019).

Using convolution on this input with a stride of 1 is equivalent to using fractional steps on the original input because the input volume has already been increased by a certain factor. Using pooling and unpooling techniques to control the spatial footprints is an additional approach to take into consideration in place of fractionally strided convolutions. In the case of fractionally strided convolutions, pooling and unpooling procedures are not required. Figure 7.5 provides an overview of the generator architecture in DCGAN (Sorin et al., 2020).

The images produced are highly responsive to the variations in the noise samples. Examples of the images generated using multiple noise samples are shown in Figure 7.6. An interesting example can be found in the sixth row, where a room undergoes a gradual transformation from being windowless to having a spacious window. In the case of the variational autoencoder, smooth transitions are also seen. Vector arithmetic can be used to alter the noise samples, enabling meaningful interpretation (Maclin & Shavlik, 1995).

For instance, one may add the noise sample of a happy male and subtract the noise sample of a neutral lady from the smiling woman. The generator uses this noise sample to produce an image sample of a happy man. This particular situation is displayed in Figure 7.7.

A convolutional neural network design has been used in the discriminator; however, the leaky ReLU was utilized in place of the ReLU. After being flattened, the discriminator's last convolutional layer is sent through a sigmoid output. Both the discriminator and the generator did not utilize fully connected layers. Convolutional neural networks typically use the ReLU activation. To lessen any issues with the vanishing and ballooning gradient concerns, batch normalization was applied (Porkodi et al., 2023).

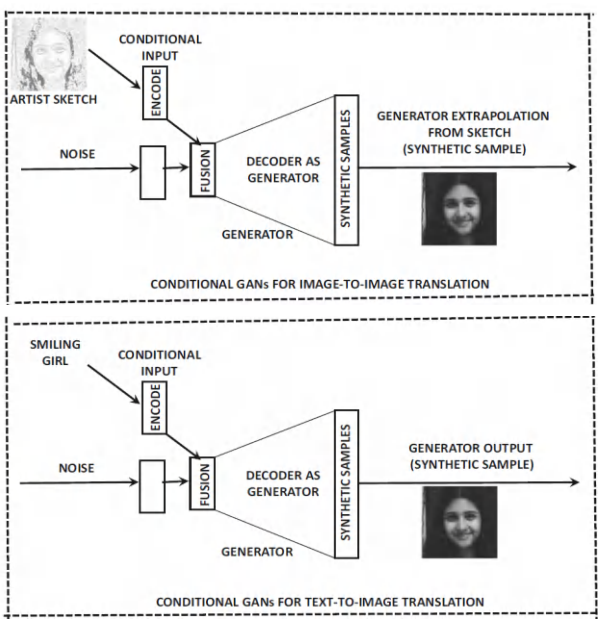
Remember

Generative Adversarial Networks (GANs) consist of two neural networks generator and discriminator that compete against each other to produce realistic synthetic data, such as images.

7.2.2. Conditional Generative Adversarial Networks

An additional input entity has an impact on both the discriminator and the generator in conditional adversarial generative networks (CGANs). This object has the ability to assume various forms, such as a caption, a label, or even another object that belongs to the same category. Usually, the input comprises of pairs of desired objects and their corresponding contexts. Usually, the contexts have some domain-specific relationship to the target objects that the model learns (Wang et al., 2018).

A setting like "smiling girl," for example, would bring up a picture of a happy girl. It is vital to remember that the CGAN can produce a wide range of images for smiling girls, with the exact choice being determined by the noise input value. Therefore, using its imagination and ingenuity, the CGAN is able to generate a universe of target things (Zhang et al., 2019).



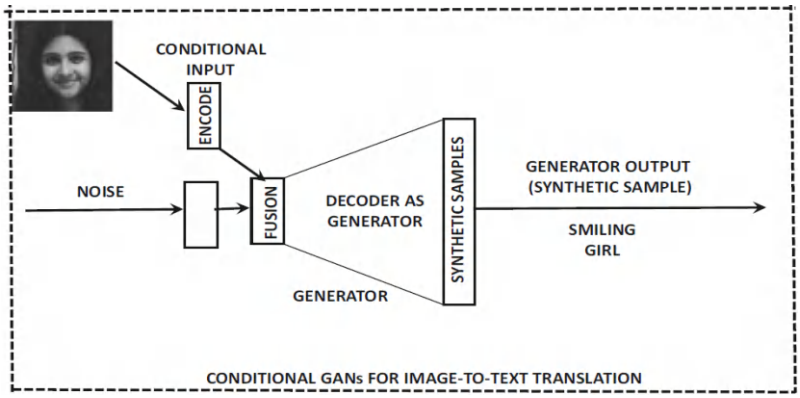


Figure 7.8. Illustration of various forms of conditional generators for adversarial networks. The examples are purely illustrative and should not be considered as an actual output from a CGAN.

Source: M. R. Caruana, Creative Commons License.

Various kinds of conditional generators for adversarial networks are shown in Figure 7.8. The samples do not represent actual CGAN output; rather, they are simply illustrative in nature. The generator has the potential to produce stable objects regardless of the noise input. The range of target objects becomes more limited, while the context becomes more complex compared to the desired output. Consequently, it often happens that the modeled objects are more complex than the surrounding inputs. For example, it is more customary for the object to be a picture and the context to be a caption, rather than the reverse. However, both scenarios are theoretically possible (Aggarwal et al., 2019).

Figure 7.8 showcases various examples of conditioning in conditional GANs. The context is crucial in providing the necessary input for the conditioning process. Typically, the context can be of any object data type, while the resulting output can be of a different data type. One of the most interesting applications of CGAN involves situations where the context is relatively simple (such as a caption) compared to the complex generated output (such as an image) (Souibgui & Kessentini, 2020).

When faced with such situations, CGANs demonstrate a notable ability to generate imaginative solutions to complete incomplete information. The specific details may vary based on the noise input provided to the generator. Here are a few examples of object-context pairs:

Labels can be assigned to each object. The label serves as the conditioning factor for generating images. For example, in the MNIST dataset, the conditioning variable can take on values between 0 and 9, representing the labels. The generator is responsible for producing an image that matches the given conditioning, specifically a digit. Similarly, when working with an image data set, you can use a label like "carrot" to represent the desired condition, and the expected result would be an image displaying a carrot. In the initial research on conditional adversarial nets, the experiments yielded a 784-dimensional representation of a digit by utilizing a label that ranged from 0 to 9 (Douzas & Bacao, 2018). The type of the target object and its context may be the same, but the context's information content may differ from the target objects. As an example,

consider a scenario where a sketch of a purse created by a human artist is compared to a photograph of the same purse, but with every complex detail added. Here's another example: Imagine a criminal suspect drawn by an artist, with the target object derived from the subject's real photo. This situation offers some background information for comprehending the generator's output (Oliveira et al., 2018).

The objective is to utilize a provided sketch to create a range of real samples with carefully filled-in details. An example is shown in the upper section of Figure 7.8. When working with complex depictions of contextual entities, like photographs or textual sentences, it may be essential to transform them into a multidimensional representation using an encoder. This allows for the fusion of multidimensional Gaussian noise. An encoder can take different forms depending on the context. For image context, it may be a convolutional network, while for text context, it could be a recurrent neural network or a word2vec model (Luo et al., 2021). Every object can be linked to a written description, such as an image paired with a caption, which gives it additional meaning. The caption serves as the foundation for the object's characteristics. The concept is that by offering a context such as "blue bird with sharp claws," the generator is expected to produce a fantasy image that accurately portrays this description (Guo et al., 2019). A visual representation of a happy young girl can be seen in Figure 7.8.

It is worth noting that an image context can be utilized to generate a caption using a GAN, as demonstrated at the bottom of the figure. On the other hand, it is more frequently observed that complicated objects, such as images, are created from simpler contexts like captions, rather than the other way around. The availability of various supervised learning methods makes it possible to generate simple objects, such as caption or labels, from complicated objects like images with greater accuracy (Ma et al., 2021).

7.3. COMPETITIVE LEARNING

The learning techniques covered in this book mostly concentrate on correcting mistakes in the neural network by varying its weights. A different paradigm is offered by competitive learning, where the goal is not only to map inputs to outputs and fix mistakes. Instead, the neurons engage in a competitive process to determine which subset of similar input data they will respond to, and then adjust their weights accordingly. Consequently, the learning process exhibits notable differences compared to the commonly used backpropagation algorithm in neural networks (Grossberg, 1987).

The following is the general concept of training. When the weight vectors of an input and output neuron are more comparable, the activation of the output neuron increases. It is expected that the input and the neuron's weight vector are of the same dimensionality. Using the weight vector's and the input's Euclidian distances to calculate the activation is a common technique. Greater activations occur at shorter distances. The output unit that activates in response to an input the highest is deemed the winner and is advanced toward the input (Ahalt et al., 1990).

The winner-take-all technique modifies only the neuron with the highest activation that emerges as the winner, leaving the other neurons unchanged. Additional variations of the competitive learning paradigm enable the involvement of neighboring neurons in the update process, as determined by predefined relationships. In addition, there are also mechanisms available that enable neurons to inhibit each other (Gutiérrez et al., 2019).

These regularization techniques are beneficial for learning representations with a certain kind of predefined structure, which is important for applications such as two-dimensional visualization. First, we go over a basic implementation of the winner-take-all strategy in the competitive learning algorithm (Revilla et al., 2008).

Consider the following: an input vector (X) in dimension d, paired with the weight vector , which corresponds to the ith neuron in the same number of dimensions. Let's look at an example where we use m neurons overall, which is typically considerably less than the amount of the data collection n. The following steps function by repeatedly taking a sample of X from the input data and performing subsequent calculations (Kaski & Kohonen, 1994).

- For every i, we calculate the Euclidean distance If a particular neuron has the lowest Euclidean distance value, it is deemed the winner. It is important to consider that the value of | is regarded as the activation value of the ith neuron.

- The following rule is applied to update the pth neuron:

$$\bar{W}_p \Leftarrow \bar{W}_p + \alpha\left(\bar{X} - \bar{W}_p\right) \quad (7.6)$$

The learning rate in this case is $\alpha > 0$. The value of α is usually substantially smaller than 1. Sometimes, as the algorithm advances, the learning rate α decreases (Yair et al., 1992).

Weight vectors are considered prototypes in competitive learning, just like centroids are in k-means clustering. The winning prototype is then adjusted slightly towards the training instance. The α parameter controls how much the point's movement, , is influenced by the distance between the weight vector and the point (Pal et al., 1996). It is worth noting that k-means clustering can also accomplish similar objectives, although it takes a different approach. Indeed, once a point is assigned to the winning centroid, it gradually shifts that centroid closer to the training instance by a small distance during each iteration. Competitive learning offers a range of possibilities within this framework, making it suitable for unsupervised tasks such as clustering and dimensionality reduction (Boisot, 1995).

This is a very ambiguous activity we propose to run following the exercise: https://keras.io/examples/generative/conditional_gan/

Did you know?

Competitive Learning in neural networks is inspired by biological neurons competing to activate based on input stimuli, contributing to efficient pattern recognition and learning in artificial intelligence systems.

ACTIVITY 7.1.

Objective: Explore and understand advanced deep learning techniques including Attention Mechanisms, Generative Adversarial Networks (GANs), and Competitive Learning.

Implement a code to generate an handwritten image of a number between 0 and 9 from the MINST Dataset, see the following example as a guide https://keras.io/examples/generative/conditional_gan/ from Keras.io.

Materials Needed:

- Access to a computer with deep learning libraries (e.g., TensorFlow, PyTorch);
- Internet access for research and tutorials.

Activity Steps:

- Introduction and Hands-on Implementation
- Begin with a brief introduction to Attention Mechanisms, GANs, and Competitive Learning, emphasizing their applications and benefits.
- Implement and run the GAN network code example from Keras.io and explain each of the steps.
- Allow participants to work through the code, experimenting with parameters and datasets.
- Encourage discussion among participants to share insights, challenges, and solutions.
- Conclude with a group discussion to reflect on the practical implications and potential future applications of these techniques.

SUMMARY

- The chapter "Advanced Deep Learning Techniques" explores various important subjects in modern machine learning. The article explores Attention Mechanisms, which improve the performance of neural networks by directing their attention to important inputs. Applying the concept of Recurrent Models of Visual Attention to visual data has shown significant improvements in tasks such as object recognition.
- Attention mechanisms in machine translation enhance translations by selectively focusing on specific parts of sentences. Generative Adversarial Networks (GANs) present a framework in which two neural networks engage in a competitive process to produce lifelike data, particularly images. Exploring the application of GANs for Image Generation provides a comprehensive analysis of this topic.
- Conditional Generative Adversarial Networks enhance GANs by incorporating additional information to influence the generated outputs. Competitive Learning delves into models where neurons engage in a fierce competition to efficiently represent data patterns.

REVIEW QUESTIONS

1. How do Attention Mechanisms improve deep learning models?
2. What are Recurrent Models of Visual Attention, and how are they applied in computer vision?
3. Explain the concept and structure of Generative Adversarial Networks (GANs).
4. Provide a specific application where GANs are used and explain their advantages.
5. What distinguishes Conditional Generative Adversarial Networks (cGANs) from traditional GANs?
6. What is Competitive Learning in neural networks, and how does it contribute to pattern recognition?

MULTIPLE CHOICE QUESTIONS

1. What do Attention Mechanisms in deep learning primarily enhance?

a. Data preprocessing
b. Model regularization
c. Memory utilization
d. Input relevance selection

2. Recurrent Models of Visual Attention are most beneficial for improving:

a. Image classification
b. Speech recognition

c. Text summarization

d. Object detection

3. Generative Adversarial Networks (GANs) consist of:

a. One network for generation and one for reinforcement

b. Two networks that compete against each other

c. Three networks that collaborate on image generation

d. A hierarchical structure of multiple networks

4. `What is a primary application of GANs?

a. Document classification

b. Sentiment analysis

c. Image generation

d. Regression analysis

5. Conditional Generative Adversarial Networks (cGANs) improve upon traditional GANs by:

a. Introducing attention mechanisms

b. Using reinforcement learning

c. Conditioning generated outputs on additional information

d. Applying unsupervised learning techniques

6. Competitive Learning in neural networks involves:

a. Collaboration between neurons to enhance learning speed

b. Neurons working independently to improve robustness

c. Reinforcement of synaptic connections between neurons

d. Neurons competing to determine which best matches the input pattern

Answers to Multiple Questions

1. (b); 2. (c); 3. (c); 4. (c); 5. (c); 6. (d).

REFERENCES

1. Abdullakutty, F., Elyan, E., & Johnston, P. (2021). A review of state-of-the-art in face presentation attack detection: From early development to advanced deep learning and multi-modal fusion methods. Information Fusion, 75, 55–69.
2. Aggarwal, K., Kirchmeyer, M., Yadav, P., Keerthi, S. S., & Gallinari, P. (2019). Conditional generative adversarial networks for regression. arXiv Preprint. https://arxiv.org/abs/1905.12868 (accessed on 5 September 2024).
3. Ahalt, S. C., Krishnamurthy, A. K., Chen, P., & Melton, D. E. (1990). Competitive learning algorithms for vector quantization. Neural Networks, 3(3), 277–290.

4. Aljohany, D. A., Al-Barhamtoshy, H. M., & Abukhodair, F. A. (2022). Arabic machine translation (ARMt) based on LSTM with attention mechanism architecture. In 2022 20th International Conference on Language Engineering (ESOLEC) (pp. 78–83).
5. Boisot, M. H. (1995). Is your firm a creative destroyer? Competitive learning and knowledge flows in the technological strategies of firms. Research Policy, 24(4), 489–506.
6. Brauwers, G., & Frasincar, F. (2021). A general survey on attention mechanisms in deep learning. IEEE Transactions on Knowledge and Data Engineering, 35(4), 3279–3298.
7. Choi, H., Cho, K., & Bengio, Y. (2018). Fine-grained attention mechanism for neural machine translation. Neurocomputing, 284, 171–176.
8. Cox, G. E., Palmeri, T. J., Logan, G. D., Smith, P. L., & Schall, J. D. (2022). Salience by competitive and recurrent interactions: Bridging neural spiking and computation in visual attention. Psychological Review, 129(5), 1144.
9. Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., & Bharath, A. A. (2018). Generative adversarial networks: An overview. IEEE Signal Processing Magazine, 35(1), 53–65.
10. Deco, G., & Rolls, E. T. (2004). A neurodynamical cortical model of visual attention and invariant object recognition. Vision Research, 44(6), 621–642.
11. Douzas, G., & Bacao, F. (2018). Effective data generation for imbalanced learning using conditional generative adversarial networks. Expert Systems with Applications, 91, 464–471.
12. Garg, R., Kumar, A., Bansal, N., Prateek, M., & Kumar, S. (2021). Semantic segmentation of PolSAR image data using advanced deep learning model. Scientific Reports, 11(1), 15365.
13. Grossberg, S. (1987). Competitive learning: From interactive activation to adaptive resonance. Cognitive Science, 11(1), 23–63.
14. Guo, M. H., Lu, C. Z., Liu, Z. N., Cheng, M. M., & Hu, S. M. (2023). Visual attention network. Computational Visual Media, 9(4), 733–752.
15. Guo, M. H., Xu, T. X., Liu, J. J., Liu, Z. N., Jiang, P. T., Mu, T. J., & Hu, S. M. (2022). Attention mechanisms in computer vision: A survey. Computational Visual Media, 8(3), 331–368.
16. Guo, X., Nie, R., Cao, J., Zhou, D., Mei, L., & He, K. (2019). FuseGAN: Learning to fuse multi-focus image via conditional generative adversarial network. IEEE Transactions on Multimedia, 21(8), 1982–1996.
17. Gutiérrez-Braojos, C., Montejo-Gamez, J., Marin-Jimenez, A., & Campaña, J. (2019). Hybrid learning environment: Collaborative or competitive learning? Virtual Reality, 23(4), 411–423.
18. Han, C., Rundo, L., Araki, R., Nagano, Y., Furukawa, Y., Mauri, G., & Hayashi, H. (2019). Combining noise-to-image and image-to-image GANs: Brain MR image augmentation for tumor detection. IEEE Access, 7, 156966–156977.

19. Han, J., Zhang, D., Cheng, G., Liu, N., & Xu, D. (2018). Advanced deep-learning techniques for salient and category-specific object detection: A survey. IEEE Signal Processing Magazine, 35(1), 84–100.

20. He, W., Wu, Y., & Li, X. (2021). Attention mechanism for neural machine translation: A survey. In 2021 IEEE 5th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC) (pp. 1485–1489).

21. Jeyaraj, P. R., & Nadar, E. R. S. (2020). Effective textile quality processing and an accurate inspection system using the advanced deep learning technique. Textile Research Journal, 90(9–10), 971–980.

22. Kardakis, S., Perikos, I., Grivokostopoulou, F., & Hatzilygeroudis, I. (2021). Examining attention mechanisms in deep learning models for sentiment analysis. Applied Sciences, 11(9), 3883.

23. Kaski, S., & Kohonen, T. (1994). Winner-take-all networks for physiological models of competitive learning. Neural Networks, 7(6–7), 973–984.

24. Le Meur, O., Le Callet, P., Barba, D., & Thoreau, D. (2006). A coherent computational approach to model bottom-up visual attention. IEEE Transactions on Pattern Analysis and Machine Intelligence, 28(5), 802–817.

25. Lee, Y., Shin, J., & Kim, Y. (2021). Simultaneous neural machine translation with a reinforced attention mechanism. ETRI Journal, 43(5), 775–786.

26. Lei, J. Z., & Ghorbani, A. A. (2012). Improved competitive learning neural networks for network intrusion and fraud detection. Neurocomputing, 75(1), 135–145.

27. Li, X., Liu, L., Tu, Z., Li, G., Shi, S., & Meng, M. Q. H. (2021). Attending from foresight: A novel attention mechanism for neural machine translation. IEEE/ACM Transactions on Audio, Speech, and Language Processing, 29, 2606–2616.

28. Liu, H. I., & Chen, W. L. (2022). X-transformer: A machine translation model enhanced by the self-attention mechanism. Applied Sciences, 12(9), 4502.

29. Liu, J. W., Liu, J. W., & Luo, X. L. (2021). Research progress in attention mechanism in deep learning. Chinese Journal of Engineering, 43(11), 1499–1511.

30. Luo, J., Huang, J., & Li, H. (2021). A case study of conditional deep convolutional generative adversarial networks in machine fault diagnosis. Journal of Intelligent Manufacturing, 32(2), 407–425.

31. Ma, N., Wang, J., Liu, J., & Meng, M. Q. H. (2021). Conditional generative adversarial networks for optimal path planning. IEEE Transactions on Cognitive and Developmental Systems, 14(2), 662–671.

32. Maclin, R., & Shavlik, J. W. (1995). Combining the predictions of multiple classifiers: Using competitive learning to initialize neural networks. In Proceedings of the IJCAI (pp. 524–531).

33. Mohamed, S. A., Elsayed, A. A., Hassan, Y. F., & Abdou, M. A. (2021). Neural machine translation: Past, present, and future. Neural Computing and Applications, 33, 15919–15931.

34. Mutasa, S., Varada, S., Goel, A., Wong, T. T., & Rasiej, M. J. (2020). Advanced deep learning techniques applied to automated femoral neck fracture detection and classification. Journal of Digital Imaging, 33(5), 1209–1217.

35. Nath, B., Sarkar, S., Das, S., & Mukhopadhyay, S. (2024). Neural machine translation for Indian language pair using hybrid attention mechanism. Innovations in Systems and Software Engineering, 20(2), 175–183.

36. Niu, Z., Zhong, G., & Yu, H. (2021). A review on the attention mechanism of deep learning. Neurocomputing, 452, 48–62.

37. Niv, Y., Daniel, R., Geana, A., Gershman, S. J., Leong, Y. C., Radulescu, A., & Wilson, R. C. (2015). Reinforcement learning in multidimensional environments relies on attention mechanisms. Journal of Neuroscience, 35(21), 8145–8157.

38. Oliveira, D. A., Ferreira, R. S., Silva, R., & Brazil, E. V. (2018). Interpolating seismic data with conditional generative adversarial networks. IEEE Geoscience and Remote Sensing Letters, 15(12), 1952–1956.

39. Osman, A., & Samek, W. (2019). DRAU: Dual recurrent attention units for visual question answering. Computer Vision and Image Understanding, 185, 24–30.

40. Pal, N. R., Bezdek, J. C., & Hathaway, R. J. (1996). Sequential competitive learning and the fuzzy c-means clustering algorithms. Neural Networks, 9(5), 787–796.

41. Pan, Z., Yu, W., Yi, X., Khan, A., Yuan, F., & Zheng, Y. (2019). Recent progress on generative adversarial networks (GANs): A survey. IEEE Access, 7, 36322–36333.

42. Porkodi, S. P., Sarada, V., Maik, V., & Gurushankar, K. (2023). Generic image application using GANs (generative adversarial networks): A review. Evolving Systems, 14(5), 903–917.

43. Revilla, M. A., Manzoor, S., & Liu, R. (2008). Competitive learning in informatics: The UVa online judge experience. Olympiads in Informatics, 2(10), 131–148.

44. Sadad, T., Rehman, A., Munir, A., Saba, T., Tariq, U., Ayesha, N., & Abbasi, R. (2021). Brain tumor detection and multi-classification using advanced deep learning techniques. Microscopy Research and Technique, 84(6), 1296–1308.

45. Saxena, D., & Cao, J. (2021). Generative adversarial networks (GANs) challenges, solutions, and future directions. ACM Computing Surveys (CSUR), 54(3), 1–42.

46. Skandarani, Y., Jodoin, P. M., & Lalande, A. (2023). GANs for medical image synthesis: An empirical study. Journal of Imaging, 9(3), 69.

47. Sorin, V., Barash, Y., Konen, E., & Klang, E. (2020). Creating artificial images for radiology applications using generative adversarial networks (GANs)–A systematic review. Academic Radiology, 27(8), 1175–1185.

48. Souibgui, M. A., & Kessentini, Y. (2020). De-GAN: A conditional generative adversarial network for document enhancement. IEEE Transactions on Pattern Analysis and Machine Intelligence, 44(3), 1180–1191.

49. Spratling, M. W., & Johnson, M. H. (2004). A feedback model of visual attention. Journal of Cognitive Neuroscience, 16(2), 219–237.

50. Sussner, P., & Esmi, E. L. (2011). Morphological perceptrons with competitive learning: Lattice-theoretical framework and constructive learning algorithm. Information Sciences, 181(10), 1929–1950.

51. Wang, Y., Wu, C., Herranz, L., Van de Weijer, J., Gonzalez-Garcia, A., & Raducanu, B. (2018). Transferring GANs: Generating images from limited data. In Proceedings of the European Conference on Computer Vision (ECCV) (pp. 218–234).

52. Wang, Y., Yu, B., Wang, L., Zu, C., Lalush, D. S., Lin, W., & Zhou, L. (2018). 3D conditional generative adversarial networks for high-quality PET image estimation at low dose. NeuroImage, 174, 550–562.

53. Yair, E., Zeger, K., & Gersho, A. (1992). Competitive learning and soft competition for vector quantizer design. IEEE Transactions on Signal Processing, 40(2), 294–309.

54. Yan, C., Tu, Y., Wang, X., Zhang, Y., Hao, X., Zhang, Y., & Dai, Q. (2019). STAT: Spatial-temporal attention mechanism for video captioning. IEEE Transactions on Multimedia, 22(1), 229–241.

55. Zaidi, H., & El Naqa, I. (2021). Quantitative molecular positron emission tomography imaging using advanced deep learning techniques. Annual Review of Biomedical Engineering, 23(1), 249–276.

56. Zhang, B., Xiong, D., & Su, J. (2018). Neural machine translation with deep attention. IEEE Transactions on Pattern Analysis and Machine Intelligence, 42(1), 154–163.

57. Zhang, H., Sindagi, V., & Patel, V. M. (2019). Image de-raining using a conditional generative adversarial network. IEEE Transactions on Circuits and Systems for Video Technology, 30(11), 3943–3956.

CHAPTER

8 Applications of Deep Learning

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

- Comprehend the transformative impact of deep learning on natural language processing.
- Learn about syntactic parsing and Google's SyntaxNet for NLP tasks.
- Understand the concept and application of distributed word representations in NLP
- Understand knowledge graph representation, completion, and embedding techniques NLP.
- Understand multimodal learning and neural models for image caption generation.
- Know the advanced techniques in automatic speech recognition using deep learning.

KEY TERMS FROM THIS CHAPTER

Deep learning (DL)
Long short-term memory (LSTM)
Natural language processing (NLP)
Recurrent neural networks (RNN)
SyntaxNet
Knowledge graphs (KG)
Machine translation
Parsing
Sequence-to-sequence models
Word embedding's

UNIT INTRODUCTION

DL has changed the field of Natural Language Processing (NLP), making significant advancements in understanding and processing human language. DL has had a significant impact in this particular field, following its influence in image and audio processing. As an example, the majority of NLP projects conducted at Stanford University, a highly esteemed institution in this field, revolve around research on deep learning (Chadha et al., 2015).

Language understanding is a challenging problem in AI due to several factors. First of all, given that every language can include hundreds of thousands of words, it is a high-dimensional undertaking. Secondly, the data is skewed, meaning that certain words or phrases occur more frequently than others due to the zip law distribution. Thirdly, language data follows grammar rules with intricate structures, where even a single word, negation, or punctuation mark can significantly alter the meaning. Additionally, the meaning of words is deeply intertwined with implicit cultural assumptions. Unlike images, text does not possess a distinct spatial-temporal organization. Consequently, words that are adjacent to each other may not necessarily pertain to the same concept, in contrast to the interconnectedness of pixels in images (Sarikaya et al., 2014). With the increasing availability of a vast amount of data on the Internet, Deep Learning (DL) is a logical choice for addressing the various challenges associated with comprehending human language. Below is a list of significant issues linked to NLP:

- Parsing;
- Part of speech tagging;
- Translation;
- Text summarization;
- Name entity recognition (NER);
- Sentiment analysis;
- Question and answer (conversational);
- Topic modeling;
- Disambiguation.

DL enhances the precision of various challenging NLP tasks, particularly parsing, which involves part-of-speech tagging and translation (Li, 2018). Nevertheless, despite the enhanced precision, many aspects of these challenges persist, and the technology is not yet prepared for complete commercialization, particularly in the context of unconstrained conversations (Lauriola et al., 2022). Deep learning models for language compactly extract information encoded in training data after extensive training on a big amount of data. Language models, trained on movie subtitles, can produce simple responses to questions about the colors or facts of objects. Complex tasks like machine translation can be resolved by using conditional language models in conjunction with recent sequence-to-sequence models. Although simpler models like n-grams rely on a limited amount of prior words to predict the next word, they remain an essential element in language modeling. It has been shown by recent studies on large-scale language models that combining RNNs with n-grams can be quite successful because of their complementing features (Izbassarova et al., 2020).

8.1. PARSING

Parsing is the process of dissecting a sentence into its individual components, such as nouns, verbs, adverbs, etc., and establishing the syntactic connections between them. This relationship is known as the parsing tree. The challenge is complex due to the uncertainty in potential decompositions (see Figure 8.1) that show two alternative methods of analyzing a sentence (Berridge & Robinson, 2003).

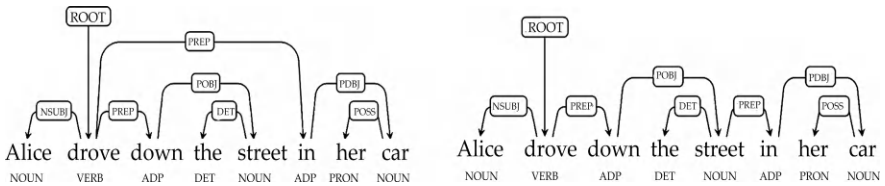


Figure 8.1. Illustration of two possible parsings of a sentence.

Source: Armando Vieira, Creative Commons License.

For example, there are at least two alternative dependent parses for the sentence "Alice drove down the street in her car." The first fits the (true) interpretation in which Alice is operating her vehicle, whereas the second fits the (ludicrous but plausible) interpretation in which Alice is inside her vehicle. Because the preposition "in" might modify either drove or street, there is uncertainty. Humans use common sense to distinguish between these possibilities as we are aware that automobiles are not capable of locating streets. Information is highly tough for machines that are integrating this environment (Kennedy, 2016). Google has recently unveiled SyntaxNet, a cutting-edge solution designed to address the complex parsing problem. (The code is developed using TensorFlow and is available on GitHub, A sentence containing 20 to 30 words can exhibit a wide range of syntactic structures. Google used a cutting-edge globally normalized transition-based neural network model to achieve outstanding outcomes in part-of-speech tagging, dependency parsing, and sentence compression. The model is a simple feed-forward neural network that operates on a task-specific transition scheme. However, it is possible to attain comparable or even better levels of accuracy when compared to recurrent models (Fodor, 1978). A feed-forward neural network using SyntaxNet analyzes a sentence and produces a distribution of potential syntactical dependencies, or hypotheses. SyntaxNet applies an independent analysis to each word, evaluating many hypotheses using a heuristic search approach known as beam search. When new, more highly scored hypotheses arise, it discards only the most implausible ones. The main idea is based on a fresh demonstration of the label bias issue. Parsey McParseface, a SyntaxNet English language parser, is regarded as the best parser and has occasionally outperformed human accuracy. The service was recently extended to around 40 languages (Momma & Phillips, 2018).

8.2. DISTRIBUTED REPRESENTATIONS

A fundamental issue in natural language processing (NLP) is the high dimensionality of data, which creates a vast search space and makes grammatical rule inference difficult. Hinton was one of the first to suggest that thick, scattered representations could be used to represent words. Bengio originally created this concept in the area of statistical language modeling. Semantics may be readily accessed and knowledge from other fields and even languages can be transmitted due to distributed representations (Zheng & Callan, 2015).

Word embedding refers to the process of learning a distributed representation for each word, typically in a vectorized form. Word2vec has gained significant popularity as a widely-used method for generating distributed word representations. This library offers a publicly accessible implementation of skip-gram vector representations for words, which is known for its efficiency.

Mikolov's work served as the foundation for both the model and its execution. Every word in a sizable corpus is used as input in Word2vec, and the words that surround it within a specified window are used as outputs. The neural network that has been trained as a classifier is then fed (see Figure 8.2). Following training, it will project the probability that each word will truly show up in the window surrounding the focal word (Rissman & Wagner, 2012).

Furthermore, the authors go beyond just implementing the model by offering vector representations of words and phrases. These representations are obtained through training the model on a massive data set consisting of approximately 100 billion words from Google News. Vectors have the potential to be incredibly high-dimensional, encompassing an extensive range of words and phrases. It is worth noting that these vector representations have the ability to capture linear regularities in the language, which is very interesting (Gelder, 1992). For example, when we apply a vectorized word equation by subtracting "Spain" from "Madrid" and adding "France," the result is "Paris."

Word2vec is a highly popular method for solving NLP problems, often used after applying the bag of words (BOW) technique with the Term Frequency-Inverse Document Frequency (TF-IDF) trick. It is relatively simple to apply and evaluate for

comprehending concealed connections between words. Gensim is a Python implementation of Word2vec that is highly regarded and extensively documented. Word2vec can be utilized either with pre-trained vectors or taught to acquire the embeddings from the beginning, using a substantial training corpus, often consisting of millions of documents (Howard & Kahana, 2002).

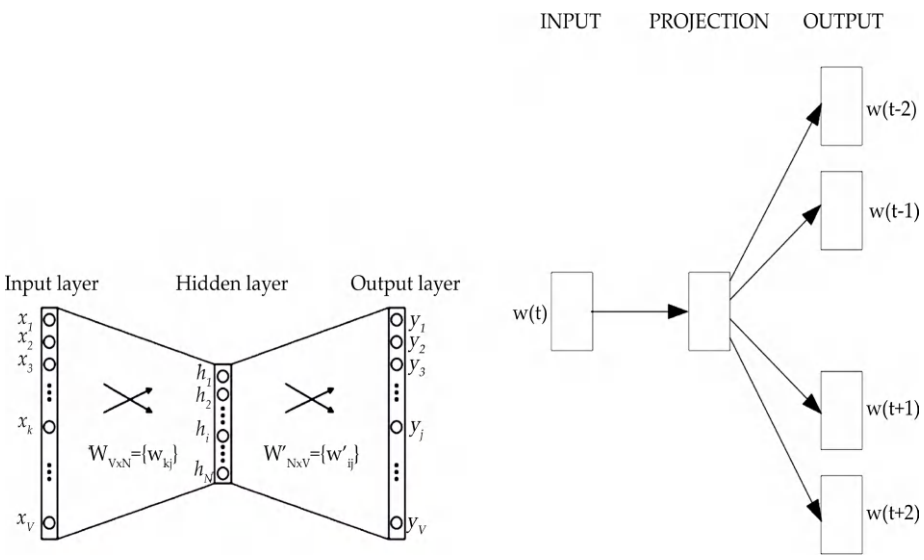


Figure 8.2.. Left: The word2vec model representation of a siamese network. The vectorized version of the word is contained in the hidden nodes h1,..., hN. Right: The word2vec schematic represented by skip-gram.

Source: Bernardete Ribeiro, Creative Commons License.

Quoc Le introduced an original strategy for encoding complete paragraphs, using a technique similar to Word2vec. This method, known as Paragraph Vector, has proven to be highly effective. Every paragraph is associated with a vector, while each individual word is also associated with a vector. Next, the paragraph vector and word vectors are combined by either averaging or concatenating them. This combination is used to make predictions about the next word based on a given context. This functions as a memory unit that locates the absent part in the given context, or, to put it more simply, the paragraph's subject (Hummel & Holyoak, 1997).

The context vectors are sampled from a sliding window that runs the length of the text paragraph and have a fixed length. Although it is separated from other paragraphs, the paragraph vector is shared by all contexts that are created from the same paragraph.

Kiros presented an innovative approach to encoding sentences by utilizing unsupervised learning and skip-through vectors. Within

Remember

Distributed representations, such as word embeddings generated by models like Word2Vec or GloVe, capture semantic links between words by considering their contextual usage in extensive text corpora. These representations facilitate the comprehension and manipulation of language by deep learning models more efficiently compared to conventional approaches.

a given section, the model used a recurrent network (RNN) to rebuild neighboring sentences. Related vector representations are created for sentences that have similar semantic and syntactic properties (Chrisman, 1991).

The model underwent evaluation on a range of tasks, encompassing image-sentence ranking, benchmark sentiment, subjectivity data sets, semantic similarity, question-type classification, paraphrase detection. The result was an encoder capable of generating robust and flexible sentence representations (Bowers, 2002).

8.3. KNOWLEDGE GRAPHS AND REPRESENTATION

In artificial intelligence, reasoning about entities and their relationships is a major challenge. Such questions are often formulated as reasoning over knowledge representations that are graph-structured. The majority of earlier research on knowledge reasoning and representation relies on a standard pipeline that includes knowledge graph inference, relationship extraction, entity resolution and co-reference, and named entity recognition (NER). Although this procedure has the potential to be successful, it may also cause the errors from each component subsystem to compound (Ji et al., 2021).

Entities in a graph are linked by relations, forming a network of connections. These entities can also be categorized based on their relations, such as when Socrates is identified as a philosopher. Linked data has revolutionized the way we interlink various data sets in the Semantic Web. In 2012, Google introduced the concept of knowledge graph to explain how it uses semantic knowledge in web search. When referring to other online knowledge bases, such as DBpedia, this word has become increasingly common (Alshahrani et al., 2017).

The knowledge graph (KG), which consists of things (nodes) and their relationships (edges), is a complex and potent visual representation of structured data. A direct bipartite graph, with people in one set of nodes and movies in the other, can be used to illustrate a recommendation system. Similar to a weighted graph, rankings might be seen as advantageous. Moreover, other forms of benefits can be integrated, like the depiction of the text the user employed in the movie review or the tags the user assigned to the movie (Sowa, 1992).

Figure 8.3 shows how a typical knowledge graph (KG) is often sparse and incomplete, even if it may include billions of relational facts (edges) and millions of entities. The goal of the task known as knowledge graph completion is to populate a graph by inferring links between nodes based on previously established connections. The objective is to discover new relational facts, often known as triples (Peng et al., 2023).

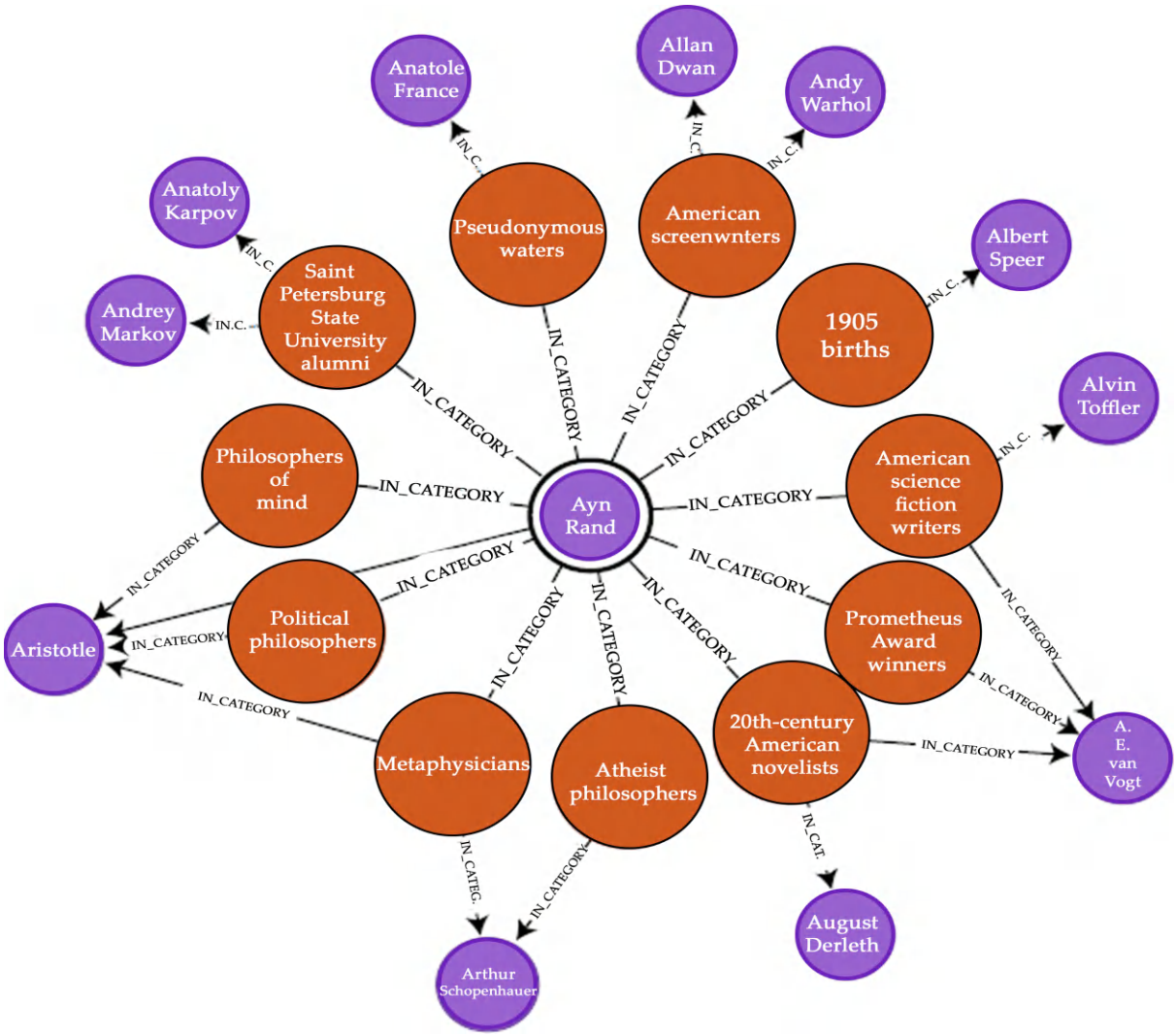


Figure 8.3. Illustration of an example of a knowledge graph.

Source: Varun Chandola, Creative Commons License.

This work could be viewed as an extension to plain text relation extraction. For the following reasons, knowledge graph completion is more challenging, much like link prediction in social network research: Knowledge graphs comprise nodes, which are entities with varying types and properties, and edges, which are relations of various kinds (not only on-off connections) in KG. Measuring the existence and nature of the relationship between two nodes is how the KG algorithm's quality is assessed (Choudhary et al., 2021).

Two large and well-known KG databases are DBpedia and Freebase. About 50 million nodes (entities) are connected by roughly 3 billion facts (edges) in Freebase. The majority of Web crawling and classification companies, including as Wolfram Alpha, Google, and Baidu, have solutions based on KGs (Nickel et al., 2015).

Embedding knowledge graphs into a continuous vector space has proven to be an incredibly valuable technique, drawing inspiration from the power of neural networks. There are various methods available, including TransE and TransH, which are known for their simplicity and effectiveness. TransE is a model that creates vector representations for entities and relationships, based on Mikolov's research (Nicholson & Greene, 2020).

The core principle underlying TransE is that the relationship between two items can be shown as a displacement between their embeddings. This means that when (h,r,t) is true, there is a relationship between the entities (see Figure 8.4). Due to certain limitations in TransE's ability to model complex relationships, a new approach called TransH was introduced. This method allows entities to have different representations depending on the type of relationship they are involved in (Tiwari et al., 2021) ransH and TransE both assume that relations and entities are embedded in the same space.

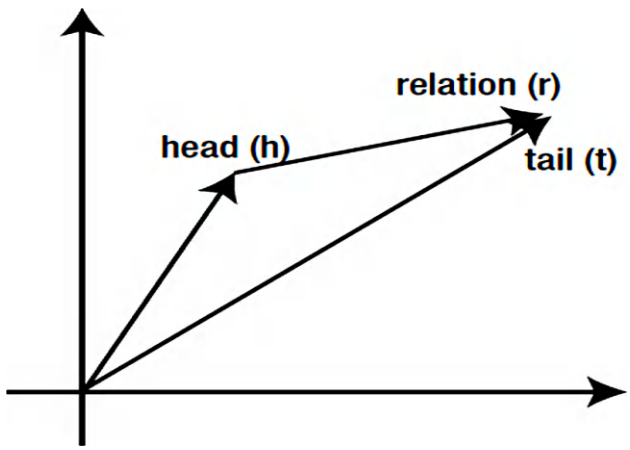


Figure 8.4. Idea behind TransE model (head, relation, tail).

Source: Arindam Banerjee, Creative Commons License.

Since they represent relations and entities as tensors, neural tensor networks (NTNs), which were first proposed by R. Socher, are renowned for their increased expressiveness. However, they require more computational resources and do not demonstrate significant performance improvements compared to simpler methods (Niu et al., 2020).

Typically, there are three methods for comparing different approaches: relation type prediction, entity prediction, and triple prediction. Both of these aspects are assessed using a ranking scale, with the highest N performance being the focus (typically N=1 and N=10). The final task involves classifying the model's performance in distinguishing genuine relationships from random ones (Chen et al., 2020).

KG completion, which is the process of predicting and filling in missing information, has various applications, particularly in personal assistants like Google Now and Cortana. These techniques can assist in addressing questions related to authors and their books, specifically focusing on the book A from author X. Google has released an API that allows anyone to obtain data from its Knowledge Graph (KG). Since the shutdown

of Freebase in December 2014, users have the ability to find entities in the Google Knowledge Graph by using standard schema types thanks to the Knowledge Graph API. The output is provided in JSON format (Sikos & Philp, 2020).

In a recent study, H. Wuang's RCNET technique outperformed humans in comprehending complicated material linked to IQ test questions. They conducted tests on many kinds of issues.

1. Analogy: Similar to how isobar is to temperature, isotherm?
   - Atmosphere;
   - Pressure;
   - Wind;
   - Current;
   - Latitude.
2. Analogy II: Choose two words (one from each set of brackets) that, when combined with the capital words, make a sense.
   - ACT (stage, audience, play);
   - CHAPTER (book, verse, read).
3. Classification: Which one stands out as unusual?
   - Quite;
   - Calm;
   - Serene;
   - Relaxed;
   - Unruffled.
4. Synonym: What word most closely resembles irrationality?
   - Irredeemable;
   - Intransigent;
   - Unsafe;
   - Nonsensical;
   - Lost.
5. Antonym: What term best contrasts with "musical"?
   - Loud;
   - Discordant;
   - Verbal;
   - Lyrical;
   - Euphonious.

These tasks can be quite challenging due to the various interpretations of words and the complex relationships between them. In order to address these challenges, the authors used a framework that aimed to enhance word embedding by taking into

account both the multiple meanings of words and the relationships between them (Fan et al., 2017) (see Figure 8.5).

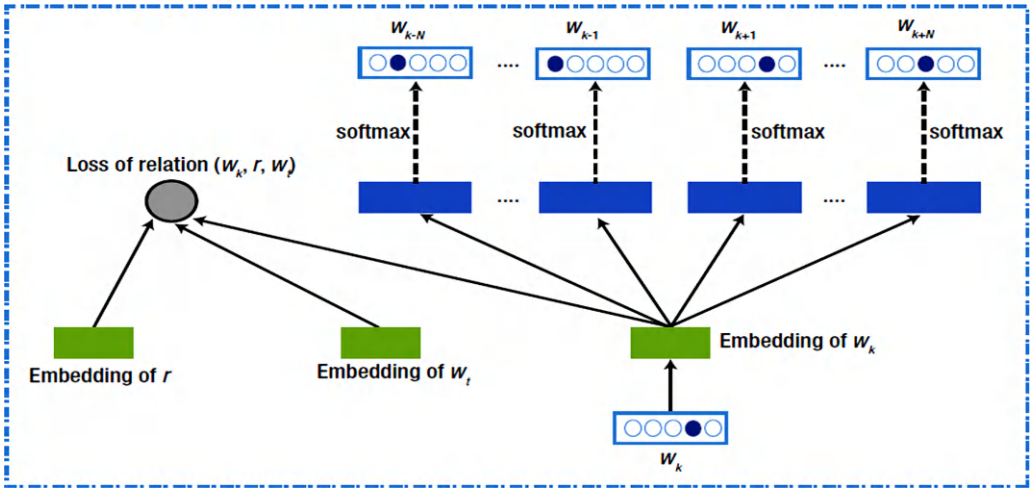


Figure 8.5. Illustration of RCNET for IQ test.

Source: Antoine Bordes, Creative Commons License.

The variational graph auto-encoder (VGAE) is a framework that utilizes the variational auto-encoder (VAE) to perform unsupervised learning and link prediction on knowledge graphs (KG). The authors obtained understandable representations for undirected graphs through the use of latent variables.

Using an inner product decoder and a graph convolutional network (GCN) encoder, the researchers were able to predict linkages in citation networks with similar results as compared to the DeepWalk model's spectral clustering method. This model has the ability to seamlessly integrate node features, resulting in enhanced prediction performance (Liu et al., 2018).

Bansal recently introduced a comprehensive method for question answering that directly represents the elements and relationships in the text as memory slots. Instead of depending on an external knowledge graph, they relied on the idea that all the necessary information is in the text itself. This was achieved through the use of memory-based neural network models for language understanding (Sheth et al., 2019).

Munkhdalai introduced RelNet, an innovative method that enhances memory-augmented neural networks by incorporating a relational memory. This enables the model to effectively analyze and understand the connections between various entities in textual data, thus enhancing its reasoning capabilities. It is a comprehensive approach that can access and modify information stored in memory edges and slots (Zhang et al., 2020). The memory slots show the entities, and the edges, which are each shown as a vector, indicate the connections between these entities. Only by answering questions about the text one can get any sort of guidance (Wang et al., 2022).

8.4. NATURAL LANGUAGE TRANSLATION

Translating natural language has proven to be a challenging task that has yet to find a satisfactory solution, despite decades of research in the field of artificial intelligence. Conventional deep neural networks (DNNs) face certain limitations when it comes to addressing this issue. One such limitation is the need for inputs and targets to be encoded with fixed-dimensional vectors. This poses a significant constraint when dealing with sequences of varying lengths (Abbaszade et al., 2021).

In addition, certain tasks, like document classification, can be effectively accomplished using a bag-of-words approach that disregards word order. However, in the case of translation, the sequence of words becomes crucial. The phrases "Virus killed by raging scientist" and "Scientist killed by raging virus" share the same bag-of-words representations (Hirschberg & Manning, 2015).

The quality of translation is evaluated by utilizing the BLEU metric, which computes the geometric mean of the n-gram precisions for various values of n, typically ranging from 1 to a designated upper limit of 4. The BLEU score takes into consideration a penalty for brevity in order to address the issue of very short translations that may have high precision (Khan et al., 2020).

In contrast to traditional statistical machine translation, deep neural networks (DNNs) use a single neural network to effectively represent the probability distributions of both languages and optimize a translation score. Models typically transform a source sentence into a vector of a predetermined length using an encoder-decoder structure. A decoder then uses this vector to produce the matching translation (Green et al., 2015).

The best choice for processing the input sequence and condensing it into a single, high-dimensional vector is to use Recurrent Neural Networks (RNNs) equipped with Long Short-Term Memory (LSTM) units. A further LSTM then uses this vector to obtain the output sequence. The second LSTM depends on the input sequence, which sets it apart from a recurrent neural network language model.

Given the possibility of large time gaps between the inputs and their matching outputs, the LSTM is a perfect fit for this specific task due to its exceptional capacity to learn from data with complex temporal connections (Luo et al., 2021).

Sutskever (2014) used RNNs with long short-term memory (LSTM) units to produce remarkable results. In an English-to-French translation challenge, this method performed better than the traditional phrase-based machine translation system. The network was made up of two models: the first LSTM, which was an encoding model, and the second LSTM, a decoding model. They used a technique called stochastic gradient descent without incorporating momentum. Additionally, they made the decision to reduce the learning rate by half twice during each epoch, but only after the initial five epochs (Khurana et al., 2023). The approach demonstrates exceptional performance, surpassing the top-performing neural network NLP systems and achieving results on par with the best published outcomes of non-neural network approaches, even those that incorporate explicit domain expertise. Their system received a BLEU score of 36.5 when it was used to review candidate translations from another system.

The implementation required the use of eight GPUS, and the training process lasted for a duration of ten days. Each layer of the LSTM was allocated its own GPU, while an extra four GPUs were dedicated only to the computation of softmax. With 1,000 nodes in each of the LSTM's hidden layers, the implementation was created in C++. There were 160,000 words in the input vocabulary and 80,000 words in the output vocabulary. Weights were randomly initialized, and their values were limited to a specific range (Alarifi & Alwadain, 2020). For the English-to-French translation challenge, Bahdanau used auto-encoders and a variable-length encoding mechanism to obtain translation performance close to the current state-of-the-art phrase-based system (Figure 8.6). (The weighted geometric average of the probability inverses is called perplexity.)

$$e^{\sum xp(x)} \log p(x)$$

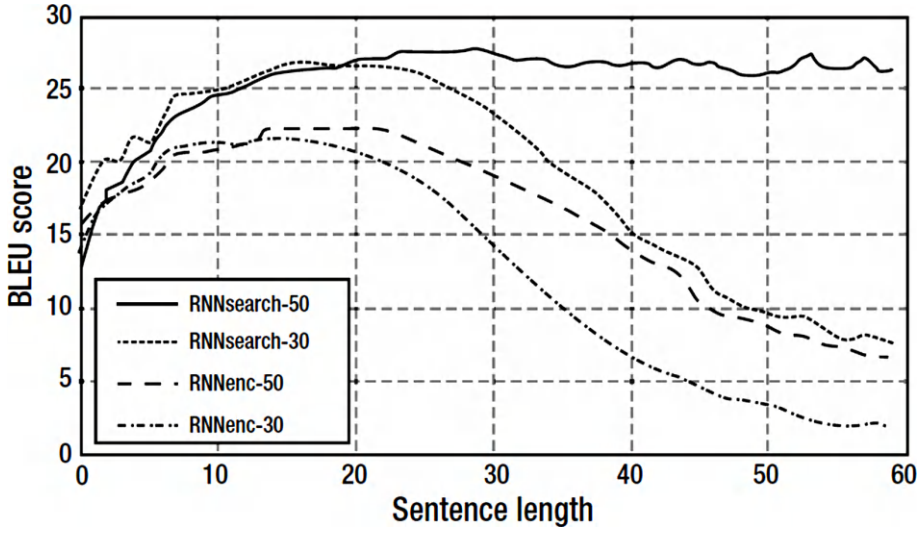


Figure 8.6. Illustration of the accuracy of the BLEU score for translation using sequence to sequence as a function of the duration of the sentence. Note the stability of model when dealing with long sentences.

Source: Simply, Creative Commons License.

The Google team has released a comprehensive document outlining the implementation of their new machine translation algorithm, which launched in November 2016. The system operates at the character level and makes use of bi-directionally stacked Long Short-Term Memory (LSTM) units with attention mechanisms. It follows the traditional encoder-decoder structure (Herzog & Wazinski, 1994).

The implementation is done using TensorFlow, and the team asserts that it achieves a level of translation performance that is comparable to that of humans. This holds true for various language pairs, including English to French, Spanish, and even Chinese, even when dealing with lengthy sentences. One limitation is that it can only translate individual sentences, making it unable to provide context for the entire document (Ali, 2016). Take a look at the original paper titled "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation."

8.5. MULTIMODAL LEARNING AND Q&A

The computer vision and natural language processing (NLP) field are becoming more closely connected. For example, the task of generating captions is far more challenging compared to tasks such as image categorization or object recognition. In addition to precisely describing the things in the image, the caption should also explain any activities or interactions between them.

A recent study has made significant progress in automatically generating detailed descriptions of images using natural language. Vinyals presents a model that utilizes a neural network to handle both image processing and language generation in a seamless manner. It produces coherent sentences in a human-like manner based on an input image. Check out "Show and Tell: A Neural Image Caption Generator." They were able to attain BLEU scores that were equivalent to those achieved by people on the COCO and Flickr datasets (Hannan et al., 2020).

In addition, modern approaches to natural language processing have been able to grasp the meaning of language by connecting it to the visual domain. The relationship between words and the interpretation of phrases can be likened to the relationship between images and words. Captions can be viewed as abstract representations of visuals. The latest methods for solving the image-caption, entailment, and hypernym problems entail creating distributed representations, or embeddings, from images or words. This efficient technique maps similar things to neighbor locations in a high-dimensional embedding space. Images are retrieved from text using a metric (typically the cosine), and vice versa.

With Vendrov's order-embedding technique, users can learn a mapping that retains order instead of distance in the embedding space, allowing them to take advantage of the partial-order structure of the visual-semantic hierarchy (Uppal et al., 2022).

The study demonstrated that order-embedding achieves exceptional outcomes in hypernymy prediction and caption-image retrieval, as well as delivering impressive performance in natural language inference. They conducted experiments using the Microsoft COCO data set, which consisted of over 120,000 images. Each image was accompanied by a minimum of five human-annotated captions.

They achieved a top one/top ten accuracy of 23.3 percent/65.0 percent in caption retrieval and 18.0 percent/57.6 percent in picture retrieval (Yuan et al., 2020).

8.6. SPEECH RECOGNITION

Voice to text translation is referred to as automatic speech recognition (ASR) difficulty. It is an old machine learning problem that was difficult to resolve using conventional methods that depended on Markov chain processes (Gaikwad et al., 2010).

The data sets Switchboard and TIMIT serve as the reference standards for this problem. TIMIT comprises high-quality recordings of 630 speakers representing eight prominent dialects of American English. Each speaker reads ten sentences that are phonetically dense. Every utterance in the TIMIT corpus has a 16-bit, 16 kHz speech waveform file along with time-aligned phonetic, orthographic, and word transcriptions.

The initial use of deep belief networks (DBNs) to the TIMIT data set yielded an accuracy rate of approximately 23%. Currently, the utilization of a DBN with post-regularization on the final layer yields an accuracy level of 16.5%. Many mobile applications now heavily depend on voice recognition due to its exceptional accuracy (Abdel-Hamid et al., 2014).

Graves revolutionized the field by using a cutting-edge technique known as deep bidirectional LSTM, which yielded an impressive 17.7 percent error rate reduction in the TIMIT database. They utilized a comprehensive approach to accurately translate sequences using recurrent neural networks. These methods remove the requirement of presegmenting the acoustic data by directly enhancing the probability of the desired sequence using the input sequence. With the help of the auditory training data, they are able to acquire an implicit language model.

Recently, an ASR model for text-to-voice translation was proposed by a Baidu team. Deep learning substitutes a single neural model for feature extraction modules, improving the algorithm's performance. In a number of languages, the Deep Speech 2 algorithm comes close to human accuracy. End-to-end deep learning using a bidirectional RNN that was trained in both clean and noisy settings forms the basis of this system (Morgan & Bourlard, 1995).

11,940 hours of Mandarin speech and 9,400 hours of English speech were used to train the speech system. During training, the data was enhanced through the use of data synthesis. Tens of exaFLOPs are needed to train a single model at these scales, and executing them on a single GPU would take three to six weeks (Malik et al., 2021).

Microsoft revealed a new algorithm in August 2017 that brought the industry-standard Switchboard test's error rate down to 5.1% for accurate audio transcription.

On the other hand, a single human transcriptionist makes 5.9% of errors on average. It made use of both bidirectional LSTM and CNNs.

However, a Temple study shows that Google leads in the accuracy of voice-activated personal assistants. See Figure 8.7.

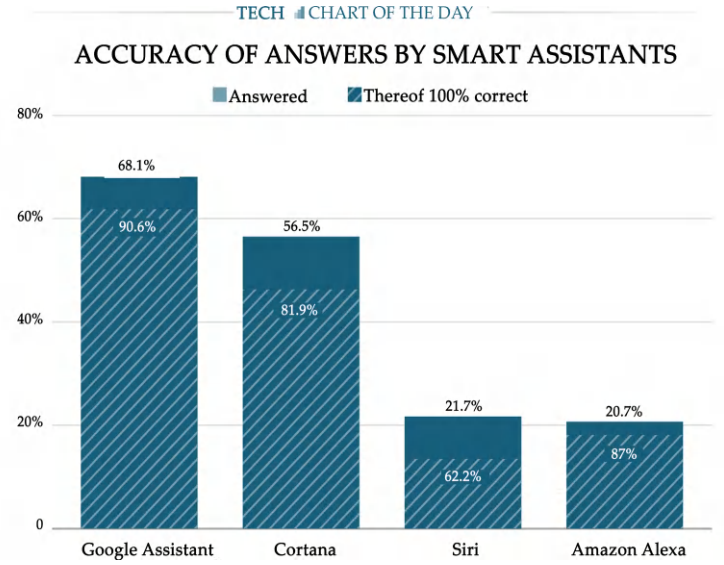


Figure 8.7. Illustration of the accuracy of several personal assistants.

Source: Nicolas Usunier, Creative Commons License.

DeepMind has recently made a release WaveNet is a remarkable product that excels in voice synthesis, commonly known as text-to-speech (TTS) or speech synthesis. It offers exceptional quality in this process. Conventional models depend on concatenative TTS, which involves recording numerous short speech fragments from one speaker and then combining them to create full utterances. WaveNet meticulously recreates the audio signal's raw waveform, producing surprisingly natural speech. WaveNet has the capability to effectively model a wide range of audio, encompassing various genres such as music (Lippmann, 1997).

ACTIVITY 8.1.

Objective: Explore practical applications of deep learning in natural language processing (NLP).

Activity Steps:

- Introduce parsing, distributed representations, knowledge graphs, translation, multimodal learning, QA systems, speech recognition, and pooling.
- Explore the experiment code for text classification, which is found here: https://keras.io/examples/nlp/text_classification_from_scratch/.
- Divide into groups, analyze and explain each step in the code, and test it with new examples. Share findings, discuss challenges, and ethical implications.

SUMMARY

- The chapter delves into the main methodologies and advancements in these fields. The discussion starts with an exploration of parsing techniques, which entail the analysis of language structures through the use of deep learning models.
- The next section explores the distributed representations, showcasing their remarkable ability to capture semantic relationships within language data. It discusses the concept of knowledge representation using graphs and highlights the role of deep learning in organizing and retrieving complex networks.
- The chapter additionally discusses natural language translation, highlighting the impact of deep learning models on machine translation tasks. Topics such as multimodal learning and question answering systems are addressed. It delves into the complexities of speech recognition techniques, with a particular focus on how deep learning can be used to enhance the precision and speed of converting spoken language into written text.

REVIEW QUESTIONS

1. What is parsing in natural language processing, and how do deep learning techniques enhance it?
2. How do distributed representations benefit natural language understanding? Provide a brief example.
3. Explain the role of knowledge graphs in deep learning for natural language processing.
4. How has deep learning transformed machine translation? Provide a key improvement.
5. Describe the concept of multimodal learning in the context of AI.

MULTIPLE CHOICE QUESTIONS

1. Which deep learning technique is primarily used for syntactic analysis of language structures?

a. Knowledge representation
b. Distributed representations
c. Parsing
d. Multimodal learning

2. What is the primary benefit of distributed representations in natural language processing?

a. Improved syntactic analysis
b. Efficient knowledge retrieval

c. Capturing semantic relationships

d. Enhancing speech recognition

3. In which application does deep learning often employ knowledge representation using graphs?

a. Natural language translation

b. Speech recognition

c. Question answering systems

d. Knowledge retrieval

4. Which task involves the use of deep learning models to convert spoken language into text?

a. Parsing

b. Multimodal learning

c. Speech recognition

d. Distributed representations

5. What is the primary role of pooling operations in deep learning for natural language processing?

a. Enhancing feature extraction

b. Improving syntactic analysis

c. Optimizing knowledge representation

d. Enabling multimodal learning

Answers to Multiple Questions

1. (c); 2. (c); 3. (d); 4. (c); 5. (a).

REFERENCES

1. Abbaszade, M., Salari, V., Mousavi, S. S., Zomorodi, M., & Zhou, X. (2021). Application of quantum natural language processing for language translation. IEEE Access, 9, 130434–130448.
2. Abdel-Hamid, O., Mohamed, A. R., Jiang, H., Deng, L., Penn, G., & Yu, D. (2014). Convolutional neural networks for speech recognition. IEEE/ACM Transactions on Audio, Speech, and Language Processing, 22(10), 1533–1545.
3. Alarifi, A., & Alwadain, A. (2020). An optimized cognitive-assisted machine translation approach for natural language processing. Computing, 102(3), 605–622.
4. Ali, M. A. (2016). Artificial intelligence and natural language processing: The Arabic corpora in online translation software. International Journal of Advanced and Applied Sciences, 3(9), 59–66.

5. Alshahrani, M., Khan, M. A., Maddouri, O., Kinjo, A. R., Queralt-Rosinach, N., & Hoehndorf, R. (2017). Neuro-symbolic representation learning on biological knowledge graphs. Bioinformatics, 33(17), 2723–2730.
6. Berridge, K. C., & Robinson, T. E. (2003). Parsing reward. Trends in Neurosciences, 26(9), 507–513.
7. Bowers, J. S. (2002). Challenging the widespread assumption that connectionism and distributed representations go hand-in-hand. Cognitive Psychology, 45(3), 413–445.
8. Chadha, N., Gangwar, R. C., & Bedi, R. (2015). Current challenges and application of speech recognition process using natural language processing: A survey. International Journal of Computer Applications, 131(11), 28–31.
9. Chen, X., Jia, S., & Xiang, Y. (2020). A review: Knowledge reasoning over knowledge graph. Expert Systems with Applications, 141, 112948.
10. Choudhary, N., Rao, N., Katariya, S., Subbian, K., & Reddy, C. (2021). Probabilistic entity representation model for reasoning over knowledge graphs. Advances in Neural Information Processing Systems, 34, 23440–23451.
11. Chrisman, L. (1991). Learning recursive distributed representations for holistic computation. Connection Science, 3(4), 345–366.
12. Fan, M., Zhou, Q., Zheng, T. F., & Grishman, R. (2017). Distributed representation learning for knowledge graphs with entity descriptions. Pattern Recognition Letters, 93, 31–37.
13. Fodor, J. D. (1978). Parsing strategies and constraints on transformations. Linguistic Inquiry, 9(3), 427–473.
14. Gaikwad, S. K., Gawali, B. W., & Yannawar, P. (2010). A review on speech recognition technique. International Journal of Computer Applications, 10(3), 16–24.
15. Gelder, T. V. (1992). Defining 'distributed representation'. Connection Science, 4(3–4), 175–191.
16. Green, S., Heer, J., & Manning, C. D. (2015). Natural language translation at the intersection of AI and HCI. Communications of the ACM, 58(9), 46–53.
17. Hannan, D., Jain, A., & Bansal, M. (2020). Manymodalqa: Modality disambiguation and QA over diverse inputs. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 34, No. 5, pp. 7879–7886).
18. Herzog, G., & Wazinski, P. (1994). Visual TRAnslator: Linking perceptions and natural language descriptions. Artificial Intelligence Review, 8, 175–187.
19. Hirschberg, J., & Manning, C. D. (2015). Advances in natural language processing. Science, 349(6245), 261–266.
20. Howard, M. W., & Kahana, M. J. (2002). A distributed representation of temporal context. Journal of Mathematical Psychology, 46(3), 269–299.
21. Hummel, J. E., & Holyoak, K. J. (1997). Distributed representations of structure: A theory of analogical access and mapping. Psychological Review, 104(3), 427.
22. Izbassarova, A., Duisembay, A., & James, A. P. (2020). Speech recognition application using deep learning neural network. In Deep Learning Classifiers with Memristive Networks: Theory and Applications (pp. 69–79).

23. Ji, S., Pan, S., Cambria, E., Marttinen, P., & Philip, S. Y. (2021). A survey on knowledge graphs: Representation, acquisition, and applications. IEEE Transactions on Neural Networks and Learning Systems, 33(2), 494–514.

24. Kennedy, M. (2016). Parsing the practice of teaching. Journal of Teacher Education, 67(1), 6–17.

25. Khan, N. S., Abid, A., & Abid, K. (2020). A novel natural language processing (NLP)–Based machine translation model for English to Pakistan sign language translation. Cognitive Computation, 12, 748–765.

26. Khurana, D., Koli, A., Khatter, K., & Singh, S. (2023). Natural language processing: State of the art, current trends and challenges. Multimedia Tools and Applications, 82(3), 3713–3744.

27. Lauriola, I., Lavelli, A., & Aiolli, F. (2022). An introduction to deep learning in natural language processing: Models, techniques, and tools. Neurocomputing, 470, 443–456.

28. Li, H. (2018). Deep learning for natural language processing: Advantages and challenges. National Science Review, 5(1), 24–26.

29. Lippmann, R. P. (1997). Speech recognition by machines and humans. Speech Communication, 22(1), 1–15.

30. Liu, W., Liu, J., Wu, M., Abbas, S., Hu, W., Wei, B., & Zheng, Q. (2018). Representation learning over multiple knowledge graphs for knowledge graphs alignment. Neurocomputing, 320, 12–24.

31. Luo, Y., Tang, N., Li, G., Tang, J., Chai, C., & Qin, X. (2021). Natural language to visualization by neural machine translation. IEEE Transactions on Visualization and Computer Graphics, 28(1), 217–226.

32. Malik, M., Malik, M. K., Mehmood, K., & Makhdoom, I. (2021). Automatic speech recognition: A survey. Multimedia Tools and Applications, 80, 9411–9457.

33. Momma, S., & Phillips, C. (2018). The relationship between parsing and generation. Annual Review of Linguistics, 4(1), 233–254.

34. Morgan, N., & Bourlard, H. (1995). Continuous speech recognition. IEEE Signal Processing Magazine, 12(3), 24–42.

35. Nicholson, D. N., & Greene, C. S. (2020). Constructing knowledge graphs and their biomedical applications. Computational and Structural Biotechnology Journal, 18, 1414–1428.

36. Nickel, M., Murphy, K., Tresp, V., & Gabrilovich, E. (2015). A review of relational machine learning for knowledge graphs. Proceedings of the IEEE, 104(1), 11–33.

37. Niu, G., Zhang, Y., Li, B., Cui, P., Liu, S., Li, J., & Zhang, X. (2020). Rule-guided compositional representation learning on knowledge graphs. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 34, No. 3, pp. 2950–2958).

38. Peng, C., Xia, F., Naseriparsa, M., & Osborne, F. (2023). Knowledge graphs: Opportunities and challenges. Artificial Intelligence Review, 56(11), 13071–13102.

39. Rissman, J., & Wagner, A. D. (2012). Distributed representations in memory: Insights from functional brain imaging. Annual Review of Psychology, 63(1), 101–128.

40. Sarikaya, R., Hinton, G. E., & Deoras, A. (2014). Application of deep belief networks for natural language understanding. IEEE/ACM Transactions on Audio, Speech, and Language Processing, 22(4), 778–784.

41. Sheth, A., Padhee, S., & Gyrard, A. (2019). Knowledge graphs and knowledge networks: The story in brief. IEEE Internet Computing, 23(4), 67–75.

42. Sikos, L. F., & Philp, D. (2020). Provenance-aware knowledge representation: A survey of data models and contextualized knowledge graphs. Data Science and Engineering, 5, 293–316.

43. Sowa, J. F. (1992). Conceptual graphs as a universal knowledge representation. Computers & Mathematics with Applications, 23(2–5), 75–93.

44. Tiwari, S., Al-Aswadi, F. N., & Gaurav, D. (2021). Recent trends in knowledge graphs: Theory and practice. Soft Computing, 25, 8337–8355.

45. Uppal, S., Bhagat, S., Hazarika, D., Majumder, N., Poria, S., Zimmermann, R., & Zadeh, A. (2022). Multimodal research in vision and language: A review of current and emerging trends. Information Fusion, 77, 149–171.

46. Wang, E., Yu, Q., Chen, Y., Slamu, W., & Luo, X. (2022). Multi-modal knowledge graphs representation learning via multi-headed self-attention. Information Fusion, 88, 78–85.

47. Yan, X., Jian, F., & Sun, B. (2021). SAKG-BERT: Enabling language representation with knowledge graphs for Chinese sentiment analysis. IEEE Access, 9, 101695–101701.

48. Yuan, Z., Sun, S., Duan, L., Li, C., Wu, X., & Xu, C. (2020). Adversarial multimodal network for movie story question answering. IEEE Transactions on Multimedia, 23, 1744–1756.

49. Zhang, Z., Cao, L., Chen, X., Tang, W., Xu, Z., & Meng, Y. (2020). Representation learning of knowledge graphs with entity attributes. IEEE Access, 8, 7435–7441.

50. Zheng, G., & Callan, J. (2015). Learning to reweight terms with distributed representations. In Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval (pp. 575–584).

Index

Elements of Ecology

Ecology encompasses the study of how individual organisms come together to form populations, how these populations interact within communities, and how these communities, in turn, fit into the larger framework of ecosystems. A specialized branch of ecology, known as autecology, focuses specifically on the ways in which single species adapt to their surroundings. This adaptation is achieved through a variety of mechanisms, including biochemical, morphological, and physiological changes, tailored to meet the demands of their environment. In the plant kingdom, the evidence of adaptation is abundant and diverse. For instance, certain desert plants have evolved thorns in place of leaves, a modification aimed at reducing water loss in arid conditions. The development of a wide spectrum of complex chemical defenses is another testament to plant adaptation, with examples including the painful sting of nettles, the toxic principles of hemlock, and the irritant properties of poison ivy. Additionally, plants such as succulents have adapted to survive in water-scarce environments by developing specialized structures for water storage, showcasing the range of adaptative strategies from the molecular level to the whole-plant level. The animal kingdom is equally rich in examples of adaptation to environmental pressures. Antarctic fishes, for example, have evolved antifreeze proteins to survive in icy waters, while seals have developed specialized cardiovascular systems that enable them to dive for extended periods without oxygen. The natural world is also replete with chemical strategies for survival, including the potent tetrodotoxin used for defense and the venomous attacks of snakes. Furthermore, the discovery of giant variants of marine species near deep-sea hydrothermal vents highlights the adaptability of life in extreme environments.

The distribution of organisms across the globe is largely influenced by environmental factors. While it might seem that a single factor can determine the success or distribution of a species, the reality is often more complex, involving a multitude of interacting elements. The study of ecological races in plants, for instance, has revealed that adaptations to environmental variables such as altitude involve a complex interplay of multiple genes, affecting traits like frost tolerance and flowering time. Advances in our understanding of these adaptations are being propelled forward by integrating whole-organism studies with research at the cellular and subcellular levels, aided by the latest developments in molecular biology and subcellular physiology. The impact of these scientific advancements extends beyond academic interest. They hold the potential to revolutionize crop productivity by applying a deeper understanding of physiological and genetic adaptation mechanisms. Moreover, these insights contribute significantly to our comprehension of evolutionary processes and the broader principles of ecology. The scope of ecology includes not only the relationship between organisms and their physical surroundings but also their interactions with members of their own and other species. These interactions are crucial in defining the roles organisms play within the natural world. Weaving together the study of these dual aspects, ecology provides a comprehensive picture of how plant and animal populations are regulated and structured within their environments, offering a holistic view of the natural world's complexity and interconnectivity.

The aim of "Elements of Ecology" is to offer a comprehensive introduction to the field of ecology, presenting the essential concepts, principles, and processes that govern the interactions within and among organisms and their environments. This book is designed primarily for undergraduate students majoring in biology, environmental science, and related disciplines who are embarking on their exploration of ecological systems and the complex relationships that sustain life on Earth. Integrating the latest topics with foundational ecological theories, the book seeks to enhance students' understanding of how ecosystems function, the role of biodiversity, the impact of human activities on natural habitats, and the strategies for conservation and sustainable management of ecosystems. "Elements of Ecology" covers a wide range of topics, including population dynamics, community interactions, ecosystem energetics, biogeochemical cycles, and global environmental changes, aiming to provide a solid foundation in ecology that is both scientifically rigorous and accessible. The readers will benefit from clear explanations, illustrative diagrams, and real-world examples that demonstrate the relevance of ecological principles in addressing environmental challenges. Through this book, students are encouraged to develop a deeper appreciation for the complexity of ecological systems and the importance of preserving the natural world for future generations.

About the Editor

Lindsay Patterson is an ecologist, wildlife researcher, and copy editor passionate about animals, conservation, and the power of the written, and the spoken word. She has an undergraduate degree in English and Philosophy studies, an honors degree in Environmental Sciences, and a doctorate in Ecology, which has deepened her understanding of working hard for what you love. Her passion for wildlife research began with the Chacma baboons of South Africa's Table Mountain National Park and led her to the Cederberg Mountain's elusive Cape leopards, Madagascar's unique diversity of lemurs, frogs, and snakes, and KwaZulu-Natal's precocious vervet monkeys. In her 10-year career as a natural history researcher, she worked on over 200 nature documentaries for award-winning production houses, including Love Nature, National Geographic, Discovery, and Smithsonian. In her downtime, she loves camping in remote spaces and is an avid reader of everything.

TAP
Toronto Academic Press

ISBN 978-1-77956-715-4