EXPERT INSIGHT

Mastering PyTorch

Create and deploy deep learning models from CNNs to multimodal models, LLMs, and beyond

Second Edition

packt

Ashish Ranjan Jha

Mastering PyTorch

Second Edition

Create and deploy deep learning models from CNNs to multimodal models, LLMs, and beyond

Ashish Ranjan Jha

packt

Mastering PyTorch

Second Edition

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

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

Publishing Product Manager: Bhavesh Amin

Acquisition Editors – Peer Reviews: Gaurav Gavas, Jane D'Souza

Project Editor: Ankit Maroli

Content Development Editor: Soham Amburle

Copy Editor: Safis Editing

Technical Editor: Gaurav Gavas

Proofreader: Safis Editing

Indexer: Tejal Soni

Presentation Designer: Pranit Padwal

Developer Relations Marketing Executive: Monika Sangwan

First published: February 2021
Second Edition: May 2024

Production reference: 1280524

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-80107-430-8

`www.packt.com`

Contributors

About the author

Ashish Ranjan Jha studied electrical engineering at IIT Roorkee, computer science at École Polytechnique Fédérale de Lausanne (EPFL), and he also completed his MBA at Quantic School of Business, with a distinction in all three degrees. He has worked for bigger tech companies like Oracle and Sony, and recent tech unicorns – Revolut and Tractable, in the fields of data science, machine learning and artificial intelligence. He currently works as head of ML and AI at XYZ Reality, based in London (a construction tech start-up where construction meets AR/VR meets ML/AI to enable real-time data driven construction intelligence). He is also an advisor to SUIND, an agritech startup that uses drones for intelligence. Along with that, he has also authored a book, *Fight Fraud with Machine Learning*.

> *To my mother, Rani Jha, and my father, Bibhuti Bhushan Jha, for their sacrifices, constant support, and for being the driving forces of my career, as well as my life. Without their love, none of this would matter. To my sisters, Shalini, Sushmita, and Nivedita, for always guiding me in life. Finally, to Packt and the entire team that is obliging enough to publish me.*

About the reviewers

Ritobrata Ghosh is a deep learning researcher with three years of experience working in computer vision. His research interests are computational neuroscience and scientific machine learning.

He was awarded the Google OSS prize (2022). He also co-created Dall-E Mini (now known as Craiyon). Apart from that, he tinkers in mathematics, functional programming, and edge AI. He has an M.Sc. in computer science, and his hobbies are reading, swimming, and motorbike riding through the countryside.

He has a passion for perpetual learning and teaching. He has also worked as a voluntary technical reviewer of a few chapters of the following books:

- *Introduction to Generative AI* by Numa Dhamani and Maggie Engler (Manning, 9781633437197)
- *Deep Learning with JAX* by Grigory Sapunov (Manning, 9781633438880)

> *I would like to thank my friends Suparna Roy and Harihar Biswas, and my parents for their continuous support for the duration of this project. Without their support, my work would not have been possible.*

Sheallika Singh is an expert in deep learning and an advisor to multiple machine learning startups. Currently, she is a staff machine learning engineer, responsible for developing personalization models used by billions of users worldwide. She has also played a pivotal role in advancing self-driving car technology through her previous work as a machine learning engineer. She has presented published research at prominent ML conferences and also serves as a program committee member for top-tier conferences. Before entering the industry, Sheallika conducted research on font-free character recognition. She holds a master's degree in data science from Columbia University and a Bachelor of Science degree in mathematics and scientific computing, with a minor in industrial management, from the Indian Institute of Technology, Kanpur.

Learn more on Discord

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

`https://packt.link/mastorch`

Table of Contents

Chapter 3: Combining CNNs and LSTMs 91

Chapter 4: Deep Recurrent Model Architectures 111

Chapter 5: Advanced Hybrid Models 141

Chapter 8: Neural Style Transfer 227

Chapter 9: Deep Convolutional GANs 245

Chapter 10: Image Generation Using Diffusion 269

Chapter 11: Deep Reinforcement Learning 299

Chapter 12: Model Training Optimizations 325

Chapter 13: Operationalizing PyTorch Models into Production 347

Chapter 14: PyTorch on Mobile Devices 391

Chapter 15: Rapid Prototyping with PyTorch 421

Chapter 16: PyTorch and AutoML 443

Chapter 17: PyTorch and Explainable AI 459

Chapter 18: Recommendation Systems with PyTorch 475

Chapter 19: PyTorch and Hugging Face 497

Other Books You May Enjoy 519

Index 523

Preface

Deep learning is driving the AI revolution and PyTorch is making it easier than ever before for anyone to build deep learning applications. This book will help you uncover expert techniques and gain insights to get the most out of your data and build complex neural network models.

The book starts with a quick overview of PyTorch and explores **convolutional neural network** (**CNN**) architectures for image classification. Similarly, you will explore **recurrent neural network** (**RNN**) architectures as well as Transformers and use them for sentiment analysis. Next, you will learn how to create arbitrary neural network architectures and build **Graph neural networks** (**GNNs**). As you advance, you'll apply **deep learning** (**DL**) across different domains such as music, text, and image generation using generative models including **Generative adversarial networks** (**GANs**) and diffusion.

Next, you'll build and train your own deep reinforcement learning models in PyTorch, as well as interpreting DL models. You will not only learn how to build models but also how to deploy them into production and to mobile devices (Android and iOS) using expert tips and techniques. Next, you will master the skills of training large models efficiently in a distributed fashion, searching neural architectures effectively with AutoML, as well as rapidly prototyping models using fastai. You'll then create a recommendation system using PyTorch. Finally, you'll use major Hugging Face libraries together with PyTorch to build cutting edge **artificial intelligence** (**AI**) models.

By the end of this PyTorch book, you'll be well equipped to perform complex deep learning tasks using PyTorch to build smart AI models.

Who this book is for

This book is for data scientists, machine learning researchers, and deep learning practitioners looking to implement advanced deep learning paradigms using PyTorch 2.x. Working knowledge of deep learning with Python programming is required.

What this book covers

Chapter 1, *Overview of Deep Learning Using PyTorch*, includes brief notes on various deep learning terminologies and concepts useful for understanding later parts of this book. This chapter also gives a quick overview of PyTorch in contrast with TensorFlow as a language and tools that will be used throughout this book for building deep learning models. Finally, we train a neural network model using PyTorch.

Chapter 2, Deep CNN Architectures, is a rundown of the most advanced deep CNN model architectures that have been developed in recent years. We use PyTorch to create many of these models and train them for appropriate tasks.

Chapter 3, Combining CNNs and LSTMs, walks through an example where we build a neural network model with a CNN and LSTM that generates text/captions as output when given images as inputs using PyTorch.

Chapter 4, Deep Recurrent Model Architectures, goes through recent advancements in recurrent neural architectures, specifically RNNs, LSTMs, and GRUs. Upon completion, you will be able to create complex recurrent architecture in PyTorch.

Chapter 5, Advanced Hybrid Models, discusses some advanced, unique hybrid neural architectures such as the Transformers that have revolutionized the world of natural language processing. This chapter also discusses RandWireNNs, taking a peek into the world of neural architecture search, using PyTorch.

Chapter 6, Graph Neural Networks, walks us through the basic concepts behind GNNs, different kinds of graph learning tasks, and different types of GNN model architectures. The chapter then dives deep into a few of those architectures, namely **Graph Convolutional Networks** (**GCNs**) and **Graph Attention Networks** (**GATs**). This chapter uses PyTorch Geometric as the library of choice for building GNNs in PyTorch.

Chapter 7, Music and Text Generation with PyTorch, demonstrates the use of PyTorch to create deep learning models that can compose music and write text with practically nothing being provided to them at runtime.

Chapter 8, Neural Style Transfer, discusses a special type of CNN model that can mix multiple input images and generate artistic-looking arbitrary images.

Chapter 9, Deep Convolutional GANs, explains GANs and trains one using PyTorch on a specific task.

Chapter 10, Image Generation Using Diffusion, implements a diffusion model from scratch as a state-of-the-art text-to-image generation model, using PyTorch.

Chapter 11, Deep Reinforcement Learning, explores how PyTorch can be used to train agents on a deep reinforcement learning task, such as a player in a video game.

Chapter 12, Model Training Optimizations, explores how to efficiently train large models with limited resources through distributed training as well as mixed precision training practices in PyTorch. By the end of this chapter, you will have mastered the skill of training large models efficiently using PyTorch.

Chapter 13, Operationalizing PyTorch Models into Production, runs through the process of deploying a deep learning model written in PyTorch into a real production system using Flask and Docker, as well as TorchServe. Then you'll learn how to export PyTorch models both using TorchScript and ONNX. You'll also learn how to ship PyTorch code as a C++ application. Finally, you'll learn how to use PyTorch on some of the popular cloud computing platforms.

Chapter 14, *PyTorch on Mobile and Embedded Devices*, walks through the process of using various pre-trained PyTorch models and deploying them on different mobile operating systems – Android and iOS.

Chapter 15, *Rapid Prototyping with PyTorch*, discusses various tools and libraries such as fastai and PyTorch Lightning that make the process of model training in PyTorch several times faster. This chapter also explains how to profile PyTorch code to understand resource utilization.

Chapter 16, *PyTorch and AutoML*, walks through setting up ML experiments effectively using AutoML and Optuna with PyTorch.

Chapter 17, *PyTorch and Explainable AI*, focuses on making machine learning models interpretable to a layman using tools such as Captum, combined with PyTorch.

Chapter 18, *Recommendation Systems with PyTorch*, builds a deep-learning-based movie recommendation system from scratch using PyTorch.

Chapter 19, *PyTorch and Hugging Face*, discusses how to use Hugging Face libraries such as Transformers, Accelerate, Optimum, and so on, with PyTorch to build cutting-edge multi-modal AI models.

To get the most out of this book

To fully benefit from this book, it is necessary that you meet the following prerequisites and recommendations:

- Hands-on Python experience as well as basic knowledge of PyTorch is expected. Because most exercises in this book are in the form of notebooks, a working experience with Jupyter notebooks is expected.
- Some of the exercises in some of the chapters might require a GPU for faster model training, and therefore having an NVIDIA GPU is a plus.
- Finally, having registered accounts with cloud computing platforms such as AWS, Google Cloud, and Microsoft Azure will be helpful to navigate parts of *Chapter 13* as well as to facilitate distributed training in *Chapter 12* over several virtual machines.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/arj7192/MasteringPyTorchV2`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://packt.link/gbp/9781801074308`.

Conventions used

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

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: “Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system.”

A block of code is set as follows:

```
def forward(self, source):
    source = self.enc(source) * torch.sqrt(self.num_inputs)
    source = self.position_enc(source)
    op = self.enc_transformer(source, self.mask_source)
    op = self.dec(op)
    return op
```

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

```
def forward(self, source):
    source = self.enc(source) * torch.sqrt(self.num_inputs)
    source = self.position_enc(source)
    op = self.enc_transformer(source, self.mask_source)
    op = self.dec(op)
    return op
```

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

```
loss improvement on epoch: 1
[001/200] train: 1.1996 - val: 1.0651
loss improvement on epoch: 2
[002/200] train: 1.0806 - val: 1.0494
...
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “Select **System info** from the **Administration** panel.”

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packtpub.com/submit-errata`, click **Submit Errata**, and fill in the form.

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

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

Share your thoughts

Once you've read *Mastering Pytorch, Second Edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

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

Download a free PDF copy of this book

Thanks for purchasing this book!

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

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

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

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

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

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:

   `https://packt.link/free-ebook/9781801074308`

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

1

Overview of Deep Learning Using PyTorch

Deep learning is a class of machine learning methods that has revolutionized the way computers/machines are used to build automated solutions for real-life problems in a way that wasn't possible before. Deep learning uses large amounts of data to learn non-trivial relationships between inputs and outputs in the form of complex nonlinear functions. Some of the inputs and outputs, as demonstrated in *Figure 1.1*, could be the following:

- **Input**: An image of a text; **output**: Text
- **Input**: Text; **output**: A natural voice speaking the text
- **Input**: A natural voice speaking the text; **output**: Transcribed text

And so on. (The above examples deliberately exclude tabular input data because gradient boosted trees (XGBoost, LightGBM, CatBoost) still outperform deep learning on such data.)

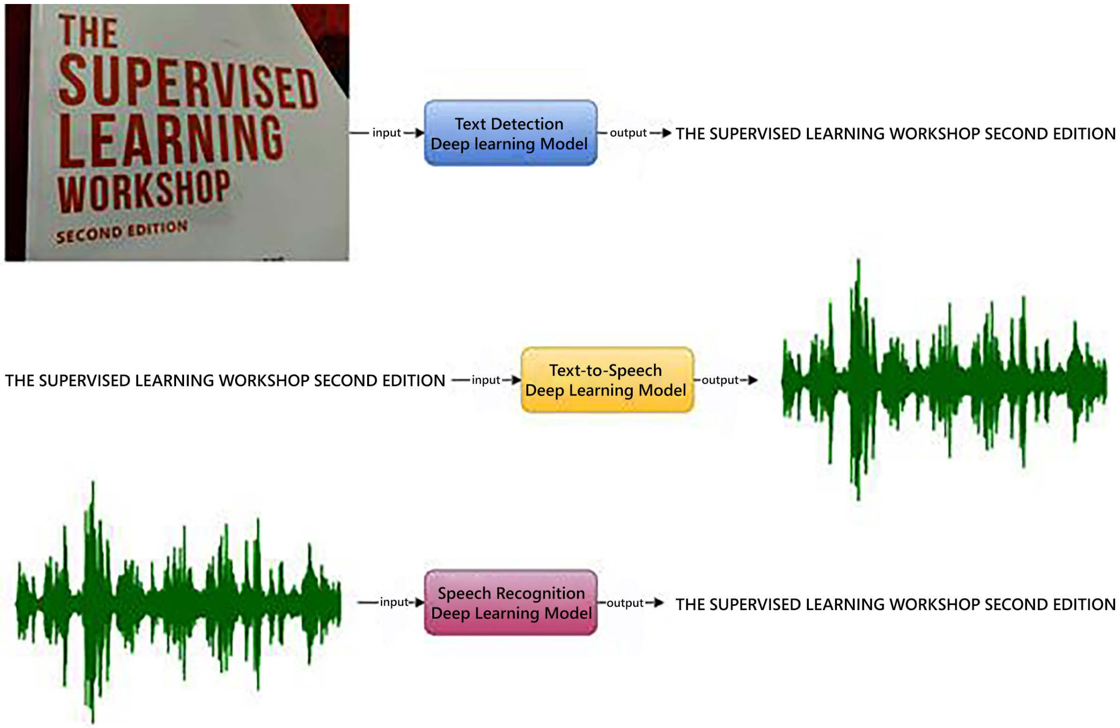

THE SUPERVISED LEARNING WORKSHOP SECOND EDITION
input
Text Detection Deep learning Model
output
THE SUPERVISED LEARNING WORKSHOP SECOND EDITION
THE SUPERVISED LEARNING WORKSHOP SECOND EDITION
input
Text-to-Speech Deep Learning Model
output
input
Speech Recognition Deep Learning Model
output
THE SUPERVISED LEARNING WORKSHOP SECOND EDITION

Figure 1.1: Deep learning model examples

Deep neural networks involve a lot of mathematical computations, linear algebraic equations, nonlinear functions, and various optimization algorithms. In order to build and train a deep neural network from scratch using a programming language such as Python, it would require us to write all the necessary equations, functions, and optimization schedules. Furthermore, the code would have to be written such that large amounts of data can be loaded efficiently, and training can be performed in a reasonable amount of time. This amounts to implementing several lower-level details each time we build a deep learning application.

Deep learning libraries such as Theano and TensorFlow, among various others, have been developed over the years to abstract these details out. PyTorch is one such Python-based deep learning library that can be used to build deep learning models.

TensorFlow was introduced as an open source deep learning Python (and C++) library by Google in late 2015, which revolutionized the field of applied deep learning. Facebook, in 2016, responded with its own open source deep learning library and called it Torch. Torch was initially used with a scripting language called Lua, and soon enough, the Python equivalent emerged called PyTorch. Around the same time, Microsoft released its own library – CNTK. Amidst the hot competition, PyTorch has been growing fast to become one of the most used deep learning libraries.

This book is meant to be a hands-on resource on some of the most advanced deep learning problems, how they are solved using complex deep learning architectures, and how PyTorch can be effectively used to build, train, and evaluate these complex models.

While the book keeps PyTorch at the center, it also includes comprehensive coverage of some of the most recent and advanced deep learning models. The book is intended for data scientists, machine learning engineers, or researchers who have a working knowledge of Python and who, preferably, have used PyTorch before. For those who are not familiar with PyTorch or are familiar with TensorFlow but not PyTorch, I recommend spending more time on this chapter alongside other resources such as basic tutorials on Torch's website to get comfortable with the basics of PyTorch first.

Due to the hands-on nature of this book, it is highly recommended to try the examples in each chapter by yourself on your computer to become proficient in writing PyTorch code. We begin with this introductory chapter and subsequently explore various deep learning problems and model architectures that will expose the various functionalities PyTorch has to offer.

This chapter will review some of the concepts behind deep learning and will provide a brief overview of the PyTorch library. For those familiar with TensorFlow who are looking to transition to PyTorch, we will also see how PyTorch's APIs differ from TensorFlow's at various points in this chapter. We will conclude this chapter with a hands-on exercise where we train a deep learning model using PyTorch.

The following topics will be covered in this chapter:

- A refresher on deep learning
- Exploring the PyTorch library in contrast to TensorFlow
- Training a neural network using PyTorch

A refresher on deep learning

Neural networks are a sub-type of machine learning methods that are inspired by the structure and function of the biological brain, such as the biological neuron shown in *Figure 1.2*. In neural networks, each computational unit, analogically called a neuron, is connected to other neurons in a layered fashion. When the number of such layers is more than two, the neural network thus formed is called a **Deep Neural Network** (**DNN**). Such models are generally called deep learning models.

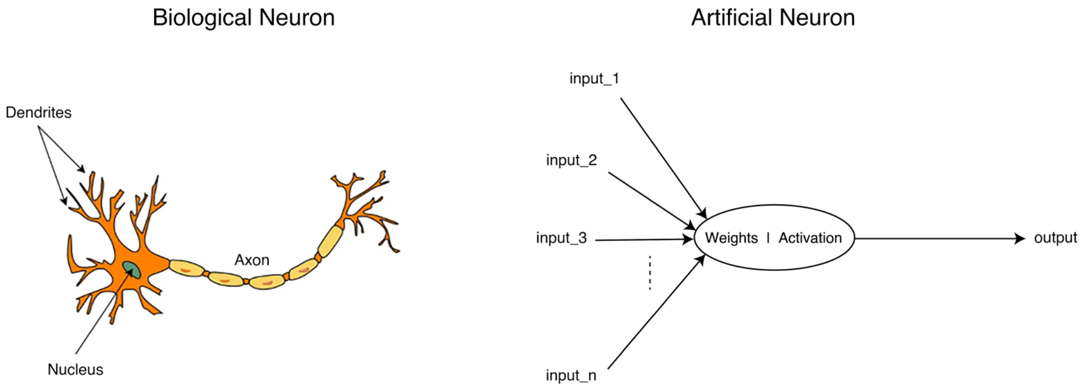


Figure 1.2: Artificial neuron inspired by biological neuron. (Biological neuron image by: https://pixabay.com/users/clker-free-vector-images-3736)

Deep learning models have been proven superior to other classical machine learning models because of their ability to learn highly complex relationships between input data and the output (ground truth). In recent times, deep learning has gained a lot of attention, and rightly so, primarily because of the following two reasons:

- The availability of powerful computing machines, including GPUs
- The availability of huge amounts of data

Owing to Moore's law, which states that the processing power of computers will double every two years, we are now living in a time when deep learning models with several thousands of layers can be trained within a realistic and reasonably short amount of time. At the same time, with the exponential increase in the use of digital devices everywhere, our digital footprint has exploded, resulting in gigantic amounts of data being generated across the world every moment.

Hence, it has been possible to train deep learning models for some of the most difficult cognitive tasks that were either intractable earlier or had sub-optimal solutions through other machine learning techniques.

Deep learning, or neural networks in general, have another advantage over the classical machine learning models. Usually, in a classical machine learning-based approach, feature engineering plays a crucial role in the overall performance of a trained model. However, a deep learning model does away with the need to manually craft features. With large amounts of data, deep learning models can perform very well without requiring hand-engineered features and can outperform the traditional machine learning models.

The following graph indicates how deep learning models can leverage large amounts of data better than the classical machine models:

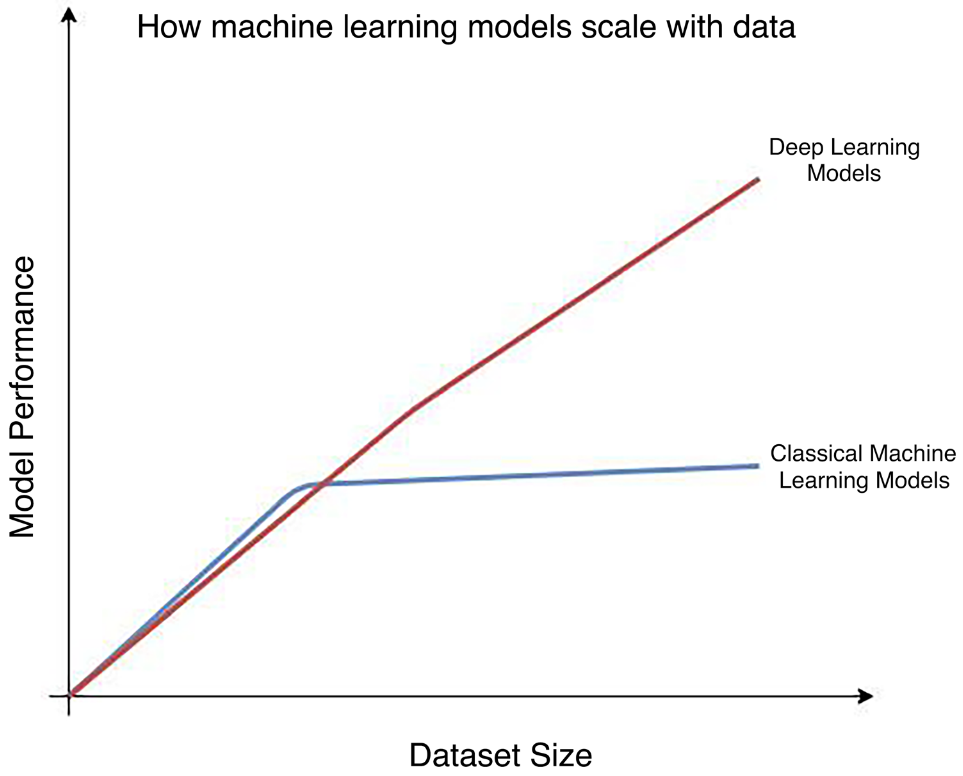


Figure 1.3: Model performance versus dataset size

As can be seen in the graph, deep learning performance isn't necessarily distinguished up to a certain dataset size. However, as the data size starts to further increase, deep neural networks begin outperforming the non-deep learning models.

A deep learning model can be built based on various types of neural network architectures that have been developed over the years. A prime distinguishing factor between the different architectures is the type and combination of layers that are used in the neural network.

Some of the well-known layers are the following:

- **Fully-connected** or **linear**: In a fully connected layer, as shown in the following diagram, all neurons preceding this layer are connected to all neurons succeeding this layer:

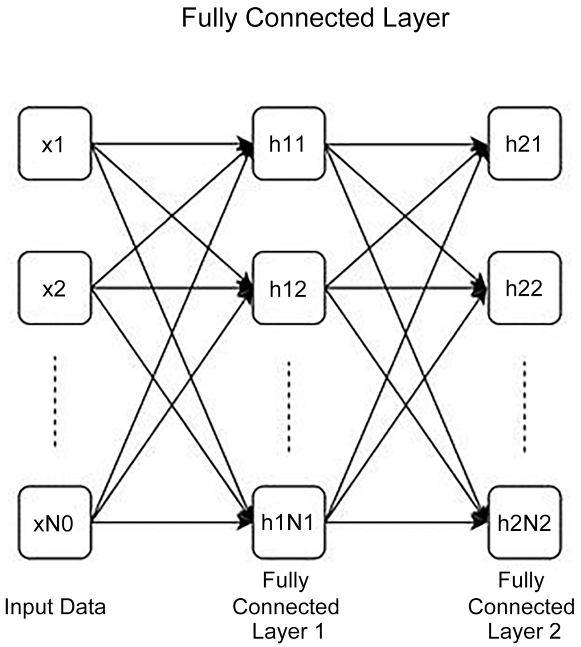
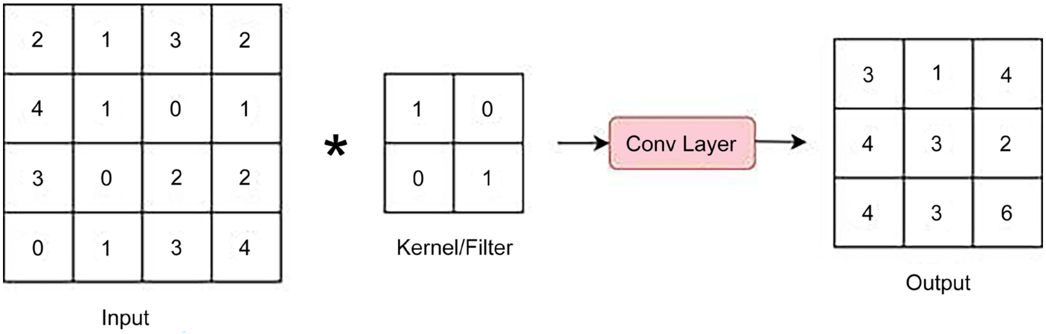


Figure 1.4: Fully connected layer

This example shows two consecutive fully connected layers with **N1** and **N2** number of neurons, respectively. Fully connected layers are a fundamental unit of many – in fact, most – deep learning classifiers.

- **Convolutional**: The following diagram shows a convolutional layer, where a convolutional kernel (or filter) is convolved over the input:



2	1	3	2
4	1	0	1
3	0	2	2
0	1	3	4

1	0
0	1

3	1	4
4	3	2
4	3	6

Figure 1.5: Convolutional layer

Convolutional layers are a fundamental unit of **Convolutional Neural Networks** (**CNNs**), which are the most effective models for solving computer vision problems.

- **Recurrent:** The following diagram shows a recurrent layer. While it looks similar to a fully connected layer, the key difference is the recurrent connection (marked with bold curved arrows):

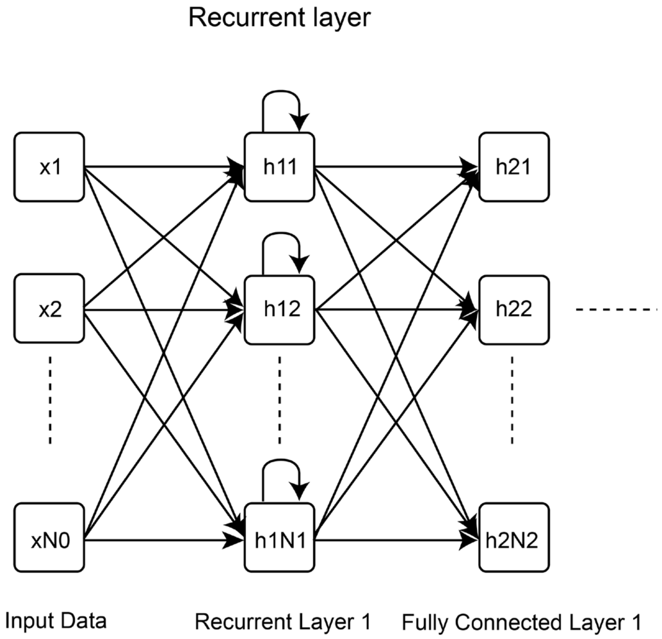


Figure 1.6: Recurrent layer

Recurrent layers have an advantage over fully connected layers in that they exhibit memorizing capabilities, which comes in handy working with sequential data where one needs to remember past inputs along with the present inputs.

- **DeConv** (**the reverse of a convolutional layer**): Quite the opposite of a convolutional layer, a **DeConvolutional Layer** works as shown in the following diagram:

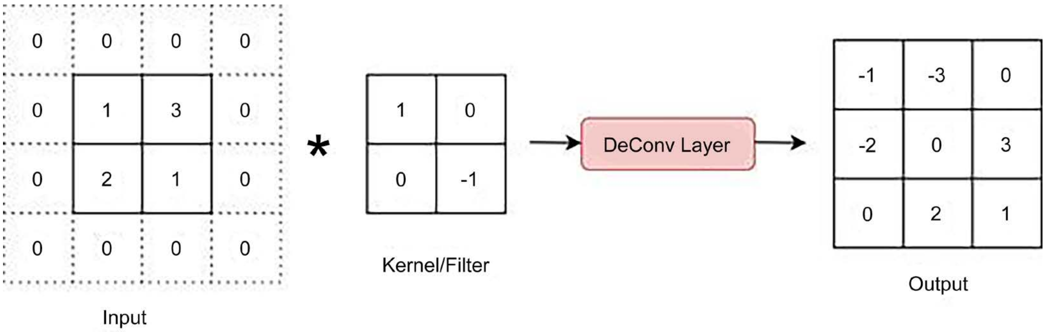


Figure 1.7: DeConvolutional Layer

This layer expands the input data spatially and hence is crucial in models that aim to generate or reconstruct images, for example.

- **Pooling:** The following diagram shows the max-pooling layer, which is perhaps the most widely used kind of pooling layer:

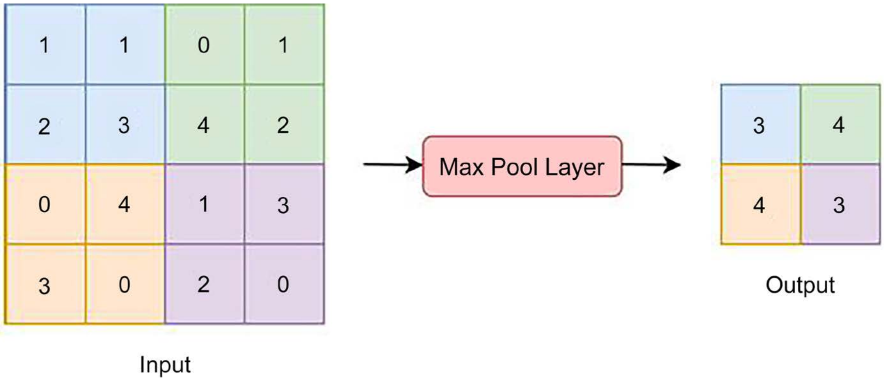
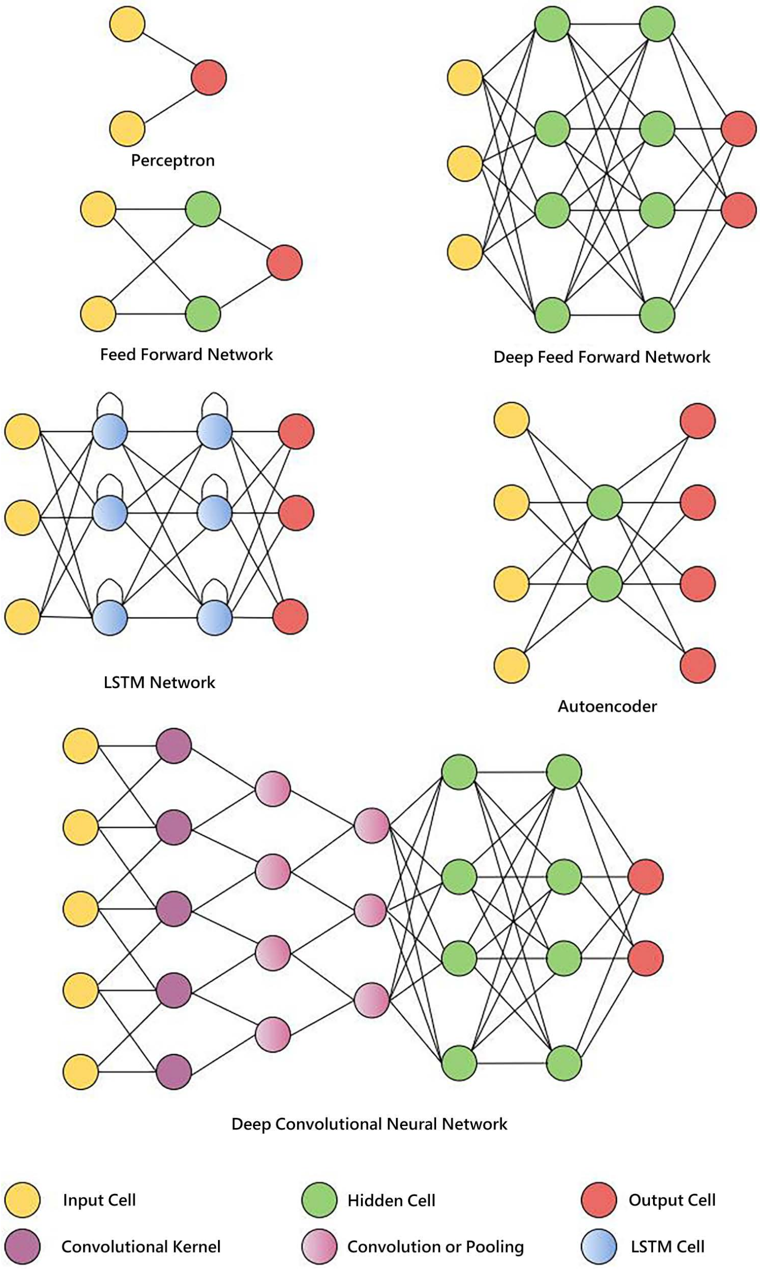


Figure 1.8: Pooling layer

This is a max-pooling layer that pools the highest number each from 2x2 sized subsections of the input. Other forms of pooling are min-pooling and average-pooling. A number of well-known architectures based on the previously mentioned layers are shown in the following diagram:



Perceptron

Feed Forward Network

Deep Feed Forward Network

LSTM Network

Autoencoder

Deep Convolutional Neural Network

Figure 1.9: Different neural network architectures

A more exhaustive set of neural network architectures can be found at [1].

Besides the types of layers and how they are connected in a network, other factors such as activation functions and the optimization schedule also define the model.

Activation functions

Activation functions are crucial to neural networks as they add the non-linearity without which, no matter how many layers we add, the entire neural network would be reduced to a simple linear model. The different types of activation functions listed here are basically different nonlinear mathematical functions.

Some of the popular activation functions are as follows:

- **Sigmoid:** A sigmoid (or logistic) function is expressed as follows:

$$y = f(x) = \frac{1}{1 + e^{-x}}$$

Equation 1.1

The function is shown in graph form as follows:

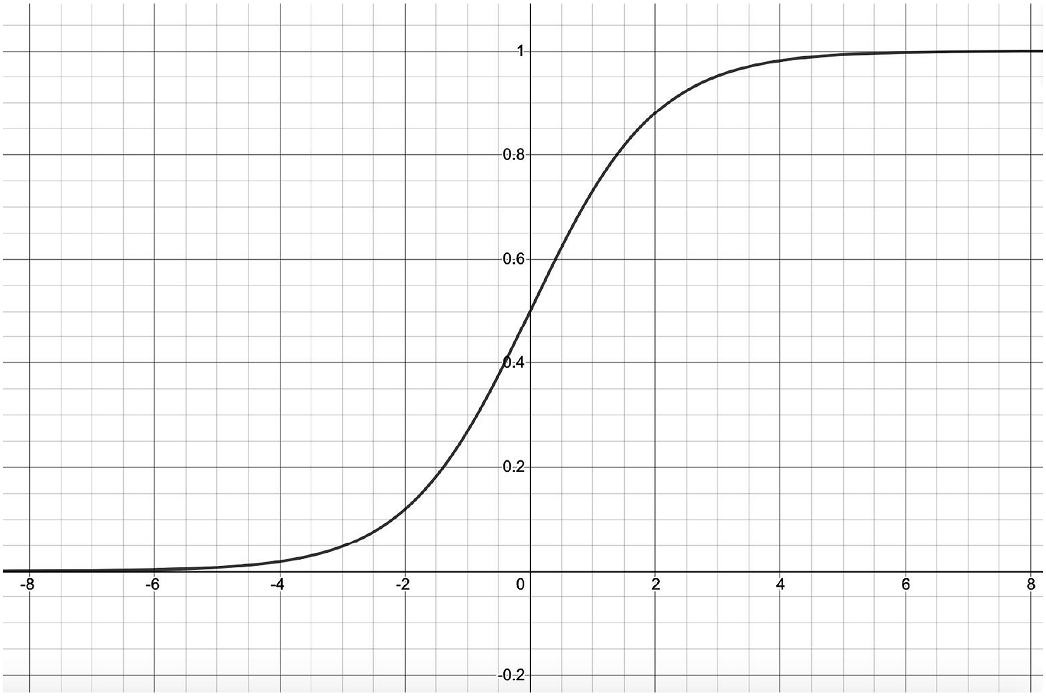


Figure 1.10: Sigmoid function

As can be seen from the graph, the sigmoid function takes in a numerical value x as input and outputs a value y in the range (0, 1).

- **TanH:** TanH is expressed as follows:

$$y = f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Equation 1.2

The function is shown in graph form as follows:

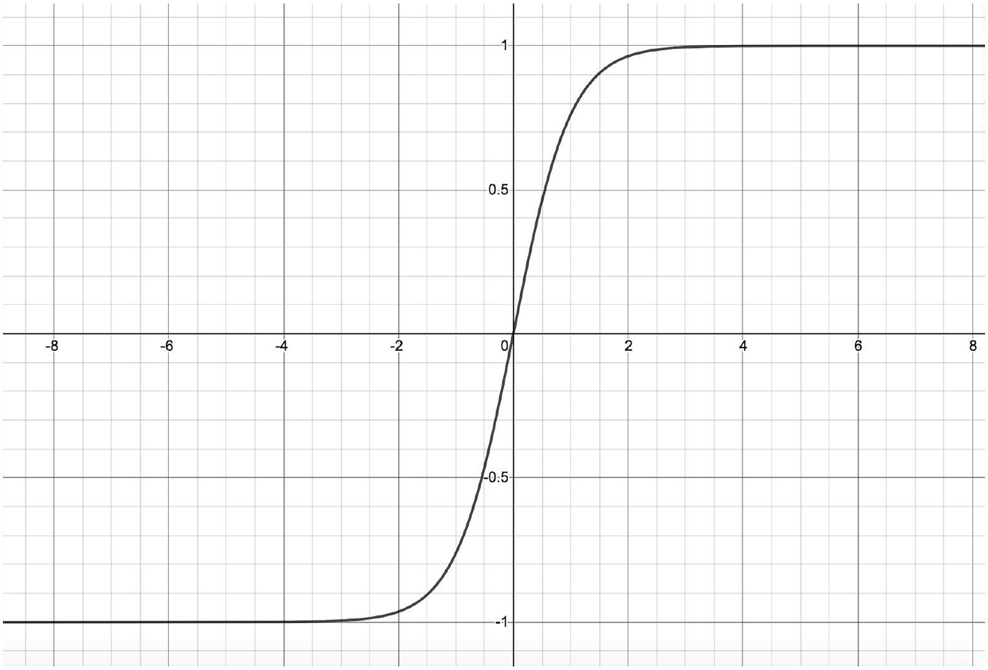


Figure 1.11: TanH function

Contrary to sigmoid, the output y varies from -1 to 1 in the case of the TanH activation function. Hence, this activation is useful in cases where we need both positive as well as negative outputs.

- **Rectified linear units** (**ReLUs**): ReLUs are more recent than the previous two and are simply expressed as follows:

$$y = f(x) = \max(0, x)$$

Equation 1.3

The function is shown in graph form as follows:

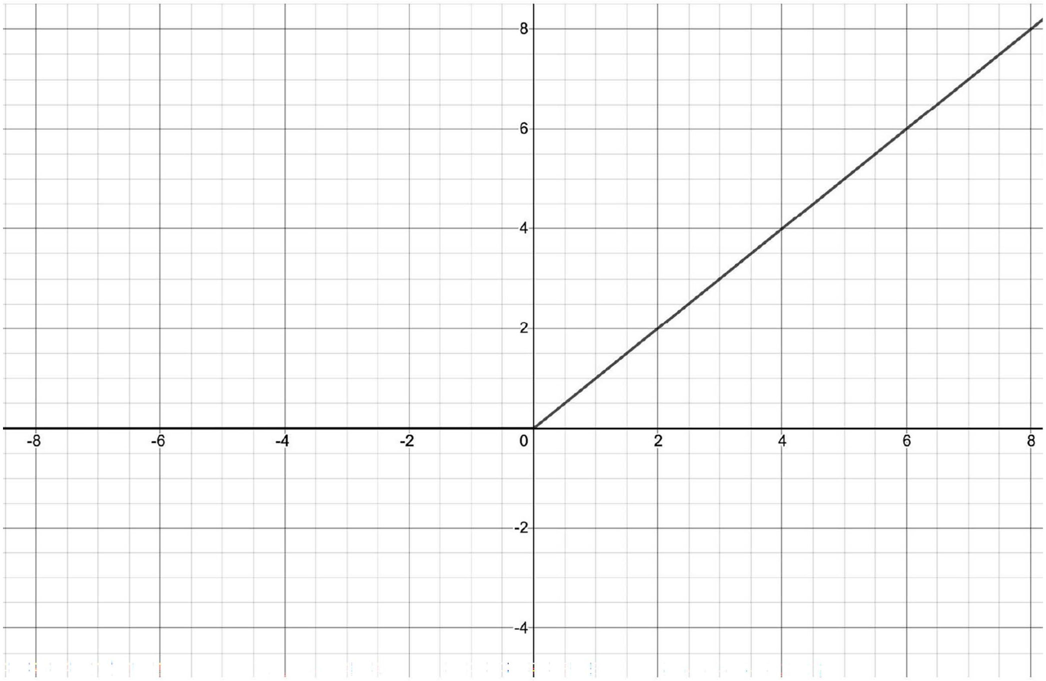


Figure 1.12: ReLU function

A distinct feature of ReLU in comparison with the sigmoid and TanH activation functions is that the output keeps growing with the input whenever the input is greater than 0. This prevents the gradient of this function from diminishing to 0 as in the case of the previous two activation functions. Although, whenever the input is negative, both the output and the gradient will be 0.

- **Leaky ReLU**: ReLUs entirely suppress any incoming negative input by outputting 0. We may, however, want to also process negative inputs for some cases. Leaky ReLUs offer the option of processing negative inputs by outputting a fraction k of the incoming negative input. This fraction k is a parameter of this activation function, which can be mathematically expressed as follows:

$$y = f(x) = \max(kx, x)$$

Equation 1.4

The following graph shows the input-output relationship for leaky ReLU:

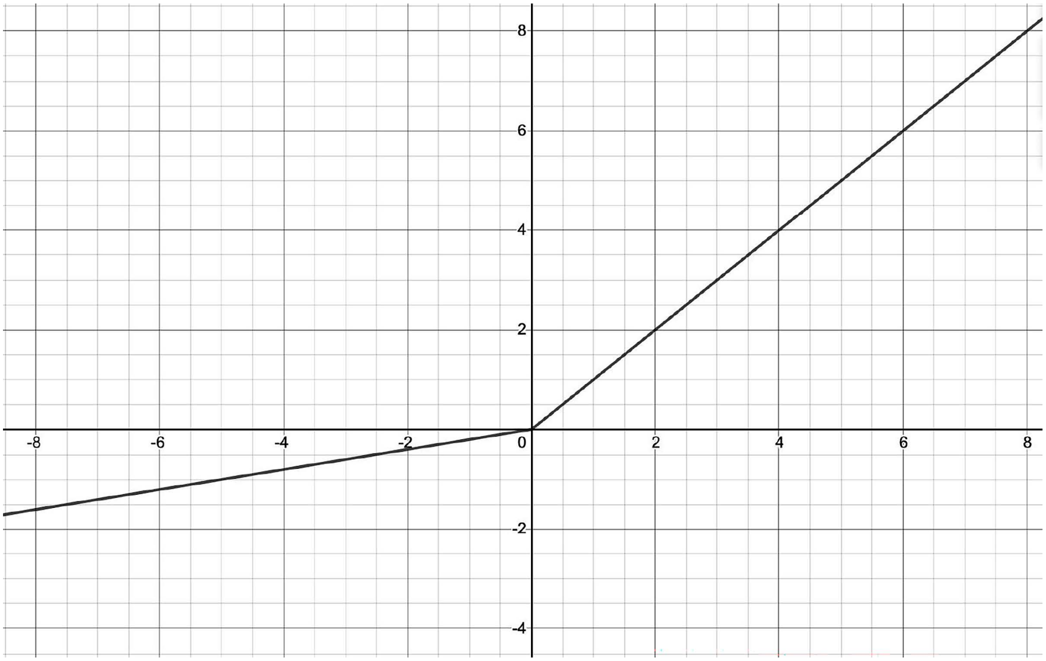


Figure 1.13: Leaky ReLU function

Activation functions are an actively evolving area of research within deep learning. It will not be possible to list all of the activation functions here but I encourage you to check out the recent developments in this domain. Many activation functions are simply nuanced modifications of the ones mentioned in this section.

Optimization schedule

So far, we have spoken of how a neural network structure is built. In order to train a neural network, we need to adopt an **optimization schedule**. Like any other parameter-based machine learning model, a deep learning model is trained by tuning its parameters. The parameters are tuned through the process of **backpropagation**, wherein the final or output layer of the neural network yields a loss. This loss is calculated with the help of a loss function that takes in the neural network's final layer's outputs and the corresponding ground truth target values. This loss is then backpropagated to the previous layers using **gradient descent** and the **chain rule of differentiation**.

The parameters or weights at each layer are accordingly modified in order to minimize the loss. The extent of modification is determined by a coefficient, which varies from 0 to 1, also known as the **learning rate**. This whole procedure of updating the weights of a neural network, which we call the **optimization schedule**, has a significant impact on how well a model is trained. Therefore, a lot of research has been done in this area and is still ongoing. The following are a few popular optimization schedules:

- **Stochastic Gradient Descent** (**SGD**): It updates the model parameters in the following fashion:

  $$\beta = \beta - \alpha * \frac{\delta L(X, y, \beta)}{\delta \beta}$$

  Equation 1.5

  β is the parameter of the model and X and y are the input training data and the corresponding labels respectively. L is the loss function and α is the learning rate. SGD performs this update for every training example pair (X, y). A variant of this –mini-batch gradient descent – performs updates for every k examples, where k is the batch size. Gradients are calculated altogether for the whole mini-batch. Another variant, batch gradient descent, performs parameter updates by calculating the gradient across the entire dataset.

- **Adagrad**: In the previous optimization schedule, we used a single learning rate for all the parameters of the model. However, different parameters might need to be updated at different paces, especially in cases of sparse data, where some parameters are more actively involved in feature extraction than others. Adagrad introduces the idea of per-parameter updates, as shown here:

  $$\beta_i^{t+1} = \beta_i^t - \frac{\alpha}{\sqrt{SSG_i^t + \epsilon}} * \frac{\delta L(X, y, \beta)}{\delta \beta_i^t}$$

  Equation 1.6

  Here, we use the subscript i to denote the i^{th} parameter and the superscript t is used to denote the time step t of the gradient descent iterations. SSG_i^t is the *sum of squared gradients* for the i^{th} parameter starting from time step 0 to time step t. ϵ is used to denote a small value added to *SSG* to avoid division by zero. Dividing the global learning rate α by the square root of *SSG* ensures smaller updates for frequently changing parameters and vice versa.

- **Adadelta**: In Adagrad, the denominator of the learning rate is a term that keeps on rising in value due to added squared terms in every time step. This causes the learning rates to decay to vanishingly small values. To tackle this problem, Adadelta introduces the idea of computing the sum of squared gradients only up to a few preceding time steps. In fact, we can express it as a running decaying average of the past gradients:

  $$SSG_i^t = \gamma * SSG_i^{t-1} + (1 - \gamma) * (\frac{\delta L(X, y, \beta)}{\delta \beta_i^t})^2$$

  Equation 1.7

γ here is the decaying factor we wish to choose for the previous sum of squared gradients. With this formulation, we ensure that the sum of squared gradients does not accumulate to a large value, thanks to the decaying average. Once SSG_i^t is defined, we can use *Equation 1.6* to define the update step for Adadelta.

However, if we look closely at *Equation 1.6*, the root mean squared gradient is not a dimensionless quantity and hence should ideally not be used as a coefficient for the learning rate. To resolve this, we define another running average, this time for the squared parameter updates. Let's first define the parameter update:

$$\Delta\beta_i^t = \beta_i^{t+1} - \beta_i^t = -\frac{\alpha}{\sqrt{SSG_i^t + \epsilon}} * \frac{\delta L(X, y, \beta)}{\delta\beta_i^t}$$

Equation 1.8

And then, similar to *Equation 1.7*, we can define the square sum of parameter updates as follows:

$$SSPU_i^t = \gamma * SSPU_i^{t-1} + (1 - \gamma) * (\Delta\beta_i^t)^2$$

Equation 1.9

Here, *SSPU* is the sum of squared parameter updates. Once we have this, we can adjust for the dimensionality problem in *Equation 1.6* with the final Adadelta equation:

$$\beta_i^{t+1} = \beta_i^t - \frac{\sqrt{SSPU_i^t + \epsilon}}{\sqrt{SSG_i^t + \epsilon}} * \frac{\delta L(X, y, \beta)}{\delta\beta_i^t}$$

Equation 1.10

Noticeably, the final Adadelta equation doesn't require any learning rate. One can still, however, provide a learning rate as a multiplier. Hence, the only mandatory hyperparameter for this optimization schedule is the decaying factors:

- **RMSprop**: We have implicitly discussed the internal workings of RMSprop while discussing Adadelta as both are pretty similar. The only difference is that RMSprop does not adjust for the dimensionality problem and hence the update equation stays the same as *Equation 1.6*, wherein the SSG_i^t is obtained from *Equation 1.7*. This essentially means that we do need to specify both a base learning rate as well as a decaying factor in the case of RMSprop.
- **Adaptive Moment Estimation** (**Adam**): This is another optimization schedule that calculates customized learning rates for each parameter. Just like Adadelta and RMSprop, Adam also uses the decaying average of the previous squared gradients as demonstrated in *Equation 1.7*. However, it also uses the decaying average of previous gradient values:

$$SG_i^t = \gamma' * SG_i^{t-1} + (1 - \gamma') * \frac{\delta L(X, y, \beta)}{\delta\beta_i^t}$$

Equation 1.11

SG and *SSG* are mathematically equivalent to estimating the first and second moments of the gradient respectively, hence the name of this method – **adaptive moment estimation**. Usually, γ and γ' are close to 1, and in that case, the initial values for both *SG* and *SSG* might be pushed towards zero. To counteract that, these two quantities are reformulated with the help of bias correction:

$$SG_i^t = \frac{SG_i^t}{1-\gamma'}$$

Equation 1.12

and

$$SSG_i^t = \frac{SSG_i^t}{1-\gamma}$$

Equation 1.13

Once they are defined, the parameter update is expressed as follows:

$$\beta_i^{t+1} = \beta_i^t - \frac{\alpha}{\sqrt{SSG_i^t + \epsilon}} * SG_i^t$$

Equation 1.14

Basically, the gradient on the extreme right-hand side of the equation is replaced by the decaying average of the gradient. Noticeably, Adam optimization involves three hyperparameters – the base learning rate, and the two decaying rates for the gradients and squared gradients. Adam is one of the most successful, if not the most successful, optimization schedule in recent times for training complex deep learning models.

So, which optimizer shall we use? It depends. If we are dealing with sparse data, then the adaptive optimizers (numbers 2 to 5) will be advantageous because of the per-parameter learning rate updates. As mentioned earlier, with sparse data, different parameters might be worked at different paces and hence a customized per-parameter learning rate mechanism can greatly help the model in reaching optimal solutions. SGD might also find a decent solution but will take much longer in terms of training time. Among the adaptive ones, Adagrad has the disadvantage of vanishing learning rates due to a monotonically increasing learning rate denominator.

RMSprop, Adadelta, and Adam are quite close in terms of their performance on various deep learning tasks. RMSprop is largely similar to Adadelta, except for the use of the base learning rate in RMSprop versus the use of the decaying average of previous parameter updates in Adadelta. Adam is slightly different in that it also includes the first-moment calculation of gradients and accounts for bias correction. Overall, Adam could be the optimizer to go with, all else being equal. We will use some of these optimization schedules in the exercises in this book. Feel free to switch them with another one to observe changes in the following:

- Model training time and trajectory (convergence)
- Final model performance

In the coming chapters, we will use many of these architectures, layers, activation functions, and optimization schedules in solving different kinds of machine learning problems with the help of PyTorch. In the example included in this chapter, we will create a convolutional neural network that contains convolutional, linear, max-pooling, and dropout layers. **Log-Softmax** is used for the final layer and ReLU is used as the activation function for all the other layers. And the model is trained using an Adadelta optimizer with a fixed learning rate of 0.5.

Exploring the PyTorch library in contrast to TensorFlow

PyTorch is a machine learning library for Python based on the Torch library. PyTorch is extensively used as a deep learning tool both for research as well as building industrial applications. It is primarily developed by Meta. PyTorch is competition for the other well-known deep learning library – TensorFlow, which is developed by Google. The initial difference between these two was that PyTorch was based on eager execution whereas TensorFlow was built on graph-based deferred execution. Although, TensorFlow now also provides an eager execution mode.

Eager execution is basically an imperative programming mode where mathematical operations are computed immediately. A deferred execution mode would have all the operations stored in a computational graph without immediate calculations and then the entire graph would be evaluated later. Eager execution is considered advantageous for reasons such as intuitive flow, easy debugging, and less scaffolding code.

PyTorch is more than just a deep learning library. With its NumPy-like syntax/interface, it provides tensor computation capabilities with strong acceleration using GPUs. But what is a tensor? Tensors are computational units, very similar to NumPy arrays, except that they can also be used on GPUs to accelerate computing.

With accelerated computing and the facility to create dynamic computational graphs, PyTorch provides a complete deep learning framework. Besides all that, it is truly Pythonic in nature, which enables PyTorch users to exploit all the features Python provides, including the extensive Python data science ecosystem.

In this section, we will expand on what a tensor is and how it is implemented with all of its attributes in PyTorch. We will also take a look at some of the useful PyTorch modules that extend various functionalities helpful in loading data, building models, and specifying the optimization schedule during the training of a model. We will compare these PyTorch APIs with the TensorFlow equivalent to understand the differences in how these two libraries are implemented at the root level.

Tensor modules

As mentioned earlier, tensors are conceptually similar to NumPy arrays. A tensor is an n-dimensional array on which we can operate mathematical functions, accelerate computations via GPUs, and can also keep track of a computational graph and gradients, which prove vital for deep learning. To run a tensor on a GPU, all we need is to cast the tensor into a certain data type.

Here is how we can instantiate a tensor in PyTorch:

```
points = torch.tensor([1.0, 4.0, 2.0, 1.0, 3.0, 5.0])
```

To fetch the first entry, simply write the following:

```
points[0]
```

We can also check the shape of the tensor using this:

```
points.shape
```

In TensorFlow, we typically declare a tensor as shown below:

```
points = tf.constant([1.0, 4.0, 2.0, 1.0, 3.0, 5.0])
```

And commands for accessing the first element or getting the tensor shape are the same as in PyTorch.

In PyTorch, tensors are implemented as views over a one-dimensional array of numerical data stored in contiguous chunks of memory. These arrays are called storage instances. Every PyTorch tensor has a storage attribute that can be called to output the underlying storage instance for a tensor, as shown in the following example:

```
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
points.storage()
```

This should output the following:

```
1.0
 4.0
 2.0
 1.0
 3.0
 5.0
[torch.storage._TypedStorage(dtype=torch.float32, device=cpu) of size 6]
```

TensorFlow tensors do not have the storage attribute. When we say a PyTorch tensor is a view on the storage instance, the tensor uses the following information to implement the view:

- Size
- Storage
- Offset
- Stride

Let's look into this with the help of our previous example:

```
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
```

Let's investigate what these different pieces of information mean:

```
points.size()
```

This should output the following:

```
torch.Size([3, 2])
```

As we can see, size is similar to the shape attribute in NumPy, which tells us the number of elements across each dimension. The multiplication of these numbers equals the length of the underlying storage instance (6 in this case). In TensorFlow, the shape of a tensor can be derived by using the `shape` attribute:

```
points = tf.constant([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
points.shape
```

This should output the following:

```
TensorShape([3, 2])
```

As we have already examined what the storage attribute means for a PyTorch tensor, let's look at offset:

```
points.storage_offset()
```

This should output the following:

```
0
```

The offset here represents the index of the first element of the tensor in the storage array. Because the output is `0`, it means that the first element of the tensor is the first element in the storage array.

Let's check this:

```
points[1].storage_offset()
```

This should output the following:

```
2
```

Because points[1] is [2.0, 1.0] and the storage array is [1.0, 4.0, 2.0, 1.0, 3.0, 5.0], we can see that the first element of the tensor [2.0, 1.0], that is, 2.0 is at index 2 of the storage array. The `storage_offset` attribute, just like the `storage` attribute, doesn't exist for a TensorFlow tensor.

Finally, we'll look at the `stride` attribute:

```
points.stride()
```

This should output the following:

```
(2, 1)
```

As we can see, `stride` contains, for each dimension, the number of elements to be skipped in order to access the next element of the tensor. So, in this case, along the first dimension, in order to access the element after the first one, that is, 1.0 we need to skip 2 elements (that is, 1.0 and 4.0) to access the next element, that is, 2.0. Similarly, along the second dimension, we need to skip 1 element to access the element after 1.0, that is, 4.0. Thus, using all these attributes, tensors can be derived from a contiguous one-dimensional storage array. TensorFlow tensors do not have the `stride` or `storage_offset` attributes.

The data contained within tensors is of numeric type. Specifically, PyTorch offers the following data types to be contained within tensors:

- `torch.float32` or `torch.float`—32-bit floating-point
- `torch.float64` or `torch.double`—64-bit, double-precision floating-point
- `torch.float16` or `torch.half`—16-bit, half-precision floating-point
- `torch.int8`—Signed 8-bit integers
- `torch.uint8`—Unsigned 8-bit integers
- `torch.int16` or `torch.short`—Signed 16-bit integers
- `torch.int32` or `torch.int`—Signed 32-bit integers
- `torch.int64` or `torch.long`—Signed 64-bit integers

TensorFlow offers similar data types [2].

An example of how we specify a certain data type to be used for a PyTorch tensor is as follows:

```
points = torch.tensor([[1.0, 2.0], [3.0, 4.0]], dtype=torch.float32)
```

In TensorFlow, this could be done with the following equivalent code:

```
points = tf.constant([[1.0, 2.0], [3.0, 4.0]], dtype=tf.float32)
```

Besides the data type, tensors in PyTorch also need a device specification where they will be stored. A device can be specified as instantiation:

```
points = torch.tensor([[1.0, 2.0], [3.0, 4.0]], dtype=torch.float32,
device='cpu')
```

Or, we can also create a copy of a tensor on the desired device:

```
points_2 = points.to(device='cuda')
```

As seen in the two examples, we can either allocate a tensor to a CPU (using `device='cpu'`), which happens by default if we do not specify a device, or we can allocate the tensor to a GPU (using `device='cuda'`). In TensorFlow, device allocation looks slightly different:

```
with tf.device('/CPU:0'):
    points = tf.constant([[1.0, 2.0], [3.0, 4.0]], dtype=tf.float32)
```

PyTorch currently supports NVIDIA (CUDA) and AMD GPUs.

When a tensor is placed on a GPU, the computations speed up and because the tensor APIs are largely uniform across CPU and GPU tensors in PyTorch, it is quite convenient to move the same tensor across devices, perform computations, and move it back.

If there are multiple devices of the same type, say more than one GPU, we can precisely locate the device we want to place the tensor in using the device index, such as the following:

```
points_3 = points.to(device='cuda:0')
```

You can read more about PyTorch-CUDA here [3]. And you can read more generally about CUDA here [4].

Let's now look at some important PyTorch modules aimed at building deep learning models.

PyTorch modules

The PyTorch library, besides offering the computational functions as NumPy does, also offers a set of modules that enable developers to quickly design, train, and test deep learning models. The following are some of the most useful modules.

torch.nn

When building a neural network architecture, the fundamental aspects that the network is built on are the number of layers, the number of neurons in each layer, and which of those are learnable, and so on. The PyTorch `nn` module enables users to quickly instantiate neural network architectures by defining some of these high-level aspects as opposed to having to specify all the details manually. The following is a one-layer neural network initialization without using the `nn` module:

```
import math
'''we assume a 256-dimensional input and a 4-dimensional
output for this 1-layer neural network
hence, we initialize a 256x4 dimensional matrix
filled with random values'''
weights = torch.randn(256, 4) / math.sqrt(256)
'''we then ensure that the parameters of this neural network are
trainable, that is, the numbers in the 256x4 matrix
can be tuned with the help of backpropagation of gradients'''
weights.requires_grad_()
'''finally we also add the bias weights for the
4-dimensional output, and make these trainable too'''
bias = torch.zeros(4, requires_grad=True)
```

We can instead use `nn.Linear(256, 4)` to represent the same thing in PyTorch. In TensorFlow, this could be written as `tf.keras.layers.Dense(256, input_shape=(4,), activation=None)`.

Within the `torch.nn` module, there is a submodule called `torch.nn.functional`. This submodule consists of all the functions within the `torch.nn` module, whereas all the other submodules are classes. These functions are **loss functions**, **activating functions**, and also **neural functions** that can be used to create neural networks in a functional manner (that is, when each subsequent layer is expressed as a function of the previous layer) such as pooling, convolutional, and linear functions.

An example of a loss function using the `torch.nn.functional` module could be the following:

```
import torch.nn.functional as F
loss_func = F.cross_entropy
loss = loss_func(model(X), y)
```

Here, `X` is the input, `y` is the target output, and model is the neural network model. In TensorFlow, the above code would be written as:

```
import tensorflow as tf
loss_func = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
loss = loss_func(y, model(X))
```

torch.optim

As we train a neural network, we back-propagate errors to tune the weights or parameters of the network – the process that we call optimization. The `optim` module includes all the tools and functionalities related to running various types of optimization schedules while training a deep learning model.

Let's say we define an optimizer during a training session using the `torch.optim` modules, as shown in the following snippet:

```
opt = optim.SGD(model.parameters(), lr=lr)
```

Then, we don't need to manually write the optimization step as shown here:

```
with torch.no_grad():
    # applying the parameter updates using stochastic gradient descent
    for param in model.parameters():
        param -= param.grad * lr
    model.zero_grad()
```

We can simply write this instead:

```
opt.step()
opt.zero_grad()
```

TensorFlow doesn't require such explicitly coded gradient update and flush steps and the code for the optimizer looks like the following:

```
opt = tf.keras.optimizers.SGD(learning_rate=lr)
model.compile(optimizer=opt, loss=...)
```

Next, we will look at the `utils.data` module.

torch.utils.data

Under the `utils.data` module, Torch provides its own `dataset` and `DataLoader` classes, which are extremely handy due to their abstract and flexible implementations. Basically, these classes provide intuitive and useful ways of iterating and performing other such operations on tensors.

Using these, we can ensure high performance due to optimized tensor computations and also have fail-safe data I/O. For example, let's say we use `torch.utils.data.DataLoader` as follows:

```
from torch.utils.data import (TensorDataset, DataLoader)
train_dataset = TensorDataset(x_train, y_train)
train_dataloader = DataLoader(train_dataset, batch_size=bs)
```

Then, we don't need to iterate through batches of data manually, like this:

```
for i in range((n-1)//bs + 1):
    x_batch = x_train[start_i:end_i]
    y_batch = y_train[start_i:end_i]
    pred = model(x_batch)
```

We can simply write this instead:

```
for x_batch,y_batch in train_dataloader:
    pred = model(x_batch)
```

The `torch.utils.data` is similar to the `tf.data.Dataset` in TensorFlow. The preceding code for iterating through batches of data would be written in the following way in TensorFlow:

```
import tensorflow as tf
train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
train_dataloader = train_dataset.batch(bs)

for x_batch, y_batch in train_dataloader:
    pred = model(x_batch)
```

Now that we have explored the PyTorch library (in contrast to TensorFlow) and understood the PyTorch and Tensor modules, let's learn how to train a neural network using PyTorch.

Training a neural network using PyTorch

For this exercise, we will be using the famous `MNIST` dataset [5], which is a sequence of images of handwritten postcode digits, zero through nine, with corresponding labels. The `MNIST` dataset consists of 60,000 training samples and 10,000 test samples, where each sample is a grayscale image with 28 x 28 pixels. PyTorch also provides the `MNIST` dataset under its `Dataset` module.

In this exercise, we will use PyTorch to train a deep learning multi-class classifier on this dataset and test how the trained model performs on the test samples. The full PyTorch code [6] for this exercise as well as the equivalent TensorFlow code [7] can be found in this book's GitHub repository.

1. For this exercise, we will need to import a few dependencies. Execute the following `import` statements:

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

```
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
```

2. Next, we define the model architecture as shown in the following diagram:

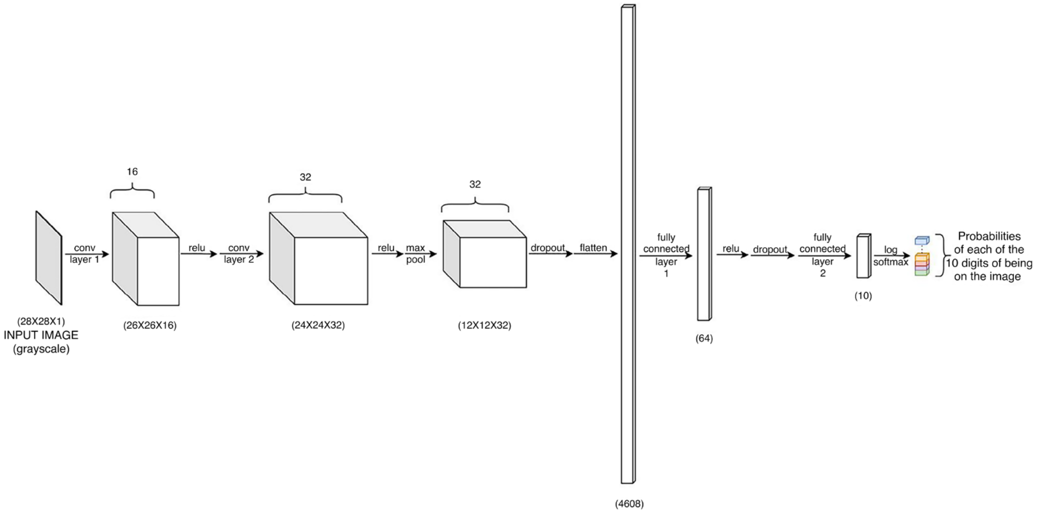


Figure 1.14: Neural network architecture

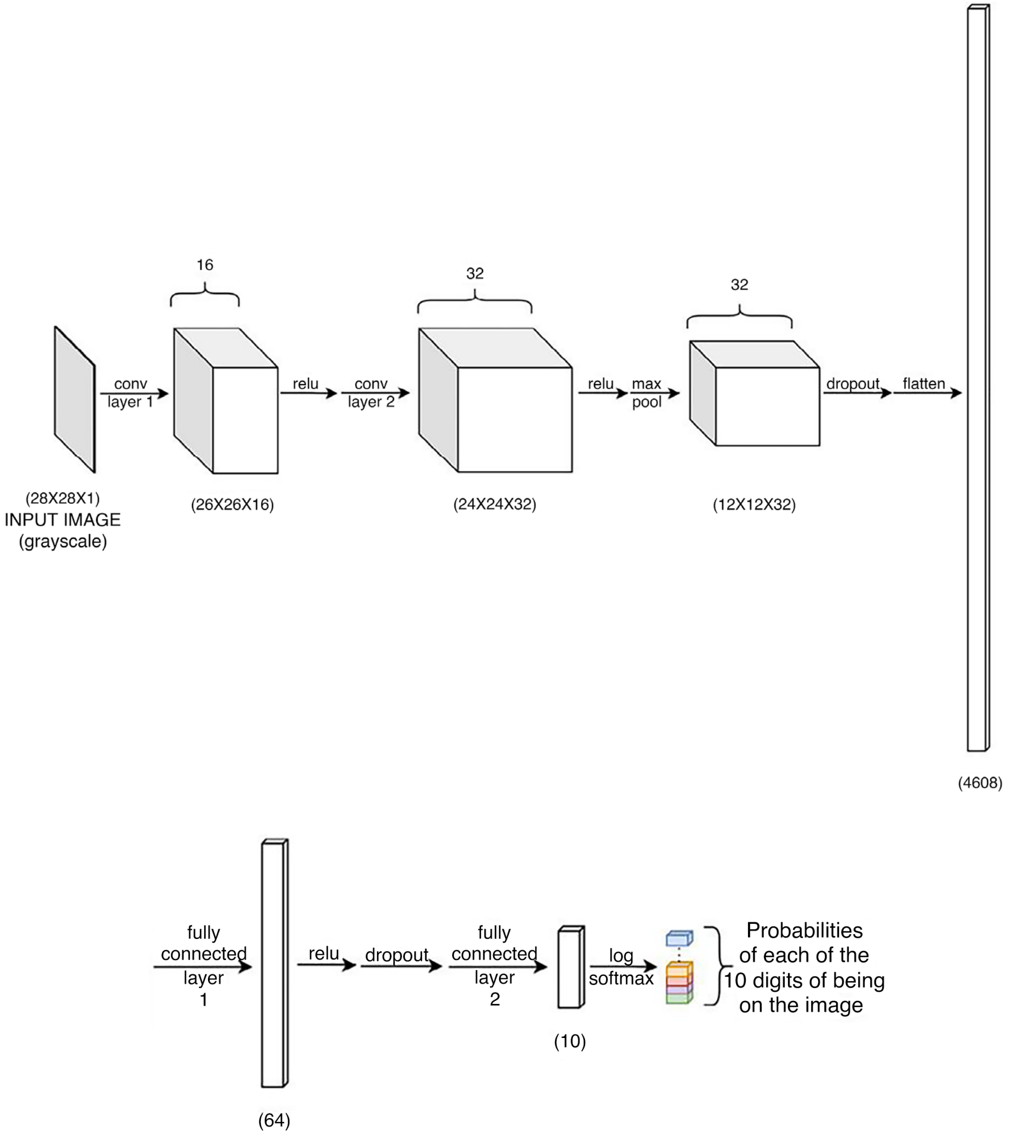


Figure 1.14: Neural network architecture

The model consists of convolutional layers, dropout layers, as well as linear/fully connected layers, all available through the `torch.nn` module:

```
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.cn1 = nn.Conv2d(1, 16, 3, 1)
        self.cn2 = nn.Conv2d(16, 32, 3, 1)
        self.dp1 = nn.Dropout2d(0.10)
        self.dp2 = nn.Dropout2d(0.25)
        self.fc1 = nn.Linear(4608, 64)
        # 4608 is basically 12 X 12 X 32
        self.fc2 = nn.Linear(64, 10)
    def forward(self, x):
        x = self.cn1(x)
        x = F.relu(x)
        ...
        x = self.fc2(x)
        op = F.log_softmax(x, dim=1)
        return op
```

The `__init__` function defines the core architecture of the model, that is, all the layers with the number of neurons at each layer. And the `forward` function, as the name suggests, does a forward pass in the network. Hence it includes all the activation functions at each layer as well as any pooling or dropout used after any layer. This function shall return the final layer output, which we call the prediction of the model, which has the same dimensions as the target output (the ground truth).

Notice that the first convolutional layer has a 1-channel input, a 16-channel output, a kernel size of 3, and a stride of 1. The 1-channel input is essentially for the grayscale images that will be fed to the model. We decided on a kernel size of 3x3 for various reasons. Firstly, kernel sizes are usually odd numbers so that the input image pixels are symmetrically distributed around a central pixel. 1x1 would be too small because then the kernel operating on a given pixel would not have any information about the neighboring pixels. 3 comes next, but why not go further to 5, 7, or, say, even 27?

Well, at the extreme high end, a 27x27 kernel convolving over a 28x28 image would give us very coarse-grained features. However, the most important visual features in the image are fairly local (in a small spatial neighborhood) and hence it makes sense to use a small kernel that looks at a few neighboring pixels at a time, for visual patterns. 3x3 is one of the most common kernel sizes used in CNNs for solving computer vision problems.

Note that we have two consecutive convolutional layers, both with 3x3 kernels. This, in terms of spatial coverage, is equivalent to using one convolutional layer with a 5x5 kernel. However, using multiple layers with a smaller kernel size is almost always preferred because it results in deeper networks, hence more complex learned features as well as fewer parameters due to smaller kernels. Using many small kernels across layers may also result in specialized kernels – one for detecting edges, one for circles, one for the color red, and so on.

The number of channels in the output of a convolutional layer is usually higher than or equal to the input number of channels. Our first convolutional layer takes in one channel's data and outputs 16 channels. This basically means that the layer is trying to detect 16 different kinds of information from the input image. Each of these channels is called a **feature map** and each of them has a dedicated kernel extracting features for them.

We escalate the number of channels from 16 to 32 in the second convolutional layer, in an attempt to extract more kinds of features from the image. This increment in the number of channels (or image depth) is common practice in CNNs. We will read more on this under *Width-based CNNs* in *Chapter 2*, *Deep CNN Architectures*.

Finally, the stride of `1` makes sense, as our kernel size is just `3`. Keeping a larger stride value – say, `10` – would result in the kernel skipping many pixels in the image and we don't want to do that. If, however, our kernel size was 100, we might have considered 10 as a reasonable stride value. The larger the stride, the lower the number of convolution operations but the smaller the overall field of view for the kernel.

The preceding code could also be written using the `torch.nn.Sequential` API:

```
model = nn.Sequential(
    nn.Conv2d(1, 16, 3, 1),
    nn.ReLU(),
    nn.Conv2d(16, 32, 3, 1),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Dropout2d(0.10),
    nn.Flatten(),
    nn.Linear(4608, 64),
    nn.ReLU(),
    nn.Dropout2d(0.25),
    nn.Linear(64, 10),
    nn.LogSoftmax(dim=1)
)
```

It is usually preferred to initialize the model with separate `__init__` and `forward` methods in order to have more flexibility in defining model functionality when not all layers are executed one after another (parallel or skip connections, for example). The sequential code written above looks very similar in TensorFlow:

```
import tensorflow as tf
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(16, 3, activation='relu',
                           input_shape=(28, 28, 1)),
    tf.keras.layers.Conv2D(32, 3, activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Dropout(0.10),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dropout(0.25),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

And the code with `__init__` and `forward` methods looks like the following in TensorFlow:

```
import tensorflow as tf

class ConvNet(tf.keras.Model):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.cn1 = tf.keras.layers.Conv2D(16, 3,
                                          activation='relu',
                                          input_shape=(28, 28, 1))
        self.fc2 = tf.keras.layers.Dense(10, activation='softmax')

    def call(self, x):
        x = self.cn1(x)
        x = self.fc2(x)
        return x
```

Instead of `forward`, we use the `call` method in TensorFlow, and the rest looks similar to PyTorch code.

3. We then define the training routine, that is, the actual backpropagation step. As can be seen, the `torch.optim` module greatly helps in keeping this code succinct:

```
def train(model, device, train_dataloader, optim, epoch):
    model.train()
    for b_i, (X, y) in enumerate(train_dataloader):
        X, y = X.to(device), y.to(device)
        optim.zero_grad()
        pred_prob = model(X)
        loss = F.nll_loss(pred_prob, y)
        # nll is the negative likelihood loss
        loss.backward()
        optim.step()
        if b_i % 10 == 0:
            print('epoch: {} [{}/{} ({:.0f}%)]\t \
                  training loss:\ {:.6f}'.format(
                epoch, b_i * len(X),
                len(train_ dataloader.dataset),
                100. * b_i / len(train_dataloader),
                loss. item()))
```

This iterates through the dataset in batches, makes a copy of the dataset on the given device, makes a forward pass with the retrieved data on the neural network model, computes the loss between the model prediction and the ground truth, uses the given optimizer to tune model weights, and prints training logs every 10 batches. The entire procedure done once qualifies as 1 epoch, that is, when the entire dataset has been read once. For TensorFlow, we will run the training directly at a high level, in *step 7*. The detailed training routine definition in PyTorch gives us the flexibility to closely control the training process as opposed to training with a single line of code at a high level.

4. Similar to the preceding training routine, we write a test routine that can be used to evaluate the model performance on the test set:

```
def test(model, device, test_dataloader):
    model.eval()
    loss = 0
    success = 0
    with torch.no_grad():
        for X, y in test_dataloader:
            X, y = X.to(device), y.to(device)
            pred_prob = model(X)
            # loss summed across the batch
            loss += F.nll_loss(pred_prob, y,
                               reduction='sum').item()
            # use argmax to get the most likely prediction
            pred = pred_prob.argmax(dim=1, keepdim=True)
            success += pred.eq(y.view_as(pred)).sum().item()
    loss /= len(test_dataloader.dataset)
    print('\nTest dataset: Overall Loss: {:.4f}, \
          Overall Accuracy: {}/{} ({:.0f}%)\n'.format(loss,
        success, len(test_dataloader.dataset),
        100. * success / len(test_dataloader.dataset)))
```

Most of this function is similar to the preceding `train` function. The only difference is that the loss computed from the model predictions and the ground truth is not used to tune the model weights using an optimizer. Instead, the loss is used to compute the overall test error across the entire test batch.

5. Next, we come to another critical component of this exercise, which is loading the dataset. Thanks to PyTorch's `DataLoader` module, we can set up the dataset loading mechanism in a few lines of code:

```
'''The mean and standard deviation values are calculated as
the mean of all pixel values of all images in
the training dataset'''
train_dataloader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=True, download=True,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1302,),
                                            (0.3069,))])),
    # train_X.mean()/256. and train_X.std()/256.
    batch_size=32, shuffle=True)
test_dataloader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=False,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1302,),
                                            (0.3069,))
                   ])),
    batch_size=500, shuffle=False)
```

As you can see, we set `batch_size` to `32`, which is a fairly common choice. Usually, there is a trade-off in deciding the batch size. A very small batch size can lead to slow training due to frequent gradient calculations and can lead to extremely noisy gradients. Very large batch sizes can, on the other hand, also slow down training due to a long waiting time to calculate gradients. It is mostly not worth waiting long before a single gradient update. It is rather advisable to make frequent, less precise gradients as it will eventually lead the model to a better set of learned parameters.

For both the training and test dataset, we specify the local storage location we want to save the dataset to, and the batch size, which determines the number of data instances that constitute one pass of a training and test run. We also specify that we want to randomly shuffle training data instances to ensure a uniform distribution of data samples across batches.

Finally, we also normalize the dataset to a normal distribution with a specified mean and standard deviation. This mean and standard deviation comes from the training dataset if we are training a model from scratch. However, if we are transfer-learning from a pre-trained model, then the mean and standard deviation values are obtained from the original training dataset of the pre-trained model. We will learn more on transfer learning in *Chapter 2, Deep CNN Architectures*.

In TensorFlow, we would use `tf.keras.datasets` to load MNIST data and the `tf.data.Dataset` module to create batches of training data out of the dataset, as shown in the following code:

```
# Load the MNIST dataset.
(x_train, y_train), (x_test, y_test) =
      tf.keras.datasets.mnist.load_data()

# Normalize pixel values between 0 and 1
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# Add a channels dimension (required for CNN)
x_train = x_train[..., tf.newaxis]
x_test = x_test[..., tf.newaxis]

# Create a dataloader for training.
train_dataloader = tf.data.Dataset.from_tensor_slices(
    (x_train, y_train))
train_dataloader = train_dataloader.shuffle(10000)
train_dataloader = train_dataloader.batch(32)

# Create a dataloader for testing.
test_dataloader = tf.data.Dataset.from_tensor_slices((x_test, y_test))
test_dataloader = test_dataloader.batch(500)
```

6. We defined the training routine earlier. Now is the time to define the optimizer and device we will use to run the model training:

```
torch.manual_seed(0)
device = torch.device("cpu")
model = ConvNet()
optimizer = optim.Adadelta(model.parameters(), lr=0.5)
```

We define the device for this exercise as `cpu`. We also set a seed to avoid unknown randomness and ensure reproducibility. We will use Adadelta as the optimizer for this exercise with a learning rate of `0.5`. While discussing optimization schedules earlier in the chapter, we mentioned that Adadelta could be a good choice if we are dealing with sparse data.

And this is a case of sparse data, because not all pixels in the image are informative. Having said that, I encourage you to try out other optimizers such as Adam on this same problem to see how it affects the training process and model performance. The following is the TensorFlow equivalent code one would use to instantiate and compile the model:

```
tf.random.set_seed(0)
model = ConvNet()
optimizer = \
    tf.keras.optimizers.experimental.Adadelta(learning_rate=0.5)
model.compile(optimizer=optimizer,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

7. And then we start the actual process of training the model for *k* number of epochs, and we also keep testing the model at the end of each training epoch:

```
for epoch in range(1, 3):
    train(model, device, train_dataloader, optimizer, epoch)
    test(model, device, test_dataloader)
```

For demonstration purposes, we will run the training for only two epochs. The output will be as follows:

```
epoch: 1 [0/60000 (0%)]       training loss: 2.31060
epoch: 1 [320/60000 (1%)]     training loss: 1.924133
epoch: 1 [640/60000 (1%)]     training loss: 1.313336
epoch: 1 [960/60000 (2%)]     training loss: 0.796470
epoch: 1 [1280/60000 (2%)]    training loss: 0.819801
...
epoch: 2 [58560/60000 (98%)]  training loss: 0.007698
epoch: 2 [58880/60000 (98%)]  training loss: 0.002685
epoch: 2 [59200/60000 (99%)]  training loss: 0.016287
epoch: 2 [59520/60000 (99%)]  training loss: 0.012645
epoch: 2 [59840/60000 (100%)] training loss: 0.007993

Test dataset: Overall Loss: 0.0416, Overall Accuracy: 9864/10000 (99%)
```

The training loop code equivalent for TensorFlow would be as follows:

```
model.fit(train_dataloader, epochs=2,
          validation_data=test_dataloader)
```

8. Now that we have trained a model, with a reasonable test set performance, we can also manually check whether the model inference on a sample image is correct:

```
test_samples = enumerate(test_dataloader)
b_i, (sample_data, sample_targets) = next(test_samples)
plt.imshow(sample_data[0][0],
           cmap='gray', interpolation='none')
```

The output will be as follows:

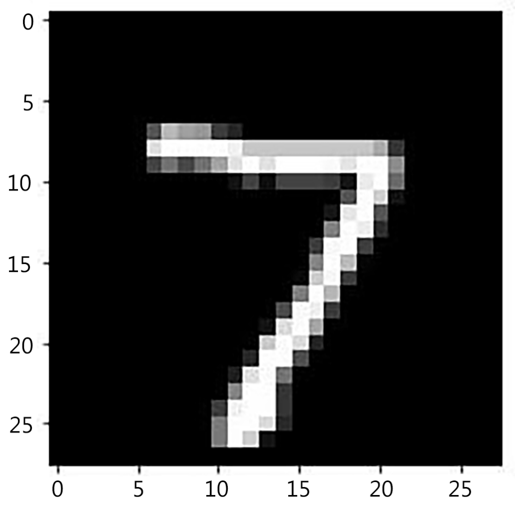


Figure 1.15: Sample handwritten image

The equivalent Tensorflow code would be the same except for using `sample_data[0]` instead of `sample_data[0][0]`:

```
test_samples = enumerate(test_dataloader)
b_i, (sample_data, sample_targets) = next(test_samples)
plt.imshow(sample_data[0],
           cmap='gray', interpolation='none')
plt.show()
```

And now we run the model inference for this image and compare it with the ground truth:

```
print(f"Model prediction is : \
      {model(sample_data).data.max(1)[1][0]}")
print(f"Ground truth is : {sample_targets[0]}")
```

Note that, for predictions, we first calculate the class with maximum probability using the `max()` function on `axis=1`. The `max()` function outputs two lists – a list of probabilities of classes for every sample in `sample_data` and a list of class labels for each sample. Hence, we choose the second list using index `[1]`.

We further select the first class label by using index `[0]` to look at only the first sample under `sample_data`. The output will be as follows:

```
Model prediction is : 7
Ground truth is : 7
```

This appears to be the correct prediction. The forward pass of the neural network done using `model()` produces probabilities. Hence, we use the `max()` function to output the class with the maximum probability. The same output can be achieved in TensorFlow with the following code:

```
print(f"Model prediction is :\
      {tf.math.argmax(model(sample_data)[0])}")
print(f"Ground truth is : {sample_targets[0]}")
```

The code pattern for this exercise is derived from the official PyTorch examples repository [8].

This concludes our exploration of the PyTorch library in the form of an end-to-end exercise while comparing the APIs of PyTorch and TensorFlow at different stages of model training – model initialization, data loading, training loop, and model evaluation. This analysis should help you in getting started with using PyTorch as well as transitioning from TensorFlow to PyTorch if you are already familiar with the former.

Summary

In this chapter, we refreshed deep learning concepts, explored the PyTorch deep learning library in contrast to TensorFlow and ran a hands-on exercise on training a deep learning model (CNN) from scratch.

In the next chapter, we will take a deeper look at the gamut of different CNN architectures developed over the years, how each of them is uniquely useful, and how they can be easily implemented using PyTorch.

Reference list

1. The Asimov Institute, *The Neural Network Zoo*: `https://www.asimovinstitute.org/neural-network-zoo/`
2. TensorFlow, *Module: tf.dtypes*: `https://www.tensorflow.org/api_docs/python/tf/dtypes`
3. PyTorch, *CUDA Semantics*: `https://pytorch.org/docs/stable/notes/cuda.html`
4. Nvidia Developer, *About CUDA*: `https://developer.nvidia.com/about-cuda`
5. The MNIST Database: `http://yann.lecun.com/exdb/mnist/`
6. GitHub 1: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter01/mnist_pytorch.ipynb`

7. GitHub 2: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter01/mnist_tensorflow.ipynb`
8. GitHub 3: `https://github.com/pytorch/examples/tree/master/mnist`

Learn more on Discord

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

`https://packt.link/mastorch`

2

Deep CNN Architectures

In this chapter, we will first briefly review the evolution of **Convolutional Neural Network** (**CNN**) architectures, and then we will study the different CNN architectures in detail. We will implement these CNN architectures using PyTorch, and in doing so, we aim to exhaustively explore the tools (modules and built-in functions) that PyTorch has to offer in the context of building **Deep CNNs.** Gaining strong CNN expertise in PyTorch will enable us to solve a number of deep learning problems involving CNNs. This will also help us in building more complex deep learning models or applications of which CNNs are a part.

This chapter will cover the following topics:

- Why are CNNs so powerful?
- Evolution of CNN architectures
- Developing LeNet from scratch
- Fine-tuning the AlexNet model
- Running a pretrained VGG model
- Exploring GoogLeNet and Inception v3
- Discussing ResNet and DenseNet architectures
- Understanding EfficientNets and the future of CNN architectures

> All the code files for this chapter can be found at `https://github.com/arj7192/MasteringPyTorchV2/tree/main/Chapter03`.

Let us start by discussing the key features of CNNs.

Why are CNNs so powerful?

CNNs are among the most powerful machine learning models at solving challenging problems such as image classification, object detection, object segmentation, video processing, natural language processing, and speech recognition. Their success is attributed to various factors, such as the following:

- **Weight sharing:** This makes CNNs parameter-efficient; that is, different features are extracted using the same set of weights or parameters. **Features** are the high-level representations of input data that the model generates with its parameters.
- **Automatic feature extraction**: Multiple feature extraction stages help a CNN to automatically learn feature representations in a dataset.
- **Hierarchical learning**: The multi-layered CNN structure helps CNNs to learn low-, mid-, and high-level features.
- The ability to explore both **spatial and temporal** correlations in the data, such as in video-processing tasks.

Besides these pre-existing fundamental characteristics, CNNs have advanced over the years with the help of improvements in the following areas:

- The use of better **activation** and **loss functions**, such as using **ReLU** to overcome the **vanishing gradient problem**.
- Parameter optimization, such as using an optimizer based on **Adaptive Momentum** (**Adam**) instead of simple stochastic gradient descent.
- **Regularization**: Applying dropouts and batch normalization besides L2 regularization.

FAQ – What is the vanishing gradient problem?

Backpropagation in neural networks works on the basis of the chain rule of differentiation. According to the chain rule, the gradient of the loss function with respect to the input layer parameters can be written as a product of gradients at each layer. If these gradients are all less than 1 – and worse still, tending toward 0 – then the product of these gradients will be a vanishingly small value. The vanishing gradient problem can cause serious trouble in the optimization process by preventing the network parameters from changing their values, which is equivalent to stunted learning.

But some of the most significant drivers of development in CNNs over the years have been the various *architectural innovations*:

- **Spatial exploration-based CNNs:** The idea behind **spatial exploration** is using different kernel sizes in order to explore different levels of visual features in input data. The following diagram shows a sample architecture for a spatial exploration-based CNN model:

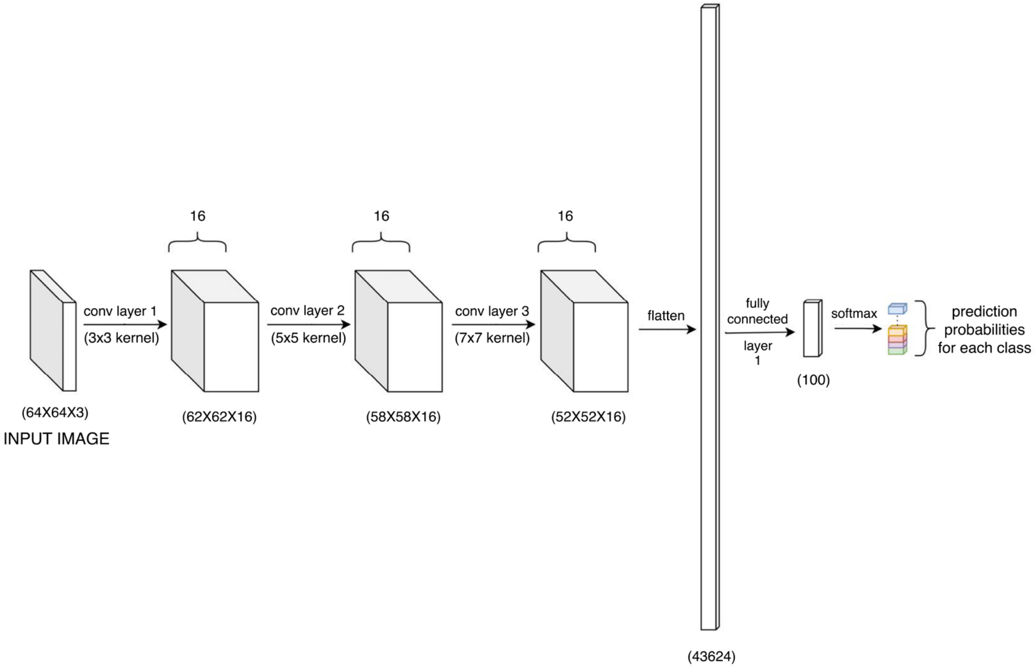


Figure 2.1: Spatial exploration-based CNN

- **Depth-based CNNs:** The **depth** here refers to the depth of the neural network, that is, the number of layers. So, the idea here is to create a CNN model with multiple convolutional layers in order to extract highly complex visual features. The following diagram shows an example of such a model architecture:

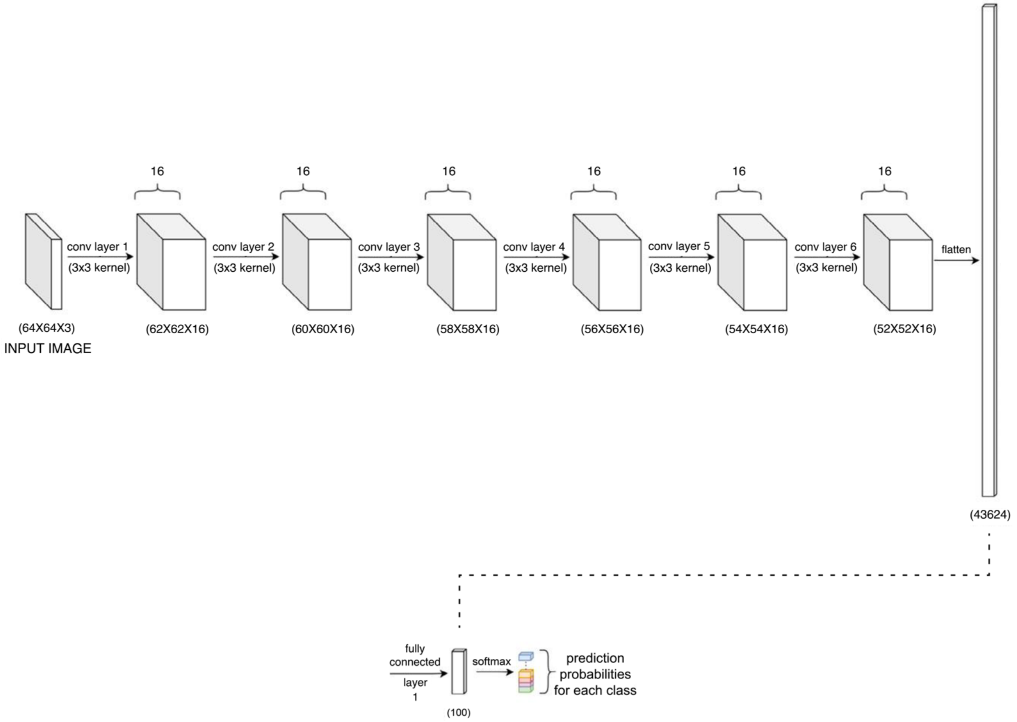


Figure 2.2: Depth-based CNN

- **Width-based CNNs:** Width refers to the number of channels or feature maps in the data or features extracted from the data. So, width-based CNNs are all about increasing the number of feature maps as we go from the input to the output layers, as demonstrated in the following diagram:

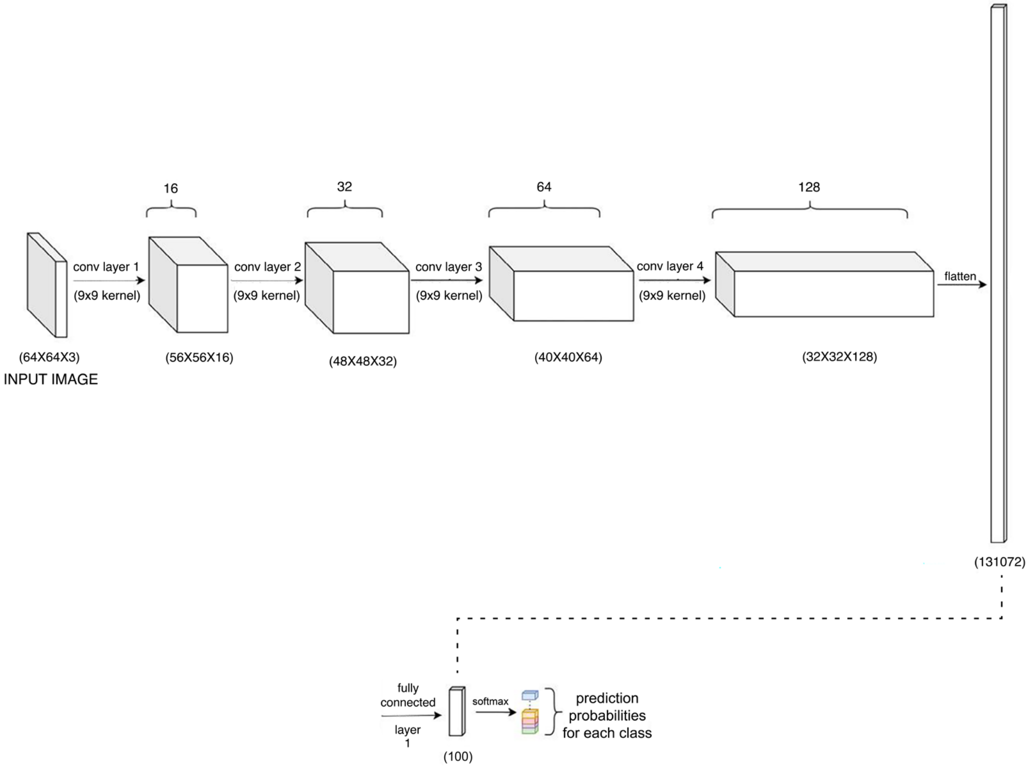


Figure 2.3: Width-based CNN

- **Multi-path-based CNNs:** So far, the preceding three types of architectures have had monotonicity in connections between layers; that is, direct connections exist only between consecutive layers. **Multi-path CNNs** brought the idea of making shortcut connections or skip connections between non-consecutive layers. The following diagram shows an example of a multi-path CNN model architecture:

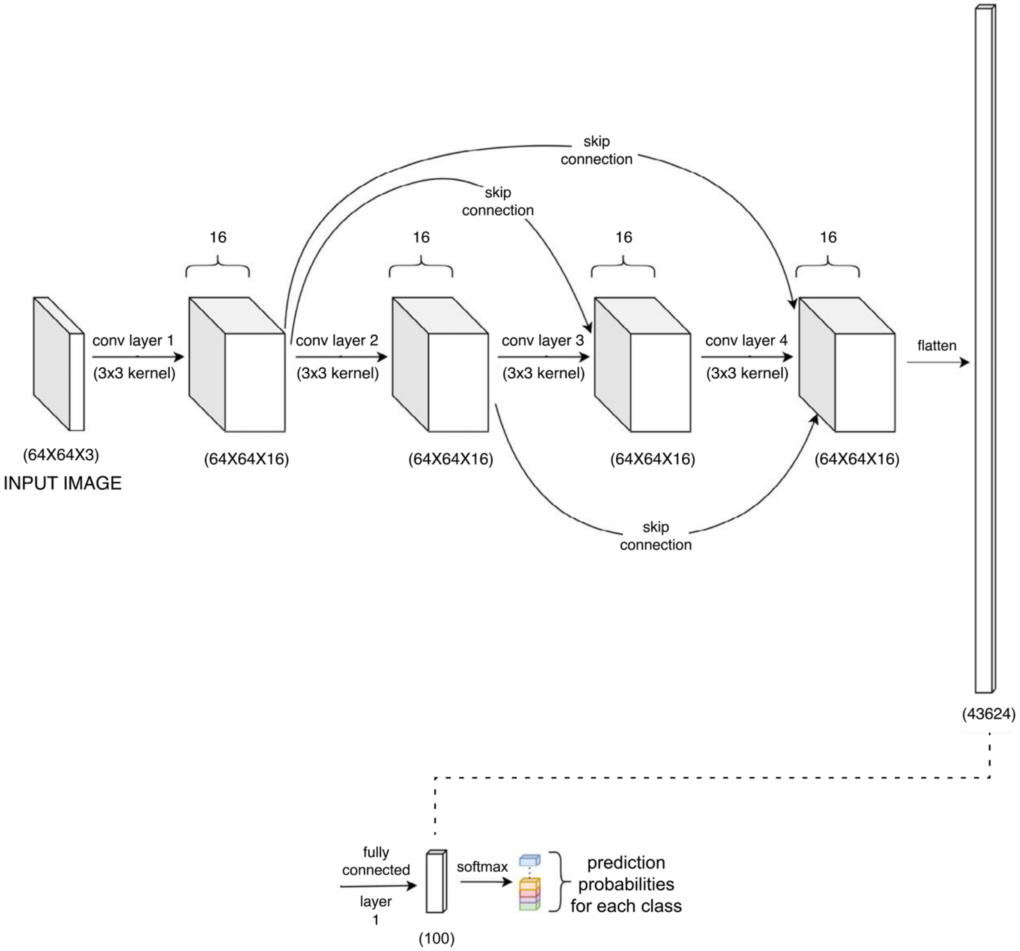


Figure 2.4: Multi-path CNN

A key advantage of multi-path architectures is a better flow of information across several layers, thanks to the skip connections. This, in turn, also lets the gradient flow back to the input layers without too much dissipation.

Having looked at the different architectural setups found in CNN models, we will now look at how CNNs have evolved over the years ever since they were first used.

Evolution of CNN architectures

CNNs have been in existence since 1989, when the first multi-layered CNN was developed by Yann LeCun. This model could perform the visual cognition task of identifying handwritten digits. In 1998, LeCun developed an improved ConvNet model called **LeNet**. Due to its high accuracy in optical recognition tasks, LeNet was adopted for industrial use soon after its invention. Ever since, CNNs have been successful not only in academic research but also in practical industry use cases. The following diagram shows a brief timeline of architectural developments in the lifetime of CNNs, starting from 1989 all the way to 2020:

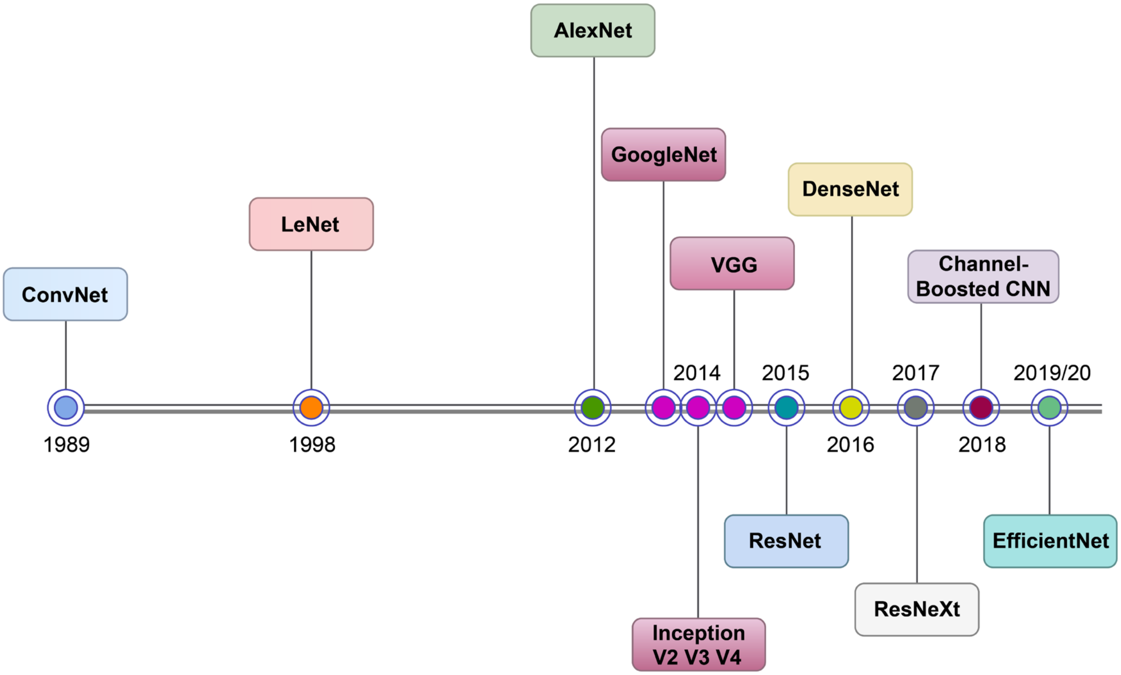


Figure 2.5: CNN architecture evolution – a broad picture

As we can see, there is a significant gap between the years 1998 and 2012. This was for two reasons:

i. There wasn't a dataset big and suitable enough to demonstrate the capabilities of CNNs, especially deep CNNs.

ii. The available computing power was limited.

And to add to the first reason, on the existing small datasets of the time such as MNIST, classical machine learning models such as SVMs were starting to beat CNN's performance.

The above two limitations were alleviated as we transitioned from 1998 to 2012 and beyond. Firstly, we had an exponential growth in digital data thanks to the advent of the internet and access to affordable devices such as digital cameras and smartphones. Secondly, we saw an enormous increase in our computational capabilities including the arrival of GPUs.

These changes led to a few CNN developments. The ReLU activation function was developed in order to deal with the gradient explosion and decay problem during backpropagation. Non-random initialization of network parameter values proved to be crucial. **Max pooling** was invented as an effective method for subsampling. GPUs were getting popular for training neural networks, especially CNNs, at scale.

Finally, and most importantly, a large-scale dedicated dataset of annotated images called **ImageNet** [1] was created by a research group at Stanford. This dataset is still one of the primary benchmarking datasets for CNN models to date.

With all of these developments compounding over the years, in 2012, a different architectural design brought about a massive improvement in CNN performance on the `ImageNet` dataset. This network was called **AlexNet** (named after the creator, Alex Krizhevsky). AlexNet, along with having various novel aspects such as random cropping and pretraining, established the trend of uniform and modular convolutional layer design. The uniform and modular layer structure was taken forward by repeatedly stacking such modules (of convolutional layers), resulting in very deep CNNs also known as **VGGs.**

Another approach of branching the blocks/modules of convolutional layers and stacking these branched blocks on top of each other proved extremely effective for tailored visual tasks. This network was called **GoogLeNet** (as it was developed at Google) or **Inception v1** (inception being the term for those branched blocks). Several variants of the VGG and Inception networks followed, such as **VGG16**, **VGG19**, **Inception v2**, **Inception v3**, and so on.

The next phase of development began with **skip connections**. To tackle the problem of gradient decay while training CNNs, non-consecutive layers were connected via skip connections lest information dissipate between them due to small gradients. It should be noted that skip connections are essentially a special case of multi-path-based CNNs discussed earlier. A popular type of network that emerged with this trick, among other novel characteristics such as batch normalization, was **ResNet**.

A logical extension of ResNet was **DenseNet**, where layers were densely connected to each other; that is, each layer gets the input from all the previous layers' output feature maps. Furthermore, hybrid architectures were then developed by mixing successful architectures from the past such as **Inception-ResNet** and **ResNeXt**, where the parallel branches within a block were increased in number.

Lately, the **channel boosting** technique has proven useful in improving CNN performance. The idea here is to learn novel features and exploit pre-learned features through transfer learning. Most recently, automatically designing new blocks and finding optimal CNN architectures has been a growing trend in CNN research. Examples of such CNNs are **MnasNets** and **EfficientNets**. The approach behind these models is to perform a neural architecture search to deduce an optimal CNN architecture with a uniform model scaling approach.

In the next section, we will go back to one of the earliest CNN models and take a closer look at the various CNN architectures developed since. We will build these architectures using PyTorch, training some of the models on real-world datasets. We will also explore PyTorch's pretrained CNN models repository, popularly known as **model-zoo**. We will learn how to fine-tune these pretrained models as well as running predictions on them.

Developing LeNet from scratch

LeNet, originally known as **LeNet-5**, is one of the earliest CNN models, developed in 1998. The number 5 in LeNet-5 represents the *total number of layers* in this model, that is, two convolutional and three fully connected layers. With roughly 60,000 total parameters, this model gave state-of-the-art performance on image recognition tasks for handwritten digit images in the year 1998. As expected from a CNN model, LeNet demonstrated rotation, position, and scale invariance as well as robustness against distortion in images. Contrary to the classical machine learning models of the time, such as SVMs, which treated each pixel of the image separately, LeNet exploited the correlation among neighboring pixels.

Note that although LeNet was developed for handwritten digit recognition, it can certainly be extended for other image classification tasks, as we shall see in our next exercise. The following diagram shows the architecture of a LeNet model:

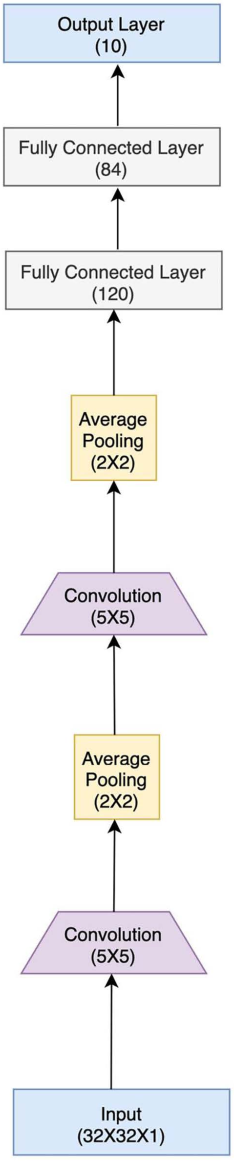


Figure 2.6: LeNet architecture

As mentioned earlier, there are two convolutional layers followed by three fully connected layers (including the output layer). This approach of stacking convolutional layers followed by fully connected layers later became a common practice in CNN research and is still applied to the latest CNN models.

This is because as we reach the final convolutional layer output, the output has small spatial dimensions (length and width) but a high depth, which makes the output look like an embedding of the input image. This embedding is like a vector that can be fed into a fully connected network, which is essentially a bunch of fully connected layers. Besides these layers, there are pooling layers in between. These are basically subsampling layers that reduce the spatial size of image representation, thereby reducing the number of parameters and computations as well as effectively condensing the input information. The pooling layer used in LeNet was an average pooling layer that had trainable weights. Soon after, **max pooling** emerged as the most commonly used pooling function in CNNs.

The numbers in brackets in each layer in the figure demonstrate the dimensions (for input, output, and fully connected layers) or window size (for convolutional and pooling layers). The expected input size for a grayscale image is 32x32 pixels. This image is then operated on by 5x5 convolutional kernels, followed by 2x2 pooling, and so on. The output layer size is 10, representing the 10 classes.

In this section, we will use PyTorch to build LeNet from scratch and train and evaluate it on a dataset of images for the task of image classification. We will see how easy and intuitive it is to build the network architecture in PyTorch using the outline from *Figure 2.6*.

Furthermore, we will demonstrate how effective LeNet is, even on a dataset different from the ones it was originally developed on (that is, MNIST) and how PyTorch makes it easy to train and test the model in a few lines of code.

Using PyTorch to build LeNet

Observe the following steps to build the model:

1. For this exercise, we will need to import a few dependencies. Execute the following `import` statements:

   ```
   import numpy as np
   import matplotlib.pyplot as plt
   import torch
   import torchvision
   import torch.nn as nn
   import torch.nn.functional as F
   import torchvision.transforms as transforms
   torch.use_deterministic_algorithms(True)
   ```

   Besides the usual imports, we also invoke the `use_deterministic_algorithms` function to ensure the reproducibility of this exercise.

2. Next, we will define the model architecture based on the outline given in *Figure 2.6*:

   ```
   class LeNet(nn.Module):
       def __init__(self):
           super(LeNet, self).__init__()
           # 3 input image channel, 6 output
   ```

```
        # feature maps and 5x5 conv kernel
        self.cn1 = nn.Conv2d(3, 6, 5)
        # 6 input image channel, 16 output
        # feature maps and 5x5 conv kernel
        self.cn2 = nn.Conv2d(6, 16, 5)
        # fully connected layers of size 120, 84 and 10
        # 5*5 is the spatial dimension at this layer
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
    def forward(self, x):
        # Convolution with 5x5 kernel
        x = F.relu(self.cn1(x))
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(x, (2, 2))
        # Convolution with 5x5 kernel
        x = F.relu(self.cn2(x))
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(x, (2, 2))
        # Flatten spatial and depth dimensions
        # into a single vector
        x = x.view(-1, self.flattened_features(x))
        # Fully connected operations
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
    def flattened_features(self, x):
        # all except the first (batch) dimension
        size = x.size()[1:]
        num_feats = 1
        for s in size:
            num_feats *= s
        return num_feats
lenet = LeNet()
print(lenet)
```

In the last two lines, we instantiate the model and print the network architecture. The output will be as follows:

```
LeNet(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

There are the usual `__init__` and `forward` methods for architecture definition and running a forward pass, respectively. The additional `flattened_features` method is meant to calculate the total number of features in an image representation layer (usually an output of a convolutional layer or pooling layer). This method helps to flatten the spatial representation of features into a single vector of numbers, which is then used as input to fully connected layers.

Besides the details of the architecture mentioned earlier, ReLU is used throughout the network as the activation function. Also, unlike the original LeNet network, which takes in single-channel images, the current model is modified to accept RGB images, that is, three channels, as input. This is done in order to adapt to the dataset that is used for this exercise.

3. We then define the training routine, that is, the actual backpropagation step:

```
def train(net, trainloader, optim, epoch):
    # initialize loss
    loss_total = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        # ip refers to the input images, and ground_truth
        # refers to the output classes the images belong to
        ip, ground_truth = data
        # zero the parameter gradients
        optim.zero_grad()
        # forward-pass + backward-pass + optimization -step
        op = net(ip)
        loss = nn.CrossEntropyLoss()(op, ground_truth)
        loss.backward()
        optim.step()
        # update loss
        loss_total += loss.item()
        # print loss statistics
        if (i+1) % 1000 == 0:
            # print at the interval of 1000 mini-batches
            print('[Epoch number : %d, Mini-batches: %5d] \
                  loss: %.3f' % (epoch + 1, i + 1,
                                 loss_total / 200))
            loss_total = 0.0
```

For each epoch, this function iterates through the entire training dataset, runs a forward pass through the network, and, using backpropagation, updates the parameters of the model based on the specified optimizer. After iterating through each of the 1,000 mini-batches of the training dataset, this method also logs the calculated loss.

4. Similar to the training routine, we will define the test routine that we will use to evaluate model performance:

```
def test(net, testloader):
    success = 0
    counter = 0
    with torch.no_grad():
        for data in testloader:
            im, ground_truth = data
            op = net(im)
            _, pred = torch.max(op.data, 1)
            counter += ground_truth.size(0)
            success += (pred == ground_truth).sum().item()
    print('LeNet accuracy on 10000 images from test dataset: %d %%'\
        % (100 * success / counter))
```

This function runs a forward pass through the model for each test-set image, calculates the correct number of predictions, and prints the percentage of correct predictions on the test set.

5. Before we get on to training the model, we need to load the dataset. For this exercise, we will be using the `CIFAR-10` dataset.

Dataset citation

The images in this section are from *Learning Multiple Layers of Features from Tiny Images*, Alex Krizhevsky, 2009: `https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`. They are part of the CIFAR-10 dataset (toronto.edu): `https://www.cs.toronto.edu/~kriz/cifar.html`

This dataset consists of 60,000 32x32 RGB images labeled across 10 classes, with 6,000 images per class. The 60,000 images are split into 50,000 training images and 10,000 test images. More details can be found at the dataset website [2]. Torch provides the `CIFAR10` dataset under the `torchvision.datasets` module. We will be using the module to directly load the data and instantiate train and test dataloaders as demonstrated in the following code:

```
# The mean and std are kept as 0.5 for normalizing
# pixel values as the pixel values are originally
# in the range 0 to 1
train_transform = transforms.Compose(
    [transforms.RandomHorizontalFlip(),
     transforms.RandomCrop(32, 4),
     transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5),
                          (0.5, 0.5, 0.5))])
trainset = torchvision.datasets.CIFAR10(root='./data',
    train=True, download=True, transform=train_transform)
trainloader = torch.utils.data.DataLoader(trainset,
    batch_size=8, shuffle=True)
test_transform = transforms.Compose([transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5),
                         (0.5, 0.5, 0.5))])
testset = torchvision.datasets.CIFAR10(root='./data',
    train=False, download=True, transform=test_transform)
testloader = torch.utils.data.DataLoader(testset,
    batch_size=10000, shuffle=False)
# ordering is important
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog',
           'frog', 'horse', 'ship', 'truck')
```

In the next chapter, we will download the dataset and write a custom dataset class and a `dataloader` function. We will not need to write those here, thanks to the `torchvision.datasets` module.

Because we set the `download` flag to `True`, the dataset will be downloaded locally. Then, we shall see the following output:

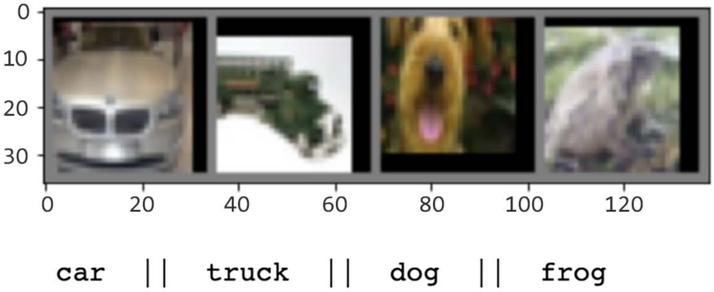
```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./
data/cifar-10-python.tar.gz
100%
170498071/170498071 [00:34<00:00, 5191345.41it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

The transformations used for training and testing datasets are different because we apply some data augmentation to the training dataset, such as flipping and cropping, which are not applicable to the test dataset. Also, after defining `trainloader` and `testloader`, we declare the 10 classes in this dataset with a pre-defined ordering.

6. After loading the datasets, let's investigate how the data looks:

```
# define a function that displays an image
def imageshow(image):
    # un-normalize the image
    image = image/2 + 0.5
    npimage = image.numpy()
    plt.imshow(np.transpose(npimage, (1, 2, 0)))
    plt.show()
# sample images from training set
dataiter = iter(trainloader)
images, labels = next(dataiter)
# display images in a grid
num_images = 4
imageshow(torchvision.utils.make_grid(images[:num_images]))
# print labels
print('    '+'  ||  '.join(classes[labels[j]]
                           for j in range(num_images)))
```

The preceding code shows us four sample images with their respective labels from the training dataset. The output will be as follows:



car || truck || dog || frog

Figure 2.7: CIFAR-10 dataset samples

The preceding output shows us four color images that are 32x32 pixels in size. These four images belong to four different labels, as displayed in the text following the images.

We will now train the LeNet model.

Training LeNet

Let us train the model with the help of the following steps:

1. We will define the `optimizer` and start the training loop as shown here:

```
# define optimizer
optim = torch.optim.Adam(lenet.parameters(), lr=0.001)
# training loop over the dataset multiple times
for epoch in range(50):
    train(lenet, trainloader, optim, epoch)
    print()
    test(lenet, testloader)
    print()
print('Finished Training')
```

The output will be as follows:

```
[Epoch number : 1, Mini-batches:  1000] loss: 9.804
[Epoch number : 1, Mini-batches:  2000] loss: 8.783
[Epoch number : 1, Mini-batches:  3000] loss: 8.444
[Epoch number : 1, Mini-batches:  4000] loss: 8.118
[Epoch number : 1, Mini-batches:  5000] loss: 7.819
[Epoch number : 1, Mini-batches:  6000] loss: 7.672
LeNet accuracy on 10000 images from test dataset: 44 %
...
[Epoch number : 50, Mini-batches:  1000] loss: 5.022
[Epoch number : 50, Mini-batches:  2000] loss: 5.067
[Epoch number : 50, Mini-batches:  3000] loss: 5.137
[Epoch number : 50, Mini-batches:  4000] loss: 5.009
[Epoch number : 50, Mini-batches:  5000] loss: 5.107
[Epoch number : 50, Mini-batches:  6000] loss: 4.977
LeNet accuracy on 10000 images from test dataset: 67 %
Finished Training
```

2. Once the training is finished, we can save the model file locally:

```
model_path = './cifar_model.pth'
torch.save(lenet.state_dict(), model_path)
```

Having trained the LeNet model, we will now test its performance on the test dataset in the next section.

Testing LeNet

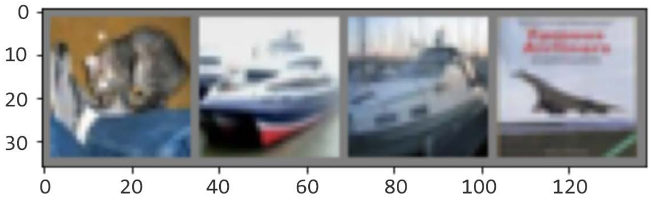
The following steps need to be followed to test the LeNet model:

1. Let's make predictions by loading the saved model and running it on the test dataset:

```
# load test dataset images
d_iter = iter(testloader)
im, ground_truth = next(d_iter)
# print images and ground truth
imageshow(torchvision.utils.make_grid(im[:4]))
print('Label:      ', ' '.join('%5s' %
                               classes[ground_truth[j]]
                               for j in range(4)))
```

```
# load model
lenet_cached = LeNet()
lenet_cached.load_state_dict(torch.load(model_path))
# model inference
op = lenet_cached(im)
# print predictions
_, pred = torch.max(op, 1)
print('Prediction: ', ' '.join('%5s' % classes[pred[j]]
                               for j in range(4)))
```

The output will be as follows:



```
Label:         cat  ship  ship plane
Prediction:    cat   car  ship plane
```

Figure 2.8: LeNet predictions

Evidently, three out of four predictions are correct.

2. Finally, we will check the overall accuracy of this model on the test dataset as well as the per-class accuracy:

```
success = 0
counter = 0
with torch.no_grad():
    for data in testloader:
        im, ground_truth = data
        op = lenet_cached(im)
        _, pred = torch.max(op.data, 1)
        counter += ground_truth.size(0)
        success += (pred == ground_truth).sum().item()
print('Model accuracy on 10000 images from test dataset: %d %%'\
    % (100 * success / counter))
```

The output will be as follows:

```
Model accuracy on 10000 images from test dataset: 67 %
```

3. For per-class accuracy, the code is as follows:

```
class_sucess = list(0. for i in range(10))
class_counter = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        im, ground_truth = data
        op = lenet_cached(im)
        _, pred = torch.max(op, 1)
        c = (pred == ground_truth).squeeze()
        for i in range(10000):
            ground_truth_curr = ground_truth[i]
            class_sucess[ground_truth_curr] += c[i].item()
            class_counter[ground_truth_curr] += 1
for i in range(10):
    print('Model accuracy for class %5s : %2d %%' % (
        classes[i], 100 * class_sucess[i] / class_counter[i]))
```

The output will be as follows:

```
Model accuracy for class plane : 70 %
Model accuracy for class   car : 83 %
Model accuracy for class  bird : 45 %
Model accuracy for class   cat : 37 %
Model accuracy for class  deer : 80 %
Model accuracy for class   dog : 52 %
Model accuracy for class  frog : 81 %
Model accuracy for class horse : 71 %
Model accuracy for class  ship : 76 %
Model accuracy for class truck : 74 %
```

Some classes have better performance than others. Overall, the model is far from perfect (that is, 100% accuracy) but much better than a model making random predictions, which would have an accuracy of 10% (due to the 10 classes).

Having built a LeNet model from scratch and evaluated its performance using PyTorch, we will now move on to a successor of LeNet – **AlexNet**. For LeNet, we built the model from scratch, trained, and tested it. For AlexNet, we will use a pretrained model, fine-tune it on a smaller dataset, and test it.

Fine-tuning the AlexNet model

In this section, we will first take a quick look at the AlexNet architecture and how to build one using PyTorch. Then we will explore PyTorch's pretrained CNN models repository, and finally, use a pre-trained AlexNet model for fine-tuning on an image classification task, as well as making predictions.

AlexNet is a successor of LeNet with incremental changes in the architecture, such as 8 layers (5 convolutional and 3 fully connected) instead of 5, and 60 million model parameters instead of 60,000, as well as using `MaxPool` instead of `AvgPool`. Moreover, AlexNet was trained and tested on a much bigger dataset – ImageNet, which is over 100 GB in size – as opposed to the MNIST dataset (on which LeNet was trained), which amounts to a few MB. AlexNet truly revolutionized CNNs as it emerged as a significantly more powerful class of models on image-related tasks than the other classical machine learning models, such as SVMs. *Figure 2.9* shows the AlexNet architecture:

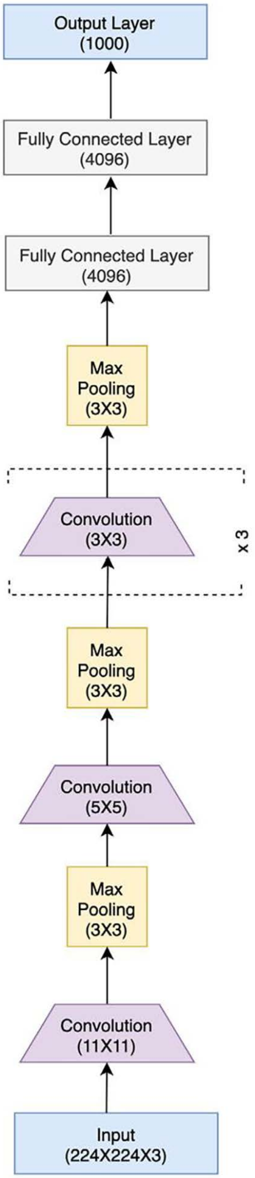


Figure 2.9: AlexNet architecture

As we can see, the architecture follows the common theme from LeNet of having convolutional layers stacked sequentially, followed by a series of fully connected layers toward the output end. PyTorch makes it easy to translate such a model architecture into actual code. This can be seen in the following PyTorch-code-equivalent of the architecture:

```
class AlexNet(nn.Module):
    def __init__(self, number_of_classes):
        super(AlexNet, self).__init__()
        self.feats = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=64,
                      kernel_size=11, stride=4, padding=5),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(in_channels=64, out_channels=192,
                      kernel_size=5, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(in_channels=192, out_channels=384,
                      kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=384, out_channels=256,
                      kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=256, out_channels=256,
                      kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.clf = nn.Linear(in_features=256, out_features=number_of_classes)
    def forward(self, inp):
        op = self.feats(inp)
        op = op.view(op.size(0), -1)
        op = self.clf(op)
        return op
```

The code is quite self-explanatory, wherein the `__init__` function contains the initialization of the whole layered structure, consisting of convolutional, pooling, and fully connected layers, along with ReLU activations. The `forward` function simply runs a data point x through this initialized network. Please note that the second line of the `forward` method already performs the flattening operation so that we need not define that function separately as we did for LeNet.

But besides the option of initializing the model architecture and training it ourselves, PyTorch, with its `torchvision` package, provides a `models` sub-package, which contains definitions of CNN models meant for solving different tasks, such as image classification, semantic segmentation, object detection, and so on. The following is a non-exhaustive list of available models for the task of image classification [3]:

- AlexNet
- VGG
- ResNet
- SqueezeNet
- DenseNet
- Inception v3
- GoogLeNet
- ShuffleNet v2
- MobileNet v2
- ResNeXt
- Wide ResNet
- MnasNet
- EfficientNet

In the next section, we will use a pretrained AlexNet model as an example and demonstrate how to fine-tune it using PyTorch in the form of an exercise.

Using PyTorch to fine-tune AlexNet

In the following exercise, we will load a pretrained AlexNet model and fine-tune it on an image classification dataset different from ImageNet (on which it was originally trained). Finally, we will test the fine-tuned model's performance to see if it could transfer-learn from the new dataset. Some parts of the code in the exercise are trimmed for readability but you can find the full code in our GitHub repository [4].

For this exercise, we will need to import a few dependencies. Execute the following `import` statements:

```
import os
import time
import copy
import numpy as np
import matplotlib.pyplot as plt
import torch
import torchvision
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
from torchvision import datasets, models, transforms
torch.use_deterministic_algorithms(True)
```

Next, we will download and transform the dataset. For this fine-tuning exercise, we will use a small image dataset of bees and ants. There are 240 training images and 150 validation images divided equally between the two classes (bees and ants).

We download the dataset from Kaggle [5] and store it in the current working directory. More information about the dataset can be found at the dataset's website [6].

Dataset citation

Elsik, C. G., Tayal, A., Diesh, C. M., Unni, D. R., Emery, M. L., Nguyen, H. N., Hagen, D. E. *Hymenoptera Genome Database: integrating genome annotations* in HymenopteraMine. Nucleic Acids Research, 2016, Jan. 4; 44(D1):D793-800. DOI: 10.1093/nar/gkv1208. Epub 2015, Nov. 17. PubMed PMID: 26578564.

To download the dataset, you will need to log in to Kaggle. If you do not already have a Kaggle account, you will need to register. Let's download and transform the dataset:

```
# Creating a local data directory
ddir = 'hymenoptera_data'

# Data normalization and augmentation transformations
# for train dataset
# Only normalization transformation for validation dataset
# The mean and std for normalization are calculated as the
# mean of all pixel values for all images in the training
# set per each image channel - R, G and B
data_transformers = {
    'train': transforms.Compose([transforms.RandomResizedCrop(224),
                                 transforms.RandomHorizontalFlip(),
                                 transforms.ToTensor(),
                                 transforms.Normalize(
                                     [0.490, 0.449, 0.411],
                                     [0.231, 0.221, 0.230])]),
    'val': transforms.Compose([transforms.Resize(256),
                               transforms.CenterCrop(224),
                               transforms.ToTensor(),
                               transforms.Normalize(
                                   [0.490, 0.449, 0.411],
                                   [0.231, 0.221, 0.230])])}

img_data = {k: datasets.ImageFolder(os.path.join(ddir, k), data_transformers[k])
            for k in ['train', 'val']}
```

```
dloaders = {k: torch.utils.data.DataLoader(img_data[k], batch_size=8,
                                            shuffle=True)
            for k in ['train', 'val']}
dset_sizes = {x: len(img_data[x]) for x in ['train', 'val']}
classes = img_data['train'].classes
dvc = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

Now that we have completed the pre-requisites, let's begin:

1. Let's visualize some sample training dataset images:

```
def imageshow(img, text=None):
    img = img.numpy().transpose((1, 2, 0))
    avg = np.array([0.490, 0.449, 0.411])
    stddev = np.array([0.231, 0.221, 0.230])
    img = stddev * img + avg
    img = np.clip(img, 0, 1)
    plt.imshow(img)
    if text is not None:
        plt.title(text)
# Generate one train dataset batch
imgs, cls = next(iter(dloaders['train']))
# Generate a grid from batch
grid = torchvision.utils.make_grid(imgs)
imageshow(grid, text=[classes[c] for c in cls])
```

We have used the `np.clip()` method from `numpy` to ensure that the image pixel values are restricted between `0` and `1` to make the visualization clear. The output will be as follows:

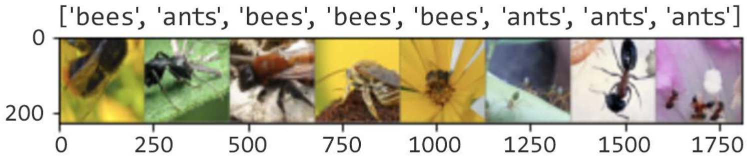


Figure 2.10: Bees versus ants dataset

2. We now define the fine-tuning routine, which is essentially a training routine performed on a pretrained model:

```
def finetune_model(pretrained_model, loss_func, optim, epochs=10):
    ...
    for e in range(epochs):
        for dset in ['train', 'val']:
            if dset == 'train':
                # set model to train mode
                # (i.e. trainbale weights)
                pretrained_model.train()
            else:
                # set model to validation mode
                pretrained_model.eval()
            # iterate over the (training/validation) data.
            for imgs, tgts in dloaders[dset]:
                ...
                optim.zero_grad()
                with torch.set_grad_enabled(dset == 'train'):
                    ops = pretrained_model(imgs)
                    _, preds = torch.max(ops, 1)
                    loss_curr = loss_func(ops, tgts)
                    # backward pass only if in training mode
                    if dset == 'train':
                        loss_curr.backward()
                        optim.step()
                loss += loss_curr.item() * imgs.size(0)
                successes += torch.sum(preds == tgts.data)
            loss_epoch = loss / dset_sizes[dset]
            accuracy_epoch = successes.double() / dset_sizes[dset]
            if dset == 'val' and accuracy_epoch > accuracy:
                accuracy = accuracy_epoch
                model_weights = copy.deepcopy(
                    pretrained_model.state_dict())
    # load the best model version (weights)
    pretrained_model.load_state_dict(model_weights)
    return pretrained_model
```

In this function, we require the pretrained model (that is, the architecture as well as the weights) as input along with the loss function, optimizer, and number of epochs. Basically, instead of starting from a random initialization of weights, we start with the pretrained weights of AlexNet. The other parts of this function are pretty similar to our previous exercises.

3. Before starting to fine-tune (train) the model, we will define a function to visualize the model predictions:

```
def visualize_predictions(pretrained_model, max_num_imgs=4):
   was_model_training = pretrained_model.training
    pretrained_model.eval()
    imgs_counter = 0
    fig = plt.figure()
    with torch.no_grad():
        for i, (imgs, tgts) in enumerate(dloaders['val']):
            imgs = imgs.to(dvc)
            tgts = tgts.to(dvc)
            ops = pretrained_model(imgs)
            _, preds = torch.max(ops, 1)
            for j in range(imgs.size()[0]):
                imgs_counter += 1
                ax = plt.subplot(max_num_imgs//2, 2, imgs_counter)
                ax.axis('off')
                ax.set_title(f'Prediction:
                             {classes[preds[j]]},
                             Ground Truth:
                             {classes[tgts[j]]}')
                imageshow(imgs.cpu().data[j])
                if imgs_counter == max_num_imgs:
                   pretrained_model.train(mode=was_model_training)
                    return
        pretrained_model.train(mode=was_model_training)
```

4. Finally, we get to the interesting part. Let's use PyTorch's `torchvision.models` sub-package to load the pretrained AlexNet model:

```
model_finetune = models.alexnet(pretrained=True)
```

This model object has the following two main components:

i. `features`: The feature extraction component, which contains all the convolutional and pooling layers

ii. `classifier`: The classifier block, which contains all the fully connected layers leading to the output layer

5. We can visualize these components as shown here:

```
print(model_finetune.features)
```

This should output the following:

```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
  (1): ReLU(inplace=True)
  (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_
mode=False)
  (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (4): ReLU(inplace=True)
  (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_
mode=False)
  (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (7): ReLU(inplace=True)
  (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (9): ReLU(inplace=True)
  (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (11): ReLU(inplace=True)
  (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_
mode=False)
```

6. Next, we inspect the `classifier` block as follows:

```
print(model_finetune.classifier)
```

This should output the following:

```
Sequential(
  (0): Dropout(p=0.5, inplace=False)
  (1): Linear(in_features=9216, out_features=4096, bias=True)
  (2): ReLU(inplace=True)
  (3): Dropout(p=0.5, inplace=False)
  (4): Linear(in_features=4096, out_features=4096, bias=True)
  (5): ReLU(inplace=True)
  (6): Linear(in_features=4096, out_features=1000, bias=True)
```

7. As you may have noticed, the pretrained model has an output layer of size `1000`, but we only have `2` classes in our fine-tuning dataset. So, we shall alter that, as shown here:

```
# change the last layer from 1000 classes to 2 classes
model_finetune.classifier[6] = nn.Linear(4096, len(classes))
```

8. And now, we are all set to define the optimizer and loss function, and thereafter run the training routine as follows:

```
loss_func = nn.CrossEntropyLoss()
optim_finetune = optim.SGD(model_finetune.parameters(), lr=0.0001)
# train (fine-tune) and validate the model
model_finetune = finetune_model(model_finetune, loss_func,
                                optim_finetune, epochs=10)
```

The output will be as follows:

```
Epoch number 0/9
====================
train loss in this epoch: 0.6528244360548551, accuracy in this epoch:
0.610655737704918
val loss in this epoch: 0.5563900120118085, accuracy in this epoch:
0.7320261437908496
Epoch number 1/9
====================
train loss in this epoch: 0.5144887796190919, accuracy in this epoch:
0.75
val loss in this epoch: 0.4758027388769038, accuracy in this epoch:
0.803921568627451
Epoch number 2/9
====================
train loss in this epoch: 0.4620713156754853, accuracy in this epoch:
0.7950819672131147
val loss in this epoch: 0.4326762077855129, accuracy in this epoch:
0.803921568627451
...
Epoch number 7/9
====================
train loss in this epoch: 0.3297723409582357, accuracy in this epoch:
0.860655737704918
val loss in this epoch: 0.3347476099441254, accuracy in this epoch:
0.869281045751634
Epoch number 8/9
====================
train loss in this epoch: 0.32671376110100353, accuracy in this epoch:
0.8524590163934426
val loss in this epoch: 0.32516936344258923, accuracy in this epoch:
0.8823529411764706
Epoch number 9/9
====================
train loss in this epoch: 0.3130935803055763, accuracy in this epoch:
0.8770491803278688
val loss in this epoch: 0.3200583465251268, accuracy in this epoch:
0.8888888888888888
Training finished in 5.0mins 50.6720712184906secs
Best validation set accuracy: 0.8888888888888888
```

9. Let's visualize some of the model predictions to see whether the model has indeed learned the relevant features from this small dataset:

   ```
   visualize_predictions(model_finetune)
   ```

   This should output the following:

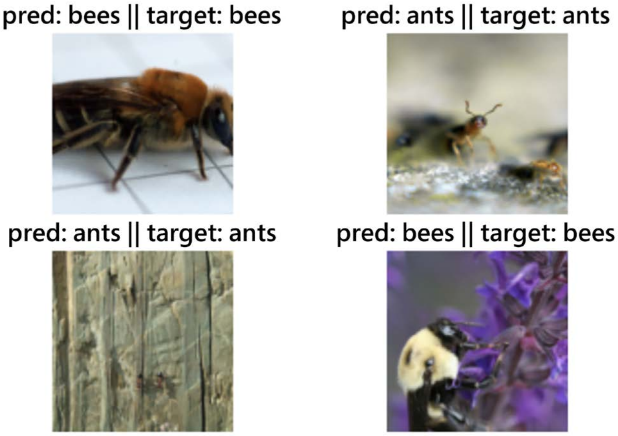

pred: bees || target: bees
pred: ants || target: ants
pred: ants || target: ants
pred: bees || target: bees

Figure 2.11: AlexNet predictions

   Clearly, the pretrained AlexNet model has been able to transfer-learn on this rather tiny image classification dataset. This demonstrates both the power of transfer learning as well as the speed and ease with which we can fine-tune well-known models using PyTorch.

In the next section, we will discuss an even deeper and more complex successor of AlexNet – the VGG network. We have demonstrated the model definition, dataset loading, model training (or fine-tuning), and evaluation steps in detail for LeNet and AlexNet. In subsequent sections, we will focus mostly on model architecture definition, as the PyTorch code for other aspects (such as data loading and evaluation) will be similar.

Running a pretrained VGG model

We have already discussed LeNet and AlexNet, two of the foundational CNN architectures. As we progress in the chapter, we will explore increasingly complex CNN models. That being said, the key principles in building these model architectures will be the same. We will see a modular model-building approach in putting together convolutional layers, pooling layers, and fully connected layers into blocks/modules and then stacking these blocks sequentially or in a branched manner. In this section, we look at the successor to AlexNet – VGGNet.

The name VGG is derived from the **Visual Geometry Group of Oxford University**, where this model was invented. Compared to the 8 layers and 60 million parameters of AlexNet, VGG consists of 13 layers (10 convolutional layers and 3 fully connected layers) and 138 million parameters. VGG basically stacks more layers onto the AlexNet architecture with smaller convolution kernels (2x2 or 3x3).

Hence, VGG's novelty lies in the unprecedented level of depth that it brings with its architecture. *Figure 2.12* shows the VGG architecture:

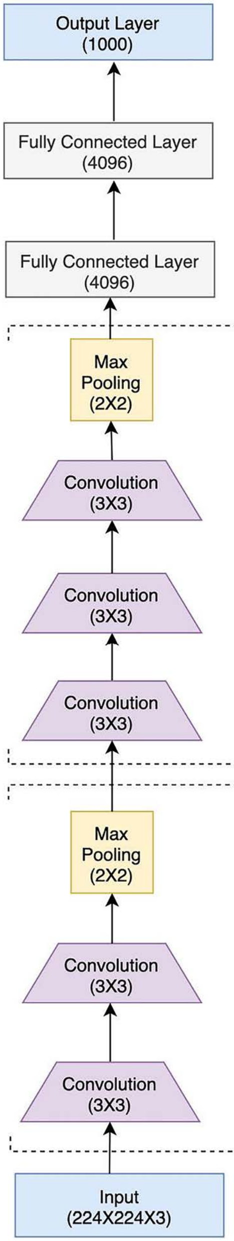


Figure 2.12: VGG16 architecture

The preceding VGG architecture is called **VGG13**, because of the 13 layers. Other variants are VGG16 and VGG19, consisting of 16 and 19 layers, respectively. There is another set of variants – **VGG13_bn**, **VGG16_bn**, and **VGG19_bn**, where **bn** suggests that these models also consist of **batch-normalization layers**.

PyTorch's `torchvision.model` sub-package provides the pretrained VGG model (with all of the six variants discussed earlier) trained on the ImageNet dataset. In the following exercise, we will use the pretrained VGG13 model to make predictions on a small dataset of bees and ants (used in the previous exercise). We will focus on the key pieces of code here, as most other parts of our code will overlap with that of the previous exercises. We can always refer to our notebooks to explore the full code [7]:

1. First, we need to import dependencies, including `torchvision.models`.
2. Download the data and set up the ants and bees dataset and dataloader, along with the transformations.
3. In order to make predictions on these images, we will need to download the 1,000 labels of the ImageNet dataset [8].
4. Once downloaded, we need to create a mapping between the class indices 0 to 999 and the corresponding class labels, as shown here:

```
import ast
with open('./imagenet1000_clsidx_to_labels.txt') as f:
    classes_data = f.read()
classes_dict = ast.literal_eval(classes_data)
print({k: classes_dict[k] for k in list(classes_dict)[:5]})
```

This should output the first five class mappings, as shown below:

```
{0: 'tench, Tinca tinca', 1: 'goldfish, Carassius auratus', 2: 'great
white shark, white shark, man-eater, man-eating shark, Carcharodon
carcharias', 3: 'tiger shark, Galeocerdo cuvieri', 4: 'hammerhead,
hammerhead shark'}
```

5. Define the model prediction visualization function that takes in the pretrained model object and the number of images to run predictions on. This function should output the images with predictions.
6. Load the pretrained `VGG13` model:

```
model_finetune = models.vgg13(pretrained=True)
```

The `VGG13` model is downloaded in this step.

FAQ – What is the disk size of a VGG13 model?

A `VGG13` model will consume roughly 508 MB on your hard disk.

7. Finally, we run predictions on our ants and bees dataset using this pretrained model:

```
visualize_predictions(model_finetune)
```

This should output the following:

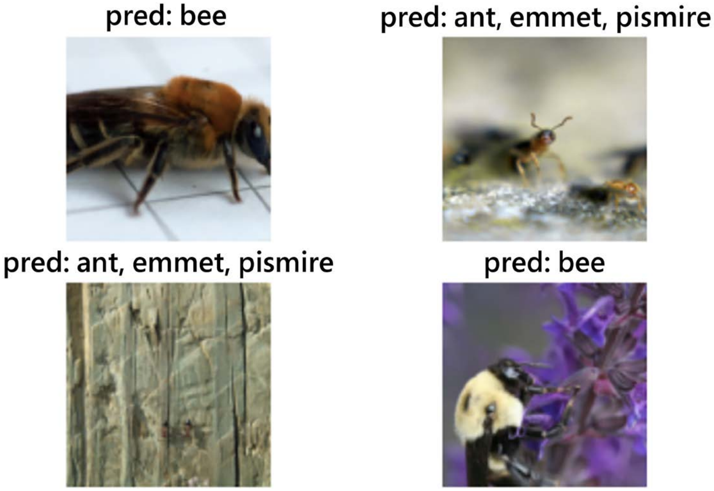


Figure 2.13: VGG13 predictions

The `VGG13` model trained on an entirely different dataset seems to predict all the test samples correctly in the ants and bees dataset. Basically, the model grabs the two most similar animals from the dataset out of the 1,000 classes and finds them in the images. By doing this exercise, we see that the model is still able to extract relevant visual features out of the images and the exercise demonstrates the utility of PyTorch's out-of-the-box inference feature.

In the next section, we are going to study a different type of CNN architecture – one that involves modules that have multiple parallel convolutional layers. The modules are called **Inception** modules and the resulting network is called the **Inception Network** – named after the movie *Inception* – because this model contains several branching modules much like the branching dreams of the movie. We will explore the various parts of this network and the reasoning behind its success. We will also build the Inception modules and the Inception network architecture using PyTorch.

Exploring GoogLeNet and Inception v3

As we have discovered the progression of CNN models from LeNet to VGG so far, we have observed the sequential stacking of more convolutional and fully connected layers. This resulted in deep networks with a lot of parameters to train. *GoogLeNet* emerged as a radically different type of CNN architecture that is composed of a module of parallel convolutional layers called the inception module. Because of this, GoogLeNet is also called **Inception v1** (v1 marked the first version as more versions came along later). Some of the drastically new elements introduced by GoogLeNet were the following:

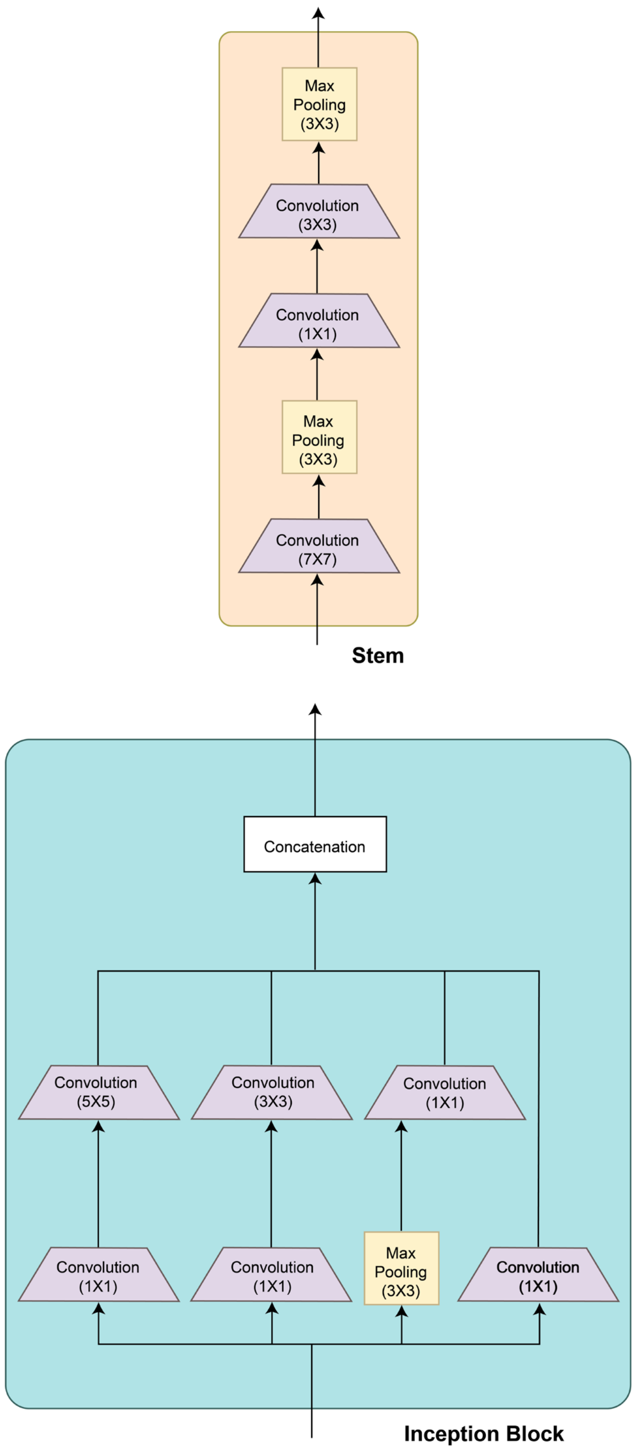
- The inception module – a module of several parallel convolutional layers
- Using **1x1 convolutions** to reduce the number of model parameters
- **Global average pooling** instead of a fully connected layer – reduces overfitting
- Using **auxiliary classifiers** for training – for regularization and gradient stability

GoogLeNet has 22 layers, which is more than the number of layers of any VGG model variant. Yet, due to some of the optimization tricks used, the number of parameters in GoogLeNet is 5 million, which is far less than the 138 million parameters of VGG. Let's expand on some of the key features of this model.

Inception modules

Perhaps the single most important contribution of this model was the development of a convolutional module with several convolutional layers running in parallel, which are finally concatenated to produce a single output vector. These parallel convolutional layers operate with different kernel sizes ranging from 1x1 to 3x3 to 5x5. The idea is to extract all levels of visual information from the image. Besides these convolutions, a 3x3 max pooling layer adds another level of feature extraction.

Figure 2.14 shows the inception block diagram along with the overall GoogLeNet architecture:



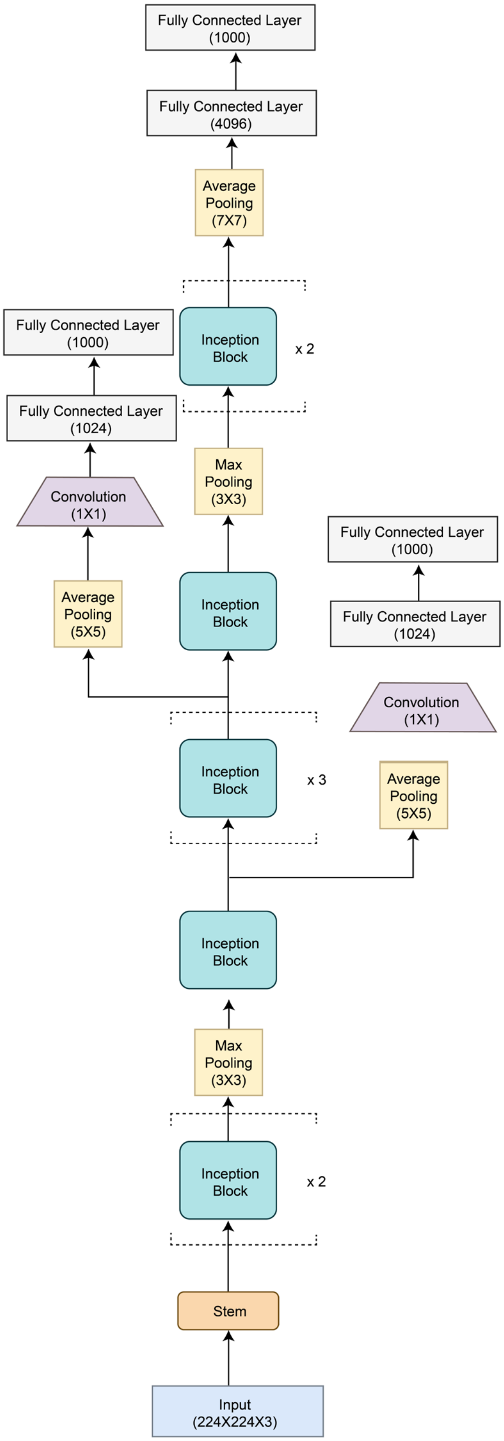


Figure 2.14: GoogLeNet architecture

By using this architecture diagram, we can build the inception module in PyTorch as shown here:

```
class InceptionModule(nn.Module):
    def __init__(self, input_planes, n_channels1x1, n_channels3x3red,
                 n_channels3x3, n_channels5x5red, n_channels5x5,
                 pooling_planes):
        super(InceptionModule, self).__init__()
        # 1x1 convolution branch
        self.block1 = nn.Sequential(
            nn.Conv2d(input_planes, n_channels1x1, kernel_size=1),
                      nn.BatchNorm2d(n_channels1x1),nn.ReLU(True),)
        # 1x1 convolution -> 3x3 convolution branch
        self.block2 = nn.Sequential(
            nn.Conv2d(input_planes, n_channels3x3red, kernel_size=1),
            nn.BatchNorm2d(n_channels3x3red),
            nn.ReLU(True), nn.Conv2d(n_channels3x3red, n_channels3x3,
                                     kernel_size=3, padding=1),
                                     nn.BatchNorm2d(n_channels3x3),
                                     nn.ReLU(True),)
        # 1x1 conv -> 5x5 conv branch
        self.block3 = nn.Sequential(
            nn.Conv2d(input_planes, n_channels5x5red, kernel_size=1),
            nn.BatchNorm2d(n_channels5x5red),
            nn.ReLU(True), nn.Conv2d(n_channels5x5red, n_channels5x5,
                                     kernel_size=3, padding=1),
                                     nn.BatchNorm2d(n_channels5x5),
                                     nn.ReLU(True),
            nn.Conv2d(n_channels5x5, n_channels5x5,
                      kernel_size=3, padding=1),
                      nn.BatchNorm2d(n_channels5x5),
                      nn.ReLU(True),)
        # 3x3 pool -> 1x1 conv branch
        self.block4 = nn.Sequential(
            nn.MaxPool2d(3, stride=1, padding=1),
            nn.Conv2d(input_planes, pooling_planes, kernel_size=1),
            nn.BatchNorm2d(pooling_planes),
            nn.ReLU(True),)
    def forward(self, ip):
        op1 = self.block1(ip)
        op2 = self.block2(ip)
        op3 = self.block3(ip)
        op4 = self.block4(ip)
        return torch.cat([op1,op2,op3,op4], 1)
```

Next, we will look at another important feature of GoogLeNet – 1x1 convolutions.

1x1 convolutions

In addition to the parallel convolutional layers in an inception module, each parallel layer has a preceding **1x1 convolutional layer**. The reason behind using these 1x1 convolutional layers is *dimensionality reduction*. 1x1 convolutions do not change the width and height of the image representation but can alter the depth of an image representation. This trick is used to reduce the depth of the input visual features before performing the 1x1, 3x3, and 5x5 convolutions parallelly. Reducing the number of parameters not only helps build a lighter model but also combats overfitting.

Global average pooling

If we look at the overall GoogLeNet architecture in *Figure 2.14*, the penultimate output layer of the model is preceded by a 7x7 average pooling layer. This layer again helps in reducing the number of parameters of the model, thereby reducing overfitting. Without this layer, the model would have millions of additional parameters due to the dense connections of a fully connected layer.

Auxiliary classifiers

Figure 2.14 also shows two extra or auxiliary output branches in the model. These auxiliary classifiers are supposed to tackle the vanishing gradient problem by adding to the gradients' magnitude during backpropagation, especially for the layers toward the input end. Because these models have a large number of layers, vanishing gradients can become a major limitation. Hence, using auxiliary classifiers has proven useful for this 22-layer deep model. Additionally, the auxiliary branches also help in regularization. Please note that these auxiliary branches are switched off/discarded while making predictions.

Once we have the inception module defined using PyTorch, we can easily instantiate the entire Inception v1 model as follows:

```
class GoogLeNet(nn.Module):
    def __init__(self):
        super(GoogLeNet, self).__init__()
        self.stem = nn.Sequential(
            nn.Conv2d(3, 192, kernel_size=3, padding=1),
            nn.BatchNorm2d(192),
            nn.ReLU(True),)
        self.im1 = InceptionModule(192,  64,  96, 128, 16, 32, 32)
        self.im2 = InceptionModule(256, 128, 128, 192, 32, 96, 64)
        self.max_pool = nn.MaxPool2d(3, stride=2, padding=1)
        self.im3 = InceptionModule(480, 192,  96, 208, 16,  48,  64)
        self.im4 = InceptionModule(512, 160, 112, 224, 24,  64,  64)
        self.im5 = InceptionModule(512, 128, 128, 256, 24,  64,  64)
        self.im6 = InceptionModule(512, 112, 144, 288, 32,  64,  64)
        self.im7 = InceptionModule(528, 256, 160, 320, 32, 128, 128)
        self.im8 = InceptionModule(832, 256, 160, 320, 32, 128, 128)
        self.im9 = InceptionModule(832, 384, 192, 384, 48, 128, 128)
        self.average_pool = nn.AvgPool2d(7, stride=1)
        self.fc = nn.Linear(4096, 1000)
    def forward(self, ip):
        op = self.stem(ip)
        out = self.im1(op)
        out = self.im2(op)
        op = self.maxpool(op)
        op = self.a4(op)
        op = self.b4(op)
        op = self.c4(op)
        op = self.d4(op)
        op = self.e4(op)
        op = self.max_pool(op)
        op = self.a5(op)
        op = self.b5(op)
        op = self.avgerage_pool(op)
        op = op.view(op.size(0), -1)
        op = self.fc(op)
        return op
```

Besides instantiating our own model, we can always load a pretrained GoogLeNet with just two lines of code:

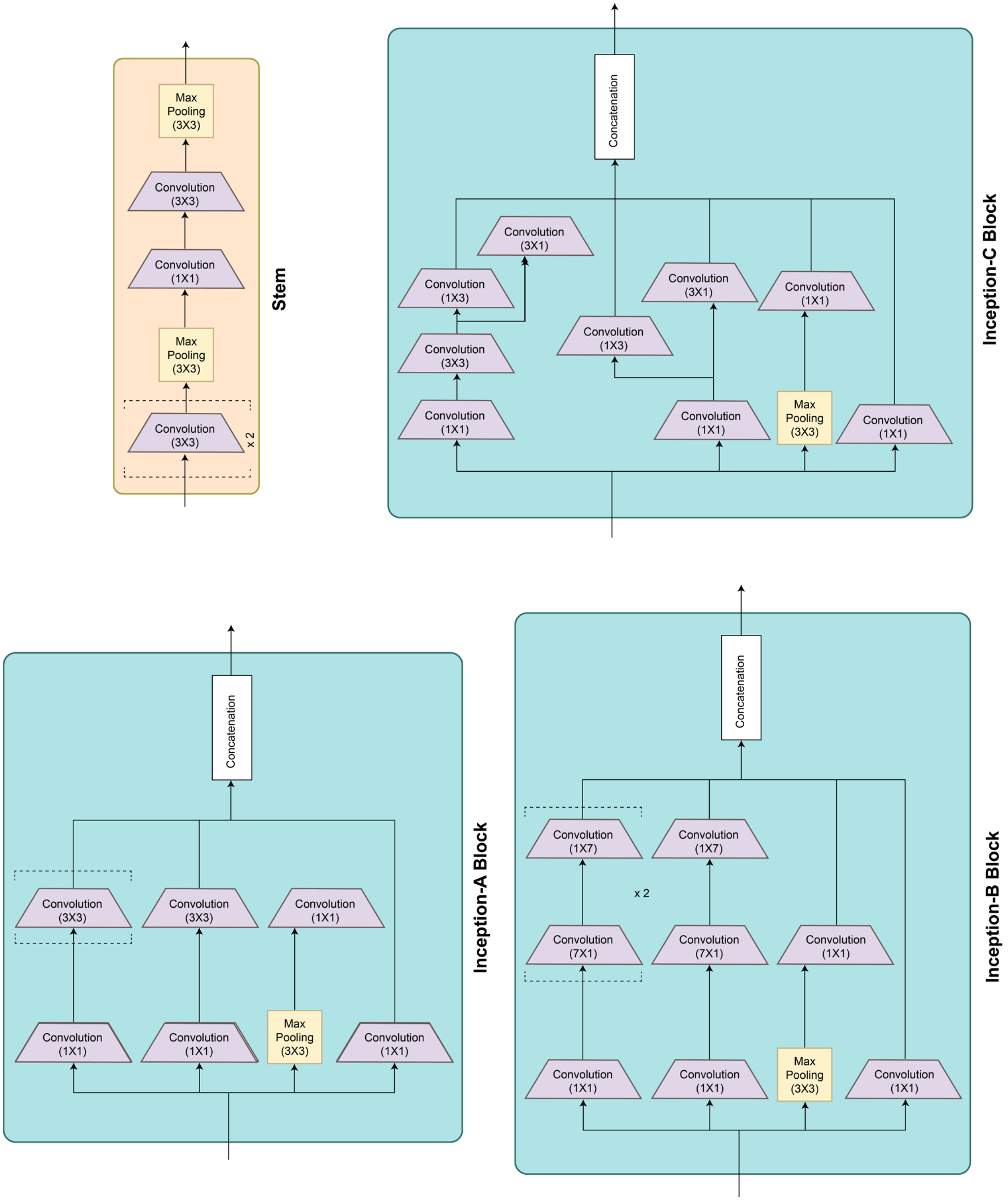
```
import torchvision.models as models
model = models.googlenet(pretrained=True)
```

Finally, as mentioned earlier, a number of versions of the Inception model were developed later. One of the eminent ones was Inception v3, which we will briefly discuss next.

Inception v3

This successor of Inception v1 has a total of 24 million parameters as compared to 5 million in v1. Besides the addition of several more layers, this model introduced different kinds of inception modules, which are stacked sequentially.

Figure 2.15 shows the different inception modules and the full model architecture:



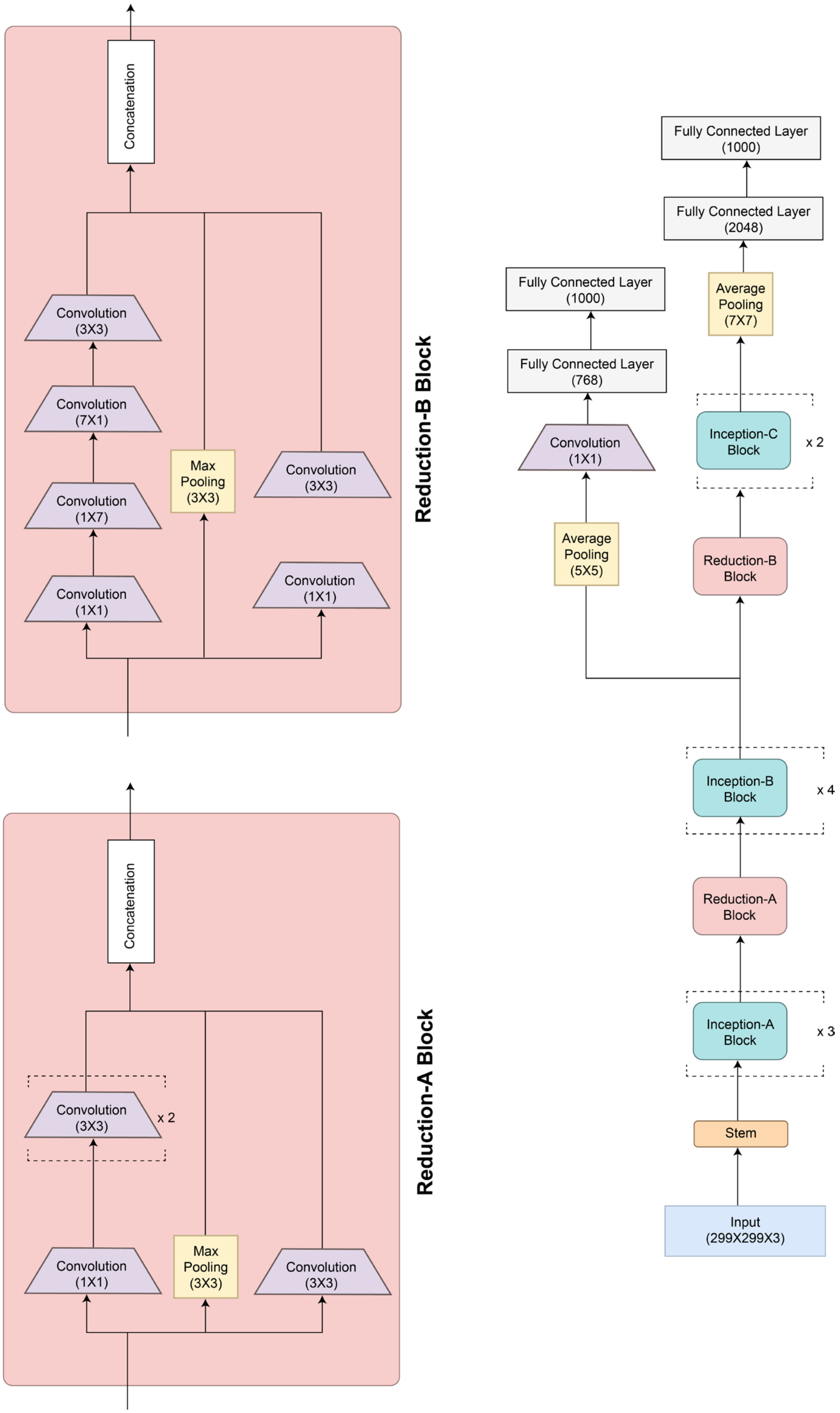


Figure 2.15: Inception v3 architecture

It can be seen from the architecture that this model is an architectural extension of the Inception v1 model. Once again, besides building the model manually, we can use the pretrained model from PyTorch's repository as follows:

```
import torchvision.models as models
model = models.inception_v3(pretrained=True)
```

In the next section, we will go through the classes of CNN models that have effectively combatted the vanishing gradient problem in very deep CNNs – **ResNet** and **DenseNet**. We will learn about the novel techniques of skip connections and dense connections and use PyTorch to code the fundamental modules behind these advanced architectures.

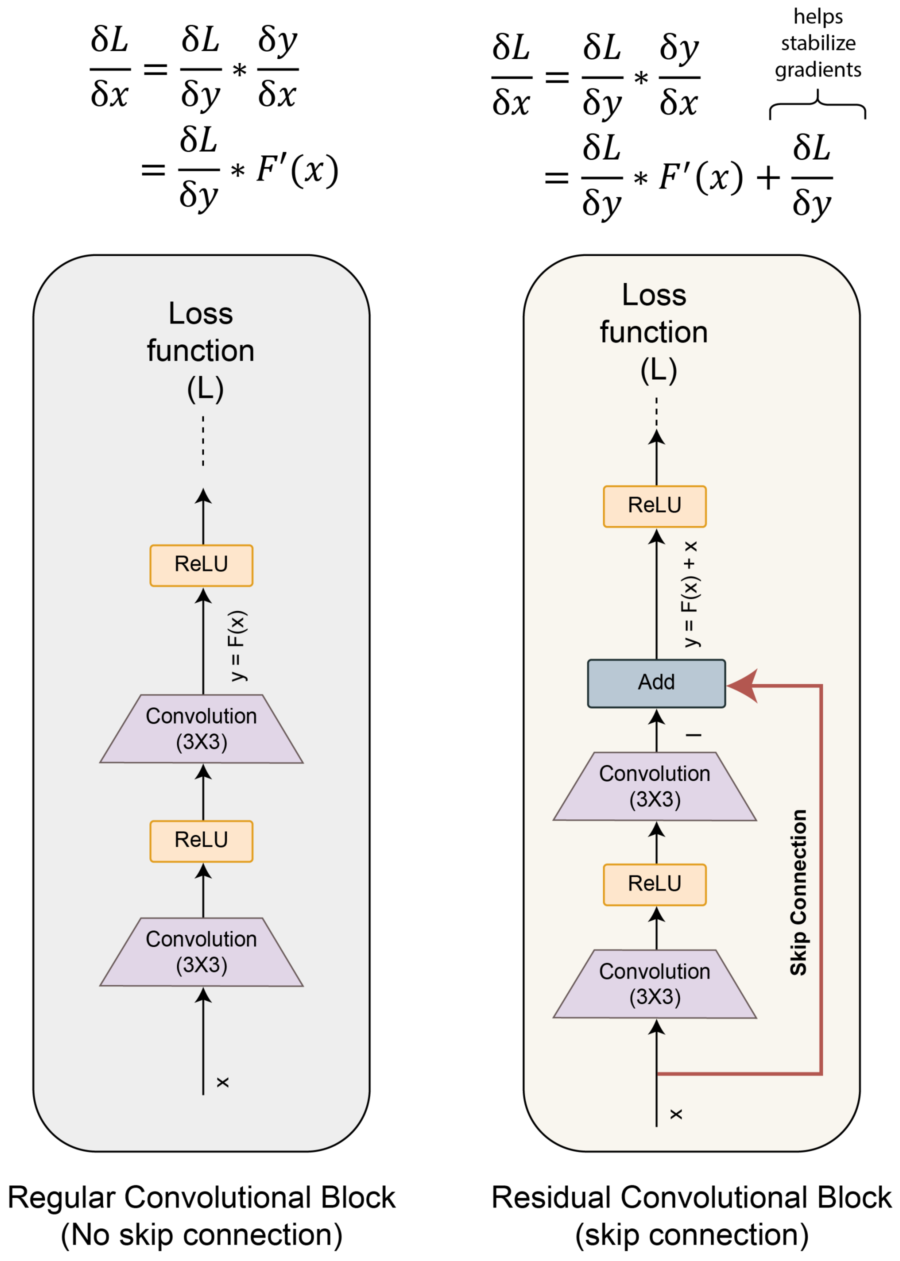
Discussing ResNet and DenseNet architectures

In the previous section, we explored Inception models, which have a reduced number of model parameters as the number of layers increased, thanks to the 1x1 convolutions and global average pooling. Furthermore, auxiliary classifiers were used to combat the vanishing gradient problem. In this section, we will discuss ResNet and DenseNet models.

ResNet

ResNet introduced the concept of **skip connections**. This simple yet effective trick overcomes the problem of both parameter overflow and vanishing gradients. The idea, as shown in the following diagram, is quite simple. The input is first passed through a non-linear transformation (convolutions followed by non-linear activations) and then the output of this transformation (referred to as the residual) is added to the original input.

Each block of such computation is called a **residual block**, hence the name of the model – **residual network** or **ResNet**:



$$\frac{\delta L}{\delta x} = \frac{\delta L}{\delta y} * \frac{\delta y}{\delta x} = \frac{\delta L}{\delta y} * F'(x)$$

$$\frac{\delta L}{\delta x} = \frac{\delta L}{\delta y} * \frac{\delta y}{\delta x} = \frac{\delta L}{\delta y} * F'(x) + \overbrace{\frac{\delta L}{\delta y}}^{\text{helps stabilize gradients}}$$

Figure 2.16: Skip connections

Using these skip (or shortcut) connections, the number of parameters is limited to 26 million parameters for a total of 50 layers (ResNet-50). Due to the limited number of parameters, ResNet has been able to generalize well without overfitting even when the number of layers is increased to 152 (ResNet-152). The following diagram shows the ResNet-50 architecture:

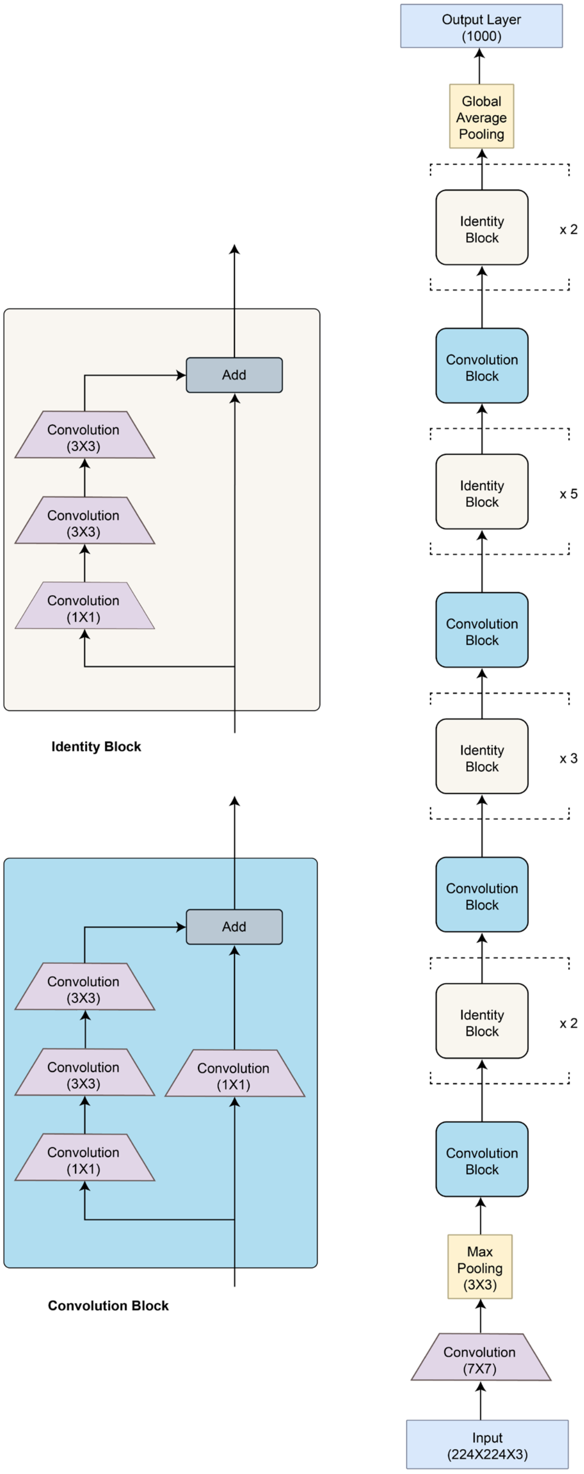


Figure 2.17: ResNet architecture

There are two kinds of residual blocks – **convolutional** and **identity**, both having skip connections. For the convolutional block, there is an added 1x1 convolutional layer, which further helps to reduce dimensionality. A residual block for ResNet can be implemented in PyTorch as shown here:

```
class BasicBlock(nn.Module):
    multiplier=1
    def __init__(self, input_num_planes, num_planes, strd=1):
        super(BasicBlock, self).__init__()
        self.conv_layer1 = nn.Conv2d(in_channels=input_num_planes,
                                     out_channels=num_planes,
                                     kernel_size=3,
                                     stride=stride, padding=1,
                                     bias=False)
        self.batch_norm1 = nn.BatchNorm2d(num_planes)
        self.conv_layer2 = nn.Conv2d(in_channels=num_planes,
                                     out_channels=num_planes,
                                     kernel_size=3, stride=1,
                                     padding=1, bias=False)
        self.batch_norm2 = nn.BatchNorm2d(num_planes)
        self.res_connnection = nn.Sequential()
        if strd > 1 or input_num_planes != self.multiplier*num_planes:
            self.res_connnection = nn.Sequential(
                nn.Conv2d(in_channels=input_num_planes,
                          out_channels=self.multiplier*num_planes,
                          kernel_size=1, stride=strd, bias=False),
                nn.BatchNorm2d(self.multiplier*num_planes))
    def forward(self, inp):
        op = F.relu(self.batch_norm1(self.conv_layer1(inp)))
        op = self.batch_norm2(self.conv_layer2(op))
        op += self.res_connection(inp)
        op = F.relu(op)
        return op
```

To get started quickly with ResNet, we can always use the pretrained ResNet model from PyTorch's repository:

```
import torchvision.models as models
model = models.resnet50(pretrained=True)
```

ResNet uses the identity function (by directly connecting input to output) to preserve the gradient during backpropagation (as the gradient will be 1). Yet, for extremely deep networks, this principle might not be sufficient to preserve strong gradients from the output layer back to the input layer.

The CNN model we will discuss next is designed to ensure a strong gradient flow, as well as a further reduction in the number of required parameters.

DenseNet

The skip connections of ResNet connected the input of a residual block directly to its output. However, the inter-residual-blocks connection is still sequential; that is, residual block number 3 has a direct connection with block 2 but no direct connection with block 1.

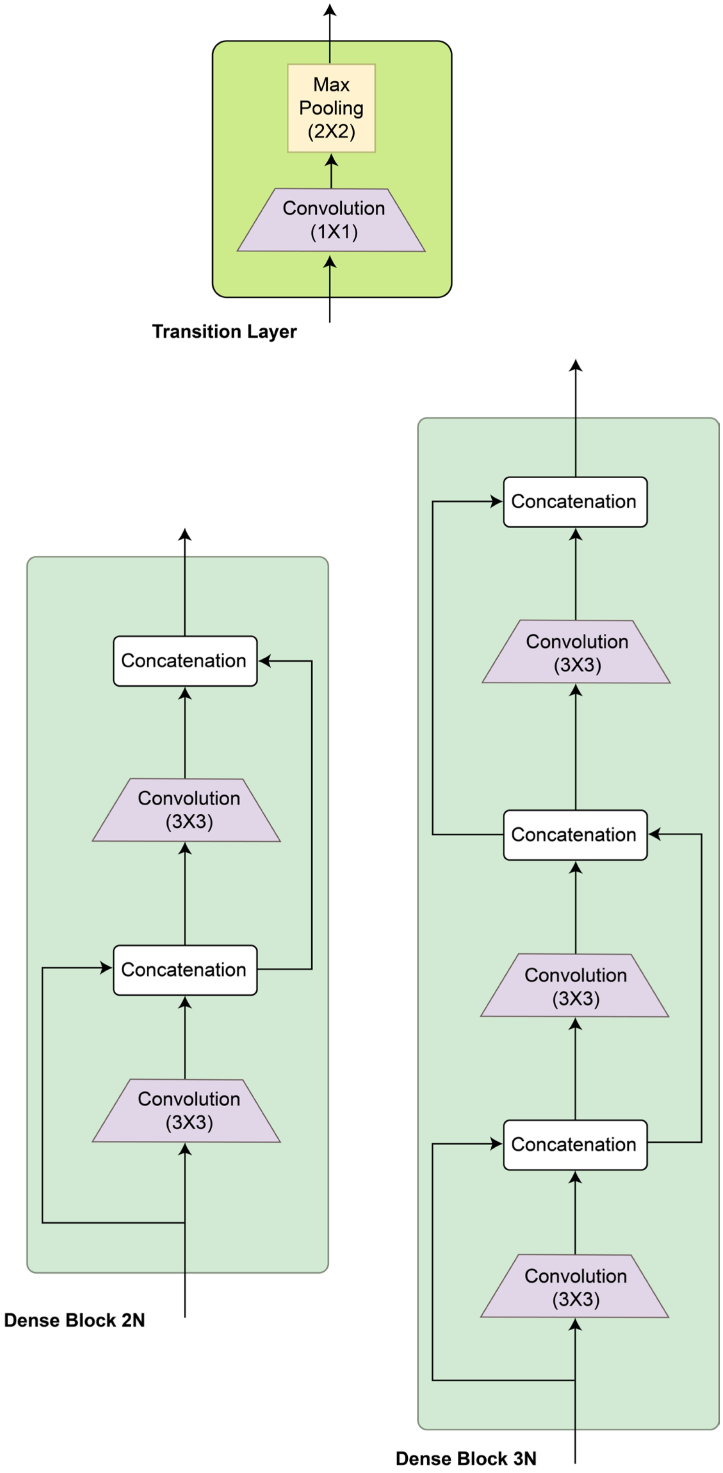
DenseNet, or dense networks, introduced the idea of connecting every convolutional layer with every other layer within what is called a **dense block**. And every dense block is connected to every other dense block in the overall DenseNet. A dense block is simply a module of two 3x3 densely connected convolutional layers.

These dense connections ensure that every layer is receiving information from all of the preceding layers of the network. This ensures that there is a strong gradient flow from the last layer down to the very first layer. Counterintuitively, the number of parameters of such a network setting will also be low. As every layer receives the feature maps from all the previous layers, the required number of channels (depth) can be fewer. In the earlier models, the increasing depth represented the accumulation of information from earlier layers, but we don't need that anymore, thanks to the dense connections everywhere in the network.

One key difference between ResNet and DenseNet is also that, in ResNet, the input was added to the output using skip connections. But in the case of DenseNet, the preceding layers' outputs are concatenated with the current layer's output. And the concatenation happens in the depth dimension.

This might raise a question about the exploding size of outputs as we proceed further in the network. To combat this compounding effect, a special type of block called the **transition block** is devised for this network. Composed of a 1x1 convolutional layer followed by a 2x2 pooling layer, this block standardizes or resets the size of the depth dimension so that the output of this block can then be fed to the subsequent dense block(s).

The following diagram shows the DenseNet architecture:



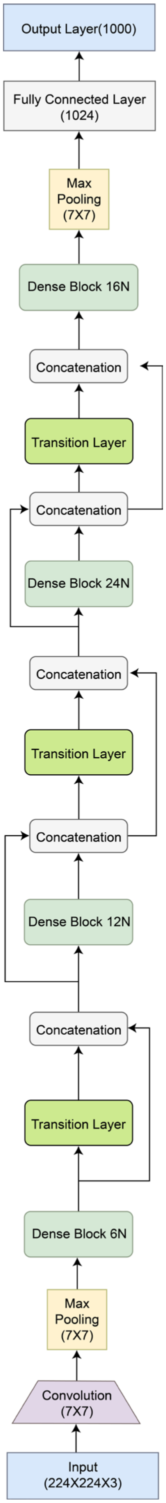


Figure 2.18: DenseNet architecture

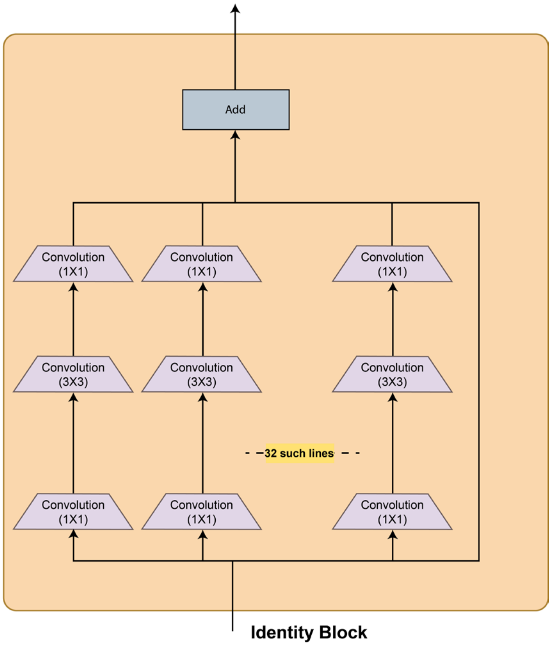
As mentioned earlier, there are two types of blocks involved – the **dense block** and the **transition block**. These blocks can be written as classes in PyTorch in a few lines of code, as shown here:

```
class DenseBlock(nn.Module):
    def __init__(self, input_num_planes, rate_inc):
        super(DenseBlock, self).__init__()
        self.batch_norm1 = nn.BatchNorm2d(input_num_planes)
        self.conv_layer1 = nn.Conv2d(in_channels=input_num_planes,
                                     out_channels=4*rate_inc,
                                     kernel_size=1, bias=False)
        self.batch_norm2 = nn.BatchNorm2d(4*rate_inc)
        self.conv_layer2 = nn.Conv2d(in_channels=4*rate_inc,
                                     out_channels=rate_inc,
                                     kernel_size=3, padding=1,
                                     bias=False)
    def forward(self, inp):
        op = self.conv_layer1(F.relu(self.batch_norm1(inp)))
        op = self.conv_layer2(F.relu(self.batch_norm2(op)))
        op = torch.cat([op,inp], 1)
        return op
class TransBlock(nn.Module):
    def __init__(self, input_num_planes, output_num_planes):
        super(TransBlock, self).__init__()
        self.batch_norm = nn.BatchNorm2d(input_num_planes)
        self.conv_layer = nn.Conv2d(in_channels=input_num_planes,
                                    out_channels=output_num_planes,
                                    kernel_size=1, bias=False)
    def forward(self, inp):
        op = self.conv_layer(F.relu(self.batch_norm(inp)))
        op = F.avg_pool2d(op, 2)
        return op
```

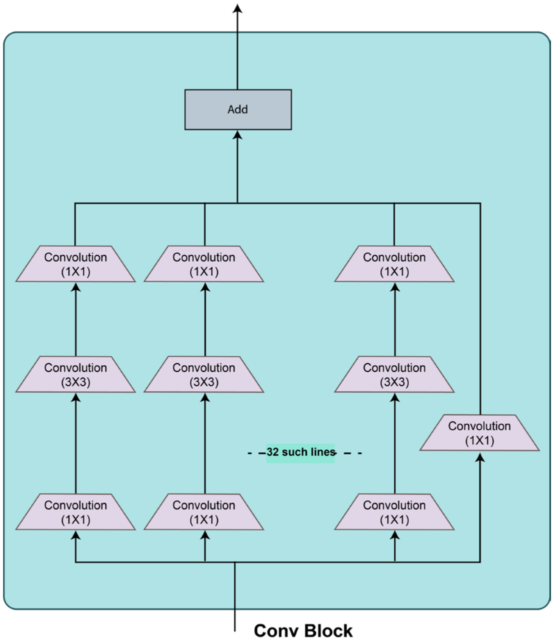
These blocks are then stacked densely to form the overall DenseNet architecture. DenseNet, like ResNet, comes in variants, such as **DenseNet121**, **DenseNet161**, **DenseNet169**, and **DenseNet201**, where the numbers represent the total number of layers. Such large numbers of layers are obtained by the repeated stacking of the dense and transition blocks plus a fixed 7x7 convolutional layer at the input end and a fixed fully connected layer at the output end. PyTorch provides pretrained models for all of these variants:

```
import torchvision.models as models
densenet121 = models.densenet121(pretrained=True)
densenet161 = models.densenet161(pretrained=True)
densenet169 = models.densenet169(pretrained=True)
densenet201 = models.densenet201(pretrained=True)
```

DenseNet outperforms all the models discussed so far on the ImageNet dataset. Various hybrid models have been developed by mixing and matching the ideas presented in the previous sections. The Inception-ResNet and ResNeXt models are examples of such hybrid networks. The following diagram shows the ResNeXt architecture:



Identity Block



Conv Block

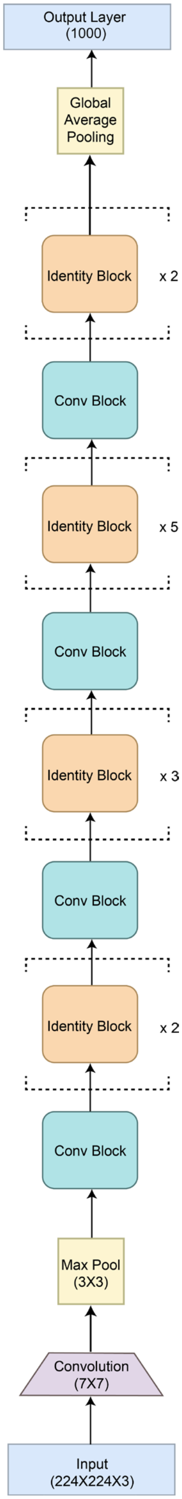


Figure 2.19 – ResNeXt architecture

As you can see, it looks like a wider variant of a *ResNet* + *Inception* hybrid because there is a large number of parallel convolutional branches in the residual blocks – and the idea of parallelism is derived from the inception network.

In the next and last section of this chapter, we are going to look at the one of the best-performing CNN architectures till date – EfficientNets. We will also discuss the future of CNN architectural development while touching upon the use of CNN architectures for tasks beyond image classification.

Understanding EfficientNets and the future of CNN architectures

So far in our exploration from LeNet to DenseNet, we have noticed an underlying theme in the advancement of CNN architectures. That theme is the expansion or scaling of the CNN model through one of the following:

- An increase in the number of layers
- An increase in the number of feature maps or channels in a convolutional layer
- An increase in the spatial dimension going from 32x32 pixel images in LeNet to 224x224 pixel images in AlexNet and so on

These three different aspects on which scaling can be performed are identified as *depth*, *width*, and *resolution*, respectively. Instead of manually scaling these attributes, which often leads to suboptimal results, **EfficientNets** use neural architecture search to calculate the optimal scaling factors for each of them.

Scaling up depth is deemed important because the deeper the network, the more complex the model, and hence it can learn highly complex features. However, there is a trade-off because, with increasing depth, the vanishing gradient problem escalates along with the general problem of overfitting.

Similarly, scaling up width should theoretically help, as with a greater number of channels, the network should learn more fine-grained features. However, for extremely wide models, the accuracy tends to saturate quickly.

Finally, higher-resolution images, in theory, should work better as they have more fine-grained information. Empirically, however, the increase in resolution does not yield a linearly equivalent increase in the model performance. All of this is to say that there are trade-offs to be made while deciding the scaling factors and hence, neural architecture search helps in finding the optimal scaling factors.

EfficientNet proposes finding the architecture that has the right balance between depth, width, and resolution, and all three of these aspects are scaled together using a global scaling factor. The EfficientNet architecture is built in two steps. First, a basic architecture (called the **base network**) is devised by fixing the scaling factor to `1`. At this stage, the relative importance of depth, width, and resolution is decided for the given task and dataset. The base network obtained is pretty similar to a well-known CNN architecture – **MnasNet**, short for **Mobile Neural Architecture Search Network**.

PyTorch offers the pretrained MnasNet model, which can be loaded as shown here:

```
import torchvision.models as models
model = models.mnasnet1_0()
```

Once the base network is obtained in the first step, the optimal global scaling factor is then computed with the aim of maximizing the accuracy of the model and minimizing the number of computations (or flops). The base network is called **EfficientNet B0** and the subsequent networks derived for different optimal scaling factors are called **EfficientNet B1-B7**. PyTorch provides pretrained models for all of these variants:

```
import torchvision.models as models
efficientnet_b0 = models.efficientnet_b0(pretrained=True)
efficientnet_b1 = models.efficientnet_b1(pretrained=True)
...
efficientnet_b7 = models.efficientnet_b7(pretrained=True)
```

As we go forward, efficient scaling of CNN architecture is going to be a prominent direction of research along with the development of more sophisticated modules inspired by the inception, residual, and dense modules. Another aspect of CNN architecture development is minimizing the model size while retaining performance. **MobileNets** [9] are a prime example and there is a lot of ongoing research on this front.

Besides the top-down approach of looking at architectural modifications of a pre-existing model, there will be continued efforts to adopt the bottom-up view of fundamentally rethinking the units of CNNs such as the convolutional kernels, pooling mechanism, more effective ways of flattening, and so on. One concrete example of this would be **CapsuleNet** [10], which revamped the convolutional units to cater to the third dimension (depth) in images.

CNNs are a huge topic of study in themselves. In this chapter, we have touched upon the architectural development of CNNs, mostly in the context of image classification. However, these same architectures are used across a wide variety of applications. One well-known example is the use of ResNets for object detection and segmentation in the form of **RCNNs** [11].

Some of the improved variants of RCNNs are **Faster R-CNN**, **Mask-RCNN**, and **Keypoint-RCNN**. PyTorch provides pretrained models for all three variants:

```
faster_rcnn = models.detection.fasterrcnn_resnet50_fpn()
mask_rcnn = models.detection.maskrcnn_resnet50_fpn()
keypoint_rcnn = models.detection.keypointrcnn_resnet50_fpn()
```

PyTorch also provides pretrained models for ResNets that are applied to video-related tasks such as video classification. Two such ResNet-based models used for video classification are **ResNet3D** and **ResNet Mixed Convolution**:

```
resnet_3d = models.video.r3d_18()
resnet_mixed_conv = models.video.mc3_18()
```

While we do not extensively cover these different applications and corresponding CNN models in this chapter, we encourage you to read more on them. PyTorch's website can be a good starting point [12].

Summary

This chapter has been all about CNN architectures. In the next chapter, we will touch upon another type of neural network model - recurrent neural network. We will build an end-to-end deep learning application - an image caption generator - by combining CNNs with LSTMs (a kind of recurrent neural network).

References

1. ImageNet dataset: `https://image-net.org/`
2. CIFAR-10 dataset: `https://www.cs.toronto.edu/~kriz/cifar.html`
3. PyTorch vision models: `https://pytorch.org/vision/stable/models.html`
4. Mastering PyTorch GitHub notebook link to fine-tune AlexNet: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter02/transfer_learning_alexnet.ipynb`
5. Hymenoptera dataset (Kaggle link): `https://www.kaggle.com/datasets/ajayrana/hymenoptera-data`
6. Hymenoptera Genome Database: `https://hymenoptera.elsiklab.missouri.edu/`
7. Mastering PyTorch GitHub notebook link to run VGG model inference: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter02/vgg13_pretrained_run_inference.ipynb`
8. ImageNet class IDs to labels: `https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a`
9. PyTorch MobileNetV2: `https://pytorch.org/hub/pytorch_vision_mobilenet_v2/`
10. Capsule neural network: `https://en.wikipedia.org/wiki/Capsule_neural_network`
11. Region-based convolutional neural networks: `https://en.wikipedia.org/wiki/Region_Based_Convolutional_Neural_Networks`
12. Object detection, instance segmentation, and person keypoint detection TorchVision models: `https://pytorch.org/vision/stable/models.html#object-detection-instance-segmentation-and-person-keypoint-detection`

Learn more on Discord

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

`https://packt.link/mastorch`

3

Combining CNNs and LSTMs

Convolutional Neural Networks (**CNNs**) are a type of deep learning model used to solve machine learning problems related to images, video, speech, and audio, such as image classification, object detection, segmentation, speech recognition, audio classification, and more. This is because CNNs use a special type of layer called **convolutional layers**, which have shared learnable parameters. The weight or parameter sharing works because the patterns to be learned in an image (such as edges or contours) are assumed to be independent of the location of the pixels in the image. Just as CNNs are applied to images, **Long Short-Term Memory** (**LSTM**) networks – which are a type of **Recurrent Neural Network** (**RNN**) – prove to be extremely effective at solving machine learning problems related to **sequential data**. An example of sequential data could be text. For example, in a sentence, each word is dependent on the previous word(s). LSTM models are meant to model such sequential dependencies.

These two different types of networks – CNNs and LSTMs – can be combined to form a multimodal model that takes in images or video and outputs text. One well-known application of such a hybrid model is image captioning, where the model takes in an image and outputs a plausible textual description of the image. Since 2010, machine learning has been used to perform the task of image captioning [1].

However, neural networks were first successfully used for this task in around 2014/2015 [2]. Ever since, image captioning has been actively researched. With significant improvements each year, this deep learning application can become useful for real-world applications such as generating alt text on websites to make them more accessible for the visually impaired.

This chapter first discusses the architecture of such a multimodal model, along with the related implementational details in PyTorch, and at the end of the chapter, we will build an image captioning system from scratch using PyTorch. This chapter covers the following topics:

- Building a neural network with CNNs and LSTMs
- Building an image caption generator using PyTorch

All the code files for this chapter can be found at `https://github.com/arj7192/MasteringPyTorchV2/tree/main/Chapter02`

Let us now discuss the combined CNN and LSTM architecture first.

Building a neural network with CNNs and LSTMs

A CNN-LSTM network architecture consists of a convolutional layer(s) for extracting features from the input data (image), followed by an LSTM layer(s) to perform sequential predictions. This kind of model is deep both spatially – thanks to the CNN component – as well as temporally – thanks to the LSTM network. The convolutional part of the model is often used as an **encoder** that takes in an input image and outputs high-dimensional features or embeddings.

In practice, the CNN used for these hybrid networks is often pretrained on, say, an image classification task. The last hidden layer of the pretrained CNN model is then used as an input to the LSTM component, which is used as a **decoder** to generate text.

When we are dealing with textual data, we need to transform the words and other symbols (punctuation, identifiers, and more) – together referred to as **tokens** – into numbers. We do so by representing each token in the text with a unique corresponding number. In the following sub-section, we will demonstrate an example of text encoding.

Text encoding demo

Let's assume we're building a machine learning model with textual data; say, for example, that our text is as follows:

```
<start> PyTorch is a deep learning library. <end>
```

Then, we would map each of these words/tokens to numbers, as follows:

```
<start> : 0
PyTorch : 1
is : 2
a : 3
deep : 4
learning : 5
library : 6
. : 7
<end> : 8
```

Once we have the mapping, we can represent this sentence numerically as a list of numbers:

```
<start> PyTorch is a deep learning library. <end> -> [0, 1, 2, 3, 4, 5, 6, 7,
8]
```

Also, for example, `<start> PyTorch is deep. <end>` would be encoded as `-> [0, 1, 2, 4, 7, 8]` and so on. This mapping, in general, is referred to as a **vocabulary**, and building a vocabulary is a crucial part of most text-related machine learning problems.

The LSTM model, which acts as the decoder, takes in a CNN embedding as input at time-step `t=0`. Then, each LSTM cell makes a token prediction at each time-step, which is fed as the input to the next LSTM cell. The overall architecture thus generated can be visualized as shown in *Figure 3.1*:

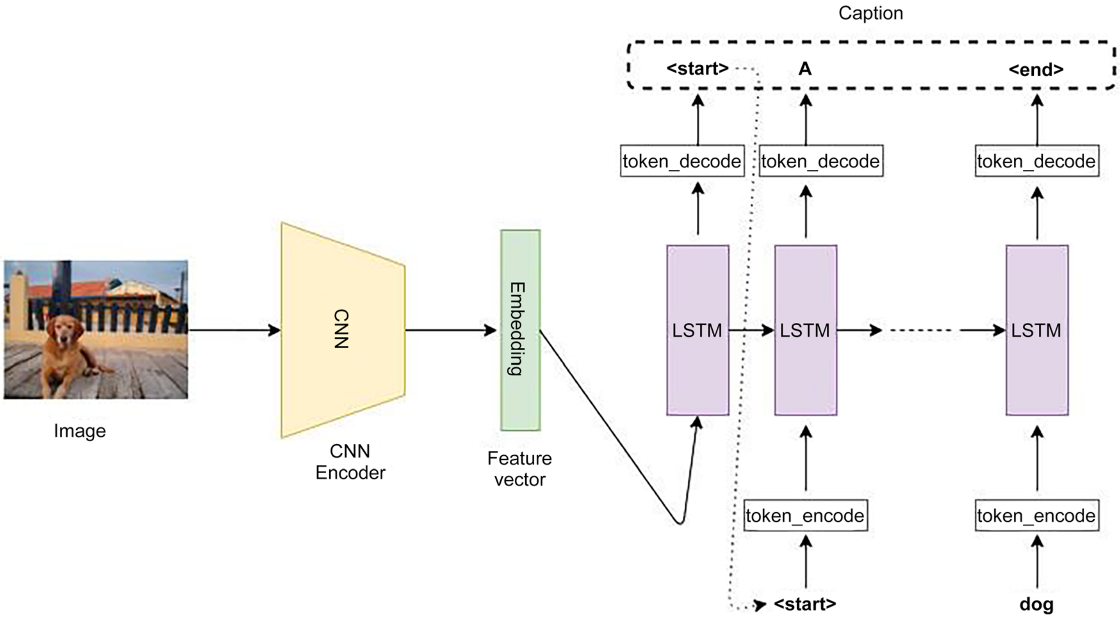


Figure 3.1: An example of CNN-LSTM architecture

The demonstrated architecture is suitable for an image captioning task. If instead of just having a single image we had a sequence of images (say, in a video) as the input to the CNN layer, then we would include the CNN embedding as the LSTM cell input at each time-step, not just at `t=0`. This kind of architecture would be useful for applications such as activity recognition or video description.

In the next section, we will implement an image captioning system in PyTorch that includes building a hybrid model architecture as well as data loading, preprocessing, model training, and model evaluation pipelines.

Building an image caption generator using PyTorch

For this exercise, we will be using the **Common Objects in Context** (**COCO**) dataset [3], which is a large-scale object detection, segmentation, and captioning dataset.

This dataset consists of over 200,000 labeled images with five captions for each image. The COCO dataset emerged in 2014 and has helped significantly in the advancement of object recognition-related computer vision tasks. It stands as one of the most commonly used datasets for benchmarking tasks such as object detection, object segmentation, instance segmentation, and image captioning.

In this exercise, we will use PyTorch to train a CNN-LSTM model on this dataset and use the trained model to generate captions for unseen samples. Before we do that, though, there are a few prerequisites that we need to take care of.

We will be referring to only the important snippets of code for illustration purposes. The full exercise code can be found in our GitHub repository [4].

Downloading the image captioning datasets

Before we begin building the image captioning system, we need to download the required datasets. If you do not have the datasets downloaded, then run the following script with the help of Jupyter Notebook. This should help with downloading the datasets locally.

We are using a slightly older version of the dataset as it is slightly smaller in size, enabling us to get the results faster.

The training and validation datasets are 13 GB and 6 GB in size, respectively. Downloading and extracting the dataset files, as well as cleaning and processing them, might take a while. A good idea is to execute these steps as follows and let them finish overnight:

```
# download images and annotations to the data directory
!wget http://images.cocodataset.org/annotations/annotations_trainval2014.zip -P
./data_dir/ -P ./data_dir/
!wget http://images.cocodataset.org/zips/train2014.zip -P ./data_dir/
!wget http://images.cocodataset.org/zips/val2014.zip -P ./data_dir/
# extract zipped images and annotations and remove the zip files
!unzip ./data_dir/annotations_trainval2014.zip -d ./data_dir/
!rm ./data_dir/annotations_trainval2014.zip
!unzip ./data_dir/train2014.zip -d ./data_dir/
!rm ./data_dir/train2014.zip
!unzip ./data_dir/val2014.zip -d ./data_dir/
!rm ./data_dir/val2014.zip
```

You should see the following output:

```
--2022-11-30 11:15:15--  http://images.cocodataset.org/annotations/annotations_
trainval2014.zip
Resolving images.cocodataset.org (images.cocodataset.org)... 52.217.92.164,
52.216.240.252, 52.217.107.92, …
Connecting to images.cocodataset.org (images.cocodataset.
org)|52.217.92.164|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 252872794 (241M) [application/zip]
Saving to: './data_dir/annotations_trainval2014.zip'
```

```
annotations_trainva 100%[===================>] 241.16M  6.50MB/s    in 34s
...
extracting: ./data_dir/val2014/COCO_val2014_000000551804.jpg
extracting: ./data_dir/val2014/COCO_val2014_000000045516.jpg
extracting: ./data_dir/val2014/COCO_val2014_000000347233.jpg
extracting: ./data_dir/val2014/COCO_val2014_000000154202.jpg
extracting: ./data_dir/val2014/COCO_val2014_000000038210.jpg
```

This step basically creates a data folder (`./data_dir`), downloads the zipped images and annotation files, and extracts them inside the data folder.

Preprocessing caption (text) data

The downloaded image captioning datasets consist of both text (captions) and images. In this section, we will preprocess the text data to make it usable for our CNN-LSTM model. The exercise is laid out as a sequence of steps. The first three steps are focused on processing the text data:

1. For this exercise, we will need to import a few dependencies. Some of the crucial modules we will import for this chapter are as follows:

   ```
   import nltk
   from pycocotools.coco import COCO
   import torch.utils.data as data
   import torchvision.models as models
   import torchvision.transforms as transforms
   from torch.nn.utils.rnn import pack_padded_sequence
   ```

   `nltk` is the natural language toolkit, which will be helpful in building our vocabulary, while `pycocotools` is a helper tool to work with the COCO dataset. The various Torch modules we have imported here have already been discussed in the previous chapter, except the last one – that is, `pack_padded_sequence`. This function will be useful to transform sentences with variable lengths (number of words) into fixed-length sentences by applying padding.

   Besides importing the `nltk` library, we will also need to download its `punkt` tokenizer model, as follows:

   ```
   nltk.download('punkt')
   ```

   This will enable us to tokenize given text into its constituent words.

2. Next, we build the vocabulary – that is, a dictionary that can convert actual textual tokens (such as words) into numeric tokens. This step is essential for most text-related tasks:

   ```
   def build_vocabulary(json, threshold):
       """Build a vocab wrapper."""
       coco = COCO(json)
   ```

```
    counter = Counter()
    ids = coco.anns.keys()
    for i, id in enumerate(ids):
        caption = str(coco.anns[id]['caption'])
        tokens = \
            nltk.tokenize.word_tokenize(caption.lower())
        counter.update(tokens)
        if (i+1) % 1000 == 0:
            print("[{}/{}] Tokenized the captions."
                  .format(i+1, len(ids)))
```

First, inside the vocabulary builder function, JSON text annotations are loaded, and individual words in the annotation/caption are tokenized or converted into numbers and stored in a counter.

Then, tokens with fewer than a certain number of occurrences are discarded, and the remaining tokens are added to a vocabulary object beside some wildcard tokens – `start` (of the sentence), `end`, `unknown_word`, and padding tokens, as follows:

```
    # If word freq < 'thres', then word is discarded.
    tokens = [token for token,
              cnt in counter.items() if cnt >= threshold]
    # Create vocab wrapper + add special tokens.
    vocab = Vocab()
    vocab.add_token('<pad>')
    vocab.add_token('<start>')
    vocab.add_token('<end>')
    vocab.add_token('<unk>')
    # Add words to vocab.
    for i, token in enumerate(tokens):
        vocab.add_token(token)
    return vocab
```

Finally, using the vocabulary builder function, a vocabulary object, `vocab`, is created and saved locally for further reuse, as shown in the following code:

```
vocab = build_vocabulary(
    json='data_dir/annotations/captions_train2014.json', threshold=4)
vocab_path = './data_dir/vocabulary.pkl'
with open(vocab_path, 'wb') as f:
    pickle.dump(vocab, f)
print("Total vocabulary size: {}".format(len(vocab)))
print("Saved the vocabulary wrapper to '{}'"
      .format(vocab_path))
```

The output for this is as follows:

```
loading annotations into memory...
Done (t=0.67s)
creating index...
index created!
[1000/414113] Tokenized the captions.
[2000/414113] Tokenized the captions.
[3000/414113] Tokenized the captions.
...
[412000/414113] Tokenized the captions.
[413000/414113] Tokenized the captions.
[414000/414113] Tokenized the captions.
Total vocabulary size: 9948
Saved the vocabulary wrapper to './data_dir/vocabulary.pkl'
```

Once we have built the vocabulary, we can deal with the textual data by transforming it into numbers at runtime.

Preprocessing image data

After downloading the data and building the vocabulary for the text captions, we need to perform some preprocessing for the image data:

1. Because the images in the dataset can come in various sizes or shapes, we need to reshape all the images to a fixed shape so that they can be inputted to the first layer of our CNN model, as follows:

```python
def reshape_images(image_path, output_path, shape):
    images = os.listdir(image_path)
    num_im = len(images)
    for i, im in enumerate(images):
        with open(os.path.join(image_path, im), 'r+b') as f:
            with Image.open(f) as image:
                image = reshape_image(image, shape)
                image.save(os.path.join(output_path, im),
                           image.format)
        if (i+1) % 100 == 0:
            print ("[{}/{}] Resized the images and saved into '{}'."
                   .format(i+1, num_im, output_path))
reshape_images(image_path, output_path, image_shape)
```

The output for this will be as follows:

```
[100/82783] Resized the images and saved into './data_dir/resized_
images/'.
[200/82783] Resized the images and saved into './data_dir/resized_
images/'.
[300/82783] Resized the images and saved into './data_dir/resized_
images/'.
...
[82500/82783] Resized the images and saved into './data_dir/resized_
images/'.
[82600/82783] Resized the images and saved into './data_dir/resized_
images/'.
[82700/82783] Resized the images and saved into './data_dir/resized_
images/'.
```

We have reshaped all the images to 256 x 256 pixels, which makes them compatible with our CNN model architecture.

Defining the image captioning data loader

We have already downloaded and preprocessed the image captioning data. Now it is time to cast this data as a PyTorch dataset object. This dataset object can subsequently be used to define a PyTorch data loader object, which we will use in our training loop to fetch batches of data as follows:

1. Now, we will implement our own custom `Dataset` module and a custom data loader:

```
class CustomCocoDataset(data.Dataset):
    """COCO Dataset compatible with
       torch.utils.data.DataLoader."""
    def __init__(self, data_path, coco_json_path,
                 vocabulary, transform=None):
        """Set path for images, texts and vocab wrapper.

        Args:
            data_path: image directory.
            coco_json_path: coco annotation file path.
            vocabulary: vocabulary wrapper.
            transform: image transformer.
        """
        ...
    def __getitem__(self, idx):
        """Returns one data sample (X, y)."""
        ...
        return image, ground_truth
    def __len__(self):
        return len(self.indices)
```

First, in order to define our custom PyTorch `Dataset` object, we have defined our own `__init__`, `__get_item__`, and `__len__` methods for instantiation, fetching items, and returning the size of the dataset, respectively.

2. Next, we define `collate_function`, which returns mini-batches of data in the form of `X`, `y`, as follows:

```
def collate_function(data_batch):
    """Creates mini-batches of data
    We build custom collate function
    rather than using standard collate function,
    because padding is not supported in
    the standard version.
    Args:
        data: list of (image, caption)tuples.
            - image: tensor of shape (3, 256, 256).
            - caption: tensor of shape (:); variable length.
    Returns:
        images: tensor of size (batch_size, 3, 256, 256).
        targets: tensor of size (batch_size, padded_length).
        lengths: list.
    """
    ...
    return imgs, tgts, cap_lens
```

Usually, we would not need to write our own `collate` function, but we do so to deal with variable-length sentences so that when the length of a sentence (say, `k`) is less than the fixed length, `n`, then we need to pad the `n-k` tokens with padding tokens using the `pack_padded_sequence` function.

3. Finally, we will implement the `get_loader` function, which returns a custom data loader for the COCO dataset in the following code:

```
def get_loader(data_path, coco_json_path, vocabulary, transform, batch_
size, shuffle):
    # COCO dataset
    coco_dataset = CustomCocoDataset(data_path=data_path,
                       coco_json_path=coco_json_path,
                       vocabulary=vocabulary,
                       transform=transform)
    custom_data_loader = \
        torch.utils.data.DataLoader(dataset=coco_dataset,
                                batch_size=batch_size,
                                shuffle=shuffle,
                                collate_fn=collate_function)
    return custom_data_loader
```

During the training loop, this function will be extremely useful and efficient in fetching mini-batches of data.

This completes the work needed to set up the data pipeline for model training. We will now work toward the actual model itself.

Defining the CNN-LSTM model

1. Now that we have set up our data pipeline, we will define the model architecture as per the description in *Figure 3.1*, as follows:

```
class CNNModel(nn.Module):
    def __init__(self, embedding_size):
        """Load pretrained ResNet-152 & replace
        last fully connected layer."""
        super(CNNModel, self).__init__()
        resnet = models.resnet152(pretrained=True)
        module_list = list(resnet.children())[:-1]

# delete last fully connected layer.
        self.resnet_module = nn.Sequential(*module_list)
        self.linear_layer = nn.Linear(
            resnet.fc.in_features, embedding_size)
        self.batch_norm = nn.BatchNorm1d(
            embedding_size, momentum=0.01)

    def forward(self, input_images):
        """Extract feats from images."""
        with torch.no_grad():
            resnet_features = \
                self.resnet_module(input_images)
            resnet_features = \
                resnet_features.reshape(
                                   resnet_features.size(0), -1)
            final_features = \
                self.batch_norm(
                                self.linear_layer(
                                    resnet_features))
        return final_features
```

We have defined two sub-models – that is, a CNN model and an RNN model. For the CNN part, we use a pretrained CNN model available under the PyTorch models repository: the ResNet 152 architecture. As we have learned in *Chapter 2*, *Deep CNN Architectures*, this deep CNN model with 152 layers is pretrained on the ImageNet dataset [5]. The ImageNet dataset contains over 1.4 million RGB images labeled over 1,000 classes. These 1,000 classes belong to categories such as plants, animals, food, sports, and more.

We remove the last layer of this pretrained ResNet model and replace it with a fully connected layer followed by a batch normalization layer.

FAQ – Why are we able to replace the fully connected layer?

The neural network can be seen as a sequence of weight matrices starting from the weight matrix between the input layer and the first hidden layer straight up to the weight matrix between the penultimate layer and the output layer. A pretrained model can then be seen as a sequence of nicely tuned weight matrices.

By replacing the final layer, we are essentially replacing the final weight matrix (K x 1000-dimensional, assuming K number of neurons in the penultimate layer) with a new randomly initialized weight matrix (K x 256-dimensional, where 256 is the new output size).

The batch normalization layer normalizes the fully connected layer outputs with a mean of `0` and a standard deviation of `1` across the entire batch. This is similar to the standard input data normalization that we perform using `torch.transforms`. Performing batch normalization helps limit the extent to which the hidden layer output values fluctuate. It also generally helps with faster learning. We can use higher learning rates because of a more uniform (`0` mean, `1` standard deviation) optimization hyperplane.

Since this is the final layer of the CNN sub-model, batch normalization helps insulate the LSTM sub-model against any data shifts that the CNN might introduce. If we do not use batch normalization, then in the worst-case scenario, the CNN final layer could output values with, say, mean > 0.5 and standard deviation = 1 during training (say, because the training data had a limited data distribution). But during inference, if for a certain image the CNN outputs values with mean < 0.5 and standard deviation = 1, then the LSTM sub-model would struggle to operate on this unforeseen data distribution. Hence, we need to standardize the CNN output, which essentially becomes the LSTM input, using batch normalization. This ensures LSTM doesn’t produce unexpected outputs.

Coming back to the fully connected layer, we introduce our own layer because we do not need the 1,000 class probabilities of the ResNet model. Instead, we want to use this model to generate an embedding vector for each image. This embedding can be thought of as a one-dimensional, numerically encoded version of a given input image.

This embedding is then fed to the LSTM model:

2. We will explore LSTMs in detail in *Chapter 4*, *Deep Recurrent Model Architectures*. But, as we have seen in *Figure 3.1*, the LSTM layer takes in the embedding vectors as input and outputs a sequence of words that should ideally describe the image from which the embedding was generated:

```
class LSTMModel(nn.Module):
    def __init__(self, embedding_size, hidden_layer_size,
                 vocabulary_size, num_layers,
                 max_seq_len=20):
        ...
        self.lstm_layer = nn.LSTM(embedding_size,
                                  hidden_layer_size,
                                  num_layers,
                                  batch_first=True)
        self.linear_layer = nn.Linear(hidden_layer_size,
                                      vocabulary_size)
        ...

    def forward(self, input_features, capts, lens):
        ...
        hidden_variables, _ = self.lstm_layer(lstm_input)
        model_outputs = \
            self.linear_layer(hidden_variables[0])
        return model_outputs
```

The LSTM model consists of an LSTM layer followed by a fully connected linear layer. The LSTM layer is a recurrent layer, which can be imagined as LSTM cells unfolded along the time dimension, forming a temporal sequence of LSTM cells. For our use case, these cells will output word prediction probabilities at each time-step and the word with the highest probability is appended to the output sentence.

The LSTM cell at each time-step also generates an internal cell state, which is passed on as input to the LSTM cell of the next time-step. The process continues until an LSTM cell outputs an `<end>` token/word. The `<end>` token is appended to the output sentence. The completed sentence is our predicted caption for the image.

Note that we also specify the maximum allowed sequence length as `20` under the `max_seq_len` variable. This will essentially mean that any sentence shorter than 20 words will have empty word tokens padded at the end and sentences longer than 20 words will be curtailed to just the first 20 words.

Why do we do it and why 20? If we truly want our LSTM to handle sentences of any length, we might want to set this variable to an extremely large value, say, 9,999 words. However, (a) not many image captions come with that many words, and (b), more importantly, if there were ever such extra-long outlier sentences, the LSTM would struggle with learning temporal patterns across such a huge number of time-steps.

We know that LSTMs are better than RNNs at dealing with longer sequences; however, it is difficult to retain memory across such sequence lengths. We choose `20` as a reasonable number given the usual image caption lengths and the maximum length of captions we would like our model to generate.

3. Both the LSTM layer and the linear layer objects in the previous code are derived from `nn.module` and we define the `__init__` and `forward` methods to construct the model and run a forward pass through the model, respectively. For the LSTM model, we additionally implement a `sample` method, as shown in the following code, which will be useful for generating captions for a given image:

```
def sample(self, input_features, lstm_states=None):
    """Generate caps for feats with greedy search."""
    sampled_indices = []
    ...
    for i in range(self.max_seq_len):
    ...
        sampled_indices.append(predicted_outputs)
        ...
    sampled_indices = torch.stack(sampled_indices, 1)
    return sampled_indices
```

The `sample` method makes use of greedy search to generate sentences; that is, it chooses the sequence with the highest overall probability.

This brings us to the end of the image captioning model definition step. We are now all set to train this model.

Training the CNN-LSTM model

As we have already defined the model architecture in the previous section, we will now train the CNN-LSTM model. Let's examine the details of this step one by one:

1. First, we define the device. If there is a GPU available, use it for training; otherwise, use the CPU:

```
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available()
                      else 'cpu')
```

Although we have already reshaped all the images to a fixed shape, `(256, 256)`, that is not enough. We still need to normalize the data.

FAQ – Why do we need to normalize the data?

Normalization is important because different data dimensions might have different distributions, which might skew the overall optimization space and lead to inefficient gradient descent (think of an ellipse versus a circle).

2. We will use PyTorch's `transform` module to normalize the input image pixel values:

```
# Image pre-processing, normalization for pretrained resnet
transform = transforms.Compose([
    transforms.RandomCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.485, 0.456, 0.406),
                         (0.229, 0.224, 0.225))])
```

   Furthermore, we augment the available dataset.

   FAQ – Why do we need data augmentation?

   Augmentation helps not only in generating larger volumes of training data but also in making the model robust against potential variations in input data.

   Using PyTorch's `transform` module, we implement two data augmentation techniques here:

   i. Random cropping, resulting in the reduction of the image size from `(256, 256)` to `(224, 224)`. This helps us generate multiple images from a single image by cropping different parts of an image. This also enables the model to learn more robust image representations not only fixating on the exact `(256, 256)` images available in the training dataset.

   ii. Horizontal flipping of the images. This produces twice as many images as we could have, and more data is always better. And for the given dataset, horizontal flipping doesn't distort the fundamental meaning of the image – a dog is still a dog in a horizontally flipped image – so this augmentation makes sense. If we were detecting digits, then horizontally flipping the image of digit `5` and still training the model to classify it as `5` will confuse the model because the flipped image will look more like `2` than `5`.

3. Next, we load the vocabulary that we built in the *Preprocessing caption (text) data* section. We also initialize the data loader using the `get_loader()` function defined in the *Defining the image captioning data loader* section:

```
# Load vocab wrapper
with open('data_dir/vocabulary.pkl', 'rb') as f:
    vocabulary = pickle.load(f)

# Instantiate data loader
custom_data_loader = get_loader('data_dir/resized_images',
    'data_dir/annotations/captions_train2014.json',
    vocabulary, transform, 128, shuffle=True)
```

4. Next, we arrive at the main section of this step, where we instantiate the CNN and LSTM models in the form of encoder and decoder models. Furthermore, we also define the loss function – **cross entropy loss** – and the optimization schedule – the **Adam optimizer** – as follows:

```
# Build models
encoder_model = CNNModel(256).to(device)
decoder_model = LSTMModel(256, 512,
                          len(vocabulary), 1).to(device)

# Loss & optimizer
loss_criterion = nn.CrossEntropyLoss()
parameters = list(decoder_model.parameters()) + \
        list(encoder_model.linear_layer.parameters()) + \
        list(encoder_model.batch_norm.parameters())
optimizer = torch.optim.Adam(parameters, lr=0.001)
```

As discussed in *Chapter 1*, *Overview of Deep Learning Using PyTorch*, Adam is possibly the best choice for an optimization schedule when dealing with sparse data. Here, we are dealing with both images and text – perfect examples of sparse data because not all pixels contain useful information and numericized/vectorized text is a sparse matrix in itself.

5. Finally, we run the training loop (for five epochs) where we use the data loader to fetch a mini-batch of the COCO dataset, run a forward pass with the mini-batch through the encoder and decoder networks, and finally, tune the parameters of the CNN-LSTM model using backpropagation (backpropagation through time, for the LSTM network):

```
for epoch in range(5):
    for i, (imgs, caps, lens) in enumerate(custom_data_loader):
        tgts = pack_padded_sequence(caps, lens,
                                    batch_first=True)[0]
        # Forward pass, backward propagation
        feats = encoder_model(imgs)
        outputs = decoder_model(feats, caps, lens)
        loss = loss_criterion(outputs, tgts)
        decoder_model.zero_grad()
        encoder_model.zero_grad()
        loss.backward()
        optimizer.step()
```

Every 1,000 iterations into the training loop, we save a model checkpoint. For demonstration purposes, we have run the training for just two epochs, as follows:

```
        # Log training steps
        if i % 10 == 0:
           print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}, Perplexity: 
{:5.4f}'.format(epoch, 5, i, total_num_steps,
                         loss.item(), np.exp(loss.item()))) 
        # Save model checkpoints
        if (i+1) % 1000 == 0:
            torch.save(decoder_model.state_dict(), 
                       os.path.join('models_dir/', 'decoder-{}-{}.ckpt'
                                    .format(epoch+1, i+1)))
            torch.save(encoder_model.state_dict(), 
                       os.path.join('models_dir/', 'encoder-{}-{}.ckpt'
                                    .format(epoch+1, i+1)))
```

The output will be as follows:

```
loading annotations into memory...
Done (t=0.65s)
creating index...
index created!
Epoch [0/5], Step [0/3236], Loss: 9.2049, Perplexity: 9946.0549
Epoch [0/5], Step [10/3236], Loss: 5.7688, Perplexity: 320.1389
Epoch [0/5], Step [20/3236], Loss: 5.4142, Perplexity: 224.5734
...
Epoch [2/5], Step [300/3236], Loss: 2.0740, Perplexity: 7.9563
Epoch [2/5], Step [310/3236], Loss: 1.9858, Perplexity: 7.2848
Epoch [2/5], Step [320/3236], Loss: 2.0391, Perplexity: 7.6838
```

Generating image captions using the trained model

We have trained an image captioning model in the previous section. In this section, we will use the trained model to generate captions for images previously unseen by the model:

1. We have stored a sample image, `sample.jpg`, to run inference on. We define a function to load the image and reshape it to `(224, 224)` pixels. Then, we define the transformation module to normalize the image pixels, as follows:

```
image_file_path = 'sample.jpg'
# Device config
device = torch.device(
    'cuda' if torch.cuda.is_available() else 'cpu')
def load_image(image_file_path, transform=None):
    img = Image.open(image_file_path).convert('RGB')
    img = img.resize([224, 224], Image.LANCZOS)
    if transform is not None:
        img = transform(img).unsqueeze(0)
    return img
# Image pre-processing
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.485, 0.456, 0.406),
                         (0.229, 0.224, 0.225))])
```

2. Next, we load the vocabulary and instantiate the encoder and decoder models:

```
# Load vocab wrapper
with open('data_dir/vocabulary.pkl', 'rb') as f:
    vocabulary = pickle.load(f)
# Build models
encoder_model = CNNModel(256).eval()
# eval mode (batchnorm uses moving mean/variance)

decoder_model = LSTMModel(256, 512, len(vocabulary), 1)
encoder_model = encoder_model.to(device)
decoder_model = decoder_model.to(device)
```

Once we have the model scaffold ready, we will use the latest saved checkpoint from the two epochs of training to set the model parameters:

```
# Load trained model params
encoder_model.load_state_dict(
    torch.load('models_dir/encoder-2-3000.ckpt'))
decoder_model.load_state_dict(
    torch.load('models_dir/decoder-2-3000.ckpt'))
```

After this point, the model is ready to use for inference.

3. Next, we load the image and run model inference – that is, first we use the encoder model to generate an embedding from the image, and then we feed the embedding to the decoder network to generate sequences, as follows:

```
# Prepare image
img = load_image(image_file_path, transform)
img_tensor = img.to(device)
# Generate caption text from image
feat = encoder_model(img_tensor)
sampled_indices = decoder_model.sample(feat)
sampled_indices = sampled_indices[0].cpu().numpy()
# (1, max_seq_length) -> (max_seq_length)
```

4. At this stage, the caption predictions are still in the form of numeric tokens. We need to convert the numeric tokens into actual text using the vocabulary in reverse:

```
# Convert numeric tokens to text tokens
predicted_caption = []
for token_index in sampled_indices:
    word = vocabulary.i2w[token_index]
    predicted_caption.append(word)
    if word == '<end>':
        break
predicted_sentence = ' '.join(predicted_caption)
```

5. Once we have transformed our output into text, we can visualize both the image as well as the generated caption:

```
# Print image & generated caption text
print (predicted_sentence)
img = Image.open(image_file_path)
plt.imshow(np.asarray(img))
```

The output will be as follows:

```
<start> a dog is standing on a sidewalk near a building . <end>
```

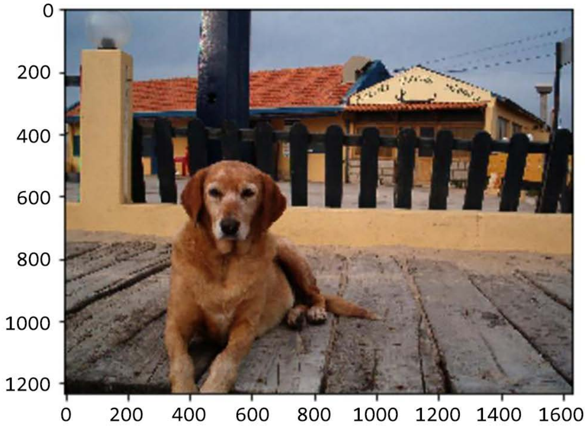


Figure 3.2: Model inference on a sample image

It seems that although the model is not absolutely perfect, within two epochs, it is already trained well enough to generate sensible captions.

Summary

This chapter discussed the concept of combining a CNN model and an LSTM model in an encoder-decoder framework, jointly training them, and using the combined model to generate captions for an image.

While we have mostly worked with CNNs in the chapters thus far, in the next chapter we dig deeper into recurrent models such as the LSTM model used in this chapter. We explore different recurrent network architectures, and learn how to use them with PyTorch.

References

1. Ahmet Aker and Robert Gaizauskas. 2010. *Generating image descriptions using dependency relational patterns*: `https://dl.acm.org/doi/10.5555/1858681.1858808`
2. Oriol Vinyals, Alexander Toshev, Samy Bengio, Dumitru Erhan; Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015. *Show and Tell: A Neural Image Caption Generator*: `https://www.cv-foundation.org/openaccess/content_cvpr_2015/html/Vinyals_Show_and_Tell_2015_CVPR_paper.html`
3. Common Objects in Context (COCO) dataset: `https://cocodataset.org/#overview`
4. Mastering PyTorch GitHub repository link: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter02/image_captioning_pytorch.ipynb`
5. ImageNet dataset: `https://image-net.org/`

Learn more on Discord

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

`https://packt.link/mastorch`

4

Deep Recurrent Model Architectures

Neural networks are powerful machine learning tools that are used to help us learn complex patterns between the inputs (X) and outputs (y) of a dataset. In *Chapter 2*, *Deep CNN Architectures*, we discussed convolutional neural networks, which learn a one-to-one mapping between X and y; that is, each input, X, is independent of the other inputs, and each output, y, is independent of the other outputs of the dataset.

In the previous chapter we combined a CNN model with a recurrent model (LSTM) to build an image caption generator. In this chapter, we will expand on the recurrent model. We will discuss a class of neural networks that can model sequences where X (or y) is not just a single independent data point, but a temporal sequence of data points [X_1, X_2, .. X_t] (or [y_1, y_2, .. y_t]). Note that X_2 (which is the data point at time step 2) is dependent on X_1, X_3 is dependent on X_2 and X_1, and so on.

Such networks are classified as **Recurrent Neural Networks** (**RNNs**). These networks are capable of modeling the temporal aspect of data by including additional weights in the model that create cycles in the network. This helps maintain a state, as shown in the following diagram:

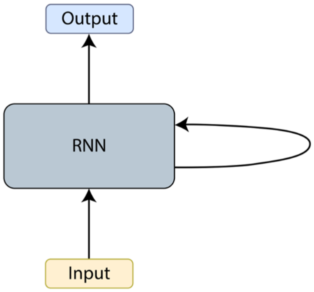


Figure 4.1: RNN

The concept of cycles explains the term *recurrence*, and this recurrence helps establish the concept of memory in these networks. Essentially, such networks facilitate the use of intermediate outputs at *time step t* as inputs for *time step t+1*, while maintaining a hidden internal state. These connections across time steps are called **recurrent connections**.

This chapter will focus on the various recurrent neural network architectures that have been developed over the years, such as different types of RNNs, **Long Short-Term Memory** (**LSTM**), and **Gated Recurrent Units** (**GRUs**). We will use PyTorch to implement some of these architectures and train and test recurrent models on real-world sequential modeling tasks. Besides model training and testing, we will also learn how to efficiently use PyTorch to load and preprocess sequential data. By the end of this chapter, you will be ready to solve machine learning problems with sequential datasets using RNNs in PyTorch.

This chapter covers the following topics:

- Exploring the evolution of recurrent networks
- Training RNNs for sentiment analysis
- Building a bidirectional LSTM
- Discussing GRUs and attention-based models

> All the code files for this chapter can be found at: `https://github.com/arj7192/MasteringPyTorchV2/tree/main/Chapter04`.

Exploring the evolution of recurrent networks

Recurrent networks have been around since the 80s. In this section, we will explore the evolution of the recurrent network architecture since its inception. We will discuss and reason about the developments that were made to the architecture by going through the key milestones in the evolution of RNNs. Before jumping right into the timelines, we'll quickly review the different types of RNNs and how they relate to a general feed-forward neural network.

Types of recurrent neural networks

While most supervised machine learning models model one-to-one relationships, RNNs can model the following types of input-output relationships:

- Many-to-many (instantaneous)

  Example: Named entity recognition – Given a sentence/text, tag the words with named entity categories such as names, organizations, locations, and so on.

- Many-to-many (encoder-decoder)

  Example: Machine translation (say, from English text to German text) – Takes in a sentence/piece of text in a natural language, encodes it into a consolidated fixed-size representation, and decodes that representation to produce an equivalent sentence/piece of text in another language.

- Many-to-one

  Example: Sentiment analysis – Given a sentence/piece of text, classify it as positive, negative, neutral, and so on.

- One-to-many

  Example: Image captioning – Given an image, produce a sentence/piece of text describing it.

- One-to-one (although not very useful)

  Example: Image classifications (by processing image pixels sequentially).

The following diagram shows these RNN types in contrast to the regular NN:

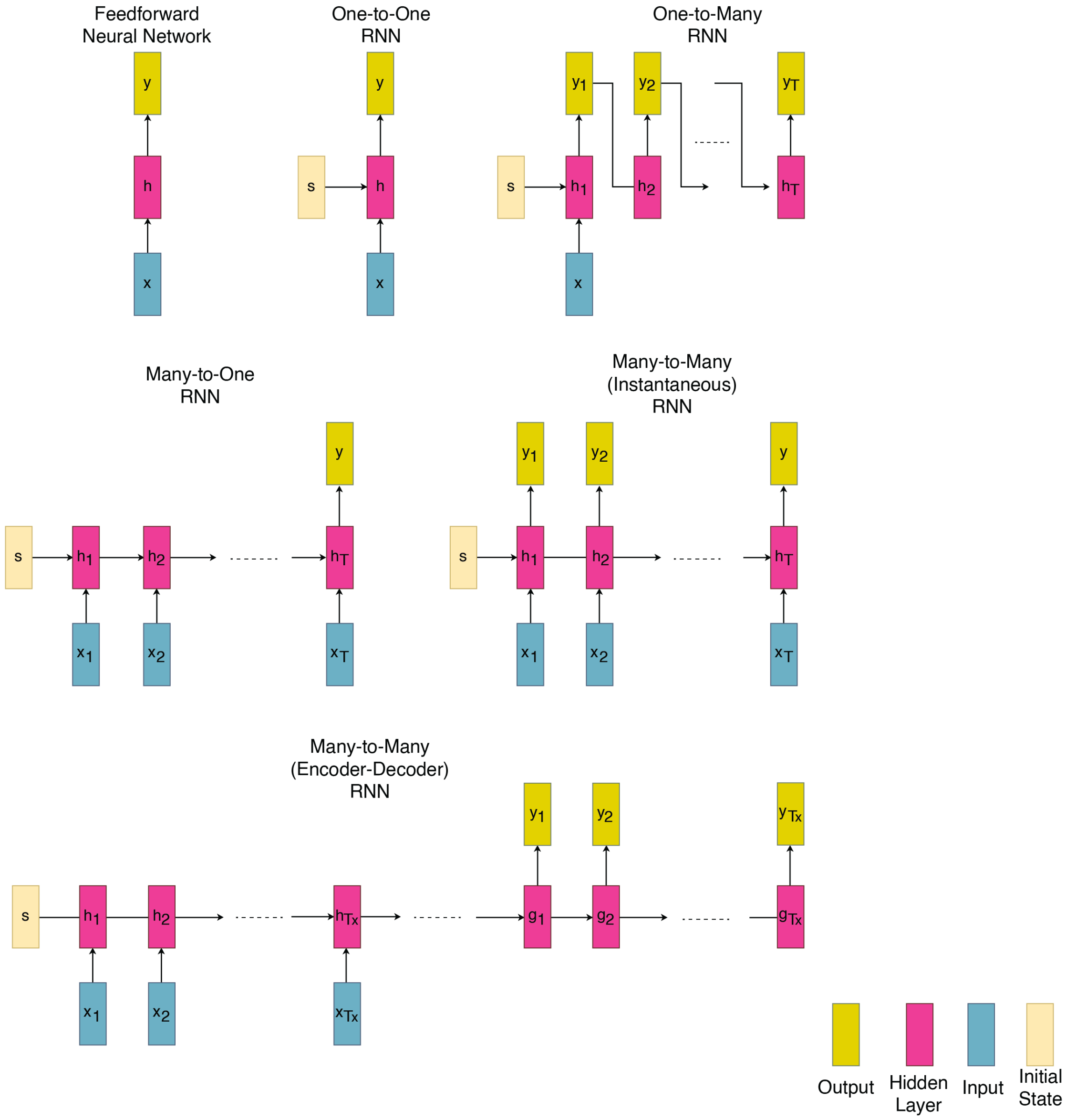


Figure 4.2: Types of RNNs

Note

We provided an example of a one-to-many recurrent neural network in the image captioning exercise in *Chapter 3, Combining CNNs and LSTMs*.

As we can see, recurrent neural architectures have recurrent connections that do not exist in regular NNs. These recurrent connections are unfolded along the time dimension in the preceding diagram. The following diagram shows the structure of an RNN in both **time-folded** and **time-unfolded** forms:

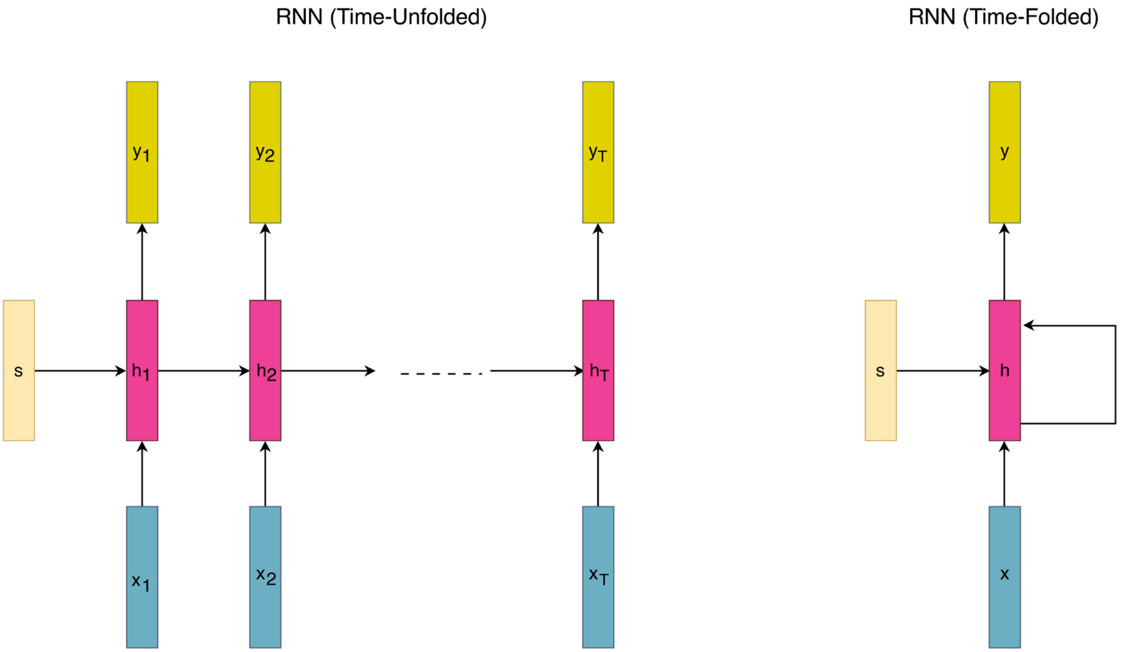


Figure 4.3: Temporal unfolding of an RNN

In the following sections, we will be using the time-unfolded version to demonstrate RNN architectures. In the preceding diagrams, we have marked the RNN layer in red as the hidden layer of the neural network. Although the network might seem to just have one hidden layer, once this hidden layer is unrolled along the time dimension, we can see that the network actually has *T* hidden layers. Here, *T* is the total number of time steps in the sequential data.

One of the powerful features of RNNs is that they can deal with sequential data of varying sequence lengths (*T*). One way of dealing with this variability in length is by padding shorter sequences and truncating longer sequences, as we will see in the exercises provided later in this chapter.

Next, we will delve into the history and evolution of recurrent architectures, starting with basic RNNs.

RNNs

The idea behind RNNs became evident with the emergence of the Hopfield network in 1982, which is a special type of RNN that tries to emulate the workings of human memory. RNNs later came into their own existence based on the works of David Rumelhart, among others, in 1986. These RNNs were able to process sequences with an underlying concept of memory. From here, a series of improvements were made to its architecture, as shown in the following diagram:

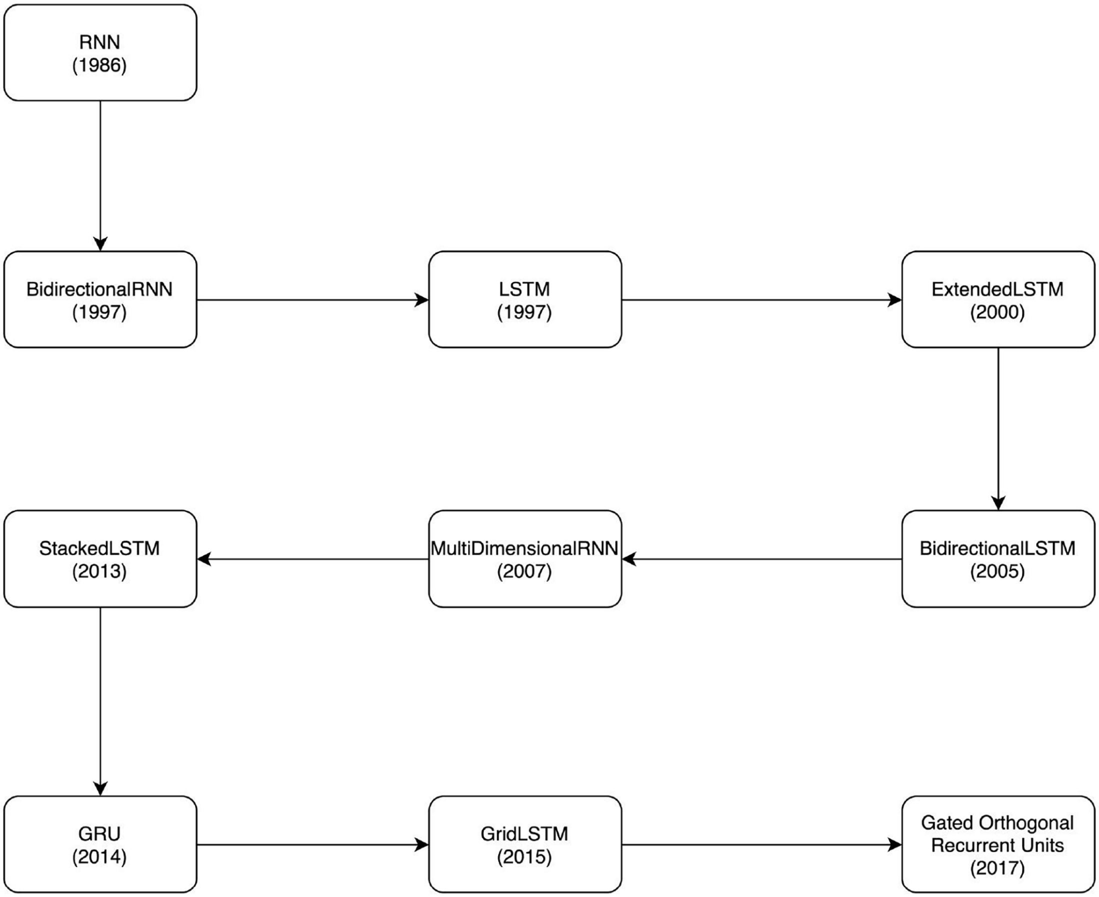


Figure 4.4: RNN architecture evolution – a broad picture

The preceding diagram does not cover the entire history of the architecture evolution of RNNs, but it does cover the important checkpoints. Next, we will discuss the successors of RNNs chronologically, starting with bidirectional RNNs.

Bidirectional RNNs

Although RNNs performed well on sequential data, it was later realized that some sequence-related tasks, such as language translation, can be done more efficiently by looking at both past and future information. For example, *I see you* in English would be translated to *Je te vois* in French. Here, *te* means *you* and *vois* means *see*. Hence, in order to correctly translate English into French, we need all three words in English before writing the second and third words in French.

To overcome this limitation, **bidirectional RNNs** were invented in 1997. These are pretty similar to conventional RNNs except that bidirectional RNNs have two RNNs working internally: one running the sequence from start to end, and another running the sequence backward from end to start, as shown in the following diagram:

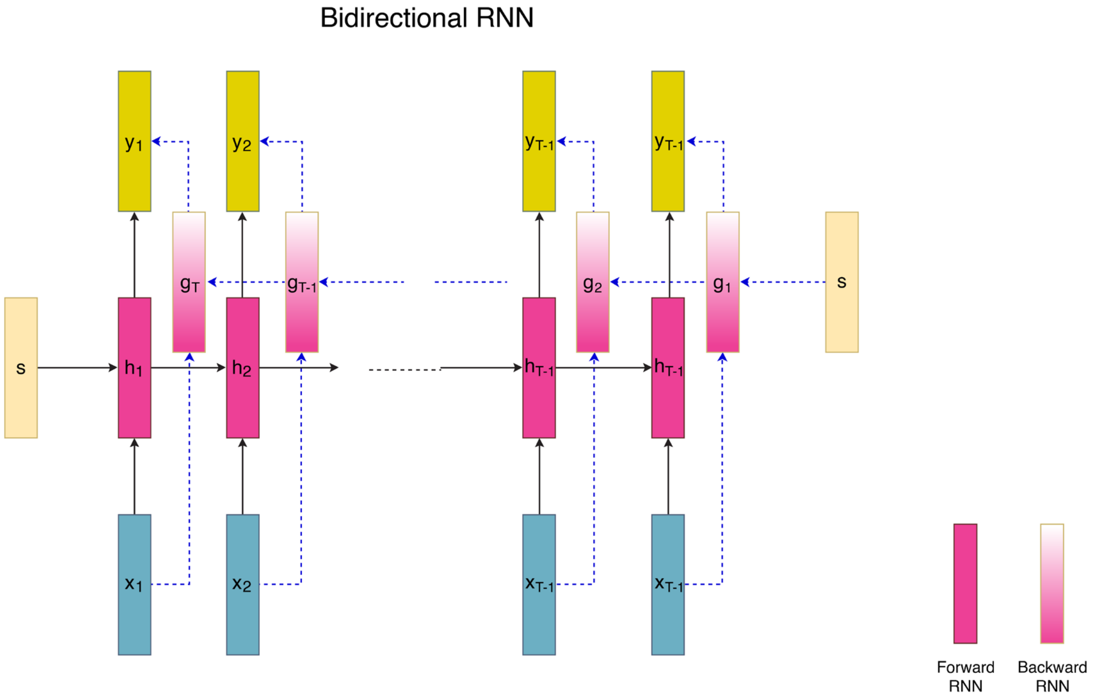


Figure 4.5: Bidirectional RNNs

Next, we will learn about LSTMs.

LSTMs

While RNNs were able to deal with sequential data and remember information, they suffered from the problem of exploding and vanishing gradients. This happened because of the extremely deep networks that resulted from unfolding the recurrent networks in the time dimension.

In 1997, a different approach was devised. The RNN cell was replaced with a more sophisticated memory cell – the **Long short-term memory** (**LSTM**) cell. The RNN cell usually has a **sigmoid** or **tanh** activation function.

These functions are chosen because of their ability to control the output between the values of 0 (no information flow) to 1 (complete information flow), or -1 to 1 in the case of tanh.

Tanh is additionally advantageous for providing a 0 mean output value and larger gradients in general – both of which contribute to faster learning (convergence). These activation functions are applied to the concatenation of the current time step's input and the previous time step's hidden state, as shown in the following diagram:

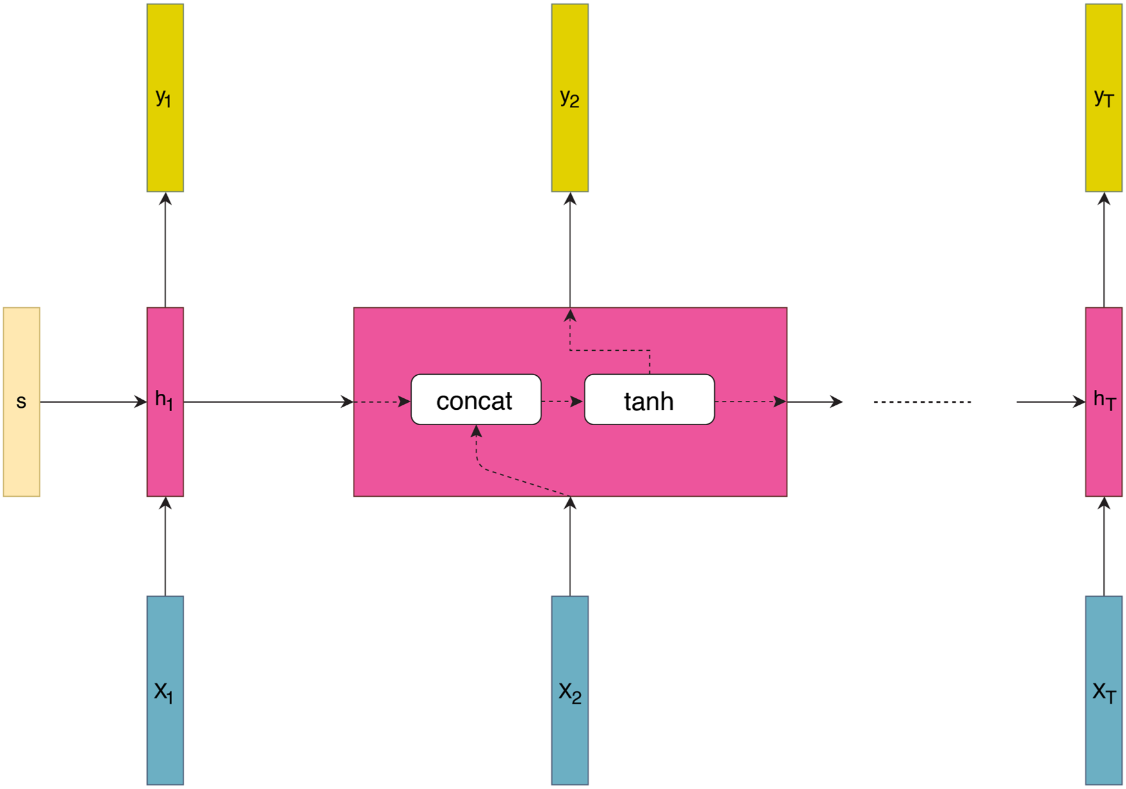


Figure 4.6: RNN cell

During **backpropagation**, the gradient either keeps diminishing across several of these RNN cells or keeps growing, due to the multiplication of gradient terms across time-unfolded RNN cells. So, while RNNs can remember the sequential information across short sequences, they tend to struggle with long sequences due to the larger number of multiplications. LSTMs resolve this issue by controlling their input and output using gates.

An LSTM layer essentially consists of various time-unfolded LSTM cells. Information passes from one cell to another in the form of cell states. These cell states are controlled or manipulated using multiplications and additions using the mechanism of gates.

These gates, as shown in the following diagram, control the flow of information to the next cell while preserving or forgetting the information that's coming in from the previous cell:

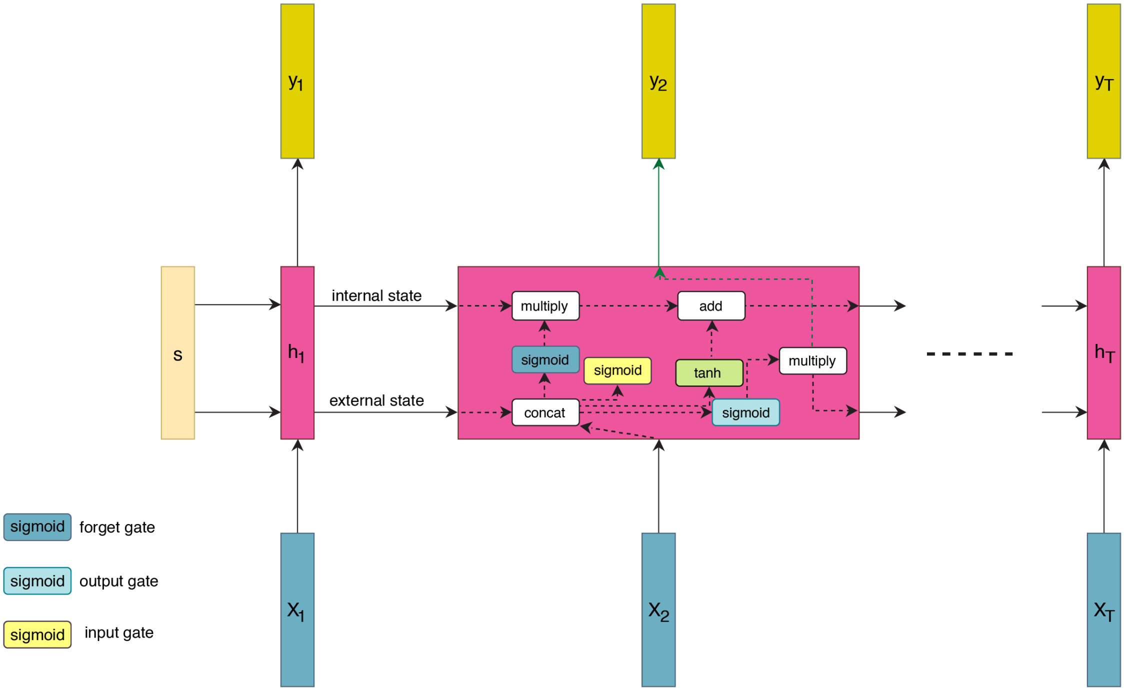


Figure 4.7: LSTM network

LSTMs revolutionized recurrent networks as they can efficiently deal with much longer sequences. Next, we discuss more advanced variants of LSTMs.

Extended and bidirectional LSTMs

Originally, in 1997, LSTMs were invented with just the input and output gates. Soon after, in 2000, an extended LSTM was developed with forget gates, which is mostly used nowadays. A few years later, in 2005, bidirectional LSTMs were developed, which are similar in concept to bidirectional RNNs.

Multi-dimensional RNNs

In 2007, **multi-dimensional RNNs** (**MDRNNs**) were invented. Here, a single recurrent connection between RNN cells is replaced by as many connections as there are dimensions in the data. This was useful in video processing, for example, where the data is a sequence of images that is inherently two-dimensional.

Stacked LSTMs

Although single-layer LSTM networks did seem to overcome the problem of vanishing and exploding gradients, stacking more LSTM layers proved more helpful in learning highly complex patterns across various sequential processing tasks, such as speech recognition.

These powerful models were called **stacked LSTMs**. The following diagram shows a stacked LSTM model with two LSTM layers:

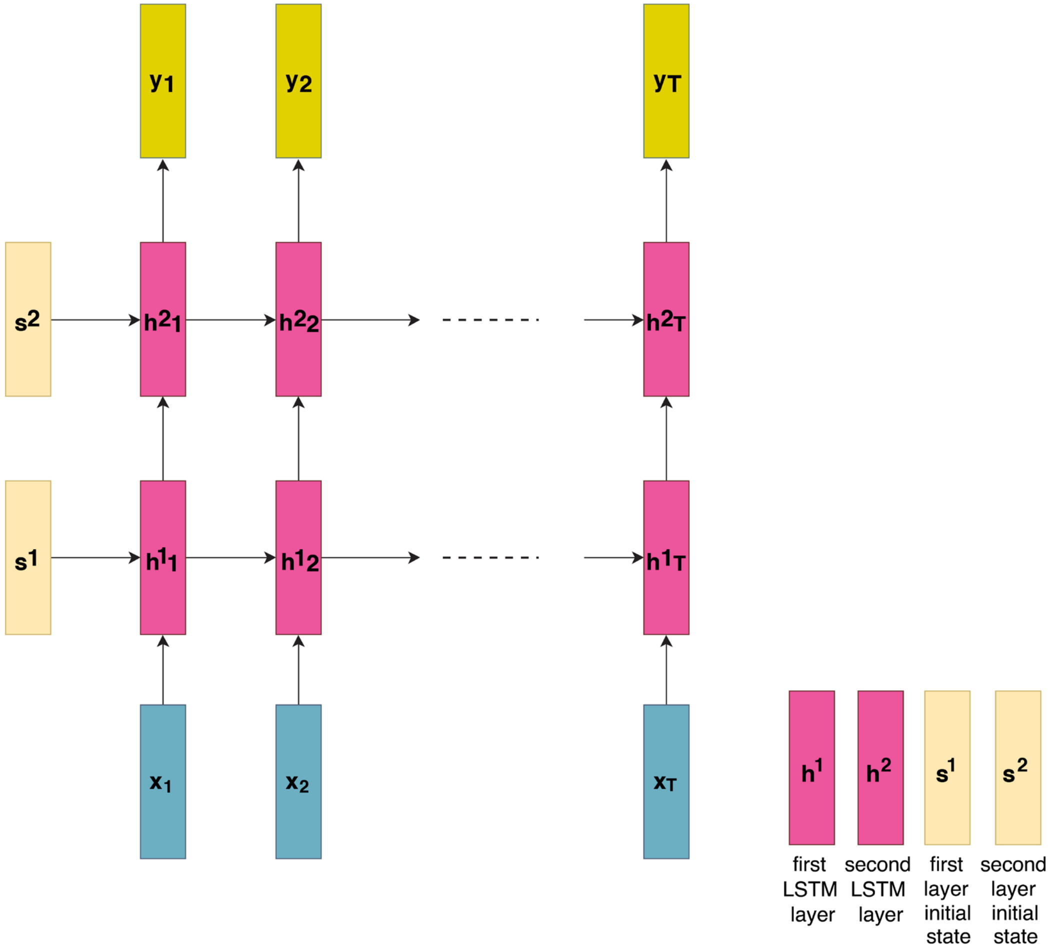


Figure 4.8: Stacked LSTMs

LSTM cells are, by their very nature, stacked in the time dimension of an LSTM layer. Stacking several such layers in the space dimension provides them with the additional depth in space they need. The downside of these models is that they are significantly slower to train due to the extra depth and extra recurrent connections they have. Furthermore, the additional LSTM layers need to be unrolled (in the time dimension) at every training iteration. Hence, training stacked recurrent models, in general, is not parallelizable.

GRUs

The LSTM cell has two states – internal and external – as well as three different gates – **input gate**, **forget gate**, and **output gate**. A similar type of cell, named a **gated recurrent unit** (GRU), was invented in 2014 with the goal of learning long-term dependencies while effectively dealing with the exploding and vanishing gradients problem.

GRUs have just one state and only two gates – a **reset gate** (a combination of the input and forget gates) and an **update gate**. The following diagram shows a GRU network:

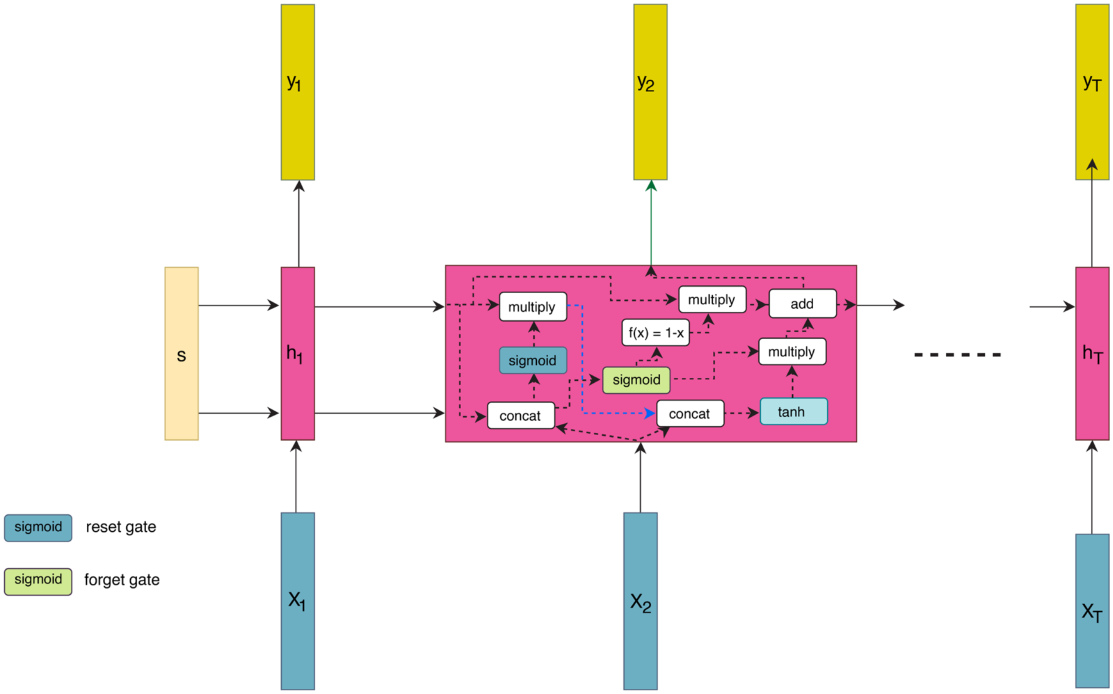


Figure 4.9: GRU network

Next up is the grid LSTM.

Grid LSTMs

A year later, in 2015, the **grid LSTM** model was developed as a successor to the MDLSTM model, as the LSTM equivalent of multi-dimensional RNNs. LSTM cells are arranged into a multi-dimensional grid in a grid LSTM model. These cells are connected along the spatiotemporal dimensions of the data, as well as between the network layers.

Gated orthogonal recurrent units

In 2017, **gated orthogonal recurrent units** were devised, which brought together the ideas of GRUs and **unitary RNNs**. Unitary RNNs are based on the idea of using **unitary matrices** (which are **orthogonal matrices**) as the **hidden-state loop matrices** of RNNs to deal with the problem of exploding and vanishing gradients. This works because deviating gradients are attributed to deviating the **eigenvalues** of the **hidden-to-hidden** weight matrices from one. Due to this, these matrices have been replaced with orthogonal matrices to solve the gradients problem. You can read more about unitary RNNs in the original paper [1].

We briefly covered the evolution of recurrent neural architectures in this section. Next, we will dive deep into RNNs by performing an exercise with a simple RNN model architecture based on a text classification task. We will also explore how PyTorch plays an important role in processing sequential data, as well as building and evaluating recurrent models.

Training RNNs for sentiment analysis

In this section, we will train an RNN model using PyTorch for a text classification task – sentiment analysis. In this task, the model takes in a piece of text – a sequence of words – as input and outputs either `1` (meaning positive sentiment) or `0` (negative sentiment). In order to go from text to 1s and 0s, we will need the help of tokenization and embeddings.

Tokenization is the process of converting words into numerical tokens or integers, as we will see in this exercise. Sentences are then equivalent to an array of numbers, each number in the ordered array representing a word. While tokenization provides us with integer indices for each word, we still want to represent each word as a vector of numbers – as a feature – in the feature space of words. Why? Because the information contained in a word cannot just be represented by a single number. This process of representing words as vectors is called embedding, which we will also use later in this exercise. An embedding matrix, which can be learned during model training, acts as a lookup for word vectors. If a word has a token index of 123, then the embedding for that word is the vector contained in row 123 of the embedding matrix.

For this binary classification task involving sequential data, we will use a **unidirectional single-layer RNN**.

Before training the model, we will manually process the textual data and convert it into a usable numeric form. Upon training the model, we will test it on some sample texts. We will demonstrate the use of various PyTorch functionalities to efficiently perform this task. The code for this exercise can be found in our GitHub repository [2].

Loading and preprocessing the text dataset

For this exercise, we will need to import a few dependencies:

1. First, execute the following `import` statements:

```
import os
import time
import numpy as np
from tqdm import tqdm
from string import punctuation
from collections import Counter
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
```

```python
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
torch.use_deterministic_algorithms(True)
```

Besides importing the regular `torch` dependencies, we have also imported `punctuation` and `Counter` for text processing. We have also imported `matplotlib` to display images, `numpy` for array operations, and `tqdm` for visualizing progress bars. Besides imports, we have also set the random seed to ensure the reproducibility of this exercise, as shown in the last line of the code snippet.

2. Next, we will read the data from the text files. For this exercise, we will be using the IMDb sentiment analysis dataset [3]. This IMDb dataset consists of several movie reviews as texts and corresponding sentiment labels (positive or negative). First, we will download the dataset and run the following lines of code in order to read and store the list of texts and corresponding sentiment labels:

```python
# read sentiments and reviews data from the text files
review_list = []
label_list = []
for label in ['pos', 'neg']:
    for fname in tqdm(os.listdir(
        f'./aclImdb/train/{label}/')):
        if 'txt' not in fname:
            continue
        with open(os.path.join(f'./aclImdb/train/{label}/',
                                      fname), encoding="utf8") as f:
            review_list += [f.read()]
            label_list += [label]
print ('Number of reviews :', len(review_list))
```

This should output the following:

```
Number of reviews : 25000
```

As we can see, there are a total of 25,000 movie reviews, with 12,500 positive and 12,500 negative.

Dataset citation

Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. (2011). *Learning Word Vectors for Sentiment Analysis*. The 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011).

3. Following the data loading step, we will now start processing the text data, as follows:

```
# pre-processing review text
review_list = [review.lower() for review in review_list]
review_list = [''.join([letter for letter in review
                        if letter not in punctuation])
                        for review in tqdm(review_list)]
# accumulate all review texts together
reviews_blob = ' '.join(review_list)
# generate list of all words of all reviews
review_words = reviews_blob.split()
# get the word counts
count_words = Counter(review_words)
# sort words as per counts (decreasing order)
total_review_words = len(review_words)
sorted_review_words = count_words.most_common(total_review_words)
print(sorted_review_words[:10])
```

This should output the following:

```
[('the', 334691), ('and', 162228), ('a', 161940), ('of', 145326), ('to',
135042), ('is', 106855), ('in', 93028), ('it', 77099), ('i', 75719),
('this', 75190)]
```

As you can see, first, we lower-cased the entire text corpus and, subsequently, removed all punctuation marks from the review texts. Then, we accumulated all the words in all the reviews together to get word counts and sorted them in decreasing order of counts, to see the most popular words. Note that the most popular words are all **non-nouns** such as determiners, pronouns, and more, as shown in the preceding output.

Ideally, these non-nouns, also referred to as **stop words**, would be removed from the corpus as they do not carry a lot of meaning. However, we will skip those advanced text-processing steps to keep things simple.

4. We will continue with data processing by converting these individual words into numbers or tokens. This is a crucial step because machine learning models only understand numbers, not words:

```
# create word to integer (token) dictionary
# in order to encode text as numbers
vocab_to_token = {word:idx+1 for idx,
                  (word, count) in enumerate(sorted_review_words)}
print(list(vocab_to_token.items())[:10])
```

This should output the following:

```
[('the', 1), ('and', 2), ('a', 3), ('of', 4), ('to', 5), ('is', 6),
('in', 7), ('it', 8), ('i', 9), ('this', 10)]
```

Starting with the most popular word, numbers are assigned to words 1 onward.

5. We obtained the word-to-integer mapping in the previous step, which is also known as the vocabulary of our dataset. In this step, we will use the vocabulary to translate movie reviews in our dataset into a list of numbers:

```
reviews_tokenized = []
for review in review_list:
    word_to_token = [vocab_to_token[word] for word in
                     review.split()]
    reviews_tokenized.append(word_to_token)
print(review_list[0])
print()
print (reviews_tokenized[0])
```

This should output something like the following:

```
for a movie that gets no respect there sure are a lot of memorable quotes
listed for this gem imagine a movie where joe piscopo is actually funny
maureen stapleton is a scene stealer the moroni character is an absolute
scream watch for alan the skipper hale jr as a police sgt

[15, 3, 17, 11, 201, 56, 1165, 47, 242, 23, 3, 168, 4, 891, 4325, 3513,
15, 10, 1514, 822, 3, 17, 112, 884, 14623, 6, 155, 161, 7307, 15816, 6,
3, 134, 20049, 1, 32064, 108, 6, 33, 1492, 1943, 103, 15, 1550, 1, 18993,
9055, 1809, 14, 3, 549, 6906]
```

6. We shall also encode the sentiment targets – pos and neg – into numbers 1 and 0, respectively:

```
# encode sentiments as 0 or 1
encoded_label_list = [1 if label =='pos'
                      else 0 for label in label_list]
reviews_len = [len(review) for review in reviews_tokenized]
reviews_tokenized = [reviews_tokenized[i]
                     for I, l in enumerate(reviews_len)
                     if l>0 ]
encoded_label_list = np.array([encoded_label_list[i]
                              for i, l in enumerate(reviews_len)
                              if l> 0 ], dtype''float3'')
```

7. Before we train the model, we need a final data-processing step. Different reviews can be of different lengths. However, we will define our simple RNN model for a fixed sequence length. Hence, we need to normalize different-length reviews so that they are all the same length.

   For this, we will define a sequence length *L* (`512`, in this case), and then pad sequences that are smaller than *L* in length and truncate sequences that are longer than *L*:

```
def pad_sequence(reviews_tokenized, sequence_length):
        ''' returns the tokenized review sequences padded
            with 's or truncated to the sequence_length.'''
    padded_reviews = np.zeros((len(reviews_tokenized),
                               sequence_length),
                               dtype = int)
    for idx, review in enumerate(reviews_tokenized):
        review_len = len(review)
        if review_len <= sequence_length:
            zeroes = list(np.zeros(
                sequence_length-review_len))
            new_sequence = zeroes+review
        elif review_len > sequence_length:
            new_sequence = review[0:sequence_length]
        padded_reviews[idx,:] = np.array(new_sequence)
    return padded_reviews
sequence_length = 512
padded_reviews = pad_sequence(reviews_tokenized=reviews_tokenized,
             sequence_length=sequence_length)
plt.hist(reviews_len);
```

The output will be as follows:

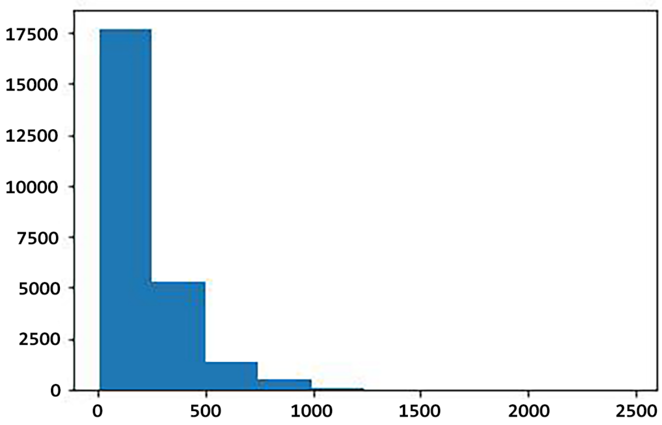


Figure 4.10: Histogram of review lengths

As we can see, the reviews are mostly below 500, so we have chosen 512 (a power of 2) as the sequence length for our model and modified the sequences that are not exactly 512 words long accordingly.

8. Finally, we can train the model. To do this, we must split our dataset into training and validation sets with a 75:25 ratio:

```
train_val_split = 0.75
train_X = \
    padded_reviews[:int(
        train_val_split*len(padded_reviews))]
train_y = \
    encoded_label_list[:int(
        train_val_split*len(padded_reviews))]
validation_X = \
    padded_reviews[int(
        train_val_split*len(padded_reviews)):]
validation_y = \
    encoded_label_list[int(
        train_val_split*len(padded_reviews)):]
```

9. At this stage, we can start using PyTorch to generate the `dataset` and `dataloader` objects from the processed data:

```
# generate torch datasets
train_dataset = TensorDataset(
    torch.from_numpy(train_X).to(device),
    torch.from_numpy(train_y).to(device))
validation_dataset = TensorDataset(
    torch.from_numpy(validation_X).to(device),
    torch.from_numpy(validation_y).to(device))
batch_size = 32
# torch dataloaders (shuffle data)
train_dataloader = DataLoader(
    train_dataset, batch_size=batch_size, shuffle=True)
validation_dataloader = DataLoader(
    validation_dataset, batch_size=batch_size, shuffle=True)
```

10. To get a feeling of what the data looks like before we feed it to the model, let's visualize a batch of 32 reviews and the corresponding sentiment labels:

```
# get a batch of train data
train_data_iter = iter(train_dataloader)
```

```
X_example, y_example = next(train_data_iter)
# batch_size, seq_length
print('Example Input size: ', X_example.size())
print('Example Input:\n', X_example)
print()
# batch_size
print('Example Output size: ', y_example.size())
print('Example Output:\n', y_example)
```

The output will be as follows:

```
Example Input size:  torch.Size([32, 512])
Example Input:
tensor([[    0,     0,     0,  ...,    31,   183,   472],
[    0,     0,     0,  ...,   410,     7,  1272],
[    0,     0,     0,  ...,     5,     3, 27493],
...,
[    0,     0,     0,  ...,    63,     4,  3226],
[    0,     0,     0,  ...,    89,   713,     8],
[    0,     0,     0,  ...,    22,    15,     8]])
Example Output size:  torch.Size([32])
Example Output:
tensor([1., 1., 1., 0., 0., 1., 0., 1., 1., 1., 0., 1., 0., 1., 0., 1.,
0., 1., 1., 1., 0., 1., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Having loaded and processed the textual dataset into sequences of numerical tokens, next, we will create the RNN model object in PyTorch and train the RNN model.

Instantiating and training the model

Now that we have prepared our datasets, we can instantiate our unidirectional single-layer RNN model. Firstly, PyTorch makes it incredibly compact through its **nn.RNN** module to instantiate the RNN layer. All it takes in is the input/embedding dimension, the hidden-to-hidden state dimension, and the number of layers. Let's get started:

1. Let's define our own wrapper RNN class. This instantiates the whole RNN model, which is composed of the embedding layer, followed by the RNN layer, and finally followed by a fully connected layer, as follows:

```
class RNN(nn.Module):
    def __init__(self, input_dimension, embedding_dimension,
                 hidden_dimension, output_dimension):
        super().__init__()
        self.embedding_layer = nn.Embedding(input_dimension,
                                        embedding_dimension)
        self.rnn_layer = nn.RNN(embedding_dimension,
                                hidden_dimension,
                                num_layers=1)
        self.fc_layer = nn.Linear(hidden_dimension,
                                  output_dimension)
    def forward(self, sequence):
        # sequence shape = (sequence_length, batch_size)
        embedding = self.embedding_layer(sequence)
        # embedding shape = [sequence_length, batch_size,
        #                     embedding_dimension]
        output, hidden_state = self.rnn_layer(embedding)
        # output shape = [sequence_length, batch_size,
        #                  hidden_dimension]
        # hidden_state shape = [1, batch_size,
        #                       hidden_dimension]
        final_output = self.fc_layer(
            hidden_state[-1,:,:].squeeze(0))
        return final_output
```

The embedding layer's functionality is provided under the `nn.Embedding` module, which stores word embeddings (in the form of a lookup table) and retrieves them using indices. In this exercise, we set the `embedding` dimension to `100`. This implies that if we have a total of 1,000 words in our vocabulary, then the embeddings lookup table will be 1000x100 in size.

For example, the word *it*, which is tokenized as number *8* in our vocabulary, will be stored as a vector of size 100 at the 8th row in this lookup table. You can initialize the embeddings lookup table with pre-trained embeddings for better performance, but we will be training it from scratch in this exercise.

2. In the following code, we are instantiating the RNN model:

```
input_dimension = len(vocab_to_token)+1
# +1 to account for padding
embedding_dimension = 100
hidden_dimension = 32
output_dimension = 1
rnn_model = RNN(input_dimension, embedding_dimension,
                hidden_dimension, output_dimension)
optim = optim.Adam(rnn_model.parameters())
loss_func = nn.BCEWithLogitsLoss()
rnn_model = rnn_model.to(device)
loss_func = loss_func.to(device)
```

We use the nn.BCEWithLogitsLoss module to compute losses. This PyTorch module provides a numerically stable computation of a **sigmoid** function, followed by a **binary cross-entropy** function, which is exactly what we want as a loss function for our binary classification problem. The hidden dimension of 32 simply means that each RNN cell (hidden) state will be a vector of size 32.

3. We will also define an accuracy metric to measure the performance of our trained model on the validation set. We will be using simple 0-1 accuracy for this exercise:

```
def accuracy_metric(predictions, ground_truth):
    """
    Returns 0-1 accuracy for the given set
    of predictions and ground truth
    """
    # round predictions to either 0 or 1
    rounded_predictions = \
        torch.round(torch.sigmoid(predictions))
    # convert into float for division
    success = (rounded_predictions == ground_truth).float()
    accuracy = success.sum() / len(success)
    return accuracy
```

4. Once we've completed the model instantiation and metrics definition, we can define the training and validation routines. The code for the training routine is as follows:

```
def train(model, dataloader, optim, loss_func):
    loss = 0
    accuracy = 0
    model.train()
    for sequence, sentiment in dataloader:
        optim.zero_grad()
        preds = model(sequence.T).squeeze()
        loss_curr = loss_func(preds, sentiment)
        accuracy_curr = accuracy_metric(preds, sentiment)
        loss_curr.backward()
        optim.step()
        loss += loss_curr.item()
        accuracy += accuracy_curr.item()
    return loss/len(dataloader), accuracy/len(dataloader)
```

The code for the validation routine is as follows:

```
def validate(model, dataloader, loss_func):
    loss = 0
    accuracy = 0
    model.eval()
    with torch.no_grad():
        for sequence, sentiment in dataloader:
            preds = model(sequence.T).squeeze()
            loss_curr = loss_func(preds, sentiment)
            accuracy_curr = accuracy_metric(preds, sentiment)
            loss += loss_curr.item()
            accuracy += accuracy_curr.item()
    return loss/len(dataloader), accuracy/len(dataloader)
```

5. Finally, we are now ready to train the model:

```
num_epochs = 10
best_validation_loss = float('inf')
for ep in range(num_epochs):
    time_start = time.time()
    training_loss, train_accuracy = train(rnn_model,
                                          train_dataloader,
                                          optim, loss_func)
    validation_loss, validation_accuracy = validate(
        rnn_model, validation_dataloader, loss_func)
    time_end = time.time()
    time_delta = time_end - time_start
    if validation_loss < best_validation_loss:
        best_validation_loss = validation_loss
        torch.save(rnn_model.state_dict(), 'rnn_model.pt')
    print(f'epoch number: {ep+1} | time elapsed: {time_delta}s')
    print(f'training loss: {training_loss:.3f} | training accuracy:
{train_accuracy*100:.2f}%')
    print(f'\tvalidation loss: {validation_loss:.3f} |  validation
accuracy: {validation_accuracy*100:.2f}%')
```

The output will be as follows:

```
epoch number: 1 | time elapsed: 170.42595100402832s
training loss: 0.614 | training accuracy: 67.31%
validation loss: 1.011 |  validation accuracy: 31.37%

epoch number: 2 | time elapsed: 156.29844784736633s
training loss: 0.540 | training accuracy: 73.79%
validation loss: 0.762 |  validation accuracy: 51.39%
```

```
...
epoch number: 9 | time elapsed: 156.29339694976807s
training loss: 0.212 | training accuracy: 92.17%
validation loss: 1.392 |  validation accuracy: 49.42%

epoch number: 10 | time elapsed: 154.8834547996521s
training loss: 0.179 | training accuracy: 93.62%
validation loss: 1.033 |  validation accuracy: 63.94%
```

The model seems to have learned especially well on the training set through overfitting. The model has 512 layers in the time dimension, which explains why this powerful model can learn the training set quite well. The performance of the validation set starts from a low value but then rises and fluctuates.

6. Let's quickly define a helper function to make a real-time inference on the trained model:

```python
def sentiment_inference(model, sentence):
    model.eval()
    # text transformations
    sentence = sentence.lower()
    sentence = ''.join([c for c in sentence
                       if c not in punctuation])
    tokenized = [vocab_to_token.get(token, 0)
                 for token in sentence.split()]
    tokenized = np.pad(tokenized,
                       (512-len(tokenized), 0), 'constant')
    # model inference
    model_input = torch.LongTensor(tokenized).to(device)
    model_input = model_input.unsqueeze(1)
    pred = torch.sigmoid(model(model_input))
    return pred.item()
```

7. As the last step of this exercise, we will test the performance of this model on some manually entered review texts:

```python
print(sentiment_inference(rnn_model,
                          "This film is horrible"))
print(sentiment_inference(rnn_model,
                          "Director tried too hard but \
                           this film is bad"))
```

```
print(sentiment_inference(rnn_model,
                          "This film will be houseful for weeks"))
print(sentiment_inference(rnn_model,
                          " I just really loved the movie"))
```

The output will be as follows:

```
0.005014493595808744
0.05119464173913002
0.4609886109828949
0.5695606470108032
```

Here, we can see that the model indeed picks up on the notion of positive and negative. Also, it seems to be able to deal with sequences of variable lengths, even if they are all much shorter than 512 words.

In this exercise, we have trained a rather simple RNN model that has limitations not only on the model architecture aspect but also on the data processing side. In the next exercise, we will use a more evolved recurrent architecture – a bidirectional LSTM model – to tackle the same task. We will use some regularization methods to overcome the problem of overfitting that we observed in this exercise. Moreover, we will use PyTorch's `torchtext` module to handle the data loading and processing pipelines more efficiently and concisely.

Building a bidirectional LSTM

So far, we have trained and tested a simple RNN model on the sentiment analysis task, which is a binary classification task based on textual data. In this section, we will try to improve our performance on the same task by using a more advanced recurrent architecture – LSTMs.

LSTMs, as we know, are more capable of handling longer sequences due to their memory cell gates, which help retain important information from several time steps before and forget irrelevant information even if it was recent. With the exploding and vanishing gradients problem in check, LSTMs should be able to perform well when processing long movie reviews.

Moreover, we will be using a bidirectional model as it broadens the context window at any time step for the model to make a more informed decision about the sentiment of the movie review. The RNN model we looked at in the previous exercise overfitted the dataset during training, so to tackle that, we will be using dropouts as a regularization mechanism in our LSTM model.

Loading and preprocessing the text dataset

In this exercise, we will demonstrate the power of PyTorch's `torchtext` module. In the previous exercise, we roughly dedicated half of the exercise to loading and processing the text dataset. Using `torchtext`, we will do the same in less than 10 lines of code.

Instead of manually downloading the dataset, we will use the pre-existing IMDb dataset under `torchtext.legacy.datasets` to load it. We will also use `torchtext.legacy.data` to tokenize words and generate vocabulary.

Finally, we will use the nn.LSTM module to directly pad sequences instead of manually padding them. Please note that torchtext.legacy is not supported by PyTorch V2, and hence we are using V1.9 for this exercise. The code for this exercise can be found in our GitHub repository [4]. Let's get started:

1. For this exercise, we will need to import a few dependencies. First, we will execute the same import statements as we did for the previous exercise. However, we will also need to import the following:

   ```
   import random
   from torchtext.legacy import (data, datasets)
   ```

   We use the legacy API from torchtext in order to use the Field and LabelField data structures in the next steps, which were discontinued from the main torchtext module since version 0.9.0 [5].

2. Next, we will use the datasets submodule from the torchtext (legacy) module to directly download the IMDb sentiment analysis dataset. We will separate the review texts and the sentiment labels into two separate fields and split the dataset into training, validation, and test sets:

   ```
   TEXT_FIELD = data.Field(tokenize = data.get_tokenizer("basic_english"),
                      include_lengths = True)
   LABEL_FIELD = data.LabelField(dtype = torch.float)
   train_dataset, test_dataset = datasets.IMDB.splits(
       TEXT_FIELD, LABEL_FIELD)
   train_dataset, valid_dataset = train_dataset.split(
       random_state = random.seed(123))
   ```

3. Next, we will use the build_vocab method of torchtext.legacy.data.Field and torchtext.legacy.data.LabelField to build the vocabulary for the movie reviews text dataset and the sentiment labels, respectively:

   ```
   MAX_VOCABULARY_SIZE = 25000
   TEXT_FIELD.build_vocab(train_dataset, max_size = MAX_VOCABULARY_SIZE)
   LABEL_FIELD.build_vocab(train_dataset)
   ```

   As we can see, it takes just three lines of code to build the vocabulary using the predefined functions.

4. Before we get into the model-related details, we will also create dataset iterators for the training, validation, and test sets.

Now that we've loaded and processed the dataset and derived the dataset iterators, let's create the LSTM model object and train the LSTM model.

Instantiating and training the LSTM model

In this section, we will instantiate the LSTM model object. We will then define the optimizer, the loss function, and the model training performance metrics. Finally, we will run the model training loop using the defined model training and model validation routines. Let's get started:

1. First, we must instantiate the bidirectional LSTM model with dropout. While most of the model instantiation looks the same as in the previous exercise, the following line of code is the key difference:

```
self.lstm_layer = nn.LSTM(
    embedding_dimension, hidden_dimension, num_layers=1,
    bidirectional=True, dropout=dropout)
```

2. We have added two special types of tokens – `unknown_token` (for words that do not exist in our vocabulary) and `padding_token` (for tokens that are just added for padding the sequence) – to our vocabulary. Hence, we will need to set the embeddings to all zeros for these two tokens:

```
UNK_INDEX = TEXT_FIELD.vocab.stoi[TEXT_FIELD.unk_token]
lstm_model.embedding_layer.weight.data[UNK_INDEX] = \
    torch.zeros(EMBEDDING_DIMENSION)
lstm_model.embedding_layer.weight.data[PAD_INDEX] = \
    torch.zeros(EMBEDDING_DIMENSION)
```

3. Next, we will define the optimizer (*Adam*) and the loss function (*sigmoid* followed by *binary cross-entropy*). We will also define an accuracy metric calculation function, as we did in the previous exercise.
4. We will then define the training and validation routines.
5. Finally, we will run the training loop with 10 epochs. This should output the following:

```
epoch number: 1 | time elapsed: 1547.8699600696564s
training loss: 0.686 | training accuracy: 54.57%
validation loss: 0.666 |  validation accuracy: 60.02%

epoch number: 2 | time elapsed: 1537.510666847229s
training loss: 0.650 | training accuracy: 61.54%
validation loss: 0.607 |  validation accuracy: 68.02%
...
epoch number: 9 | time elapsed: 1367.163740158081s
training loss: 0.526 | training accuracy: 73.12%
validation loss: 0.549 |  validation accuracy: 76.29%

epoch number: 10 | time elapsed: 1369.546238899231s
```

```
training loss: 0.430 | training accuracy: 80.90%
validation loss: 0.556 |  validation accuracy: 75.66%
```

As we can see, the model is learning well as the epochs progress. Also, dropout seems to control overfitting as both the training and validation set accuracies are increasing at a similar pace. However, compared to RNNs, LSTMs are slower to train. As we can see, the epoch time for LSTMs is roughly 9 to 10 times that of RNNs. This is also because we are using a bidirectional network in this exercise.

6. The previous step also saves the best-performing model. In this step, we will load the best-performing model and evaluate it on the test set:

```
lstm_model.load_state_dict(torch.load('lstm_model.pt'))
test_loss, test_accuracy = validate(
    lstm_model, test_data_iterator, loss_func)
print(f'test loss: {test_loss:.3f} | test accuracy: {test_
accuracy*100:.2f}%')
```

This should output the following:

```
test loss: 0.561 | test accuracy: 76.31%
```

7. Finally, we will define a sentiment inference function, as we did in the previous exercise, and run some manually entered movie reviews against the trained model:

```
print(sentiment_inference(rnn_model, "This film is horrible"))
print(sentiment_inference(rnn_model,\
                          "Director tried too hard \
                           but this film is bad"))
print(sentiment_inference(rnn_model, \
                          "This film will be houseful \
                           for weeks"))
print(sentiment_inference(rnn_model, \
                          "I just really loved the movie"))
```

This should output the following:

```
0.14579516649246216
0.03841548413038254
0.6569563150405884
0.8203923106193542
```

Clearly, the LSTM model has outperformed the RNN models in terms of performance on the validation set. Dropout helped to prevent overfitting, and the bidirectional LSTM architecture seems to have learned the sequential patterns in the movie review text sentences.

The previous two exercises have both been about a many-to-one type sequence task, where the input is a sequence and the output is a binary label. These two exercises, together with the one-to-many exercise in *Chapter 3*, *Combining CNNs and LSTMs*, should have provided you with enough context to get hands-on with different recurrent architectures using PyTorch.

In the next and final section, we will briefly discuss GRUs and how to use them in PyTorch. Then, we will introduce the concept of attention and how it is used in recurrent architectures.

Discussing GRUs and attention-based models

In the final section of this chapter, we will briefly look at GRUs, how they are similar yet different from LSTMs, and how to initialize a GRU model using PyTorch. We will also look at attention-based RNNs. We will conclude this section by describing how attention-only-based models (no recurrence or convolutions) outperform the recurrent family of neural models when it comes to sequence modeling tasks.

GRUs and PyTorch

As we discussed in the *Exploring the evolution of recurrent networks* section, GRUs are a type of memory cell with two gates – a reset gate and an update gate – as well as one hidden state vector. In terms of configuration, GRUs are simpler than LSTMs and yet equally effective in dealing with the exploding and vanishing gradients problem. Tons of research has been done to compare the performance of LSTMs and GRUs. While both perform better than the simple RNNs on various sequence-related tasks, one is slightly better than the other on some tasks and vice versa.

GRUs train faster than LSTMs, and on many tasks such as language modeling, GRUs can perform as well as LSTMs with much less training data. However, theoretically, LSTMs are supposed to retain information from longer sequences than GRUs. PyTorch provides the **nn.GRU** module to instantiate a GRU layer in one line of code. The following code creates a deep GRU network with two bidirectional GRU layers, each with 80% recurrent dropout:

```
self.gru_layer = nn.GRU(input_size, hidden_size,num_layers=2, dropout=0.8,
bidirectional=True)
```

As we can see, it takes one line of code to get started with a PyTorch GRU model. I encourage you to plug the `gru` layer instead of the `lstm` layer or `rnn` layer into the previous exercises and see how it impacts the model training time, as well as model performance.

Attention-based models

The models we have discussed in this chapter have been pathbreaking in solving problems related to sequential data. However, in 2017, a novel attention-only-based approach was invented that subsequently took the shine off these recurrent networks. The concept of attention is derived from the idea of how we, as humans, pay different levels of attention to different parts of a sequence (say, text) at different times.

For example, if we were to complete the statement *Martha sings beautifully, I am hooked to ___ voice.*, we would pay more attention to the word *Martha* to guess that the missing word might be *her*. On the other hand, if we were to complete the statement *Martha sings beautifully, I am hooked to her ____.*, then we would pay more attention to the word *sings* to guess that the missing word is either *voice*, *songs*, *singing*, and so on.

In all our recurrent architectures, a mechanism for focusing on specific parts of the sequence in order to predict the output at the current time step does not exist. Instead, the recurrent models can only get a summary of the past sequence in the form of a condensed hidden state vector.

Attention-based recurrent networks were the first ones to exploit the concept of attention around the years 2014-2015. In these models, an additional attention layer was added on top of the usual recurrent layer. This attention layer learned attention weights for each of the preceding words in the sequence.

A context vector was computed as an attention-weighted average of all the preceding words' hidden state vectors. This context vector was fed to the output layer, in addition to the regular hidden state vector at any time step, *t*. The following diagram shows the architecture of an attention-based RNN:

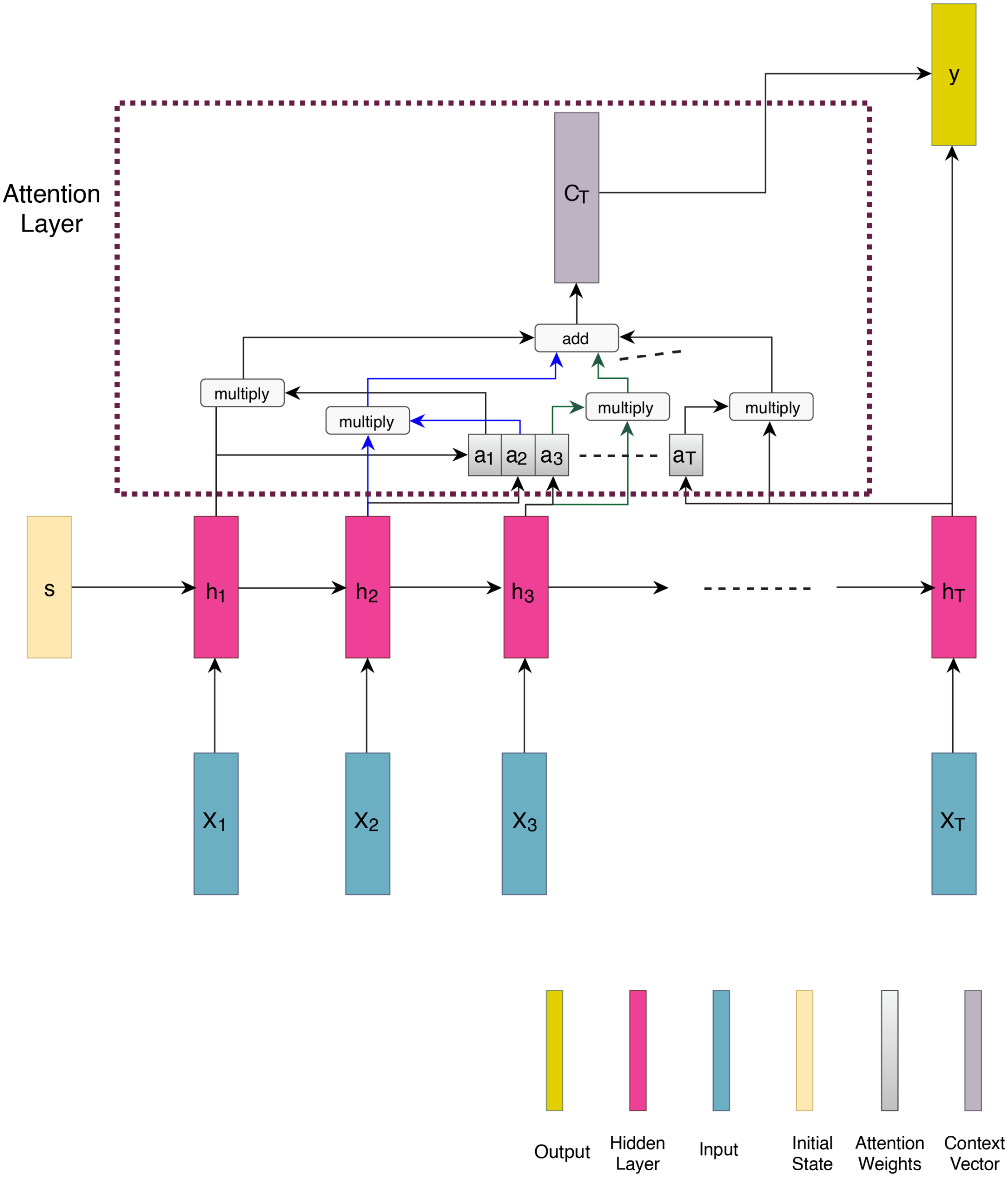


Figure 4.11: Attention-based RNN

In this architecture, a global context vector is calculated at each time step. Variants of this architecture were then devised using a local context vector – not paying attention to all the preceding words but only *k* previous words. Attention-based RNNs outperformed the state-of-the-art recurrent models on tasks such as machine translation.

A couple of years later, in 2017, the *Attention Is All You Need* paper demonstrated the ability to solve sequential tasks solely with the use of an attention mechanism without requiring the recurrent layers. Over the last few years, such models using attention have outperformed recurrent models on various tasks and have helped make immense progress in the field of **Natural Language Processing** (**NLP**) and deep learning in general. Recurrent networks need to be unrolled in time, which makes them non-parallelizable. However, a new model called the **transformer** model, which we will discuss in the next chapter, has no recurrent (and convolutional) layers, making it both parallelizable and lightweight (in terms of computation flops).

Summary

In this chapter, we have extensively explored recurrent neural architectures. In the next chapter, we will elaborate on transformers and other such model architectures, which are neither purely recurrent nor convolutional yet have achieved state-of-the-art results.

References

1. Martin Arjovsky, Amar Shah, and Yoshua Bengio. 2016. *Unitary Evolution Recurrent Neural Networks*: `https://arxiv.org/pdf/1511.06464.pdf`
2. The book's GitHub repository link for training RNNs for sentiment analysis: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter04/rnn.ipynb`
3. IMDB sentiment analysis data: `https://ai.stanford.edu/~amaas/data/sentiment/`
4. The book's GitHub repository link for building a bidirectional LSTM: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter04/lstm.ipynb`
5. `Torchtext` 0.90 release notes: `https://github.com/pytorch/text/releases/tag/v0.9.0-rc5`

Learn more on Discord

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

`https://packt.link/mastorch`

5

Advanced Hybrid Models

In the previous three chapters, we learned extensively about the various convolutional and recurrent network architectures available, along with their implementations in PyTorch. In this chapter, we will take a look at some other deep learning model architectures that have proven to be successful on various machine learning tasks and are neither purely convolutional nor recurrent in nature. We will continue from where we left off in both *Chapter 2*, *Deep CNN Architectures*, and *Chapter 4*, *Deep Recurrent Model Architectures*.

First, we will explore transformers, which, as we have learnt toward the end of *Chapter 4*, *Deep Recurrent Model Architectures*, have outperformed recurrent architectures on various sequential tasks (including LLMs), and have lately become the de-facto AI model for all kinds of tasks (multimodal models, generative AI, etc.). Then, we will pick up from the **EfficientNets** discussion at the end of *Chapter 2*, *Deep CNN Architectures*, and explore the idea of generating randomly wired neural networks, also known as **RandWireNNs.**

In this chapter, we aim to conclude our discussion of different kinds of neural network architectures in this book. After completing this chapter, you will have a detailed understanding of transformers and how to apply these powerful models to sequential tasks using PyTorch. Furthermore, by building your own RandWireNN model, you will have hands-on experience in performing a neural architecture search in PyTorch. This chapter is broken down into the following topics:

- Building a transformer model for language modeling
- Developing a RandWireNN model from scratch

All relevant code for this chapter can be found at: `https://github.com/arj7192/MasteringPyTorchV2/tree/main/Chapter05`.

Building a transformer model for language modeling

In this section, we will explore what transformers are and build one using PyTorch for the task of language modeling. We will also learn how to use some advanced transformer-based models, such as **BERT** and **GPT**, via PyTorch's pretrained model repository. The pretrained model repository contains PyTorch models trained on general tasks such as language modeling (predicting the next word given the sequence of preceding words). These pretrained models can then be fine-tuned for specific tasks such as sentiment analysis (whether a given piece of writing is positive, negative or neutral). Before we start building a transformer model, let's quickly recap what language modeling is.

Reviewing language modeling

Language modeling is the task of figuring out the probability of the occurrence of a word or a sequence of words that should follow a given sequence of words. For example, if we are given *French is a beautiful* _____ as our sequence of words, what is the probability that the next word will be *language* or *word*, and so on? These probabilities are computed by modeling the language using various probabilistic and statistical techniques. The idea is to observe a text corpus and learn the grammar by learning which words occur together and which words never occur together. This way, a language model establishes probabilistic rules around the occurrence of different words or sequences, given various different sequences.

Recurrent models have been a popular way of learning a language model. However, as with many sequence-related tasks, transformers have outperformed recurrent networks on this task as well. We will implement a transformer-based language model for the English language by training it on a text corpus based on articles from the Wall Street Journal.

Now, let's start training a transformer for language modeling. During this exercise, we will demonstrate only the most important parts of the code. The full code can be accessed in our GitHub repository [1].

We will delve deeper into the various components of the transformer architecture in-between the exercise.

For this exercise, we will need to import a few dependencies. One of the important `import` statements is listed here:

```
from torch.nn import TransformerEncoder, TransformerEncoderLayer
```

Besides importing the regular `torch` dependencies, we must import some modules specific to the transformer model; these are provided directly under the `torch` library. We'll also import `torchtext` in order to download a text dataset directly from the available datasets under `torchtext.datasets`.

In the next section, we will define the transformer model architecture and look at the details of the model's components.

Understanding the transformer model architecture

This is perhaps the most important step of this exercise. Here, we define the architecture of the transformer model.

First, let's briefly discuss the model architecture and then look at the PyTorch code for defining the model. *Figure 5.1* shows the model architecture:

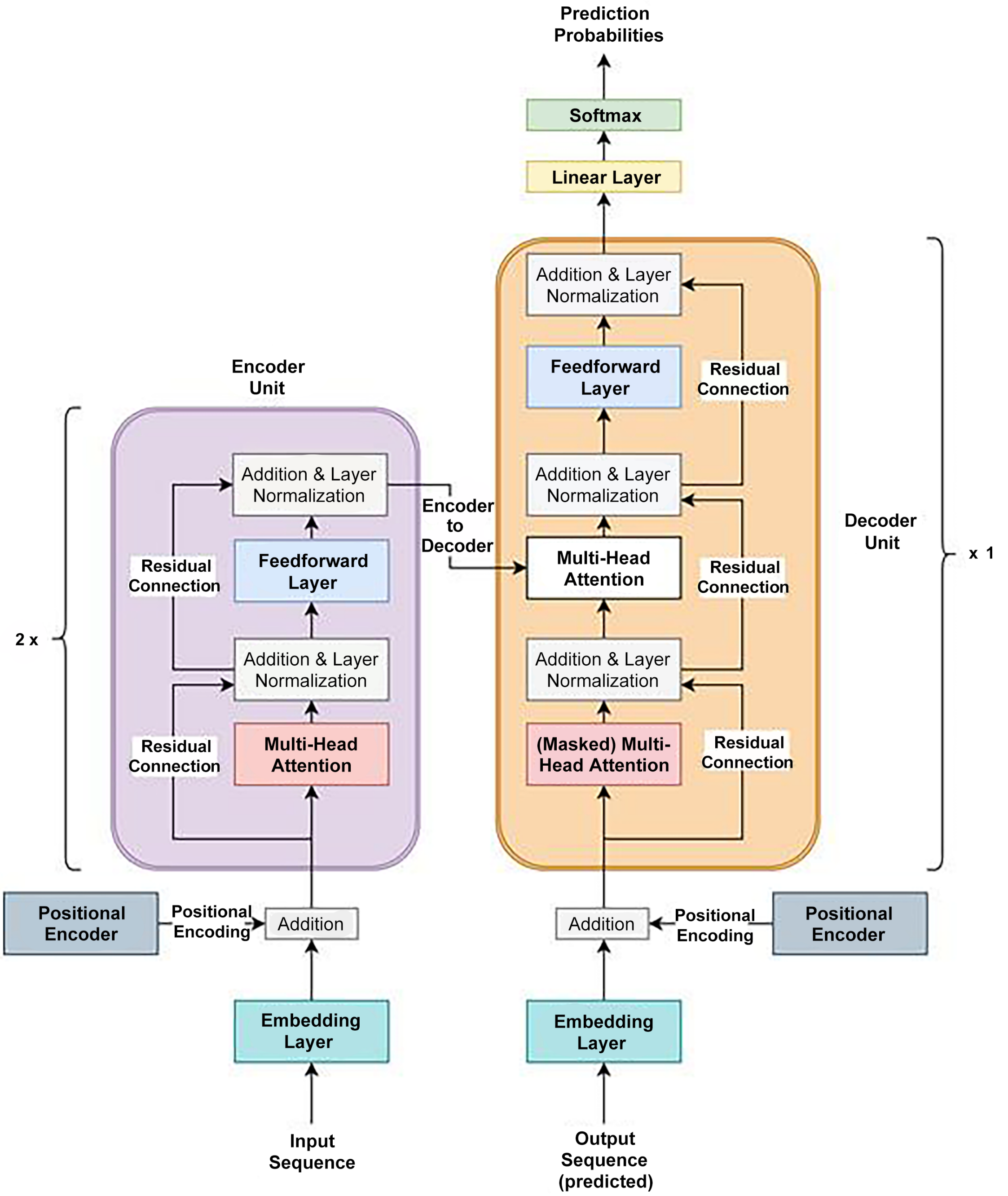


Figure 5.1: Transformer model architecture

The first thing to notice is that this is essentially an encoder-decoder-based architecture, with the **Encoder Unit** on the left (in purple) and the **Decoder Unit** (in orange) on the right. The encoder and decoder units can be tiled multiple times for even deeper architectures. In our example, we have two cascaded encoder units and a single decoder unit. This encoder-decoder setup essentially means that the encoder takes a sequence as input and generates as many embeddings as there are words in the input sequence (that is, one embedding per word). These embeddings are then fed to the decoder, along with the predictions made thus far by the model.

Let's walk through the various layers in this model:

- **Embedding Layer:** This layer is simply meant to perform the traditional task of converting each input word of the sequence into a vector of numbers – that is, an embedding. As always, here, we use the `torch.nn.Embedding` module to code this layer.
- **Positional Encoder:** Note that transformers do not have any recurrent layers in their architecture, yet they outperform recurrent networks on sequential tasks. How? Using a neat trick known as *positional encoding*, the model is provided a sense of sequentiality or sequential-order in the data. Basically, vectors that follow a particular sequential pattern are added to the input word embeddings.

  These vectors are generated in a way that enables the model to understand that the second word comes after the first word and so on. The vectors are generated using the `sinusoidal` and `cosinusoidal` functions to represent a systematic periodicity and distance between subsequent words, respectively. The implementation of this layer for our exercise is as follows:

```
class PosEnc(nn.Module):
    def __init__(self, d_m, dropout=0.2, size_limit=5000):
        # d_m is same as the dimension of the embeddings
        pos = torch.arange(size_limit, dtype=torch.float).unsqueeze(1)
        divider = torch.exp(
            torch.arange(0, d_m, 2).float() * (
                -torch.log(10000.0) / d_m))
        '''divider is the list of radians, multiplied by
           position indices of words, and fed to the
           sinusoidal and cosinusoidal function.'''
        p_enc[:, 0, 0::2] = torch.sin(pos * divider)
        p_enc[:, 0, 1::2] = torch.cos(pos * divider)
    def forward(self, x):
        return self.dropout(x + self.p_enc[:x.size(0)])
```

  As you can see, the `sinusoidal` and `cosinusoidal` functions are used alternately to give the sequential pattern. There are many ways to implement positional encoding though. Without a positional encoding layer, the model will be clueless about the order of the words.

- **Multi-Head Attention:** Before we look at the multi-head attention layer, let's first understand what a **self-attention layer** is. We covered the concept of attention in *Chapter 4*, *Deep Recurrent Model Architectures*, with respect to recurrent networks. Here, as the name suggests, the attention mechanism is applied to self – that is, each word of the sequence. Each word embedding of the sequence goes through the self-attention layer and produces an individual output that is exactly the same length as the word embedding. *Figure 5.2* describes the process of this in detail:

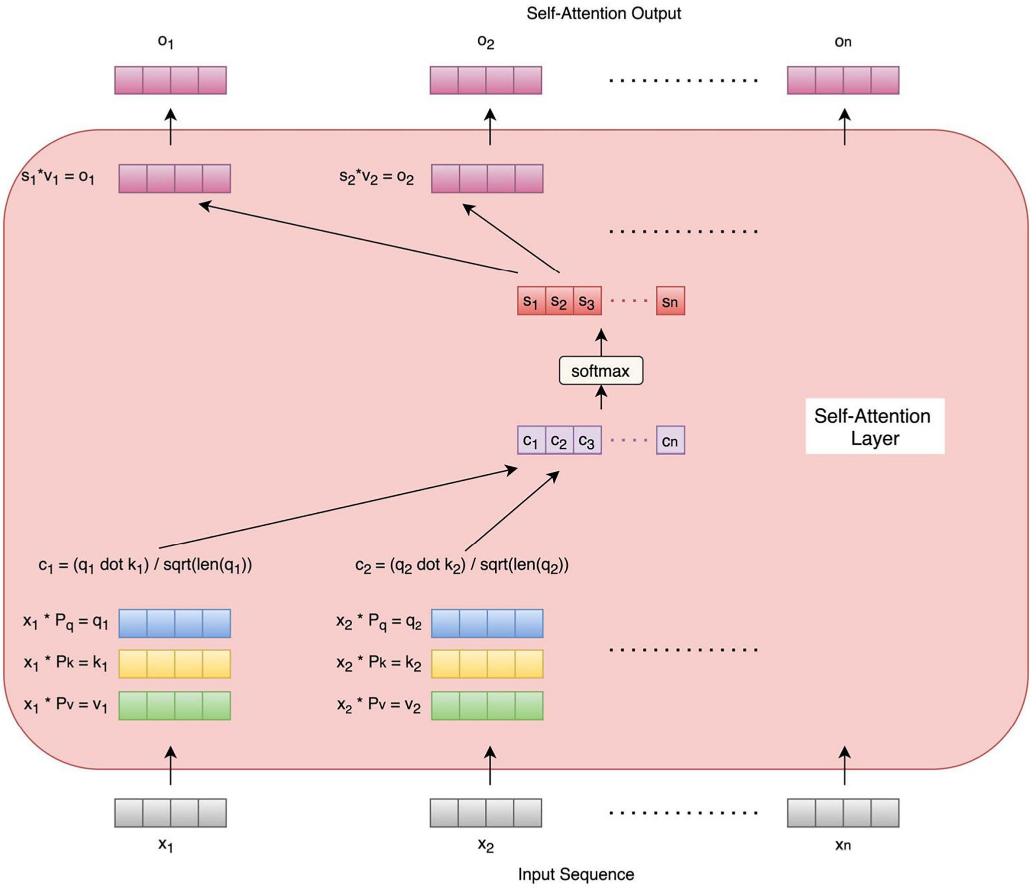


Figure 5.2: Self-attention layer

As we can see, for each word, three vectors are generated through three learnable parameter matrices (P_q, P_k, and P_v). The three vectors are query, key, and value vectors. The query and key vectors are dot-multiplied to produce a number for each word. These numbers are normalized by dividing the square root of the key vector length for each word. The resultant numbers for all words are then Softmaxed at the same time to produce probabilities that are finally multiplied by the respective value vectors for each word. This results in one output vector for each word of the sequence, with the lengths of the output vector and the input word embedding being the same.

A multi-head attention layer is an extension of the self-attention layer where multiple self-attention modules compute outputs for each word. These individual outputs are concatenated and matrix-multiplied with yet another parameter matrix (P_m) to generate the final output vector, whose length is equal to the input embedding vector's. The following diagram shows the multi-head attention layer, along with two self-attention units that we will be using in this exercise:

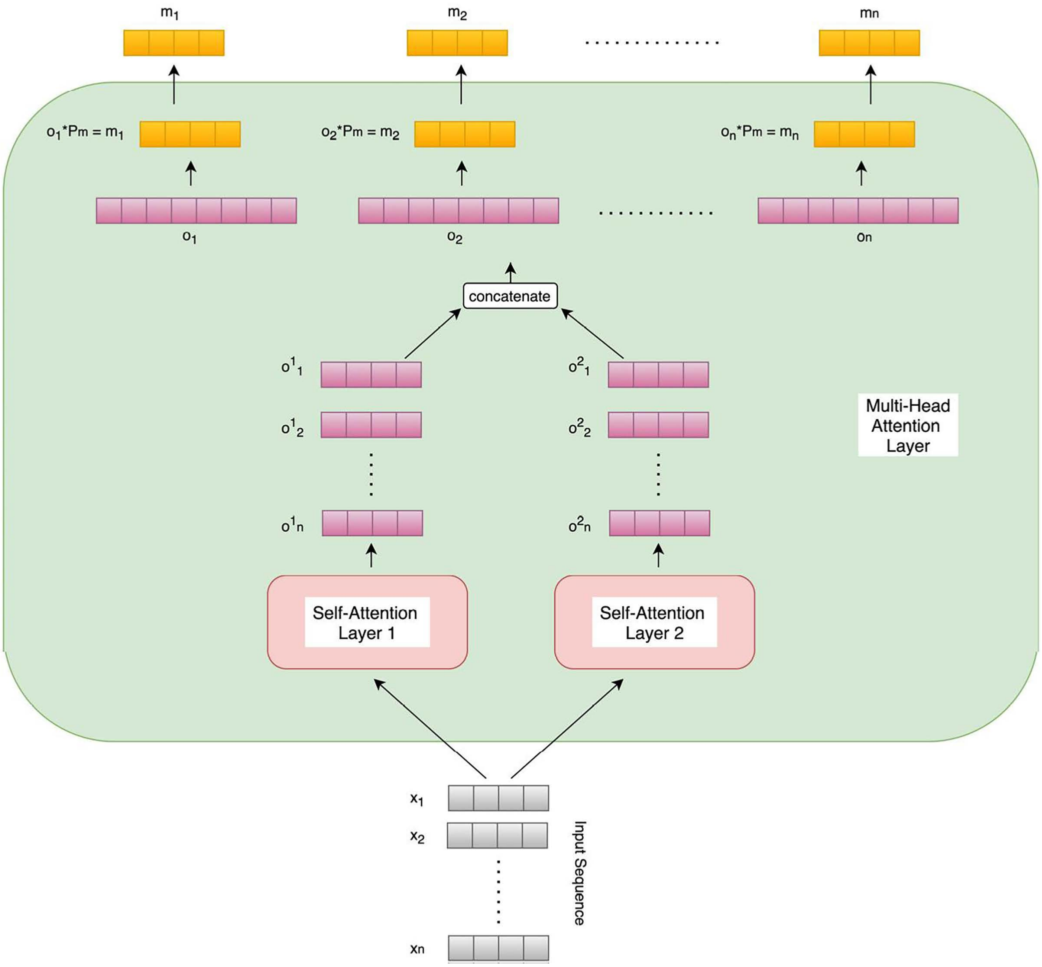


Figure 5.3: Multi-head attention layer with two self-attention units

Having multiple self-attention heads helps different heads focus on different aspects of the sequence of words, similar to how different feature maps learn different patterns in a convolutional neural network. Due to this, the multi-head attention layer performs better than an individual self-attention layer and will be used in our exercise.

Also, note that the masked multi-head attention layer in the decoder unit works in exactly the same way as a multi-head attention layer, except for the added masking – that is, given the time step *t* of processing the sequence, all words from *t+1* to *n* (the length of the sequence) are masked/hidden.

During training, the decoder is provided with two types of inputs. On one hand, it receives query and key vectors from the final encoder as inputs to its (unmasked) multi-head attention layer, where these query and key vectors are matrix transformations of the final encoder output. On the other hand, the decoder receives its own predictions from previous time steps as sequential input to its masked multi-head attention layer.

- **Addition** and **Layer Normalization:** We discussed the concept of a residual connection in *Chapter 2*, *Deep CNN Architectures*, while discussing ResNets. In *Figure 5.1*, we can see that there are residual connections across the addition and layer normalization layers. In each instance, a residual connection is established by directly adding the input word-embedding vector to the output vector of the multi-head attention layer. This helps with easier gradient flow throughout the network and avoids problems with exploding and vanishing gradients. Also, it helps with efficiently learning identity functions across layers.

  Furthermore, layer normalization is used as a normalization trick. Here, we normalize each feature independently so that all the features have a uniform mean and standard deviation. Please note that these additions and normalizations are applied individually to each word vector of the sequence at each stage of the network.

- **Feedforward Layer**: Within both the encoder and decoder units, the normalized residual output vectors for all the words of the sequence are passed through a common feedforward layer. Due to there being a common set of parameters across words, this layer helps with learning broader patterns across the sequence.
- **Linear** and **Softmax Layer:** So far, each layer is outputting a sequence of vectors, one per word. For our task of language modeling, we need a single final output. The linear layer transforms the sequence of vectors into a single vector whose size is equal to the length of our word vocabulary. The **Softmax** layer converts this output into a vector of probabilities summing to 1. These probabilities are the probabilities that the respective words (in the vocabulary) occur as the next words in the sequence.

Now that we have elaborated on the various elements of a transformer model, let's look at the PyTorch code to instantiate the model.

Defining a transformer model in PyTorch

Using the architecture details described in the previous section, we will now write the necessary PyTorch code to define a transformer model, as follows:

```
class Transformer(nn.Module):
    def __init__(self, num_token, num_inputs, num_heads, num_hidden,
                 num_layers, dropout=0.3):
        self.position_enc = PosEnc(num_inputs, dropout)
        layers_enc = TransformerEncoderLayer(
            num_inputs, num_heads, num_hidden, dropout)
        self.enc_transformer = TransformerEncoder(
            layers_enc, num_layers)
        self.enc = nn.Embedding(num_token, num_inputs)
        self.num_inputs = num_inputs
        self.dec = nn.Linear(num_inputs, num_token)
```

As we can see, in the `__init__` method of the class, thanks to PyTorch's `TransformerEncoder` and `TransformerEncoderLayer` functions, we do not need to implement these ourselves. For our language modeling task, we just need a single output for the input sequence of words. Due to this, the decoder is just a linear layer that transforms the sequence of vectors from an encoder into a single output vector. A position encoder is also initialized using the definition that we discussed earlier.

In the `forward` method, the input is positionally encoded and then passed through the encoder, followed by the decoder:

```
    def forward(self, source):
        source = self.enc(source) * torch.sqrt(self.num_inputs)
        source = self.position_enc(source)
        op = self.enc_transformer(source, self.mask_source)
        op = self.dec(op)
        return op
```

Now that we have defined the transformer model architecture, we shall load the text corpus to train it on.

Loading and processing the dataset

In this section, we will discuss the steps related to loading a text dataset for our task and making it usable for the model training routine. Let's get started:

1. For this exercise, we will be using texts from the Wall Street Journal, available as the Penn Treebank dataset.

   Dataset citation

   Marcus Mitchell P., Marcinkiewicz Mary Ann, and Santorini Beatrice. 1993. *Building a large annotated corpus of english: The Penn Treebank*: `https://github.com/wojzaremba/lstm/tree/master/data`

   We'll use the functionality of `torchtext` to download the training dataset (available under the `torchtext` datasets) and tokenize its vocabulary:

   ```
   tr_iter = PennTreebank(split='train')
   tkzer = get_tokenizer('basic_english')
   vocabulary = build_vocab_from_iterator(
       map(tkzer, tr_iter), specials=['<unk>'])
   vocabulary.set_default_index(vocabulary['<unk>'])
   ```

2. We will then use the vocabulary to convert raw text into tensors for the training, validation, and testing datasets:

```
def process_data(raw_text):
    numericalised_text = [
        torch.tensor(vocabulary(tkzer(text)),
                     dtype=torch.long) for text in raw_text]
    return torch.cat(
        tuple(filter(lambda t: t.numel() > 0,
                     numericalised_text)))
tr_iter, val_iter, te_iter = PennTreebank()
training_text = process_data(tr_iter)
validation_text = process_data(val_iter)
testing_text = process_data(te_iter)
```

3. We'll also define the batch sizes for training and evaluation and declare a batch generation function, as shown here:

```
def gen_batches(text_dataset, batch_size):
    num_batches = text_dataset.size(0) // batch_size
    text_dataset = text_dataset[:num_batches * batch_size]
    text_dataset = text_dataset.view(
        batch_size, num_batches).t().contiguous()
    return text_dataset.to(device)
training_batch_size = 32
evaluation_batch_size = 16
training_data = gen_batches(training_text, training_batch_size)
```

4. Next, we must define the maximum sequence length and write a function that will generate input sequences and output targets for each batch, accordingly:

```
max_seq_len = 64
def return_batch(src, k):
    sequence_length = min(max_seq_len, len(src) - 1 - k)
    sequence_data = src[k:k+sequence_length]
    sequence_label = src[k+1:k+1+sequence_length].reshape(-1)
    return sequence_data, sequence_label
```

Having defined the model and prepared the training data, we will now train the transformer model.

Training the transformer model

In this section, we will define the necessary hyperparameters for model training, define the model training and evaluation routines, and finally, execute the training loop. Let's get started:

1. In this step, we define all the model hyperparameters and instantiate our transformer model. The following code is self-explanatory:

```
num_tokens = len(vocabulary) # vocabulary size
embedding_size = 256 # dimension of embedding layer

# transformer encoder's hidden (feed forward) layer dimension
num_hidden_params = 256

# num of transformer encoder layers within transformer encoder
num_layers = 2

# num of heads in (multi head) attention models
num_heads = 2

# value (fraction) of dropout
dropout = 0.25
loss_func = nn.CrossEntropyLoss()

# learning rate
lrate = 4.0
optim_module = torch.optim.SGD(transformer_model.parameters(), lr=lrate)
sched_module = torch.optim.lr_scheduler.StepLR(
    optim_module, 1.0, gamma=0.88)
transformer_model = Transformer(
    num_tokens, embedding_size, num_heads,
    num_hidden_params, num_layers, dropout).to(device)
```

2. Before starting the model training and evaluation loop, we need to define the training and evaluation routines:

```
def train_model():
    for b, i in enumerate(
        range(0, training_data.size(0) - 1, max_seq_len)):
        train_data_batch, train_label_batch = return_batch(
            training_data, i)
        sequence_length = train_data_batch.size(0)
        # only on last batch
        if sequence_length != max_seq_len:
            mask_source = mask_source[:sequence_length,
                                      :sequence_length]

        op = transformer_model(train_data_batch, mask_source)
        loss_curr = loss_func(op.view(-1, num_tokens), train_label_batch)
        optim_module.zero_grad()
        loss_curr.backward()
torch.nn.utils.clip_grad_norm_(transformer_model.parameters(), 0.6)
optim_module.step()
loss_total += loss_curr.item()
def eval_model(eval_model_obj, eval_data_source):
...
```

3. Finally, we must run the model training loop. For demonstration purposes, we are training the model for 5 epochs, but you are encouraged to run it for longer in order to get better performance:

```
min_validation_loss = float("inf")
eps = 5
best_model_so_far = None
for ep in range(1, eps + 1):
    ep_time_start = time.time()
    train_model()
    validation_loss = eval_model(transformer_model, validation_data)
    if validation_loss < min_validation_loss:
        min_validation_loss = validation_loss
        best_model_so_far = transformer_model
```

This should result in the following output:

```
epoch 1, 100/1000 batches, training loss 8.77, training perplexity
6460.73
epoch 1, 200/1000 batches, training loss 7.30, training perplexity
1480.28
epoch 1, 300/1000 batches, training loss 6.88, training perplexity 969.18
...
epoch 5, 900/1000 batches, training loss 5.19, training perplexity 178.59
epoch 5, 1000/1000 batches, training loss 5.27, training perplexity 193.60

epoch 5, validation loss 5.32, validation perplexity 204.29
```

Besides the cross-entropy loss, the perplexity is also reported. **Perplexity** is a popularly used metric in natural language processing to indicate how well a **probability distribution** (a language model, in our case) fits or predicts a sample. The lower the perplexity, the better the model is at predicting the sample. Mathematically, perplexity is just the exponential of the cross-entropy loss. Intuitively, this metric is used to indicate how perplexed or confused the model is while making predictions.

4. Once the model has been trained, we can conclude this exercise by evaluating the model's performance on the test set:

```
testing_loss = eval_model(best_model_so_far, testing_data)
print(f"testing loss {testing_loss:.2f}, testing perplexity {math.
exp(testing_loss):.2f}")
```

This should result in the following output:

```
testing loss 5.23, testing perplexity 187.45
```

In this exercise, we built a transformer model using PyTorch for the task of language modeling. We explored the transformer architecture in detail and how it is implemented in PyTorch. We used the Penn Treebank dataset and `torchtext` functionalities to load and process the dataset. We then trained the transformer model for 5 epochs and evaluated it on a separate test set. This shall provide us with all the information we need to get started on working with transformers.

Besides the original transformer model, which was devised in 2017, a number of successors have since been developed over the years, especially around the field of language modeling, such as the following:

- **Bidirectional Encoder Representations from Transformers** (**BERT**), 2018
- **Generative Pretrained Transformer** (**GPT**), 2018
- GPT-2, 2019
- **Conditional Transformer Language Model** (**CTRL**), 2019
- Transformer-XL, 2019
- **Distilled BERT** (**DistilBERT**), 2019
- **Robustly optimized BERT pretraining Approach** (**RoBERTa**), 2019
- GPT-3, 2020
- **Text-To-Text-Transfer-Transformer** (**T5**), 2020
- **Language Model for Dialogue Operations** (**LaMDA**), 2021
- **Pathways Language Model** (**PaLM**), 2022
- GPT-3.5 (ChatGPT), 2022
- **Large Language Model Meta AI** (**LLaMA**), 2023
- GPT-4, 2023
- LLaMA-2, 2023
- Grok, 2023
- Gemini, 2023
- Sora, 2024

- Gemini-1.5, 2024
- LLaMA-3, 2024

While we will not cover these models in detail in this chapter, you can nonetheless get started using these models with PyTorch thanks to the `transformers` library, developed by Hugging Face [2]. We will explore Hugging Face in detail in *Chapter 19*, *PyTorch x Hugging Face*. The `transformers` library provides pretrained transformer models for various tasks, such as language modeling, text classification, translation, question-answering, and so on.

Besides the models themselves, it also provides tokenizers for the respective models. For example, if we wanted to use a pretrained BERT model for language modeling, we would need to write the following code once we have installed the `transformers` library:

```
import torch
from transformers import BertForMaskedLM, BertTokenizer
bert_model = BertForMaskedLM.from_pretrained('bert-base-uncased')
token_gen = BertTokenizer.from_pretrained('bert-base-uncased')
ip_sequence = token_gen("I love PyTorch !", return_tensors="pt")["input_ids"]
op = bert_model(ip_sequence, labels=ip_sequence)
total_loss, raw_preds = op[:2]
```

As we can see, it takes just a couple of lines to get started with a BERT-based language model. This demonstrates the power of the PyTorch ecosystem. You are encouraged to explore this with more complex variants, such as *DistilBERT* or *RoBERTa*, using the `transformers` library. For more details, please refer to their GitHub page, which was mentioned previously.

This concludes our exploration of transformers. We did this by both building one from scratch as well as by reusing pretrained models. The invention of transformers in the natural language processing space has a parallel with the ImageNet moment in the field of computer vision, so this is going to be an active area of research. PyTorch will have a crucial role to play in the research and deployment of these types of models.

In the next and final section of this chapter, we will resume the neural architecture search discussions we provided at the end of *Chapter 2*, *Deep CNN Architectures,* where we briefly discussed the idea of generating optimal network architectures. We will explore a type of model where we do not decide what the model architecture will look like, and instead run a network generator that will find an optimal architecture for the given task. The resultant network is called a **randomly wired neural network** (**RandWireNN**), and we will develop one from scratch using PyTorch.

Developing a RandWireNN model from scratch

We discussed EfficientNets in *Chapter 2*, *Deep CNN Architectures*, where we explored the idea of finding the best model architecture instead of specifying it manually. RandWireNNs, or randomly wired neural networks, as the name suggests, are built on a similar concept. In this section, we will study and build our own RandWireNN model using PyTorch.

Understanding RandWireNNs

First, a random graph generation algorithm is used to generate a random graph with a predefined number of nodes. This graph is converted into a neural network by a few definitions being imposed on it, such as the following:

- **Directed:** The graph is restricted to be a directed graph, and the direction of the edge is the direction of the data flow in the equivalent neural network.
- **Aggregation:** Multiple incoming edges to a node (or neuron) are aggregated by weighted sum, where the weights are learnable.
- **Transformation**: Inside each node of this graph, a standard operation is applied: ReLU, followed by 3x3 separable convolution (that is, a regular 3x3 convolution followed by a 1x1 pointwise convolution), followed by batch normalization. This operation is also referred to as a **ReLU-Conv-BN triplet**.
- **Distribution**: Lastly, multiple outgoing edges from each neuron carry a copy of the triplet operation.

One final piece in the puzzle is to add a single input node (source) and a single output node (sink) to this graph to fully transform the random graph into a neural network. Once the graph is realized as a neural network, it can be trained for various machine learning tasks.

In the **ReLU-Conv-BN triplet unit**, the output number of channels/features are the same as the input number of channels/features for repeatability reasons. However, depending on the type of task at hand, you can stage several of these graphs with an increasing number of channels downstream (and decreasing spatial size of the data/images). Finally, these staged graphs can be connected to each other by connecting the sink of one to the source of the other in a sequential manner. You can read about the graph generation algorithm in greater detail in section 3 of this paper: *Exploring Randomly Wired Neural Networks for Image Recognition* [3].

Next, in the form of an exercise, we will build a RandWireNN model from scratch using PyTorch.

Developing RandWireNNs using PyTorch

We will now develop a RandWireNN model for an image classification task. This will be performed on the CIFAR-10 dataset. We will start from an empty model, generate a random graph, transform it into a neural network, train it for the given task on the given dataset, evaluate the trained model, and finally, explore the resulting model that was generated. In this exercise, we will only show the important parts of the code for demonstration purposes. In order to access the full code, visit the book's GitHub repository [1].

Defining a training routine and loading data

In the first subsection of this exercise, we will define the training function that will be called by our model training loop and define our dataset loader, which will provide us with batches of data for training. Let's get started:

1. First, we need to import some libraries. Some of the new libraries that will be used in this exercise are as follows:

    ```
    from torchviz import make_dot
    import networkx as nx
    ```

2. Next, we must define the training routine, which takes in a trained model that can produce prediction probabilities given an RGB input image:

    ```
    def train(model, train_dataloader, optim, loss_func, epoch_num, lrate):
        for training_data, training_label in train_dataloader:
            pred_raw = model(training_data)
            curr_loss = loss_func(pred_raw, training_label)
            training_loss += curr_loss.data
        return training_loss / data_size, training_accuracy / data_size
    ```

3. Next, we define the dataset loader. We will use the `CIFAR-10` dataset for this image classification task, which is a well-known database of 60,000 32x32 RGB images, labeled across 10 different classes containing 6,000 images per class. We will use the `torchvision.datasets` module to directly load the data from the `torch` dataset repository.

    Dataset citation

    Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009.

    The code is as follows:

    ```
    def load_dataset(batch_size):
        train_dataloader = torch.utils.data.DataLoader(
            datasets.CIFAR10('dataset',
                             transform=transform_train_dataset,
                             train=True, download=True),
            batch_size=batch_size, shuffle=True)
        return train_dataloader, test_dataloader
    train_dataloader, test_dataloader = load_dataset(batch_size)
    ```

    This should give us the following output:

    ```
    Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
    dataset/cifar-10-python.tar.gz
    Extracting dataset/cifar-10-python.tar.gz to dataset
    ```

We will now move on to designing the neural network model. For this, we will need to design the randomly wired graph.

Defining the randomly wired graph

In this section, we will define a graph generator in order to generate a random graph that will be later used as a neural network. Let's get started.

As shown in the following code, we must define the random graph generator class:

```
class RndGraph(object):
    def __init__(self, num_nodes, graph_probability,
                 nearest_neighbour_k=4, num_edges_attach=5):
    def make_graph_obj(self):
        graph_obj = nx.random_graphs.connected_watts_strogatz_graph(
            self.num_nodes, self.nearest_neighbour_k,
            self.graph_probability)
        return graph_obj
```

In this exercise, we'll be using a well-known random graph model – the **Watts Strogatz** (**WS**) model. This is one of the three models that was experimented on in the original research paper about RandWireNNs. In this model, there are two parameters:

- The number of neighbors for each node (which should be strictly even), *K*
- A rewiring probability, *P*

First, all the *N* nodes of the graph are organized in a ring fashion and each node is connected to *K/2* nodes to its left and *K/2* to its right. Then, we traverse each node clockwise *K/2* times. At the m^{th} traversal ($0<m<K/2$), the edge between the current node and its m^{th} neighbor to the right is *rewired* with a probability, *P*.

Here, rewiring means that the edge is replaced by another edge between the current node and another node different from itself, as well as the m^{th} neighbor. In the preceding code, the `make_graph_obj` method of our random graph generator class instantiates the WS graph model using the `networkx` library.

In the preceding code, the `make_graph_obj` method of our random graph generator class instantiates the WS graph model using the `networkx` library.

Furthermore, we add a `get_graph_config` method to return the list of nodes and edges in the graph. This will come in handy while we're transforming the abstract graph into a neural network. We will also define some graph saving and loading methods to cache the generated graph, for both reproducibility and efficiency reasons:

```
    def get_graph_config(self, graph_obj):
        return node_list, incoming_edges
    def save_graph(self, graph_obj, path_to_write):
        nx.write_yaml(graph_obj, "./cached_graph_obj/" + path_to_write)
    def load_graph(self, path_to_read):
        return nx.read_yaml("./cached_graph_obj/" + path_to_read)
```

Next, we will work on creating the actual neural network model.

Defining RandWireNN model modules

Now that we have the random graph generator, we need to transform it into a neural network. But before that, we will design some neural modules to facilitate that transformation. Let's get started:

1. Starting from the lowest level of the neural network, first, we will define a separable 2D convolutional layer, as follows:

```
class SepConv2d(nn.Module):
    def __init__(self, input_ch, output_ch, kernel_length=3,
                 dilation_size=1, padding_size=1,
                  stride_length=1, bias_flag=True):
        super(SepConv2d, self).__init__()
        self.conv_layer = nn.Conv2d(
            input_ch, input_ch, kernel_length,
            stride_length, padding_size, dilation_size,
            bias=bias_flag, groups=input_ch)
        self.pointwise_layer = nn.Conv2d(
            input_ch,output_ch, kernel_size=1, stride=1,
            padding=0, dilation=1, groups=1, bias=bias_flag)
    def forward(self, x):
        return self.pointwise_layer(self.conv_layer(x))
```

The separable convolutional layer is a cascade of a regular 3x3 2D convolutional layer, followed by a pointwise 1x1 2D convolutional layer.

Having defined the separable 2D convolutional layer, we can now define the ReLU-Conv-BN triplet unit:

```
class UnitLayer(nn.Module):
    def __init__(self, input_ch, output_ch, stride_length=1):
        self.unit_layer = nn.Sequential(
            nn.ReLU(), SepConv2d(input_ch, output_ch,
                               stride_length=stride_length),
                               nn.BatchNorm2d(output_ch),
                               nn.Dropout(self.dropout))
    def forward(self, x):
        return self.unit_layer(x)
```

As we mentioned earlier, the triplet unit is a cascade of a ReLU layer, followed by a separable 2D convolutional layer, followed by a batch normalization layer. We must also add a dropout layer for regularization.

With the triplet unit in place, we can now define a node in the graph with all of the `aggregation`, `transformation`, and `distribution` functionalities we need, as discussed at the beginning of this exercise:

```
class GraphNode(nn.Module):
    def __init__(self, input_degree, input_ch,
                 output_ch, stride_length=1):
        self.unit_layer = UnitLayer(
            input_ch, output_ch,
            stride_length=stride_length)
    def forward(self, *ip):
        if len(self.input_degree) > 1:
            op = (ip[0] * torch.sigmoid(self.params[0]))
            for idx in range(1, len(ip)):
                op += (ip[idx] * torch.sigmoid(self.params[idx]))
            return self.unit_layer(op)
        else:
            return self.unit_layer(ip[0])
```

In the `forward` method, we can see that if the number of incoming edges to the node is more than `1`, then a weighted average is calculated and these weights are learnable parameters of this node. The triplet unit is applied to the weighted average and the transformed (ReLU-Conv-BN-ed) output is returned.

2. We can now consolidate all of our graph and graph node definitions in order to define a randomly wired graph class, as shown here:

```
class RandWireGraph(nn.Module):
    def __init__(self, num_nodes, graph_prob, input_ch,
                 output_ch, train_mode, graph_name):
        # get graph nodes and in edges
        rnd_graph_node = RndGraph(self.num_nodes, self.graph_prob)
        if self.train_mode is True:
            rnd_graph = rnd_graph_node.make_graph_obj()
            self.node_list, self.incoming_edge_list = \
                rnd_graph_node.get_graph_config(rnd_graph)
        else:
        # define source Node
        self.list_of_modules = nn.ModuleList(
            [GraphNode(self.incoming_edge_list[0],
                       self.input_ch, self.output_ch,
                       stride_length=2)])
        # define the sink Node
```

```
self.list_of_modules.extend(
    [GraphNode(self.incoming_edge_list[n], self.output_ch,
               self.output_ch)
               for n in self.node_list if n > 0])
```

In the `__init__` method of this class, first, an abstract random graph is generated, from this a list of nodes and edges is derived. Using the `GraphNode` class, each abstract node of this abstract random graph is encapsulated as a neuron of the desired neural network. Finally, a source or input node and a sink or an output node are added to the network to make the neural network ready for the image classification task.

The `forward` method is also unconventional, as shown here:

```
    def forward(self, x):
        # source vertex
        op = self.list_of_modules[0].forward(x)
        mem_dict[0] = op
        # the rest of the vertices
        for n in range(1, len(self.node_list) - 1):
            if len(self.incoming_edge_list[n]) > 1:
                op = self.list_of_modules[n].forward(
                    *[mem_dict[incoming_vtx]
                      for incoming_vtx
                      in self.incoming_edge_list[n]])
            mem_dict[n] = op
        for incoming_vtx in range(1, len(
        self.incoming_edge_list[self.num_nodes + 1])):
            op += \
                mem_dict[self.incoming_edge_list
                         [self.num_nodes + 1][incoming_vtx]]
        return op / len(self.incoming_edge_list[self.num_nodes + 1])
```

First, a forward pass is run for the source neuron, and then a series of forward passes are run for the subsequent neurons based on the `list_of_nodes` for the graph. The individual forward passes are executed using `list_of_modules`. Finally, the forward pass through the sink neuron gives us the output of this graph.

Next, we will use these defined modules and the randomly wired graph class to build the actual RandWireNN model class.

Transforming a random graph into a neural network

In the previous step, we defined one randomly wired graph. However, as we mentioned at the beginning of this exercise, a randomly wired neural network consists of several staged randomly wired graphs. The rationale behind that is to have a different (increasing) number of channels/features as we progress from the input neuron to the output neuron in an image classification task.

This would be impossible with just one randomly wired graph because the number of channels is constant through one such graph, by design. Let's get started:

1. In this step, we define the ultimate randomly wired neural network. This will have three randomly wired graphs cascaded next to each other. Each graph will have double the number of channels compared to the previous graph to help us align with the general practice of increasing the number of channels (while downsampling spatially) in an image classification task:

```
class RandWireNNModel(nn.Module):
    def __init__(self, num_nodes, graph_prob,
                 input_ch, output_ch, train_mode):
        self.conv_layer_1 = nn.Sequential(
            nn.Conv2d(in_channels=3,
                      out_channels=self.output_ch,
                      kernel_size=3, padding=1),
            nn.BatchNorm2d(self.output_ch))
        self.conv_layer_2 = …
        self.conv_layer_3 = …
        self.conv_layer_4 = …
        self.classifier_layer = nn.Sequential(
            nn.Conv2d(in_channels=self.input_ch*8,
                      out_channels=1280, kernel_size=1),
                      nn.BatchNorm2d(1280))
        self.output_layer = nn.Sequential(
            nn.Dropout(self.dropout),
            nn.Linear(1280, self.class_num))
```

   The `__init__` method starts with a regular 3x3 convolutional layer, followed by three staged randomly wired graphs with channels that double in terms of numbers. This is followed by a fully connected layer that flattens the convolutional output from the last neuron of the last randomly wired graph into a vector that's `1280` in size.

2. Finally, another fully connected layer produces a 10-sized vector containing the probabilities for the 10 classes, as follows:

```
    def forward(self, x):
        x = self.conv_layer_1(x)
        x = self.conv_layer_2(x)
        x = self.conv_layer_3(x)
        x = self.conv_layer_4(x)
        x = self.classifier_layer(x)
        # global average pooling
        _, _, h, w = x.size()
        x = F.avg_pool2d(x, kernel_size=[h, w])
        x = torch.squeeze(x)
        x = self.output_layer(x)
        return x
```

The `forward` method is quite self-explanatory, besides the global average pooling that is applied right after the first fully connected layer. This helps reduce dimensionality and the number of parameters in the network.

At this stage, we have successfully defined the RandWireNN model, loaded the datasets, and defined the model training routine. Now, we are all set to run the model training loop.

Training the RandWireNN model

In this section, we will set the model's hyperparameters and train the RandWireNN model. Let's get started:

1. We have defined all the building blocks for our exercise. Now, it is time to execute it. First, let's declare the necessary hyperparameters:

   ```
   num_epochs = 5
   graph_probability = 0.7
   node_channel_count = 64
   num_nodes = 16
   lrate = 0.1
   batch_size = 64
   train_mode = True
   ```

2. Having declared the hyperparameters, we instantiate the RandWireNN model, along with the optimizer and loss function:

   ```
   rand_wire_model = RandWireNNModel(
       num_nodes, graph_probability, node_channel_count,
       node_channel_count, train_mode).to(device)
   optim_module = optim.SGD(
       rand_wire_model.parameters(),
       lr=lrate, weight_decay=1e-4, momentum=0.8)
   loss_func = nn.CrossEntropyLoss().to(device)
   ```

3. Finally, we begin training the model. We're training the model for `5` epochs here for demonstration purposes, but you are encouraged to train for longer to see the boost in performance:

   ```
   for ep in range(1, num_epochs + 1):
       epochs.append(ep)
       training_loss, training_accuracy = train(
           rand_wire_model, train_dataloader,
           optim_module, loss_func, ep, lrate)
       test_accuracy = accuracy(rand_wire_model, test_dataloader)
       test_accuracies.append(test_accuracy)
   ```

```
    training_losses.append(training_loss)
    training_accuracies.append(training_accuracy)
    if best_test_accuracy < test_accuracy:
        torch.save(model_state, './model_checkpoint/' \
                   + model_filename + 'ckpt.t7')
    print("model train time: ", time.time() - start_time)
```

This should result in the following output:

```
epoch 1, loss: 1.9863920211791992, accuracy: 25.0
epoch 1, loss: 1.7622356414794922, accuracy: 31.25
epoch 1, loss: 1.6300958395004272, accuracy: 35.9375
...
epoch 5, loss: 1.042513132095337, accuracy: 62.5
test acc: 68.60%, best test acc: 63.73%
model train time:  14912.663238048553
```

It is evident from these logs that the model is progressively learning as the epochs progress. The performance on the validation set seems to be consistently increasing, which indicates model generalizability.

With that, we have created a model with no particular architecture in mind that can reasonably perform the task of image classification on the CIFAR-10 dataset.

Evaluating and visualizing the RandWireNN model

Finally, we will look at this model's test set performance before briefly exploring the model architecture visually. Let's get started:

1. Once the model has been trained, we can evaluate it on the test set:

```
rand_wire_nn_model.load_state_dict(model_checkpoint['model'])
for test_data, test_label in test_dataloader:
    success += pred.eq(test_label.data).sum()
    print(f"test accuracy: {float(success) * 100. / len(
        test_dataloader.dataset)} %")
```

This should result in the following output:

```
best model accuracy: 68.6%, last epoch: 5
test accuracy: 68.6 %
```

The best-performing model was found at the fourth epoch, with over 67% accuracy. Although the model is not perfect yet, we can train it for more epochs to achieve better performance. Also, a random model for this task would perform at an accuracy of 10% (because of 10 equally likely classes), so an accuracy of 67.73% is still promising, especially given the fact that we are using a randomly generated neural network architecture.

2. To conclude this exercise, let's look at the model architecture that was learned. The original image is too large to be displayed here. You can find the full image in our GitHub repository in both `.svg` [4] and `.pdf` [5] format. In the following figure, we have vertically stacked three parts – the input section, a mid-section, and the output section of the original neural network:

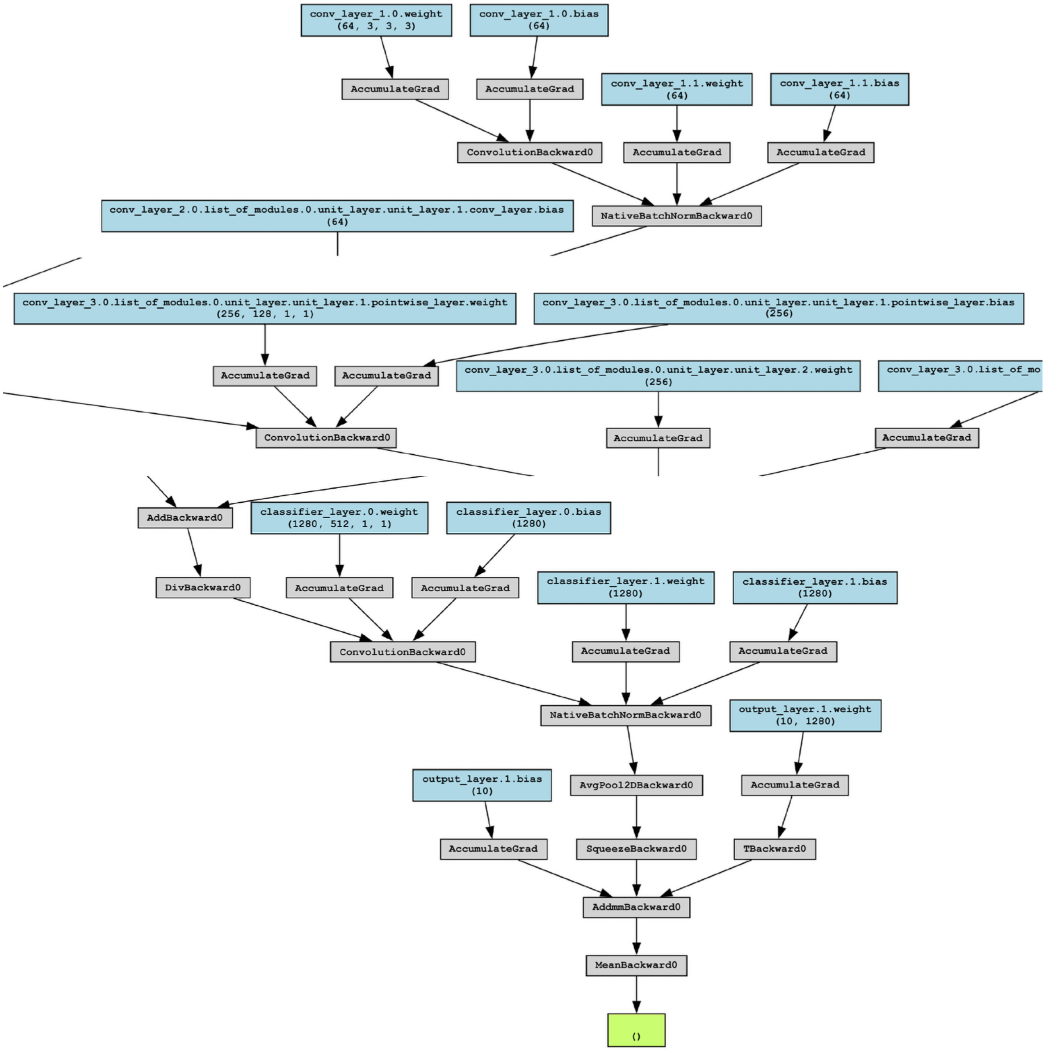

conv_layer_1.0.weight (64, 3, 3, 3)
conv_layer_1.0.bias (64)
AccumulateGrad
AccumulateGrad
conv_layer_1.1.weight (64)
conv_layer_1.1.bias (64)
ConvolutionBackward0
AccumulateGrad
AccumulateGrad
conv_layer_2.0.list_of_modules.0.unit_layer.unit_layer.1.conv_layer.bias (64)
NativeBatchNormBackward0
conv_layer_3.0.list_of_modules.0.unit_layer.unit_layer.1.pointwise_layer.weight (256, 128, 1, 1)
conv_layer_3.0.list_of_modules.0.unit_layer.unit_layer.1.pointwise_layer.bias (256)
AccumulateGrad
AccumulateGrad
conv_layer_3.0.list_of_modules.0.unit_layer.unit_layer.2.weight (256)
conv_layer_3.0.list_of_mo
ConvolutionBackward0
AccumulateGrad
AccumulateGrad
AddBackward0
classifier_layer.0.weight (1280, 512, 1, 1)
classifier_layer.0.bias (1280)
DivBackward0
AccumulateGrad
AccumulateGrad
classifier_layer.1.weight (1280)
classifier_layer.1.bias (1280)
ConvolutionBackward0
AccumulateGrad
AccumulateGrad
NativeBatchNormBackward0
output_layer.1.weight (10, 1280)
output_layer.1.bias (10)
AvgPool2DBackward0
AccumulateGrad
AccumulateGrad
SqueezeBackward0
TBackward0
AddmmBackward0
MeanBackward0
()

Figure 5.4: RandWireNN architecture

From this graph, we can observe the following key points:

- At the top, we can see the beginning of this neural network, which consists of a 64-channel 3x3 2D convolutional layer, followed by a 64-channel 1x1 pointwise 2D convolutional layer.
- In the middle section, we can see the transition between the third- and fourth-stage random graphs, where we can see the sink neuron, `conv_layer_3`, of the stage 3 random graph, followed by the source neuron, `conv_layer_4`, of the stage 4 random graph.

- Lastly, the lowermost section of the graph shows the final output layers – the sink neuron (a 512-channel separable 2D convolutional layer) of the stage 4 random graph, followed by a fully connected flattening layer, resulting in a 1,280-size feature vector, followed by a fully connected softmax layer that produces the 10 class probabilities.

Hence, we have built, trained, tested, and visualized a neural network model for image classification without specifying any particular model architecture. We did specify some overarching constraints over the structure, such as the penultimate feature vector length (`1280`), the number of channels in the separable 2D convolutional layers (`64`), the number of stages in the RandWireNN model (`4`), the definition of each neuron (ReLU-Conv-BN triplet), and so on.

However, we didn't specify what the structure of this neural network architecture should look like. We used a random graph generator to do this for us, which opens up an almost infinite number of possibilities in terms of finding optimal neural network architectures.

Neural architecture search is an ongoing and promising area of research in the field of deep learning. Largely, this fits in well with the field of training custom machine learning models for specific tasks, referred to as AutoML.

AutoML stands for **Automated Machine Learning** as it does away with the necessity of having to manually load datasets, predefine a particular neural network model architecture to solve a given task, and manually deploy models into production systems. In *Chapter 16*, *PyTorch and AutoML*, we will discuss AutoML in detail and learn how to build such systems with PyTorch.

Summary

In this chapter, we looked at two distinct hybrid types of neural networks. First, we looked at the transformer model – the attention-only-based models with no recurrent connections that have outperformed all recurrent models on multiple sequential tasks. We ran through an exercise where we built, trained, and evaluated a transformer model on a language modeling task with the Penn Treebank dataset, using PyTorch. In the second and final section of this chapter, we took up from where we left off in *Chapter 2*, *Deep CNN Architectures*, where we discussed the idea of optimizing for model architectures rather than optimizing for just the model parameters while fixing the architecture. We explored one of the approaches to do that – using **randomly wired neural networks** (**RandWireNNs**) – where we generated random graphs, assigned meanings to the nodes and edges of these graphs, and interconnected these graphs to form a neural network.

Speaking of graphs, in the next chapter, we will learn about yet another class of neural networks that can learn from graph datasets - graph neural networks.

We will solve a classification problem on a graph dataset with graph neural networks implemented using PyTorch.

References

1. GitHub Repository: `https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/Chapter05/transformer.ipynb`
2. Transformers by Hugging Face: `https://huggingface.co/docs/transformers/en/index`
3. Exploring Randomly Wired Neural Networks for Image Recognition: `https://arxiv.org/pdf/1904.01569.pdf`
4. SV image: `https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/Chapter05/randwirenn.svg`
5. PNG image: `https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/Chapter05/randwirenn%5Brepresentational_purpose_only%5D.png`

Learn more on Discord

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

`https://packt.link/mastorch`

6

Graph Neural Networks

In the previous chapters, we have discussed various kinds of neural architectures, ranging from convolutional to recurrent, from attention-based transformers to auto-generated **neural networks** (**NNs**). While these architectures cover a wide range of deep learning problems, they work best with data that exists in a continuous space, typically represented as vectors, or coordinates in a Euclidean space such as **text** (**1D**), **images** (**2D**), and **videos** (**3D**). However, a huge portion of real-world datasets exists in the form of graphs or networks, such as social networks, protein-interaction networks, literature citation networks, and the World Wide Web, to name a few. In this chapter, we'll learn about **graph neural networks** (**GNNs**) – a class of deep learning models that can natively learn patterns from graph structures.

We'll first understand the basic concepts related to graphs and GNNs. Then, we'll explore different types of graph-learning tasks and study a few prominent GNN models. Finally, we'll go hands-on with two coding exercises. First, we'll train a GNN model called a **graph convolutional network** (**GCN**) on a graph dataset using PyTorch Geometric – a library built on top of PyTorch that facilitates the effortless creation and training of GNNs. We'll then iterate on the same graph dataset using a different GNN model called the **graph attention network** (**GAT**).

This chapter is broken down into the following topics:

- Introduction to GNNs
- Types of graph learning tasks
- Reviewing prominent GNN models
- Building a GCN model using PyTorch Geometric
- Training a GAT model with PyTorch Geometric

Introduction to GNNs

To understand GNNs, we will start with reviewing the graph data structure. That said, so what is a *graph*? In computer science, it refers to a data structure containing two components: *nodes* (or vertices) and *edges*, where nodes typically represent things or objects, such as a person, place, or thing, and edges connect the nodes, for example, by representing the relationship between the nodes. *Figure 6.1* shows a graph with persons as nodes and their relationships as edges. Note that the edges are drawn as arrows, which indicates that this is a *directed graph* wherein there is a strict order in relationships (**B** is the parent of **A**, not the other way round). *Undirected graphs*, on the other hand, have edges that can be traversed both ways (think of the *sibling* relationship) and are drawn as straight lines between nodes.

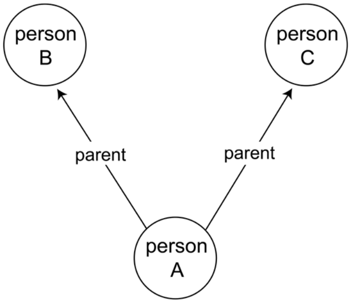


Figure 6.1: Example graph containing people (nodes) and their relationships (edges)

In technical terms, a graph can be represented as

$$G = (V, E)$$

Equation 6.1: Representation of a graph G, in terms of its vertices V, and edges E

In this equation, *V* is the set of vertices (nodes) and *E* is the set of edges between the vertices. We can more concretely represent a graph as an adjacency matrix **A**, where the rows and columns represent graph vertices, and the entries of the matrix indicate whether pairs of vertices are adjacent (connected) or not.

isParent	personA	personB	personC
personA	0	0	0
personB	1	0	0
personC	1	0	0

Table 6.1: Adjacency matrix for a directed graph of people and relationships

Table 6.1 shows the adjacency matrix for the toy graph presented in *Figure 6.1*. If any (*row, column*) entry in this matrix is 1, it means that the *column* is the parent of the *row*. Note that the matrix is not symmetric because it is a directed graph. An undirected graph produces a symmetric adjacency matrix.

Understanding the intuition behind GNNs

In **natural language processing** (**NLP**), the words of a sentence, taken one at a time, might not be useful for training machine learning models on tasks such as sentiment analysis, language translation, and text-to-image generation. We typically need to look at a sequence of several words to understand the sentiment. Similarly, graph data can't be used as is in a machine learning setup. We need to represent the neighborhood of a node in a graph in a meaningful way.

Also, words of a sentence are usually encoded into what we call embedding vectors in such a way that synonymous words have a small distance (cosine distance, for example) and antonyms have a large (cosine) distance between them. These word embedding vectors are then used for specific downstream tasks. With graphs, we need to do something similar – to represent the nodes as embeddings, for example, where the embeddings capture not only the node-specific information but also the information from the neighborhood of a given node, as shown in *Figure 6.2*.

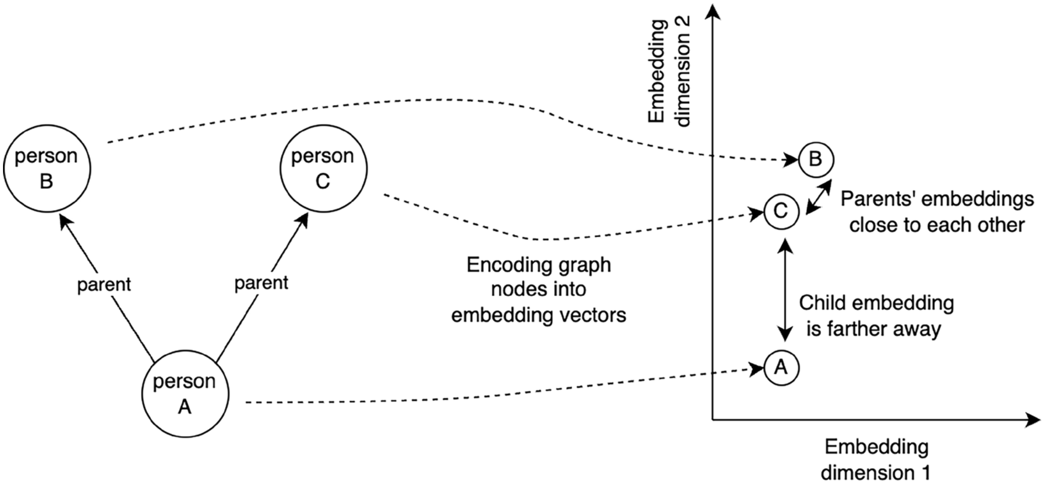

person B
person C
parent
parent
person A
Encoding graph nodes into embedding vectors
Embedding dimension 2
B
C
Parents' embeddings close to each other
Child embedding is farther away
A
Embedding dimension 1

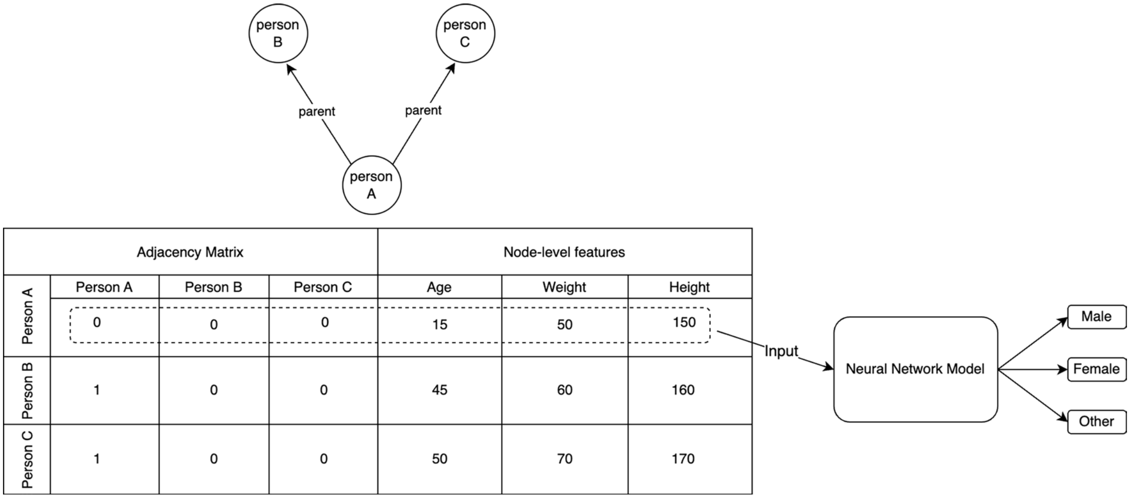
Figure 6.2: Encoding graph vertices into embedding vectors placing similar nodes together and vice versa

As we can see in *Figure 6.2*, embeddings of parent nodes (**B** and **C**) are placed close to each other in the embedding space while child node (**A**) is placed farther away. This is happening because the (hypothetical) encoding function is making use of the graph structure, which reveals the similarity between nodes **B** and **C**, as they are of a similar age, for example. Capturing the graph structure besides the node-specific information is what gave birth to the new class of deep learning models in the form of GNNs, or else regular NNs would have been sufficient. Let us briefly check how we could apply a regular NN model to graph data.

Using regular NNs on graph data – a thought experiment

We can use the adjacency matrix to start with, for representing each graph node. Each row of the adjacency matrix contains the respective node's information about its relationships with the rest of the nodes. Besides the adjacency matrix, each node in itself can be made up of different features.

For example, in the graph presented in *Figure 6.1*, each person might be represented not just by an **ID** (**A**, **B**, or **C**), but as a set of different features such as **ID**, age, gender, height, and weight. Given the adjacent matrix and intrinsic node features, *Figure 6.3* shows how we could use this information to train a regular (feedforward) NN model.



	Adjacency Matrix			Node-level features		
	Person A	Person B	Person C	Age	Weight	Height
Person A	0	0	0	15	50	150
Person B	1	0	0	45	60	160
Person C	1	0	0	50	70	170

Figure 6.3: Using adjacency matrix and node-level features to train a regular NN model for an example task of predicting the gender of a given graph node (person)

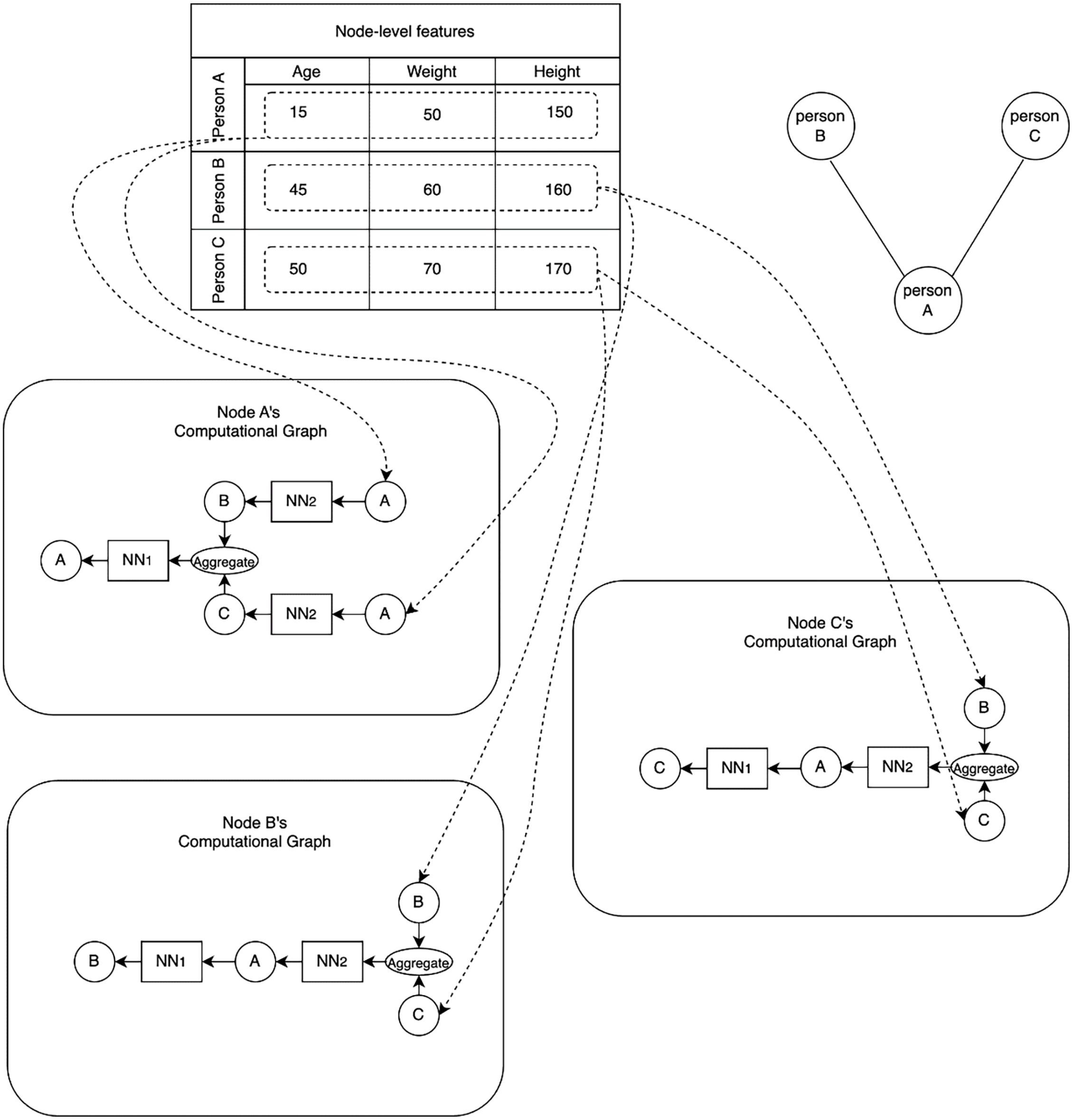
As shown in *Figure 6.3*, we could just concatenate the adjacency matrix data with node-level features to pass as input to a gender prediction NN model. Although it looks like this approach might work for the given task, it has several limitations:

- First, this approach only captures the first-degree graph information around a node in the form of immediate-neighbor adjacency features. In large graph datasets, while first-degree (or direct) neighbors already add value, second or higher-degree neighbors – where nodes are connected via one or more nodes in between respectively – can be crucial in extracting the full extent of graph information.
- Secondly, if we change the ordering of graph nodes from **A**, **B**, **C** to **B**, **C**, **A** in our adjacency matrix, then the adjacency features for node **B** will change from `[1, 0, 0]` to `[0, 0, 1]`. This means that such a feature representation relies on strict node ordering and a model trained in this setup will malfunction when presented with the same graph with a different node ordering.
- Thirdly, if a new node (person) is added to our persons graph, the feature vector length will increase from `6` to `7` for the new, as well as existing nodes, thereby requiring a retraining of the model from scratch for all nodes.
- Lastly, the features obtained from the adjacency matrix are going to be sparse because only one of the entries of this feature vector will have a value of **1**, while the remainder of the features will contain all **0**s.

The preceding limitations make it clear that we need a drastically different solution for graphs and this is where GNNs come in.

Understanding the power of GNNs with computational graphs

To address the first limitation of using regular NNs, we need something more than just the adjacency matrix to represent the complete graph information around a given node – including the first, second, and further degree connections. This can be achieved by using computational graphs, as shown in *Figure 6.4*.



Node-level features			
	Age	Weight	Height
Person A	15	50	150
Person B	45	60	160
Person C	50	70	170

Figure 6.4: Computational graphs (of depth 2) for all 3 nodes of a given undirected graph demonstrating the first and second-degree connections of a given node

Each node in the above image is represented by the respective node-level features and if a node has multiple neighbouring nodes, these neighbour nodes' features are aggregated before being propagated to the given node. The original graph is shown at the top right.

For each node in the graph, we create a computational graph where we represent the node's neighborhood layer-wise – each layer representing a different degree of connection. In *Figure 6.4*, we see that for the given undirected graph, node **A** is immediately connected to nodes **B** and **C**, which are in turn connected back to node **A**. Hence, node **A** has a first-degree connection to nodes **B** and **C** and a second-degree connection to itself.

We limit the depth of computational graphs in this example to 2, but one can generalize this to depth n, where the last layer of such a computational graph will represent the n^{th}-degree connection to the original node.

On node **A**'s computational graph in *Figure 6.4*, we can see that the forward pass starts at the right-most end with the node-level features of node **A**. These features are propagated through an NN (NN_2) to generate latent representations of nodes **B** and **C** that are then aggregated and the aggregation is propagated through yet another NN (NN_1) to generate a latent representation of node **A**. We then use this representation of node **A** (and similarly, nodes **B** and **C** from their computational graphs) as its embedding vector to train for downstream tasks such as gender classification.

Note

The notations NN_1 and NN_2 might look confusing, but they are essentially numbered as per the degree of connection to the original node.

It is important to note that weights are shared between all NNs of a given layer and between all computational graphs. For example, all NN_1s in *Figure 6.4* share the same parameters across all computation graphs. Also, the *convolution* in GCNs (which we discuss in a later section) comes from this shared weights mechanism. This weight sharing prevents the number of overall model parameters from exploding as the number of graph nodes or the depth of the computation graph increases. Weight sharing also makes the system robust against different ordering of nodes or the addition of a new node to the original graph.

Thanks to multi-layer computational graphs, we address the first limitation that we faced using a regular NN for graphs. Thanks to the aggregation of multiple node features at each layer, we address the second limitation of using regular NNs on graphs because the ordering of nodes in the graph doesn't matter anymore. This implicitly means that the aggregation function needs to be order independent, such as sum, average, maximum, and minimum. We cannot use aggregation functions like concatenation. To address the third limitation, where we have to retrain the entire NN when we add a new node, thanks to computational graphs, we only need to create one more computational graph for this newly added node. Finally, we overcome the fourth limitation related to sparsity where we might end up with lots of zeros in the input features for each node, because we do not use the adjacency matrix information with computational graphs and rely only on the intrinsic node features.

We have discussed the core principles of GNNs. The exact aggregation function used at each layer of a computational graph, the exact choice of NN architecture used for NN_1 and NN_2, and the process of training the GNN end to end all vary across different GNN architectures, which we discuss in a later section in this chapter. But for now, let us briefly learn about the different types of learning tasks that can be performed using GNNs on graph data.

Types of graph learning tasks

We have discussed graphs and the basic principles of GNNs but what type of tasks can be performed using GNNs on graph data? (By tasks, we are referring to the downstream tasks mentioned in the previous section.) Broadly, there are three different categories of graph learning tasks:

- Node-level tasks
- Edge-level tasks
- Graph-level tasks

Let us briefly discuss these.

Understanding node-level tasks

Node-level tasks are aimed at predicting the class of a given node within a graph. Our demonstrations in the previous sections (in *Figure 6.3*) are based on node-level tasks where the task was to predict the gender of each node – male, female, or other. For such tasks, the latent feature representations of each node (final output of the node's computational graph) are used to train a downstream task.

This task can be:

- **A classification task** – involving discrete labels (gender, color, etc.)
- **A regression task** – involving continuous values (age, height, etc.)
- **A clustering task** – where nodes of a graph are segregated into clusters

All of this is shown in *Figure 6.5*.

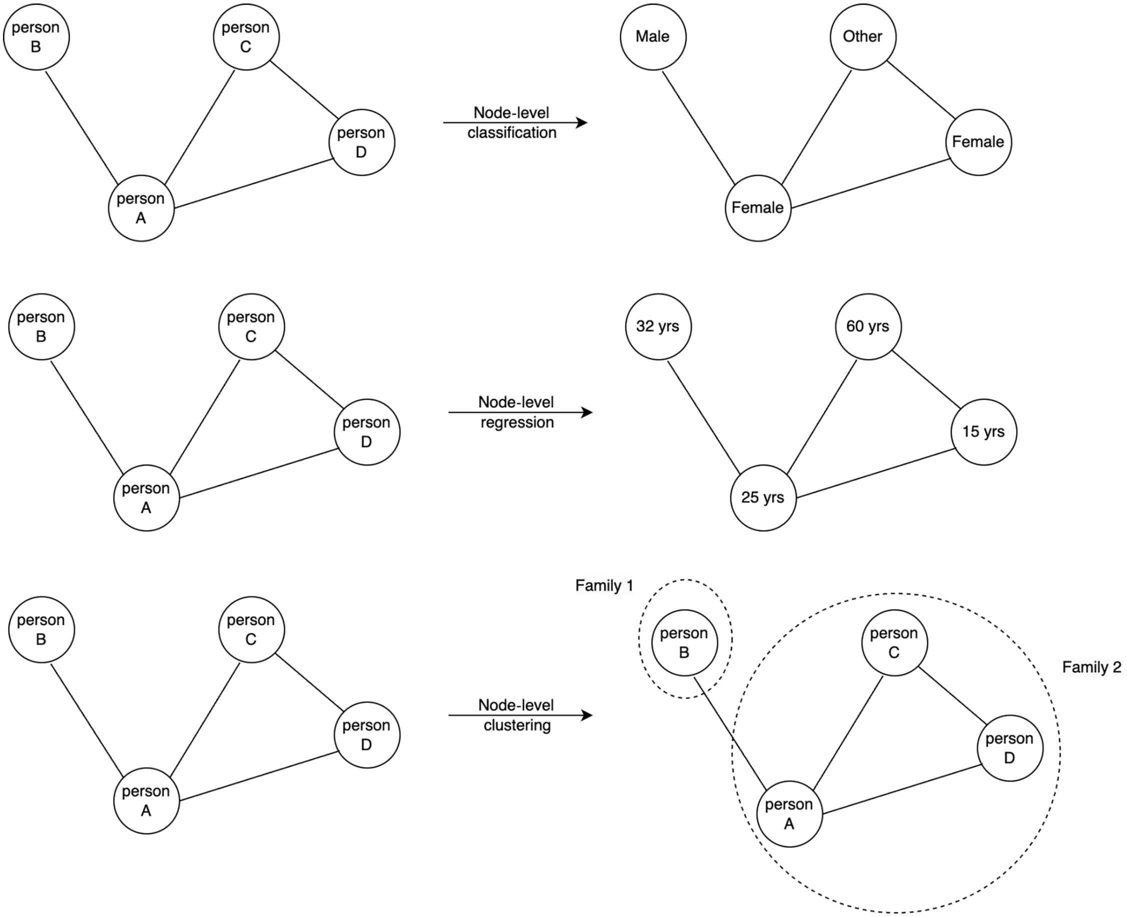


Figure 6.5: Examples of different types of node-level tasks – classification, regression, and clustering

If images were to be thought of as a graph of pixels, an appropriate node-level task equivalent would be semantic segmentation, where a segment class is predicted for each pixel (node). Also, if a sentence were to be considered as a graph of words, then part-of-speech tagging would be a node-level task where each word is assigned a part-of-speech tag (noun, verb, adjective, etc.).

Understanding edge-level tasks

Similar to node-level tasks, in edge-level tasks, the goal is to classify edges. Each edge can be represented numerically by combining/concatenating the node-level features of the nodes that it connects. Additionally, in some graphs, edges have their edge-level features too. Typically, an edge-level task uses the accompanying node features as well as the edge features to train a downstream classification task such as predicting the type of relationship (parent, sibling, etc.) in a graph of families, as shown in *Figure 6.6*.

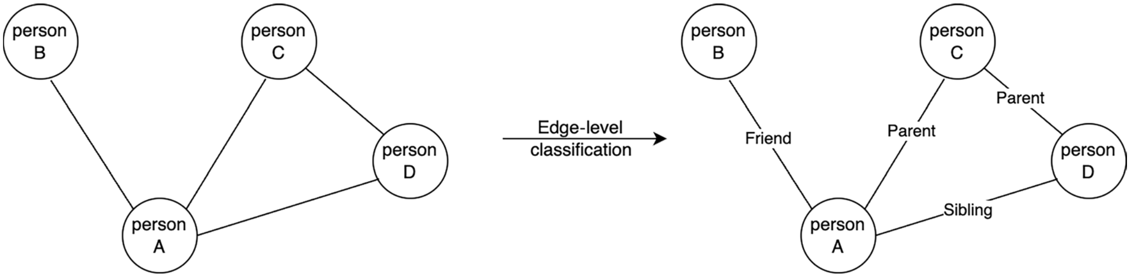


Figure 6.6: Example of an edge-level task – classifying relationships between people in a graph

In *Figure 6.6*, for predicting the relationship type between person **A** and **B**, both node **A** and **B**'s latent features are used. In the case of an undirected graph, both nodes' features are combined using a permutation-invariant aggregation function such as `sum`, `mean`, `min`, and `max`, so that the order of nodes (**A**->**B** or **B**->**A**) doesn't affect the edge features.

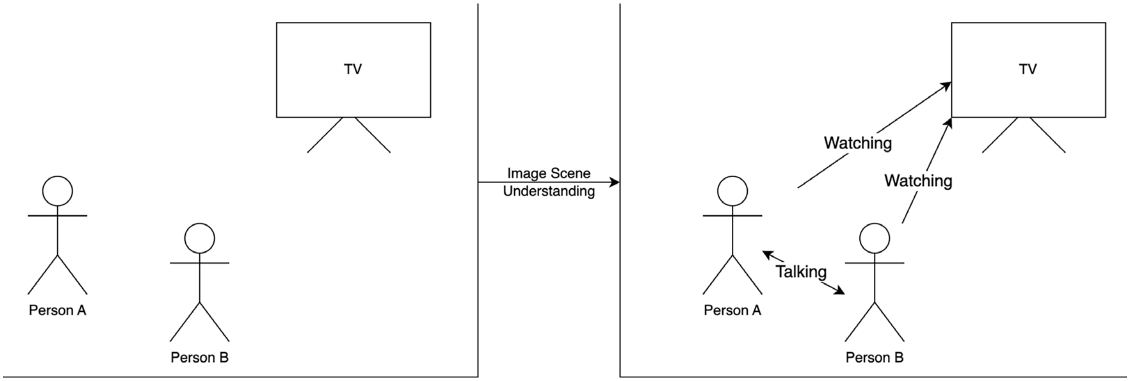


Figure 6.7: Example of image scene understanding where the goal is to predict the relationship between objects – person A, person B, and a TV

To find equivalence in the world of computer vision, image scene understanding (see *Figure 6.7*) is an appropriate example of an edge-level task equivalent where different objects in an image are the nodes and the relationships between these objects are the edges, and the goal is to predict the type of relationship between these objects in the image.

Understanding graph-level tasks

In graph-level tasks, we predict a class or a numerical value for the entire graph, by using the (permutation invariant) aggregation of the latent features of all nodes in the graph. Many graph datasets have several disjointed graphs inside of them, such as a graph dataset of molecules where each molecule is a graph structure. Predicting molecule type, in this case, is an example of a graph-level task.

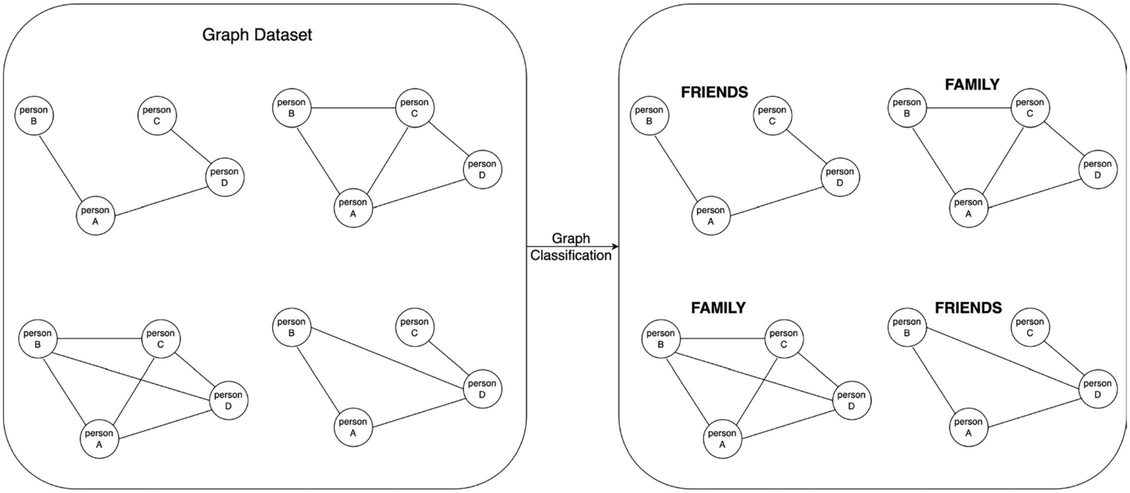


Figure 6.8: Example of a graph-level task where different relationship graphs within a graph dataset of people are categorized into friend groups or family

In the world of images, image classification is the graph-level task equivalent because all pixels (nodes) of the image (graph) are used to attribute a single value to the entire image (graph).

This concludes our discussion of the various types of graph-learning tasks. In the next section, we delve deeper into some of the prominent GNN models that have been developed over the last decade (as of May 2024). We'll learn about the key architectural and algorithmic features of some of the well-known GNN approaches.

Reviewing prominent GNN models

In this section, we will discuss a few different kinds of popular GNN models. Although a lot of GNN architectures have been developed in the last two decades, we'll limit our review to the following models, as they capture a major chunk of GNN concepts used in graph modeling today (as of May 2024):

- GCN
- GAT
- GraphSAGE

We'll briefly discuss the inner workings of these models and highlight how these model variants differ from each other.

Understanding graph convolutions with GCNs

We have already mentioned that the term *convolution* in GCNs [1] comes from the shared weights (NN_1, NN_2) across the graph (referring to *Figure 6.4*). To appreciate how GCNs work and how GCNs extract additional graph information from a graph dataset, let us remind ourselves of how to solve graph problems with traditional NNs, as shown in *Figure 6.9*.

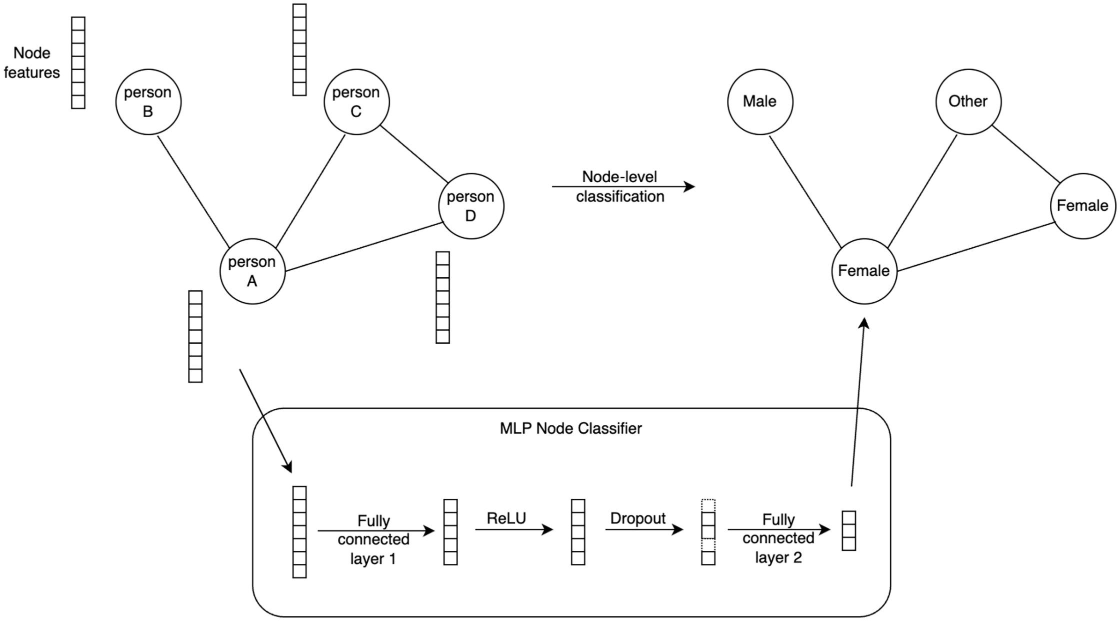


Figure 6.9: Using a feedforward NN with two fully connected layers to classify nodes in a graph using the node-level local features

As shown in *Figure 6.9*, a traditional (feedforward) NN can only utilize the node-level local information to predict the node class. In contrast, GCN uses computational graphs for each node to extract information from the node itself, the node's neighbors, a neighbor of neighbors, and so on, as shown in *Figure 6.10*.

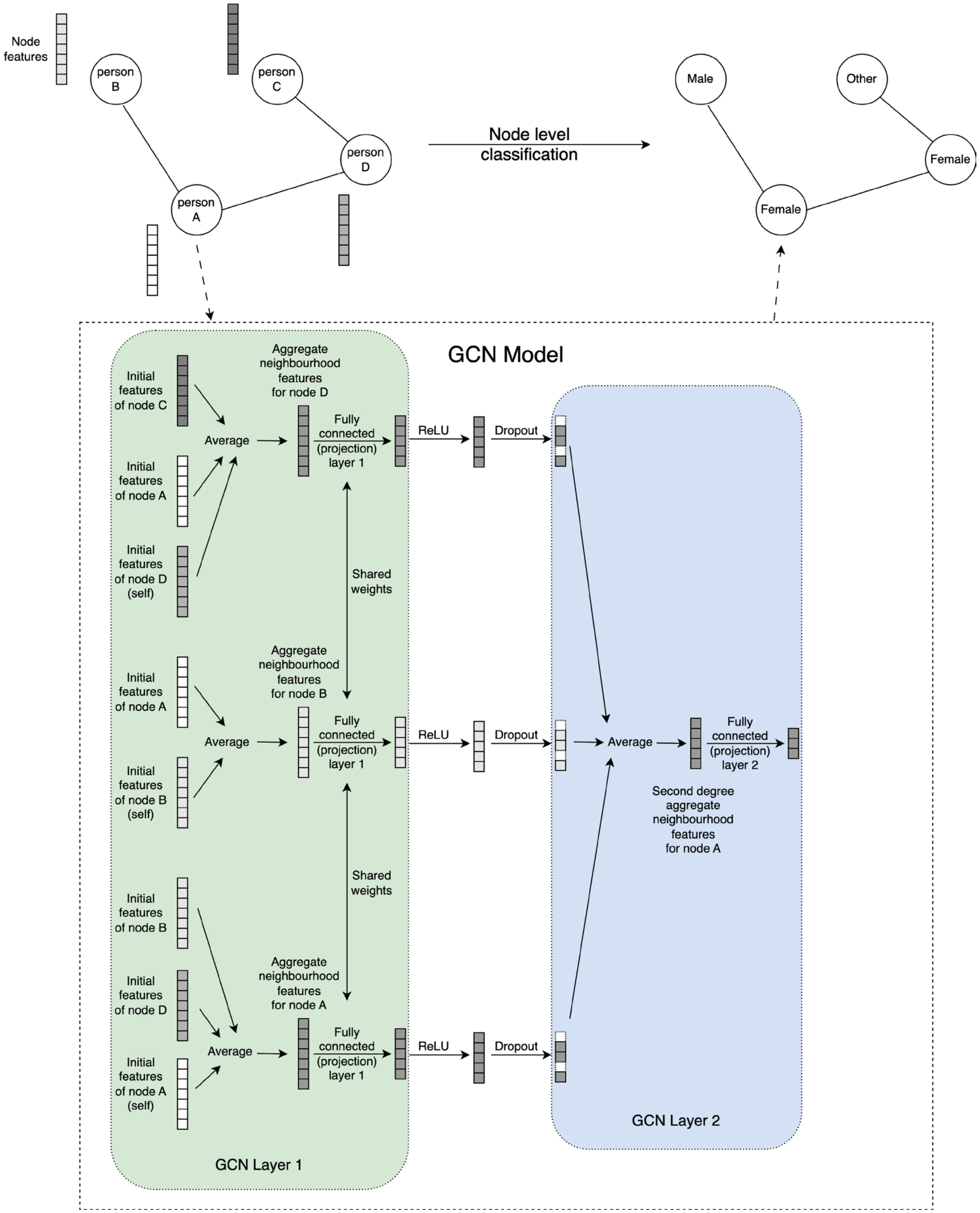

Node features
person B
person C
person D
person A
Node level classification
Male
Other
Female
Female
GCN Model
Initial features of node C
Initial features of node A
Initial features of node D (self)
Aggregate neighbourhood features for node D
Average
Fully connected (projection) layer 1
ReLU
Dropout
Shared weights
Initial features of node A
Initial features of node B (self)
Aggregate neighbourhood features for node B
Average
Fully connected (projection) layer 1
ReLU
Dropout
Shared weights
Initial features of node B
Initial features of node D
Initial features of node A (self)
Aggregate neighbourhood features for node A
Average
Fully connected (projection) layer 1
ReLU
Dropout
GCN Layer 1
Average
Fully connected (projection) layer 2
Second degree aggregate neighbourhood features for node A
GCN Layer 2

Figure 6.10: Two-layer GCN-based node classification model, demonstrating the computational graph for node A – in the first layer, the features from the second-level neighbors (neighbor of neighbors) of node A are aggregated to produce a latent representation of node A's neighbors; in the second layer, these latent representations are aggregated to produce final features for node A, which are then used for node classification

As shown in *Figure 6.10*, to predict the class of node **A**, a two-layer deep GCN model fetches information from node **A**'s neighbors (including itself), {**A**, **B**, **D**}, in the first layer. The second layer fetches information from neighbors of nodes **A**, **B**, and **D** (including themselves) – {**A**, **B**, **D**}, {**A**, **B**}, and {**A**, **C**, **D**} respectively. At each layer, an aggregation is performed – averaging to be precise, which ensures that the feature length is constant across layers, and the order of nodes does not impact the aggregation. Between the two GCN layers, the **rectified linear unit** (**ReLU**) and dropout layers are added for non-linearity.

If we compare *Figures 6.10* and *6.9*, the fully connected layers in *6.9* are essentially replaced by the GCN layers in *Figure 6.10*. The GCN layer itself consists of a fully connected layer but also contains the neighborhood feature aggregation component, which is the secret sauce that makes GCN work well on graph datasets. While we are using node classification as an example to discuss GCNs, GCNs can also perform graph classification, wherein the aggregation is performed across all nodes of the graph.

While GCNs use the averaging of information from neighbors, which already extracts valuable graph information, a lot of work has been done on finding better ways of aggregating information from neighbors. An important milestone in this aspect of GNN research is GAT, which we'll discuss next.

Using attention in graphs with GAT

GCN uses averaging as a mechanism to aggregate information from neighboring node features. This has an inherent limitation as it assumes that all neighbors are to be treated equally, which might not necessarily be the case. For example, if two nodes, X and Y, have the same initial feature values and the same set of neighbors, a GCN model would identify them under the same class or cluster. But this might not be true. To capture this level of nuanced information in graphs, we can replace the simple averaging mechanism with the attention mechanism. This is where GATs [2] come into play.

We learned about the attention mechanism in *Chapter 5*, in the context of textual data wherein we assign different importance to different words in a sentence, focusing on specific parts for further processing. In the context of GATs, attention allows the model to place different weights on different neighbors of a node while classifying the node type, thereby enabling a more complex and powerful model. With the attention mechanism, we learn attention coefficients for each neighbor that add more trainable parameters to the model. *Figure 6.11* shows the contrast between GAT and GCN approaches in aggregating the neighboring node features.

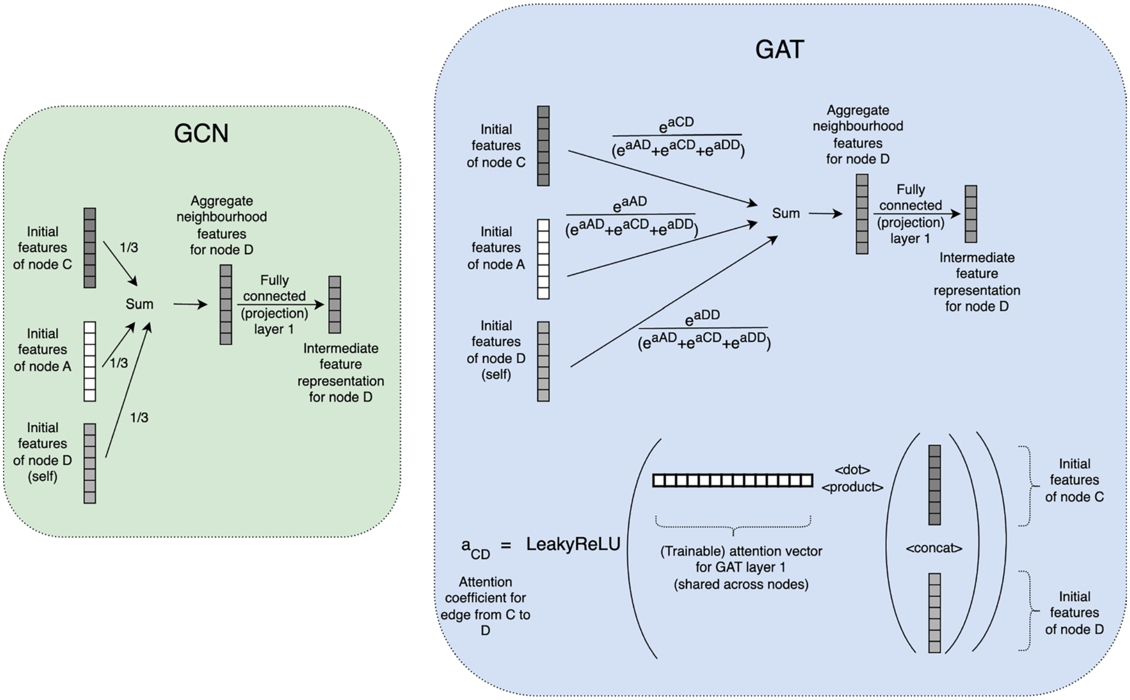

GCN
Initial features of node C
1/3
Aggregate neighbourhood features for node D
Sum
Fully connected (projection) layer 1
Initial features of node A
1/3
Intermediate feature representation for node D
Initial features of node D (self)
1/3
GAT
Initial features of node C
$\frac{e^{aCD}}{(e^{aAD}+e^{aCD}+e^{aDD})}$
Aggregate neighbourhood features for node D
Initial features of node A
$\frac{e^{aAD}}{(e^{aAD}+e^{aCD}+e^{aDD})}$
Sum
Fully connected (projection) layer 1
Intermediate feature representation for node D
Initial features of node D (self)
$\frac{e^{aDD}}{(e^{aAD}+e^{aCD}+e^{aDD})}$
a_{CD} = LeakyReLU
Attention coefficient for edge from C to D
<dot> <product>
(Trainable) attention vector for GAT layer 1 (shared across nodes)
<concat>
Initial features of node C
Initial features of node D

Figure 6.11: Demonstration of the contrast between aggregation mechanisms in GCNs and GATs – a uniform ⅓ weight is assigned to each of the 3 neighbors for aggregation in the case of GCN, while in GAT, the weights are calculated as the softmax over attention coefficients of all neighbors; the attention coefficients are calculated by using leaky ReLU on the dot-product of an attention vector with the concatenation of features of the node and the neighbor

As shown in *Figure 6.11*, we introduce a new set of trainable parameters in the form of an attention vector. The attention vector is twice the length of an individual node feature vector because it is dot-multiplied by the concatenation of a given node's feature with the node's neighbor's feature at a time. The learnable attention vector is shared across all <node, neighbor> pairs at a given layer. This additional set of trainable parameters provides GATs the ability to learn different weights for different feature dimensions, and for different neighbors.

The attention coefficient of each <node, neighbor> pair is calculated by passing the dot product of the attention vector and the concatenation of the node's and neighbor's features, through a leaky ReLU function. Finally, the weight of each neighbor is calculated as the softmaxed attention coefficients, where the softmax function is applied to attention coefficients. This softmax function, besides adding further non-linearity, also ensures that the weights sum up to 1. In this way, GATs significantly enhance the mechanism of extracting information from neighboring nodes using trainable attention parameters.

So far, we have seen algorithms that work on the entire graph. To classify a given node, we look at all of its neighbors (layer 1), a neighbor of neighbors (layer 2), and so on. Real-world graphs can have billions of nodes (think social networks). Scalability can become an issue when using such GNN models on the entire graph.

To handle such large graph datasets more efficiently, a lot of graph sampling methods have been developed over the years. **GraphSAGE** is one of them, and we'll discuss it next.

Performing graph sampling with GraphSAGE

GraphSAGE [3], short for graph sample and aggregate, randomly and uniformly samples neighbors for a given node, and uses only these selected neighbors to extract graph information, as opposed to GCN and GAT, which uses all neighbors. This algorithm is hence useful for large and dense graphs. *Figure 6.12* demonstrates how GraphSAGE works in the context of node classification.

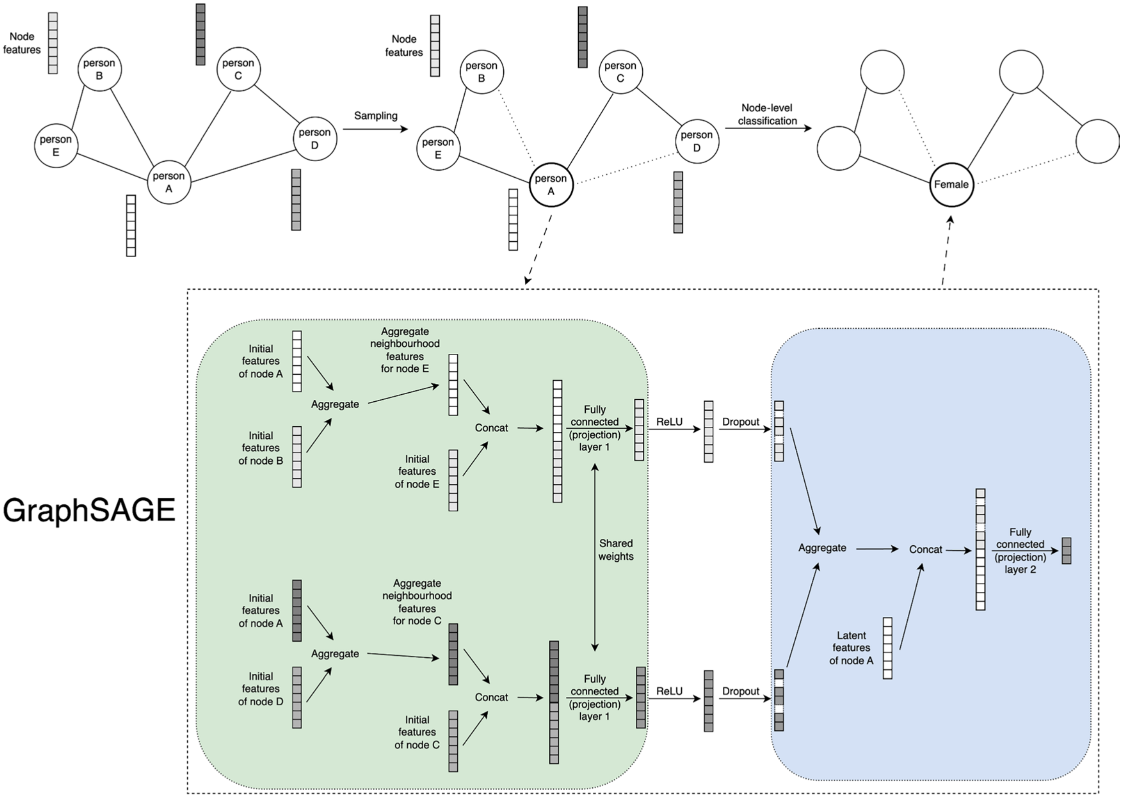

Node features
person B
person C
person E
person D
person A
Sampling
Node features
person B
person C
person E
person D
person A
Node-level classification
Female
GraphSAGE
Initial features of node A
Aggregate neighbourhood features for node E
Aggregate
Initial features of node B
Concat
Initial features of node E
Fully connected (projection) layer 1
ReLU
Dropout
Shared weights
Initial features of node A
Aggregate neighbourhood features for node C
Aggregate
Initial features of node D
Concat
Initial features of node C
Fully connected (projection) layer 1
ReLU
Dropout
Aggregate
Concat
Latent features of node A
Fully connected (projection) layer 2

Figure 6.12: Demonstration of a two-layer graphSAGE model for node classification including the node sampling step – two out of four neighbors are randomly sampled for node A; for both of these neighbors, two further neighbors are sampled, and this subgraph is then used for performing node classification using the principle of aggregating information from neighbors, thereby reducing the computational requirements of the model as compared to GCNs and GATs

In the example demonstrated in *Figure 6.12*, for node **A**, the set of neighbors to sample from is {**B**, **C**, **D**, **E**}. GraphSAGE randomly samples {**C**, **E**} out of the mix. At the second level of connection (second-level neighbors) (i.e., for nodes **C** and **E**), GraphSAGE samples two out of the two possible neighbors (i.e., {**A**, **D**} and {**A**, **B**} respectively). Next, features from nodes {**A**, **D**} are aggregated and the aggregation is concatenated with node **C**'s features.

A feedforward NN layer is applied to the concatenation, followed by ReLU and dropout. The same operation is performed on nodes {**A**, **B**} – neighbors of node **E**.

At the end of layer 1 of GraphSAGE, we have the latent representations of nodes **C** and **E**. In the second layer of the model, these latent representations are further aggregated and the aggregation is concatenated with the latent features of node **A**. A feedforward layer on this concatenation finally produces the node class probabilities.

For the aggregation function mentioned in *Figure 6.12*, the authors of the original GraphSAGE paper mention three different ways of aggregating the neighboring nodes' features:

- Averaging
- **Long short-term memory** (**LSTM**) model
- Max pooling

Besides the usual averaging technique, we can reshuffle the order of neighboring nodes and pass them through an LSTM model to generate an output embedding. Alternatively, each neighbor's feature vector is passed through a feedforward NN to produce `num_neighbors` number of output feature vectors, and the maximum value of each output feature is taken across all neighbors.

One of the milestone successors of the GraphSAGE model is **PinSAGE**, developed by Pinterest and used in their recommendation system, consisting of more than 3 billion nodes (users) and over 18 billion edges. This concludes our brief discussion of some of the well-known GNN models, the underlying architectures, and algorithms. In the next sections, we transition from GNN literature into hands-on coding exercises involving regular NNs, GCNs, and GATs.

Building a GCN model using PyTorch Geometric

In the previous sections, we covered a lot of GNN theory. It's time to go hands-on. In this section, we use PyTorch Geometric [4] – the GNN library for PyTorch, which comes with features such as:

- Optimized graph data loading/handling functionalities
- A repository of popular graph datasets
- Implementations of prominent GNN architectures
- Pre-trained popular GNN model weights

We'll build our own GCN model on the well-known *CiteSeer* graph dataset. It is a dataset of citation networks containing scientific publications as nodes connected based on citations. We'll perform the node-level task of classifying scientific publications in this graph into one of the six available classes in this dataset.

First, we'll explore and understand the dataset using PyTorch Geometric's data APIs. We'll then build a simple NN-based classification solution as a baseline for this task. This baseline approach won't use any graph information, but only the intrinsic features of each node (publication). After this, we'll build, train, and evaluate a GCN model on the same task and see whether this approach surpasses the baseline by utilizing the graph information. We'll see how PyTorch Geometric enables us to achieve all of this with a few lines of code. All the code for this chapter can be accessed from our GitHub repository [5].

Loading and exploring the citation networks dataset

As with any machine learning project, it all begins with the data. In this section, we'll learn how to use PyTorch Geometric to load, process, and visualize the `CiteSeer` graph dataset, which can be loaded directly from the PyTorch Geometric library. Before loading the data, we need to import a few important modules from this library:

```
from torch_geometric.nn import GCNConv
from torch_geometric.utils import to_networkx
from torch_geometric.datasets import Planetoid
```

The first line simply imports the GCN model class that we later use to swiftly develop our GCN model. The second line imports the `to_networkx` function, which converts a PyTorch Geometric dataset into a NetworkX-friendly graph object. NetworkX [6] is a Python library that provides a wide range of tools for creating, analyzing, and visualizing complex networks/graphs and their properties. The last line imports the citation networks dataset called the Planetoid dataset.

Now, we are ready to load the dataset and explore its key properties, using the following lines of code:

```
dataset = Planetoid(root='data/Planetoid', name='CiteSeer')
print(f'Dataset: {dataset}:')
print('======================')
print(f'Number of graphs: {len(dataset)}')
print(f'Number of features: {dataset.num_features}')
print(f'Number of classes: {dataset.num_classes}')
```

This should produce the following output:

```
Dataset: CiteSeer():
======================
Number of graphs: 1
Number of features: 3703
Number of classes: 6
```

The first thing to notice here is that there is only one graph in this dataset, unlike other graph datasets that have multiple disjointed graphs (see *Figure 6.8*). Secondly, we now know that each node in this graph has `3703` features representing a node. Thirdly, we can see that there are a total of `6` node classes in this dataset. But how many nodes and edges are there in this dataset? Let's find out with the following code:

```
data = dataset[0]  # Get the first graph object.
print(data)
print('================================================')
print(f'Number of nodes: {data.num_nodes}')
print(f'Number of edges: {data.num_edges}')
...
print(f'Is undirected: {data.is_undirected()}')
```

This should produce the following output:

```
Data(x=[3327, 3703], edge_index=[2, 9104], y=[3327], train_mask=[3327], val_
mask=[3327], test_mask=[3327])
==========================================================
Number of nodes: 3327
Number of edges: 9104
Average node degree: 2.74
Number of training nodes: 120
Training node label rate: 0.04
Has isolated nodes: True
Has self-loops: False
Is undirected: True
```

First, we fetch the only graph in this dataset under the `data` variable. This graph contains `3327` nodes and `9104` edges where `120` out of the `3327` nodes have labels (one of the six classes) associated with them. There are no disconnected nodes in this dataset, and also no node is connected to itself (self-loop). Finally, the edges are bidirectional in this graph as it is an undirected graph.

We have already gathered some information about the citation graph using PyTorch Geometric's functions. Before we move on to building a model on this dataset, let us visualize this graph dataset using the following piece of code:

```python
def visualize_graph(G, color):
    plt.figure(figsize=(15,15))
    plt.xticks([])
    plt.yticks([])
    nx.draw_networkx(
        G, pos=nx.spring_layout(G), with_labels=False,
        node_color=color, cmap="Set2")
    plt.show()
G = to_networkx(data, to_undirected=True)
visualize_graph(G, color=data.y)
```

This should produce an output as shown in *Figure 6.13*.

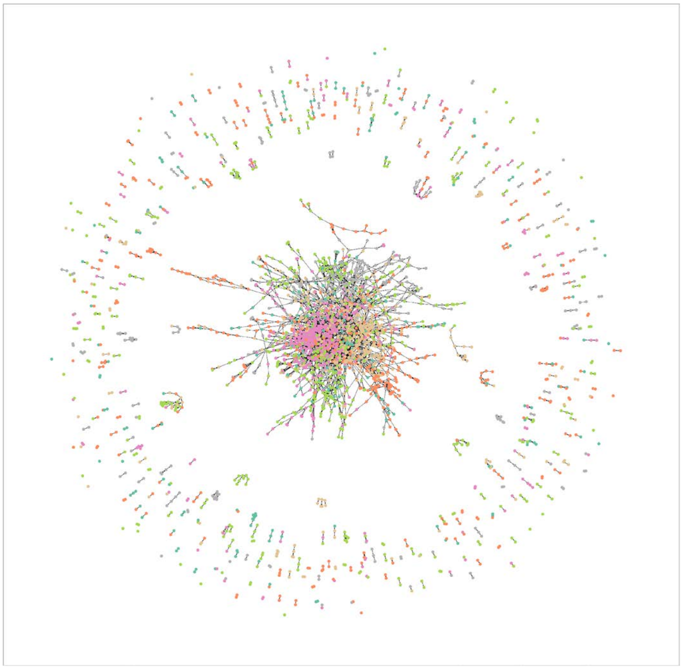


Figure 6.13: Visualization of the CiteSeer citation networks dataset – different node colors represent the six different classes; there is a densely connected network, and then there are smaller, fragmented sub-networks of publications

As we can see in *Figure 6.13*, there is a dense network of publications and then there are multiple smaller sub-networks of publications (on the periphery) in this graph.

Now that we have loaded, explored, and visualized the CiteSeer graph dataset, let us build a baseline NN-based node classifier using this dataset.

Building a simple NN-based node classifier

In this section, we'll build and train a **multilayer perceptron** (**MLP**) model on the CiteSeer graph dataset using PyTorch. First, we instantiate, with random weights, an MLP model that takes in the features of a node as input and produces the node class as output. Then, we visualize the output node embeddings (of size 6) produced by the randomly initialized MLP model. Next, we train the MLP model on the multi-class classification task. We finally evaluate the trained MLP model and visualize the node embeddings produced from the trained model.

Figure 6.9 shows an MLP model architecture for node classification, containing an input layer (the same size as the number of node features), a hidden layer with 16 neurons, and an output layer with 6 neurons (for the six different node classes). We use ReLU activation and dropout on the hidden layer.

The equivalent PyTorch code for instantiating the MLP model is shown here:

```
class MLP(torch.nn.Module):
    def __init__(self, hidden_channels):
        super().__init__()
        torch.manual_seed(12345)
        self.lin1 = Linear(dataset.num_features, hidden_channels)
        self.lin2 = Linear(hidden_channels, dataset.num_classes)
    def forward(self, x):
        x = self.lin1(x)
        x = x.relu()
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.lin2(x)
        return x
model = MLP(hidden_channels=16)
print(model)
```

This should produce the following output:

```
MLP(
  (lin1): Linear(in_features=3703, out_features=16, bias=True)
  (lin2): Linear(in_features=16, out_features=6, bias=True)
)
```

As we can see, there are 3703 input features and 6 output features in this model. We'll now use this untrained model to get the 6 output features for all of the nodes of our dataset and apply the **t-distributed stochastic neighbor embedding** (**t-SNE**) algorithm on these 6-dimensional feature embeddings to find a representative 2D embedding. t-SNE is a dimensionality reduction technique used for visualizing high-dimensional data in a lower-dimensional space while preserving pairwise similarities. You can read in detail about t-SNE at [7]. We plot all nodes of the dataset on a 2D plot using the t-SNE embeddings with the following code:

```
def visualize(data, labels):
    tsne = TSNE(n_components=2, init='pca', random_state=7)
    tsne_res = tsne.fit_transform(data)
    v = pd.DataFrame(data,columns=[str(i) for i in range(data.shape[1])])
    v['color'] = labels
    v['label'] = v['color'].apply(lambda i: str(i))
    v["dim1"] = tsne_res[:,0]
    v["dim2"] = tsne_res[:,1]
    plt.figure(figsize=(12,12))
    sns.scatterplot(
        x="dim1", y="dim2",
        hue="color",
        palette=sns.color_palette(
            ["#52D1DC", "#8D0004", "#845218","#563EAA",
             "#E44658", "#63C100", "#FF7800"]),
        legend=False,
        data=v,
    )
model.eval()
out = model(data.x)
visualize(out.detach().cpu().numpy(), data.y)
```

This should produce the output in *Figure 6.14*.

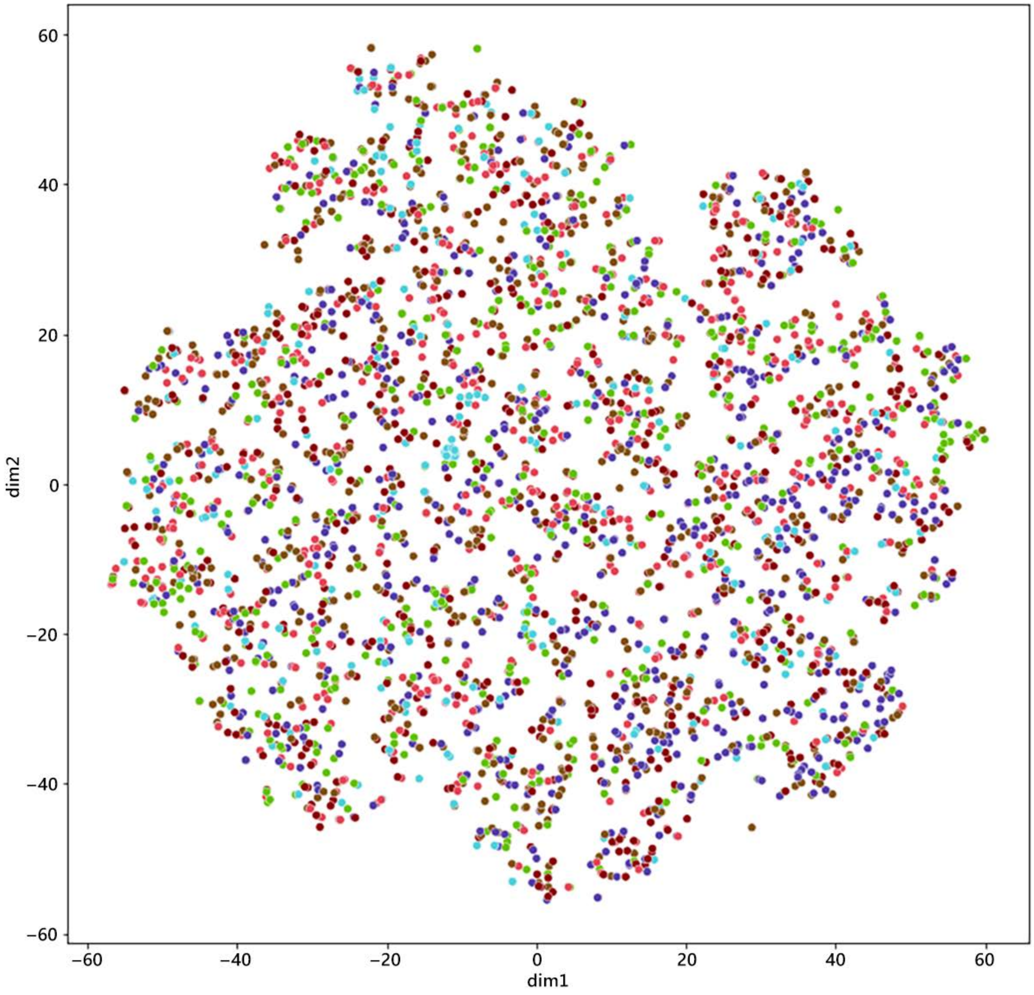


Figure 6.14: Initial node embeddings produced by the MLP node classification model

As we can see in *Figure 6.14*, all nodes belonging to the different classes (represented by `6` different colors) are distributed randomly in this 2D distribution. This is expected, as the MLP model that was used to obtain these embeddings is not trained and has randomly initialized weights.

In the visualization code, we defined a `visualize` function that takes in the node embeddings (MLP output) as well as the node class label (out of the `6` classes). This function transforms the node embeddings of size `6` to size `2` using the t-SNE algorithm. We use this visualize function to visually check the performance of different models in how they can spread apart nodes of different classes into homogenous clusters. For now, we'll move on to training the MLP model on the node-level features, using the following code:

```
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=5e-3)
def train():
      model.train()
      optimizer.zero_grad()
      out = model(data.x)
      loss = criterion(out[data.train_mask], data.y[data.train_mask])
      loss.backward()
      optimizer.step()
      return loss
def test(mask):
      model.eval()
      out = model(data.x)
      pred = out.argmax(dim=1)
      correct = pred[mask] == data.y[mask]
      acc = int(correct.sum()) / int(mask.sum())
      return acc
for epoch in range(1, 101):
    loss = train()
    val_acc = test(data.val_mask)
    print(f'Epoch: {epoch:03d}, Loss: {loss:.4f}, Val: {val_acc:.4f}')
```

First, we define the loss function – `CrossEntropyLoss` for the multi-class classification task. Then, we define `Adam` as the optimizer of our choice. Next, we define the `train` function, where the MLP model is trained using gradient descent and backpropagation. We also define a `test` function to evaluate the performance of the trained MLP model. The `test` function has a `mask` parameter that can take variables such as `train_mask`, `val_mask`, or `test_mask` as input, where `mask` is a list of length equal to the total number of nodes in the graph. The `mask` list contains `0`s and `1`s, indicating which node in the graph is to be ignored and to be chosen, respectively. If there are `3` nodes in the graph, where the first node belongs to the training set, the second to the validation set, and the third to the test set, then the training mask will be `[1, 0, 0]`, the validation mask will be `[0, 1, 0]`, and the test mask will be `[0, 0, 1]`.

Finally, the preceding code runs a training loop for `100` epochs, and prints model performance on training and validation sets. The preceding code should produce the following output:

```
Epoch: 001, Loss: 1.8052, Val: 0.1020
Epoch: 002, Loss: 1.7371, Val: 0.1540
Epoch: 003, Loss: 1.6468, Val: 0.1940
Epoch: 004, Loss: 1.5291, Val: 0.2820
Epoch: 005, Loss: 1.3691, Val: 0.3720
...
Epoch: 096, Loss: 0.2214, Val: 0.5720
Epoch: 097, Loss: 0.1520, Val: 0.5720
Epoch: 098, Loss: 0.2303, Val: 0.5740
Epoch: 099, Loss: 0.2675, Val: 0.5700
Epoch: 100, Loss: 0.2040, Val: 0.5620
```

The training logs indicate that the model is learning something from the training set, as evidenced by the decreasing training loss. The logs also show an increase in validation set accuracy from 10% in the first epoch to around 56-57% toward the 100th epoch. Let us now check the performance of the trained model (at the 100th epoch) on the untouched test set, using the following code:

```
test_acc = test(data.test_mask)
print(f'Test Accuracy: {test_acc:.4f}')
```

This should produce the following output:

```
Test Accuracy: 0.5710
```

The accuracy on the test set is quite similar to that on the training set, which indicates that our training results are generalizable on the unseen parts of the graph. We can therefore assume that we have successfully trained an MLP-based node classifier. Let us now go back to the t-SNE-based visualization and use the trained MLP model to generate node embeddings and visualize them on a 2D plot, using the following code:

```
out = model(data.x)
visualize(out.detach().cpu().numpy(), data.y)
```

This should produce the output in *Figure 6.15*.

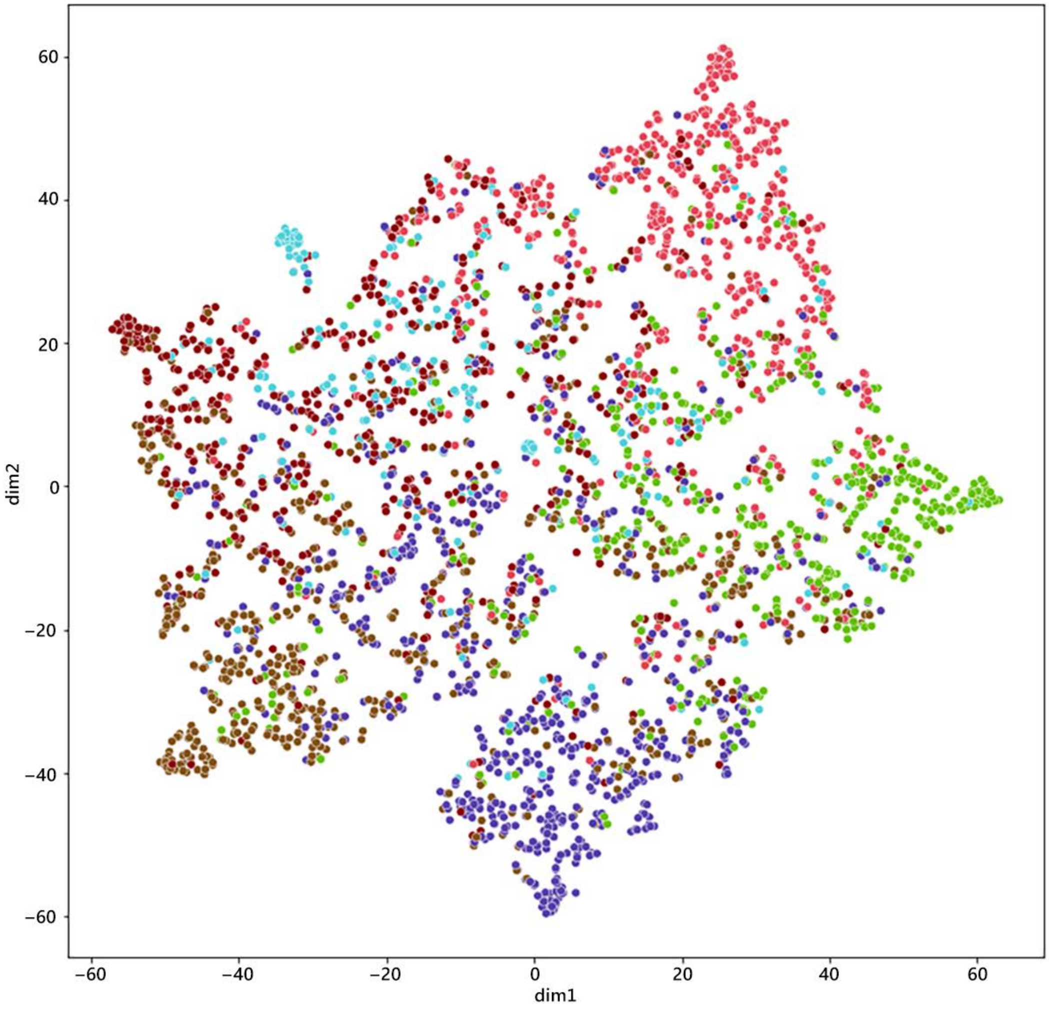


Figure 6.15: Node embeddings produced by the MLP node classifier after training

Figure 6.15 shows somewhat of a clustering of nodes of the same colors into 6 separate clusters or groups. The node clusters have some overlaps but it is visually evident that the embeddings produced by the trained MLP model are far better in distinguishing nodes of different classes than the untrained MLP model. Therefore, the MLP model is indeed able to use the `3703` node-level features alone (without any graph context) to classify the nodes with a decent accuracy of 57%. (Remember that, with 6 classes, a random classifier would have an accuracy of 16.67%.) But is 57% enough? Or can we do better? Are we using the full extent of information available in this graph dataset to classify nodes? We are using the node-level features but what about the neighborhood context? Is that information helpful? How can we use that information in building a node classifier? We'll learn about all of this next.

Building a GCN model for node classification

In the previous section, we utilized the node-level features of the CiteSeer graph dataset to classify different types of nodes using a simple MLP model. In this section, we'll go beyond the node-level information and further utilize the neighborhood information of a given node, using GCN, to perform node classification. We'll use PyTorch Geometric to build and train a GCN model in a few lines of code. Let's get started:

1. First, we define the GCN model architecture, along with the forward pass of the model, using the following lines of code:

    ```
    class GCN(torch.nn.Module):
        def __init__(self, hidden_channels):
            super().__init__()
            torch.manual_seed(1234567)
            self.conv1 = GCNConv(dataset.num_features, hidden_channels)
            self.conv2 = GCNConv(hidden_channels, dataset.num_classes)
        def forward(self, x, edge_index):
            x = self.conv1(x, edge_index)
            x = x.relu()
            x = F.dropout(x, p=0.5, training=self.training)
            x = self.conv2(x, edge_index)
            return x
    model = GCN(hidden_channels=16)
    print(model)
    ```

    This should produce the following output:

    ```
    GCN(
      (conv1): GCNConv(3703, 16)
      (conv2): GCNConv(16, 6)
    )
    ```

    The preceding code defines and instantiates a two-layer GCN model. All nodes in the graph are transformed from `3703` input features to `16` intermediate features in the first layer, and from `16` features to the `6` output classes in the second/output layer, similar to the demonstration in *Figure 6.10*.

2. We now use the instantiated untrained GCN model to get the `6` output features for all nodes of our dataset, apply t-SNE on these `6` features to transform them into `2` features, and then plot all nodes of the graph dataset on a 2D plot, with the following code:

    ```
    model.eval()
    out = model(data.x, data.edge_index)
    visualize(out.detach().cpu().numpy(), data.y)
    ```

This should produce the output in *Figure 6.16*.

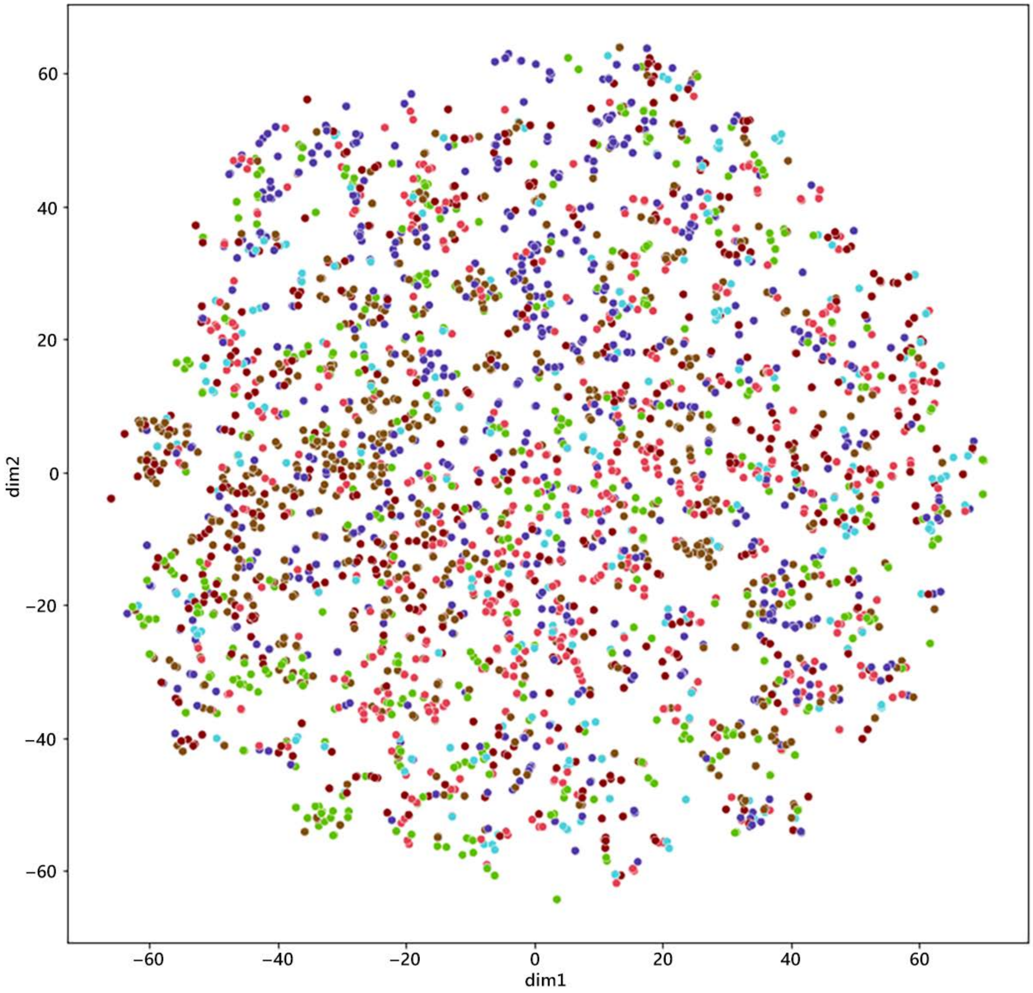


Figure 6.16: Initial node embeddings produced by the GCN node classifier before training

Similar to *Figure 6.14*, we can see in *Figure 6.16* that all nodes belonging to the different classes (represented by `6` different colors) are distributed randomly. This is expected because of the randomly initialized GCN model weights.

3. Next, we train the GCN model for `100` epochs after defining the optimizer, the loss function, and the model training and evaluation routines, as shown in the following code:

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_
decay=5e-3)
criterion = torch.nn.CrossEntropyLoss()
def train():
      model.train()
      optimizer.zero_grad()
      out = model(data.x, data.edge_index)
      loss = criterion(out[data.train_mask], data.y[data.train_mask])
      loss.backward()
      optimizer.step()
      return loss
def test(mask):
      model.eval()
      out = model(data.x, data.edge_index)
      pred = out.argmax(dim=1)
      correct = pred[mask] == data.y[mask]
      acc = int(correct.sum()) / int(mask.sum())
      return acc
for epoch in range(1, 101):
    loss = train()
    val_acc = test(data.val_mask)
    print(f'Epoch: {epoch:03d}, Loss: {loss:.4f}, Val: {val_acc:.4f}')
```

This should produce the following output:

```
Epoch: 001, Loss: 1.7871, Val: 0.3620
Epoch: 002, Loss: 1.6260, Val: 0.4100
Epoch: 003, Loss: 1.4544, Val: 0.5100
Epoch: 004, Loss: 1.2277, Val: 0.5740
Epoch: 005, Loss: 1.0790, Val: 0.6200
. . .
Epoch: 096, Loss: 0.1098, Val: 0.6880
Epoch: 097, Loss: 0.1258, Val: 0.6940
Epoch: 098, Loss: 0.0871, Val: 0.6960
Epoch: 099, Loss: 0.1123, Val: 0.7000
Epoch: 100, Loss: 0.1076, Val: 0.7000
```

The preceding code differs from the previous MLP model training code only in the forward pass, where, besides providing the node-level features (`data.x`), we also provide the adjacency matrix represented in a compact format under `data.edge_index`, which lists all edges of the graph as pairs of node indices. This extra information helps the GCN model to run the computation graph for each node, as demonstrated for node A in *Figure 6.10*. The training logs indicate that the training losses are lower than MLP training losses, and the validation set accuracies are higher than the MLP validation set accuracies.

4. Having trained the GCN model, we evaluate the trained model on the same test set that we used for evaluating the MLP model with the following code:

```
test_acc = test(data.test_mask)
print(f'Test Accuracy: {test_acc:.4f}')
```

This should produce the following output:

```
Test Accuracy: 0.6960
```

As we can see, the GCN model accuracy is 69.60%, a significant jump from the 57.10% test set accuracy obtained using the MLP model. This is intuitive because the GCN model uses extra graph information besides the node-level features that the MLP model uses.

5. We finally use the trained GCN model to run a forward pass on all graph nodes, generate the `6` output probabilities, transform these `6` numbers to `2` numbers using t-SNE, and visualize them on a 2D plot, using the following code:

```
out = model(data.x, data.edge_index)
visualize(out.detach().cpu().numpy(), data.y)
```

This should produce the output in *Figure 6.17*.

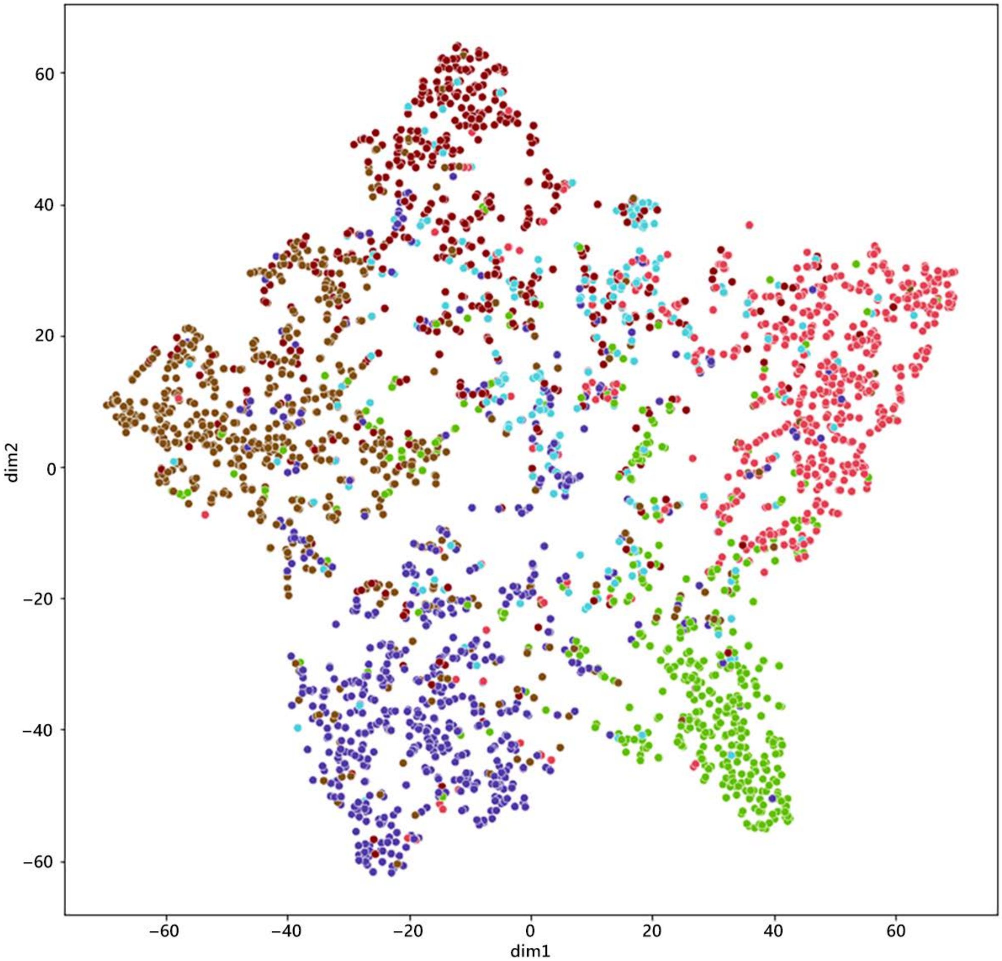


Figure 6.17: Node embeddings produced by the GCN model after training

Figure 6.17 further confirms that we have trained a GCN model that works well on the node classification task, as we can see a significantly better clustering of nodes into the six classes compared to the random distribution of nodes in *Figure 6.16*.

Moreover, *Figure 6.17* even shows a better clustering of nodes than the clusters produced by the trained MLP model in *Figure 6.15*. *Figure 6.17* almost forms a star-like formation, with five node classes pointing toward the five corners of the star, and the sixth class is dispersed in the center.

The GCN model has managed to leverage the additional graph information to push the nodes of different classes further apart. While we have now used both node-level features as well as graph-level (neighborhood) information to classify nodes with a GCN model, can we make any further improvements to increase the node classification accuracy? Can we further optimize the model – the graph learning algorithm? In the next and final section of this chapter, we'll improve the node classification performance with the help of the attention mechanism.

Training a GAT model with PyTorch Geometric

In the previous section, we exceeded our baseline MLP performance on the node classification task by using a GCN model. In this section, we'll further improve our solution by replacing the GCN model with the GAT model. Essentially, we'll replace the averaging mechanism (aggregate the information from neighboring nodes) as shown in *Figure 6.10* with the attention mechanism shown in *Figure 6.11*.

With the help of PyTorch Geometric, we refactor our GCN-based solution into a GAT-based solution with a few lines of code, as demonstrated in the following steps:

1. First, we define our GAT model architecture and its forward pass function:

```
class GAT(torch.nn.Module):
    def __init__(self, hidden_channels, heads):
        super().__init__()
        torch.manual_seed(1234567)
        self.conv1 = GATConv(dataset.num_features,
                             hidden_channels, heads)
        self.conv2 = GATConv(hidden_channels * heads,
                             dataset.num_classes, heads=1)
    def forward(self, x, edge_index):
        x = F.dropout(x, p=0.6, training=self.training)
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, p=0.6, training=self.training)
        x = self.conv2(x, edge_index)
        return x
model = GAT(hidden_channels=16, heads=8)
print(model)
```

This should produce the following output:

```
GAT(
  (conv1): GATConv(3703, 16, heads=8)
  (conv2): GATConv(128, 6, heads=1)
)
```

An important aspect of this model is the number of attention `heads` per `GATConv` layer. In the first layer, we keep eight attention heads. This means that we are going to derive eight parallel aggregations of neighboring node features using eight parallel and independent trainable attention coefficients, as demonstrated in *Figure 6.18*.

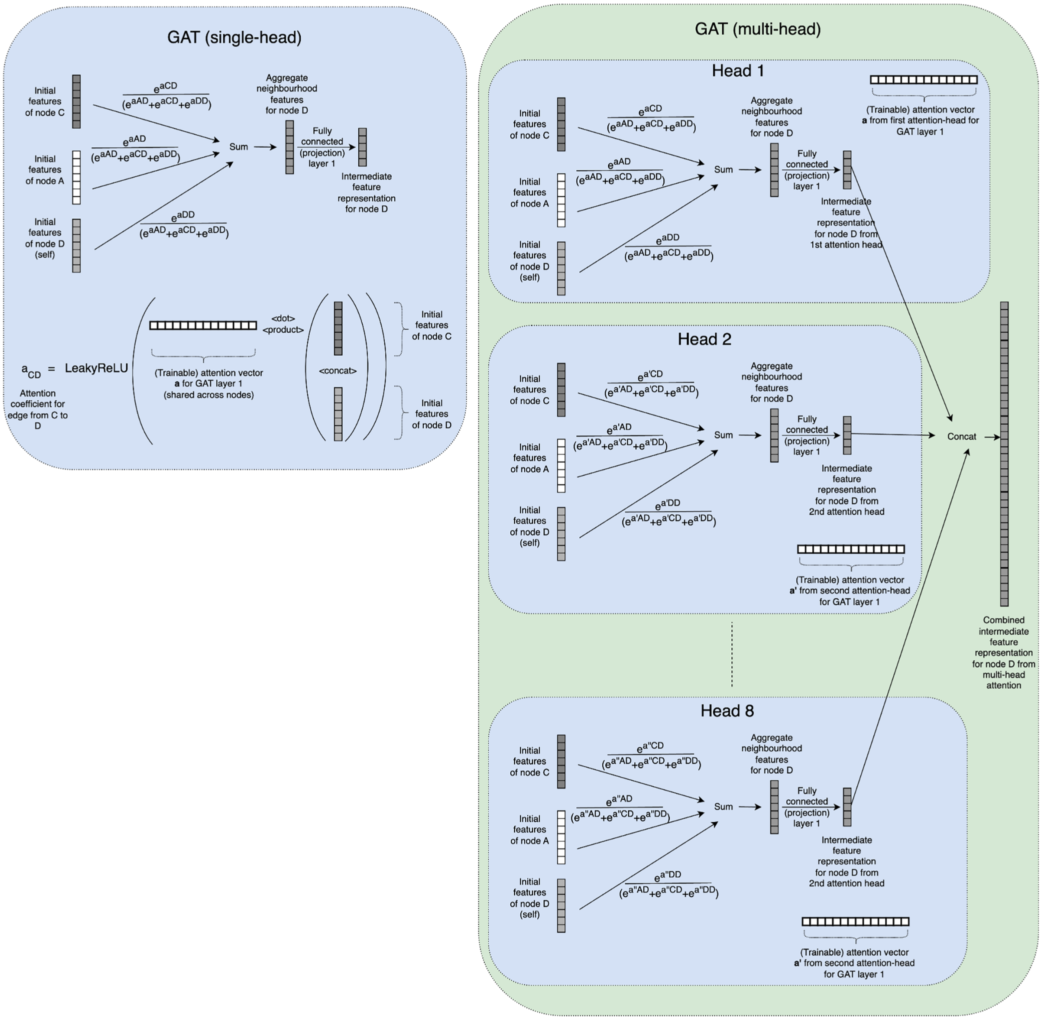

GAT (single-head)
Initial features of node C
Initial features of node A
Initial features of node D (self)
$\frac{e^{aCD}}{(e^{aAD}+e^{aCD}+e^{aDD})}$
$\frac{e^{aAD}}{(e^{aAD}+e^{aCD}+e^{aDD})}$
$\frac{e^{aDD}}{(e^{aAD}+e^{aCD}+e^{aDD})}$
Sum
Aggregate neighbourhood features for node D
Fully connected (projection) layer 1
Intermediate feature representation for node D
a_{CD} = LeakyReLU
Attention coefficient for edge from C to D
(Trainable) attention vector **a** for GAT layer 1 (shared across nodes)
<dot> <product>
<concat>
Initial features of node C
Initial features of node D
GAT (multi-head)
Head 1
Initial features of node C
Initial features of node A
Initial features of node D (self)
$\frac{e^{aCD}}{(e^{aAD}+e^{aCD}+e^{aDD})}$
$\frac{e^{aAD}}{(e^{aAD}+e^{aCD}+e^{aDD})}$
$\frac{e^{aDD}}{(e^{aAD}+e^{aCD}+e^{aDD})}$
Sum
Aggregate neighbourhood features for node D
Fully connected (projection) layer 1
Intermediate feature representation for node D from 1st attention head
(Trainable) attention vector **a** from first attention-head for GAT layer 1
Head 2
Initial features of node C
Initial features of node A
Initial features of node D (self)
$\frac{e^{a'CD}}{(e^{a'AD}+e^{a'CD}+e^{a'DD})}$
$\frac{e^{a'AD}}{(e^{a'AD}+e^{a'CD}+e^{a'DD})}$
$\frac{e^{a'DD}}{(e^{a'AD}+e^{a'CD}+e^{a'DD})}$
Sum
Aggregate neighbourhood features for node D
Fully connected (projection) layer 1
Intermediate feature representation for node D from 2nd attention head
(Trainable) attention vector **a'** from second attention-head for GAT layer 1
Concat
Combined intermediate feature representation for node D from multi-head attention
Head 8
Initial features of node C
Initial features of node A
Initial features of node D (self)
$\frac{e^{a''CD}}{(e^{a''AD}+e^{a''CD}+e^{a''DD})}$
$\frac{e^{a''AD}}{(e^{a''AD}+e^{a''CD}+e^{a''DD})}$
$\frac{e^{a''DD}}{(e^{a''AD}+e^{a''CD}+e^{a''DD})}$
Sum
Aggregate neighbourhood features for node D
Fully connected (projection) layer 1
Intermediate feature representation for node D from 2nd attention head
(Trainable) attention vector **a'** from second attention-head for GAT layer 1

Figure 6.18: Demonstration of contrast between the workings of a single-head and a multi-head GAT – the multi-head GAT simply replicates 8x the attention-heads and concatenates the results of these heads to produce an 8x richer node embedding

At the end of the first `GATConv` layer, the resulting feature vectors (each of size `16`) of these `8` attention heads are concatenated together, resulting in an output of size `128`. The second `GATConv` layer consists of `1` attention head and outputs the `6` node classes. In the forward pass of this model, we use multiple dropouts to combat potential overfitting due to the increased model complexity (especially with multiple attention heads, providing an even more nuanced passage of information through the graph).

2. As we did for the MLP and GCN models, we now run all nodes of our dataset through the untrained GAT model that we have just defined, to produce the six class probability vectors for the six node classes. We then apply t-SNE on these 6 numbers to reduce them to 2 features, and then plot all nodes of the graph dataset using the 2D representation, with the following code:

```
model.eval()
out = model(data.x, data.edge_index)
visualize(out.detach().cpu().numpy(), data.y)
```

This should produce the output in *Figure 6.19*.

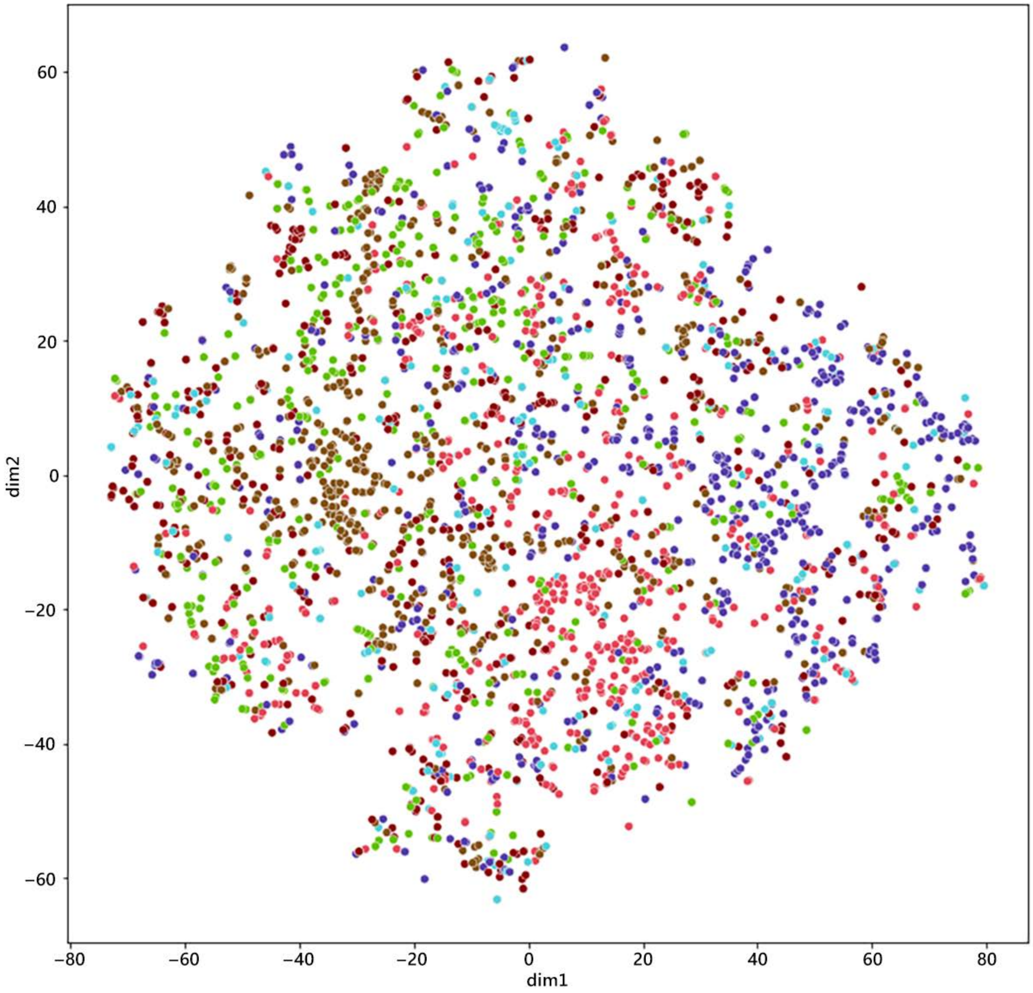


Figure 6.19: Node embeddings produced by the GAT model before training

As expected, the distribution is random. We are now ready to train the GAT model.

Next, we train the GAT model for 100 epochs after defining the optimizer, loss function, and the model training and evaluation routines, as shown in the following code:

```
# hyperparemeters inspired by torch geometric example on GAT
# https://colab.research.google.com/
# drive/14OvFnAXggxB8vM4e8vSURUp1TaKnovzX?usp=sharing
optimizer = torch.optim.Adam(
    model.parameters(), lr=0.0005, weight_decay=1e-1)
criterion = torch.nn.CrossEntropyLoss()
def train():
      model.train()
      optimizer.zero_grad()
      out = model(data.x, data.edge_index)
      loss = criterion(out[data.train_mask], data.y[data.train_mask])
      loss.backward()
      optimizer.step()
      return loss
def test(mask):
      model.eval()
      out = model(data.x, data.edge_index)
      pred = out.argmax(dim=1)
      correct = pred[mask] == data.y[mask]
      acc = int(correct.sum()) / int(mask.sum())
      return acc
for epoch in range(1, 101):
    loss = train()
    val_acc = test(data.val_mask)
    test_acc = test(data.test_mask)
    print(f'Epoch: {epoch:03d}, Loss: {loss:.4f}, Val: {val_acc:.4f}')
```

This should produce the following output:

```
Epoch: 001, Loss: 1.7793, Val: 0.2180
Epoch: 002, Loss: 1.7874, Val: 0.2320
Epoch: 003, Loss: 1.7888, Val: 0.2480
Epoch: 004, Loss: 1.7795, Val: 0.2760
Epoch: 005, Loss: 1.7772, Val: 0.2980
. . .
Epoch: 096, Loss: 1.1057, Val: 0.7180
Epoch: 097, Loss: 1.1514, Val: 0.7200
Epoch: 098, Loss: 1.1342, Val: 0.7220
Epoch: 099, Loss: 1.1354, Val: 0.7220
Epoch: 100, Loss: 1.1131, Val: 0.7220
```

The training and evaluation routines for the GAT model are the same as for the GCN model. From the training logs, it looks like the validation set accuracy (72.20%) is slightly higher towards the end of the 100 training epochs than the validation set accuracy (70.00%) of the GCN model. We can confirm this trend by evaluating the test set results.

3. We use the trained GAT model to evaluate its accuracy on the test set with the following code:

```
test_acc = test(data.test_mask)
print(f'Test Accuracy: {test_acc:.4f}')
```

This should produce the following output:

```
Test Accuracy: 0.7210
```

Compared to the 69.60% accuracy obtained using the GCN model, we get a further boost of 2.50% by using the GAT model. This is expected because of the powerful nature of attention layers, which adds more trainable parameters to the model and provides more flexibility to the graph model to learn custom relationships between different neighboring nodes.

4. Finally, we can visualize the 2D node embeddings by making predictions on all graph nodes using the trained GAT model and using t-SNE to transform the 6D node-class probabilities into 2D with the following code:

```
out = model(data.x, data.edge_index)
visualize(out.detach().cpu().numpy(), data.y)
```

This should produce the output in *Figure 6.20*.

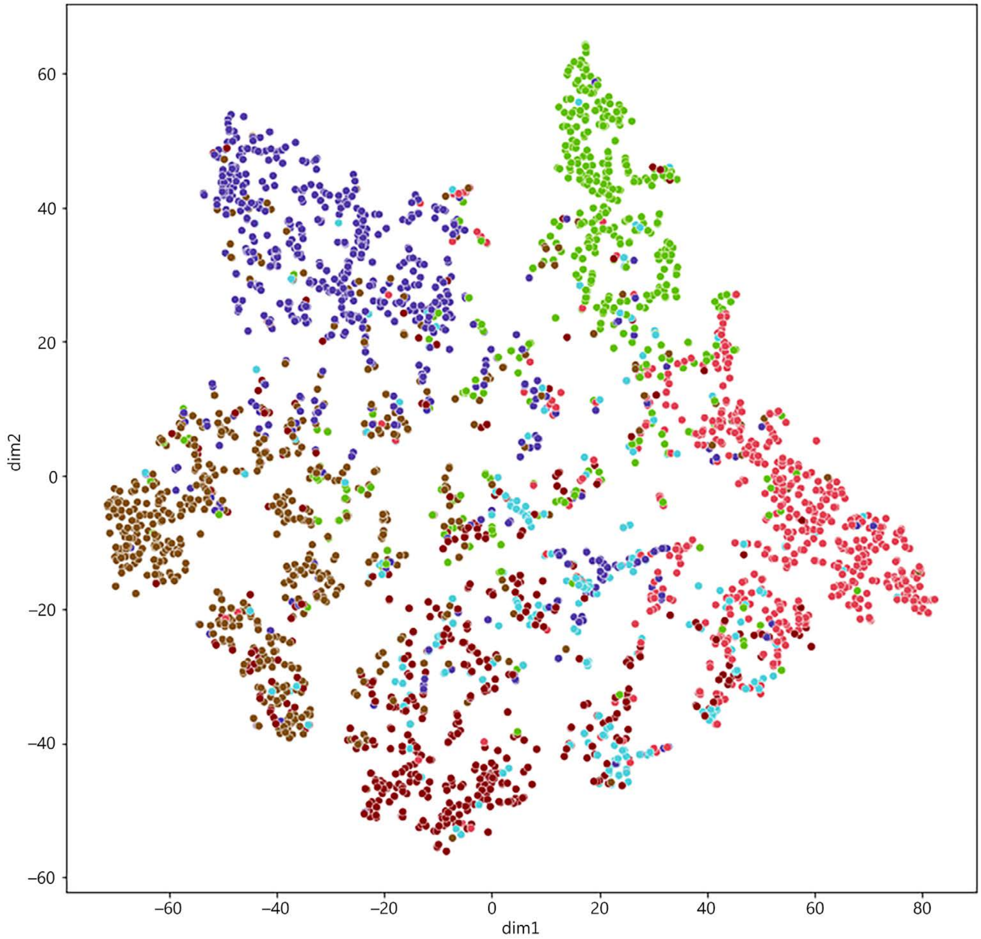


Figure 6.20: Node embeddings produced by the GAT node classifier after training

Firstly, compared to *Figure 6.19*, the nodes are no longer randomly scattered and are visibly clustered, which indicates the model has indeed been trained correctly. But also, this model seems to have learned even better node representations than the GCN model, as is evident from the clearer separation of nodes of different classes in *Figure 6.20* as compared to *Figure 6.17*.

This brings us to the end of exploring GNNs using PyTorch Geometric. While we have covered GCN and GAT-based node-classification models on the CiteSeer dataset, PyTorch Geometric provides a wide range of functionalities that I encourage you to explore [8], such as working on edge classification and graph classification tasks, using some other model types such as GraphSAGE, or working on different, perhaps larger graph datasets offered by this library, to further strengthen your GNN skills with PyTorch.

Summary

In this chapter, we first had a brief overview of GNNs. We understood the different types of graph-learning tasks. Next, we looked at some of the well-known GNN models. Finally, we worked on a few hands-on exercises on the CiteSeer graph dataset using the PyTorch Geometric library. Among the exercises, we trained a feedforward NN model on graph data on the task of node classification. We then replaced the feedforward NN model with a GCN model, which significantly improved classification accuracy. Finally, we replaced the GCN model with a GAT model to further enhance model performance. In the next chapter, we will switch gears and learn about music and text generation with PyTorch..

Reference list

1. GCNs: `https://tkipf.github.io/graph-convolutional-networks/`
2. GANs: `https://arxiv.org/abs/1710.10903`
3. Inductive Representation Learning on Large Graphs: `https://cs.stanford.edu/people/jure/pubs/graphsage-nips17.pdf`
4. PyG Documentation: `https://pytorch-geometric.readthedocs.io/en/latest/index.html`
5. GitHub 1: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter06/GNN.ipynb`
6. Software for Complex Networks: `https://networkx.org/documentation/stable/index.html`
7. How to Use t-SNE Effectively: `https://distill.pub/2016/misread-tsne/`
8. Colab Notebooks and Video Tutorials: `https://pytorch-geometric.readthedocs.io/en/latest/notes/colabs.html`

Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below to get a free eBook of your choice.

7

Music and Text Generation with PyTorch

PyTorch is a fantastic tool for both researching deep learning models and developing deep learning-based applications. In the previous chapters, we looked at model architectures across various domains and model types. We used PyTorch to build these architectures from scratch and used pre-trained models from the PyTorch model zoo. We will switch gears from this chapter onward and dive deep into generative models.

In the previous chapters, most of our examples and exercises revolved around developing models for classification, which is a supervised learning task. However, deep learning models have also proven extremely effective when it comes to unsupervised learning tasks. Deep generative models are one such example. These models are trained using lots of unlabeled data. Once trained, the model can generate similar meaningful data. It does so by learning the underlying structure and patterns in the input data.

In this chapter, we will develop text and music generators. To develop the text generator, we will utilize the transformer-based language model we trained in *Chapter 5*, *Hybrid Advanced Models*. We will extend the transformer model using PyTorch so that it works as a text generator. Furthermore, we will demonstrate how to use advanced pre-trained transformer models such as GPT-2 and GPT-3 in PyTorch to set up a text generator in a few lines of code. Finally, we will build a music generator model that's been trained on a MIDI dataset from scratch using PyTorch.

By the end of this chapter, you should be able to create your own text and music generation models in PyTorch. You will also be able to apply different sampling or generation strategies to generate data from such models. This chapter covers the following topics:

- Building a transformer-based text generator with PyTorch
- Using GPT (Generative Pre-trained Transformer) models as text generators
- Generating MIDI music with LSTMs using PyTorch

Building a transformer-based text generator with PyTorch

We built a transformer-based language model using PyTorch in the previous chapter. Because a language model models the probability of a certain word following a given sequence of words, we are more than halfway through building our own text generator. In this section, we will learn how to extend this language model as a deep generative model that can generate arbitrary yet meaningful sentences, given an initial textual cue in the form of a sequence of words.

Training the transformer-based language model

In the previous chapter, we trained a language model for 5 epochs. In this section, we will follow those exact same steps but will train the model for longer – 50 epochs. The goal here is to obtain a better-performing language model that can then generate realistic sentences. Please note that model training can take several hours. Hence, train it in the background; for example, overnight. In order to follow the steps for training the language model, please follow the complete code at GitHub [1].

Upon training for 50 epochs, we get the following output:

```
epoch 1, 100/1000 batches, training loss 8.81, training perplexity 6724.85
epoch 1, 200/1000 batches, training loss 7.35, training perplexity 1555.26
epoch 1, 300/1000 batches, training loss 6.90, training perplexity 991.85
epoch 1, 400/1000 batches, training loss 6.67, training perplexity 792.05
epoch 1, 500/1000 batches, training loss 6.54, training perplexity 694.65
epoch 1, 600/1000 batches, training loss 6.39, training perplexity 597.00
...
epoch 50, 600/1000 batches, training loss 4.45, training perplexity 86.05
epoch 50, 700/1000 batches, training loss 4.46, training perplexity 86.37
epoch 50, 800/1000 batches, training loss 4.33, training perplexity 76.17
epoch 50, 900/1000 batches, training loss 4.39, training perplexity 80.44
epoch 50, 1000/1000 batches, training loss 4.45, training perplexity 85.74
epoch 50, validation loss 5.07, validation perplexity 159.50
```

Now that we have successfully trained the transformer model for 50 epochs, we can move on to the actual exercise, where we will extend this trained language model as a text generation model.

Saving and loading the language model

Here, we will simply save the best-performing model checkpoint once the training is complete. We can then separately load this pre-trained model:

1. Once the model has been trained, it is ideal to save it locally so that you avoid having to retrain it from scratch. You can save it as follows:

   ```
   mdl_pth = './transformer.pth'
   torch.save(best_model_so_far.state_dict(), mdl_pth)
   ```

2. We can now load the saved model so that we can extend this language model as a text generation model:

```
# load the best trained model
transformer_cached = Transformer(
    num_tokens, embedding_size, num_heads,
    num_hidden_params, num_layers, dropout).to(device)
transformer_cached.load_state_dict(torch.load(mdl_pth))
```

In this section, we re-instantiated a transformer model object and then loaded the pre-trained model weights into this new model object. Next, we will use this model to generate text.

Using the language model to generate text

Now that the model has been saved and loaded, we can extend the trained language model to generate text:

1. First, we must define the target number of words we want to generate and provide an initial sequence of words as a cue to the model:

```
ln = 5
sntc = 'They are  _'
sntc_split = sntc.split()
mask_source = gen_sqr_nxt_mask(max_seq_len).to(device)
```

2. Finally, we can generate the words one by one in a loop. At each iteration, we can append the predicted word in that iteration to the input sequence. This extended sequence becomes the input to the model in the next iteration, and so on. The random seed is added to ensure consistency. By changing the seed, we can generate different texts, as shown in the following code block:

```
torch.manual_seed(34)
with torch.no_grad():
    for i in range(ln):
        sntc = ' '.join(sntc_split)
        txt_ds = Tensor(
            vocabulary(sntc_split)).unsqueeze(0).to(torch.long)
        num_b = txt_ds.size(0)
        txt_ds = txt_ds.narrow(0, 0, num_b)
        txt_ds = txt_ds.view(1, -1).t().contiguous().to(device)
        ev_X, _ = return_batch(txt_ds, i+1)
        sequence_length = ev_X.size(0)
        if sequence_length != max_seq_len:
            mask_source = mask_source[:sequence_length,
                                      :sequence_length]
        op = transformer_cached(ev_X, mask_source)
        op_flat = op.view(-1, num_tokens)
        res = vocabulary.get_itos()[op_flat.argmax(1)[0]]
        sntc_split.insert(-1, res)

print(sntc[:-2])
```

This should output the following:

```
They are often used for the
```

As we can see, using PyTorch, we can train a language model (a transformer-based model, in this case) and then use it to generate text with a few additional lines of code. The generated text seems to make sense. The result of such text generators is limited by the amount of data the underlying language model is trained on, as well as how powerful the language model is. In this section, we have essentially built a text generator from scratch.

In the next section, we will load the pre-trained language model and use it as a text generator. We will be using an advanced successor of the transformer model – the **generative pre-trained transformer** (**GPT-2**, and **GPT-3**). We will demonstrate how to build an out-of-the-box advanced text generator using PyTorch in less than 10 lines of code. We will also look at some strategies involved in generating text from a language model.

Using GPT models as text generators

Using libraries such as Hugging Face's `transformers` or `openai` together with PyTorch, we can load most of the latest advanced transformer models for performing various tasks such as language modeling, text classification, machine translation, and so on. We demonstrated how to do so in *Chapter 5*, *Advanced Hybrid Models*.

In this section, we will first load the GPT-2-language model using `transformers`. We will then extend this 1.5-billion-parameters model so that we can use it as a text generator. Then, we will explore the various strategies we can follow to generate text from a pre-trained language model and use PyTorch to demonstrate those strategies.

Finally, we will load the 175-billion-parameters GPT-3 model using `openai` and demonstrate its capability to generate realistic natural language.

Out-of-the-box text generation with GPT-2

In the form of an exercise, we will load the GPT-2 language model using the `transformers` library and extend this language model as a text generation model to generate arbitrary yet meaningful texts. We will only show the important parts of the code for demonstration purposes. In order to access the full code, go to GitHub [2]. Follow these steps:

1. First, we need to import the necessary libraries:

   ```
   from transformers import GPT2LMHeadModel, GPT2Tokenizer
   import torch
   ```

   We will import the GPT-2 multi-head language model [3] and corresponding tokenizer [4] to generate the vocabulary.

2. Next, we will instantiate `GPT2Tokenizer` and the language model. And then, we will provide an initial set of words as a cue to the model, as follows:

   ```
   torch.manual_seed(799)
   tkz = GPT2Tokenizer.from_pretrained("gpt2")
   mdl = GPT2LMHeadModel.from_pretrained('gpt2')
   ln = 10
   cue = "They"
   gen = tkz(cue, return_tensors="pt")
   to_ret = gen["input_ids"][0]
   ```

3. Finally, we will iteratively predict the next word for a given input sequence of words using the language model. At each iteration, the predicted word is appended to the input sequence of words for the next iteration:

   ```
   prv=None
   for i in range(ln):
       outputs = mdl(**gen)
       next_token_logits = torch.argmax(outputs.logits[-1, :])
       to_ret = torch.cat([to_ret, next_token_logits.unsqueeze(0)])
       gen = {"input_ids": to_ret}
   seq = tkz.decode(to_ret)
   print(seq)
   ```

   The output should be as follows:

   ```
   They are not the only ones who are being targeted.
   ```

This way of generating text is also called **greedy search**. In the next section, we will look at greedy search in more detail and some other text generation strategies as well.

Text generation strategies using PyTorch

When we use a trained text generation model to generate text, we typically make predictions word by word. We then consolidate the resulting sequence of predicted words as predicted text. When we are in a loop iterating over word predictions, we need to specify a method of finding/predicting the next word given the previous *k* predictions. These methods are also known as text generation strategies, and we will discuss some well-known strategies in this section.

Greedy search

The name *greedy* is justified by the fact that the model selects the word with the maximum probability at the current iteration, regardless of how many time steps further ahead they are. With this strategy, the model could potentially miss a highly probable word hiding (further ahead in time) behind a low-probability word merely because the model did not pursue the low-probability word at the current time-step. *Figure 7.1* demonstrates the greedy search strategy by illustrating a hypothetical scenario of what might be happening under the hood in *step 3* of the previous exercise. At each time step, the text generation model outputs possible words, along with their probabilities:

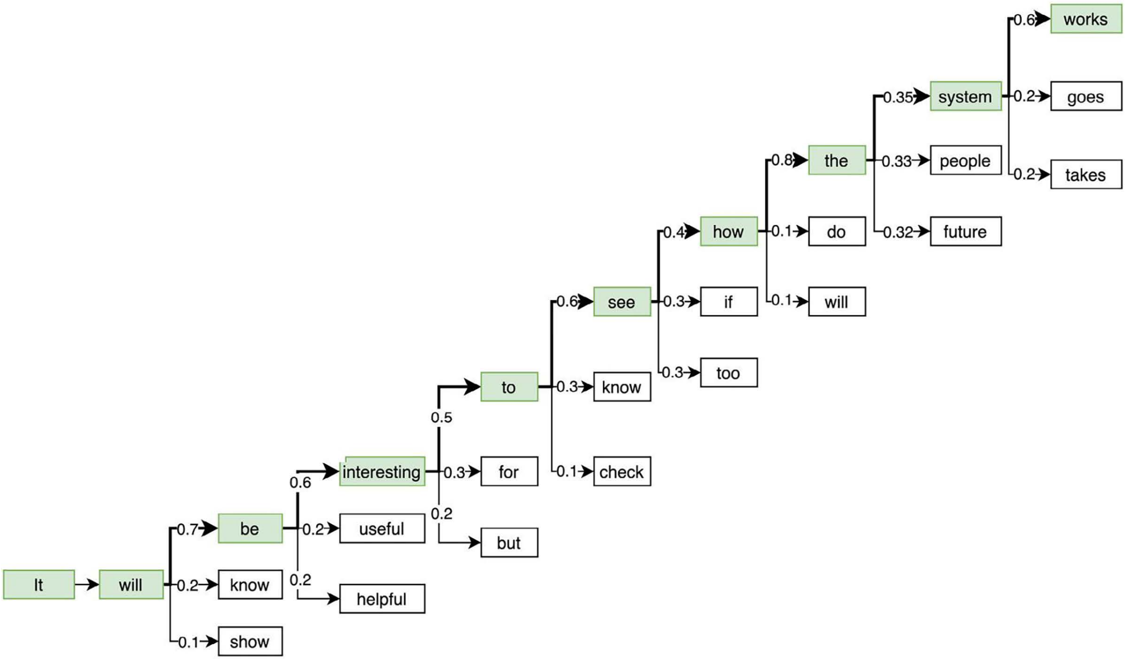


Figure 7.1: Greedy search

As we can see, at each step, the word with the highest probability is picked up by the model under the greedy search strategy of text generation. Note the penultimate step, where the model predicts the words *system*, *people*, and *future* with roughly equal probabilities. With greedy search, *system* is selected as the next word due to it having a slightly higher probability than the rest. However, you could argue that *people* or *future* could have led to a better or more meaningful generated text.

This is the core limitation of the greedy search approach. Besides, greedy search also results in repetitive results due to a lack of randomness. If someone wants to use such a text generator artistically, greedy search is not the best approach, merely due to its monotonicity.

In the previous section, we manually wrote the text generation loop. Thanks to the `transformers` library, we can write the text generation step in three lines of code:

```python
ip_ids = tkz.encode(cue, return_tensors='pt')
op_greedy = mdl.generate(ip_ids, max_length=ln, pad_token_id=tkz.eos_token_id)
seq = tkz.decode(op_greedy[0], skip_special_tokens=True)
print(seq)
```

This should output the following:

```
They are not the only ones who are being targeted
```

Notice that the generated sentence has one less token (full-stop) than the sentence that was generated using a manual text-generation loop. This difference is because, in the latter code, the `max_length` argument includes the cue words. So, if we have one cue word, only nine new words would be predicted, as is the case here.

Beam search

Greedy search is not the only way of generating texts. **Beam search** is a development of the greedy search method wherein we maintain a list of potential candidate sequences based on the overall predicted sequence probability rather than just the next-word probability. The number of candidate sequences to be pursued is the number of beams along the tree of word predictions.

Figure 7.2 demonstrates how beam search with a beam size of three would be used to produce three candidate sequences (ordered as per the overall sequence probability) of five words each:

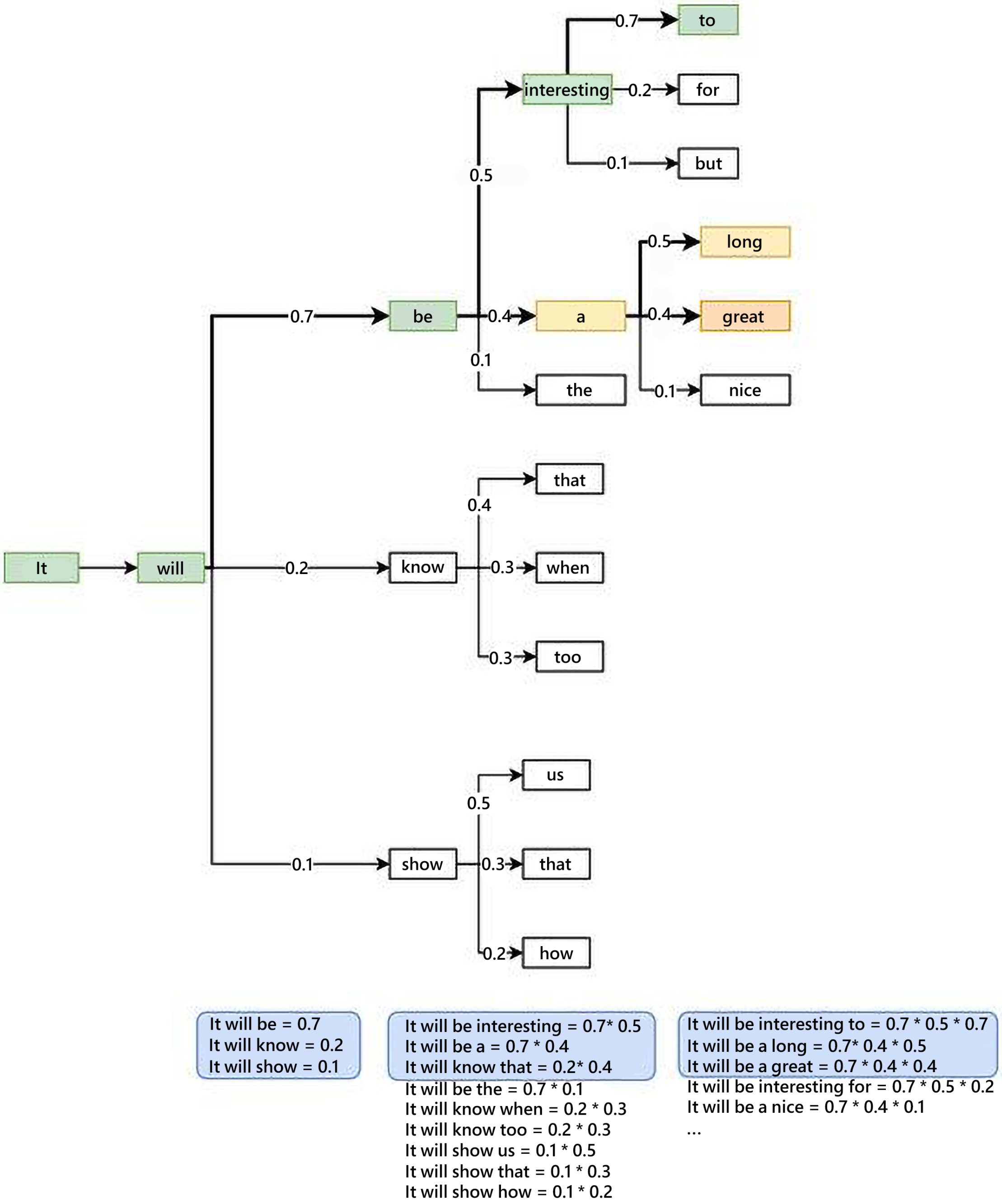

to
0.7
interesting
0.2
for
0.1
but
0.5
long
0.5
be
0.7
0.4
a
0.4
great
0.1
the
0.1
nice
that
0.4
It
will
0.2
know
0.3
when
0.3
too
us
0.5
0.1
show
0.3
that
0.2
how
It will be = 0.7
It will know = 0.2
It will show = 0.1
It will be interesting = 0.7* 0.5
It will be a = 0.7 * 0.4
It will know that = 0.2* 0.4
It will be the = 0.7 * 0.1
It will know when = 0.2 * 0.3
It will know too = 0.2 * 0.3
It will show us = 0.1 * 0.5
It will show that = 0.1 * 0.3
It will show how = 0.1 * 0.2
It will be interesting to = 0.7 * 0.5 * 0.7
It will be a long = 0.7* 0.4 * 0.5
It will be a great = 0.7 * 0.4 * 0.4
It will be interesting for = 0.7 * 0.5 * 0.2
It will be a nice = 0.7 * 0.4 * 0.1
...

Figure 7.2: Beam search

At each iteration in this beam search example, the three most likely candidate sequences are maintained. As we proceed further in the sequence, the possible number of candidate sequences increases exponentially. However, we are only interested in the top three sequences. This way, we do not miss potentially better sequences as we might with greedy search.

In PyTorch, we can use beam search out of the box in one line of code. The following code demonstrates beam search-based text generation with three beams generating the three most likely sentences, each containing five words:

```
op_beam = mdl.generate(
    ip_ids,
    max_length=5,
    num_beams=3,
    num_return_sequences=3,
    pad_token_id=tkz.eos_token_id
)
for op_beam_cur in op_beam:
    print(tkz.decode(op_beam_cur, skip_special_tokens=True))
```

This gives us the following output:

```
They have a lot of
They have a lot to
They are not the only
```

The problem of repetitiveness or monotonicity still remains with the beam search. Different runs would result in the same set of results as it deterministically looks for the sequence with the maximum overall probabilities. In the next section, we will look at some of the ways we can make the generated text more unpredictable or creative.

Top-k and top-p sampling

Instead of always picking the next word with the highest probability, we can randomly sample the next word out of the possible set of next words based on their relative probabilities. For example, in *Figure 7.2*, the words *be*, *know*, and *show* have probabilities of 0.7, 0.2, and 0.1, respectively. Instead of always picking *be* against *know* and *show*, we can randomly sample any one of these three words based on their probabilities. If we repeat this exercise 10 times to generate 10 separate texts, *be* will be chosen roughly seven times, and *know* and *show* will be chosen two and one times, respectively. This gives us far too many different possible combinations of words that beam or greedy search would never generate.

Two of the most popular ways of generating texts using sampling techniques are known as **top-k** and **top-p** sampling. Under top-k sampling, we predefine a parameter, *k*, which is the number of candidate words that should be considered while sampling the next word. All the other words are discarded, and the probabilities are normalized among the top *k* words. In our previous example, if *k* is 2, then the word *show* will be discarded, and the words *be* and *know* will have their probabilities (0.7 and 0.2, respectively) normalized to 0.78 and 0.22, respectively.

The following code demonstrates the top-k text generation method:

```
for i in range(3):
    torch.manual_seed(i+10)
    op = mdl.generate(
        ip_ids,
        do_sample=True,
        max_length=5,
        top_k=2,
        pad_token_id=tkz.eos_token_id
    )
    seq = tkz.decode(op[0], skip_special_tokens=True)
    print(seq)
```

This should generate the following output:

```
They are the most important
They have a lot to
They are not going to
```

To sample from all possible words, instead of just the top-k words, we shall set the `top_k` argument to `0` in our code. As shown in the preceding code output, different runs produce different results, as opposed to greedy search, which would result in the exact same result on each run, as shown in the following code:

```
for i in range(3):
    torch.manual_seed(i+10)
    op_greedy = mdl.generate(ip_ids, max_length=5, pad_token_id=tkz.eos_token_id)
    seq = tkz.decode(op_greedy[0], skip_special_tokens=True)
    print(seq)
```

This should output the following:

```
They are not the only
They are not the only
They are not the only
```

Under the top-p sampling strategy, instead of defining the top k words to look at, we can define a cumulative probability threshold (p) and then retain words whose probabilities add up to p. In our example, if p is between 0.7 and 0.9, then we discard *know* and *show*; if p is between 0.9 and 1.0, then we discard *show*, and if p is 1.0, then we keep all three words; that is, *be*, *know*, and *show*.

The top-k strategy can sometimes be unfair in scenarios where the probability distribution is flat. This is because it clips off words that are almost as probable as the ones that have been retained. In those cases, the top-p strategy would retain a larger pool of words to sample from and would retain a smaller pool of words in cases where the probability distribution is rather sharp.

The following code demonstrates the top-p sampling method:

```
for i in range(3):
    torch.manual_seed(i+10)
    op = mdl.generate(
        ip_ids,
        do_sample=True,
        max_length=5,
        top_p=0.75,
        top_k=0,
        pad_token_id=tkz.eos_token_id
    )
    seq = tkz.decode(op[0], skip_special_tokens=True)
    print(seq)
```

This should output the following:

```
They got them here in
They have also challenged foreign
They said it would be
```

We can set both top-k and top-p strategies together. In this example, we have set `top_k` to `0` to essentially disable the top-k strategy, and `top_p` is set to `0.75`. Once again, this results in different sentences across runs and can lead us to more creatively generated texts as opposed to greedy or beam searches. There are many more text-generation strategies available, and a lot of research is happening in this area. We encourage you to follow up on this further.

A great starting point is playing around with the available text generation strategies in the `transformers` library. You can read more about it in their blog post [5]. Let us now look at GPT-2's successor – GPT-3, and use this model via the `openai` library to generate meaningful text.

Text generation with GPT-3

As in the previous exercise, we will load a pre-trained GPT-3 model and use it as a text generator. We will use APIs provided under the `openai` library. OpenAI is the company that developed both the GPT-2 and GPT-3 models. To access the GPT-3 model, we need an OpenAI API key [6], which requires creating an OpenAI account.

Once we have the API key, we can head over to our mini exercise, the code for which is provided on GitHub [7].

1. First, we need to import the necessary libraries and set the environment variables:

   ```
   import os
   from openai import OpenAI
   client = OpenAI(api_key = "<your-open-ai-api-key-here>")
   ```

2. And then, we define our prompt (the same as we did before):

   ```
   prompt = "They"
   ```

3. Finally, with one code statement, we can prompt the GPT-3 model to generate text using OpenAI's `Completion.create` API:

   ```
   response = client.chat.completions.create(
     model="gpt-3.5-turbo-instruct",
     response_format={ "type": "json_object" },
     messages=[
       {"role": "user", "content": prompt}
     ],
     temperature=0.5,
     max_tokens=5,
     top_p=1.0,
     frequency_penalty=0.0,
     presence_penalty=0.0
   )
   ```

   The arguments `temperature` and `top_p` are used to tune the sampling strategy of the next word in the sequence, where a higher value of either parameter results in more randomness in sampling. However, it is advised to vary one of these, not both at the same time. You can read about them in greater detail under OpenAI's API reference [8].

   The precise model used here is called `gpt-3.5-turbo-instruct`, which is one of the most capable GPT-3 model according to OpenAI [9].

4. We can now print the response:

   ```
   print(response.choices[0].message.content)
   ```

   This should produce something like the following:

   ```
   are an important part of
   ```

   Because we have chosen `top_p=1`, it is likely that you might observe a different yet meaningful result. However, the number of words will be limited to 5 as specified under the `max_tokens` argument.

5. You might be wondering why we need 175 billion parameters just to generate a meaningful English sentence starting with *they*. GPT-3 is capable of much more. Here is one example:

   ```
   prompt = "Write a poem starting with they"
   response = client.chat.completions.create(
     model="gpt-3.5-turbo-instruct",
     response_format={ "type": "json_object" },
     messages=[
   ```

```
    {"role": "user", "content": prompt}
  ],
  temperature=0.5,
  max_tokens=100,
  top_p=1.0,
)
```

I got the following output from the above code:

```
They always say
That life is a mystery
And we will never really know
What happens after we die
But I think
That we can be pretty sure
That there is something
After this life
Something better
And we will finally be able
To rest in peace
```

Hopefully, this example establishes the range of capabilities of the GPT-3 model.

Your creativity is the limit when it comes to prompting such powerful models. In fact, there is a whole new discipline that emerged in light of such powerful language models called prompt engineering. OpenAI's website has a good guide on prompting GPT-3 model for different use cases [10]. Finally, I highly recommend Packt's *Exploring-GPT-3* [11] to further explore GPT-3.

This concludes our exploration of using PyTorch to generate text. In the next section, we will perform a similar exercise but this time for music instead of text. The idea is to train an unsupervised model on a music dataset and use the trained model to generate melodies like those in the training dataset.

Generating MIDI music with LSTMs using PyTorch

Moving on from text, in this section, we will use PyTorch to create a machine learning model that can compose classical-like music. We used transformers to generate text in the previous section. Here, we will use a **Long Short-Term Memory** (**LSTM**) model to process sequential music data. We will train the model on Mozart's classical music compositions.

Each musical piece will essentially be broken down into a sequence of piano notes. We will be reading music data in the form of **Musical Instruments Digital Interface** (**MIDI**) files, which is a well-known and commonly used format for conveniently reading and writing musical data across devices and environments.

After converting the MIDI files into sequences of piano notes (which we call the piano roll), we will use them for training a next-piano-note detection system. In this system, we will build an LSTM-based classifier that will predict the next piano note for the given preceding sequence of piano notes, of which there are 88 in total (as per the standard 88 piano keys).

We will now demonstrate the entire process of building the AI music composer in the form of an exercise. Our focus will be on the PyTorch code that's used for data loading, model training, and generating music samples. Please note that the model training process may take several hours and, therefore, it is recommended to run the training process in the background; for example, overnight. The code presented here has been curtailed in the interest of keeping the text short.

Details of handling the MIDI music files are beyond the scope of this book, although you are encouraged to explore the full code, which is available on GitHub [12].

Loading the MIDI music data

First, we will demonstrate how to load the music data that is available in MIDI format. We will briefly mention the code for handling MIDI data and then illustrate how to make PyTorch dataloaders out of it. Let's get started:

1. As always, we will begin by importing the important libraries. Some of the new ones we'll be using in this exercise are as follows:

   ```
   import skimage.io as io
   from struct import pack, unpack
   from io import StringIO, BytesIO
   ```

   `skimage` is used to visualize the sequences of the music samples that are generated by the model. `struct` and `io` are used for handling the process of converting MIDI music data into piano rolls.

2. Next, we will write the helper classes and functions for loading MIDI files and converting them into sequences of piano notes (matrices) that can be fed to the LSTM model. First, we define some MIDI constants in order to configure various music controls such as pitch, channels, the start of the sequence, the end of the sequence, and so on:

   ```
   NOTE_MIDI_OFF = 0x80
   NOTE_MIDI_ON = 0x90
   CHNL_PRESS = 0xD0
   MIDI_PITCH_BND = 0xE0
   ...
   ```

   Then, we will define a series of classes that will handle MIDI data input and output streams, the MIDI data parser, and so on, as follows:

   ```
   class MOStrm:
   # MIDI Output Stream
   ...
   class MIFl:
   ```

```
# MIDI Input File Reader
...
class MOFl(MOStrm):
# MIDI Output File Writer
...
class RIStrFl:
# Raw Input Stream File Reader
...
class ROStrFl:
# Raw Output Stream File Writer
...
class MFlPrsr:
# MIDI File Parser
...
class EvtDspch:
# Event Dispatcher
...
class MidiDataRead(MOStrm):
# MIDI Data Reader
...
```

3. Having handled all the MIDI data I/O-related code, we are all set to instantiate our own PyTorch dataset class. Before we do that, we must define two crucial functions – one for converting the read MIDI file into a piano roll and one for padding the piano roll with empty notes. This will normalize the lengths of the musical pieces across the dataset:

```
def md_fl_to_pio_rl(md_fl):
    md_d = MidiDataRead(md_fl, dtm=0.3)
    pio_rl = md_d.pio_rl.transpose()
    pio_rl[pio_rl > 0] = 1
    return pio_rl
def pd_pio_rl(pio_rl, mx_l=132333, pd_v=0):
    orig_rol_len = pio_rl.shape[1]
    pdd_rol = np.zeros((88, mx_l))
    pdd_rol[:] = pd_v
    pdd_rol[:, - orig_rol_len:] = pio_rl
    return pdd_rol
```

Now, we can define our PyTorch dataset class, as follows:

```
class NtGenDataset(data.Dataset):
    def __init__(self, md_pth, mx_seq_ln=1491):
        ...
    def mx_len_upd(self):
        ...
    def __len__(self):
        return len(self.md_fnames_ful
    def __getitem__(self, index):
        md_fname_ful = self.md_fnames_ful[index]
        pio_rl = md_fl_to_pio_rl(md_fname_ful)
        seq_len = pio_rl.shape[1] - 1
        ip_seq = pio_rl[:, :-1]
        gt_seq = pio_rl[:, 1:]
        ...
        return (torch.FloatTensor(ip_seq_pad),
                torch.LongTensor(gt_seq_pad),
                torch.LongTensor([seq_len]))
```

4. Besides the dataset class, we must add another helper function to post-process the music sequences in a batch of training data into three separate lists. These will be input sequences, output sequences, and lengths of sequences, ordered by the lengths of the sequences in descending order:

```
def pos_proc_seq(btch):
    ip_seqs, op_seqs, lens = btch
    ...
    ord_tr_data_tups = sorted(tr_data_tups,
                              key=lambda c: int(c[2]),
                              reverse=True)
    ip_seq_splt_btch, op_seq_splt_btch, btch_splt_lens = \
        zip(*ord_tr_data_tups)
    ...
    return tps_ip_seq_btch, ord_op_seq_btch, list(
        ord_btch_lens_l)
```

5. For this exercise, we will be using a set of Mozart's compositions. You can download the dataset from the piano-midi website [13]. The downloaded folder consists of 21 MIDI files, which we will split into 18 training and 3 validation set files. The downloaded data is stored under `./mozart/train` and `./mozart/valid`. Once downloaded, we can read the data and instantiate our own training and validation dataset loaders:

```
training_dataset = NtGenDataset(
    './mozart/train', mx_seq_ln=None)
training_datasetloader = data.DataLoader(
    training_dataset, batch_size=5,shuffle=True,
    drop_last=True)
```

```
validation_dataset = NtGenDataset(
    './mozart/valid/', mx_seq_ln=None)
validation_datasetloader = data.DataLoader(
    validation_dataset, batch_size=3,
    shuffle=False, drop_last=False)
X_validation = next(iter(validation_datasetloader))
X_validation[0].shape
```

This should give us the following output:

```
torch.Size([3, 1587, 88])
```

As we can see, the first validation batch consists of 3 sequences of length 1,587 (notes), where each sequence is encoded into an 88-size vector, with 88 being the total number of piano keys. For those of you who are trained musicians, here is a music sheet equivalent of the first few notes of one of the validation set music files:

Figure 7.3: Music sheet of a Mozart composition

Alternatively, we can visualize the sequence of notes as a matrix with 88 rows, one per piano key. The following is a visual matrix representation of the preceding melody (the first 300 notes out of 1,587):

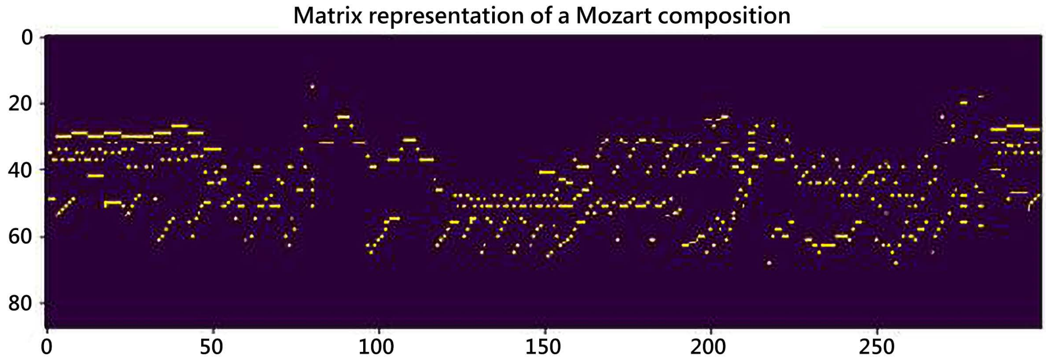


Figure 7.4: Matrix representation of a Mozart composition

Dataset citation

The MIDI, audio (MP3, OGG), and video files of Bernd Krueger are licensed under the CC BY-SA Germany License. Name: Bernd Krueger. Source: `http://www.piano-midi.de`. The distribution or public playback of these files is only allowed under identical license conditions. The scores are open source.

We will now define the LSTM model and training routine.

Defining the LSTM model and training routine

So far, we have managed to successfully load a MIDI dataset and use it to create our own training and validation data loaders. In this section, we will define the LSTM model architecture, as well as the training and evaluation routines that shall be run during the model training loop. Let's get started:

1. First, we must define the model architecture. As we mentioned earlier, we will use an LSTM model that consists of an encoder layer that encodes the 88-dimensional representation of the input data at each time step of the sequence into a 512-dimensional hidden layer representation. The encoder is followed by two LSTM layers, followed by a fully connected layer, and finally, a softmax layer, which produces 88 probabilities for 88 classes (piano keys).
2. As per the different types of **recurrent neural networks** (**RNNs**) we discussed in *Chapter 4*, *Deep Recurrent Model Architectures*, this is a many-to-one sequence classification task, where the input is the entire sequence from time step *0* to time step *t* and the output is one of the 88 classes at time step *t+1*, as follows:

```
class MusicLSTM(nn.Module):
    def __init__(self, ip_sz, hd_sz, n_cls, lyrs=2):
        ...
        self.nts_enc = nn.Linear(in_features=ip_sz, out_features=hd_sz)
        self.bn_layer = nn.BatchNorm1d(hd_sz)
        self.lstm_layer = nn.LSTM(hd_sz, hd_sz, lyrs)
        self.fc_layer = nn.Linear(hd_sz, n_cls)

    def forward(self, ip_seqs, ip_seqs_len, hd=None):
        ...
        pkd = torch.nn.utils.rnn.pack_padded_sequence(
            nts_enc_ful, ip_seqs_len)
        op, hd = self.lstm_layer(pkd, hd)
        ...
        lgts = self.fc_layer(op_nrm_drp.permute(2,0,1))
        ...
        zero_one_lgts = torch.stack(
            (lgts, rev_lgts), dim=3).contiguous()
        flt_lgts = zero_one_lgts.view(-1, 2)
        return flt_lgts, hd
```

3. Once the model architecture has been defined, we can specify the model training routine. We will use the Adam optimizer with gradient clipping to avoid overfitting. Another measure that's already in place to counter overfitting is the use of a dropout layer, as specified in the previous step:

```
def lstm_model_training(
    lstm_model, lr, ep=10, val_loss_best=float("inf")):
    ...
    for curr_ep in range(ep):
        ...
        for batch in training_datasetloader:
            ...
            lgts, _ = lstm_model(ip_seq_b_v, seq_l)
            loss = loss_func(lgts, op_seq_b_v)
            ...
        if vl_ep_cur < val_loss_best:
            torch.save(lstm_model.state_dict(), 'best_model.pth')
            val_loss_best = vl_ep_cur
    return val_loss_best, lstm_model
```

4. Similarly, we will define the model evaluation routine, where a forward pass is run on the model with its parameters remaining unchanged:

```
def evaluate_model(lstm_model):
    ...
    for batch in validation_datasetloader:
        ...
        lgts, _ = lstm_model(ip_seq_b_v, seq_l)
        loss = loss_func(lgts, op_seq_b_v)
        vl_loss_full += loss.item()
        seq_len += sum(seq_l)
    return vl_loss_full/(seq_len*88)
```

Now, let's train and test the music generation model.

Training and testing the music generation model

In this final section, we will actually train the LSTM model. We will then use the trained music generation model to generate a music sample that we can listen to and analyze. Let's get started:

1. We are all set to instantiate our model and start training it. We have used categorical cross-entropy as the loss function for this classification task. We are training the model with a learning rate of `0.01` for `10` epochs:

```
loss_func = nn.CrossEntropyLoss().cpu()
lstm_model = MusicLSTM(ip_sz=88, hd_sz=512, n_cls=88).cpu()
val_loss_best, lstm_model = lstm_model_training(lstm_model, lr=0.01, ep=10)
```

This should output the following:

```
ep 0 , train loss = 2.3905489842096963
ep 0 , val loss = 3.8042128349324635e-06
ep 1 , train loss = 0.9679248531659445
ep 1 , val loss = 2.122019823561201e-06
ep 2 , train loss = 0.2935091306765874
ep 2 , val loss = 1.2749193585637908e-06
...
ep 7 , train loss = 0.16012300054232279
ep 7 , val loss = 1.2555179474370303e-06
ep 8 , train loss = 0.12387428929408391
ep 8 , val loss = 1.4818597425925305e-06
ep 9 , train loss = 0.13243193179368973
ep 9 , val loss = 1.6489400508525355e-06
```

2. Here comes the fun part. Once we have a next-musical-note-predictor, we can use it as a music generator. All we need to do is simply initiate the prediction process by providing an initial note as a cue. The model can then recursively make predictions for the next note at each time step, wherein the predictions at time step *t* are appended to the input sequence at time *t+1*.

3. Here, we will write a music generation function that takes in the trained model object, the intended length of music to be generated, a starting note to the sequence, and temperature. Temperature is a standard mathematical operation over the `softmax` function at the classification layer. It is used to manipulate the distribution of softmax probabilities, either by broadening or shrinking the softmaxed probabilities distribution. The code is as follows:

```
def generate_music(lstm_model, ln=100, tmp=1, seq_st=None):
    ...
    for i in range(ln):
        op, hd = lstm_model(seq_ip_cur, [1], hd)
        probs = nn.functional.softmax(op.div(tmp), dim=1)
        ...
    gen_seq = torch.cat(op_seq, dim=0).cpu().numpy()
    return gen_seq
```

Finally, we can use this function to create a brand-new music composition:

```
seq = generate_music(lstm_model, ln=100, tmp=0.8, seq_st=None)
midiwrite('generated_music.mid', seq, dtm=0.2)
```

4. This should create the musical piece and save it as a MIDI file in the current directory. We can open the file and play it to hear what the model has produced. We can also view the visual matrix representation of the produced music:

```
io.imshow(seq)
```

This should give us the following output:

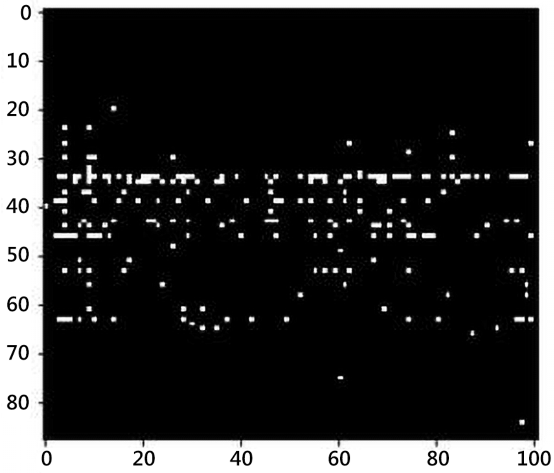


Figure 7.5: Matrix representation of an AI-generated music sample

Furthermore, here is what the generated music would look like as a music sheet:

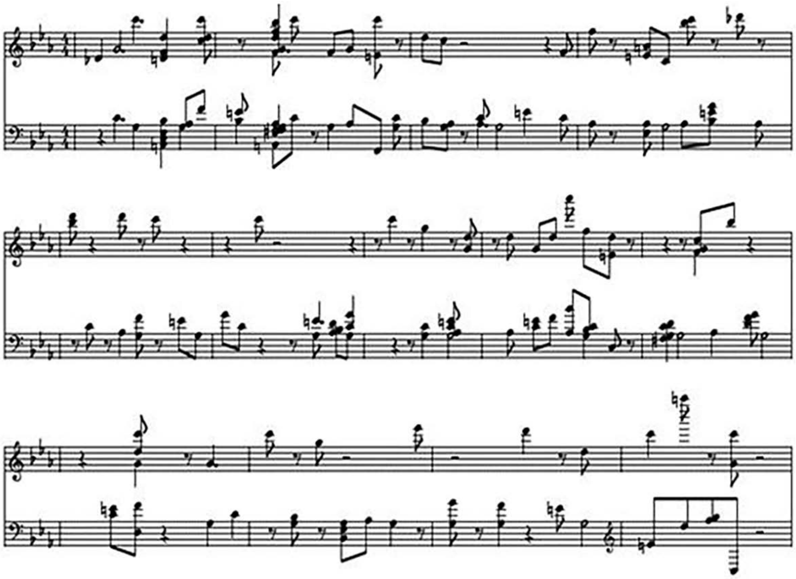


Figure 7.6: Music sheet of an AI-generated music sample

Here, we can see that the generated melody seems to be not quite as melodious as Mozart's original compositions. Nonetheless, you can see consistencies in some key combinations that the model has learned. Moreover, the generated music quality can be enhanced by training the model on more data, as well as training it for more epochs.

This concludes our exercise on using machine learning to generate music. In this section, we have demonstrated how to use existing musical data to train a note predictor model from scratch and use the trained model to generate music. In fact, you can extend the idea of using generative models to generate samples of any kind of data. PyTorch is an extremely effective tool when it comes to such use cases, especially due to its straightforward APIs for data loading, model building/training/testing, and using trained models as data generators. You are encouraged to try out more such tasks on different use cases and data types.

Summary

In this chapter, we explored generative models using PyTorch. In the same artistic vein, in the next chapter, we shall learn how machine learning can be used to transfer the style of one image to another. With PyTorch at our disposal, we will use CNNs to learn artistic styles from various images and impose those styles on different images – a task better known as neural style transfer.

References

1. GitHub Repository: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter07/text_generation.ipynb`
2. GitHub Repository (out of the box): `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter07/text_generation_out_of_the_box.ipynb`
3. GPT-2 multi-head language model: `https://huggingface.co/docs/transformers/model_doc/gpt2#transformers.GPT2LMHeadModel`
4. Tokenizer: `https://huggingface.co/docs/transformers/model_doc/gpt2#transformers.GPT2Tokenizer`
5. Hugging Face transforms blog: `https://huggingface.co/blog/how-to-generate`
6. OpenAI API Key: `https://platform.openai.com/account/api-keys`
7. GitHub Repository (Text Generation 3): `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter07/text_generation_out_of_the_box_gpt3.ipynb`
8. OpenAI API reference: `https://platform.openai.com/docs/api-reference/completions/create`
9. OpenAI Davinci: `https://platform.openai.com/docs/models/gpt-3`
10. Prompting GPT-3: `https://platform.openai.com/docs/guides/completion`
11. Exploring GPT-3: `https://www.packtpub.com/product/exploring-gpt-3/9781800563193`
12. GitHub Repository (MIDI music files): `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter07/music_generation.ipynb`
13. Piano-midi website: `http://www.piano-midi.de/mozart.htm`

Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below to get a free eBook of your choice.

8

Neural Style Transfer

In the previous chapter, we started exploring generative models using PyTorch. We built machine learning models that can generate text and music by training the models without supervision on text and music data, respectively. We will continue exploring generative modeling in this chapter by applying a similar methodology to image data.

We will mix different aspects of two different images, **A** and **B**, to generate a resultant image, **C**, that contains the content of image **A** and the style of image **B**. This task is also popularly known as **neural style transfer** because, in a way, we are transferring the style of image **B** to image **A** to achieve image **C**, as illustrated in *Figure 8.1*:

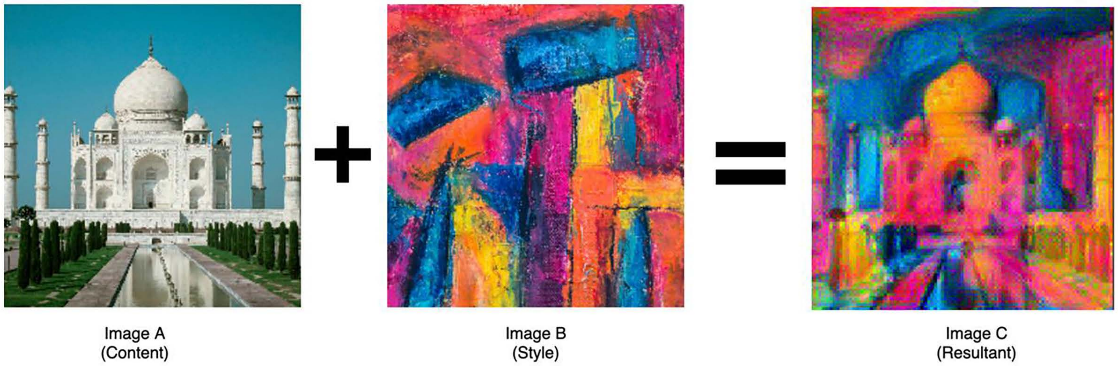


Figure 8.1: Neural style transfer example

First, we will briefly discuss how to approach this problem and understand the idea behind achieving style transfer. Using PyTorch, we will then implement our own neural style transfer system and apply it to a pair of images. Through this implementation exercise, we will also try to understand the effects of different parameters in the style transfer mechanism.

By the end of this chapter, you will understand the concepts behind neural style transfer and be able to build and test your own neural style transfer model using PyTorch.

This chapter covers the following topics:

- Understanding how to transfer style between images
- Implementing neural style transfer using PyTorch

All the code files for this chapter can be found at `https://github.com/arj7192/MasteringPyTorchV2/tree/main/Chapter08`.

Let us now look at the idea and related mathematics behind neural style transfer.

Understanding how to transfer style between images

In *Chapter 2*, *Deep CNN Architectures*, we discussed **convolutional neural networks** (**CNNs**) in detail. CNNs are some of the most successful models when working with image data on tasks such as image classification and object detection, among others. One of the core reasons behind this success is the ability of convolutional layers to learn spatial representations.

For example, in a dog versus cat classifier, the CNN model is essentially able to capture the content of an image while extracting higher-level features, which helps it detect dog-specific features against cat-specific features. We will leverage this ability of an image classifier CNN to grasp the content of an image.

We know that VGG is a powerful image classification model, as discussed in *Chapter 2*, *Deep CNN Architectures*. We are going to use the convolutional part of the VGG model (excluding the linear layers) to extract content-related features from an image.

We know that each convolutional layer produces, say, N feature maps of dimensions $X*Y$ each. For example, let's say we have a single channel (grayscale) input image of size (3,3) and a convolutional layer where the number of output channels (N) is 3, the kernel size is (2,2) with a stride of (1,1), and there's no padding. This convolutional layer will produce 3 (N) feature maps each of size 2x2, hence $X=2$ and $Y=2$ in this case.

We can represent these N feature maps produced by the convolutional layer as a 2D matrix of size $N*M$, where $M=X*Y$. By defining the output of each convolutional layer as a 2D matrix, we can define a loss function that's attached to each convolutional layer.

This loss function, called the **content loss**, is the squared loss between the expected and predicted outputs of the convolutional layers, as demonstrated in *Figure 8.2*, with *N*=3, *X*=2, and *Y*=2:

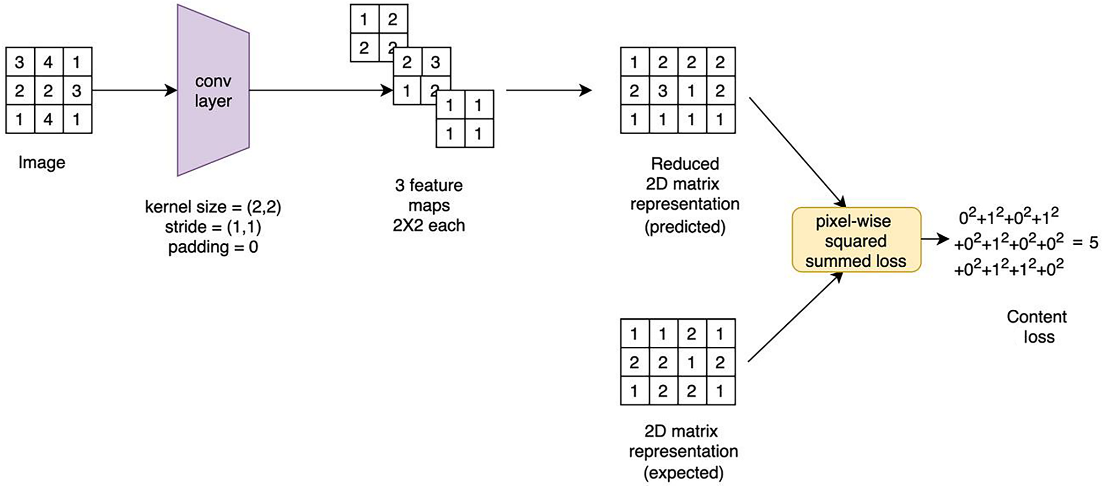


Figure 8.2: Content loss schematic

As we can see, the input image (image *C*, as per our notation in *Figure 8.1*) in this example is transformed into **three feature maps** by the **convolutional** (**conv**) **layer**. These three feature maps, of size 2x2 each, are formatted into a 3x4 matrix. This matrix is compared with the expected output, which is obtained by passing image *A* (the content image) through the same flow. The pixel-wise squared summed loss is then calculated, which we call the **content loss**.

Now, to extract the style from an image, we will use gram matrices [1] derived from the inner product between the rows of the reduced 2D matrix representations, as demonstrated in *Figure 8.3*:

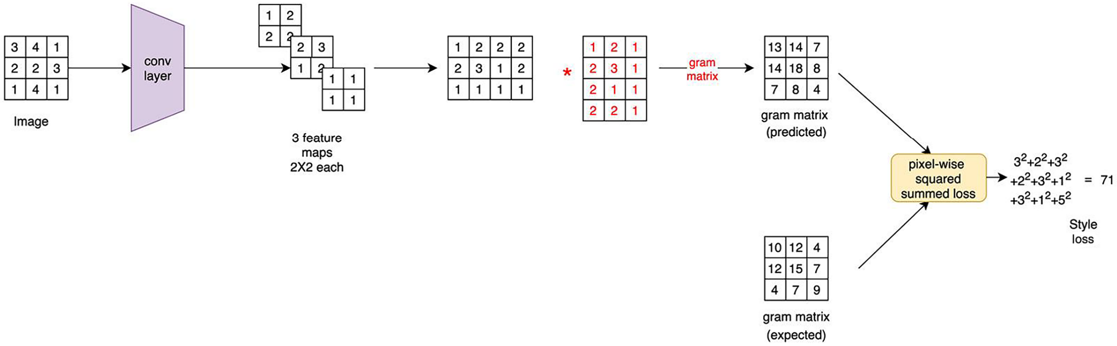


Figure 8.3: Style loss schematic

The gram matrix carries the correlations between different convolutional feature maps (three maps in our example in *Figure 8.3*). If the predicted gram matrix is close to the expected gram matrix, it means that the three convolutional feature maps in both cases (expected and predicted) have similar correlations (not absolute values) to each other. And correlations represent the relationship between pixels as opposed to the absolute pixel value. Hence, the gram matrix captures this relationship in the form of the style or texture of an image.

The **gram matrix** computation is the only extra step here compared to the content loss calculations. Also, as we can see, the output of the pixel-wise squared summed loss is quite a large number compared to the content loss. Hence, this number is normalized by dividing it by *N***X***Y*; that is, the number of feature maps (*N*) times the length (*X*) times the breadth (*Y*) of a feature map. This also helps standardize the **style loss** metric across different convolutional layers, which have a different *N*, *X*, and *Y*. Details of the implementation can be found in the original paper that introduced neural style transfer [2].

Now that we understand the concept of content and style loss, let's take a look at how neural style transfer works, as follows:

1. For the given VGG (or any other CNN) network, we define which convolutional layers in the network should have a content loss attached to them. Repeat this exercise for style loss.
2. Once we have those lists, we pass the content image through the network and compute the expected convolutional outputs (2D matrices) at the convolutional layers where the content loss is to be calculated.
3. Next, we pass the style image through the network and compute the expected gram matrices at the convolutional layers. This is where the style loss is to be calculated, as demonstrated in *Figure 8.4*.

In the following diagram, for example, the content loss is to be calculated at the second and third convolutional layers, while the style loss is to be calculated at the second, third, and fifth convolutional layers:

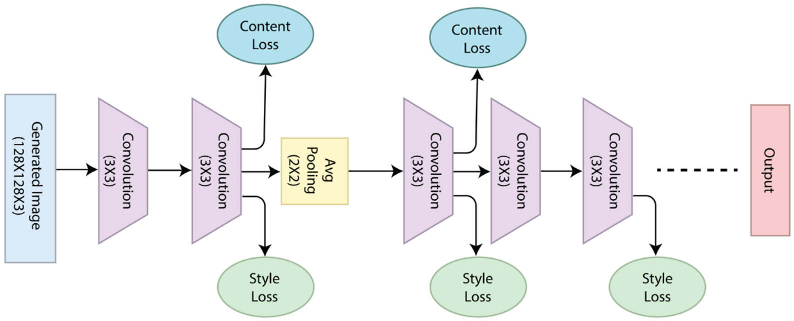


Figure 8.4: Style transfer architecture schematic

Now that we have the content and style targets at the decided convolutional layers, we are all set to generate an image that contains the content of the content image and the style of the style image.

For initialization, we can either use a random noise matrix as our starting point for the generated image or directly use the content image to start with. We pass this image through the network and compute the style and content losses at the pre-selected convolutional layers. We add style losses to get the total style loss, and content losses to get the total content loss. Finally, we obtain the total loss by summing these two components in a weighted fashion.

If we give more weight to the style component, the generated image will have more style reflected on it, and vice versa. Using gradient descent, we backpropagate the loss all the way back to the input in order to update our generated image. After a few epochs, the generated image should evolve in such a way that it produces the content and style representations that minimize the respective losses, thereby producing a style-transferred image.

In *Figure 8.4*, the pooling layer is average pooling-based instead of the traditional max pooling. Average pooling is deliberately used for style transfer to ensure smooth gradient flow. We want the generated images not to have sharp changes between pixels. Also, it is worth noticing that the network in the preceding diagram ends at the layer where the last style or content loss is calculated. Hence, in this case, because there is no loss associated with the sixth convolutional layer of the original network, it is meaningless to talk about layers beyond the fifth convolutional layer in the context of style transfer.

In the next section, we will implement our own neural style transfer system using PyTorch. With the help of a pretrained VGG model, we will use the concepts we've discussed in this section to generate artistically styled images. We will also explore the impact of tuning the various model parameters on the content and texture/style of generated images.

Implementing neural style transfer using PyTorch

Having discussed the internals of a neural style transfer system, we are all set to build one using PyTorch. In the form of an exercise, we will load a style and a content image. Then, we will load the pretrained VGG model. After defining which layers to compute the style and content loss on, we will trim the model so that it only retains the relevant layers. Finally, we will train the neural style transfer model to refine the generated image epoch by epoch.

Loading the content and style images

In this exercise, we will only show the important parts of the code for demonstration purposes. To access the full code, go to our GitHub repository [3]. Follow these steps:

1. First, we need to import the necessary libraries:

```
from PIL import Image
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
dvc = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Among other libraries, we import the `torchvision` library to load the pretrained VGG model and other computer vision-related utilities.

2. Next, we need a style and a content image. We will use the Unsplash website [4] to download an image of each kind. The downloaded images are included in the code repository of this book. In the following code, we are writing a function that will load the images as tensors:

```
def image_to_tensor(image_filepath,  image_dimension=128):
    img = Image.open(image_filepath).convert('RGB')
    # display image
    ...
    torch_transformation = torchvision.transforms.Compose([
        torchvision.transforms.Resize(img_size),
        torchvision.transforms.ToTensor()])
    img = torch_transformation(img).unsqueeze(0)
    return img.to(dvc, torch.float)
style_image = image_to_tensor("./images/style.jpg")
content_image =image_to_tensor("./images/content.jpg")
```

The `unsqueeze` command adds a dimension in the 0th axis of a tensor converting, say, a (32,32) sized tensor into a (1, 32, 32) sized tensor. Feel free to change the parameter from 0 to 1 and see what happens.

The preceding code should give us the following output:

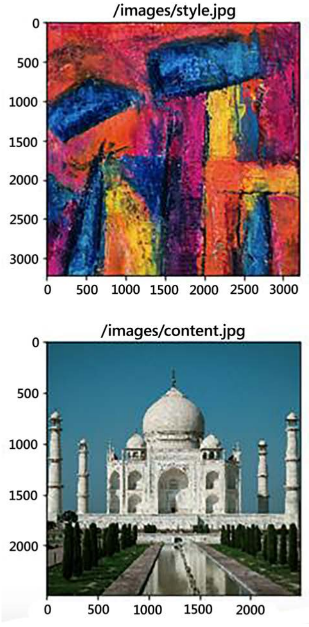


Figure 8.5: Style and content images

So, the content image is a real-life photograph of the Taj Mahal, whereas the style image is an art painting. Using style transfer, we hope to generate an artistic Taj Mahal painting. However, before we do that, we need to load and trim the VGG19 model.

Loading and trimming the pretrained VGG19 model

In this part of the exercise, we will use a pretrained VGG model and retain its convolutional layers. We will make some minor changes to the model to make it usable for neural style transfer. Let's get started:

1. We will first load the pretrained VGG19 model and use its convolutional layers to generate the content and style targets to yield the content and style losses, respectively:

```
vgg19_model = torchvision.models.vgg19(pretrained=True).to(dvc)
print(vgg19_model)
```

The output should be as follows:

```
VGG(
(features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    ...
    (35): ReLU(inplace=True)
    (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    ...
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

2. We do not need the linear layers; that is, we only need the convolutional part of the model. In the preceding code, this can be achieved by only retaining the `features` attribute of the `model` object, as follows:

```
vgg19_model = vgg19_model.features
```

> In this exercise, we are not going to tune the parameters of the VGG model. All we are going to tune is the pixels of the generated image, right at the input end of the model. Hence, we will ensure that the parameters of the loaded VGG model are fixed.

3. We must freeze the parameters of the VGG model with the following code:

```
for param in vgg19_model.parameters():
    param.requires_grad_(False)
```

4. Now that we've loaded the relevant section of the VGG model, we need to change the `maxpool` layers into average pooling layers, as discussed in the previous section. While doing so, we will take note of where the convolutional layers are located in the model:

```
conv_indices = []
for i in range(len(vgg19_model)):
    if vgg19_model[i]._get_name() == 'MaxPool2d':
        vgg19_model[i] = \
            nn.AvgPool2d(
                kernel_size=vgg19_model[i].kernel_size,
                stride=vgg19_model[i].stride,
                padding=vgg19_model[i].padding)
    if vgg19_model[i]._get_name() == 'Conv2d':
        conv_indices.append(i)
conv_indices = dict(enumerate(conv_indices, 1))
print(vgg19_model)
```

The output should be as follows:

```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace=True)
  (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
...
  (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (35): ReLU(inplace=True)
  (36): AvgPool2d(kernel_size=2, stride=2, padding=0)
)
```

As we can see, the linear layers have been removed and the max pooling layers have been replaced by average pooling layers, as indicated by the highlighted code.

In the preceding steps, we loaded a pretrained VGG model and modified it in order to use it as a neural style transfer model. Next, we will transform this modified VGG model into a neural style transfer model.

Building the neural style transfer model

At this point, we can define which convolutional layers we want the content and style losses to be calculated on. In the original paper, style loss was calculated on the first five convolutional layers, while content loss was calculated on the fourth convolutional layer only.

We will follow the same convention, although you are welcome to try out different combinations and observe their effects on the generated image. Follow these steps:

1. First, we list the layers we need to have the style and content loss on:

   ```
   layers = {1: 's', 2: 's', 3: 's', 4: 'sc', 5: 's'}
   ```

   Here, we have defined the first to fifth convolutional layers, which are attached to the style loss, and the fourth convolutional layer, which is attached to the content loss. Note that s and c refer to style and content losses, respectively.

2. Now, let's remove the unnecessary parts of the VGG model. We shall only retain it until the fifth convolutional layer, as shown here:

   ```
   vgg_layers = nn.ModuleList(vgg19_model)
   last_layer_idx = conv_indices[max(layers.keys())]
   vgg_layers_trimmed = vgg_layers[:last_layer_idx+1]
   neural_style_transfer_model = nn.Sequential(*vgg_layers_trimmed)
   print(neural_style_transfer_model)
   ```

   This should give us the following output:

   ```
   Sequential(
     (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
     (1): ReLU(inplace=True)
     (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
     ...
     (8): ReLU(inplace=True)
     (9): AvgPool2d(kernel_size=2, stride=2, padding=0)
     (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
   1))
   )
   ```

   As we can see, we have transformed the VGG model with 16 convolutional layers into a neural style transfer model with five convolutional layers.

Training the style transfer model

In this section, we'll start working on the image that will be generated. We can initialize this image in many ways, such as by using a random noise image or using the content image as the initial image. Currently, we are going to start with random noise. Later, we will also see how using the content image as the starting point impacts the results. Follow these steps:

1. The following code demonstrates the process of initializing a torch tensor with random numbers:

   ```
   '''initialize as the content image
      ip_image = content_image.clone()
   ```

```
   initialize as random noise:'''
ip_image = torch.randn(content_image.data.size(), device=dvc)
plt.figure()
plt.imshow(ip_image.squeeze(0).cpu().detach().numpy().transpose(1,2,0).
clip(0,1));
```

This should give us the following output:

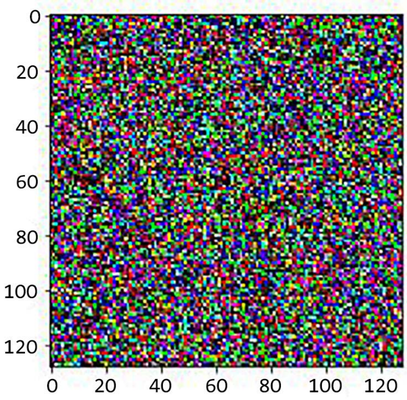


Figure 8.6: Random noise image

2. Finally, we can start the model training loop. First, we will define the number of epochs to train for and the relative weightage to provide for the style and content losses, and instantiate the Adam optimizer for gradient descent-based optimization with a learning rate of `0.1`:

```
num_epochs=180
wt_style=1e6
wt_content=1
style_losses = []
content_losses = []
opt = optim.Adam([ip_image.requires_grad_()], lr=0.1)
```

3. Upon starting the training loop, we initialize the style and content losses to zero at the beginning of the epoch, and then clip the pixel values of the input image between `0` and `1` for numerical stability:

```
for curr_epoch in range(1, num_epochs+1):
    ip_image.data.clamp_(0, 1)
    opt.zero_grad()
    epoch_style_loss = 0
    epoch_content_loss = 0
```

4. At this stage, we have reached a crucial step in the training iteration. Here, we must calculate the style and content losses for each of the pre-defined style and content convolutional layers.

The individual style losses and content losses for each of the respective layers are added together to get the total style and content loss for the current epoch:

```
    for k in layers.keys():
        if 'c' in layers[k]:
            target = \
                neural_style_transfer_model[
                    :conv_indices[k]+1]
                    (content_image).detach()
            ip = \
                neural_style_transfer_model[
                    :conv_indices[k]+1](ip_image)
            epoch_content_loss += \
                torch.nn.functional.mse_loss(ip, target)
        if 's' in layers[k]:
            target = gram_matrix(
            neural_style_transfer_model[
                :conv_indices[k]+1]
                (style_image)).detach()
            ip = \
                gram_matrix(
                    neural_style_transfer_model[
                        :conv_indices[k]+1](ip_image))
            epoch_style_loss += \
                torch.nn.functional.mse_loss(ip, target)
```

As shown in the preceding code, for both the style and content losses, first, we compute the style and content targets (ground truths) using the style and content image. We use `.detach()` for the targets to indicate that these are not trainable but just fixed target values. Next, we compute the predicted style and content outputs based on the generated image as input, at each of the style and content layers. Finally, we compute the style and content losses.

5. For the style loss, we also need to compute the gram matrix using a pre-defined gram matrix function, as shown in the following code:

```
def gram_matrix(ip):
    num_batch, num_channels, height, width = ip.size()
    feats = ip.view(num_batch * num_channels, width * height)
    gram_mat = torch.mm(feats, feats.t())
    return gram_mat.div(
        num_batch * num_channels * width * height)
```

As we mentioned earlier, we can compute an inner dot product using the `torch.mm` function. This computes the gram matrix and normalizes the matrix by dividing it by the number of feature maps times the width times the height of each feature map.

6. Moving on in our training loop, now that we've computed the total style and content losses, we need to compute the final total loss as a weighted sum of these two, using the weights we defined earlier:

```
        epoch_style_loss *= wt_style
        epoch_content_loss *= wt_content
        total_loss = epoch_style_loss + epoch_content_loss
        total_loss.backward()
```

   Finally, at every *k* epochs, we can see the progression of our training by looking at the losses as well as looking at the generated image. The following figure shows the evolution of the generated style transferred image for the previous code for a total of 180 epochs recorded at every 20 epochs:

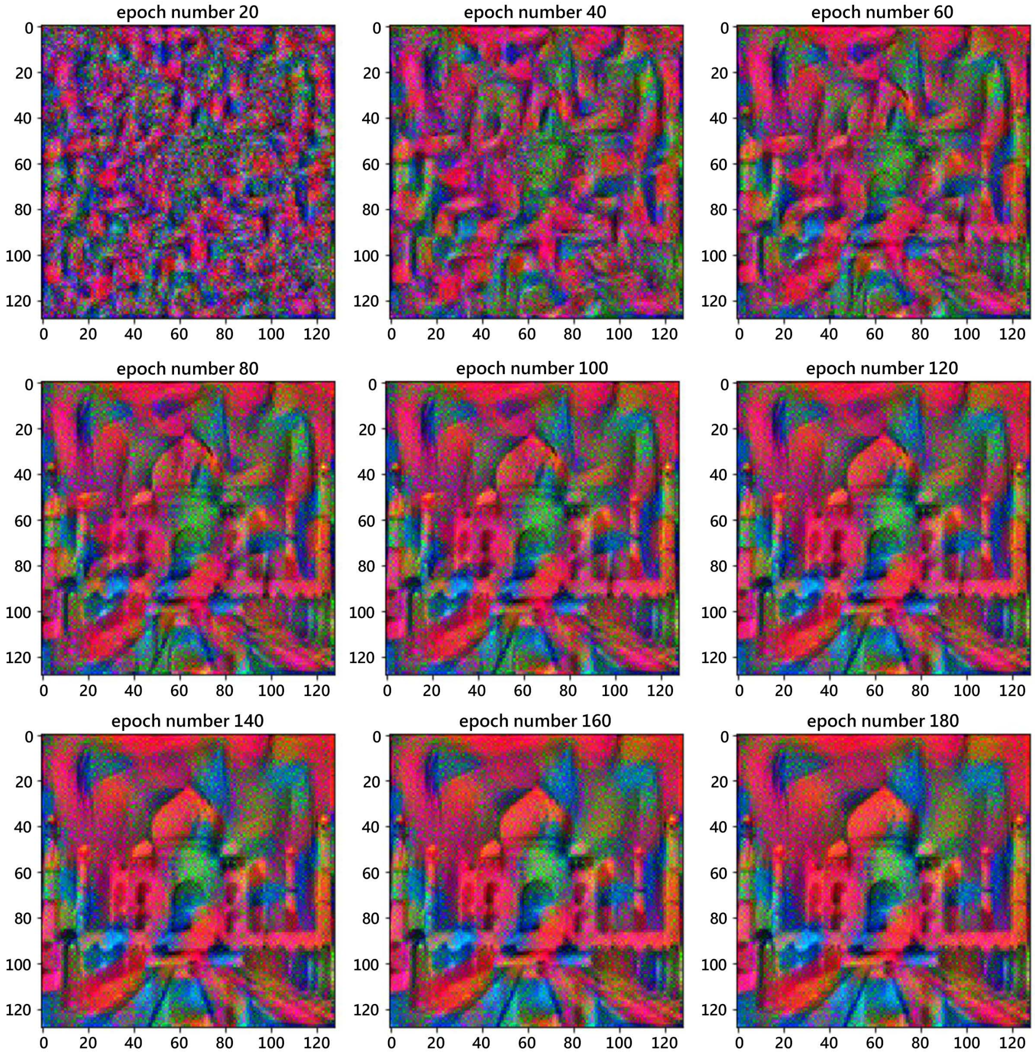


Figure 8.7: Neural style transfer epoch-wise generated image

It is quite clear that the model begins by applying the style from the style image to the random noise. As training proceeds, the content loss starts playing its role, thereby imparting content to the styled image. By epoch **180**, we can see the generated image, which looks like a good approximation of an artistic painting of the Taj Mahal. The following graph shows the decreasing style and content losses as the epochs progress from **0** to **180**:

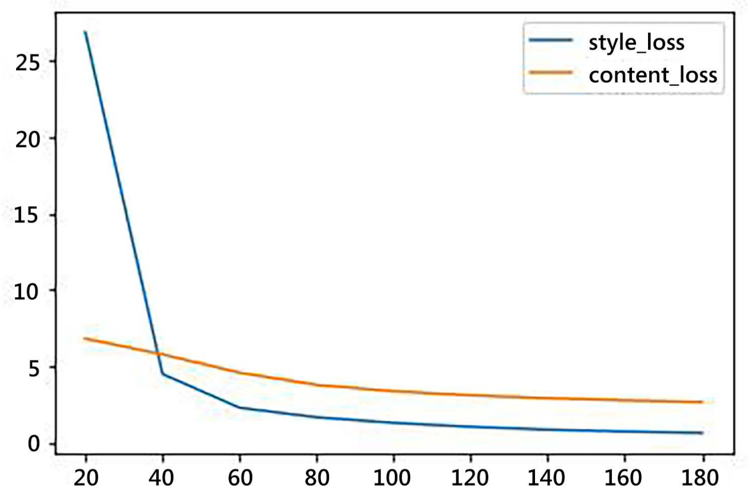


Figure 8.8: Style and content loss curves

Noticeably, the style loss sharply goes down initially, which is also evident in *Figure 8.7* in that the initial epochs mark the imposition of style on the image more than the content. At the advanced stages of training, both losses decline together gradually, resulting in a style-transferred image, which is a decent compromise between the artwork of the style image and the realism of a photograph that's been taken with a camera.

Experimenting with the style transfer system

Having successfully trained a style transfer system in the previous section, we will now look at how the system responds to different hyperparameter settings. Follow these steps:

1. In the preceding section, we set the content weight to `1` and the style weight to `1e6`. Let's increase the style weight 10x further – that is, to `1e7` – and observe how it affects the style transfer process. Upon training with the new weights for 600 epochs, we get the following progression of style transfer:

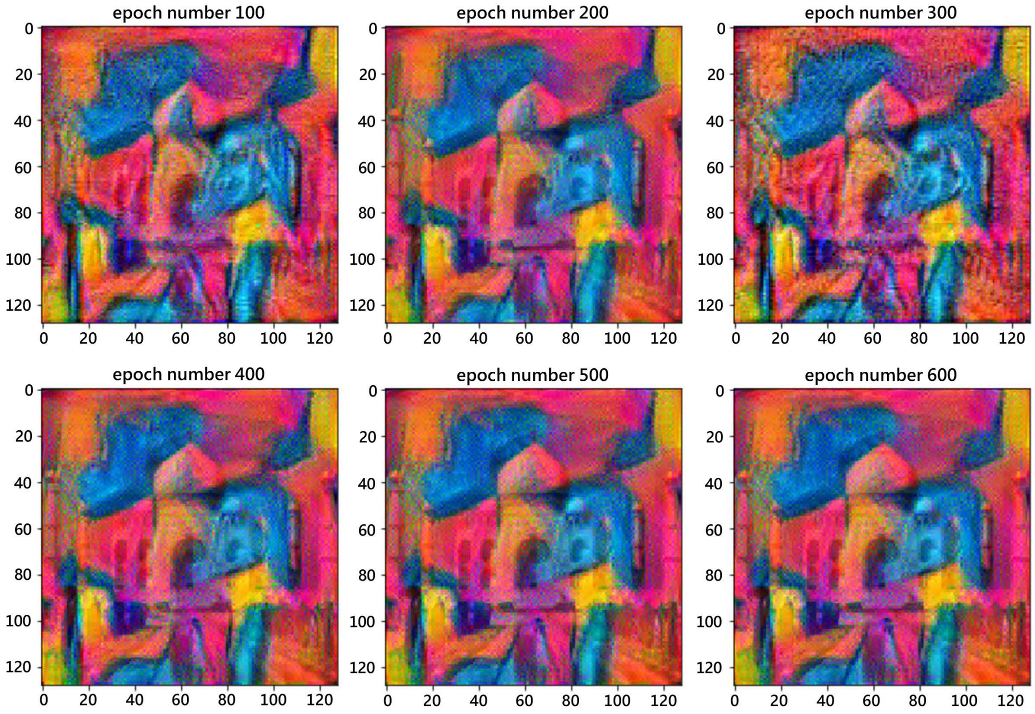


Figure 8.9: Style transfer epochs with higher style weights

Here, we can see that initially, it required many more epochs than in the previous scenario to reach a reasonable result. More importantly, the higher style weight does seem to have an effect on the generated image. When we look at the images in the preceding figure compared to the ones in *Figure 8.7*, we find that the former has a stronger resemblance to the style image shown in *Figure 8.5*.

2. Likewise, reducing the style weight from `1e6` to `1e5` produces a more content-focused result, as can be seen in *Figure 8.10*:

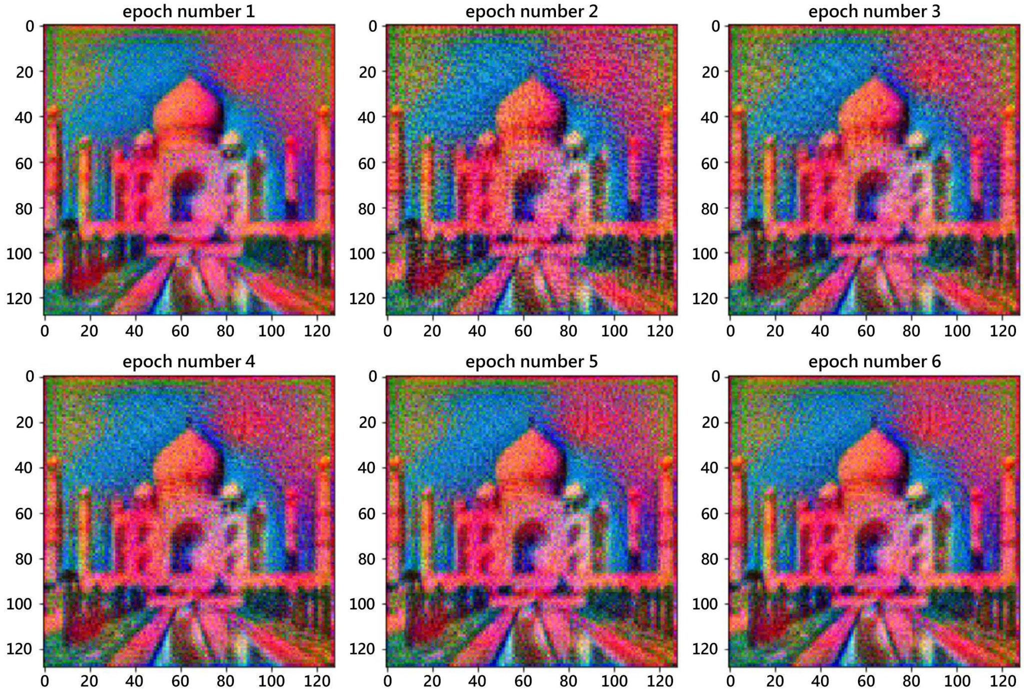


Figure 8.10: Style transfer epochs with lower style weights

Compared to the scenario with a higher style weight, having a lower style weight means it takes far fewer epochs to get a reasonable-looking result. The amount of style in the generated image is much smaller and is mostly filled with the content image data. We only trained this scenario for 6 epochs, as the results saturate after that point.

3. A final change could be to initialize the generated image with the content image instead of the random noise, while using the original style and content weights of `1e6` and `1`, respectively. *Figure 8.11* shows the epoch-wise progression in this scenario:

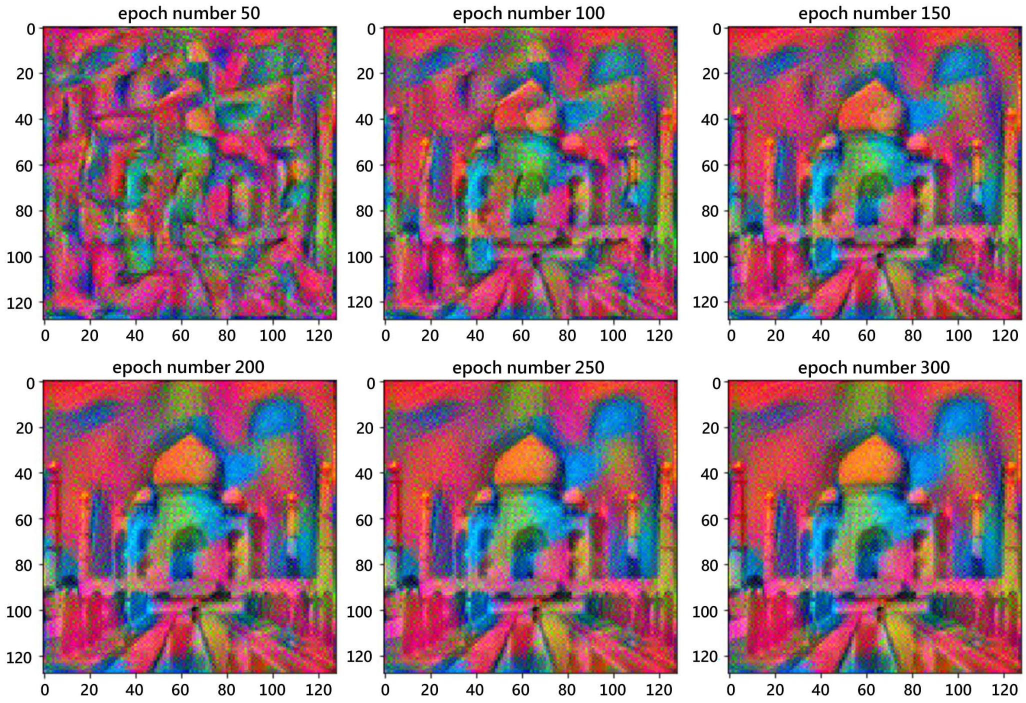


Figure 8.11: Style transfer epochs with content image initialization

By comparing the preceding figure to *Figure 8.7*, we can see that having the content image as a starting point gives us a different path of progression to getting a reasonable style transferred image. It seems that both the content and style components are being imposed on the generated image more simultaneously than in *Figure 8.7*, where the style was imposed first, followed by the content. The following graph confirms this hypothesis:

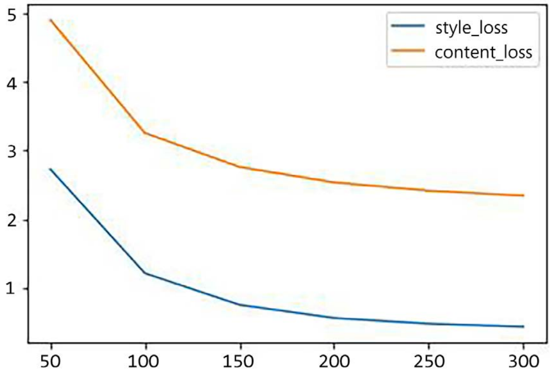


Figure 8.12: Style and content loss curves with content image initialization

As we can see, both style and content losses are decreasing together as the epochs progress, eventually saturating toward the end. Nonetheless, the end results in both *Figures 8.17* and *8.11*, and even *Figures 8.9* and *8.10*, all represent reasonable artistic impressions of the Taj Mahal. The two loss lines seem further apart in *Figure 8.12* compared to *Figure 8.8* only because the *y*-axis ranges are different in the two curves.

We have successfully built a neural style transfer model using PyTorch, wherein using a content image (a photograph of the beautiful Taj Mahal) and a style image (a canvas painting), we generated a reasonable approximation of an artistic painting of the Taj Mahal. This application can be extended to various other combinations. Swapping the content and style images could also produce interesting results and give more insight into the inner workings of the model.

You are encouraged to extend the exercise we discussed in this chapter by doing the following:

- Changing the list of style and content layers
- Using larger image sizes
- Trying more combinations of style and content loss weights
- Using other optimizers, such as SGD and LBFGS
- Training for longer epochs with different learning rates to observe the differences in the generated images across all these approaches

Summary

In this chapter, we applied the concept of generative machine learning to images by generating an image that contains the content of one image and the style of another – a task known as neural style transfer.

In the next chapter, we will expand on this paradigm, where we'll have a generator that generates *fake* data and a discriminator that distinguishes *fake* data from *real* data. Such models are popularly known as **generative adversarial networks** (**GANs**). We will be exploring **deep convolutional GANs** (**DCGANs**) in the next chapter.

References

1. Gram matrix: `https://mathworld.wolfram.com/GramMatrix.html`
2. Leon A. Gatys, Alexander S. Ecker, Matthias Bethge. 2015. *A Neural Algorithm of Artistic Style*: `https://arxiv.org/pdf/1508.06576.pdf`
3. Mastering PyTorch GitHub repository link: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter08/neural_style_transfer.ipynb`
4. *Unsplash: Beautiful Free Images & Pictures*: `https://unsplash.com/`

Learn more on Discord

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

`https://packt.link/mastorch`

9

Deep Convolutional GANs

Generative neural networks have become a popular and active area of research and development. A huge amount of credit for this trend goes to a class of models that we are going to discuss in this chapter. These models are called **generative adversarial networks** (**GANs**) and were introduced in 2014. Ever since the introduction of the basic GAN model, various types of GANs have been, and are being, invented for different applications.

Generative models are not limited though to GANs. **Variational Autoencoders** (**VAEs**) (the secret sauce of OpenAI's DALL-E), which can learn the underlying distribution of data and can generate new samples by sampling from that distribution, and auto-regressive models (the secret sauce of LLMs), which generate data one element at a time, conditioned on the previous elements, are also among a long list of well-known generative models. However, GANs leverage their ability to generate highly realistic and diverse samples that resemble the training data without requiring explicit modeling of the data distribution.

Essentially, a GAN is composed of two neural networks – a **generator** and a **discriminator**. Let's look at an example of the GAN that is used to generate images. For such a GAN, the task of the generator would be to generate realistic-looking fake images, and the task of the discriminator would be to tell the real images apart from the fake images.

In a joint optimization procedure, the generator would ultimately learn to generate such good fake images that the discriminator will essentially be unable to tell them apart from real images. Once such a model is trained, the generator part of it can then be used as a reliable data generator. Besides being used as a generative model for unsupervised learning, GANs have also proven useful in semi-supervised learning.

In the image example, for instance, the features learned by the discriminator model could be used to improve the performance of classification models trained on the image data. Besides semi-supervised learning, GANs have also proven to be useful in reinforcement learning, which is a topic that we will discuss in *Chapter 11*, *Deep Reinforcement Learning*.

A particular type of GAN that we will focus on in this chapter is the **deep convolutional GAN** (**DCGAN**). A DCGAN is deep and hence differs from shallow GANs (simpler architecture with fewer layers) in having a more complex architecture with multiple layers, which can generate higher-quality images with more intricate details. A DCGAN is essentially an unsupervised **convolution neural network** (**CNN**) model. Both the generator and the discriminator in a DCGAN are purely *CNNs with no fully connected layers*. DCGANs have performed well in generating realistic images, and they can be a good starting point for learning how to build, train, and run GANs from scratch. In this chapter, we will first understand the various components within a GAN – the generator and the discriminator models and the joint optimization schedule. We will then focus on building a DCGAN model using PyTorch. Next, we will use an image dataset to train and test the performance of the DCGAN model. We will conclude this chapter by revisiting the concept of style transfer on images and exploring the pix2pix GAN model, which can efficiently perform a style transfer on any given pair of images.

We will also learn how the various components of a pix2pix GAN model relate to that of a DCGAN model. After finishing this chapter, we will truly understand how GANs work and will be able to build any type of GAN model using PyTorch. This chapter is broken down into the following topics:

- Defining the generator and discriminator networks
- Training a DCGAN using PyTorch
- Using GANs for style transfer

> All the code files for this chapter can be found at `https://github.com/arj7192/MasteringPyTorchV2/tree/main/Chapter09`.

Defining the generator and discriminator networks

As mentioned earlier, GANs are composed of two components – the generator and the discriminator. Both of these are essentially neural networks. Generators and discriminators with different neural architectures produce different types of GANs. You can find a list of different types of GANs along with their PyTorch implementations in this reference list [1].

For any GAN that is used to generate some kind of real data, the generator usually takes random noise as input and produces an output with the same dimensions as the real data. We call this generated output **fake data**. The discriminator, on the other hand, works as a **binary classifier**. It takes in the generated fake data and the real data (one at a time) as input and predicts whether the input data is real or fake. *Figure 9.1* shows a diagram of the overall GAN model schematic:

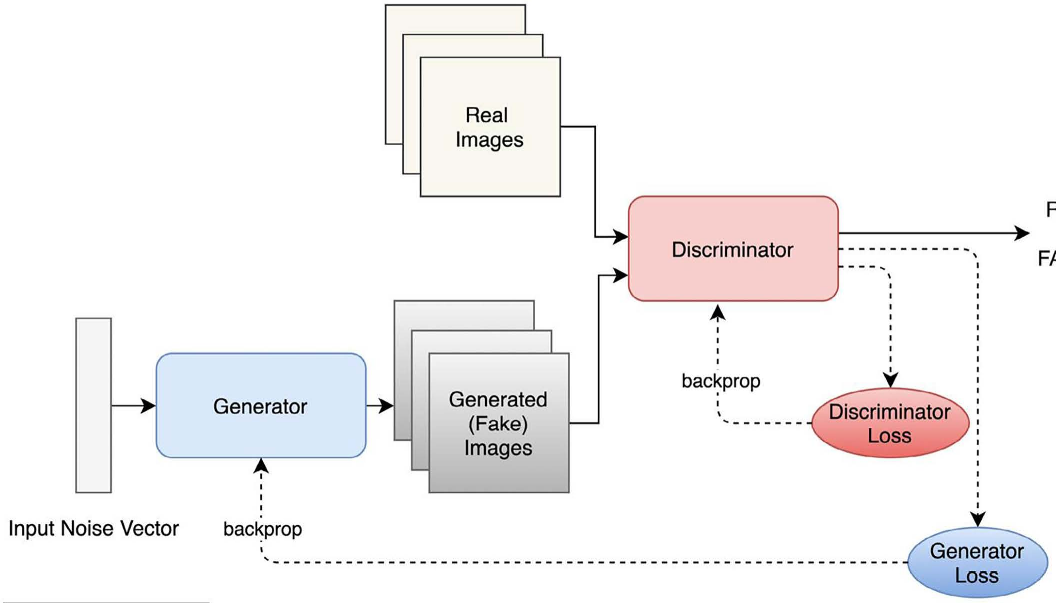


Figure 9.1: A GAN schematic

The discriminator network is optimized like any binary classifier, that is, using the binary cross-entropy function. Therefore, the discriminator model's motivation is to correctly classify real images as real and fake images as fake. The generator network has quite the opposite motivation. The generator loss is mathematically expressed as $-log(D(G(x)))$, where x is random noise inputted into the generator model, G; $G(x)$ is the generated fake image by the generator model; and $D(G(x))$ is the output probability of the discriminator model, D – that is, the probability of the image being real.

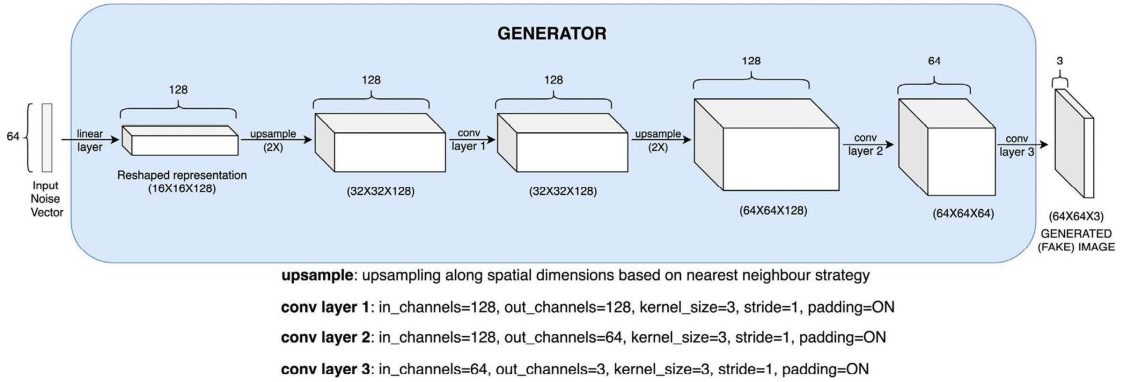
Therefore, the generator loss is minimized when the discriminator thinks that the generated fake image is real. Essentially, the generator is trying to fool the discriminator in this joint optimization problem.

In execution, these two loss functions are backpropagated alternatively. That is, at every iteration of training, first, the discriminator is frozen, and the parameters of the generator networks are optimized by backpropagating the gradients from the generator loss.

Then, the tuned generator is frozen while the discriminator is optimized by backpropagating the gradients from the discriminator loss. This is what we call joint optimization. It has also been referred to as being equivalent to a two-player Minimax game in the original GAN paper [2].

Understanding the DCGAN generator and discriminator

For the particular case of DCGANs, let's consider what the generator and discriminator model architectures look like. As already mentioned, both are purely convolutional models. *Figure 9.2* shows the generator model architecture for a DCGAN. All layer dimensions are specific to our implementation of DCGAN.



upsample: upsampling along spatial dimensions based on nearest neighbour strategy

conv layer 1: in_channels=128, out_channels=128, kernel_size=3, stride=1, padding=ON

conv layer 2: in_channels=128, out_channels=64, kernel_size=3, stride=1, padding=ON

conv layer 3: in_channels=64, out_channels=3, kernel_size=3, stride=1, padding=ON

Figure 9.2: The DCGAN generator model architecture

First, the random noise input vector of size **64** is reshaped and projected into **128** feature maps of size **16x16** each. This projection is achieved using a linear layer. From there on, a series of upsampling and convolutional layers follow.

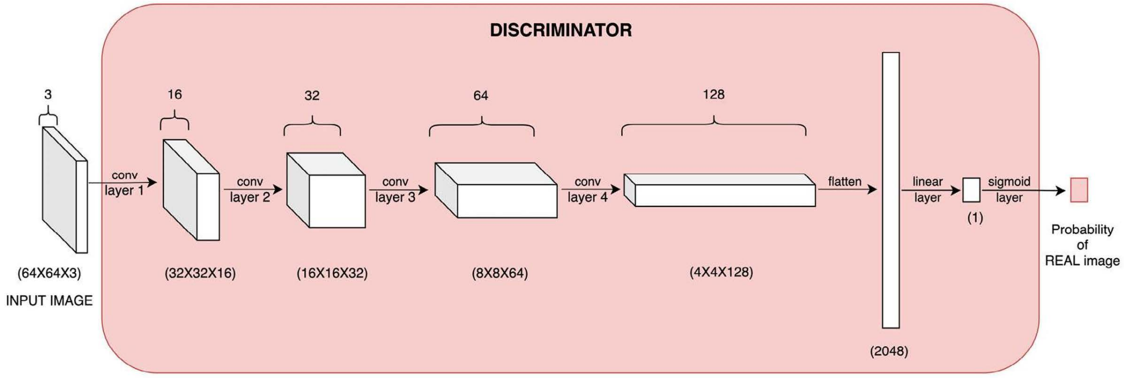
Upsampling in CNNs refers to the process of increasing the spatial resolution of feature maps by inserting additional rows and columns of zeros or by using interpolation methods, such as bilinear or nearest-neighbor interpolation. This is commonly used in tasks such as image segmentation, where the final output needs to have the same spatial dimensions as the input image.

The first upsampling layer in our model simply transforms the **16x16** feature maps into **32x32** feature maps using the nearest neighbor upsampling strategy. This is followed by a 2D convolutional layer with a **3x3** kernel size and **128** output feature maps. The **128 32x32** feature maps outputted by this convolutional layer are further upsampled to **64x64**-sized feature maps, which are followed by two **2D** convolutional layers, resulting in the generated (fake) RGB image of size **64x64**.

Note

We have omitted the batch normalization and leaky ReLU layers to avoid clutter in the preceding architectural representation. The PyTorch code in the next section will have these details mentioned and explained.

Now that we know what the generator model looks like, let's examine what the discriminator model looks like. *Figure 9.3* shows the discriminator model architecture:



conv layer 1: in_channels=3, out_channels=16, kernel_size=3, stride=2, padding=ON

conv layer 2: in_channels=16, out_channels=32, kernel_size=3, stride=2, padding=ON

conv layer 3: in_channels=32, out_channels=64, kernel_size=3, stride=2, padding=ON

conv layer 4: in_channels=64, out_channels=128, kernel_size=3, stride=2, padding=ON

Figure 9.3: The DCGAN discriminator model architecture

As you can see, a stride of 2 at every convolutional layer in this architecture helps to reduce the spatial dimension, while the depth (that is, the number of feature maps) keeps growing. This is a classic CNN-based binary classification architecture being used here to classify between real images and generated fake images.

Having understood the architectures of the generator and the discriminator network, we can now build the entire DCGAN model based on the schematic in *Figure 9.1* and train the DCGAN model on an image dataset.

In the next section, we will use PyTorch for this task. We will discuss, in detail, the DCGAN model instantiation, loading the image dataset, jointly training the DCGAN generator and discriminator, and generating sample fake images from the trained DCGAN generator.

Training a DCGAN using PyTorch

In this section, we will build, train, and test a DCGAN model using PyTorch in the form of an exercise. We will use an image dataset to train the model and test how well the generator of the trained DCGAN model performs when producing fake images.

Defining the generator

In the following exercise, we will only show the important parts of the code for demonstration purposes. In order to access the full code, you can refer to our GitHub repository [3]:

1. First, we need to import the required libraries, as follows:

    ```
    import os
    import numpy as np
    import torch
    import torch.nn as nn
    import torch.nn.functional as F
    from torch.utils.data import DataLoader
    from torch.autograd import Variable
    import torchvision.transforms as transforms
    from torchvision.utils import save_image
    from torchvision import datasets
    ```

    In this exercise, we only need `torch` and `torchvision` to build the DCGAN model.

2. After importing the libraries, we specify some model hyperparameters, as shown in the following code:

    ```
    num_eps=10
    bsize=32
    lrate=0.001
    lat_dimension=64
    image_sz=64
    chnls=1
    logging_intv=200
    ```

    We will be training the model for `10` epochs with a batch size of `32` and a learning rate of `0.001`. The expected image size is 64x64x3. `lat_dimension` is the length of the random noise vector, which essentially means that we will draw the random noise from a 64-dimensional latent space as input to the generator model.

3. Now we define the generator model object. The following code is in direct accordance with the architecture shown in *Figure 9.2*:

    ```
    class GANGenerator(nn.Module):
        def __init__(self):
            super(GANGenerator, self).__init__()
            self.inp_sz = image_sz // 4
            self.lin = nn.Sequential(nn.Linear(
                lat_dimension, 128 * self.inp_sz ** 2))
            self.bn1 = nn.BatchNorm2d(128)
    ```

```
        self.up1 = nn.Upsample(scale_factor=2)
        self.cn1 = nn.Conv2d(128, 128, 3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(128, 0.8)
        self.rl1 = nn.LeakyReLU(0.2, inplace=True)
        self.up2 = nn.Upsample(scale_factor=2)
        self.cn2 = nn.Conv2d(128, 64, 3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(64, 0.8)
        self.rl2 = nn.LeakyReLU(0.2, inplace=True)
        self.cn3 = nn.Conv2d(64, chnls, 3, stride=1, padding=1)
        self.act = nn.Tanh()
```

4. After defining the `_init_` method, we define the `forward` method, which is essentially just calling the layers in a sequential manner:

```
    def forward(self, x):
        x = self.lin(x)
        x = x.view(x.shape[0], 128, self.inp_sz, self.inp_sz)
        x = self.bn1(x)
        x = self.up1(x)
        x = self.cn1(x)
        x = self.bn2(x)
        x = self.rl1(x)
        x = self.up2(x)
        x = self.cn2(x)
        x = self.bn3(x)
        x = self.rl2(x)
        x = self.cn3(x)
        out = self.act(x)
        return out
```

We have used the explicit layer-by-layer definition in this exercise as opposed to the `nn.Sequential` method; this is because it makes it easier to debug the model if something goes wrong. We can also see the batch normalization and leaky ReLU layers in the code, which are not mentioned in *Figure 9.2*.

FAQ – Why are we using batch normalization?

Batch normalization is used after the linear or convolutional layers to both fasten the training process and reduce sensitivity to the initial network weights.

FAQ – Why are we using leaky ReLU?

ReLU might lose all the information for inputs with negative values. A leaky ReLU set with a 0.2 negative slope gives 20% weightage to incoming negative information, which might help us to prevent vanishing gradients during the training of a GAN model.

Next, we will take a look at the PyTorch code to define the discriminator network.

Defining the discriminator

Similar to the generator, we will now define the discriminator model as follows:

1. Once again, the following code is the PyTorch equivalent for the model architecture shown in *Figure 9.3*:

```
class GANDiscriminator(nn.Module):
    def __init__(self):
        super(GANDiscriminator, self).__init__()
        def disc_module(ip_chnls, op_chnls, bnorm=True):
            mod = [nn.Conv2d(ip_chnls, op_chnls, 3, 2, 1),
                   nn.LeakyReLU(0.2, inplace=True),
                   nn.Dropout2d(0.25)]
            if bnorm:
                mod += [nn.BatchNorm2d(op_chnls, 0.8)]
            return mod
        self.disc_model = nn.Sequential(
            *disc_module(chnls, 16, bnorm=False),
            *disc_module(16, 32),
            *disc_module(32, 64),
            *disc_module(64, 128),
        )
        # width and height of the down-sized image
        ds_size = image_sz // 2 ** 4
        self.adverse_lyr = nn.Sequential(nn.Linear(
            128 * ds_size ** 2, 1), nn.Sigmoid())
```

First, we have defined a general discriminator module, which is a cascade of a convolutional layer, an optional batch normalization layer, a leaky ReLU layer, and a dropout layer. To build the discriminator model, we repeat this module sequentially four times – each time with a different set of parameters for the convolutional layer.

The goal is to input a 64x64x3 RGB image and to increase the depth (that is, the number of channels) and decrease the height and width of the image as it is passed through the convolutional layers.

The final discriminator module's output is flattened and passed through the adversarial layer. Essentially, the adversarial layer fully connects the flattened representation to the final model output (that is, a binary output). This model output is then passed through a sigmoid activation function to give us the probability of the image being real (or not fake).

2. The following is the `forward` method for the discriminator, which takes in a 64x64 RGB image as input and produces the probability of it being a real image:

```
    def forward(self, x):
        x = self.disc_model(x)
        x = x.view(x.shape[0], -1)
        out = self.adverse_lyr(x)
        return out
```

3. Having defined the generator and discriminator models, we can now instantiate one of each. We also define our adversarial loss function as the binary cross-entropy loss function in the following code:

```
# instantiate the discriminator and generator models
gen = GANGenerator()
disc = GANDiscriminator()
# define the loss metric
adv_loss_func = torch.nn.BCELoss()
```

The adversarial loss function will be used to define the generator and discriminator loss functions later in the training loop. Conceptually, we are using binary cross-entropy as the loss function because the targets are essentially binary – that is, either real images or fake images. And binary cross-entropy loss is a well-suited loss function for binary classification tasks.

Loading the image dataset

For the task of training a DCGAN to generate realistic-looking fake images, we are going to use the well-known MNIST dataset, which contains images of handwritten digits from 0 to 9. By using `torchvision.datasets`, we can directly download the MNIST dataset and create a `dataset` and a `dataloader` instance out of it:

```
# define the dataset and corresponding dataloader
dloader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "./data/mnist/", download=True,
        transform=transforms.Compose(
            [transforms.Resize((image_sz, image_sz)),
             transforms.ToTensor(),
             transforms.Normalize([0.5], [0.5])]),),
             batch_size=bsize, shuffle=True,)
```

Here is an example of a real image from the MNIST dataset:

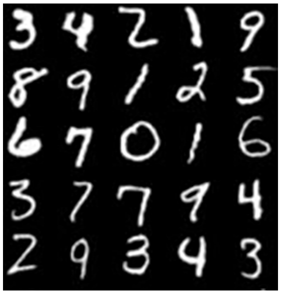


Figure 9.4: 25 randomly sampled images from the MNIST dataset

Dataset citation

[LeCun et al., 1998a] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. *Gradient-based learning applied to document recognition*. Proceedings of the IEEE, 86(11):2278-2324, November 1999.

Yann LeCun (Courant Institute, NYU) and Corinna Cortes (Google Labs, New York) hold the copyright of the MNIST dataset, which is a derivative work from the original NIST datasets. The MNIST dataset is made available under the terms of the Creative Commons Attribution-Share Alike 3.0 license [4].

So far, we have defined the model architecture and the data pipeline. Now it is time for us to actually write the DCGAN model training routine, which we will do in the following section.

Training loops for DCGANs

In this section, we will train the DCGAN model:

1. **Defining the optimization schedule**: Before starting the training loop, we will define the optimization schedule for both the generator and the discriminator. We will use the `Adam` optimizer for our model. In the original DCGAN paper [5], the *beta1* and *beta2* parameters of the Adam optimizer are set to *0.5* and *0.999*, as opposed to the usual *0.9* and *0.999*.

   We have retained the default values of *0.9* and *0.999* in our exercise. However, you are welcome to use the exact same values mentioned in the paper for similar results:

   ```
   # define the optimization schedule for both G and D
   opt_gen = torch.optim.Adam(gen.parameters(), lr=lrate)
   opt_disc = torch.optim.Adam(disc.parameters(), lr=lrate)
   ```

2. **Training the generator**: Finally, we can now run the training loop to train the DCGAN. As we will be jointly training the generator and the discriminator, the training routine will consist of both these steps – training the generator model and training the discriminator model – in an alternate fashion. We will begin with training the generator in the following code:

```
os.makedirs("./images_mnist", exist_ok=True)
for ep in range(num_eps):
    for idx, (images, _) in enumerate(dloader):
        # generate ground truths for real and fake images
        good_img = Variable(torch.FloatTensor(
            images.shape[0], 1).fill_(1.0),
            requires_grad=False)
        bad_img = Variable(torch.FloatTensor(
            images.shape[0], 1) .fill_(0.0),
            requires_grad=False)
        # get a real image
        actual_images = Variable(
            images.type(torch.FloatTensor))
        # train the generator model
        opt_gen.zero_grad()
        # generate a batch of images based on
        # random noise as input
        noise = Variable(torch.FloatTensor(
            np.random.normal(0, 1, (
                images.shape[0], lat_dimension))))
        gen_images = gen(noise)
        # generator model optimization - how well
        # can it fool the discriminator
        generator_loss = adv_loss_func(
            disc(gen_images), good_img)
        generator_loss.backward()
        opt_gen.step()
```

In the preceding code, we first generate the ground truth labels for real and fake images. Real images are labeled as `1`, and fake images are labeled as `0`. These labels will serve as the target outputs for the discriminator model, which is a binary classifier.

Next, we load a batch of real images from the MNIST dataset loader, and we also use the generator to generate a batch of fake images using random noise as input.

Finally, we define the generator loss as the adversarial loss between the following:

i. The probability of the realness of the fake images (produced by the generator model) as predicted by the discriminator model.

ii. The ground truth value of `1`.

Essentially, if the discriminator is fooled to perceive the fake generated image as a real image, then the generator has succeeded in its role, and the generator loss will be low. Once we have formulated the generator loss, we can use it to backpropagate gradients along the generator model in order to tune its parameters.

In the preceding optimization step of the generator model, we left the discriminator model parameters unchanged and simply used the discriminator model for a forward pass.

3. **Training the discriminator**: Next, we will do the opposite – that is, we will retain the parameters of the generator model and train the discriminator model:

```
        # train the discriminator model
        opt_disc.zero_grad()
        # calculate discriminator loss as average of
        # mistakes(losses) in confusing real images
        # as fake and vice versa
        actual_image_loss = adv_loss_func(disc(
            actual_images), good_img)
        fake_image_loss = adv_loss_func(disc(
            gen_images.detach()), bad_img)
        discriminator_loss = (
            actual_image_loss + fake_image_loss) / 2
        # discriminator model optimization
        discriminator_loss.backward()
        opt_disc.step()
        batches_completed = ep * len(dloader) + idx
        if batches_completed % logging_intv == 0:
            print(f"epoch number {ep} \
                  | batch number {idx} \
                  | generator loss = \
                    {generator_loss.item()}\
                  | discriminator loss = \
                    {discriminator_loss.item()}")
            save_image(
                gen_images.data[:25],
                f"images_mnist/{batches_completed}.png",
                nrow=5, normalize=True)
```

Remember that we have a batch of both real and fake images. In order to train the discriminator model, we will need both. We define the discriminator loss simply to be the adversarial loss or the binary cross entropy loss as we do for any binary classifier.

We compute the discriminator loss for the batches of both real and fake images, keeping the target values at `1` for the batch of real images and at `0` for the batch of fake images. We then use the mean of these two losses as the final discriminator loss and use this loss to backpropagate gradients to tune the discriminator model parameters.

After every few epochs and batches, we log the model's performance results – that is, the generator loss and the discriminator loss. For the preceding code, we should get an output similar to the following:

```
epoch number 0 | batch number 0 | generator loss = 0.7016811370849609 |
discriminator loss = 0.691551923751831
epoch number 0 | batch number 200 | generator loss = 0.6610271334648132 |
discriminator loss = 0.649569034576416
epoch number 0 | batch number 400 | generator loss = 1.4007172584533691 |
discriminator loss = 0.3584232032299042
...
epoch number 9 | batch number 1325 | generator loss = 2.019848585128784 |
discriminator loss = 0.7362138032913208
epoch number 9 | batch number 1525 | generator loss = 6.319630146026611 |
discriminator loss = 0.10377539694309235
epoch number 9 | batch number 1725 | generator loss = 1.8041318655014038
| discriminator loss = 0.6188918352127075
```

Notice how the losses are fluctuating a bit; that generally tends to happen during the training of GAN models due to the adversarial nature of the joint training mechanism. Besides outputting logs, we also save some network-generated images at regular intervals. *Figure 9.5* shows the progression of those generated images in the first few epochs:

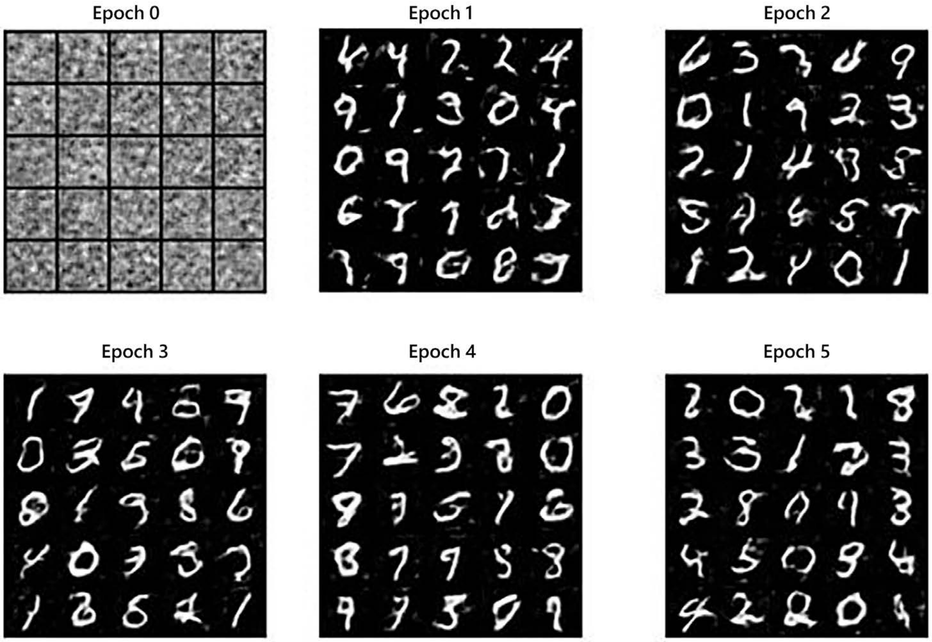


Figure 9.5: DCGAN epoch-wise image generation

If we compare the results from the later epochs to the original MNIST images in *Figure 9.4*, it looks like the DCGAN has learned reasonably well how to generate realistic-looking fake images of handwritten digits.

That is it. We have learned how to use PyTorch to build a DCGAN model from scratch. The original DCGAN paper has a few nuanced details, such as the normal initialization of the layer parameters of the generator and discriminator models, using specific *beta1* and *beta2* values for the Adam optimizers, and more. We have omitted some of those details in the interest of focusing on the main parts of the GAN code. These nuanced changes may impact the optimization space during model training, which can impact the model training time as well as the final optimal loss achieved. You are encouraged to incorporate those details and see how that changes the results.

Additionally, we have only used the `MNIST` database in our exercise. However, we can use any image dataset to train the DCGAN model. You are encouraged to try out this model on other image datasets. One popular image dataset that is used for DCGAN training is the celebrity faces dataset [6].

A DCGAN trained with this model can then be used to generate the faces of celebrities who do not exist. *ThisPersonDoesNotExist* [7] is one such project that generates the faces of humans that do not exist. Spooky? Yes. That is how powerful DCGANs and GANs in general are. Also, thanks to PyTorch, we can now build our own GANs in a few lines of code.

In the next and final section of this chapter, we will go beyond DCGANs and take a brief look at another type of GAN – the pix2pix model. The pix2pix model can be used to generalize the task of style transfer in images and, more generally, the task of image-to-image translation. We will discuss the architecture of the pix2pix model, its generator and discriminator, and use PyTorch to define the generator and discriminator models. We will also contrast pix2pix with a DCGAN in terms of their architecture and implementation.

Using GANs for style transfer

So far, we have only looked at DCGANs in detail. Hundreds of different types of GAN models exist already, and many more are in the making. Each of these GAN variants differs by either the application they are catering to, their underlying model architecture, or due to some tweaks in their optimization strategy, such as modifying the loss function. For example, **Super-Resolution GAN** (**SRGAN**) are used to enhance the resolution of a low-resolution image. The CycleGAN uses two generators instead of one, and the generators consist of ResNet-like blocks. The **Least Squares GAN** (**LSGAN**) uses the mean square error as the discriminator loss function instead of the usual cross-entropy loss used in most GANs.

It is impossible to discuss all of these GAN variants in a single chapter or even a book. However, in this section, we will explore one more type of GAN model that relates to both the DCGAN model discussed in the previous section and the neural style transfer model discussed in *Chapter 8*, *Neural Style Transfer*.

This special type of GAN generalizes the task of style transfer between images and, furthermore, provides a general image-to-image translation framework. It is called pix2pix, and we will briefly explore its architecture and the PyTorch implementation of its generator and discriminator components.

Understanding the pix2pix architecture

From *Chapter 8*, *Neural Style Transfer*, you may recall that a fully trained neural style transfer model only works on a given pair of images. pix2pix is a more general model that can transfer style between any pair of images once trained successfully. In fact, the model goes beyond just style transfer and can be used for any image-to-image translation application, such as background masking (removing the background of an image or video to isolate the foreground objects of interest, color palette completion (automatically generating a complete set of colors for an image or video, based on the existing color information in the input), and more.

Note

pix2pix is a conditional GAN that differs from regular GANs in that it uses paired input-output data during training, which allows it to learn a direct mapping between input and output images. This enables it to generate high-quality images with fine details, while other GANs may struggle with this due to the lack of paired training data. Additionally, pix2pix can generate images that satisfy specific constraints or requirements, making it useful for tasks such as image-to-image translation and image editing.

Essentially, pix2pix works like any GAN model. There is a generator and a discriminator involved. Instead of taking in random noise as input and generating an image, as shown in *Figure 9.1*, the generator in a pix2pix model takes in a real image as input and tries to generate a translated version of that image. If the task at hand is style transfer, then the generator will try to generate a style-transferred image.

Subsequently, the discriminator now looks at a pair of images instead of just a single image, as was the case in *Figure 9.1*. A real image and its equivalent translated image are fed as input to the discriminator. If the translated image is a genuine one, then the discriminator is supposed to output *1*, and if the translated image is generated by the generator, then the discriminator is supposed to output *0*.

Figure 9.6 shows the schematic for a pix2pix model:

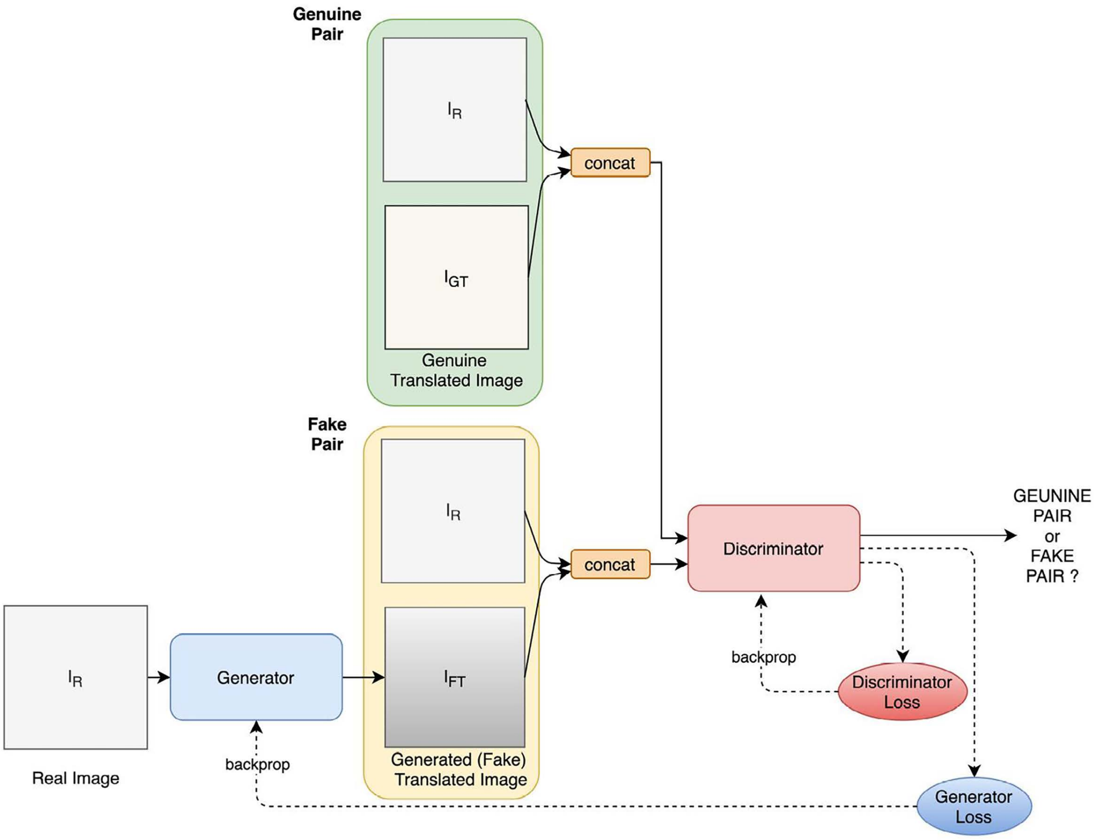


Figure 9.6: A pix2pix model schematic

Figure 9.6 shows significant similarities to *Figure 9.1*, which implies that the underlying idea is the same as a regular GAN. The only difference is that the real or fake question to the discriminator is posed on a pair of images as opposed to a single image.

Exploring the pix2pix generator

The generator sub-model used in the pix2pix model is a well-known CNN used for image segmentation – the **UNet**. *Figure 9.8* shows the architecture of the UNet, which is used as a generator for the pix2pix model:

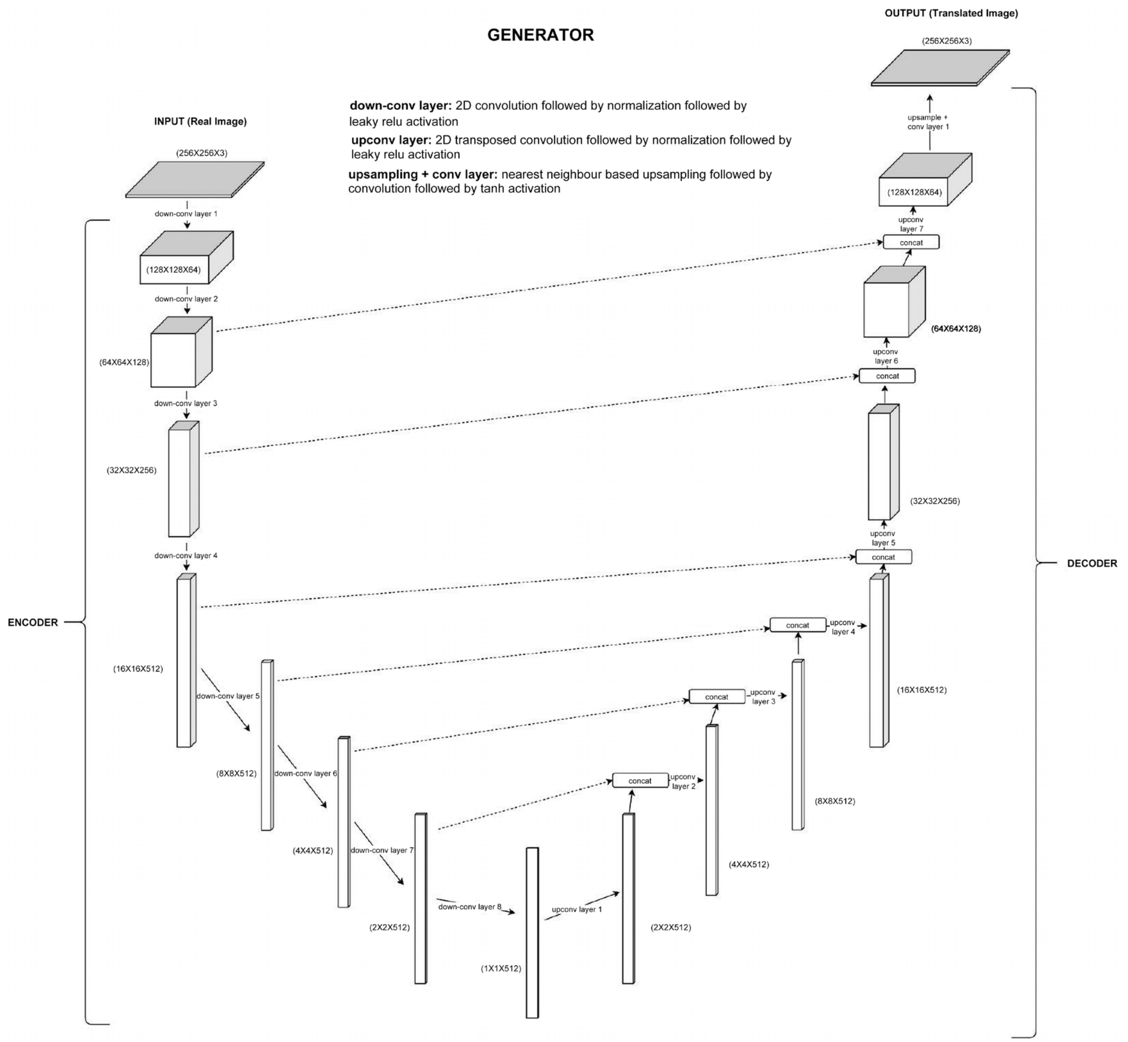


Figure 9.7: The pix2pix generator model architecture

Firstly, the name UNet comes from the *U* shape of the network, as is made evident from the preceding diagram. There are two main components in this network, as follows:

- From the upper-left corner to the bottom lies the encoder part of the network, which encodes the **256x256** RGB input image into a **512**-sized feature vector.
- From the upper-right corner to the bottom lies the decoder part of the network, which generates an image from the embedding vector of size **512**.

A key property of UNet is the **skip connections** – that is, the concatenation of features (along the depth dimension) from the encoder section to the decoder section, as shown by the dotted arrows in *Figure 9.7*.

FAQ – Why do we have encoder-decoder skip connections in UNet?

Using features from the encoder section helps the decoder to better localize the high-resolution information at each upsampling step.

Essentially, the encoder section is a sequence of down-convolutional blocks, where each down-convolutional block is itself a sequence of a 2D convolutional layer, an instance normalization layer, and a leaky ReLU activation. Similarly, the decoder section consists of a sequence of up-convolutional blocks, where each block is a sequence of a 2D-transposed convolutional layer, an instance normalization layer, and a ReLU activation layer.

The final part of this UNet generator architecture is a nearest neighbor-based upsampling layer, followed by a 2D convolutional layer, and, finally, a `tanh` activation function. Let's now look at the PyTorch code for the UNet generator:

1. Here is the equivalent PyTorch code for defining the UNet-based generator architecture:

```
class UNetGenerator(nn.Module):
    def __init__(self, chnls_in=3, chnls_op=3):
        super(UNetGenerator, self).__init__()
        self.down_conv_layer_1 = DownConvBlock(
            chnls_in, 64, norm=False)
        self.down_conv_layer_2 = DownConvBlock(64, 128)
        self.down_conv_layer_3 = DownConvBlock(128, 256)
        self.down_conv_layer_4 = DownConvBlock(
            256, 512, dropout=0.5)
        self.down_conv_layer_5 = DownConvBlock(
            512, 512, dropout=0.5)
        self.down_conv_layer_6 = DownConvBlock(
            512, 512, dropout=0.5)
        self.down_conv_layer_7 = DownConvBlock(
            512, 512, dropout=0.5)
        self.down_conv_layer_8 = DownConvBlock(
            512, 512, norm=False, dropout=0.5)
        self.up_conv_layer_1 = UpConvBlock(
            512, 512, dropout=0.5)
        self.up_conv_layer_2 = UpConvBlock(
            1024, 512, dropout=0.5)
        self.up_conv_layer_3 = UpConvBlock(
            1024, 512, dropout=0.5)
        self.up_conv_layer_4 = UpConvBlock(
            1024, 512, dropout=0.5)
        self.up_conv_layer_5 = UpConvBlock(1024, 256)
        self.up_conv_layer_6 = UpConvBlock(512, 128)
        self.up_conv_layer_7 = UpConvBlock(256, 64)
        self.upsample_layer = nn.Upsample(scale_factor=2)
        self.zero_pad = nn.ZeroPad2d((1, 0, 1, 0))
        self.conv_layer_1 = nn.Conv2d(128, chnls_op, 4, padding=1)
        self.activation = nn.Tanh()
```

As you can see, there are 8 down-convolutional layers and 7 up-convolutional layers. The up-convolutional layers have two inputs, one from the previous up-convolutional layer output and another from the equivalent down-convolutional layer output, as shown by the dotted lines in *Figure 9.7*.

2. We have used the `UpConvBlock` and `DownConvBlock` classes to define the layers of the UNet model. The following is the definition of these blocks, starting with the `UpConvBlock` class:

```
class UpConvBlock(nn.Module):
    def __init__(self, ip_sz, op_sz, dropout=0.0):
        super(UpConvBlock, self).__init__()
        self.layers = [
            nn.ConvTranspose2d(ip_sz, op_sz, 4, 2, 1),
            nn.InstanceNorm2d(op_sz), nn.ReLU(),]
        if dropout:
            self.layers += [nn.Dropout(dropout)]
    def forward(self, x, enc_ip):
        x = nn.Sequential(*(self.layers))(x)
        op = torch.cat((x, enc_ip), 1)
        return op
```

The transpose convolutional layer in this up-convolutional block consists of a 4x4 kernel with a stride of `2` steps, which essentially doubles the spatial dimensions of its output compared to the input.

In this transpose convolution layer, the 4x4 kernel is passed through every other pixel (due to a stride of 2) in the input image. At each pixel, the pixel value is multiplied by each of the 16 values in the 4x4 kernel.

The overlapping values of the kernel multiplication results across the image are then summed up, resulting in an output twice the length and twice the breadth of the input image. Also, in the preceding `forward` method, the concatenation operation is performed after the forward pass is done via the up-convolutional block.

3. Next, here is the PyTorch code for defining the `DownConvBlock` class:

```
class DownConvBlock(nn.Module):
    def __init__(self, ip_sz, op_sz, norm=True, dropout=0.0):
        super(DownConvBlock, self).__init__()
        self.layers = [nn.Conv2d(ip_sz, op_sz, 4, 2, 1)]
        if norm:
            self.layers.append(nn.InstanceNorm2d(op_sz))
        self.layers += [nn.LeakyReLU(0.2)]
        if dropout:
            self.layers += [nn.Dropout(dropout)]
    def forward(self, x):
        op = nn.Sequential(*(self.layers))(x)
        return op
```

The convolutional layer inside the down-convolutional block has a kernel of size 4x4, a stride of `2`, and the padding is activated. Because the stride value is `2`, the output of this layer is half the spatial dimensions of its input.

A leaky ReLU activation is also used for similar reasons as DCGANs – the ability to deal with negative inputs, which also helps with alleviating the vanishing gradients problem.

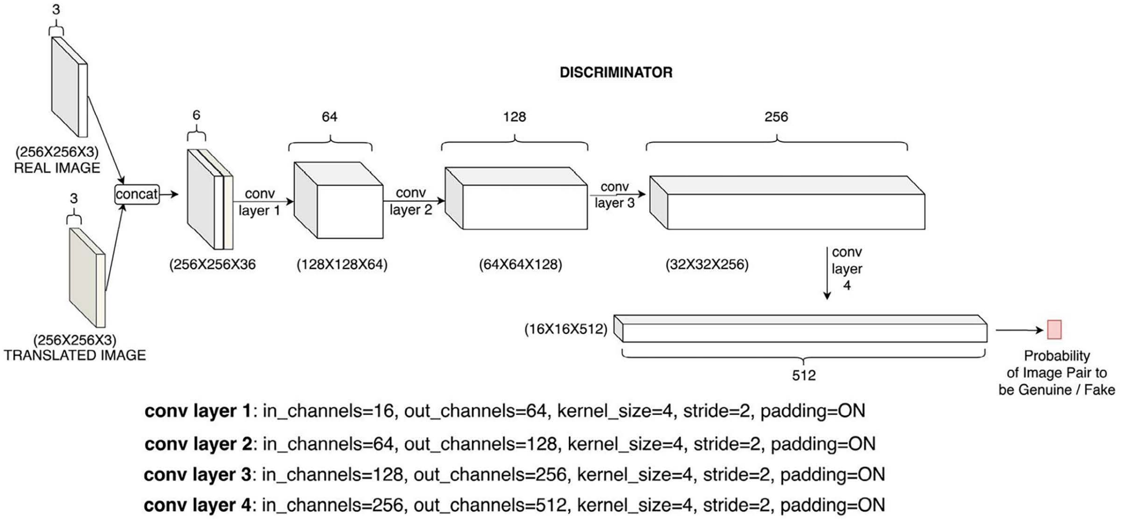
So far, we have seen the `__init__` method of our UNet-based generator. The `forward` method is pretty straightforward hereafter:

```
    def forward(self, x):
        enc1 = self.down_conv_layer_1(x)
        enc2 = self.down_conv_layer_2(enc1)
        enc3 = self.down_conv_layer_3(enc2)
        enc4 = self.down_conv_layer_4(enc3)
        enc5 = self.down_conv_layer_5(enc4)
        enc6 = self.down_conv_layer_6(enc5)
        enc7 = self.down_conv_layer_7(enc6)
        enc8 = self.down_conv_layer_8(enc7)
        dec1 = self.up_conv_layer_1(enc8, enc7)
        dec2 = self.up_conv_layer_2(dec1, enc6)
        dec3 = self.up_conv_layer_3(dec2, enc5)
        dec4 = self.up_conv_layer_4(dec3, enc4)
        dec5 = self.up_conv_layer_5(dec4, enc3)
        dec6 = self.up_conv_layer_6(dec5, enc2)
        dec7 = self.up_conv_layer_7(dec6, enc1)
        final = self.upsample_layer(dec7)
        final = self.zero_pad(final)
        final = self.conv_layer_1(final)
        return self.activation(final)
```

Having discussed the generator part of the pix2pix model, let's take a look at the discriminator model as well.

Exploring the pix2pix discriminator

The discriminator model, in this case, is also a binary classifier – just as it was for the DCGAN. The only difference is that this binary classifier takes in two images as inputs. The two inputs are concatenated along the depth dimension. *Figure 9.8* shows the discriminator model's high-level architecture:



conv layer 1: in_channels=16, out_channels=64, kernel_size=4, stride=2, padding=ON

conv layer 2: in_channels=64, out_channels=128, kernel_size=4, stride=2, padding=ON

conv layer 3: in_channels=128, out_channels=256, kernel_size=4, stride=2, padding=ON

conv layer 4: in_channels=256, out_channels=512, kernel_size=4, stride=2, padding=ON

Figure 9.8: The pix2pix discriminator model architecture

It is a CNN where the last 3 convolutional layers are followed by a normalization layer as well as a leaky ReLU activation. The PyTorch code to define this discriminator model will be as follows:

```
class Pix2PixDiscriminator(nn.Module):
    def __init__(self, chnls_in=3):
        super(Pix2PixDiscriminator, self).__init__()
        def disc_conv_block(chnls_in, chnls_op, norm=1):
            layers = [nn.Conv2d(chnls_in, chnls_op, 4, stride=2, padding=1)]
            if normalization:
                layers.append(nn.InstanceNorm2d(chnls_op))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers
        self.lyr1 = disc_conv_block(chnls_in * 2, 64, norm=0)
        self.lyr2 = disc_conv_block(64, 128)
        self.lyr3 = disc_conv_block(128, 256)
        self.lyr4 = disc_conv_block(256, 512)
```

As you can see, the 4 convolutional layers subsequently double the depth of the spatial representation at each step. Layers 2, 3, and 4 have added normalization layers after the convolutional layer, and a leaky ReLU activation with a negative slope of `0.2` is applied at the end of every convolutional block. Finally, here is the `forward` method of the discriminator model class in PyTorch:

```
    def forward(self, real_image, translated_image):
        ip = torch.cat((real_image, translated_image), 1)
        op = self.lyr1(ip)
        op = self.lyr2(op)
        op = self.lyr3(op)
        op = self.lyr4(op)
        op = nn.ZeroPad2d((1, 0, 1, 0))(op)
        op = nn.Conv2d(512, 1, 4, padding=1)(op)
        return op
```

First, the input images are concatenated and passed through the four convolutional blocks and finally led into a single binary output that tells us the probability of the pair of images being genuine or fake (that is, generated by the generator model). In this way, the pix2pix model is trained at runtime so that the generator of the pix2pix model can take in any image as input and apply the image translation function that it has learned during training.

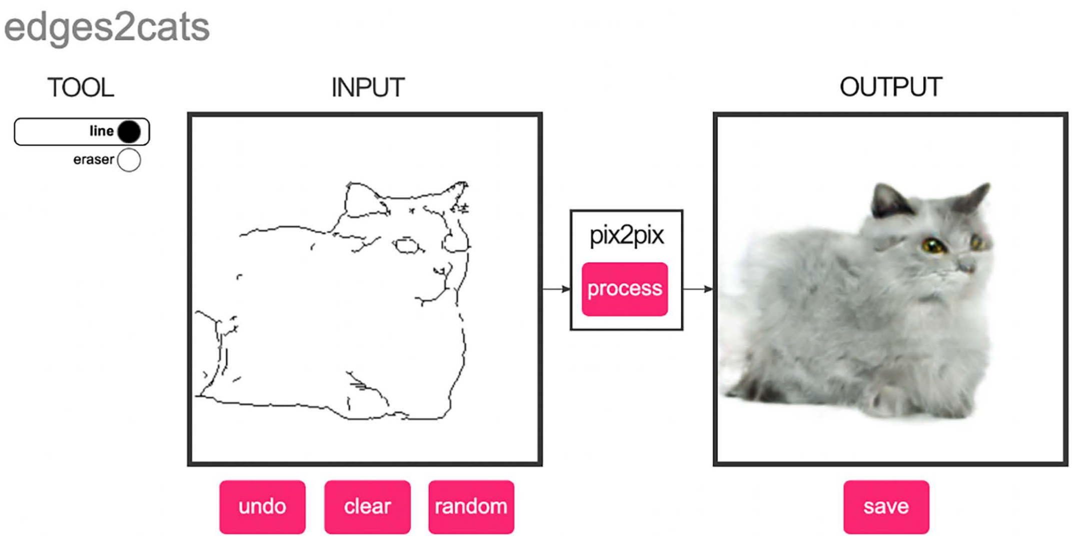


Figure 9.9: Example of pix2pix generation from edges to full image (image courtesy: https://affinelayer.com/pixsrv/)

One example of the translation function is drawing a full image as output using edges as input. *Figure 9.9* shows such an example where the pix2pix model generates a real-looking cat image from a sketch. You can try out several pix2pix demos at the link in this reference list [8].

The pix2pix model will be considered successful if the generated fake-translated image is difficult to tell apart from a genuine translated version of the original image.

This concludes our exploration of the pix2pix model. In principle, the overall model schematic for pix2pix is quite similar to that of the DCGAN model. The discriminator network for both of these models is a CNN-based binary classifier. The generator network for the pix2pix model is a slightly more complex architecture inspired by the UNet image segmentation model.

Overall, we have been able to both successfully define the generator and discriminator models for DCGAN and pix2pix using PyTorch, and understand the inner workings of these two GAN variants.

After finishing this section, you should be able to get started with writing PyTorch code for the many other GAN variants out there. Building and training various GAN models using PyTorch can be a good learning experience and certainly is a fun exercise. We encourage you to use the information from this chapter to work on your own GAN projects using PyTorch.

Summary

GANs have been an active area of research and development in recent years, ever since their inception in 2014. This chapter was an exploration of the concepts behind GANs, including their components – namely, the generator and the discriminator. We discussed the architectures of each of these components and the overall schematic of a GAN model.

In the next chapter, we will go a step further in our pursuit of generative models. We will explore how to generate images from text using cutting-edge deep learning techniques.

References

1. PyTorch GAN: `https://github.com/eriklindernoren/PyTorch-GAN`
2. *Generative Adversarial Nets*: `https://arxiv.org/pdf/1406.2661.pdf`
3. GitHub: `https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/Chapter08/dcgan.ipynb`
4. MNIST digits classification dataset: `https://keras.io/api/datasets/mnist/`
5. *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*: `https://arxiv.org/pdf/1511.06434.pdf`
6. CelebA dataset: `http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html`
7. This Is Not A Person: `https://this-person-does-not-exist.com/`
8. Image-to-image demo: `https://affinelayer.com/pixsrv/`

Learn more on Discord

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

`https://packt.link/mastorch`

10

Image Generation Using Diffusion

We learned how to generate images using GANs in *Chapter 9*, *Deep Convolutional GANs*. In this chapter, we will explore a more recent approach to generating images – by using **diffusion**. The idea behind diffusion, just like all great ideas, is pretty simple and intuitive. We will first understand how diffusion works. We will then use PyTorch to train a diffusion model from scratch to generate realistic images. We will then further our understanding of diffusion for generating images from text. Finally, we will generate some high-quality, realistic images from text using a pre-trained diffusion model with the help of **PyTorch** and **Hugging Face**.

By the end of this chapter, you will understand the core idea behind most of the cutting-edge generative AI models in computer vision, such as **Stable Diffusion, DALL-E, Imagen, Midjourney**, and so on. You will learn how to train a diffusion model from scratch using PyTorch, and how to use cutting-edge diffusion models with the help of Hugging Face.

This chapter is broken down into the following topics:

- Understanding **image generation** using diffusion
- Training a **diffusion model** for image generation
- Understanding **text-to-image** generation using diffusion
- Using the **Stable Diffusion** model to generate images from text

Understanding image generation using diffusion

Does *Figure 10.1* seem normal to you?

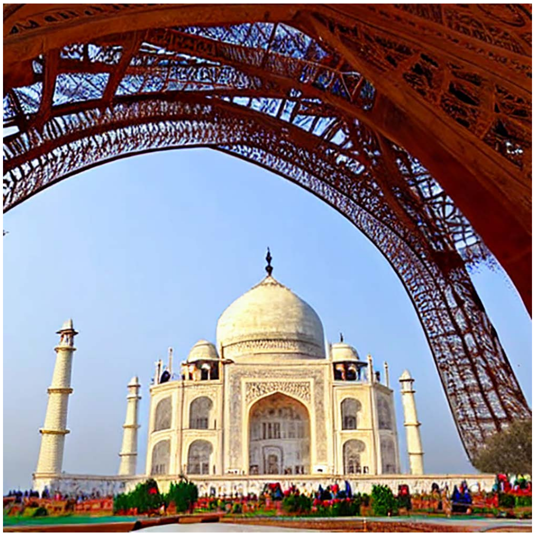


Figure 10.1: AI-generated image using diffusion showing the Taj Mahal, India, right next to the Eiffel Tower, France

The Eiffel Tower and the Taj Mahal are situated in two different countries. However, this AI-generated image places them next to each other in a parallel world. This image was generated starting from random noise using a process called **diffusion**, as shown in *Figure 10.2*.

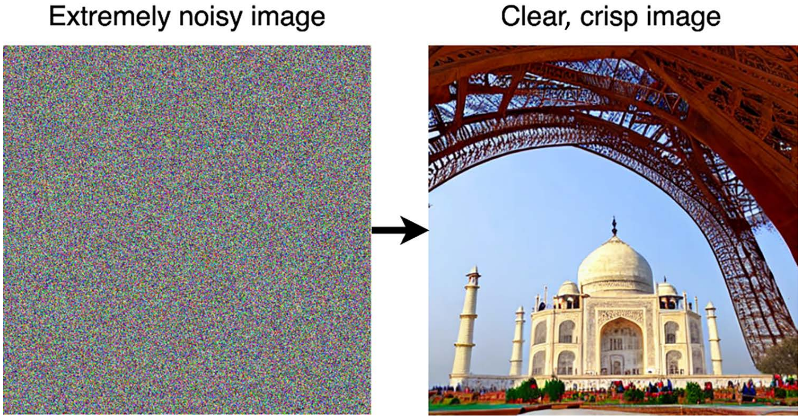


Figure 10.2: Generating a realistic image from pure noise using diffusion

Diffusion in the context of generative AI consists of a step-by-step process where simple noise is transformed multiple times to create diverse realistic data, resulting in high-quality sample generation by refining noise iteratively until it resembles the desired data, as demonstrated in *Figure 10.3*.

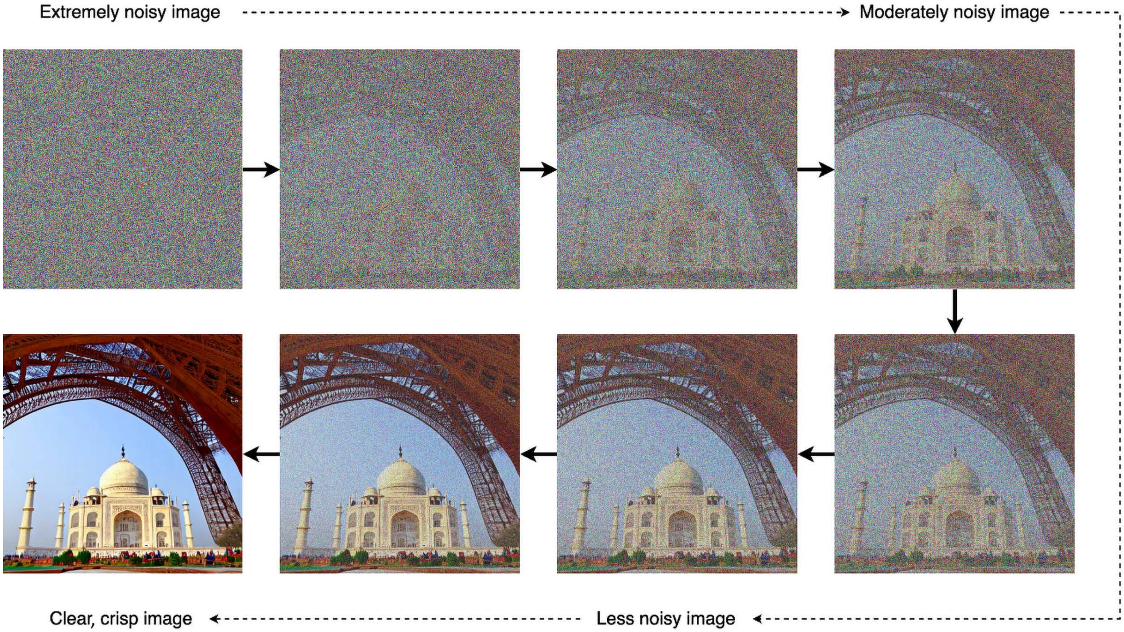


Figure 10.3: Diffusion as a step-by-step process of denoising an initial noisy image into a photo-realistic image

But how does diffusion work? How do we get from pure noise to a meaningful image? Let us dive into it.

Understanding how diffusion works

Deriving a realistic image from pure noise, as shown in *Figure 10.2*, can be generalized to the objective of deriving a less noisy image from a more noisy image, as can be seen upon inspecting any two consecutive images in *Figure 10.3*. To derive a less noisy image from a more noisy image, we first train a deep learning model with a somewhat opposite objective – to learn pure noise contained within a noisy image, referred to as **forward diffusion**, as shown in *Figure 10.4*.

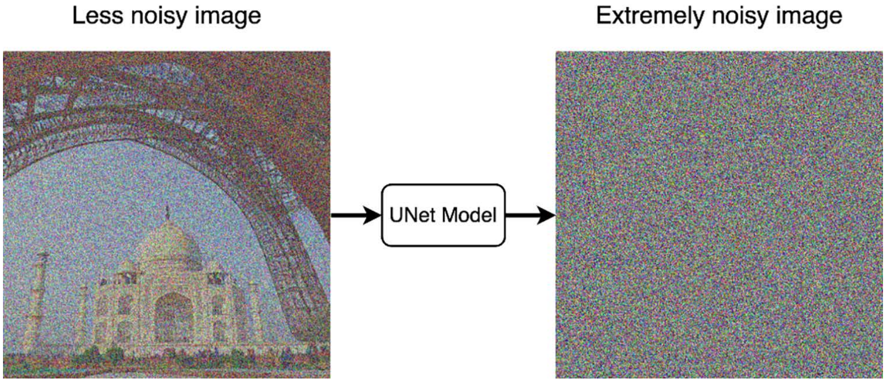


Figure 10.4: Using a UNet model to learn only the noise contained within a noisy image

The deep learning model used here is the UNet model, which is capable of generating an output of the same spatial dimensions as the input, as we discussed earlier, in *Chapter 9*, *Deep Convolutional GANs*, as part of the *pix2pix* model (see *Figure 9.8*). Two questions immediately arise from *Figure 10.4*:

1. How do we train such a UNet model?
2. Why are we learning the noise contained within noisy images? How does it help with generating realistic images?

Let us first understand how to train the UNet model.

Training a forward diffusion model

Preparing a dataset for such a model is quite intuitive. Let us take the example of the Taj Mahal image. You can find this image file on GitHub [1]. With the following lines of code, we can add noise to the image:

```
from PIL import Image
import numpy as np
img = Image.open("./taj.png")
# 256 is the max px value, 3 is num_channels
noise = 256*np.random.rand(*img.size, 3)
noisy_img = ((img + noise)/2).astype(np.uint8)
Image.fromarray(noisy_img)
```

This should produce an output like that shown in *Figure 10.5*.

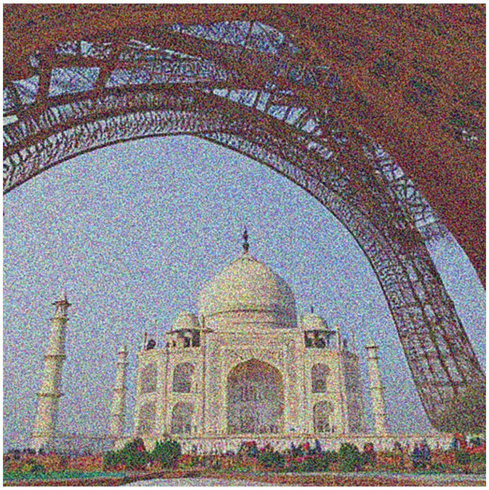


Figure 10.5: Noisy image generated by adding noise to a noiseless image

The image in *Figure 10.5* becomes an input sample for UNet, as shown in *Figure 10.4*.

Note

The methodology shown for adding noise to the image in this section is for demonstration of the overall idea. We'll learn about the specifics of adding noise to the image in the next section while building an actual model using PyTorch.

As for the output image, you simply need the noise itself:

```
Image.fromarray(noise.astype(np.uint8))
```

This should produce an output similar to *Figure 10.6*.

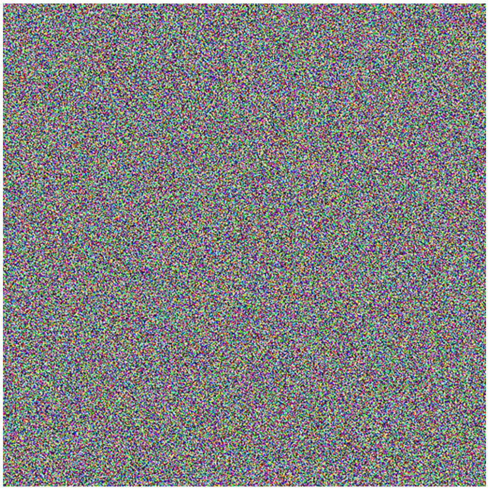


Figure 10.6: Pure noise image generated by using the NumPy random.rand function

Similar to the input and output image pairs produced in *Figure 10.5* and *Figure 10.6* respectively, we can generate a large number of dataset samples for the UNet model by using different real images and adding noise to them.

Note

For each image, we generate random noise with a different seed to have a variety of noise signals added to different images so that the UNet model learns to predict all kinds of noise patterns from the input data.

Moreover, the amount of noise added can also be varied to make the dataset even more varied. To generate the image in *Figure 10.5*, we summed the original image and noise in equal proportion (50% each). We can either repetitively add noise to the image or increase (or decrease) the amount of noise to generate more noisy UNet input samples, as shown in the following lines of code:

```
noisy_img = ((img + 3*noise)/4).astype(np.uint8)
Image.fromarray(noisy_img)
```

This should produce an output similar to *Figure 10.7*.

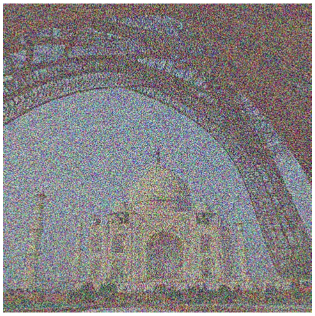


Figure 10.7: Noisy image generated by adding 75% noise and 25% original image

Now that we have learned how to generate the dataset for the UNet model, we will move on to training the model, which is straightforward, as demonstrated in *Figure 10.8*.

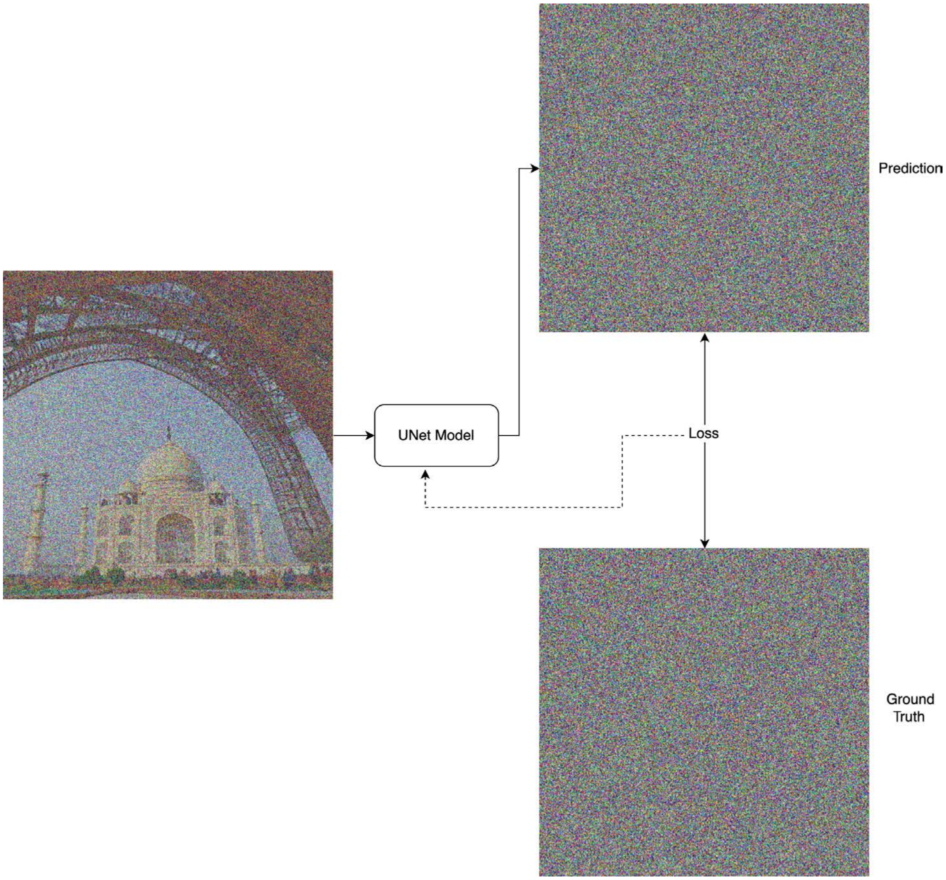


Figure 10.8: Training a UNet model to learn noise from a noisy image

We'll train this model using PyTorch in the next section. For now, let us address the second point related to UNet – how do we use this model to generate realistic images?

Performing reverse diffusion or denoising

What we have done so far is the forward diffusion process. To achieve realistic images, we need to perform **denoising** or **reverse diffusion**. More precisely, we use the trained UNet model to predict the noise signal contained within an extremely noisy image, and we then subtract this predicted noise from the extremely noisy image to derive a less noisy image, as shown in *Figure 10.9*.

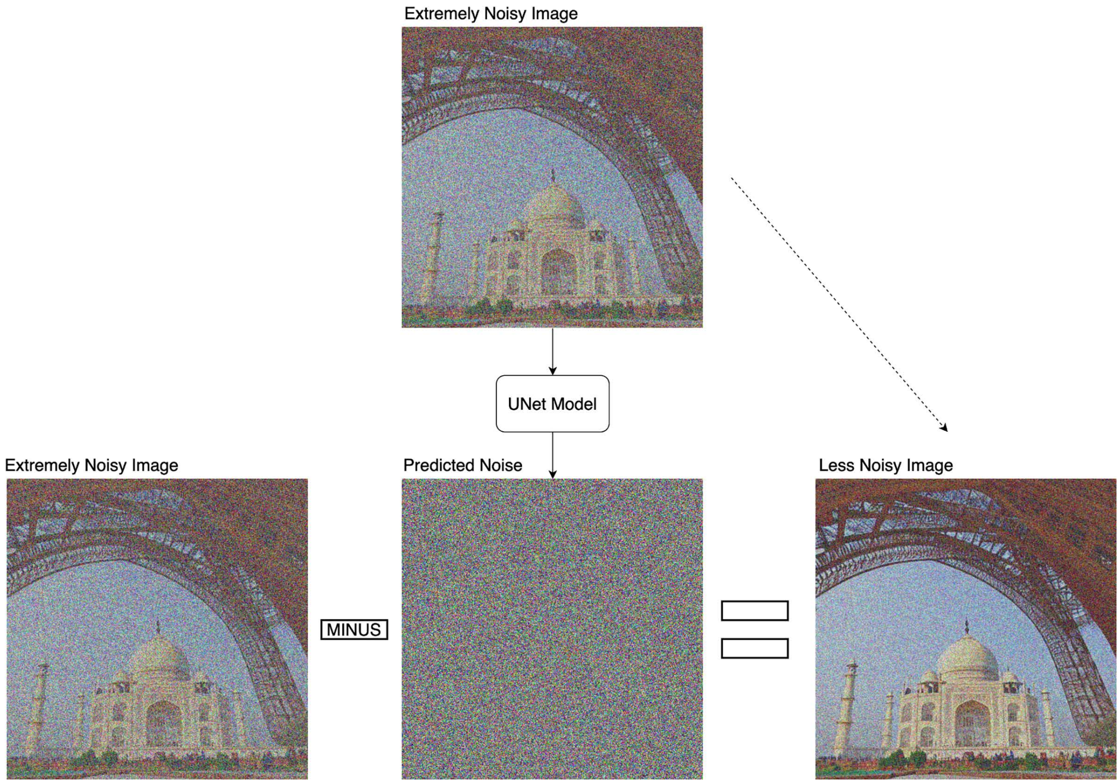


Figure 10.9: Denoising or reverse diffusion process where (predicted) noise is subtracted from a noisy image to create a less noisy image

If we start with pure noise and repeat the reverse diffusion steps enough times, we'll end up generating a high-quality realistic image, as shown in *Figure 10.10*. It all happens in small increments. The topmost image in *Figure 10.10* contains a meaningful picture overlaid with extreme amounts of noise, making the image look like practically pure noise. The trained UNet model knows how to extract pure noise from the extremely noisy image because it was trained for that precise objective.

Note

Running UNet on the original noisy image with different random seeds will yield slightly different results that would incrementally, over hundreds and thousands of iterations, result in different final images.

Subtracting the UNet-extracted noise from the original extremely noisy image renders a less noisy image. Repeating this process hundreds or thousands of times ultimately produces an image with virtually zero noise, and all that remains is the meaningful picture/content.

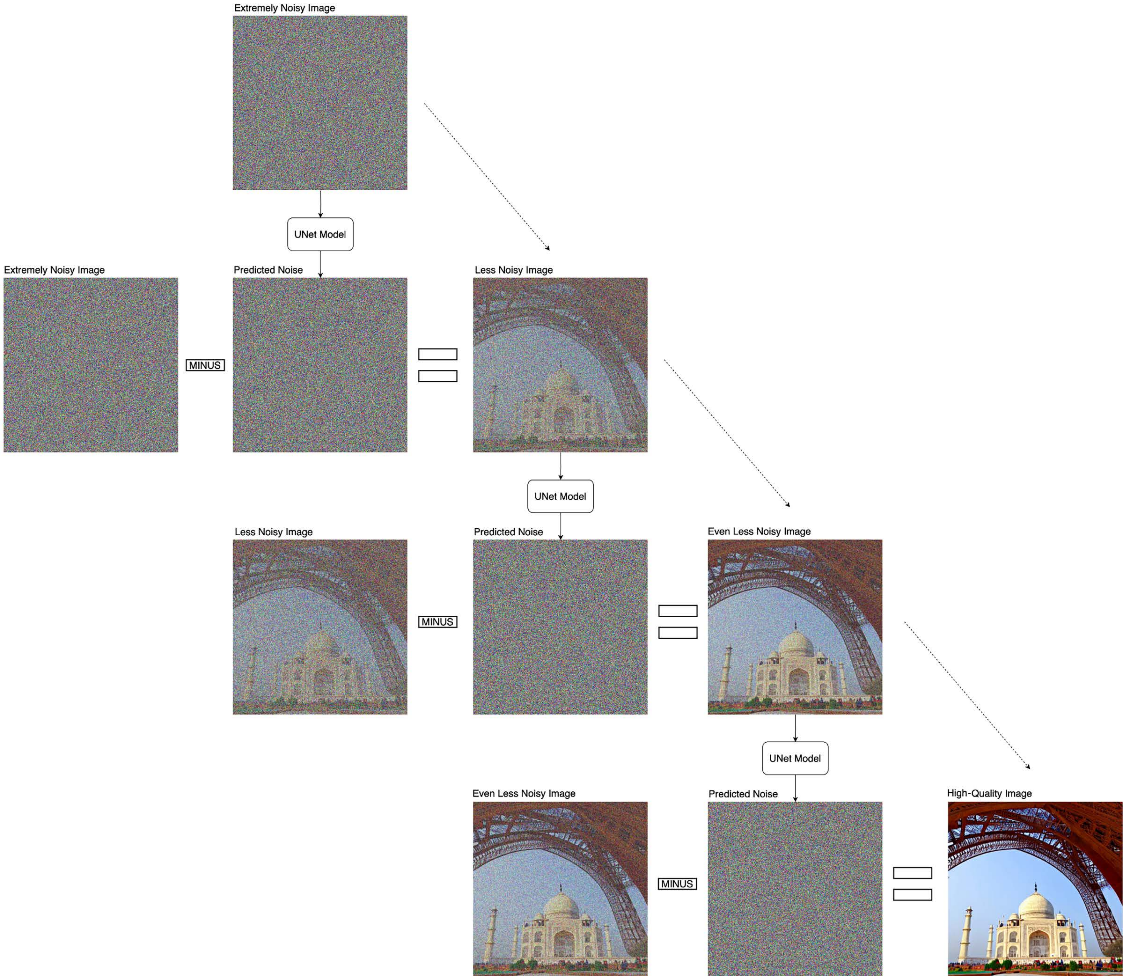


Figure 10.10: Several denoising steps are performed on input noise to generate a high-quality, realistic image

Note that *Figure 10.10* is a deeper dive into how the process demonstrated in *Figure 10.3* works. In this figure, we perform four denoising steps to get from noise to a realistic image. In reality, a higher number of denoising steps are performed (for example, 50) to obtain high-quality images. The process depicted in *Figure 10.10*, leading us from noise to realistic images, constitutes what is referred to as the **Denoising Diffusion Probabilistic Model** (**DDPM**) [2]. We'll use DDPM in the next section to build a diffusion model from scratch, written using PyTorch, that generates images starting from noise.

Training a diffusion model for image generation

In this section, we'll implement a diffusion model from scratch using PyTorch. By the end, this model will be able to generate realistic, high-quality images. Besides PyTorch, we'll use Hugging Face (an open-source platform that offers diverse AI tools and a collaborative hub for sharing and accessing pre-trained AI models and datasets) to load an image dataset. Besides the dataset, we'll use the `diffusers` library [3] from Hugging Face, which provides implementations for models such as UNet and DDPM. We'll also use Hugging Face's `accelerate` library [4] to speed up the diffusion training process by utilizing the **Graphical Processing Unit** (**GPU**). We'll learn more about Hugging Face in *Chapter 19*, *PyTorch and Hugging Face*.

Note

GPUs might not be readily available to you. In that case, you can access GPUs via Google Colab: `https://colab.google/`.

All code for this section is available on GitHub [5]. Before proceeding with the code, we need to install the following dependencies:

```
pip install torch torchvision datasets diffusers accelerate
```

Loading the dataset using Hugging Face datasets

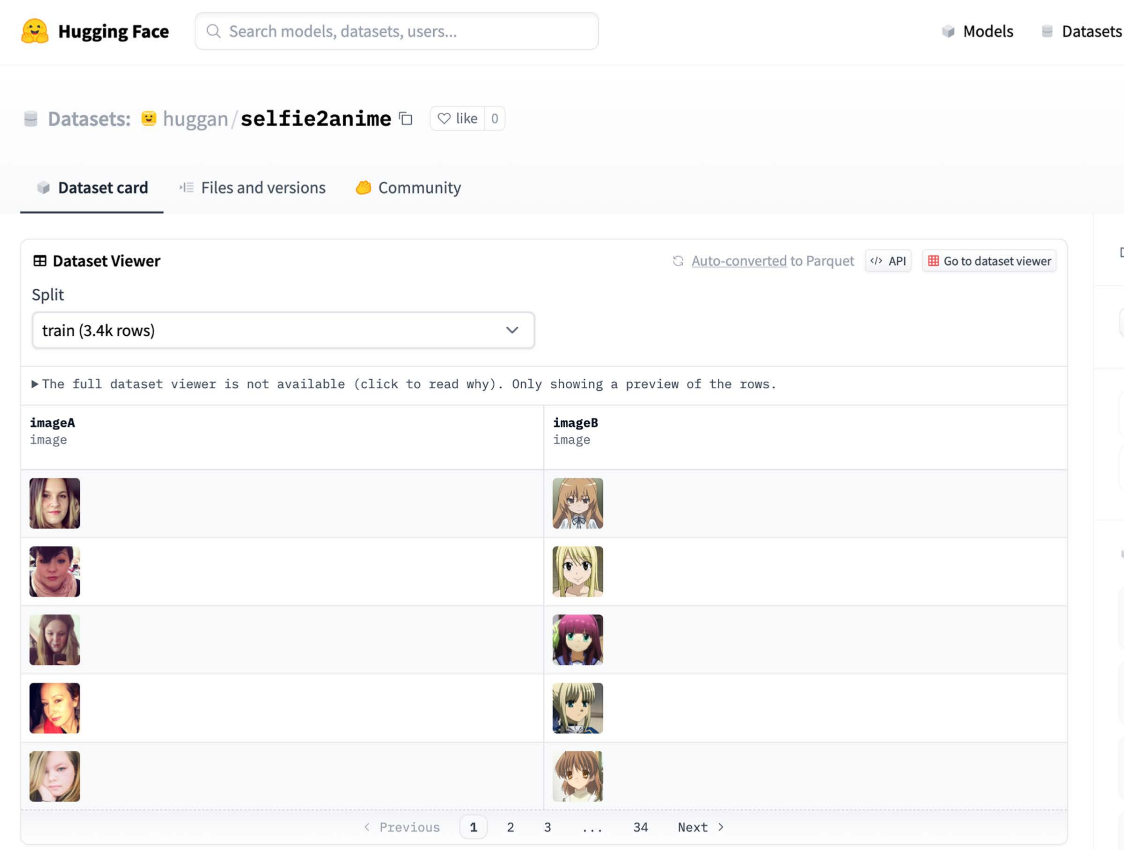
Let us begin with loading the dataset. We'll load an image dataset containing anime faces from Hugging Face called the *selfie2anime* dataset. As the name suggests, the dataset is originally meant to train models that can predict anime faces given real faces. However, we will use only the anime images part of the dataset. As an alternative, you may use the real faces as well. To load the dataset, we write the following lines of code:

```
from datasets import load_dataset
dataset = load_dataset("huggan/selfie2anime", split="train")
```

If you print the `dataset` object, you shall see the following:

```
Dataset({
    features: ['imageA', 'imageB'],
    num_rows: 3400
})
```

This indicates that there is a total of 3,400 images in this dataset. We can also check the dataset using the Hugging Face website [6], as shown in *Figure 10.11*.



Split

train (3.4k rows)

▸The full dataset viewer is not available (click to read why). Only showing a preview of the rows.

Figure 10.11: The selfie2anime dataset page from the Hugging Face website

From *Figure 10.11*, we can see that `imageB` refers to the anime images, which we are interested in. We can access them from the `dataset` object with the following command:

```
dataset["imageB"]
```

This should produce an output like the following:

```
[<PIL.PngImagePlugin.PngImageFile image mode=RGB size=256x256>,
 … (3400 repetitions)
 <PIL.PngImagePlugin.PngImageFile image mode=RGB size=256x256>]
```

The dataset is essentially a list of `PIL.Image` objects where each image is `256x256x3` in size (3 because of the RGB channels). Let us examine one of these images from the list with the following lines of code:

```
img = dataset["imageB"][0]
img
```

This should produce an output like that shown in *Figure 10.12*.

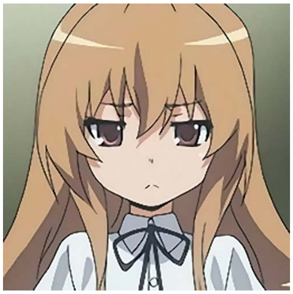


Figure 10.12: A sample image from the anime dataset

Using a utility function from the `diffusers` library, we can display a 4x4 grid of such images with the following lines of code:

```
from diffusers.utils import make_image_grid
make_image_grid(dataset["imageB"][:16], rows=4, cols=4)
```

This should produce an output like that shown in *Figure 10.13*.

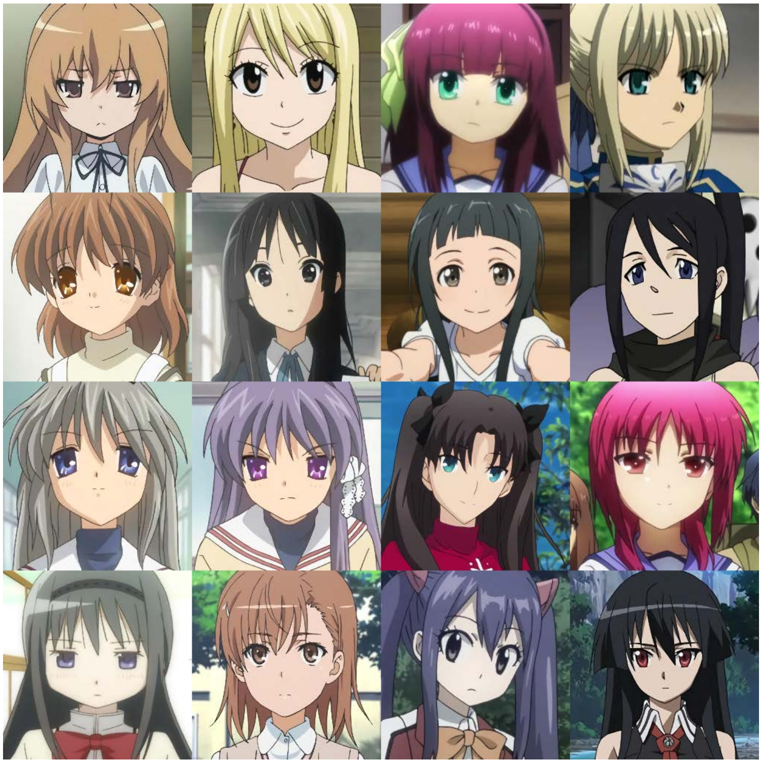


Figure 10.13: A 4x4 grid of sample images from the anime dataset

Now that we have loaded the dataset, let us remind ourselves of the task we want to perform with this dataset – to be able to generate realistic anime faces from pure noise, as shown in *Figure 10.14*.

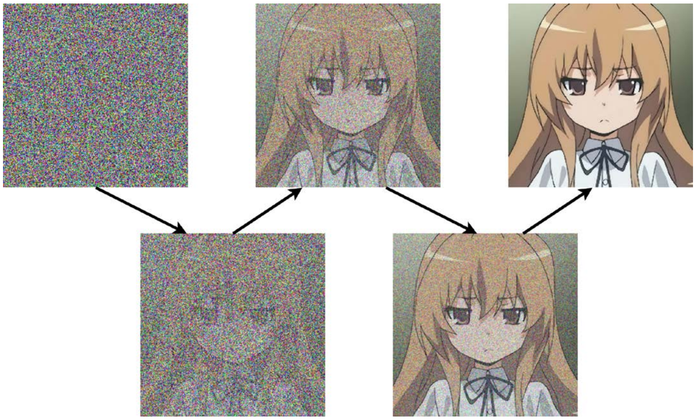


Figure 10.14: A 4x4 grid of sample images from the anime dataset

Before we define and train the diffusion model, we need to process the dataset.

Processing the dataset using torchvision transforms

With the following lines of code, we apply some transformations to the anime images dataset to make them usable for training the UNet model:

```
from torchvision import transforms
IMAGE_SIZE = 128
# add horizontal flipping to augment data as it
# still retains the facial structure while producing new image

# normalizing pixel values ensure that the mean pixel
# value is 0.5, with a standard deviation of 0.5
preprocess = transforms.Compose(
    [
        transforms.Resize((IMAGE_SIZE, IMAGE_SIZE)),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.5], [0.5]),
    ]
)
def transform(examples):
    images = [preprocess(image) for image in examples["imageB"]]
    return {"images": images}
dataset.set_transform(transform)
```

We use the `torchvision.transforms` API to transform the anime images. Firstly, we resize the 256x256x3 images to 128x128x3 to be able to train a diffusion model with less GPU (or CPU) memory and in less time.

> **Note**
>
> The code discussed in this chapter can run on both GPUs and CPUs. However, GPUs are preferred because the training time on CPUs alone may be too long (several days).

We then perform a data augmentation technique of randomly horizontally flipping the anime images, because horizontally flipping them will still retain the face structure and orientation while producing a different image than the original. Then, we convert the `PIL.Image` objects to PyTorch tensors and normalize the pixel values with `0.5` mean and `0.5` standard deviation of pixel values. These transformations are applied to each image of the Hugging Face dataset object with the help of the `transform` function called inside the `set_transform` method. The mean and standard deviation values ensure that the normalized pixel values have a confined mean and standard deviation of `0.5` each, which helps with model training stability.

After converting the PIL images into PyTorch tensors, we are ready to create our training dataloader with the following lines of code:

```
import torch
BSIZE = 16  # batch size
train_dataloader = torch.utils.data.DataLoader(
    dataset, batch_size=BSIZE, shuffle=True)
```

We define a batch size of `16`, and thanks to the Hugging Face `datasets` library, we are swiftly able to convert the created Hugging Face `dataset` object into a PyTorch dataloader. Now that we have all 3,400 anime images in the expected PyTorch format, let us add noise to these images to create training samples for the UNet model.

Adding noise to images using diffusers

The `diffusers` library from Hugging Face provides tools and pre-built models for creating generative AI models using diffusion processes. In this section, we will use one such tool from the diffusers toolkit to perform forward diffusion on anime images by adding noise to the images. We add different levels of noise to different images as demonstrated in *Figure 10.5* and *Figure 10.7*. To do this, we need to create a **noise scheduler** [7] with the following lines of code:

```
from diffusers import DDPMScheduler
noise_scheduler = DDPMScheduler(num_train_timesteps=1000)
```

The number of **timesteps** indicates the levels or layers of noise we want to mix with the original anime images. The greater the number of timesteps, the more we iteratively add noise on top of the image. We can test this functionality with the following lines of code:

```
clean_images = next(iter(train_dataloader))["images"]
# Sample noise to add to the images
noise = torch.randn(clean_images.shape, device=clean_images.device)
bs = clean_images.shape[0]  # batch size
# Get timesteps from 10 to 160 for each of the 16 images
timesteps = torch.range(10, 161, 10, dtype=torch.int64)
# Add noise to the clean images according to the noise
# magnitude at each timestep
# (this is the forward diffusion process)
noisy_images = noise_scheduler.add_noise(clean_images, noise, timesteps)
```

First, we fetch a batch of 16 anime images processed as tensors. We then initialize a random noise signal of the same dimensions as an anime image tensor. We then define 16 different timesteps for the 16 different images, starting from `10` up to `160`.

Note

The noise signal for the entire batch of 16 images remains the same. All that changes is the extent of the noise addition. Across batches, we get different noise distributions to help the model learn all kinds of noise patterns.

Then, we apply noise to the 16 different images as per the respective timesteps. The first image gets noise addition iteratively for 10 timesteps, the second image for 20 timesteps, the sixteenth image for 160 timesteps, and so on. We then visualize the original image tensors and the resulting noisy image tensors with the following lines of code:

```
# visualize original images
make_image_grid([transforms.ToPILImage()(clean_image) for clean_image in clean_
images], rows=4, cols=4)
# visualize noisy images
make_image_grid([transforms.ToPILImage()(noisy_image) for noisy_image in noisy_
images], rows=4, cols=4)
```

Running the above two lines in two different notebook cells should produce outputs similar to the images shown in *Figure 10.15*.

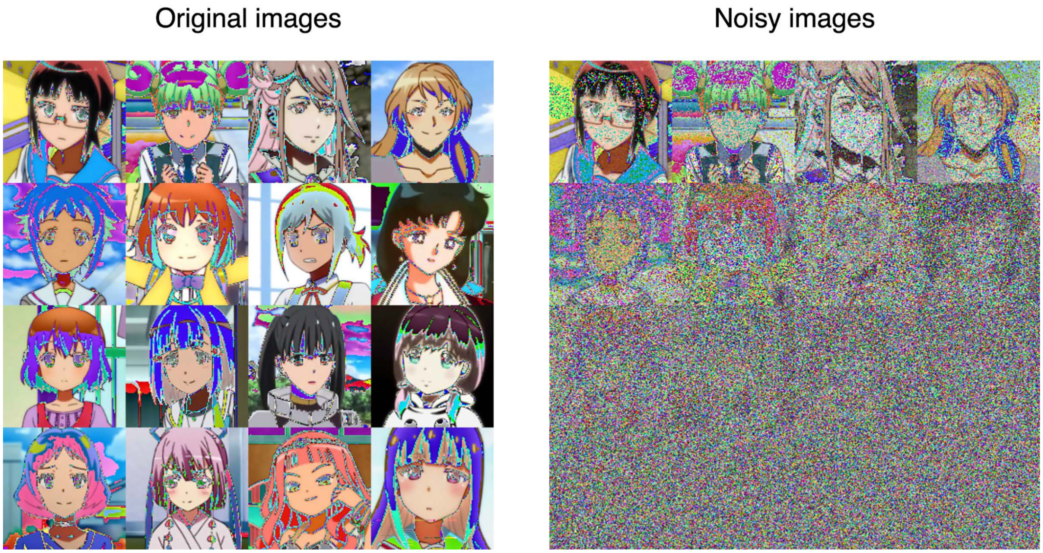


Figure 10.15: Adding noise to anime image tensors by using a noise scheduler. The noise scheduler iteratively adds more and more noise at each timestep. We increase the timesteps from the top-left image (10) to the bottom-right image (160)

While we used an increasing range of timesteps for demonstration in the above code, in reality, we fetch 16 different random timesteps between `1` and `1,000` to apply different noise levels to the 16 different images of a batch, as shown in the following lines of code:

```
# Sample a random timestep for each image
timesteps = torch.randint(
    0, noise_scheduler.config.num_train_timesteps, (bs,),
    device=clean_images.device, dtype=torch.int64
)
```

We now have the mechanism to generate the training dataset batches inside the model training loop. It is time to create the UNet model.

Defining the UNet model

Thanks to the `diffusers` library from Hugging Face, we define the UNet model with a few lines of code, as shown below:

```
from diffusers import UNet2DModel
model = UNet2DModel(
    sample_size=IMAGE_SIZE,  # the target image resolution
    in_channels=3,  # the number of input channels, 3 for RGB images
    out_channels=3,  # the number of output channels
    layers_per_block=2,     # how many ResNet layers to use per UNet block
    # the number of output channels for each UNet block
    block_out_channels=(128, 128, 256, 256, 512, 512),
    down_block_types=(
        "DownBlock2D",      # a regular ResNet downsampling block
        "DownBlock2D",
        "DownBlock2D",
        "DownBlock2D",
        "AttnDownBlock2D",  # a ResNet downsampling block
                            # with spatial self-attention
        "DownBlock2D",
    ),
    up_block_types=(
        "UpBlock2D",        # a regular ResNet upsampling block
        "AttnUpBlock2D",    # a ResNet upsampling block
                            # with spatial self-attention
        "UpBlock2D",
        "UpBlock2D",
        "UpBlock2D",
        "UpBlock2D",
    ),
)
```

To recall what this model architecture looks like, refer to *Figure 9.8* from *Chapter 9*, *Deep Convolutional GANs*. The above code specifies the input and output dimensions, which are the same in this case because we want to generate an output that has the same shape as the input. The code specifies the number of down-sampling and up-sampling blocks as well as the number of feature maps (or channels) within each of those blocks.

We have defined the model and the training dataset is already in place. The next step is to train the model.

Training the UNet model

In this section, we first set up the UNet model training components, such as the optimizer and learning rate. We then set up the `accelerate` dependencies from Hugging Face to speed up model training. Finally, we run the model training loop.

Defining the optimizer and learning schedule

Before we go to the model training loop, we need to define the optimizer and the learning schedule to train the UNet model:

```
from diffusers.optimization import get_cosine_schedule_with_warmup
# hyperparameters inspired from
# https://huggingface.co/docs/diffusers/en/tutorials/basic_training
```

```
NUM_EPOCHS = 20
LR = 1e-4
LR_WARMUP_STEPS = 500
optimizer = torch.optim.AdamW(model.parameters(), lr=LR)
lr_scheduler = get_cosine_schedule_with_warmup(
    optimizer=optimizer,
    num_warmup_steps=LR_WARMUP_STEPS,
    num_training_steps=(len(train_dataloader) * NUM_EPOCHS),
)
```

In the above code, we define the number of epochs to train the UNet model, and we define a specific learning schedule called the *cosine schedule with warmup* [8]. This schedule is directly available from the `diffusers` library. Essentially, it starts the learning rate from value `0` and *linearly* raises it up to the defined learning rate value (`LR=1e-4`) for the first `500` (*warmup*) steps, and then decreases the value from `1e-4` to `0`, following a *cosine* behavior from step number `501` to `4,250`, where `4,250` is the multiplication of `213`, the length of the training dataloader (which is `3,400` total samples divided by the batch size of `16`), and `20`, the number of epochs. We will observe the learning rate behavior on the TensorBoard upon training the model later in this section.

Besides the learning rate schedule, we also define the model optimizer as `AdamW` [9], which is the Adam optimizer with an added method to decay weight. Next, we'll define a model training accelerator.

Using Hugging Face Accelerate to accelerate training

We need a GPU to train the model in a reasonable amount of time (less than a day), because CPUs alone may take several days. To utilize the GPU for training the UNet model, we use another library from Hugging Face called Accelerate [4]. This library enables any PyTorch training code to maximally utilize the available hardware, be it one or more CPUs, GPUs, or **Tensor Processing Units** (**TPUs**). We discuss accelerate in more detail in *Chapter 19*, *PyTorch and Hugging Face*. For now, we need to simply set up an accelerator object with the following lines of code:

```
import os
from accelerate import Accelerator
MODEL_SAVE_DIR = "anime-128"
# Initialize accelerator and tensorboard logging
accelerator = Accelerator(
    mixed_precision="fp16",
    log_with="tensorboard",
    project_dir=os.path.join(MODEL_SAVE_DIR, "logs"),
)
```

As you can see, the `accelerate` library also provides additional features such as *mixed precision* and TensorBoard integration. We'll learn more about mixed precision in *Chapter 12*, *Model Training Optimizations*.

Next, we'll perform a few more steps, such as creating a model saving directory and allocating the model and the dataset to the defined accelerator:

```
if accelerator.is_main_process:
    if MODEL_SAVE_DIR is not None:
        os.makedirs(MODEL_SAVE_DIR, exist_ok=True)
    accelerator.init_trackers("train_example")
model, optimizer, train_dataloader, lr_scheduler = accelerator.prepare(
        model, optimizer, train_dataloader, lr_scheduler
    )
```

Having set up the accelerator object, we are now ready to train the UNet model.

Running the model training loop

We run the model training loop with the following lines of code:

```
global_step = 0
for epoch in range(NUM_EPOCHS):
    for step, batch in enumerate(train_dataloader):
        clean_images = batch["images"]
        ...
        noisy_images = noise_scheduler.add_noise(clean_images,
                                                 noise, timesteps)
        with accelerator.accumulate(model):
            # Predict the noise residual
            noise_pred = model(noisy_images, timesteps, return_dict=False)[0]
            loss = F.mse_loss(noise_pred, noise)
            accelerator.backward(loss)
            accelerator.clip_grad_norm_(model.parameters(), 1.0)
            optimizer.step()
            lr_scheduler.step()
            optimizer.zero_grad()
        logs = {"loss": loss.detach().item(),
                "lr": lr_scheduler.get_last_lr()[0], "step": global_step}
        accelerator.log(logs, step=global_step)
        global_step += 1
```

The most important part of the model training loop is the lines starting from predicting the noise residual. We use the UNet model to predict noise signals from the 16 noisy anime images of a training batch. We then compute the *mean squared error* loss between the predicted noise and the actual noise that was added to the anime images. This loss is then backpropagated through the UNet model to update the model parameters, as demonstrated earlier, in *Figure 10.8*. This code should produce an output that looks like the following:

```
Epoch 0: 100%|██████████| 213/213 [02:52<00:00,  1.24it/s, loss=0.0501,
lr=4.26e-5, step=212]
...
Epoch 20: 100%|██████████| 213/213 [02:49<00:00,  1.26it/s, loss=0.00908,
lr=9.97e-5, step=4472]
```

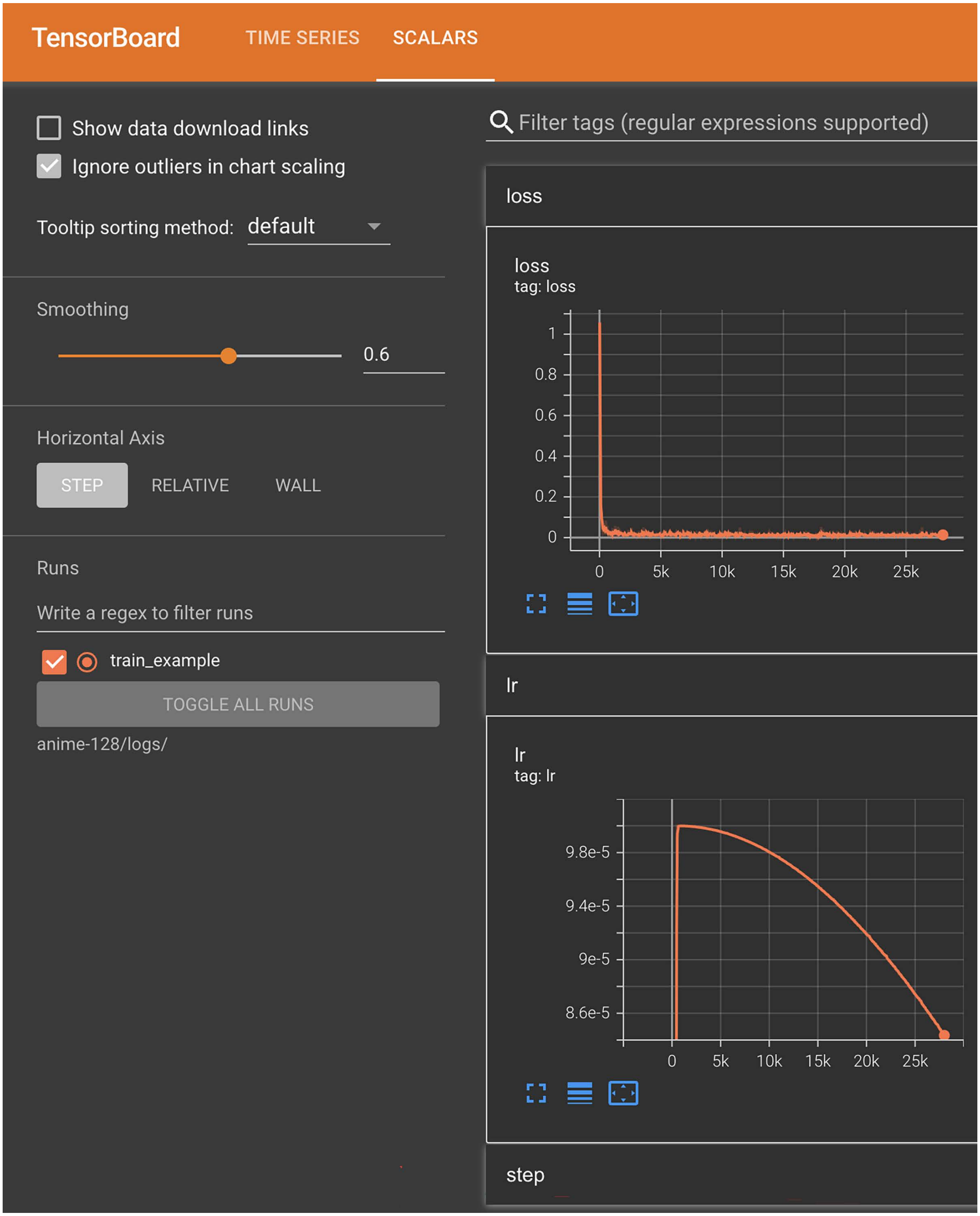
In a terminal window, you can run the following command to launch a TensorBoard session:

```
tensorboard --logdir anime-128/logs/
```

This should produce a terminal output like the following:

```
Serving TensorBoard on localhost; to expose to the network, use a proxy or pass
--bind_all
TensorBoard 2.15.1 at http://localhost:6006/ (Press CTRL+C to quit)
```

Let us open a web browser and check what is happening at `http://localhost:6006`. You should see a TensorBoard page similar to *Figure 10.16*.



□ Show data download links

☑ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0.6

Horizontal Axis: STEP RELATIVE WALL

Runs

Write a regex to filter runs

☑ ● train_example

TOGGLE ALL RUNS

anime-128/logs/

Figure 10.16: TensorBoard page showing the loss and learning rate evolution during the training of a UNet model for the diffusion process

Figure 10.16 confirms the cosine schedule with warmup that we are using for our learning rate because the learning rate first increases to the specified value (`1e-4`) and then decreases to `0` in a *cosinusoidal* manner. Also, the training loss seems to be going down severely, which indicates that the UNet model is indeed learning how to predict the noise signal from a noisy image. This concludes our discussion of the UNet model training loop.

The remaining piece of the diffusion puzzle is to be able to use the UNet model to generate realistic (anime) images from pure noise. In the next section, we'll use the `diffusers` library to perform that final step.

Generating realistic anime images using (reverse) diffusion

The `diffusers` library provides the `DDPMPipeline` API, which helps build the denoising pipeline we need (as shown in *Figure 10.10*) using the trained UNet model. Within the model training loop, at the end of each epoch, we run the following lines of code, to use the trained UNet model (so far) to generate realistic anime images:

```
RANDOM_SEED = 42
pipeline = DDPMPipeline(unet=accelerator.unwrap_model(model), scheduler=noise_
scheduler)
if (epoch + 1) % SAVE_ARTIFACT_EPOCHS == 0 or epoch == NUM_EPOCHS - 1:
    images = pipeline(
        batch_size=BSIZE,
        generator=torch.manual_seed(RANDOM_SEED),
    ).images
    # Make a grid out of the images
    image_grid = make_image_grid(images, rows=4, cols=4)
    # Save the images
    test_dir = os.path.join(MODEL_SAVE_DIR, "samples")
    os.makedirs(test_dir, exist_ok=True)
    image_grid.save(f"{test_dir}/{epoch:04d}.png")
    pipeline.save_pretrained(MODEL_SAVE_DIR)
```

The above code uses the UNet model to iteratively detect and subtract noise from an initial extremely noisy image, to generate a realistic anime image. We save a batch of `16` such generated images with a fixed random seed used to generate noise at the end of each epoch. The fixed seed ensures that we generate the same 16 images with each epoch so that we can compare the DDPM performance across epochs. *Figure 10.17* shows the evolution of our UNet and DDPM training pipeline:

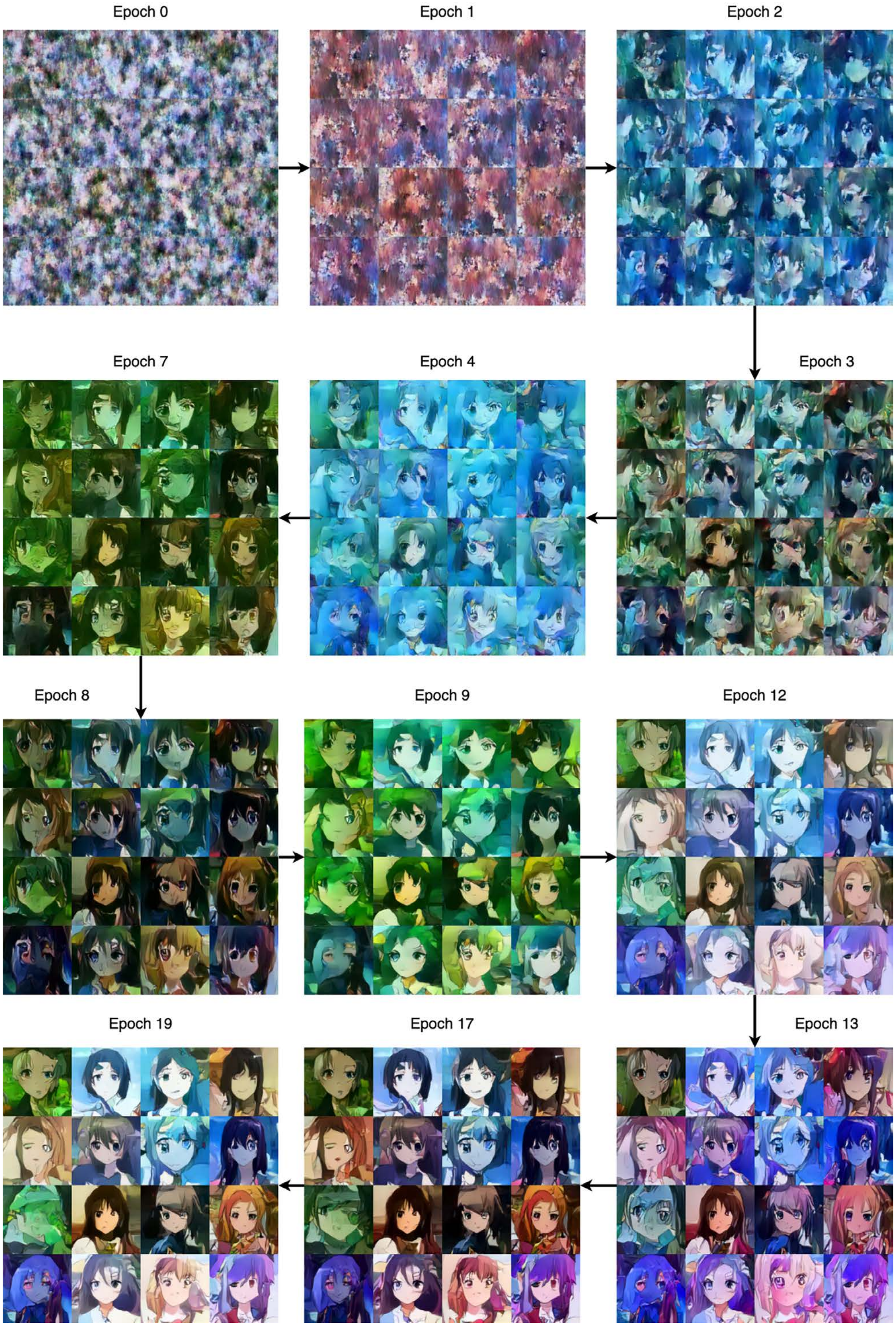


Figure 10.17: Epoch-wise progress of the DDPM pipeline in generating realistic anime images

At epoch 0, we essentially have noise, and after 20 epochs the DDPM pipeline is producing decent anime-like images. Notice the similarity between *Figure 10.17* and *Figure 10.3*. Congratulations – you have successfully built a diffusion-based generative AI model to generate (anime) images!

In the next section, we'll add text to the diffusion mix. We'll learn how to guide the image generation process by providing a text description of the desired image. Instead of generating realistic images from pure noise, we'll generate images from pure noise plus text.

Understanding text-to-image generation using diffusion

Recall *Figure 10.8*, where we demonstrated the training process of the UNet model for generating images using diffusion. We trained the UNet model to learn noise from an input noisy image. To facilitate text-to-image generation, we need to add text as an additional input to this UNet model, as demonstrated in *Figure 10.18* (in contrast to *Figure 10.8*):

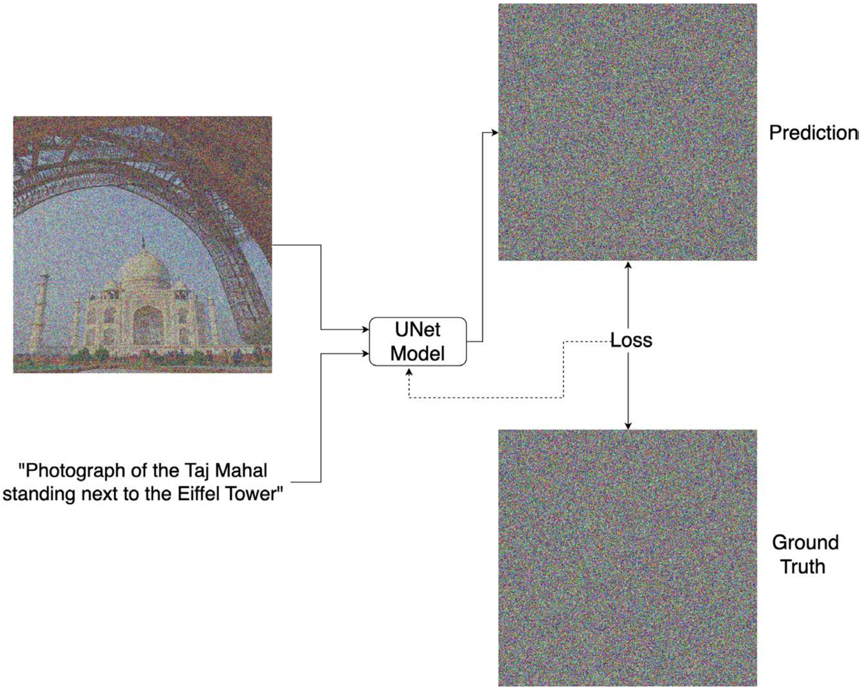

Prediction
UNet Model
Loss
"Photograph of the Taj Mahal standing next to the Eiffel Tower"
Ground Truth

Figure 10.18: UNet trained on both an input (noisy) image as well as text to predict the noise within the noisy image

Such a UNet model is called a **conditional UNet** model [11], or a text-conditional UNet model to be precise, as this model generates an image conditioned on the input text. So, how do we train such a model?

There are two parts to the answer to this question. We first need to encode the input text into an embedding vector that can be ingested into the UNet model. Then we need to modify the UNet model slightly to accommodate the extra incoming data (besides the image) in the form of embedded input text. Let us explore the text encoding first.

Encoding text input into an embedding vector

We need a separate model that takes in input text and outputs an n-dimensional vector for that text. At the same time, we want this vector to represent what this text is describing visually. This is important because, in the end, we use these vectors for the conditional generation of images inside the UNet model. A model that can provide such embeddings is the **CLIP** (**Contrastive Language-Image Pre-Training**) model [12]. This model is trained on a large number of images from the web along with their captions. The model contains two components – an image encoder and a text encoder. We are interested in the latter. *Figure 10.19* shows how the CLIP model is trained:

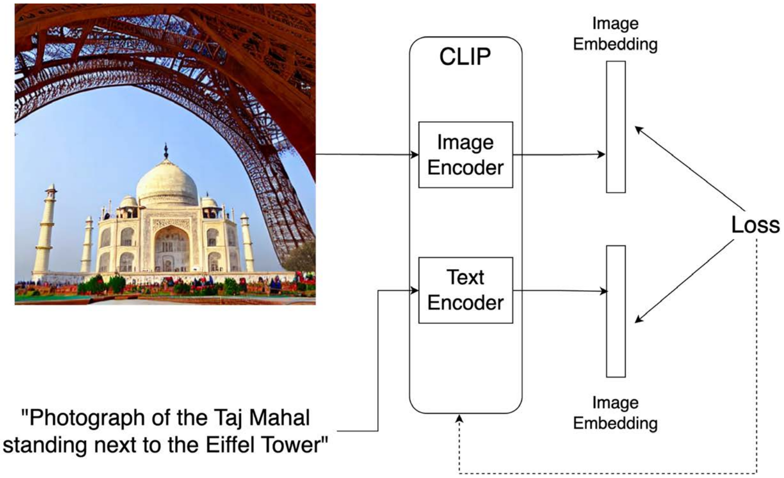


Image Embedding
CLIP
Image Encoder
Text Encoder
Loss
Image Embedding
"Photograph of the Taj Mahal standing next to the Eiffel Tower"

Figure 10.19: High-level schematic showing the training process of the CLIP model

Pairs of images and captions are fed to two different encoders to produce two different embeddings. These encoders are then trained to produce similar embeddings for the given image-caption pair(s). This results in a text encoder that captures the visual meaning behind any given text. This is a great candidate to be used for encoding text before feeding it to our UNet model.

Now that we have solved the text encoding problem, let us understand how the UNet model adapts to this additional data.

Ingesting additional text data in the (conditional) UNet model

While the traditional UNet model takes an image as input and produces an output image of the same size as the input, the conditional UNet model takes in additional text input and uses that text input alongside the image input to produce an output image (of the same size as the input image). So, what changes in a conditional UNet model?

In a UNet model, we have a series of downsampling convolutional layers followed by upsampling convolutional layers with residual connections between the downsampling and upsampling components, as seen both in *Figure 9.8* of *Chapter 9*, *Deep Convolutional GANs*, and in the UNet model definition code under the *Defining the UNet model* section.

In a conditional UNet model, *attention layers* are added between the existing convolutional layers to process the input text (embedding) and to allow the UNet to learn the correlations between the output pixels produced by it and the incoming text embedding vector. We have learned about the attention mechanism and the attention layer in *Chapter 4*, *Deep Recurrent Model Architectures*, and *Chapter 5*, *Hybrid Advanced Models*. *Figure 10.20* shows what a conditional UNet looks like, at a high level, after the addition of the text input and the attention layers.

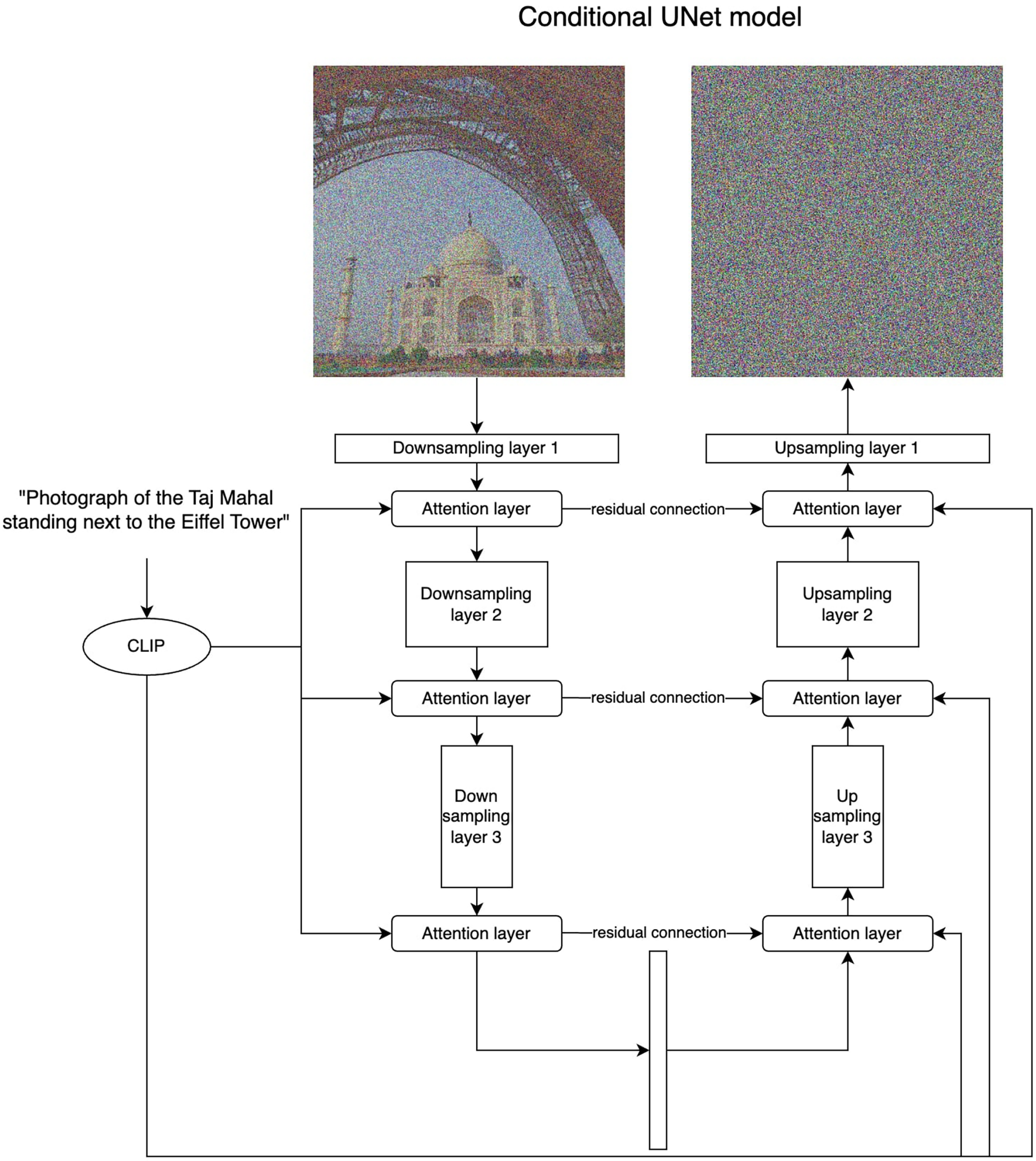


Figure 10.20: Conditional UNet model that takes in CLIP encodings of a text input alongside the original image input to produce an output of the same size as the input image

Once we train a conditional UNet model, the rest of the image generation process is the same as shown in *Figure 10.10*, using DDPM. The only difference is that we use a conditional UNet as opposed to a regular UNet and we pass both the text as well as the image as inputs to the model iteratively across timesteps, as shown in *Figure 10.21*.

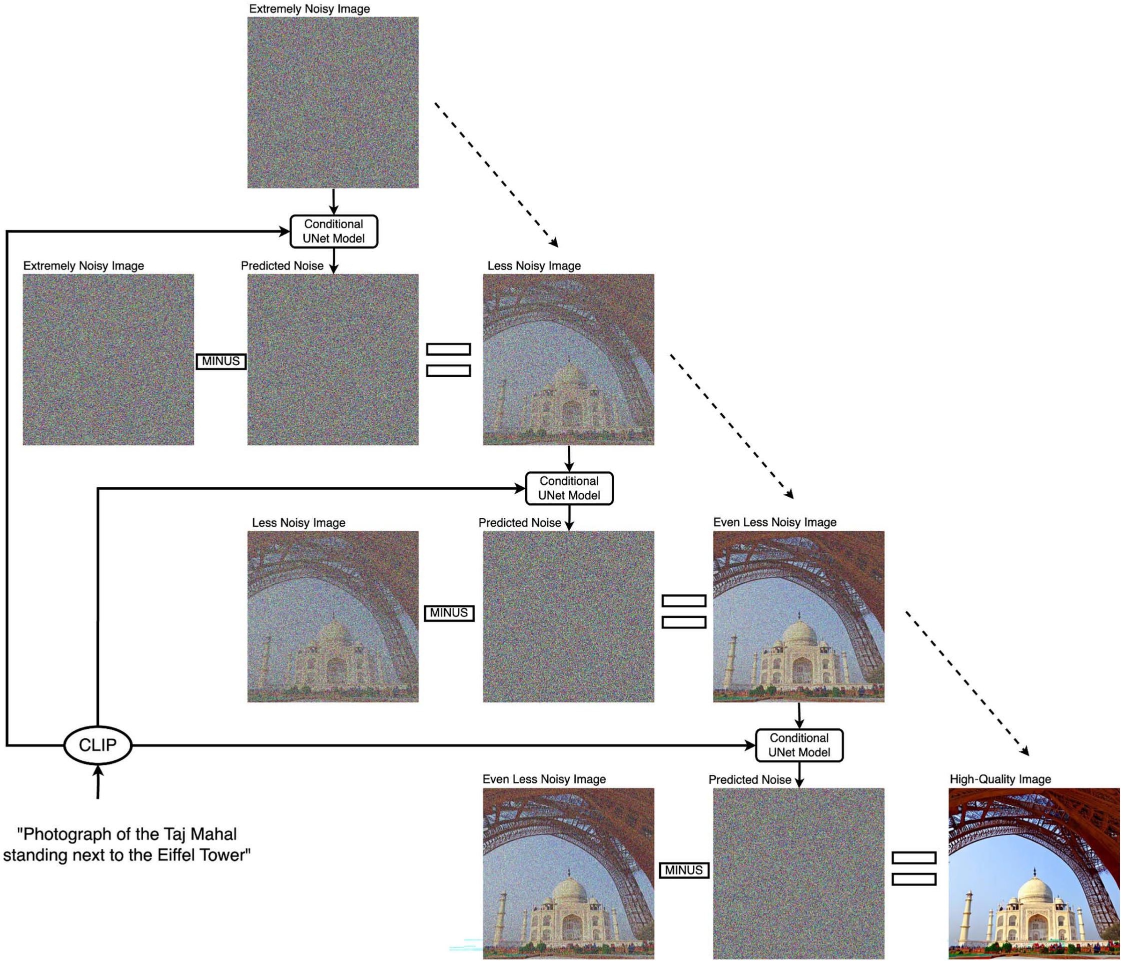


Figure 10.21: Text-to-image generation using the DDPM pipeline on a conditional UNet model

Congratulations! You now understand how **Stable Diffusion** [13] works (more or less). This text-to-image generation process is a fundamental building block of most generative AI models in the computer vision space. With this knowledge, you will be easily able to understand the fine details of the inner workings of **DALL-E** [14], **Imagen** [15], **Midjourney** [16], and so on.

This concludes our discussion on text-to-image generation using diffusion. In the next section, we'll put this text-to-image DDPM pipeline into action. We'll generate some realistic images using a pre-trained Stable Diffusion model, with the help of the `diffusers` library from Hugging Face.

Using the Stable Diffusion model to generate images from text

The `diffusers` library provides several pre-trained diffusion-based text-to-image generation models. One such model is **Stable Diffusion V1.5**. In this section, we'll use this model to generate a high-quality image with a few lines of code. All the code for this section is available on GitHub [17].

First, we load the Stable Diffusion model with the following lines of code:

```
from diffusers import AutoPipelineForText2Image
import torch
pipeline = AutoPipelineForText2Image.from_pretrained(
    "runwayml/stable-diffusion-v1-5", torch_dtype=torch.float16, variant="fp16"
)
pipeline = pipeline.to("cuda")
```

The above code defines a DDPM text-to-image pipeline. You can access the underlying conditional UNet model with the following line of code:

```
pipeline.unet
```

This should produce the following output:

```
UNet2DConditionModel(
  (conv_in): Conv2d(4, 320, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  ...
  (down_blocks): ModuleList(
    (0): CrossAttnDownBlock2D(
      (attentions): ModuleList(
...
(conv_norm_out): GroupNorm(32, 320, eps=1e-05, affine=True)
  (conv_act): SiLU()
  (conv_out): Conv2d(320, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
)
```

The above UNet model is the conditional UNet model that we roughly outlined in *Figure 10.20*. We'll now use the DDPM pipeline to produce a high-quality, realistic image using an input text (also referred to as a *prompt* in generative AI jargon):

```
generator = [torch.Generator(device="cuda").manual_seed(42)]
image = pipeline(
    "Fictional photograph of Taj Mahal on Mars", generator=generator
).images[0]
image
```

This should produce an output like that shown in *Figure 10.22*.

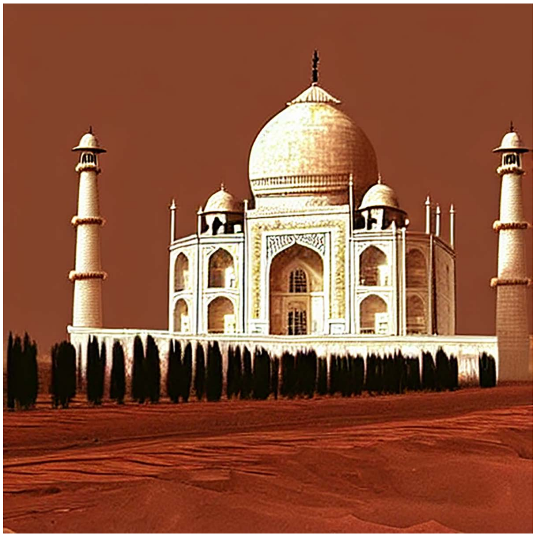


Figure 10.22: Hypothetical image of the Taj Mahal on Mars generated by the Stable Diffusion model. The image is generated with a fixed seed and is hence reproducible

Notice how the model inherently knows about the reddish surface and atmosphere on Mars while remembering the fine details of the Taj Mahal. Also, note that we have used a fixed seed to generate this image, to keep this image generation process reproducible. You can also remove the generator to observe different generated images on different runs. *Figure 10.23* shows an example of generating an image, without a fixed-seed generator, of "the Taj Mahal on the Moon with Earth in sight."

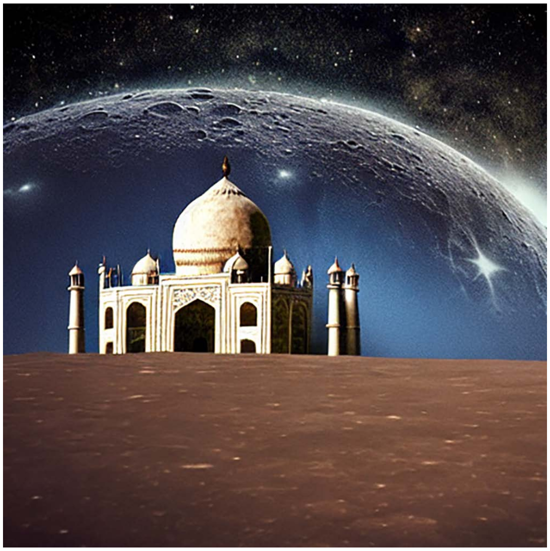


Figure 10.23: Hypothetical image of the Taj Mahal on the Moon with Earth in sight, generated by the Stable Diffusion model, without a fixed seed

This concludes our exploration of the generative AI models used for generating high-quality, surreal images using the simple process of noise diffusion. After this chapter, you should be able to understand the inner workings of cutting-edge diffusion models and train and run custom diffusion models to generate beautiful images, either from text or from pure noise.

In the next chapter, we will change tracks and discuss one of the most exciting and upcoming areas of deep learning – deep reinforcement learning. This branch of deep learning is still maturing. We will explore what PyTorch already has to offer and how it is helping to further developments in this challenging field of deep learning.

Summary

In this chapter, we first learned how diffusion works for image generation. We then trained a custom diffusion model using PyTorch and Hugging Face on a dataset of anime images. Next, we extended our understanding of diffusion models by learning about text-to-image generation. Finally, we used the Stable Diffusion V1.5 model available from Hugging Face to generate high-quality images from text.

Reference list

1. GitHub 1: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter10/taj.png`
2. *Denoising Diffusion Probabilistic Models*: `https://arxiv.org/abs/2006.11239`
3. Hugging Face Diffusers library: `https://huggingface.co/docs/diffusers/index`
4. Hugging Face Accelerate library: `https://huggingface.co/docs/accelerate/index`
5. GitHub 2: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter10/image_generation_using_diffusion.ipynb`
6. Hugging Face huggan/selfie2anime: `https://huggingface.co/datasets/huggan/selfie2anime`
7. Hugging Face DDPMScheduler: `https://huggingface.co/docs/diffusers/api/schedulers/ddpm`
8. Cosine schedule with warmup: `https://huggingface.co/docs/transformers/main_classes/optimizer_schedules#transformers.get_constant_schedule_with_warmup`
9. Defining the model optimizer as AdamW: `https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html`
10. Hugging Face conditional UNet model: `https://huggingface.co/docs/diffusers/api/models/unet2d-cond`
11. CLIP model: `https://openai.com/research/clip`
12. Stable Diffusion: `https://stability.ai/news/stable-diffusion-public-release`
13. DALL-E 3: `https://openai.com/dall-e-3`
14. Imagen 2: `https://deepmind.google/technologies/imagen-2/`
15. Midjourney: `https://mid-journey.ai/midjourney-v6-release/`
16. GitHub 3: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter10/text_to_image_generation_using_stable_diffusion_v1_5.ipynb`

Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below to get a free eBook of your choice.

11

Deep Reinforcement Learning

Machine learning is usually classified into different paradigms, such as **supervised learning**, **unsupervised learning**, semi-supervised, self-supervised learning, and **reinforcement learning** (**RL**). Supervised learning requires labeled data and is currently the most popularly used machine learning paradigm. However, applications based on unsupervised and semi-supervised learning, which require few or no labels, have been steadily on the rise, especially in the form of generative models. Better still, the rise of **Large Language Models** (**LLMs**) have shown that self-supervised learning (where labels are implicit within the data) is an even more promising machine learning paradigm.

RL, on the other hand, is a different branch of machine learning that is considered to be the closest we have reached in terms of emulating how humans learn. It is an area of active research and development and is in its early stages, with some promising results. A prominent example is the famous AlphaGo model, built by Google's DeepMind, which defeated the world's best Go player.

In supervised learning, we usually feed the model with atomic input-output data pairs and hope for the model to learn the output as a function of the input. In RL, we are not keen on learning such individual input to individual output functions. Instead, we are interested in learning a strategy (or policy) that enables us to take a sequence of steps (or actions), starting from the input (state), in order to obtain the final output or achieve the final goal.

Looking at a photo and deciding whether it's a cat or a dog is an atomic input-output learning task that can be solved through supervised learning. However, looking at a chess board and deciding the next move with the aim of winning the game requires strategy, and we need RL for such tasks.

In the previous chapters, we came across examples of supervised learning such as building a classifier to classify handwritten digits using the MNIST dataset. We also explored unsupervised learning while building a text generation model using an unlabeled text corpus.

In this chapter, we will uncover some of the basic concepts of RL and **deep reinforcement learning** (**DRL**). We will then focus on a specific and popular type of DRL model – the **deep Q-learning network** (**DQN**) model. Using PyTorch, we will build a DRL application. We will train a DQN model to learn how to play the game of Pong against a computer opponent (otherwise known as a bot).

By the end of this chapter, you will have all the necessary context to start working on your own DRL project in PyTorch. Additionally, you will have hands-on experience in building a DQN model for a real-life problem. The skills you'll have gained in this chapter will be useful for working on other such RL problems.

This chapter is broken down into the following topics:

- Reviewing RL concepts
- Discussing Q-learning
- Understanding deep Q-learning
- Building a DQN model in PyTorch

All the code files for this chapter can be found at `https://github.com/arj7192/MasteringPyTorchV2/tree/main/Chapter11`.

Reviewing RL concepts

In a way, RL can be defined as learning from rewards. Instead of getting the feedback for every data instance, as is the case with supervised learning, the feedback is received after a sequence of actions. *Figure 11.1* shows the high-level schematic of an RL system:

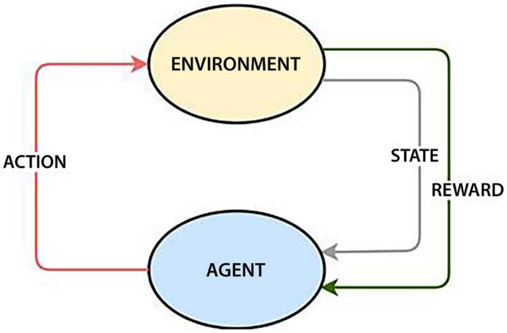


Figure 11.1: RL schematic

In an RL setting, we usually have an **agent**, which does the learning. The agent learns to make decisions and take **actions** according to these decisions. The agent operates within a provided **environment**. This environment can be thought of as a confined world where the agent lives, takes actions, and learns from its actions. An action here is simply the implementation of the decision the agent makes based on what it has learned.

We mentioned earlier that unlike supervised learning, RL does not have an output for each and every input; that is, the agent does not necessarily receive explicit feedback for each and every action. Instead, the agent works in **states and at best keeps count of the number of states traversed until the end goal (reward)**. Suppose it starts at an initial state, S_0. It then takes an action, say a_0.

This action transitions the state of the agent from S_0 to S_1, after which the agent takes another action, a_1, and the cycle goes on, as shown in *Figure 11.2*.

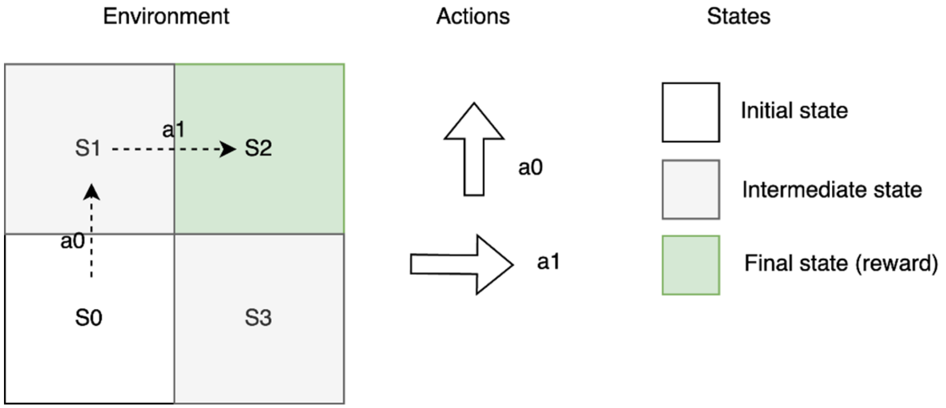


Figure 11.2: Example of an RL environment, states, and actions

Occasionally, the agent receives **rewards** based on its state. The sequence of states and actions that the agent traverses is also known as a **trajectory**. Let's say the agent received a reward at state S_2. In that case, the trajectory that resulted in this reward would be S_0, a_0, S_1, a_1, S_2.

Note

The rewards can either be positive or negative.

Based on the rewards, the agent learns to adjust its behavior so that it takes actions in a way that maximizes the long-term rewards. This is the essence of RL. The agent learns a strategy regarding how to act optimally (that is, to maximize the reward) based on the given state and reward.

Video games are one of the best examples to demonstrate RL. Let's use the video game Pong as an example, which is a virtual version of table tennis. The following is a snapshot of this game:

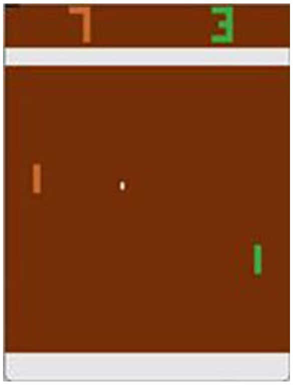


Figure 11.3: Pong video game

Consider that the player to the right is the agent, which is represented by a short vertical line. Notice that there is a well-defined environment here. The environment consists of the playing area, which is denoted by the brown pixels. The environment also consists of a ball, which is denoted by a white pixel. As well as this, the environment consists of the boundaries of the playing area, denoted by the gray stripes and edges that the ball may bounce off of. Finally, and most importantly, the environment includes an opponent, which looks like the agent but is placed on the left-hand side, opposite the agent.

Often, but not always, in an RL setting, the agent at any given state has a finite set of possible actions, referred to as a discrete action space (as opposed to a continuous action space). In this example, the agent has two possible actions at all states – move up or move down – but with two exceptions. First, it can only move down when it is at the top-most position (state), and second, it can only move up when it is at the bottom-most position (state).

The concept of reward in this case can be directly mapped to what happens in an actual table tennis game. If you miss the ball, your opponent gains a point. Whoever scores 21 points first wins the game and receives a positive reward. Losing a game means negative rewards. Scoring a point or losing a point also results in smaller intermediate positive and negative rewards, respectively. A sequence of play starting from score 0-0 and leading to either of the players scoring 21 points is called an **episode**.

Training our agent for a Pong game using RL is equivalent to training someone to play table tennis from scratch. Training results in a **policy** that the agent follows while playing the game. In any given situation – which includes the position of the ball, the position of the opponent, the scoreboard, as well as the previous reward – a successfully trained agent moves up or down to maximize its chances of winning the game.

So far, we have discussed the basic concepts behind RL by providing an example. In doing so, we have repeatedly mentioned terms such as strategy, policy, and learning. But how does the agent actually learn the policy? The answer is through an RL model, which works based on a pre-defined algorithm. Next, we will explore the different kinds of RL algorithms.

Types of RL algorithms

In this section, we will look at the types of RL algorithms, as per the literature. We will then explore some of the subtypes within these types. Broadly speaking, RL algorithms can be categorized as either of the following:

- Model-based
- Model-free

Let’s look at them one by one.

Model-based

As the name suggests, in model-based algorithms, the agent knows about the model of the environment. The model here refers to the mathematical formulation of a function that can be used to estimate rewards and how the states transition within the environment. Because the agent has some idea about the environment, it helps reduce the sample space to choose the next action from. This helps with the efficiency of the learning process.

However, in reality, a modeled environment is not directly available most of the time. If we, nonetheless, want to use the model-based approach, we need to have the agent learn the environment model with its own experience. In such cases, the agent is highly likely to learn a biased representation of the model and perform poorly in the real environment. For this reason, model-based approaches are less frequently used for implementing RL systems. We will not be discussing models based on this approach in detail in this book, but here are some examples:

- **Model-Based DRL with Model-Free Fine-Tuning** (**MBMF**)
- **Model-Based Value Estimation** (**MBVE**) for efficient Model-Free RL
- **Imagination-Augmented Agents** (**I2A**) for DRL
- **AlphaZero**, the famous AI bot that defeated Chess and Go champions

Now, let's look at the other set of RL algorithms that work with a different philosophy.

Model-Free

The Model-Free approach works without any model of the environment and is currently more popularly used for RL research and development. There are primarily two ways of training the agent in a Model-Free RL setting:

- Policy optimization
- Q-learning

Policy optimization

In this method, we formulate the policy in the form of a function of an action, given the current state, as demonstrated in the following equation:

$$Policy = F_{\beta}\ (a \mid S)$$

Equation 11.1

Here, β represents the internal parameters of this function, which is updated to optimize the policy function via gradient ascent. The objective function is defined using the policy function and the rewards. An approximation of the objective function may also be used in some cases for the optimization process. Furthermore, in some cases, an approximation of the policy function could be used instead of the actual policy function for the optimization process.

Usually, the optimizations that are performed under this approach are **on-policy**, which means that the parameters are updated based on the data gathered using the latest policy version. Some examples of policy optimization-based RL algorithms are as follows:

- **Policy gradient**: This is the most basic policy optimization method where we directly optimize the policy function using gradient ascent. The policy function outputs the probabilities of different actions to be taken next, at each time step.
- **Actor-critic**: Because of the on-policy nature of optimization under the policy gradient algorithm, every iteration of the algorithm needs the policy to be updated. This takes a lot of time. The actor-critic method introduces the use of a value function, as well as a policy function. The actor models the policy function and the critic models the value function.

By using a critic, the policy update process becomes faster. We will discuss the value function in more detail in the next section. However, we will not go into the mathematical details of the actor-critic method in this book.

- **Trust region policy optimization** (**TRPO**): Like the policy gradient method, TRPO consists of an on-policy optimization approach. In the policy-gradient approach, we use the gradient for updating the policy function parameters, β . Since the gradient is a first-order derivative, it can be noisy for sharp curvatures in the function. This may lead us to make large policy changes that may destabilize the learning trajectory of the agent.

  To prevent that, TRPO proposes a trust region. It defines an upper limit on how much the policy may change in a given update step. This ensures the stability of the optimization process.

- **Proximal policy optimization** (**PPO**): Similar to TRPO, PPO aims to stabilize the optimization process. During gradient ascent, an update is performed per data sample using the policy gradient approach. PPO, however, uses a surrogate objective function, which facilitates updates over batches of data samples. This results in estimating gradients more conservatively, thereby improving the chances of the gradient ascent algorithm converging.

Policy optimization functions directly work on optimizing the policy and hence are extremely intuitive algorithms. However, due to the on-policy nature of most of these algorithms, data needs to be resampled at each step after the policy is updated. This can be a limiting factor in solving RL problems. Next, we will discuss the other kind of Model-Free algorithm that is more sample-efficient, known as Q-learning.

Q-learning

Contrary to policy optimization algorithms, **Q-learning** relies on a value function instead of a policy function. From here on, this chapter will focus on Q-learning. We will explore the fundamentals of Q-learning in detail in the next section.

Discussing Q-learning

The key difference between policy optimization and Q-learning is the fact that in the latter, we are not directly optimizing the policy. Instead, we optimize a value function. What is a **value function**? We have already learned that RL is all about an agent learning to gain the maximum overall rewards while traversing a trajectory of states and actions. A value function is a function of a given state the agent is currently at, and this function outputs the expected sum of rewards the agent will receive by the end of the current episode.

In Q-learning, we optimize a specific type of value function, known as the **action-value function**, which depends on both the current state and the action. At a given state, S, the action-value function determines the long-term rewards (rewards until the end of the episode) the agent will receive for taking action a. This function is usually expressed as $Q(S, a)$, and hence is also called the Q-function. The action value is also referred to as the **Q-value**.

The Q-values for every (state, action) pair can be stored in a table where the two dimensions are state and action. For example, if there are four possible states, S_1, S_2, S_3, and S_4, and two possible actions, a_1 and a_2, then the eight Q-values will be stored in a 4x2 table. The goal of Q-learning, therefore, is to create this table of Q-values. Once the table is available, the agent can look up the Q-values for all possible actions from the given state and take the action with the maximum Q-value. However, the question is, where do we get the Q-values from? The answer lies in the **Bellman equation**, which is mathematically expressed as follows:

$$Q(S_t, a_t) = R + \gamma * Q\ (S_{t+1}, a_{t+1})$$

Equation 11.2

The Bellman equation is a recursive way of calculating Q-values. R in this equation is the reward received by taking action a_t at state S_t, while γ (gamma) is the **discount factor**, which is a scalar value between 0 and 1. Basically, this equation states that the Q-value for the current state, S_t, and action, a_t, is equal to the reward, R, received by taking action a_t at state S_t, plus the Q-value resulting from the most optimal action, a_{t+1}, taken from the next state, S_{t+1}, multiplied by a discount factor. The discount factor defines how much weightage is to be given to the immediate reward versus the long-term future rewards.

Now that we have defined most of the underlying concepts of Q-learning, let's walk through an example to demonstrate how Q-learning exactly works. The following diagram shows an environment that consists of five possible states:

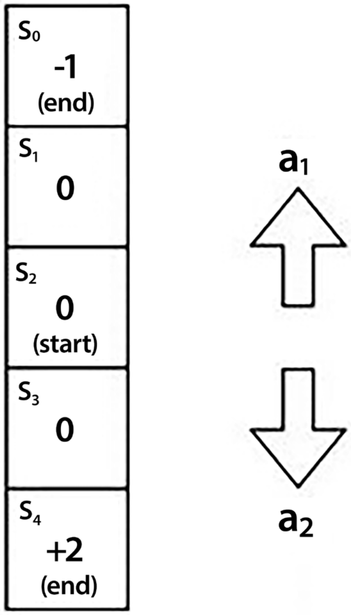


Figure 11.4: Q-learning example environment

There are two different possible actions – moving up ($\mathbf{a_1}$) or down ($\mathbf{a_2}$). There are different rewards at different states ranging from **+2** at state $\mathbf{S_4}$ to **-1** at state $\mathbf{S_0}$. Every episode in this environment starts from state $\mathbf{S_2}$ and ends at either $\mathbf{S_0}$ or $\mathbf{S_4}$. Because there are five states and two possible actions, the Q-values can be stored in a 5x2 table. The following code snippet shows how rewards and Q-values can be written in Python:

```
rwrds = [-1, 0, 0, 0, 2]
Qvals = [[0.0, 0.0],
         [0.0, 0.0],
         [0.0, 0.0],
         [0.0, 0.0],
         [0.0, 0.0]]
```

We initialize all the Q-values to `0`. Also, because there are two specific end states, we need to specify them in the form of a list, as shown here:

```
end_states = [1, 0, 0, 0, 1]
```

This basically indicates that states $\mathbf{S_0}$ and $\mathbf{S_4}$ are end states. There is one final piece we need to look at before we can run the complete Q-learning loop. At each step of Q-learning, the agent has two options with regard to taking the next action:

- Take the action that has the highest Q-value.
- Randomly choose the next action.

Why would the agent choose an action randomly?

Remember that in *Chapter 7*, *Music and Text Generation with PyTorch*, in the *Text generation strategies using PyTorch* section, we discussed how greedy search and beam search result in repetitive results, and hence introducing randomness helps in producing better results. Similarly, if the agent always chooses the next action based on Q-values, then it might get stuck choosing an action repeatedly that gives an immediate high reward in the short term. Hence, taking actions randomly once in a while will help the agent get out of such sub-optimal conditions.

Now that we've established that the agent has two possible ways of taking an action at each step, we need to decide which way the agent goes. This is where the **epsilon-greedy-action** mechanism comes into play. *Figure 11.5* shows how it works:

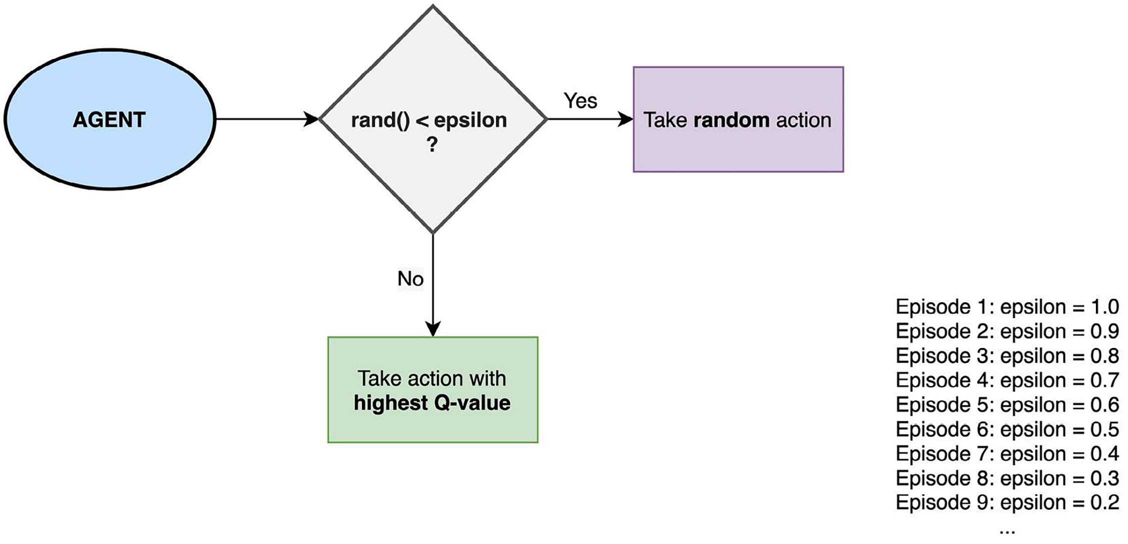


Figure 11.5: Epsilon-greedy-action mechanism

Under this mechanism, at each episode, an epsilon value is pre-decided, which is a scalar value between 0 and 1. In a given episode, for taking each next action, the agent generates a random number between 0 to 1. If the generated number is less than the pre-defined epsilon value, the agent chooses the next action randomly from the available set of next actions. Otherwise, the Q-values for each of the next possible actions are retrieved from the Q-value table, and the action with the highest Q-value is chosen. The Python code for the epsilon-greedy-action mechanism is as follows:

```python
def eps_greedy_action_mechanism(eps, S):
    rnd = np.random.uniform()
    if rnd < eps:
        return np.random.randint(0, 2)
    else:
        return np.argmax(Qvals[S])
```

Typically, we start with an epsilon value of 1 at the first episode and then linearly decrease it as the episodes progress. The idea here is that we want the agent to explore different options initially. However, as the learning process progresses, the agent is less susceptible to getting stuck collecting short-term rewards and hence it can better exploit the Q-values table.

We are now in a position to write the Python code for the main Q-learning loop, which will look as follows:

```python
n_epsds = 100
eps = 1
gamma = 0.9
for e in range(n_epsds):
    S_initial = 2 # start with state S2
    S = S_initial
    while not end_states[S]:
        a = eps_greedy_action_mechanism(eps, S)
        R, S_next = take_action(S, a)
        if end_states[S_next]:
            Qvals[S][a] = R
        else:
            Qvals[S][a] = R + gamma * max(Qvals[S_next])
        S = S_next
    eps = eps - 1/n_epsds
```

First, we define that the agent shall be trained for 100 episodes. We begin with an epsilon value of 1 and we define the discounting factor (gamma) as 0.9. Next, we run the Q-learning loop, which loops over the number of episodes. In each iteration of this loop, we run through an entire episode. Within the episode, we first initialize the state of the agent to S_2.

We then run another internal loop, which only breaks if the agent reaches an end state. Within this internal loop, we decide on the next action for the agent using the epsilon-greedy-action mechanism. The agent then takes the action, which transitions the agent to a new state and may possibly yield a reward. The implementation for the `take_action` function is as follows:

```python
def take_action(S, a):
    if a == 0: # move up
        S_next = S - 1
    else:
        S_next = S + 1
    return rwrds[S_next], S_next
```

Once we obtain the reward and the next state, we update the Q-value for the current state-action pair using *Equation 11.2*. The next state now becomes the current state, and the process repeats. At the end of each episode, the epsilon value is reduced linearly. Once the entire Q-learning loop is over, we obtain a Q-values table. This table is essentially all that the agent needs to operate in this environment in order to gain the maximum long-term rewards.

Ideally, a well-trained agent for this example would always move downward to receive the maximum reward of *+2* at S_4, and would avoid going toward S_0, which contains a negative reward of *-1*.

This completes our discussion on Q-learning. The preceding code should help you get started with Q-learning in simple environments such as the one provided here. For more complex and realistic environments, such as video games, this approach will not work. Why?

We have noticed that the essence of Q-learning lies in creating the Q-values table. In our example, we only had 5 states and 2 actions, and therefore the table was of size 10, which is manageable. But in video games such as Pong, there are far too many possible states. This explodes the Q-values table's size, which makes our Q-learning algorithm extremely memory intensive and impractical to run.

Thankfully, there is a solution where we can still use the concept of Q-learning without having our machines run out of memory. This solution combines the worlds of Q-learning and deep neural networks and provides the extremely popular RL algorithm known as **DQN**. In the next section, we will discuss the basics of DQN and some of its novel characteristics.

Understanding deep Q-learning

Instead of creating a Q-values table, DQN uses a **deep neural network** (**DNN**) that outputs a Q-value for a given state-action pair. DQN is used with complex environments such as video games, where there are far too many states for them to be managed in a Q-values table. The current image frame of the video game is used to represent the current state and is fed as input to the underlying DNN model, together with the current action.

The DNN outputs a scalar Q-value for each such input. In practice, instead of just passing the current image frame, *N* number of neighboring image frames in a given time window are passed as input to the model.

We are using a DNN to solve an RL problem. This has an inherent concern. While working with DNNs, we have always worked with **independent and identically distributed** (**iid**) data samples. However, in RL, every current output impacts the next input. For example, in the case of Q-learning, the Bellman equation itself suggests that the Q-value is dependent on another Q-value; that is, the Q-value of the next state-action pair impacts the Q-value of the current-state pair.

This implies that we are working with a constantly moving target and there is a high correlation between the target and the input. DQN addresses these issues with two novel features:

- Using two separate DNNs
- Experience replay buffer

Let's look at these in more detail.

Using two separate DNNs

Let's rewrite the Bellman equation for DQNs:

$$Q\ (S_t, a_t, \theta) = R + \ \gamma * Q\ (S_{t+1}, a_{t+1}, \theta)$$

Equation 11.3

This equation is mostly the same as for Q-learning except for the introduction of a new term, θ (theta). θ represents the weights of the DNN that the DQN model uses to get Q-values. But something is odd with this equation.

Notice that θ is placed on both the left-hand side and the right-hand side of the equation. This means that at every step, we are using the same neural network to get the Q-values of the current state-action pair as well as the next state-action pair. This means that we are chasing a non-stationary target because every step, θ, will be updated, which will change both the left-hand side as well as the right-hand side of the equation for the next step, causing instability in the learning process.

This can be more clearly seen by looking at the loss function, which the DNN will be trying to minimize using gradient descent. The loss function is as follows:

$$L = E[\left(R + \gamma * Q\ (S_{t+1}, a_{t+1}, \theta) - Q\ (S_t, a_t, \theta)\right)^2]$$

Equation 11.4

This loss is known as temporal difference loss and is one of the foundational concepts of DQNs. Keeping R (reward) aside for a moment, having the exact same network producing Q-values for current and next state-action pairs will lead to volatility in the loss function as both terms will be constantly changing. To address this issue, DQN uses two separate networks – a main DNN and a target DNN. Both DNNs have the exact same architecture.

The main DNN is used to compute the Q-values of the current state-action pair, while the target DNN is used to compute the Q-values of the next (or target) state-action pair. However, although the weights of the main DNN are updated at every learning step, the weights of the target DNN are frozen. After every K gradient descent iteration, the weights of the main network are copied to the target network. This mechanism keeps the training procedure relatively stable. The weights-copying mechanism ensures accurate predictions from the target network.

Experience replay buffer

Because the DNN expects iid data as input, we simply cache the last X number of steps (frames of the video game) into a buffer memory and then randomly sample batches of data from the buffer. These batches are then fed as inputs to the DNN. Because the batches consist of randomly sampled data, the distribution looks similar to that of the iid data samples. This helps stabilize the DNN training process.

> **Note**
>
> Without the buffer trick, the DNN would receive correlated data, which would result in poor optimization results.

These two tricks have proven significant in contributing to the success of DQNs. Now that we have a basic understanding of how DQN models work and their novel characteristics, let's move on to the final section of this chapter, where we will implement our own DQN model. Using PyTorch, we will build a CNN-based DQN model that will learn to play the Atari video game known as Pong and potentially learn to win the game against the computer opponent.

Building a DQN model in PyTorch

We discussed the theory behind DQNs in the previous section. In this section, we will take a hands-on approach. Using PyTorch, we will build a CNN-based DQN model that will train an agent to play the video game known as Pong. The goal of this exercise is to demonstrate how to develop DRL applications using PyTorch. Let's get straight into the exercise.

Initializing the main and target CNN models

In this exercise, we will only show the important parts of the code for demonstration purposes. In order to access the full code, visit our GitHub repository [1]. Follow these steps:

1. First, we need to import the necessary libraries:

```
# general imports
import cv2
import math
import numpy as np
import random
# reinforcement learning related imports
import re
import atari_py as ap
from collections import deque
from gym import make, ObservationWrapper, Wrapper
from gym.spaces import Box
# pytorch imports
import torch
import torch.nn as nn
from torch import save
from torch.optim import Adam
```

   In this exercise, besides the usual Python- and PyTorch-related imports, we are also using a Python library called `gym`. It is a Python library produced by OpenAI [2] that provides a set of tools for building DRL applications. Essentially, importing `gym` does away with the need to write all the scaffolding code for the internals of an RL system. It also consists of built-in environments, including one for the video game Pong, which we will use in this exercise. Because `gym` requires version 3.7 (or lower) of Python, and PyTorch v2 is not compatible with Python 3.7, we will use PyTorch v1.12 for this exercise.

2. After importing the libraries, we must define the CNN architecture for the DQN model. This CNN model essentially takes in the current state input and outputs the probability distribution over all possible actions. The action with the highest probability gets chosen as the next action by the agent. Instead of using a regression model to predict the Q-values for each state-action pair, we cleverly turn this into a classification problem.

The Q-value regression model will have to be run separately for all possible actions, and we will choose the action with the highest predicted Q-value. But using this classification model combines the tasks of calculating Q-values and predicting the best next action:

```
class ConvDQN(nn.Module):
    def __init__(self, ip_sz, tot_num_acts):
        super(ConvDQN, self).__init__()
        self._ip_sz = ip_sz
        self._tot_num_acts = tot_num_acts
        self.cnv1 = nn.Conv2d(ip_sz[0], 32, kernel_size=8, stride=4)
        self.activation = nn.ReLU()
        self.cnv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)
        self.cnv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1)
        self.fc1 = nn.Linear(self.feat_sz, 512)
        self.fc2 = nn.Linear(512, tot_num_acts)
```

As we can see, the model consists of three convolutional layers – `cnv1`, `cnv2`, and `cnv3` – with ReLU activations in between them, followed by two fully connected layers. Now, let's look at what a forward pass through this model entails:

```
    def forward(self, x):
        op = self.cnv1(x)
        op = self.activation(op)
        op = self.cnv2(op)
        op = self.activation(op)
        op = self.cnv3(op)
        op = self.activation(op).view(x.size()[0], -1)
        op = self.fc1(op)
        op = self.activation(op)
        op = self.fc2(op)
        return op
```

The `forward` method simply demonstrates a forward pass by the model, where the input is passed through the convolutional layers, flattened, and finally fed to the fully connected layers. Finally, let's look at the other model methods:

```
    @property
    def feat_sz(self):
        x = torch.zeros(1, *self._ip_sz)
        x = self.cnv1(x)
        x = self.activation(x)
        x = self.cnv2(x)
        x = self.activation(x)
```

```
            x = self.cnv3(x)
            x = self.activation(x)
            return x.view(1, -1).size(1)
        def perf_action(self, stt, eps, dvc):
            if random.random() > eps:
                stt=torch.from_numpy(
                    np.float32(stt)).unsqueeze(0).to(dvc)
                q_val = self.forward(stt)
                act = q_val.max(1)[1].item()
            else:
                act = random.randrange(self._tot_num_acts)
            return act
```

In the preceding code snippet, the `feat_size` method is simply meant to calculate the size of the feature vector after flattening the last convolutional layer output. Finally, the `perf_action` method is the same as the `take_action` method we discussed previously in the *Discussing Q-learning* section.

3. In this step, we define a function that instantiates the main neural network and the target neural network:

```
def models_init(env, dvc):
    mdl = ConvDQN(
        env.observation_space.shape,
        env.action_space.n).to(dvc)
    tgt_mdl = ConvDQN(
        env.observation_space.shape,
        env.action_space.n).to(dvc)
    return mdl, tgt_mdl
```

These two models are instances of the same class and hence share the same architecture. However, they are two separate instances and hence will evolve differently with different sets of weights.

Defining the experience replay buffer

As we discussed in the *Understanding deep Q-learning* section, the experience replay buffer is a significant feature of DQNs. With the help of this buffer, we can store several thousand transitions (frames) of a game and then randomly sample those video frames to train the CNN model. The following is the code for defining the replay buffer:

```
class RepBfr:
    def __init__(self, cap_max):
        self._bfr = deque(maxlen=cap_max)
    def push(self, st, act, rwd, nxt_st, fin):
        self._bfr.append((st, act, rwd, nxt_st, fin))
    def smpl(self, bch_sz):
        idxs = np.random.choice(len(self._bfr), bch_sz, False)
        bch = zip(*[self._bfr[i] for i in idxs])
        st, act, rwd, nxt_st, fin = bch
        return (np.array(st), np.array(act),
                np.array(rwd, dtype=np.float32),
                np.array(nxt_st), np.array(fin, dtype=np.uint8))
    def __len__(self):
        return len(self._bfr)
```

Here, `cap_max` is the defined buffer size; that is, the number of video game state transitions that shall be stored in the buffer. The `smpl` method is used during the CNN training loop to sample the stored transitions and generate batches of training data.

Setting up the environment

So far, we have mostly focused on the neural network side of DQNs. In this section, we will focus on building one of the foundational aspects of an RL problem – the environment. Follow these steps:

1. First, we must define some video game environment initialization-related functions:

   ```
   def gym_to_atari_format(gym_env):
       ...
   def check_atari_env(env):
       ...
   ```

   Using the `gym` library, we have access to a pre-built Pong video game environment. But here, we will augment the environment in a series of steps, which will include downsampling the video game image frames, pushing image frames to the experience replay buffer, converting images into PyTorch tensors, and so on.

2. The following are the defined classes that implement each of the environment control steps:

   ```
   class ClassicControl(Wrapper):
       ...
   class FrameDownSample(ObservationWrapper):
       ...
   class MaxAndSkipEnv(Wrapper):
       ...
   class FireResetEnv(Wrapper):
       ...
   class FrameBuffer(ObservationWrapper):
       ...
   class Image2PyTorch(ObservationWrapper):
       ...
   ```

```
class NormalizeFloats(ObservationWrapper):
    ...
```

These classes will now be used to initialize and augment the video game environment.

3. Once the environment-related classes have been defined, we must define a final method that takes in the raw Pong video game environment as input and augments the environment, as follows:

```
def wrap_env(env_ip):
    env = make(env_ip)
    is_atari = check_atari_env(env_ip)
    env = ClassicControl(env, is_atari)
    env = MaxAndSkipEnv(env, is_atari)
    try:
        env_acts = env.unwrapped.get_action_meanings()
        if "FIRE" in env_acts:
            env = FireResetEnv(env)
    except AttributeError:
        pass
    env = FrameDownSample(env)
    env = Image2PyTorch(env)
    env = FrameBuffer(env, 4)
    env = NormalizeFloats(env)
    return env
```

Some of the code in this step has been omitted as our focus is on the PyTorch aspect of this exercise. Please refer to this book's GitHub repository [1] for the full code.

Defining the CNN optimization function

In this section, we will define the loss function for training our DRL model, as well as define what needs to be done at the end of each model training iteration. Follow these steps:

1. We initialized our main and target CNN models in *step 3* of the *Initializing the main and target CNN models* section. Now that we have defined the model architecture, we shall define the `loss` function, which the model will be trained to minimize:

```
def calc_temp_diff_loss(mdl, tgt_mdl, bch, gm, dvc):
    st, act, rwd, nxt_st, fin = bch
    st = torch.from_numpy(np.float32(st)).to(dvc)
    nxt_st = torch.from_numpy(np.float32(nxt_st)).to(dvc)
    act = torch.from_numpy(act).to(dvc)
    rwd = torch.from_numpy(rwd).to(dvc)
    fin = torch.from_numpy(fin).to(dvc)
    q_vals = mdl(st)
    nxt_q_vals = tgt_mdl(nxt_st)
    q_val = q_vals.gather(1, act.unsqueeze(-1)).squeeze(-1)
    nxt_q_val = nxt_q_vals.max(1)[0]
    exp_q_val = rwd + gm * nxt_q_val * (1 - fin)
    loss = (q_val -exp_q_val.data.to(dvc)).pow(2).mean()
    loss.backward()
```

The loss function (temporal difference loss) defined here is derived from *Equation 11.4*.

2. Now that the neural network architecture and loss function are in place, we shall define the model `update` function, which is called at every iteration of neural network training:

```
def upd_grph(mdl, tgt_mdl, opt, rpl_bfr, dvc, log):
    if len(rpl_bfr) > INIT_LEARN:
        if not log.idx % TGT_UPD_FRQ:
            tgt_mdl.load_state_dict(mdl.state_dict())
        opt.zero_grad()
        bch = rpl_bfr.smpl(B_S)
        calc_temp_diff_loss(mdl, tgt_mdl, bch, G, dvc)
        opt.step()
```

This function samples a batch of data from the experience replay buffer, computes the time difference loss on this batch of data, and also copies the weights of the main neural network to the target neural network once every `TGT_UPD_FRQ` iterations. `TGT_UPD_FRQ` will be assigned a value later.

Managing and running episodes

Now, let's learn how to define the epsilon value:

1. First, we will define a function that will update the epsilon value after each episode:

```
def upd_eps(epd):
    last_eps = EPS_FINL
    first_eps = EPS_STRT
    eps_decay = EPS_DECAY
    eps = last_eps + (first_eps - last_eps) * math.exp(
        -1 * ((epd + 1) / eps_decay))
    return eps
```

This function is the same as the epsilon update step in our Q-learning loop, as discussed in the *Discussing Q-learning* section. The goal of this function is to linearly reduce the epsilon value per episode.

2. The next function is to define what happens at the end of an episode. If the overall reward that's scored in the current episode is the best we've achieved so far, we save the CNN model weights and print the reward value:

```
def fin_epsd(mdl, env, log, epd_rwd, epd, eps):
    bst_so_far = log.upd_rwds(epd_rwd)
    if bst_so_far:
        print(f"checkpointing current model weights.
              highest running_average_reward of\
              {round(log.bst_avg, 3)} achieved!")
        save(mdl.state_dict(), f"{env}.dat")
    print(f"episode_num {epd}, curr_reward: {epd_rwd}, best_reward:
{log.bst_rwd},\running_avg_reward: {round(log.avg, 3)}, curr_epsilon:
{round(eps, 4)}")
```

At the end of each episode, we also log the episode number, the reward at the end of the current episode, a running average of reward values across the past few episodes, and finally, the current epsilon value.

3. We have finally reached one of the most crucial function definitions of this exercise. Here, we must specify the DQN loop. This is where we define the steps that shall be executed in an episode:

```
def run_epsd(
    env, mdl, tgt_mdl, opt, rpl_bfr, dvc, log, epd):
    epd_rwd = 0.0
    st = env.reset()
    while True:
        eps = upd_eps(log.idx)
        act = mdl.perf_action(st, eps, dvc)
        env.render()
        nxt_st, rwd, fin, _ = env.step(act)
        rpl_bfr.push(st, act, rwd, nxt_st, fin)
        st = nxt_st
        epd_rwd += rwd
        log.upd_idx()
        upd_grph(mdl, tgt_mdl, opt, rpl_bfr, dvc, log)
        if fin:
            fin_epsd(mdl, ENV, log, epd_rwd, epd, eps)
            break
```

The rewards and states are reset at the beginning of the episode. Then, we run an endless loop that only breaks if the agent reaches one of the end states. Within this loop, in each iteration, the following steps are executed:

i. First, the epsilon value is modified as per the *linear depreciation scheme*.

ii. The next action is predicted by the main CNN model. This action is executed, resulting in the next state and a reward. This state transition is recorded in the experience replay buffer.

   iii. The next state now becomes the current state, and we calculate the time difference loss, which is used to update the main CNN model while keeping the target CNN model frozen.

   iv. If the new current state is an end state, then we break the loop (that is, end the episode) and log the results for this episode.

4. We have mentioned logging results throughout the training process. To store the various metrics around rewards and model performance, we must define a training metadata class, which will consist of various metrics as attributes:

```
class TrMetadata:
    def __init__(self):
        self._avg = 0.0
        self._bst_rwd = -float("inf")
        self._bst_avg = -float("inf")
        self._rwds = []
        self._avg_rng = 100
        self._idx = 0
```

   We will use these metrics to visualize model performance later in this exercise once we've trained the model.

5. We store the model metric attributes in the previous step as private members and publicly expose their corresponding getter functions instead:

```
    @property
    def bst_rwd(self):
        ...
    @property
    def bst_avg(self):
        ...
    @property
    def avg(self):
        ...
    @property
    def idx(self):
        ...
    ...
```

The `idx` attribute is critical for deciding when to copy the weights from the main CNN to the target CNN, while the `avg` attribute is useful for computing the running average of rewards that have been received in the past few episodes.

Training the DQN model to learn Pong

Now, we have all the necessary ingredients to start training the DQN model. Let's get started:

1. The following is a training wrapper function that will do everything we need it to do:

   ```
   def train(env, mdl, tgt_mdl, opt, rpl_bfr, dvc):
       log = TrMetadata()
       for epd in range(N_EPDS):
           run_epsd(env, mdl, tgt_mdl, opt, rpl_bfr, dvc, log, epd)
   ```

   Essentially, we initialize a logger and just run the DQN training system for a pre-defined number of episodes.

2. Before we actually run the training loop, we need to define the hyperparameter values, which are as follows:

   i. The batch size for each iteration of gradient descent to tune the CNN model
   ii. The environment, which in this case is the Pong video game
   iii. The epsilon value for the first episode
   iv. The epsilon value for the last episode
   v. The rate of depreciation for the epsilon value
   vi. Gamma; that is, the discounting factor
   vii. The initial number of iterations that are reserved just for pushing data to the replay buffer
   viii. The learning rate
   ix. The size or capacity of the experience replay buffer
   x. The total number of episodes to train the agent for
   xi. The number of iterations after which we copy the weights from the main CNN to the target CNN

   We can instantiate all of these hyperparameters in the following piece of code:

   ```
   B_S = 64
   ENV = "Pong-v4"
   EPS_STRT = 1.0
   EPS_FINL = 0.005
   EPS_DECAY = 100000
   G = 0.99
   INIT_LEARN = 10000
   LR = 1e-4
   MEM_CAP = 20000
   N_EPDS = 2000
   TGT_UPD_FRQ = 1000
   ```

   These values are experimental, and I encourage you to try changing them and observe the impact they have on the results.

3. This is the last step of the exercise where we actually execute the DQN training routine, as follows:

    i. First, we instantiate the game environment.

    ii. Then, we define the device that the training will happen on – either CPU or GPU, based on availability.

    iii. Next, we instantiate the main and target CNN models. We also define *Adam* as the optimizer for the CNN models.

    iv. We then instantiate an experience replay buffer.

    v. Finally, we begin training the main CNN model. Once the training routine finishes, we close the instantiated environment.

The code for this is as follows:

```
env = wrap_env(ENV)
dvc = torch.device("cuda") if torch.cuda.is_available() else torch.
device("cpu")
mdl, tgt_mdl = models_init(env, dvc)
opt = Adam(mdl.parameters(), lr=LR)
rpl_bfr = RepBfr(MEM_CAP)
train(env, mdl, tgt_mdl, opt, rpl_bfr, dvc)
env.close()
```

This should give us the following output:

```
checkpointing current model weights. highest running_average_reward of
-21.0 achieved!
episode_num 0, curr_reward: -21.0, best_reward: -21.0, running_avg_
reward: -21.0, curr_epsilon: 0.9972
episode_num 1, curr_reward: -21.0, best_reward: -21.0, running_avg_
reward: -21.0, curr_epsilon: 0.9947
episode_num 2, curr_reward: -21.0, best_reward: -21.0, running_avg_
reward: -21.0, curr_epsilon: 0.992
episode_num 3, curr_reward: -21.0, best_reward: -21.0, running_avg_
reward: -21.0, curr_epsilon: 0.9892
...
episode_num 2496, curr_reward: 15.0, best_reward: 21.0, running_avg_
reward: 9.76, curr_epsilon: 0.005
episode_num 2497, curr_reward: 19.0, best_reward: 21.0, running_avg_
reward: 9.76, curr_epsilon: 0.005
```

```
episode_num 2498, curr_reward: -1.0, best_reward: 21.0, running_avg_
reward: 9.76, curr_epsilon: 0.005
episode_num 2499, curr_reward: 8.0, best_reward: 21.0, running_avg_
reward: 9.88, curr_epsilon: 0.005
episode_num 2500, curr_reward: 12.0, best_reward: 21.0, running_avg_
reward: 9.84, curr_epsilon: 0.005
```

Furthermore, *Figure 11.6* shows the progression of the current rewards, best rewards, and average rewards against the progression of episodes:

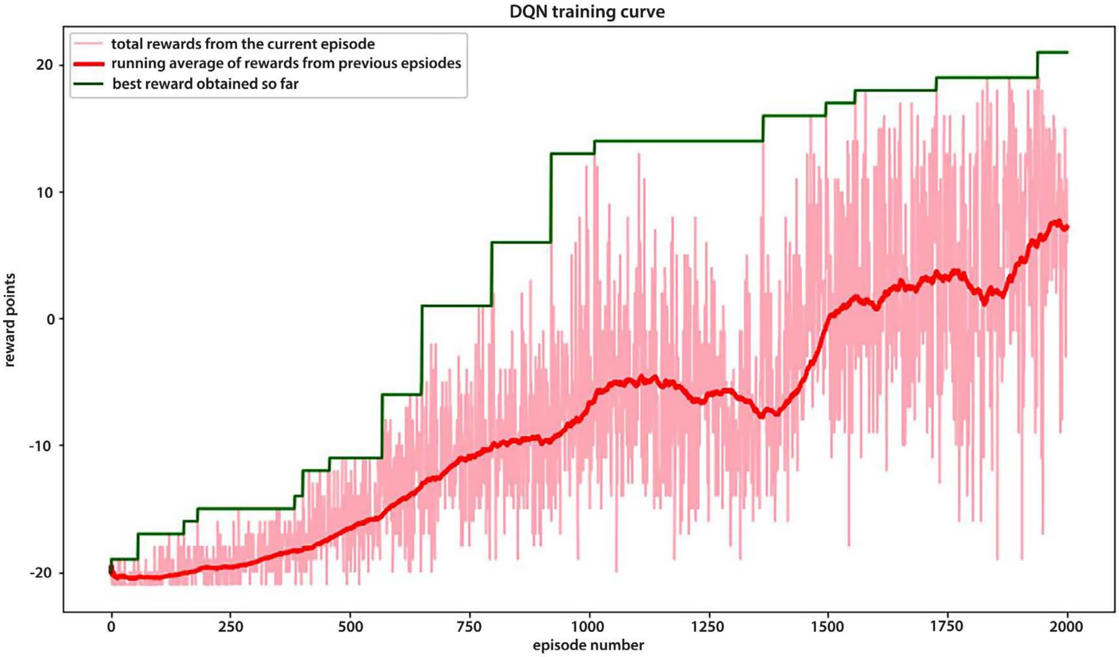


Figure 11.6: DQN training curves

Figure 11.7 shows how the epsilon value decreases over episodes during the training process:

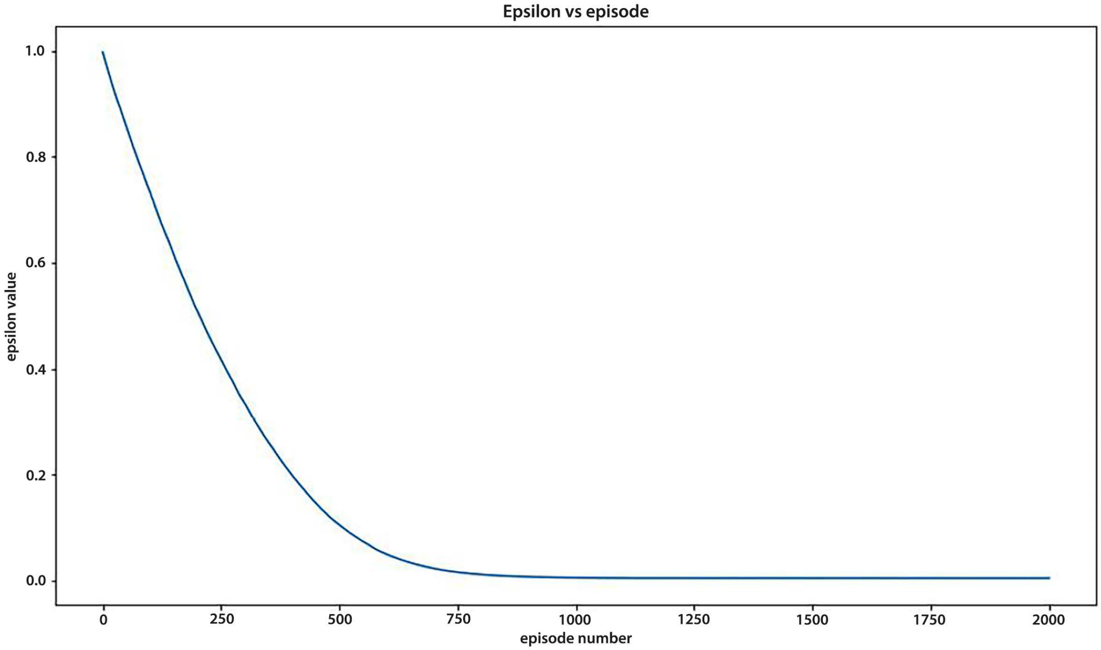


Figure 11.7: Epsilon variation over episodes

Notice that in *Figure 11.6*, the running average value of rewards in an episode (red curve) starts at **-20**, which is the scenario where the agent scores **0** points in a game and the opponent scores all **20** points. As the episodes progress, the average rewards keep increasing, and by episode number **1,500**, it crosses the zero mark. This means that after **1,500** episodes of training, the agent has leveled up against the opponent.

From here onward, the average rewards are positive, which indicates that the agent is winning against the opponent on average. We have only trained until **2,000** episodes, which already results in the agent winning by a margin of over **7** average points against the opponent. I encourage you to train it for longer and see if the agent can absolutely crush the opponent by always scoring all the points and winning by a margin of **20** points.

This concludes our deep dive into the implementation of a DQN model. DQN has been vastly successful and popular in the field of RL and is definitely a great starting point for those interested in exploring the field further. PyTorch, together with the `gym` library, is a great resource that enables us to work in various RL environments and work with different kinds of DRL models.

In this chapter, we have only focused on DQNs, but the lessons we've learned can be transferred to working with other variants of Q-learning models and other DRL algorithms.

Summary

RL is one of the fundamental branches of machine learning and is currently one of the hottest, if not the hottest, areas of research and development. RL-based AI breakthroughs such as AlphaGo from Google's DeepMind have further increased enthusiasm and interest in the field. This chapter provided an overview of RL and DRL and walked us through a hands-on exercise of building a DQN model using PyTorch.

RL is a vast field, and one chapter is not enough to cover everything. I encourage you to use the high-level discussions from this chapter to explore the details around those discussions. From the next chapter onward, we will focus on the practical aspects of working with PyTorch, such as model deployment, parallelized training, automated machine learning, and so on. In the next chapter, we discuss how to effectively train models in PyTorch using distributed training on CPUs, and GPUs, and using mixed precision training on GPUs.

Reference list

1. GitHub: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter11/pong.ipynb`
2. `gym` library: `https://github.com/openai/gym`

Learn more on Discord

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

`https://packt.link/mastorch`

12

Model Training Optimizations

Before serving pre-trained machine learning models, which we will discuss extensively in *Chapter 13*, *Operationalizing PyTorch Models into Production*, we need to train them. In *Chapters 2* to *6*, we saw the vast expanse of increasingly complex deep learning model architectures. Such gigantic models often have millions and even billions of parameters. The recent (at the time of writing) **Pathways Language Model** (**PaLM**) can have up to 540 billion parameters, for example using backpropagation to tune these many parameters requires enormous amounts of memory and compute power. And even then, model training can take days to finish.

In this chapter, we will explore ways of speeding up the model training process by distributing the training task across machines and processes within machines. We will learn about the distributed training APIs offered by PyTorch – `torch.distributed`, `torch.multiprocessing` and `torch.utils.data.distributed.DistributedSampler` – that make distributed training look easy.

We will also learn how to use mixed precision training with the help of APIs such as `torch.cuda.amp.autocast` and `torch.cuda.amp.GradScaler` to reduce the memory footprint while training deep learning models faster. Using the handwritten digits classification example from *Chapter 1*, *Overview of Deep Learning Using PyTorch*, we will demonstrate the speedup in training and reduced memory consumption on CPUs as well as GPUs by using PyTorch's distributed training as well as **Automatic Mixed Precision** (**AMP**) tools.

This chapter is broken down into the following topics:

- Distributed training with PyTorch
- Distributed training on GPUs with CUDA
- Automatic mixed precision training

At the end of this chapter, you will be able to fully utilize the hardware at your disposal for model training. And for training extremely large models, the tools discussed in this chapter will prove vital if not necessary.

Distributed training with PyTorch

In all previous exercises in this book, we have implicitly assumed model training happens in one machine and in a single Python process in that machine. In this section, we will revisit the exercise from *Chapter 1*, *Overview of Deep Learning Using PyTorch*, and transform the model training routine from regular to distributed training. In the process, we will explore the tools PyTorch offers in distributing the training process thereby making it both faster and more hardware efficient.

Training the MNIST model in a regular fashion

The handwritten digits classification model that we built in the first chapter was in the form of a Jupyter notebook. Here, we will first put that notebook code together as a single Python script file. The full code can be found on GitHub [1]. In the following steps, we recap the different parts of the model training code:

1. In the Python script, we first import the relevant libraries:

```
import torch
...
import argparse
```

2. Next, we define the CNN model architecture:

```
class ConvNet(nn.Module):
    def __init__(self):
        ...
    def forward(self, x):
        ...
```

3. We then define the model training routine. The full code is deliberately written here in order to later contrast with the distributed training mode:

```
def train(args):
    train_dataloader = torch.utils.data.DataLoader(
        datasets.MNIST(
            '../data', train=True, download=True,
                transform=transforms.Compose([
                    transforms.ToTensor(),
                    transforms.Normalize((0.1302,),
                                         (0.3069,))])),
        batch_size=128, shuffle=True)
    model = ConvNet()
    optimizer = optim.Adadelta(model.parameters(), lr=0.5)
    model.train()
```

In the first half of the function, we define the PyTorch training dataloader using the PyTorch training dataset. We instantiate our deep learning model – the ConvNet, and define the optimization module:

```
    for epoch in range(args.epochs):
        for b_i, (X, y) in enumerate(train_dataloader):
            X, y = X.to(device), y.to(device)
            pred_prob = model(X)
            # nll is the negative likelihood loss
            loss = F.nll_loss(pred_prob, y)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            if b_i % 10 == 0:
                print('epoch: {} [{}/{} ({:.0f}%)]\t training loss: 
{:.6f}'.format(
                    epoch, b_i, len(train_dataloader),
                    100. * b_i / len(train_dataloader),
                    loss.item()))
```

And in the second half, we run the training loop that runs for a defined number of epochs. Inside the loop, we run through the entire training dataset in batches with a defined batch size (128 in this case). For each batch containing 128 training data points, we run a forward pass with the model to compute prediction probabilities. We then use the predictions together with ground truth labels to compute a batch loss and use this loss to compute gradients in order to tune the model parameters using backpropagation.

4. We have all the components needed and we put it all together in this main() function:

```
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--epochs', default=1, type=int)
    args = parser.parse_args()
    start = time.time()
    train(args)
    print(f"Finished training in {time.time()-start} secs")
```

We use an arguments parser that will help us enter hyperparameters such as the number of epochs while running our Python training program from the command line. We also time the training routine so that we can later compare it with the distributed training routine.

5. The final piece of our Python script is making sure that the `main()` function runs when we execute this script from the command line:

   ```
   if __name__ == '__main__':
       main()
   ```

6. And now, we can execute the Python script by running the following command on the command line:

   ```
   python convnet_undistributed.py --epochs 1
   ```

   We are running the training for just a single epoch as the focus is not on model accuracy but on the model training time. You should see an output similar to the following:

   ```
   epoch: 0 [0/469 (0%)]  training loss: 2.314504
   epoch: 0 [10/469 (2%)] training loss: 1.530702
   epoch: 0 [20/469 (4%)] training loss: 0.880540
   epoch: 0 [30/469 (6%)] training loss: 0.571244
   ...
   epoch: 0 [430/469 (92%)] training loss: 0.057890
   epoch: 0 [440/469 (94%)] training loss: 0.141672
   epoch: 0 [450/469 (96%)] training loss: 0.063366
   epoch: 0 [460/469 (98%)] training loss: 0.087559
   Finished training in 38.82406210899353 secs
   ```

   It took roughly 39 seconds to train for 1 epoch, which equates to 469 batches, each having 128 data points, except for the last batch, which has 32 fewer data points than usual (as there are 60,000 data points in total). This training time will vary depending on the hardware used for model training.

   At this point, it is important to know what kind of machine this model is being trained on in order to have the reference context:

   ```
   Hardware Overview:
         Model Name: MacBook Pro
         Model Identifier: MacBookPro16,1
         Processor Name: 8-Core Intel Core i9
         Processor Speed: 2.3 GHz
         Number of Processors: 1
         Total Number of Cores: 8
         L2 Cache (per Core): 256 KB
         L3 Cache: 16 MB
         Hyper-Threading Technology: Enabled
         Memory: 32 GB
   ```

The above information is obtained by running the following command on a Mac terminal:

```
/Volumes/Macintosh\ HD/usr/sbin/system_profiler SPHardwareDataType
```

An important point to note from the above hardware specifications is that this machine consists of 8 CPU cores and 32 GB RAM. This is useful information when trying to parallelize the training routine, which we will get to next.

Training the MNIST model in a distributed fashion

We will basically repeat the above 6 steps, but this time we will make a few edits in the code that will enable distributed training, which should be faster than the regular training performed above. We will see that with the available distributed processing APIs from PyTorch, model training becomes much faster even with the added overhead of repeated data passing across processes or machines. The full code for this distributed training Python script can be found on GitHub [2].

1. Once again, we import the necessary libraries. This time we will have an additional couple of imports:

   ```
   import torch
   ...
   import torch.multiprocessing as mp
   import torch.distributed as dist
   ...
   import argparse
   ```

   While `torch.multiprocessing` helps spawn multiple Python processes within a machine (typically, we may spawn as many processes as there are CPU cores in the machine), `torch.distributed` enables communications between different machines as they jointly work towards training the model. During execution, we need to explicitly launch our model training script from within each of these machines.

   One of the built-in PyTorch communication backends, such as **Gloo**, will then take care of the communication between these machines. Inside each machine, multiprocessing will take care of further parallelizing the training task across several processes. I encourage you to read about multiprocessing and distribution in further detail at [3] and [4] respectively.

2. The model architecture definition step remains unchanged for obvious reasons:

   ```
   class ConvNet(nn.Module):
       def __init__(self):
           ...
       def forward(self, x):
           ...
   ```

3. And then we come to defining the `train()` function where most of the magic happens. Below is the code with highlighted additions in order to facilitate distributed training:

```
def train(cpu_num, args):
    rank = args.machine_id * args.num_processes + cpu_num
    dist.init_process_group(
        backend='gloo',
        init_method='env://',
        world_size=args.world_size,
        rank=rank)
    torch.manual_seed(0)
    device = torch.device("cpu")
```

As we can see, there is additional code at the very beginning consisting of two statements. First, a `rank` is calculated. This is essentially the ordinal ID of a process within the entire distributed system. For example, if we are using 2 machines with 4 CPU cores each, for full hardware utilization, we might want to launch a total of 8 processes, 4 in each machine. In this scenario, we will need to somehow label these 8 processes in order to later remember which process is which. We do so by assigning IDs 0 and 1 to the 2 machines and then IDs 0 to 3 to the 4 processes in each machine. Finally, the rank of the k^{th} process of the n^{th} machine is given by:

$$rank = n \times 4 + k$$

Equation 11.1

The second additional line of code uses the `torch.distributed` module's `init_process_group`, which for each launched process, specifies:

- The backend that will be used for communication between machines (`gloo` in this case)
- The total number of processes involved in distributed training (given by `args.world_size`), otherwise called `world_size`
- The rank of the process being launched

The `init_process_group` method ensures each process is blocked from further action until all the processes across the machines have been initiated using this method.

Regarding the backend, PyTorch provides the following three built-in backends for distributed training:

- Gloo
- NCCL
- MPI

In short, for distributed training on CPUs, use Gloo, and for GPUs, use NCCL. You can read about these communication backends in detail at [5]:

```
    train_dataset = datasets.MNIST(
        '../data', train=True, download=True,
        transform=transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.1302,), (0.3069,))]))
''' 0.1302, 0.3069 are mean and std
deviation computed on MNIST training dataset '''
    train_sampler = \
        torch.utils.data.distributed.DistributedSampler(
            train_dataset,
            num_replicas=args.world_size,
            rank=rank)
    train_dataloader = torch.utils.data.DataLoader(
       dataset=train_dataset,
       batch_size=args.batch_size,
       shuffle=False,
       num_workers=0,
       sampler=train_sampler)
    model = ConvNet()
    optimizer = optim.Adadelta(model.parameters(), lr=0.5)
    model = nn.parallel.DistributedDataParallel(model)
    model.train()
```

Compared to the undistributed training exercise, we now have separated the MNIST dataset instantiation from the dataloader instantiation. And in between these two steps, we have inserted a data sampler – `torch.utils.data.distributed.DistributedSampler`. The sampler's task is to divide the training dataset into `world_size` number of partitions such that all processes in the distributed training session get to work on equal portions of data. Note that we have set `shuffle` to `False` in the dataloader instantiation because we are using the sampler for distributing data.

Another addition in our code is the `nn.parallel.DistributedDataParallel` function, which is applied to the model object. This is perhaps the most important part of this code as `DistributedDataParallel` is a critical component/API that facilitates the gradient descent algorithm in a distributed fashion. The following happens under the hood:

- Each spawned process in the distributed universe gets its own model copy.
- Each model per process maintains its own optimizer and undergoes a local optimization step in sync with the global iteration.
- At each distributed training iteration, individual losses and hence gradients are calculated in each process and these gradients are then averaged across processes.
- The averaged gradient is then universally back-propagated to each of the model copies, which tunes their parameters.
- Because of the universal backpropagation step, all model parameters are the same at each iteration, and therefore are automatically synced.

`DistributedDataParallel` ensures that each Python process runs on an independent Python interpreter, doing away with the GIL limitation that could be posed if multiple models were instantiated in multiple threads under the same interpreter.

> **Note**
>
> If you are unfamiliar with Python multiprocessing and GIL, you can find excellent explanations at [6].

This further boosts performance, especially for models that require intense Python-specific processing.

```
    for epoch in range(args.epochs):
        for b_i, (X, y) in enumerate(train_dataloader):
            X, y = X.to(device), y.to(device)
            pred_prob = model(X)
            # nll is the negative likelihood loss
            loss = F.nll_loss(pred_prob, y)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            if b_i % 10 == 0 and cpu_num==0:
                print('epoch: {} [{}/{} ({:.0f}%)]\t training loss:
{:.6f}'.format(
                    epoch, b_i, len(train_dataloader),
                    100. * b_i / len(train_dataloader),
                    loss.item()))
```

Finally, the training loop is almost the same as before. The only difference is that we restrict the process with rank 0 to getting our logs. We do so because the machine with rank 0 is used to set up all communications. Hence, we notionally use the process with rank 0 as our reference to track the model training performance. If we did not restrict it, we would get as many log lines per model training iteration as the number of processes.

4. Moving on from the `train()` function to the `main()` function, we can see a lot of additions in the code:

```
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '--num-machines', default=1, type=int)
    parser.add_argument(
        '--num-processes', default=1, type=int)
    parser.add_argument('--machine-id', default=0, type=int)
    parser.add_argument('--epochs', default=1, type=int)
    parser.add_argument(
        '--batch-size', default=128, type=int)
    args = parser.parse_args()
    args.world_size = args.num_processes * args.num_machines
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '8892'
    start = time.time()
    mp.spawn(train, nprocs=args.num_processes, args=(args,))
    print(f"Finished training in {time.time()-start} secs")
```

Firstly, we observe the following additional arguments:

- num_machines: This is exactly as it is named – the number of machines.
- num_processes: The number of processes to be spawned in each machine.
- machine_id: The ordinal ID of the current machine. Remember, this Python script will need to be launched separately on each of the machines.
- batch_size: The number of data points in a batch. Why do we suddenly need this? As mentioned earlier:
    - All processes will have their own gradients, which will be averaged to get the overall gradient per iteration.
    - The full training dataset is divided into world_size number of individual datasets.

Therefore, at each iteration, the full batch of data needs to be divided into world_size number of sub-batches of data per process. And because batch_size is now coupled to world_size, we provide it as an input argument for an easier training interface.

After the additional arguments, we calculate world_size as a derived argument. And then, we specify two important environment variables:

- MASTER_ADDR: The IP address of the machine that runs the process with rank 0.
- MASTER_PORT: An available port on the machine that runs the process with rank 0.

As mentioned in *step 3*, the machine with rank 0 sets up all the backend communications, and hence it is important for the entire system to be able to locate that hosting machine at all times. That is why we provide its IP address and port. In this example, the training will be run on a single local machine, and hence a localhost address suffices. However, when running multi-machine training across servers located remotely, we need to provide the exact IP address of the rank 0 server with a free port.

The final change is the use of multiprocessing to spawn num_processes number of processes in a machine instead of simply running a single training process. The distributional arguments are passed to each of the spawned processes such that the processes and machines coordinate among themselves during the model training run.

5. The final piece of our distributed training code is the same as before:

```
if __name__ == '__main__':
    main()
```

6. We are now in a position to launch the distributed training script. We will begin with an undistributed-like run using the distributed-like script. We do so by simply setting the number of machines as well as the number of processes to 1:

```
python convnet_distributed.py --num-machines 1 --num-processes 1
--machine-id 0 --batch-size 128
```

Note that since there is a single process being used for training, `batch_size` remains unchanged in comparison to the previous exercise. You will see the following output:

```
epoch: 0 [0/469 (0%)]  training loss: 2.310591
epoch: 0 [10/469 (2%)] training loss: 1.276356
epoch: 0 [20/469 (4%)] training loss: 0.693506
epoch: 0 [30/469 (6%)] training loss: 0.666963
...
epoch: 0 [430/469 (92%)] training loss: 0.088113
epoch: 0 [440/469 (94%)] training loss: 0.139962
epoch: 0 [450/469 (96%)] training loss: 0.138155
epoch: 0 [460/469 (98%)] training loss: 0.120145
Finished training in 38.07329821586609 secs
```

If we compare this result with the output from undistributed training in the previous section, the training time is almost the same, around 38-39 seconds. The training loss evolution is also similar.

7. We will now run a truly distributed training session with 2 processes instead of 1. And accordingly, we will halve the batch size from 128 to 64:

```
python convnet_distributed.py --num-machines 1 --num-processes 2
--machine-id 0 --batch-size 64
```

You will see the following output:

```
epoch: 0 [0/469 (0%)]  training loss: 2.309349
epoch: 0 [10/469 (2%)] training loss: 1.524054
epoch: 0 [20/469 (4%)] training loss: 0.993495
epoch: 0 [30/469 (6%)] training loss: 0.777370
...
epoch: 0 [430/469 (92%)] training loss: 0.070469
epoch: 0 [440/469 (94%)] training loss: 0.065329
epoch: 0 [450/469 (96%)] training loss: 0.036549
epoch: 0 [460/469 (98%)] training loss: 0.166292
Finished training in 25.349677085876465 secs
```

As we can see, there is quite a reduction in training time from 38 seconds to 25 seconds. The training loss evolution once again seems to be unaffected, which shows how distributed training can speed up training without losing model accuracy.

8. Let us go further and use 4 processes instead of 2, and accordingly reduce the batch size from 64 to 32:

```
python convnet_distributed.py --num-machines 1 --num-processes 4
--machine-id 0 --batch-size 32
```

You will see the following output:

```
epoch: 0 [0/469 (0%)]  training loss: 2.314902
epoch: 0 [10/469 (2%)] training loss: 1.642720
epoch: 0 [20/469 (4%)] training loss: 0.802527
epoch: 0 [30/469 (6%)] training loss: 0.664064
...
epoch: 0 [430/469 (92%)] training loss: 0.079896
epoch: 0 [440/469 (94%)] training loss: 0.265193
epoch: 0 [450/469 (96%)] training loss: 0.033737
epoch: 0 [460/469 (98%)] training loss: 0.117078
Finished training in 18.8058762550354 secs
```

There is a further reduction in training time from 25 seconds to 19 seconds. The training loss evolution still seems similar to the previous runs. So far, with the help of distributed training, we have reduced the training time by 2x from 38 seconds to 19 seconds.

9. Let us go even further and use 8 processes instead of 4, and accordingly reduce the batch size from 32 to 16:

```
python convnet_distributed.py --num-machines 1 --num-processes 8
--machine-id 0 --batch-size 16
```

You will see the following output:

```
epoch: 0 [0/469 (0%)]  training loss: 2.312518
epoch: 0 [10/469 (2%)] training loss: 1.371002
epoch: 0 [20/469 (4%)] training loss: 1.176817
epoch: 0 [30/469 (6%)] training loss: 0.883302
...
epoch: 0 [430/469 (92%)] training loss: 0.063177
epoch: 0 [440/469 (94%)] training loss: 0.047881
epoch: 0 [450/469 (96%)] training loss: 0.113552
epoch: 0 [460/469 (98%)] training loss: 0.047556
Finished training in 23.093057870864868 secs
```

Contrary to expectations, the training time doesn't reduce further and, in fact, increases slightly from 19 seconds to 23 seconds. This is where we need to recall from the hardware specifications inspected earlier that the machine has 8 CPU cores and all the cores are occupied with 1 process each. As this session is being run on a local machine, there are other processes running as well (such as Google Chrome), which may fight for resources with one or more of our distributed training processes. In practice, training models in a distributed fashion is done on remote machines whose only job is model training, and on such machines, it is advisable to use as many processes (or even more) as the number of CPU cores.

10. As a final note, because we have only used 1 machine in this exercise, we only needed to launch one Python script to start training. If, however, you are training on multiple machines, then besides applying the changes to `MASTER_ADDR` and `MASTER_PORT` as advised in step 4, you need to launch one Python script on each machine. For example, if there are 2 machines, then on machine 1, run:

    ```
    python convnet_distributed.py --num-machines 2 --num-processes 2
    --machine-id 0 --batch-size 32
    ```

    And, on machine 2, run:

    ```
    python convnet_distributed.py --num-machines 2 --num-processes 2
    --machine-id 1 --batch-size 32
    ```

This concludes our hands-on discussion on training deep learning models on CPUs using PyTorch in a distributed fashion that yields significant speedups. With a few lines of added code, a general PyTorch model training script can be turned into the distributed training model. The exercise performed above is for a simple convolutional network. But because we did not even touch the model architecture code, the above exercise can easily be extended for more complex learning models where the gains will be more visible and needed. In the next section, we will briefly discuss how to apply similar code changes in order to facilitate distributed training on GPUs.

Distributed training on GPUs with CUDA

Throughout the various exercises in this book, you may have noticed a common line of PyTorch code:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

This code simply looks for the available compute device and prefers CUDA (GPU) over CPU. The preference is because of the computational speedups that GPUs can provide on regular neural network operations such as matrix multiplications and additions through parallelization. In this section, we look into speeding it up further with the help of distributed training on GPUs. We will develop the work done in the previous exercise. Most of the code looks the same. In the below steps, we will highlight the changes. Executing the script is left for readers as an exercise. The full code is available on GitHub [7]:

1. While the imports and model architecture definition code are exactly the same as before, there are a few changes in the train() function:

```
def train(gpu_num, args):
    rank = args.machine_id * args.num_processes + gpu_num
    dist.init_process_group(
    backend='nccl',
    init_method='env://',
    world_size=args.world_size,
    rank=rank)
    torch.manual_seed(0)
    model = ConvNet()
    torch.cuda.set_device(gpu_num)
    model.cuda(gpu_num)
    # nll is the negative likelihood loss
    criterion = nn.NLLLoss().cuda(gpu_num)
```

As discussed in *step 3* of the previous exercise, NCCL is the preferred choice of communication backend when working with GPUs. And both the model and the loss function need to be placed on the GPU device to ensure the utilization of sped-up parallelized matrix operations offered by the GPUs:

```
    train_dataset = ...
    train_sampler = ...
    train_dataloader = torch.utils.data.DataLoader(
       dataset=train_dataset,
       batch_size=args.batch_size,
       shuffle=False,
       num_workers=0,
       pin_memory=True,
       sampler=train_sampler)
    optimizer = optim.Adadelta(model.parameters(), lr=0.5)
    model = nn.parallel.DistributedDataParallel(
        model, device_ids=[gpu_num])
    model.train()
```

The DistributedDataParallel API takes in an additional parameter – device_ids – that takes in the rank of the GPU process it is called from. Also, we can notice an additional parameter, pin_memory, under the dataloader, which is set to True. This essentially helps in faster data transfer from the host (the CPU in this case, where the dataset is loaded) to the various devices (GPUs) during model training.

This parameter enables the dataloader to *pin* data into CPU memory, in other words, allocate the data samples into fixed page-locked CPU memory slots. The data from these slots is then copied to the respective GPUs during training. You can read more about the pinning strategy at [8]. The `pin_memory=True` mechanism works together with the `non_blocking=True` argument, as shown in the code below:

```
for epoch in range(args.epochs):
        for b_i, (X, y) in enumerate(train_dataloader):
            X, y = X.cuda(non_blocking=True), y.cuda(non_blocking=True)
            pred_prob = model(X)
            ...
```

By invoking these two parameters – `pin_memory` and `non_blocking` – we enable the overlap between:

- CPU to GPU data (ground truth) transfer
- GPU model training compute (or GPU kernel execution)

This basically makes the overall GPU training process more efficient (faster).

2. Besides the changes in the `train()` function, we change a few lines in the `main()` function as well:

```
def main():
    ...
    parser.add_argument('--num-gpu-processes', default=1, type=int)
    ...
    args.world_size = \
        args.num_gpu_processes * args.num_machines
    ...
    mp.spawn(train, nprocs=args.num_gpu_processes, args=(args,))
```

Instead of `num_process`, we now have `num_gpu_processes`. And the rest of the code changes accordingly. The rest of the GPU code is the same as before. We are all set to run the distributed training on GPUs by executing a command such as:

```
python convnet_distributed_cuda.py --num-machines 1 --num-gpu-processes 2
--machine-id 0 --batch-size 64
```

This brings us to the end of briefly discussing distributed model training on GPUs using PyTorch. As mentioned in the previous section, the code changes suggested for the above example should be extendable for other deep learning models. Using distributed training on GPUs is actually how most of the latest state-of-the-art deep learning models are trained. This should get you started with training your own amazing models using GPUs.

Furthermore, libraries such as **Horovod**, **DeepSpeed**, and **PyTorch Lightning** provide sleek APIs to facilitate distributed training of PyTorch models. I recommend you check these libraries out in addition to our discussions above. And now, we'll look at another way of optimizing model training.

Automatic mixed precision training

Most deep learning models involve float32 tensors to represent inputs, weights, and so on. These float32 tensors consist of 32 bits and hence represent information with high precision. While such high precision is important to be retained for maintaining numerical stability across sensitive deep learning operations such as loss calculation [9], many operations can be done with less precision using float16 tensors [10] without compromising model performance improvement during training. Modern GPUs are optimized to run operations with float16 tensors faster than with float32 tensors while utilizing less memory [11]. The use of float16 together with float32 type tensors during model training to make the process faster and less memory-consuming is referred to as mixed precision training.

In this section, we will first revisit our undistributed training code for MNIST, but using a GPU instead of a CPU, and then we will apply mixed precision training on top to observe the gains in memory utilization and speed.

Regular model training on a GPU

The code for undistributed training on the MNIST dataset using a GPU is available on GitHub [12]. This Python script is the same as the code we wrote in the *Training the MNIST model in a regular fashion* section, except for these two changes:

1. Instead of `device = torch.device("cpu")`, we write `device = torch.device("cuda")`.
2. Right after model instantiation `model = ConvNet()`, we write `model.to(device)` to load the model on the GPU. In the case of the CPU, this is done by default, but it's a good practice to specify this statement nonetheless.

We can execute the Python script by running the following command on the command line:

```
python convnet_undistributed_cuda.py --epochs 1
```

You will see the following output:

```
epoch: 0 [0/469 (0%)]    training loss: 2.317266
epoch: 0 [10/469 (2%)]   training loss: 1.474984
epoch: 0 [20/469 (4%)]   training loss: 0.836410
epoch: 0 [30/469 (6%)]   training loss: 0.681523
...
epoch: 0 [430/469 (92%)]        training loss: 0.094629
epoch: 0 [440/469 (94%)]        training loss: 0.123775
epoch: 0 [450/469 (96%)]        training loss: 0.099774
epoch: 0 [460/469 (98%)]        training loss: 0.152699
Finished training in 17.14671039581299 secs
```

Training with the GPU takes 17 seconds for an epoch as opposed to 39 seconds per epoch with a CPU. We are using 1 NVIDIA Tesla T4 GPU for this exercise. If you are using a different class of GPU, you might experience different training times. For example, an NVIDIA A100, which is more powerful, will finish training in less than 17 seconds.

Besides the training logs, we can also monitor the GPU consumption during training. For that, we will use the `nvidia-smi` command. If you execute this command on the terminal, you should see something like the following:

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.141.03   Driver Version: 470.141.03   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:05.0 Off |                    0 |
| N/A   41C    P8     9W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Note

If the `nvidia-smi` command does not work for you, you will need to find where nvidia drivers are installed on your system. Typically, the `nvidia-smi` binary is located at `/usr/local/nvidia/bin/nvidia-smi`, which needs to be added to the system `PATH` variable for you to be able to execute the `nvidia-smi` command. NVIDIA's CUDA installation page [13] provides you with all the relevant details, including the supported Linux distributions [14].

While the training runs, we can watch the output of `nvidia-smi` using the following command:

```
watch -n0.1 nvidia-smi
```

This shows us live GPU utilization metrics every 0.1 seconds. While training the MNIST model on a GPU, a captured snapshot of the above command looks like the following:

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.141.03   Driver Version: 470.141.03   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:05.0 Off |                    0 |
| N/A   43C    P0    30W /  70W |   1112MiB / 15109MiB |     19%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+


+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

Marked in bold are the GPU memory utilization (1112 MB) and GPU utilization percentages (19%). Next, we will train the same model on the MNIST dataset using mixed precision on a GPU and will observe the changes in these key metrics: training speed, GPU memory, and processor utilization.

Mixed precision training on a GPU

PyTorch provides us with the `torch.cuda.amp.autocast` API, which decides the precision level (float32 or float16) for different GPU operations to improve performance while maintaining model training stability and model accuracy. Thanks to this API, we only need to make a few changes in our regular training code to enable mixed precision training. The code for this exercise is available on GitHub [15]. This Python script is the same as the code we wrote in the *Regular model training on a GPU* section [12], except for the following change. Previously, we had the following model feed-forward and loss calculation steps:

```
pred_prob = model(X)
loss = F.nll_loss(pred_prob, y) # nll is the negative likelihood loss
```

And now, we will have the following:

```
with torch.cuda.amp.autocast():
    pred_prob = model(X)
    loss = F.nll_loss(pred_prob, y)
    # nll is the negative likelihood loss
```

Essentially, we let `autocast` decide the casting of input (`X`), parameters (`model`), and output (`y`) as float32 or float16. We can execute the mixed precision training script with the following command:

```
python convnet_undistributed_cuda_amp.py --epochs 1
```

You shall see the following output:

```
epoch: 0 [0/469 (0%)]	training loss: 2.317255
epoch: 0 [10/469 (2%)]	training loss: 1.468484
epoch: 0 [20/469 (4%)]	training loss: 0.890393
epoch: 0 [30/469 (6%)]	training loss: 0.573039
...
epoch: 0 [430/469 (92%)]	training loss: 0.101214
epoch: 0 [440/469 (94%)]	training loss: 0.137581
epoch: 0 [450/469 (96%)]	training loss: 0.082024
epoch: 0 [460/469 (98%)]	training loss: 0.168907
Finished training in 16.81143617630005 secs
```

And in parallel, if you were running the following command in a second terminal window/tab:

```
watch -n0.1 nvidia-smi
```

You would see the following output:

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.141.03   Driver Version: 470.141.03   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:05.0 Off |                    0 |
| N/A   46C    P0    32W /  70W |    986MiB / 15109MiB |     27%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+


+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

The three key numbers to notice are training time, GPU memory consumption, and GPU utilization percentage. The training time reduces mildly from 17.1 seconds to 16.8 seconds, and the GPU memory consumption goes down from 1112 MB to 986 MB. The GPU utilization goes up from 19% to 27%, which is an indication that `autocast` is indeed utilizing the GPU appropriately for the float16 operations.

While the performance gains are mild in this exercise, such gains tend to scale up with the size and GPU-friendliness of deep learning models. In the training logs, we can observe that the introduction of mixed precision doesn't interfere with the training progress and stability. However, if that happens, PyTorch's `torch.cuda.amp.GradScaler` can help by minimizing gradient underflow. Extremely small gradient values such as 1e-27 would be zeroed in a float16 representation leading to gradient underflow. `GradScaler` helps avoid such situations when using mixed-precision training. Thanks to this API, we simply need to replace the lines of code involving gradient update steps. Previously these lines were:

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

With `GradScaler`, these lines look like the following:

```
scaler.scale(loss).backward()
scaler.step(optimizer)
scaler.update()
optimizer.zero_grad()
```

Before starting the training loop, we would also need to instantiate this scaler:

```
scaler = torch.cuda.amp.GradScaler()
```

I encourage you to try these changes in our MNIST model training code and observe the changes in model training time, GPU memory consumption, and GPU utilization.

Using these two APIs – `torch.cuda.amp.autocast` and `torch.cuda.amp.GradScaler` – we can neatly modify any PyTorch model training code for **Automatic Mixed-Precision** (**AMP**) training. AMP can be performed on both CPUs as well as GPUs, but the yields are higher for the latter because of the specific hardware optimizations favorable to float16 vs float32. While we have demonstrated AMP on a GPU in our exercises, the torch APIs for AMP work similarly on CPUs. The only difference is that float32 is cast to `bfloat16` instead of float16, because only `torch.bfloat16` is supported in CPU mode at the time of writing [16]. (Both bfloat16 and float16 are 16-bit representations but differ slightly [17].)

Torch website is a good resource for both more information about AMP [18] as well as more AMP examples [19]. This concludes our discussions on optimizing model training performance using mixed precision. This should enable you to improve the performance of your existing deep learning model training code as well as to write more efficient training code in the future.

Summary

In this chapter, we covered an important practical aspect of machine learning, that is, how to optimize the model training process. We explored the extent and power of distributed training using PyTorch, both on CPUs as well as GPUs. We then learned how to use mixed precision training to further optimize the model training process.

In the next chapter, we will focus on some practical aspects of working with PyTorch in production. We will discuss how to deploy trained models into production systems, converting PyTorch models into universal formats such as ONNX, as well as translating PyTorch code written in Python into C++ and creating executable binaries.

Reference list

1. Notebook code as a Python script: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter12/convnet_undistributed.py`
2. Distributed training script: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter12/convnet_distributed.py`
3. PyTorch Multiprocessing: `https://pytorch.org/docs/stable/multiprocessing.html`
4. PyTorch Distributed Communication: `https://pytorch.org/docs/stable/distributed.html`
5. Communication backends: `https://pytorch.org/tutorials/intermediate/dist_tuto.html#communication-backends`
6. Python multiprocessing and GIL: `https://superfastpython.com/multiprocessing-pool-gil/`
7. Distributed training on GPUs code: `https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/11_distributed_training/convnet_distributed_cuda.py`
8. Optimizing data transfers in CUDA: `https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/`
9. Float32 autocast ops: `https://pytorch.org/docs/master/amp.html#cuda-ops-that-can-autocast-to-float32`
10. Float16 autocast ops: `https://pytorch.org/docs/master/amp.html#cuda-ops-that-can-autocast-to-float16`
11. Do More With Miced Precision Training: `https://developer.nvidia.com/automatic-mixed-precision`
12. Undistributed training on MNIST: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter12/convnet_undistributed_cuda.py`
13. Nvidia CUDA installation docs: `https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html`
14. CUDA supported Linux distributions: `https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html#overview`
15. Mixed precision training on GPU: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter12/convnet_undistributed_cuda_amp.py`

16. PyTorch Automatic Mixed Precision Package: `https://pytorch.org/docs/master/amp.html`
17. Bfloat16 vs float16 `https://github.com/stas00/ml-ways/blob/master/numbers/bfloat16-vs-float16-study.ipynb`
18. Torch AMP info: `https://pytorch.org/tutorials/recipes/recipes/amp_recipe.html`
19. Torch AMP examples: `https://pytorch.org/docs/stable/notes/amp_examples.html`

Learn more on Discord

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

`https://packt.link/mastorch`

13

Operationalizing PyTorch Models into Production

So far in this book, we have covered how to train and test different kinds of machine learning models using PyTorch. We started by reviewing the basic elements of PyTorch that enable us to work on deep learning tasks efficiently. Then, we explored a wide range of deep learning model architectures and applications that can be written using PyTorch.

In this chapter, we will focus on taking these models into production. But what does that mean? Basically, we will discuss the different ways of taking a trained and tested model (object) into a separate environment where it can be used to make predictions or inferences on incoming data. This is what is referred to as the **productionization** of a model, as the model is deployed into a production system.

We will begin by discussing some common approaches you can take to serve PyTorch models in production environments, starting from defining a simple model inference function and going all the way to using model microservices. We will then take a look at TorchServe, which is a scalable PyTorch model-serving framework that has been jointly developed by AWS and Facebook.

We will then dive into the world of exporting PyTorch models using **TorchScript**, which, through **serialization**, makes our models independent of the Python ecosystem so that they can be, for instance, loaded in a C++-based environment. We will also look beyond the Torch framework and the Python ecosystem as we explore **ONNX** – an open source universal format for machine learning models, which will help us export PyTorch-trained models to non-PyTorch and non-Pythonic environments.

Finally, we will briefly discuss how to use PyTorch for model serving with some of the well-known cloud platforms, such as **Amazon Web Services** (**AWS**), **Google Cloud**, and **Microsoft Azure**.

Throughout this chapter, we will use the handwritten digits image classification **convolutional neural network** (**CNN**) model that we trained in *Chapter 1, Overview of Deep Learning Using PyTorch*, as our reference. We will demonstrate how that trained model can be deployed and exported using the different approaches discussed in this chapter.

This chapter is broken down into the following sections:

- Model serving in PyTorch
- Serving a PyTorch model using TorchServe
- Exporting universal PyTorch models using TorchScript and ONNX
- Serving a PyTorch model in the cloud

Model serving in PyTorch

In this section, we will begin by building a simple PyTorch inference pipeline that can make predictions given some input data and the location of a previously trained and saved PyTorch model. We will proceed thereafter to place this inference pipeline on a model server that can listen to incoming data requests and return predictions. Finally, we will advance from developing a model server to creating a model microservice using Docker.

Creating a PyTorch model inference pipeline

We will work with the handwritten digits image classification CNN model that we built in *Chapter 1*, *Overview of Deep Learning Using PyTorch*, on the MNIST dataset. Using this trained model, we will build an inference pipeline that shall be able to predict a digit between 0 and 9 for a given handwritten-digit input image.

For the process of building and training the model, please refer to the *Training a neural network using PyTorch* section of *Chapter 1*, *Overview of Deep Learning Using PyTorch*. For the full code of this exercise, you can refer to our GitHub repository [1].

Saving and loading a trained model

In this section, we will demonstrate how to efficiently load a saved pre-trained PyTorch model, which will later be used to serve requests.

So, using the notebook code from *Chapter 1*, *Overview of Deep Learning Using PyTorch*, we have trained a model and evaluated it against test data samples. But what next? In real life, we would like to close this notebook and, later on, still be able to use this model that we worked hard on training to make inferences on handwritten-digit images. This is where the concept of serving a model comes in.

From here, we will get into a position where we can use the preceding trained model in a separate Jupyter notebook without having to do any (re)training. The crucial next step is to save the model object into a file that can later be restored/de-serialized. PyTorch provides two main ways of doing this:

1. The less recommended way is to save the entire model object as follows:

   ```
   torch.save(model, PATH_TO_MODEL)
   ```

   And then, the saved model can be later read as follows:

   ```
   model = torch.load(PATH_TO_MODEL)
   ```

Although this approach looks the most straightforward, this can be problematic in some cases. This is because we are not only saving the model parameters but also the model classes and directory structure used in our source code. If our class signatures or directory structures change later, loading the model will fail in potentially unfixable ways.

2. The second and more recommended way is to only save the model parameters as follows:

```
torch.save(model.state_dict(), PATH_TO_MODEL)
```

Later, when we need to restore the model, first, we instantiate an empty model object and then load the model parameters into that model object as follows:

```
model = ConvNet()
model.load_state_dict(torch.load(PATH_TO_MODEL))
```

We will use the recommended way to save the model, as shown in the following code:

```
PATH_TO_MODEL = "./convnet.pth"
torch.save(model.state_dict(), PATH_TO_MODEL)
```

The `convnet.pth` file is essentially a pickle file containing model parameters.

At this point, we can safely close the notebook we were working on and open another one, which is available in our GitHub repository [2]:

1. As a first step, we will once again need to import libraries:

```
import torch
```

2. Next, we need to instantiate an empty CNN model once again. Ideally, the model definition done in *step 1* would be written in a Python script (say, `cnn_model.py`), and then we would simply need to write this:

```
from cnn_model import ConvNet
model = ConvNet()
```

However, since we are operating in Jupyter notebooks in this exercise, we shall rewrite the model definition and then instantiate it as follows:

```
class ConvNet(nn.Module):
    def __init__(self):
        ...
    def forward(self, x):
        ...
model = ConvNet()
```

3. We can now restore the saved model parameters in this instantiated model object as follows:

   ```
   PATH_TO_MODEL = "./convnet.pth"
   model.load_state_dict(torch.load(
       PATH_TO_MODEL, map_location="cpu"))
   ```

   You shall see the following output:

   ```
   <All keys matched successfully>
   ```

   This essentially means that the parameter loading is successful. That is, the model that we have instantiated has the same structure as the model whose parameters were saved and are now being restored. We specify that we are loading the model on a CPU device as opposed to a GPU.

4. Finally, we want to specify that we do not wish to update or change the parameter values of the loaded model, and we will do so with the following line of code:

   ```
   model.eval()
   ```

   This should give the following output:

   ```
   ConvNet(
       (cn1): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1)
       (cn2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
       (dp1): Dropout2d(p=0.1, inplace=False)
       (dp2): Dropout2d(p=0.25, inplace=False)
       (fc1): Linear(in_features=4608, out_features=64, bias=True)
       (fc2): Linear(in_features=64, out_features=10, bias=True))
   ```

This again verifies that we are indeed working with the same model (architecture) that we trained.

Building the inference pipeline

Having successfully loaded a pre-trained model in a new environment (notebook) in the previous section, we shall now build our model inference pipeline and use it to run model predictions:

1. At this point, we have the previously trained model object fully restored. We shall now load an image that we can run the model prediction on using the following code:

   ```
   image = Image.open("./digit_image.jpg")
   ```

The image file should be available in the exercise folder and is as follows:

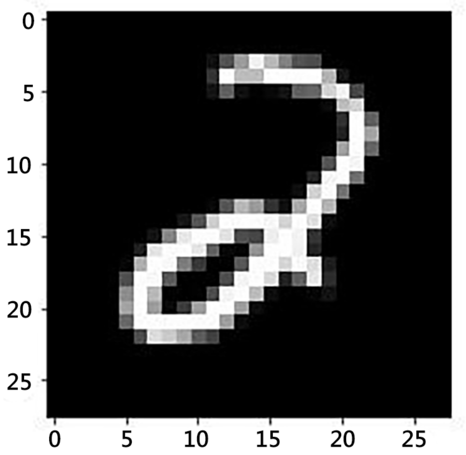


Figure 13.1: Model inference input image

It is not necessary to use this particular image in the exercise. You may use any image you want, to check how the model reacts to it.

2. In any inference pipeline, there are three main components at the core of it: (a) the data pre-processing component, (b) the model inference (a forward pass in the case of neural networks), and (c) the post-processing step.

   We will begin with the first part by defining a function that takes in an image and transforms it into the tensor that shall be fed to the model as input as follows:

```
def image_to_tensor(image):
    gray_image = transforms.functional.to_grayscale(image)
    resized_image = transforms.functional.resize(gray_image, (28, 28))
    input_image_tensor = \
        transforms.functional.to_tensor(resized_image)
    input_image_tensor_norm = \
        transforms.functional.normalize(input_image_tensor,
                                        (0.1302,), (0.3069,))
    return input_image_tensor_norm
```

   This can be seen as a series of steps:

   1. First, the RGB image is converted into a grayscale image.
   2. The image is then resized as a `28x28` pixel image because this is the image size the model is trained with.

    3. Then, the `image` array is converted into a PyTorch tensor.
    4. And finally, the pixel values in the tensor are normalized with the same mean and standard deviation values as those used during model training.

   Having defined this function, we call it to convert our loaded image into a tensor:

   ```
   input_tensor = image_to_tensor(image)
   ```

3. Next, we define the **model inference functionality**. This is where the model takes in a tensor as input and outputs the predictions. In this case, the prediction will be any digit between 0 and 9 and the input tensor will be the tensorized form of the input image:

   ```
   def run_model(input_tensor):
       model_input = input_tensor.unsqueeze(0)
       with torch.no_grad():
           model_output = model(model_input)[0]
       model_prediction = \
           model_output.detach().numpy().argmax()
       return model_prediction
   ```

   `model_output` contains the raw predictions of the model, which contains a list of predictions for each image. Because we have only one image in the input, this list of predictions will just have one entry at index `0`. The raw prediction at index `0` is essentially a tensor with 10 probability values for digits 0,1,2...9, in that order. This tensor is converted into a `numpy` array, and finally, we choose the digit that has the highest probability.

4. We can now use this function to generate our model prediction. The following code uses the `run_model` model inference function from *step 3* to generate the model prediction for the given input data, `input_tensor`:

   ```
   output = run_model(input_tensor)
   print(output)
   print(type(output))
   ```

   This should output the following:

   ```
   2
   <class 'numpy.int64'>
   ```

   As we can see the model outputs a `numpy` integer. And based on the image shown in *Figure 13.1*, the model output seems correct.

5. Besides just outputting the model prediction, we can also write a debug function to dig deeper into metrics such as raw prediction probabilities, as shown in the following code snippet:

   ```
   def debug_model(input_tensor):
       model_input = input_tensor.unsqueeze(0)
       with torch.no_grad():
   ```

```
        model_output = model(model_input)[0]
    model_prediction = model_output.detach().numpy()
    return np.exp(model_prediction)
```

This function is exactly the same as the `run_model` function except that it returns the raw list of probabilities for each digit. The model originally returns the logarithm of softmax outputs because the `log_softmax` layer is used as the final layer in the model (refer to *step 2* of this exercise).

Hence, we need to exponentiate those numbers to return the softmax outputs, which are equivalent to the model's prediction probabilities. Using this debug function, we can look at how the model is performing in more detail, such as whether the probability distribution is flat or has clear peak(s):

```
print(debug_model(input_tensor))
```

This should produce an output similar to the following:

```
[2.8729193e-05 8.9301517e-07 9.9742997e-01 1.4874781e-04 3.4777480e-05
1.3298497e-07 3.3950466e-06 8.8254643e-07 2.3501222e-03 2.3531520e-06]
```

We can see that the third probability in the list is the highest by far, which corresponds with digit 2.

6. Finally, we shall post-process the model prediction so that it can be used by other applications. In our case, we are just going to transform the digit predicted by the model from the integer type into the string type.

   The post-processing step can be more complex in other scenarios, such as speech recognition, where we might want to process the output waveform by smoothing, removing outliers, and so on:

```
def post_process(output):
    return str(output)
```

Because strings are a serializable format, this enables the model predictions to be communicated easily across servers and applications. We can check whether our final post-processed data is as expected:

```
final_output = post_process(output)
print(final_output)
print(type(final_output))
```

This should provide you with the following output:

```
2
<class 'str'>
```

As expected, the output is now of the type string.

This concludes our exercise on loading a saved model architecture, restoring its trained weights, and using the loaded model to generate predictions for sample input data (an image). We loaded a sample image, pre-processed it to transform it into a PyTorch tensor, passed it to the model as input to obtain the model prediction, and post-processed the prediction to generate the final output.

This is a step forward in the direction of serving trained models with a clearly defined input and output interface. In this exercise, the input was an externally provided image file and the output was a generated string containing a digit between 0 and 9. Such a system can be embedded by copying and pasting the provided code into any application that requires the functionality of digitizing hand-written numbers.

In the next section, we will go a level deeper into model serving, where we aim to build a system that can be interacted with by any application to use the digitizing functionality without copying and pasting any code.

Building a basic model server

We have so far built a model inference pipeline that has all the code necessary to independently perform predictions from a pre-trained model. Here, we will work on building our first model server, which is essentially a machine that hosts the model inference pipeline, actively listens to any incoming input data via an interface, and outputs model predictions on any input data through the interface.

Writing a basic app using Flask

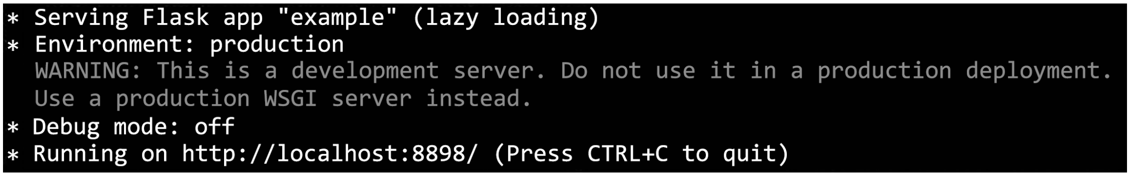
To develop our server, we will use a popular Python library – Flask [3]. **Flask** will enable us to build our model server in a few lines of code. A good example of how this library works is shown with the following code:

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
    return 'Hello, World!'
if __name__ == '__main__':
    app.run(host='localhost', port=8890)
```

Say we saved this Python script as example.py and ran it from the terminal:

```
python example.py
```

It would show the following output in the terminal:



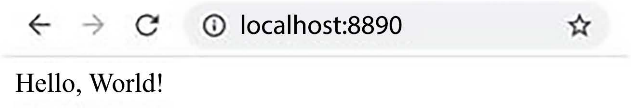
```
* Serving Flask app "example" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://localhost:8898/ (Press CTRL+C to quit)
```

Figure 13.2: Flask example app launch

Basically, it will launch a Flask server that will serve an app called `example`. Let's open a browser and go to the following URL:

```
http://localhost:8890/
```

It will result in the following output in the browser:



Hello, World!

Figure 13.3: Flask example app testing

Essentially, the Flask server is listening to port number `8890` on the IP address `0.0.0.0 (localhost)` at the endpoint `/`. As soon as we input `localhost:8890/` in a browser search bar and press *Enter*, a request is received by this server. The server then runs the `hello_world` function, which in turn returns the string `Hello, World!` as per the function definition provided in `example.py`.

Using Flask to build our model server

Using the principles of running a Flask server demonstrated in the preceding section, we will now use the model inference pipeline built in the previous section to create our first model server. At the end of the exercise, we will launch the server that will listen to incoming requests (image data input).

We will furthermore write another Python script that will make a request to this server by sending the sample image shown in *Figure 13.1*. The Flask server shall run the model inference on this image and output the post-processed predictions.

The full code for this exercise is available on GitHub, including the Flask server code [4] and the client (request-maker) code [5].

Setting up model inference for Flask serving

In this section, we will load a pre-trained model and write the model inference pipeline code:

1. First, we will build the Flask server. And for that, we once again start by importing the necessary libraries:

   ```python
   from flask import Flask, request
   import torch
   ```

Both `flask` and `torch` are vital necessities for this task, besides other basic libraries such as `numpy` and `json`.

2. Next, we will need to define the model class (architecture):

```
class ConvNet(nn.Module):
    def __init__(self):
    def forward(self, x):
```

3. Now that we have the empty model class defined, we can instantiate a model object and load the pre-trained model parameters into this model object as follows:

```
model = ConvNet()
PATH_TO_MODEL = "./convnet.pth"
model.load_state_dict(
    torch.load(PATH_TO_MODEL, map_location="cpu"))
model.eval()
```

4. We will reuse the exact `run_model` function defined in *step 3* of the *Building the inference pipeline* section:

```
def run_model(input_tensor):
    ...
    return model_prediction
```

   As a reminder, this function takes in the tensorized input image and outputs the model prediction, which is any digit between 0 and 9.

5. Next, we will reuse the exact `post_process` function defined in *step 6* of the *Building the inference pipeline* section:

```
def post_process(output):
    return str(output)
```

This will essentially convert the integer output from the `run_model` function into a string.

Building a Flask app to serve model

Having established the inference pipeline in the previous section, we will now build our own Flask app and use it to serve the loaded model:

1. We will instantiate our Flask app as shown in the following line of code:

```
app = Flask(__name__)
```

   This creates a Flask app with the same name as the Python script, which in our case is `server.py`.

2. This is the critical step where we will be defining the endpoint functionality of the Flask server. We will expose a `/test` endpoint and define what happens when a `POST` request is made to that endpoint on the server as follows:

```python
@app.route("/test", methods=["POST"])
def test():
    data = request.files['data'].read()
    md = json.load(request.files['metadata'])
    input_array = np.frombuffer(data, dtype=np.float32)
    input_image_tensor = \
        torch.from_numpy(input_array).view(md["dims"])
    output = run_model(input_image_tensor)
    final_output = post_process(output)
    return final_output
```

   Let's go through the steps one by one:

   1. First, we add a decorator to the function – `test` – which is defined right below the decorator. This decorator tells the Flask app to run this function whenever someone makes a `POST` request to the `/test` endpoint.
   2. Next, we get to define what exactly happens inside the `test` function. First, we read the data and metadata from the `POST` request. Because the data is in serialized form, we need to convert it into a numerical format – we convert it into a `numpy` array. And from a `numpy` array, we swiftly cast it as a PyTorch tensor.
   3. Next, we use the image dimensions provided in the metadata to reshape the tensor.
   4. Finally, we run a forward pass of the model loaded earlier with this tensor. This gives us the model prediction, which is then post-processed and returned by our `test` function.

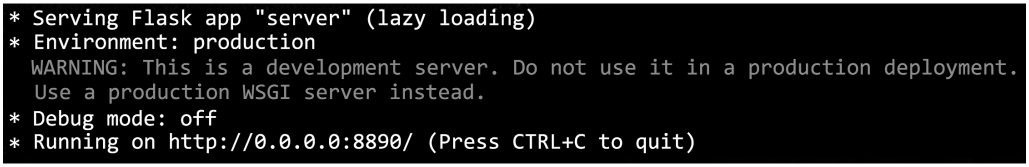
3. We have all the necessary ingredients to launch our Flask app. We will add these last two lines to our `server.py` Python script:

```python
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8890)
```

   This indicates that the Flask server will be hosted at IP address `0.0.0.0` (also known as `localhost`) and port number `8890`. We may now save the Python script and, in a new terminal window, simply execute the following:

```
python server.py
```

   This will run the entire script written in the previous steps and you shall see the following output:



```
* Serving Flask app "server" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://0.0.0.0:8890/ (Press CTRL+C to quit)
```

Figure 13.4: Flask server launch

This looks similar to the example demonstrated in *Figure 13.2*. The only difference is the app name.

Using a Flask server to run predictions

We have successfully launched our model server, which is actively listening to requests. Let's now work on making a request:

1. We will write a separate Python script in the next few steps to do this job. We begin by importing libraries:

   ```
   import requests
   from PIL import Image
   from torchvision import transforms
   ```

   The `requests` library will help us make the actual `POST` request to the Flask server. `Image` helps us to read a sample input image file, and `transforms` will help us to preprocess the input image array.

2. Next, we read an image file:

   ```
   image = Image.open("./digit_image.jpg")
   ```

   The image read here is an RGB image and may have any dimensions (not necessarily 28x28 as expected by the model as input).

3. We now define a preprocessing function that converts the read image into a format that is readable by the model (the same function that we wrote while creating the model's inference pipeline):

   ```
   def image_to_tensor(image):

       return input_image_tensor_norm
   ```

   Having defined the function, we can execute it:

   ```
   image_tensor = image_to_tensor(image)
   ```

   `image_tensor` is what we need to send as input data to the Flask server.

4. Let's now get into packaging our data together to send it over. We want to send the pixel values of the image as well as the shape of the image (28x28) so that the Flask server at the receiving end knows how to reconstruct the stream of pixel values as an image:

   ```
   dimensions = io.StringIO(json.dumps(
       {'dims': list(image_tensor.shape)}))
   data = io.BytesIO(bytearray(image_tensor.numpy()))
   ```

   We stringify the shape of our tensor and convert the image array into bytes to make it all serializable.

5. This is the most critical step in this client code. This is where we actually make the POST request:

```
r = requests.post('http://localhost:8890/test',
                  files={'metadata': dimensions,
                         'data' : data})
```

   Using the requests library, we make the POST request at the URL localhost:8890/test. This is where the Flask server is listening for requests. We send both the actual image data (as bytes) and the metadata (as a string) in the form of a dictionary.

6. The r variable in the preceding code will receive the response to the request from the Flask server. This response should contain the post-processed model prediction. We will now read that output:

```
response = json.loads(r.content)
```

   The response variable will essentially contain what the Flask server outputs, which is a digit between 0 and 9 as a string.

7. We can print the response just to be sure:

```
print("Predicted digit :", response)
```

   At this point, we can save this Python script as make_request.py and execute the following command in the terminal:

```
python make_request.py
```

   This should output the following:

```
Predicted digit : 2
```

Based on the input image (see *Figure 13.1*), the response seems correct. This concludes our current exercise.

Thus, we have successfully built a standalone model server that can render predictions for handwritten-digit images. The same set of steps can easily be extended to any other machine learning model, and so this opens up endless possibilities with regard to creating machine learning applications using PyTorch and Flask.

So far, we have moved from simply writing inference functions to creating model servers that can be hosted remotely and render predictions over the network. In our next and final model serving venture, we will go a level further. You might have noticed that in order to follow the steps in the previous two exercises, there were inherent dependencies to be considered. We are required to install certain libraries, save and load the models at particular locations, read image data, and so on. All of these manual steps slow down the development of a model server.

We can avoid repeating these manual steps every time we need to serve a model by using a microservice. Up next, we will work on creating a model microservice that can be spun up with one command and replicated across several machines.

Creating a model microservice

Imagine you know nothing about training machine learning models but want to use an already trained model without having to get your hands dirty with any PyTorch code. This is where a paradigm such as a machine learning model microservice [6] comes into play.

A machine learning model microservice can be thought of as a black box to which you send input data and it sends back predictions to you. Moreover, it is easy to spin up this black box on a given machine with just a few lines of code. The best part is that it scales effortlessly. You can scale a microservice vertically by using a bigger machine (more memory, more processing power) as well as horizontally, by replicating the microservice across multiple machines.

How do we go about deploying a machine learning model as a microservice? Thanks to the work done using Flask and PyTorch in the previous exercise, we are already a few steps ahead. We have already built a standalone model server using Flask.

In this section, we will take that idea forward and build a standalone model-serving environment using **Docker**. Docker helps containerize software, which essentially means that it helps virtualize the entire **operating system** (**OS**), including software libraries, configuration files, and even data files.

Note

Docker is a huge topic of discussion in itself. However, because this book is focused on PyTorch, we will only cover the basic concepts and usage of Docker for our limited purposes. If you are interested in reading about Docker further, its own documentation is a great place to start [7].

In our case, we have so far used the following libraries in building our model server:

- Python
- PyTorch
- Pillow (for image I/O)
- Flask

And, we have used the following data file:

- Pre-trained model checkpoint file (`convnet.pth`)

We have had to manually arrange for these dependencies by installing the libraries and placing the file in the current working directory. What if we have to redo all of this on a new machine? We would have to manually install the libraries and copy and paste the file once again. This way of working is neither efficient nor foolproof, as we might end up installing different library versions across different machines, for example.

To solve this problem, we would like to create an OS-level blueprint that can be consistently repeated across machines. This is where Docker comes in handy. Docker lets us create that blueprint in the form of a Docker image.

This image can then be built on any empty machine with no assumptions regarding pre-installed Python libraries or an already available model.

Let's actually create such a blueprint using Docker for our digit classification model. In the form of an exercise, we will go from a Flask-based standalone model server to a Docker-based model microservice. Before delving into the exercise, you will need to install Docker [8]:

1. First, we need to list the Python library requirements for our Flask model server. The requirements (with their versions) are as follows:

   ```
   torch==1.5.0
   torchvision==0.5.0
   Pillow==6.2.2
   Flask==1.1.1
   ```

   As a general practice, we will save this list as a text file – `requirements.txt`. This file is also available in our GitHub repository [9]. This list will come in handy for installing the libraries consistently in any given environment.

2. Next, we get straight to the blueprint, which, in Docker terms, will be the `Dockerfile`. A `Dockerfile` is a script that is essentially a list of instructions. The machine where this `Dockerfile` is run needs to execute the listed instructions in the file. This results in a Docker image, and the process is called *building an image*.

   An **image** here is a system snapshot that can be effectuated on any machine, provided that the machine has the minimum necessary hardware resources (for example, installing PyTorch alone requires multiple GBs of disk space).

   Let's look at our `Dockerfile` and try to understand what it does step by step. The full code for the `Dockerfile` is available in our GitHub repository [10]:

   a. The `FROM` keyword instructs Docker to fetch a standard Linux OS with `python 3.8` baked in:

      ```
      FROM python:3.8-slim
      ```

      This ensures that we will have Python installed.

   b. Next, install `wget`, which is a Unix command useful for downloading resources from the internet via the command line:

      ```
      RUN apt-get -qy update && apt-get -q install -y wget
      ```

      The `&&` symbol indicates the sequential execution of commands written before and after the symbol.

c. Here, we are copying two files from our local development environment into this virtual environment:

```
COPY ./server.py ./
COPY ./requirements.txt ./
```

We copy the `requirements` file as discussed in *step 1*, as well as the Flask model server code that we worked on in the previous exercise.

d. Next, we download the pre-trained PyTorch model checkpoint file:

```
RUN wget -q https://github.com/arj7192/MasteringPyTorchV2/raw/
main/Chapter13/convnet.pth
```

This is the same model checkpoint file that we saved in the *Saving and loading a trained model* section of this chapter.

e. Here, we are installing all the relevant libraries listed under `requirements.txt`:

```
RUN pip install -r requirements.txt
```

This `txt` file is the one we wrote in *step 1*.

f. Next, we give `root` access to the Docker client:

```
USER root
```

This step is important in this exercise as it ensures that the client has the credentials to perform all necessary operations on our behalf, such as saving model inference logs on the disk.

> **Note**
>
> In general, though, it is advised not to give root privileges to the client as per the principle of least privilege in data security [11].

g. Finally, we specify that after performing all the previous steps, Docker should execute the `python server.py` command:

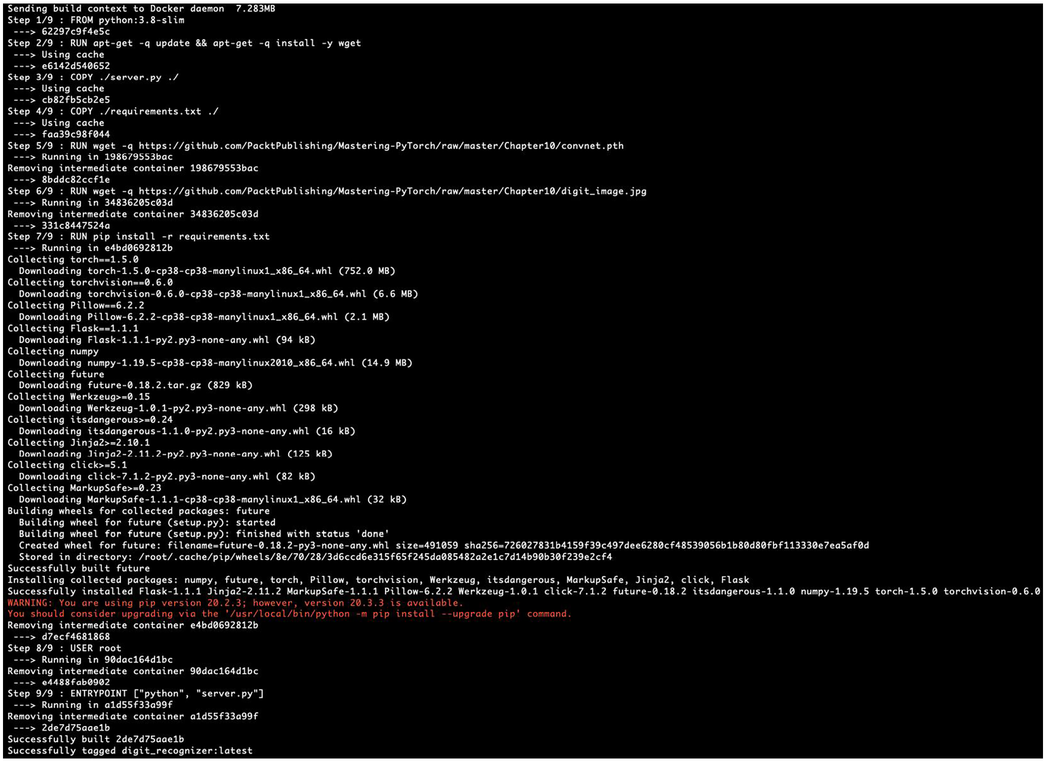
```
ENTRYPOINT ["python", "server.py"]
```

This will ensure the launch of a Flask model server in the virtual machine.

3. Let's now run this `Dockerfile`. In other words, let's build a Docker image using the `Dockerfile` from *step 2*. In the current working directory, on the command line, simply run this:

```
docker build -t digit_recognizer .
```

We are allocating a tag with the name `digit_recognizer` to our Docker image. This should output the following:



```
Sending build context to Docker daemon  7.283MB
Step 1/9 : FROM python:3.8-slim
 ---> 62297c9f4e5c
Step 2/9 : RUN apt-get -q update && apt-get -q install -y wget
 ---> Using cache
 ---> e6142d540652
Step 3/9 : COPY ./server.py ./
 ---> Using cache
 ---> cb82fb5cb2e5
Step 4/9 : COPY ./requirements.txt ./
 ---> Using cache
 ---> faa39c98f044
Step 5/9 : RUN wget -q https://github.com/PacktPublishing/Mastering-PyTorch/raw/master/Chapter10/convnet.pth
 ---> Running in 198679553bac
Removing intermediate container 198679553bac
 ---> 8bddc82ccf1e
Step 6/9 : RUN wget -q https://github.com/PacktPublishing/Mastering-PyTorch/raw/master/Chapter10/digit_image.jpg
 ---> Running in 34836205c03d
Removing intermediate container 34836205c03d
 ---> 331c8447524a
Step 7/9 : RUN pip install -r requirements.txt
 ---> Running in e4bd0692812b
Collecting torch==1.5.0
  Downloading torch-1.5.0-cp38-cp38-manylinux1_x86_64.whl (752.0 MB)
Collecting torchvision==0.6.0
  Downloading torchvision-0.6.0-cp38-cp38-manylinux1_x86_64.whl (6.6 MB)
Collecting Pillow==6.2.2
  Downloading Pillow-6.2.2-cp38-cp38-manylinux1_x86_64.whl (2.1 MB)
Collecting Flask==1.1.1
  Downloading Flask-1.1.1-py2.py3-none-any.whl (94 kB)
Collecting numpy
  Downloading numpy-1.19.5-cp38-cp38-manylinux2010_x86_64.whl (14.9 MB)
Collecting future
  Downloading future-0.18.2.tar.gz (829 kB)
Collecting Werkzeug>=0.15
  Downloading Werkzeug-1.0.1-py2.py3-none-any.whl (298 kB)
Collecting itsdangerous>=0.24
  Downloading itsdangerous-1.1.0-py2.py3-none-any.whl (16 kB)
Collecting Jinja2>=2.10.1
  Downloading Jinja2-2.11.2-py2.py3-none-any.whl (125 kB)
Collecting click>=5.1
  Downloading click-7.1.2-py2.py3-none-any.whl (82 kB)
Collecting MarkupSafe>=0.23
  Downloading MarkupSafe-1.1.1-cp38-cp38-manylinux1_x86_64.whl (32 kB)
Building wheels for collected packages: future
  Building wheel for future (setup.py): started
  Building wheel for future (setup.py): finished with status 'done'
  Created wheel for future: filename=future-0.18.2-py3-none-any.whl size=491059 sha256=726027831b4159f39c497dee6280cf48539056b1b80d80fbf113330e7ea5af0d
  Stored in directory: /root/.cache/pip/wheels/8e/70/28/3d6ccd6e315f65f245da085482a2e1c7d14b90b30f239e2cf4
Successfully built future
Installing collected packages: numpy, future, torch, Pillow, torchvision, Werkzeug, itsdangerous, MarkupSafe, Jinja2, click, Flask
Successfully installed Flask-1.1.1 Jinja2-2.11.2 MarkupSafe-1.1.1 Pillow-6.2.2 Werkzeug-1.0.1 click-7.1.2 future-0.18.2 itsdangerous-1.1.0 numpy-1.19.5 torch-1.5.0 torchvision-0.6.0
WARNING: You are using pip version 20.2.3; however, version 20.3.3 is available.
You should consider upgrading via the '/usr/local/bin/python -m pip install --upgrade pip' command.
Removing intermediate container e4bd0692812b
 ---> d7ecf4681868
Step 8/9 : USER root
 ---> Running in 90dac164d1bc
Removing intermediate container 90dac164d1bc
 ---> e4488fab0902
Step 9/9 : ENTRYPOINT ["python", "server.py"]
 ---> Running in a1d55f33a99f
Removing intermediate container a1d55f33a99f
 ---> 2de7d75aae1b
Successfully built 2de7d75aae1b
Successfully tagged digit_recognizer:latest
```

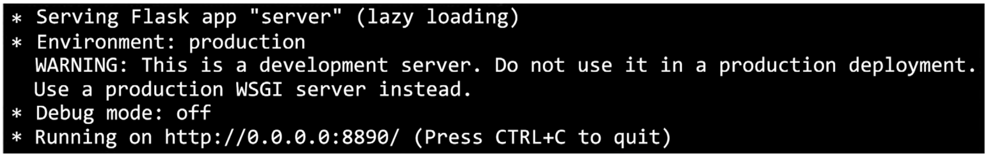
Figure 13.5: Building a Docker image

Figure 13.5 shows the sequential execution of the steps mentioned in *step 2*. Running this step might take a while, depending on your internet connection, as it downloads the entire PyTorch library, among others, to build the image.

4. At this stage, we already have a Docker image with the name `digit_recognizer`. We are all set to deploy this image on any machine. In order to deploy the image on your own machine for now, just run the following command:

```
docker run -p 8890:8890 digit_recognizer
```

With this command, we are essentially starting a virtual machine inside our machine using the `digit_recognizer` Docker image. Because our original Flask model server was designed to listen to port `8890`, we have forwarded our actual machine's port `8890` to the virtual machine's port `8890` using the `-p` argument. Running this command should output this:



```
* Serving Flask app "server" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://0.0.0.0:8890/ (Press CTRL+C to quit)
```

Figure 13.6: Running a Docker instance

The preceding screenshot is remarkably similar to *Figure 13.4* from the previous exercise, which is no surprise because the Docker instance is running the same Flask model server that we manually ran in our previous exercise.

5. We can now test whether our Dockerized Flask model server (model microservice) works as expected by using it to make model predictions. We will once again use the `make_request.py` file used in the previous exercise to send a prediction request to our model. From the current local working directory, simply execute this:

   ```
   python make_request.py
   ```

   This should output the following:

   ```
   Predicted digit : 2
   ```

   The microservice seems to be doing the job, and thus we have successfully built and tested our own machine learning model microservice using Python, PyTorch, Flask, and Docker.

6. Upon successful completion of the preceding steps, you can close the launched Docker instance from *step 4* by pressing *Ctrl* + *C* as indicated in *Figure 13.6*. And once the running Docker instance is stopped, you can delete the instance by running the following command:

   ```
   docker rm $(docker ps -a -q | head -1)
   ```

   This command basically removes the most recent inactive Docker instance, which in our case is the Docker instance that we just stopped.

7. Finally, you can also delete the Docker image that we built in *step 3* by running the following command:

   ```
   docker rmi $(docker images -q "digit_recognizer")
   ```

This will basically remove the image that has been tagged with the `digit_recognizer` tag.

This concludes our section on serving models written in PyTorch. We started off by designing a local model inference system. We took this inference system and wrapped a Flask-based model server around it to create a standalone model serving system.

Finally, we used the Flask-based model server inside a Docker container to essentially create a model-serving microservice. Using both the theory as well as the exercises discussed in this section, you should be able to get started with hosting/serving your trained models across different use cases, system configurations, and environments.

In the next section, we will stay with the model-serving theme but will discuss a particular tool that has been developed precisely to serve PyTorch models: **TorchServe**. We will also do a quick exercise to demonstrate how to use this tool.

Serving a PyTorch model using TorchServe

TorchServe, released in April 2020, is a dedicated PyTorch model-serving framework. Using the functionalities offered by TorchServe, we can serve multiple models at the same time with low prediction latency and without having to write much custom code. Furthermore, TorchServe offers features such as model versioning, metrics monitoring, and data preprocessing and post-processing.

This clearly makes TorchServe a more advanced model-serving alternative than the model microservice we developed in the previous section. However, making custom model microservices still proves to be a powerful solution for complicated machine learning pipelines (which is more common than we might think).

In this section, we will continue working with our handwritten digit classification model and demonstrate how to serve it using TorchServe. After reading this section, you should be able to get started with TorchServe and go further with utilizing its full set of features.

Installing TorchServe

Before starting with the exercise, we will need to install Java 11 SDK as a requirement. For Linux OS, run the following:

```
sudo apt-get install openjdk-11-jdk
```

And for macOS, we need to run the following command on the command line:

```
brew tap AdoptOpenJDK/openjdk
brew install --cask adoptopenjdk11
```

And thereafter, we need to install `torchserve` by running this:

```
pip install torchserve==0.6.0 torch-model-archiver==0.6.0
```

For detailed installation instructions, refer to the TorchServe documentation [12]. And if you are working with Windows, you should be able to install TorchServe from the instructions available here [13].

Notice that we also install a library called `torch-model-archiver` [14]. This archiver aims to create one model file that will contain both the model parameters as well as the model architecture definition in an independent serialized format as a `.mar` file.

Launching and using a TorchServe server

Now that we have installed everything that we need, we can start putting together our existing code from the previous exercises to serve our model using TorchServe. We will hereon go through a number of steps in the form of an exercise:

1. First, we will place the existing model architecture code in a model file saved as `convnet.py`:

```
==========================convnet.py========================
import torch
import torch.nn as nn
import torch.nn.functional as F
class ConvNet(nn.Module):
    def __init__(self):
        ...
    def forward(self, x):
        ...
```

We will need this model file as one of the inputs to `torch-model-archiver` to produce a unified `.mar` file. You can find the full model file in our GitHub repository [15].

Remember we discussed the three parts of any model inference pipeline: data pre-processing, model prediction, and post-processing. TorchServe provides *handlers*, which handle the pre-processing and post-processing parts of popular kinds of machine learning tasks: `image_classifier`, `image_segmenter`, `object_detector`, and `text_classifier`.

This list might grow in the future as TorchServe is actively being developed at the time of writing this book.

2. For our task, we will create a custom image handler that is inherited from the default `Image_classifier` handler. We choose to create a custom handler because, as opposed to the usual image classification models that deal with color (RGB) images, our model deals with grayscale images of a specific size (28x28 pixels). The following is the code for our custom handler, which you can also find in our GitHub repository [16]:

```
========================convnet_handler.py==================
from torchvision import transforms
from ts.torch_handler.image_classifier import ImageClassifier
class ConvNetClassifier(ImageClassifier):
    image_processing = transforms.Compose([
        transforms.Grayscale(), transforms.Resize((28, 28)),
        transforms.ToTensor(), transforms.Normalize(
            (0.1302,), (0.3069,))])
    def postprocess(self, output):
        return output.argmax(1).tolist()
```

First, we imported the `image_classifer` default handler, which will provide most of the basic image classification inference pipeline handling capabilities. Next, we inherit the `ImageClassifer` handler class to define our custom `ConvNetClassifier` handler class.

There are two blocks of custom code:

- The data pre-processing step, where we apply a sequence of transformations to the data exactly as we did in *step 3* of the *Building the inference pipeline* section.
- The postprocessing step, defined under the `postprocess` method, where we extract the predicted class label from the list of prediction probabilities of all classes.

3. We already produced a `convnet.pth` file in the *Saving and loading a trained model* section of this chapter while creating the model inference pipeline. Using `convnet.py`, `convnet_handler.py`, and `convnet.pth`, we can finally create the `.mar` file using `torch-model-archiver` by running the following command:

   ```
   torch-model-archiver --model-name convnet --version 1.0 --model-file ./
   convnet.py --serialized-file ./convnet.pth --handler ./convnet_handler.py
   ```

   This command should result in a `convnet.mar` file being written to the current working directory. We have specified a `model_name` argument, which names the `.mar` file. We have specified a `version` argument, which will be helpful in model versioning while working with multiple variations of a model at the same time.

   We have located where our `convnet.py` (for model architecture), `convnet.pth` (for model weights), and `convnet_handler.py` (for pre- and post-processing) files are, using the `model_file`, `serialzed_file`, and `handler` arguments, respectively.

4. Next, we need to create a new directory in the current working directory and move the `convnet.mar` file created in *step 3* to that directory by running the following on the command line:

   ```
   mkdir model_store
   mv convnet.mar model_store/
   ```

   We have to do so to follow the design requirements of the TorchServe framework.

5. Finally, we may launch our model server using TorchServe. On the command line, simply run the following:

   ```
   torchserve --start --ncs --model-store model_store --models convnet.mar
   ```

   This will silently start the model inference server and you will see some logs on the screen, including the following:

   ```
   Number of GPUs: 0
   Number of CPUs: 8
   Max heap size: 4096 M
   Python executable: /Users/ashish.jha/opt/anaconda/bin/python
   Config file: N/A
   Inference address: http://127.0.0.1:8080
   Management address: http://127.0.0.1:8081
   Metrics address: http://127.0.0.1:8082
   ```

   As you can see, TorchServe investigates the available devices on the machine, among other details. It allocates three separate URLs for *inference*, *management*, and *metrics*. To check whether the launched server is indeed serving our model, we can ping the management server with the following command:

   ```
   curl http://localhost:8081/models
   ```

This should output the following:

```
{
    "models": [
        {
            "modelName": "convnet",
            "modelUrl": "convnet.mar"
        }
    ]
}
```

This verifies that the TorchServe server is indeed hosting the model.

6. Finally, we can test our TorchServe model server by making an inference request. This time, we won't need to write a Python script because the handler will already take care of processing any input image file. So, we can directly make a request using the `digit_image.jpg` sample image file by running this:

   ```
   curl http://127.0.0.1:8080/predictions/convnet -T ./digit_image.jpg
   ```

   This should output `2` in the terminal, which is indeed the correct prediction, as evident from *Figure 13.1*.

7. Finally, once we are done with using the model server, it can be stopped by running the following on the command line:

   ```
   torchserve --stop
   ```

This concludes our exercise on how to use TorchServe to spin up our own PyTorch model server and use it to make predictions. There is a lot more to unpack here, such as model monitoring (metrics), logging, versioning, benchmarking, and so on. TorchServe's website is a great place to pursue these advanced topics in detail [17].

After finishing this section, you should be able to use TorchServe to serve your own models. I encourage you to write custom handlers for your own use cases, explore the various TorchServe configuration settings [18], and try out other advanced features of TorchServe [19].

> **Note**
>
> TorchServe is constantly evolving, with a lot of promise. My advice would be to keep an eye on the rapid updates in this territory of PyTorch.

In the next section, we will take a look at exporting PyTorch models so that they can be used in different environments, programming languages, and deep learning libraries.

Exporting universal PyTorch models using TorchScript and ONNX

We have discussed serving PyTorch models extensively in the previous sections of this chapter, which is perhaps the most critical aspect of operationalizing PyTorch models in production systems. In this section, we will look at another important aspect – exporting PyTorch models. We have already learned how to save PyTorch models and load them back from disk in the classic Python scripting environment. But we need more ways of exporting PyTorch models. Why?

Well, for starters, the Python interpreter allows only one thread to run at a time using the **global interpreter lock** (**GIL**). This keeps us from parallelizing operations. Secondly, Python might not be supported on every system or device on which we might want to run our models. To address these problems, PyTorch offers support for exporting its models in an efficient format and in a platform- or language-agnostic manner such that a model can be run in environments different from the one it was trained in.

We will first explore TorchScript, which enables us to export serialized and optimized PyTorch models into an Intermediate Representation that can then be run in a Python-independent program (say, a C++ program).

And then, we will look at ONNX and how it lets us save PyTorch models in a universal format that can then be loaded into other deep learning frameworks and different programming languages.

Understanding the utility of TorchScript

There are two key reasons why TorchScript is a vital tool when it comes to putting PyTorch models into production:

- PyTorch works on an eager execution basis, as discussed in *Chapter 1*, *Overview of Deep Learning Using PyTorch*. This has its advantages, such as easier debugging. However, executing steps/operations one by one by writing and reading intermediate results to and from memory may lead to high inference latency as well as limiting us in terms of overall operational optimizations. To tackle this problem, PyTorch provides its own **just-in-time** (**JIT**) compiler, which is based on the PyTorch-centered parts of Python.

  The JIT compiler compiles PyTorch models instead of interpreting them, which is equivalent to creating one composite graph for the entire model by looking at all of its operations at once. The JIT-compiled code is TorchScript code, which is basically a statically typed subset of Python. This compilation leads to several performance improvements and optimizations, such as getting rid of the GIL and thereby enabling multithreading.

- PyTorch is essentially built to be used with the Python programming language. Remember, we have used Python in almost the entirety of this book too. However, when it comes to productionizing models, there are more performant (that is, quicker and more memory-efficient) languages than Python, such as C++. And also, we might want to deploy our trained models on systems or devices that do not work with Python.

This is where TorchScript kicks in. As soon as we compile our PyTorch code into TorchScript code, which is an Intermediate Representation of our PyTorch model, we can serialize this representation into a C++-friendly format using the TorchScript compiler. Thereafter, this serialized file can be read in a C++ model inference program using LibTorch – the PyTorch C++ API.

We have mentioned the JIT compilation of PyTorch models several times in this section. Let's now look at two of the possible options for compiling our PyTorch models into the TorchScript format.

Model tracing with TorchScript

One way of translating PyTorch code into TorchScript is tracing the PyTorch model. Tracing requires the PyTorch model object, along with a dummy example input to the model. As the name suggests, the tracing mechanism traces the flow of this dummy input through the model (neural network), records the various operations, and renders a TorchScript **Intermediate Representation** (**IR**), which can be visualized both as a graph as well as TorchScript code.

We will now walk through the steps involved in tracing a PyTorch model using our handwritten digit classification model. The full code for this exercise is available in our GitHub repository [20].

The first five steps of this exercise are the same as the steps in the *Saving and loading a trained model* and *Building the inference pipeline* sections, where we built the model inference pipeline:

1. We will start with importing libraries by running the following code:

   ```
   import torch
   ...
   ```

2. Next, we will define and instantiate the `model` object:

   ```
   class ConvNet(nn.Module):
       def __init__(self):
           ...
       def forward(self, x):
           ...
   model = ConvNet()
   ```

3. Next, we will restore the model weights using the following lines of code:

   ```
   PATH_TO_MODEL = "./convnet.pth"
   model.load_state_dict(torch.load(PATH_TO_MODEL, map_location="cpu"))
   model.eval()
   ```

4. We then load a sample image:

   ```
   image = Image.open("./digit_image.jpg")
   ```

5. Next, we define the data pre-processing function (the same function that we wrote while creating the model's inference pipeline):

   ```
   def image_to_tensor(image):

       ...
       return input_image_tensor_norm
   ```

   And we then apply the pre-processing function to the sample image:

   ```
   input_tensor = image_to_tensor(image)
   ```

6. In addition to the code under *step 3*, we also execute the following lines of code:

   ```
   for p in model.parameters():
       p.requires_grad_(False)
   ```

   If we do not do this, all of the traced model's parameters will require gradients and we will have to load the model within the `torch.no_grad()` context.

7. We already have the loaded PyTorch model object with pre-trained weights. We are ready to trace the model with a dummy input as shown next:

   ```
   demo_input = torch.ones(1, 1, 28, 28)
   traced_model = torch.jit.trace(model, demo_input)
   ```

   The dummy input is an image with all pixel values set to `1`.

8. We can now look at the traced model graph by running this:

   ```
   print(traced_model.graph)
   ```

   This should output the following:

   ```
   graph(%self.1 : __torch__.ConvNet,
           %x.1 : Float(1, 1, 28, 28, strides=[784, 784, 28, 1], requires_
   grad=0, device=cpu)):
     %fc2 : __torch__.torch.nn.modules.linear.___torch_mangle_2.Linear =
   prim::GetAttr[name="fc2"](%self.1)
       %dp2 : __torch__.torch.nn.modules.dropout.___torch_mangle_1.Dropout2d
   = prim::GetAttr[name="dp2"](%self.1)
   ...
       %95 : NoneType = prim::Constant()
       %96 : Float(1, 10, strides=[10, 1], requires_grad=0, device=cpu) =
   aten::log_softmax(%128, %94, %95) # /Users/ashish.jha/opt/anaconda3/
   envs/mastering_pytorch_7_chaps/lib/python3.9/site-packages/torch/nn/
   functional.py:1923:0
   return (%96)
   ```

Intuitively, the first few lines in the graph show the initialization of layers of this model, such as fc2, dp2, and so on. Toward the end, we see the last layer, that is, the softmax layer. Evidently, the graph is written in a lower-level language with statically typed variables and closely resembles the TorchScript language.

9. Besides the graph, we can also look at the exact TorchScript code behind the traced model by running this:

   ```
   print(traced_model.code)
   ```

   This should output the following lines of Python-like code, which define the forward pass method for the model:

   ```
   def forward(self, x: Tensor) -> Tensor:
       fc2 = self.fc2
       dp2 = self.dp2
       fc1 = self.fc1
       dp1 = self.dp1
       cn2 = self.cn2
       cn1 = self.cn1
       input = torch.relu((cn1).forward(x, ))
       input0 = torch.relu((cn2).forward(input, ))
       input1 = torch.max_pool2d(
           input0, [2, 2], annotate(List[int], []),
           [0, 0], [1, 1])
       input2 = torch.flatten((dp1).forward(input1, ), 1)
       input3 = torch.relu((fc1).forward(input2, ))
       _0 = (fc2).forward((dp2).forward(input3, ), )
       return torch.log_softmax(_0, 1)
   ```

   This is precisely the TorchScript equivalent for the code that we wrote using PyTorch in *step 2*.

10. Next, we will export or save the traced model:

    ```
    torch.jit.save(traced_model, 'traced_convnet.pt')
    ```

11. Now we load the saved model:

    ```
    loaded_traced_model = torch.jit.load('traced_convnet.pt')
    ```

    Note that we didn't need to load the model architecture and parameters separately.

12. Finally, we can use this model for inference:

    ```
    loaded_traced_model(input_tensor.unsqueeze(0))
    ```

The output is as follows:

```
tensor([[-1.0458e+01, -1.3929e+01, -2.5733e-03, -8.8133e+00, -1.0267e+01,
-1.5833e+01, -1.2593e+01, -1.3940e+01, -6.0533e+00, -1.2960e+01]])
```

We can check these results by re-running model inference on the original model:

```
model(input_tensor.unsqueeze(0))
```

This should produce the same output as in *step 12*, which verifies that our traced model is working properly.

You can use the traced model instead of the original PyTorch model object to build more efficient Flask model servers and Dockerized model microservices, thanks to the GIL-free nature of TorchScript. While tracing is a viable option for JIT-compiling PyTorch models, it has some drawbacks.

For instance, if the forward pass of the model consists of control flows such as `if` and `for` statements, then the tracing will only render one of the multiple possible paths in the flow. In order to accurately translate PyTorch code into TorchScript code for such scenarios, we will use another compilation mechanism, called scripting.

Model scripting with TorchScript

Please follow *steps 1* to *6* from the previous exercise and then follow up with the steps given in this exercise. The full code is available in our GitHub repository [21]:

1. For scripting, we need not provide any dummy input to the model, and the following line of code transforms PyTorch code into TorchScript code directly:

   ```
   scripted_model = torch.jit.script(model)
   ```

2. Let's look at the scripted model graph by running the following line of code:

   ```
   print(scripted_model.graph)
   ```

   This should output the scripted model graph in a similar fashion to the traced model graph, as shown in the following figure:

   ```
   graph(%self : __torch__.ConvNet,
     %x.1 : Tensor):
     %51 : Function = prim::Constant[name="log_softmax"]()
     %49 : int = prim::Constant[value=3]()
     %33 : int = prim::Constant[value=-1]()
     %26 : Function = prim::Constant[name="_max_pool2d"]()
   ...
     %fc2 : __torch__.torch.nn.modules.linear.___torch_mangle_2.Linear =
   prim::GetAttr[name="fc2"](%self)
   ```

```
  %x.45 : Tensor = prim::CallMethod[name="forward"](%fc2, %x.41) # /var/
folders/gs/mjlw0j210yz02z4yrv9gshdm0000gq/T/ipykernel_13610/2721400238.
py:22:12
  %op.1 : Tensor = prim::CallFunction(%51, %x.45, %32, %49, %19) # /var/
folders/gs/mjlw0j210yz02z4yrv9gshdm0000gq/T/ipykernel_13610/2721400238.
py:23:13
  return (%op.1)
```

Once again, we can see a similar, verbose, low-level script that lists the various edges of the graph per line. Notice that the graph here is not the same as in *step 8* of the previous exercise, which indicates differences in code compilation strategy when using tracing rather than scripting.

3. We can also look at the equivalent TorchScript code by running this:

```
print(scripted_model.code)
```

This should output the following:

```
def forward(self,x: Tensor) -> Tensor:
    _0 = __torch__.torch.nn.functional._max_pool2d
    _1 = __torch__.torch.nn.functional.log_softmax
    cn1 = self.cn1
    x0 = (cn1).forward(x, )
    x1 = __torch__.torch.nn.functional.relu(x0, False, )
    cn2 = self.cn2
    x2 = (cn2).forward(x1, )
    x3 = __torch__.torch.nn.functional.relu(x2, False, )
    x4 = _0(x3, [2, 2], None, [0, 0],
            [1, 1], False, False, )
    dp1 = self.dp1
    x5 = (dp1).forward(x4, )
    x6 = torch.flatten(x5, 1)
    fc1 = self.fc1
    x7 = (fc1).forward(x6, )
    x8 = __torch__.torch.nn.functional.relu(x7, False, )
    dp2 = self.dp2
    x9 = (dp2).forward(x8, )
    fc2 = self.fc2
    x10 = (fc2).forward(x9, )
    return _1(x10, 1, 3, None, )
```

In essence, the flow is similar to that in *step 9* of the previous exercise; however, there are subtle differences in the code signature resulting from differences in compilation strategy.

4. Once again, the scripted model can be exported and loaded back in the following way:

```
torch.jit.save(scripted_model, 'scripted_convnet.pt')
loaded_scripted_model = \
    torch.jit.load('scripted_convnet.pt')
```

5. Finally, we use the scripted model for inference using this:

```
loaded_scripted_model(input_tensor.unsqueeze(0))
```

This should produce the exact same results as in *step 12* of the previous exercise, which verifies that the scripted model is working as expected.

Similar to tracing, a scripted PyTorch model is GIL-free and hence can improve model serving performance when used with Flask or Docker. *Table 13.1* shows a quick comparison between the model tracing and scripting approaches:

Tracing	Scripting
• Dummy input is needed. • Records a fixed sequence of mathematical operations by passing the dummy input to the model. • Cannot handle multiple control flows (if-else) within the model forward pass. • Works even if the model has PyTorch functionalities that are not supported by TorchScript (`https://pytorch.org/docs/stable/jit_unsupported.html`).	• No need for dummy input. • Generates TorchScript code/graph by inspecting the `nn.Module` contents within the PyTorch code. • Useful in handling all types of control flows. • Scripting can work only if the PyTorch model does not contain any functionalities which are not supported by TorchScript.

Table 13.1: Tracing versus scripting

We have so far demonstrated how PyTorch models can be translated and serialized as TorchScript models. In the next section, we will completely get rid of Python for a moment and demonstrate how to load the TorchScript serialized model using C++.

Running a PyTorch model in C++

Python can sometimes be limiting, or we might be unable to run machine learning models trained using PyTorch and Python. In this section, we will use the serialized TorchScript model objects (using tracing and scripting) that we exported in the previous section to run model inferences inside C++ code.

Note

Basic working knowledge of C++ is assumed for this section. You can read up on C++ basics here [22]. This section specifically talks a lot about C++ code compilation. You can get a refresher on C++ code compilation concepts here [23].

For this exercise, we need to install CMake, following the steps mentioned in [24], to be able to build the C++ code. After that, we will create a folder named `cpp_convnet` in the current working directory and work from that directory:

1. Let's get straight into writing the C++ file that will run the model inference pipeline. The full C++ code is available here in our GitHub repository [25]:

   ```
   #include <torch/script.h>
   ...
   int main(int argc, char **argv) {
       Mat img = imread(argv[2], IMREAD_GRAYSCALE);
   ```

   First, the `.jpg` image file is read as a grayscale image using the OpenCV library. You will need to install the OpenCV library for C++ as per your OS requirements – Mac [26], Linux [27], or Windows [28].

2. The grayscale image is then resized to 28x28 pixels as that is the requirement for our CNN model:

   ```
   resize(img, img, Size(28, 28));
   ```

3. The image array is then converted to a PyTorch tensor:

   ```
   auto input_ = torch::from_blob(img.data, { img.rows, img.cols, img.
   channels() }, at::kByte);
   ```

   For all `torch`-related operations as in this step, we use the `libtorch` library, which is the home for all `torch` C++-related APIs. If you have PyTorch installed, you need not install LibTorch separately.

4. Because OpenCV reads the grayscale in (28, 28, 1) dimensions, we need to turn it around into (1, 28, 28) to suit the PyTorch requirements. The tensor is then reshaped into shape (1,1,28,28), where the first *1* is `batch_size` for inference and the second `1` is the number of channels, which is `1` for grayscale:

   ```
       auto input = input_.permute({2,0,1}).unsqueeze_(0).reshape({1, 1,
   img.rows, img.cols}).toType(c10::kFloat).div(255);
       input = (input - 0.1302) / 0.3069;
   ```

   Because OpenCV read images have pixel values ranging from `0` to `255`, we normalize these values within a range of `0` to `1`. Thereafter, we standardize the image with mean `0.1302` and `std` `0.3069`, as we did in a previous section (see *step 2* of the *Building the inference pipeline* section).

5. In this step, we load the JIT-ed TorchScript model object that we exported in the previous exercise:

   ```
       auto module = torch::jit::load(argv[1]);
       std::vector<torch::jit::IValue> inputs;
       inputs.push_back(input);
   ```

6. Finally, we come to the model prediction, where we use the loaded model object to make a forward pass with the supplied input data (an image, in this case):

   ```
   auto output_ = module.forward(inputs).toTensor();
   ```

   The `output_` variable contains a list of probabilities for each class. Let's extract the class label with the highest probability and print it:

   ```
   auto output = output_.argmax(1);
   cout << output << '\n';
   ```

   Finally, we successfully exit the C++ routine:

   ```
       return 0;
   }
   ```

7. While *steps 1* to *6* cover the various parts of our C++ code, we also need to write a `CMakeLists.txt` file in the same working directory. The full code for this file is available in our GitHub repository [29]:

   ```
   cmake_minimum_required(VERSION 3.0 FATAL_ERROR)
   project(cpp_convnet)
   find_package(Torch REQUIRED)
   find_package(OpenCV REQUIRED)
   add_executable(cpp_convnet cpp_convnet.cpp)
   ...
   ```

   This file is basically the library installation and building script, similar to `setup.py` in a Python project. In addition to this code, the `OpenCV_DIR` environment variable needs to be set to the path where the OpenCV build artifacts are created, shown in the following code block:

   ```
   export OpenCV_DIR=/Users/ashish.jha/code/personal/MasteringPyTorchV2/
   Chapter13/cpp_convnet/build_opencv/
   ```

8. Next, we need to actually run the `CMakeLists` file to create build artifacts. We do so by creating a new directory in the current working directory and running the build process from there. On the command line, we simply need to run the following:

   ```
   mkdir build
   cd build
   cmake -DCMAKE_PREFIX_PATH=/Users/ashish.jha/opt/anaconda3/envs/mastering_
   pytorch/lib/python3.9/site-packages/torch/share/cmake/ ..
   cmake --build . --config Release
   ```

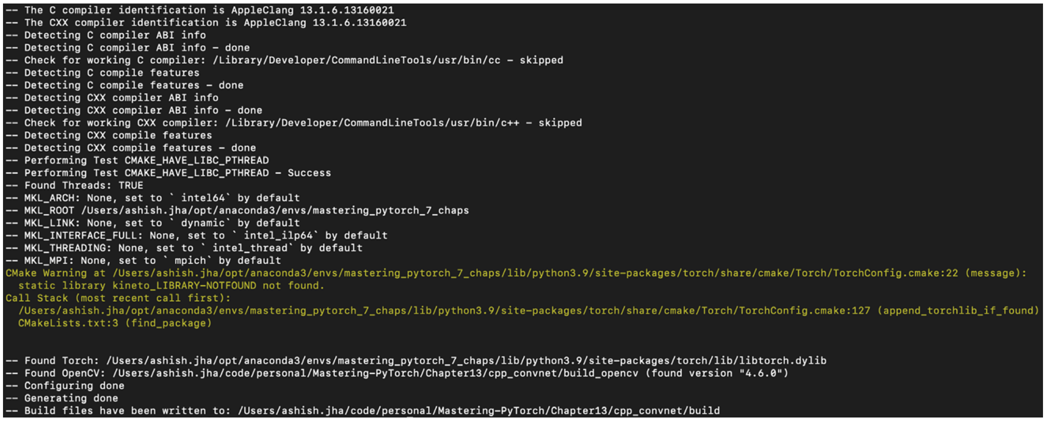
   In the third line, you shall provide the path to LibTorch. To find your own, open Python and execute this:

   ```
   import torch; torch.__path__
   ```

For me, it outputs this:

```
['/Users/ashish.jha/opt/anaconda3/envs/mastering_pytorch/lib/python3.9/
site-packages/torch']_
```

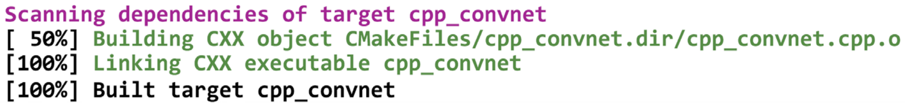
Executing the third line shall output the following:



```
-- The C compiler identification is AppleClang 13.1.6.13160021
-- The CXX compiler identification is AppleClang 13.1.6.13160021
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /Library/Developer/CommandLineTools/usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /Library/Developer/CommandLineTools/usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Success
-- Found Threads: TRUE
-- MKL_ARCH: None, set to ` intel64` by default
-- MKL_ROOT /Users/ashish.jha/opt/anaconda3/envs/mastering_pytorch_7_chaps
-- MKL_LINK: None, set to ` dynamic` by default
-- MKL_INTERFACE_FULL: None, set to ` intel_ilp64` by default
-- MKL_THREADING: None, set to ` intel_thread` by default
-- MKL_MPI: None, set to ` mpich` by default
CMake Warning at /Users/ashish.jha/opt/anaconda3/envs/mastering_pytorch_7_chaps/lib/python3.9/site-packages/torch/share/cmake/Torch/TorchConfig.cmake:22 (message):
  static library kineto_LIBRARY-NOTFOUND not found.
Call Stack (most recent call first):
  /Users/ashish.jha/opt/anaconda3/envs/mastering_pytorch_7_chaps/lib/python3.9/site-packages/torch/share/cmake/Torch/TorchConfig.cmake:127 (append_torchlib_if_found)
  CMakeLists.txt:3 (find_package)


-- Found Torch: /Users/ashish.jha/opt/anaconda3/envs/mastering_pytorch_7_chaps/lib/python3.9/site-packages/torch/lib/libtorch.dylib
-- Found OpenCV: /Users/ashish.jha/code/personal/Mastering-PyTorch/Chapter13/cpp_convnet/build_opencv (found version "4.6.0")
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/ashish.jha/code/personal/Mastering-PyTorch/Chapter13/cpp_convnet/build
```

Figure 13.7: The C++ CMake output

And the fourth line should result in this:



```
Scanning dependencies of target cpp_convnet
[ 50%] Building CXX object CMakeFiles/cpp_convnet.dir/cpp_convnet.cpp.o
[100%] Linking CXX executable cpp_convnet
[100%] Built target cpp_convnet
```

Figure 13.8: C++ model building

9. Upon successful execution of the previous step, we will have produced a C++ compiled binary with the name `cpp_convnet`. It is now time to execute this binary program. In other words, we can now supply a sample image to our C++ model for inference. We may use the scripted model as input:

   ```
   ./cpp_convnet ../../scripted_convnet.pt ../../digit_image.jpg
   ```

   Alternatively, we may use the traced model as input:

   ```
   ./cpp_convnet ../../traced_convnet.pt ../../digit_image.jpg
   ```

   Either of these should result in the following output:

   ```
   2
   [ CPULongType{1} ]
   ```

According to *Figure 13.1*, the C++ model seems to be working correctly. Because we have used a different image handling library in C++ (that is, OpenCV) as compared to in Python (PIL), the pixel values are slightly differently encoded, which will result in slightly different prediction probabilities, but the final model prediction in the two languages should not differ significantly if correct normalizations are applied.

This concludes our exploration of PyTorch model inference using C++. This exercise shall help you get started with transporting your favorite deep learning models written and trained using PyTorch into a C++ environment, which should make predictions more efficient, as well as opening up the possibility of hosting models in Python-less environments (for example, certain embedded systems, drones, and so on).

In the next section, we will move away from TorchScript and discuss a universal neural network modeling format – ONNX – that has enabled model usage across deep learning frameworks, programming languages, and OSes. We will work on loading a PyTorch-trained model for inference in TensorFlow.

Using ONNX to export PyTorch models

There are scenarios in production systems where most of the already deployed machine learning models are written in a certain deep learning library, say, TensorFlow, with its own sophisticated model-serving infrastructure. However, if a certain model is written using PyTorch, we would like it to be runnable using TensorFlow to conform to the serving strategy. This is one among various other use cases where a framework such as ONNX is useful.

ONNX is a universal format where the essential operations of a deep learning model such as matrix multiplications and activations, written differently in different deep learning libraries, are standardized. It enables us to interchangeably use different deep learning libraries, programming languages, and even operating environments to run the same deep learning model.

Here, we will demonstrate how to run a model, trained using PyTorch, in TensorFlow. We will first export the PyTorch model into ONNX format and then load the ONNX model inside the TensorFlow code.

We will work with `tensorflow==2.15.0`. We will also need to install the `onnx==1.15.0` and `onnx2tf==1.19.11` libraries for the exercise. The full code for this exercise is available in our GitHub repository [30]. Please follow *steps 1* to *11* from the *Model tracing with TorchScript* section, and then follow up with the steps given in this exercise:

1. Similar to model tracing, we again pass a dummy input through our loaded model:

   ```
   demo_input = torch.ones(1, 1, 28, 28)
   torch.onnx.export(model, demo_input, "convnet.onnx")
   ```

   This should save a model onnx file. Under the hood, the same mechanism is used for serializing the model as was used in model tracing.

2. Next, we load the saved onnx model and convert it into a TensorFlow model:

   ```
   onnx2tf.convert(
       input_onnx_file_path="convnet.onnx",
       output_folder_path="convnet_tf",
       non_verbose=True)
   ```

3. Next, we load the serialized `tensorflow` model to parse the model graph. This will help us in verifying that we have loaded the model architecture correctly as well as in identifying the input and output nodes of the graph:

   ```
   model = tf.saved_model.load("./convnet_tf/")print(model)
   ```

   This should output the following:

   ```
   <ConcreteFunction (inputs_0: TensorSpec(shape=(1, 28, 28, 1), dtype=tf.
   float32, name='inputs_0')) -> TensorSpec(shape=(1, 10), dtype=tf.float32,
   name='unknown') at 0x2F71EEBB0>
   ```

4. From the model object, we can see the input and output nodes labelled as `inputs_0` and `unknown` respectively. Finally, we run inference on the TensorFlow model to generate predictions for our sample image:

   ```
   output = model(input_tensor.unsqueeze(-1))
   print(output)
   ```

   This should output the following:

   ```
   [[-9.35050774e+00 -1.20893326e+01 -2.23922171e-03 -8.92477798e+00
   -9.81972313e+00 -1.33498535e+01 -9.04598618e+00 -1.44924192e+01
   -6.30233145e+00 -1.22827682e+01]]
   ```

As you can see, in comparison with *step 12* of the *Model tracing with TorchScript* section, the predictions are exactly the same for the TensorFlow and PyTorch versions of our model. This validates the successful functioning of the ONNX framework. I encourage you to dissect the TensorFlow model further and understand how ONNX helps regenerate the exact same model in a different deep learning library by utilizing the underlying mathematical operations in the model graph.

This concludes our discussion of the different ways of exporting PyTorch models. The techniques covered here will be useful in deploying PyTorch models in production systems, as well as in working across various platforms. As new versions of deep learning libraries, programming languages, and even OSes keep coming, this is an area that will rapidly evolve accordingly.

Hence, it is highly advisable to keep an eye on the developments and make sure to use the latest and most efficient ways of exporting models as well as operationalizing them into production.

So far, we have worked on our local machines for serving and exporting our PyTorch models. In the next and final section of this chapter, we will briefly look at serving PyTorch models on some of the well-known cloud platforms, such as AWS, Google Cloud, and Microsoft Azure.

Serving PyTorch models in the cloud

Deep learning is computationally expensive and therefore demands powerful and sophisticated computational hardware. Not everyone has access to a local machine that has enough CPUs and GPUs to train gigantic deep learning models in a reasonable amount of time. Furthermore, we cannot guarantee 100 percent availability for a local machine that is serving a trained model for inference.

For reasons such as these, cloud computing platforms are a vital alternative for both training and serving deep learning models.

In this section, we will discuss how to use PyTorch with some of the most popular cloud platforms – **AWS**, **Google Cloud**, and **Microsoft Azure**. We will explore the different ways of serving a trained PyTorch model on each of these platforms. The model-serving exercises we discussed in earlier sections of this chapter were executed on a local machine. The goal of this section is to enable you to perform similar exercises using **virtual machines** (**VMs**) on the cloud.

Using PyTorch with AWS

AWS is the oldest and one of the most popular cloud computing platforms. It has deep integrations with PyTorch. We already saw an example of this in the form of TorchServe, which is jointly developed by AWS and Facebook.

In this section, we will look at some of the common ways of serving PyTorch models using AWS. First, we will simply learn how to use an AWS instance as a replacement for our local machine (laptop/desktop) to serve PyTorch models. Then, we will briefly discuss Amazon SageMaker, which is a fully dedicated cloud machine learning platform. We will briefly discuss how TorchServe can be used together with SageMaker for model serving.

> **Note**
>
> This section assumes basic familiarity with AWS. Therefore, we will not elaborate on topics such as what an AWS EC2 instance is, what AMIs are, how to create an instance, and so on. You can refresh AWS basics at [31]. We will instead focus on the components of AWS that are related to PyTorch.

Serving a PyTorch model using an AWS instance

In this section, we will demonstrate how we can use PyTorch within a VM – an AWS instance, in this case. After reading this section, you will be able to execute the exercises discussed in the *Model serving in PyTorch* section inside an AWS instance.

First, you will need to create an AWS account if you haven't done so already [32]. Creating an account requires an email address and a payment method (a credit card).

Once you have an AWS account, you may log in to enter the AWS console [33]. From here, we basically need to instantiate a VM – an AWS instance in this case – where we can start using PyTorch for training and serving models. Creating a VM requires two decisions:

- Choosing the hardware configuration of the VM, also known as the **AWS instance type**
- Choosing the **Amazon Machine Image** (**AMI**), which entails all the required software, such as the OS (Ubuntu or Windows), Python, PyTorch, and so on

You can read more about the interaction between the preceding two components here [34]. Typically, when we refer to an AWS instance, we are referring to an **Elastic Cloud Compute** instance, also known as an **EC2** instance.

Based on the computational requirements of the VM (RAM, CPUs, and GPUs), you can choose from a long list of EC2 instances provided by AWS [35]. Because PyTorch heavily leverages GPU compute power, it is recommended to use EC2 instances that include GPUs, though they are generally costlier than CPU-only instances.

Regarding AMIs, there are two possible approaches to choosing an AMI. You may go for a bare-bones AMI that only has an OS installed, such as Ubuntu (Linux). In this case, you can then manually install Python [36] and subsequently install PyTorch [37].

An alternative and more recommended way is to start with a pre-built AMI that has PyTorch installed already. AWS offers Deep Learning AMIs, which make the process of getting started with PyTorch on AWS much faster and easier [38].

Once you have launched an instance successfully using either of the suggested approaches, you may simply connect to the instance using one of the various available methods [39].

SSH is one of the most common ways of connecting to an instance. Once you are inside the instance, it will have the same layout as working on a local machine. One of the first logical steps would then be to test whether PyTorch was working inside the machine.

To test, first open a Python interactive session by simply typing `python` on the command line. Then, execute the following line of code:

```
import torch
```

If it executes without error, that means that you have PyTorch installed on the system.

At this point, you can simply fetch all the code that we wrote in the preceding sections of this chapter on model serving. On the command line inside your home directory, simply clone this book's GitHub repository by running this:

```
git clone https://github.com/arj7192/MasteringPyTorchV2.git
```

Then, within the `Chapter13` subfolder, you will have all the code to serve the MNIST model that we worked on in the previous sections. You can basically re-run the exercises, this time, on the AWS instance instead of your local computer.

Let's review the steps we need to take for working with PyTorch on AWS:

1. Create an AWS account.
2. Log in to the AWS console.
3. Click on the **Launch a virtual machine** button in the console.
4. Select an AMI. For example, select the Deep Learning AMI (Ubuntu).
5. Select an AWS instance type. For example, select **p.2x large**, as it contains a GPU.
6. Click **Launch**.
7. Click **Create a new key pair**. Give the key pair a name and download it locally.
8. Modify the permissions for this key-pair file by running this on the command line:

   ```
   chmod 400 downloaded-key-pair-file.pem
   ```

9. On the console, click on **View Instances** to see the details of the launched instance and specifically note the public IP address of the instance.
10. Using SSH, connect to the instance by running this on the command line:

    ```
    ssh -i downloaded-key-pair-file.pem ubuntu@<Public IP address>
    ```

    The public IP address is the same as obtained in the previous step.

11. Once connected, start a `python` shell and run `import torch` in the shell to ensure that PyTorch is correctly installed on the instance.
12. Clone this book's GitHub repository by running the following on the instance's command line:

    ```
    git clone https://github.com/arj7192/MasteringPyTorchV2.git
    ```

13. Go to the `chapter13` folder within the repository and start working on the various model-serving exercises that were covered in the preceding sections of this chapter.

This brings us to the end of this section, where we essentially learned how to start working with PyTorch on a remote AWS instance. You can read more on PyTorch+AWS here [40]. If you don't want to build everything from scratch, AWS also provides a fully managed alternative. Up next, we will look at this offering, AWS's fully dedicated cloud machine learning platform – Amazon SageMaker.

Using TorchServe with Amazon SageMaker

We already discussed TorchServe in detail in the preceding section. As we know, TorchServe is a PyTorch model-serving library developed by AWS and Facebook. Instead of manually defining a model inference pipeline, model-serving APIs, and microservices, you can use TorchServe, which provides all of these functionalities.

Amazon SageMaker, on the other hand, is a cloud machine learning platform that offers functionalities such as the training of massive deep learning models as well as deploying and hosting trained models on custom instances. When working with SageMaker, all we need to do is this:

- Specify the type and number of AWS instances we would like to spin up to serve the model.
- Provide the location of the stored pre-trained model object.

We do not need to manually connect to the instance and serve the model using TorchServe. SageMaker takes care of all that. The AWS website has some useful blogs for getting started with using SageMaker and TorchServe to serve PyTorch models on an industrial scale and with just a few clicks [41]. Other AWS blogs also provide some use cases for Amazon SageMaker when working with PyTorch [42].

Tools such as SageMaker are incredibly useful for scalability during both model training and serving. However, while using such one-click tools, we often tend to lose some flexibility and debuggability. Therefore, it is for you to decide what set of tools works best for your use case. This concludes our discussion on using AWS as a cloud platform for working with PyTorch. Next, we will look at another cloud platform – Google Cloud.

Serving PyTorch models on Google Cloud

Similar to AWS, you first need to create a Google account (*@gmail.com) if you do not have one already. Furthermore, to be able to log in to the Google Cloud console [43], you will need to add a payment method (credit card details).

> **Note**
>
> We will not cover the basics of Google Cloud here, which you can read about here [44]. We will instead focus on using Google Cloud for serving PyTorch models within a VM.

Once inside the console, we need to follow the steps similar to AWS to launch a VM where we can serve our PyTorch model. You can always start with a bare-bones VM and manually install PyTorch. But we will be using Google's Deep Learning VM Image [45], which has PyTorch pre-installed. Here are the steps for launching a Google Cloud VM and using it to serve PyTorch models:

1. Launch Deep Learning VM Image on Google Cloud using the Google Cloud marketplace [46].
2. Input the deployment name in the command window. This name, suffixed with `-vm`, acts as the name of the launched VM. The command prompt inside this VM will look like this:

   ```
   <user>@<deployment-name>-vm:~/
   ```

   Here, `user` is the client connecting to the VM and `deployment-name` is the name of the VM chosen in this step.

3. Select `PyTorch` as the `Framework` in the next command window. This tells the platform to pre-install PyTorch in the VM.
4. Select the zone for this machine. Preferably, choose the zone geographically closest to you. Also, different zones have slightly different hardware offerings (VM configurations) and hence you might want to choose a specific zone for a specific machine configuration.
5. Having specified the software requirement in *step 3*, we shall now specify the hardware requirements. In the `GPU` section of the command window, we need to specify the GPU type and, subsequently, the number of GPUs to be included in the VM.

   Google Cloud provides various GPU devices/configurations [47]. In the `GPU` section, also check the checkbox that will automatically install the NVIDIA drivers that are necessary to utilize the GPUs for deep learning.

6. Similarly, under the `CPU` section, we need to provide the machine type. A list of such machine configurations can be found here [48]. Regarding *step 5* and *step 6*, please be aware that different zones provide different machine and GPU types, as well as different combinations of GPU types and GPU numbers.
7. Finally, click on the **Deploy** button. This will launch the VM and lead you to a page that will have all the instructions needed to connect to the VM from your local computer.

8. At this point, you may connect to the VM and ensure that PyTorch is correctly installed by trying to import PyTorch from within a Python shell. Once verified, clone this book's GitHub repository. Go to the `Chapter13` folder and start working on the model-serving exercises within this VM.

You can read more about creating a PyTorch Deep Learning VM Image on Google Cloud blogs [49]. This concludes our discussion of using Google Cloud as a cloud platform to work with PyTorch model serving. As you may have noticed, the process is very similar to that with AWS. In the next and final section, we will briefly look at using Microsoft's cloud platform, Azure, to work with PyTorch.

Serving PyTorch models with Azure

Once again, similar to AWS and Google Cloud, Azure requires a Microsoft-recognized email ID for signing up, along with a valid payment method.

Note

We assume a basic understanding of the Microsoft Azure cloud platform for this section. If you want to refresh these concepts, here [50] is a good resource.

Once you have access to the Azure portal [51], there are broadly two recommended ways of getting started with using PyTorch on Azure:

- Data Science Virtual Machines (DSVMs)
- Azure Machine Learning Service

We will now discuss these approaches briefly.

Working with Azure's DSVMs

Similar to Google Cloud's Deep Learning VM Images, Azure offers its own DSVM images [52], which are fully dedicated VM images for data science and machine learning, including deep learning.

These images are available for Windows [53], as well as Linux/Ubuntu [54].

The steps to create a DSVM instance using these images are quite similar to the steps discussed for Google Cloud for both Windows [55] as well as Linux/Ubuntu [56].

Once you have created the DSVM, you can launch a Python shell and try to import the PyTorch library to ensure that it is correctly installed. You may further test the functionalities available in this DSVM for Linux [57] as well as Windows [58].

Finally, you may clone this book's GitHub repository within the DSVM instance and use the code within the `Chapter13` folder to work on the PyTorch model-serving exercises discussed in this chapter.

Discussing Azure Machine Learning Service

Similar to and predating Amazon's SageMaker, Azure provides an end-to-end cloud machine learning platform. The **Azure Machine Learning Service** (**AMLS**) contains the following components (to name just a few):

- Azure Machine Learning VMs
- Notebooks
- Virtual environments
- Datastores
- Tracking machine learning experiments
- Data labeling

A key difference between AMLS VMs and DSVMs is that the former are fully managed. For instance, they can be scaled up or down based on the model training or serving requirements [59].

Just like SageMaker, AMLS is useful for training large-scale models as well as deploying and serving those models. The Azure website has a great tutorial for training PyTorch models on AMLS, as well as for deploying PyTorch models on AMLS for Windows [60] and Linux [61].

AMLS aims to provide a one-click interface to the user for all machine learning tasks. Hence, it is important to keep in mind the flexibility trade-off. Although we have not covered all the details about AML here, Azure's website is a good resource for further reading [62].

This brings us to the end of discussing what Azure has to offer as a cloud platform for working with PyTorch. You can learn more about PyTorch+Azure here [63].

And that also concludes our discussion of using PyTorch to serve models on the cloud. We have discussed AWS, Google Cloud, and Microsoft Azure in this section. Although there are more cloud platforms available out there, the nature of their offerings and the ways of using PyTorch within those platforms will be similar to what we have discussed. This section will help you in getting started with working on your PyTorch projects on a VM in the cloud.

Summary

In this chapter, we explored the world of deploying trained PyTorch deep learning models in production systems.

In the next chapter, we will learn how to deploy trained PyTorch models on different mobile operating systems – Android and iOS.

Reference list

1. Full code for *Creating a PyTorch model inference pipeline* exercise: `https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/Chapter10/mnist_pytorch.ipynb`
2. Notebook for *Creating a PyTorch model inference pipeline* exercise: `https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/Chapter10/run_inference.ipynb`
3. Flask library: `https://flask.palletsprojects.com/en/1.1.x/`

4. Full code for the *Using Flask to build our model server* exercise: `https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/Chapter10/server.py`
5. Flask server code: `https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/Chapter10/make_request.py`
6. *What are Microservices*: `https://opensource.com/resources/what-are-microservices`
7. Docker documentation: `https://docs.docker.com/get-started/overview/`
8. Docker installation guide: `https://docs.docker.com/engine/install/`
9. GitHub repo for the *Creating a model microservice* exercise: `https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/Chapter10/requirements.tx`
10. Dockerfile full code: `https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/Chapter10/Dockerfile`
11. Docker security best practices: `https://snyk.io/blog/10-docker-image-security-best-practices/`
12. TorchServe documentation: `https://github.com/pytorch/serve/blob/master/README.md#install-torchserve`
13. TorchServe Windows installation guide: `https://pytorch.org/serve/torchserve_on_win_native.html`
14. Torch model archiver: you can read about the archiver in detail here: `https://pytorch.org/serve/model-archiver.html`
15. Full model file: `https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/Chapter10/convnet.pth`
16. Custom handler code: `https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/10_operationalizing_pytorch_models_into_production/convnet_handler.py`
17. Model metrics: `https://pytorch.org/serve/`
18. TorchServe configuration settings: `https://pytorch.org/serve/configuration.html`
19. TorchServe advanced features: `https://pytorch.org/serve/server.html#advanced-features`
20. Full code for the *Model tracing with TorchScript* exercise: `https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/Chapter10/model_tracing.ipynb`
21. Full code for the *Model scripting with TorchScript* exercise: `https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/Chapter10/model_scripting.ipynb`
22. C++ coding primer: `https://www.learncpp.com/`
23. C++ coding compilation: `https://www.toptal.com/c-plus-plus/c-plus-plus-understanding-compilation`
24. CMake installation guide: `https://cmake.org/install/`
25. Full code for the *Running a PyTorch model in C++* exercise: `https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/Chapter10/cpp_convnet/cpp_convnet.cpp`
26. C++ Mac: `https://docs.opencv.org/master/d0/db2/tutorial_macos_install.html`
27. C++ Linux: `https://docs.opencv.org/3.4/d7/d9f/tutorial_linux_install.html`

28. C++ Windows: https://docs.opencv.org/master/d3/d52/tutorial_windows_install.html
29. Full code for the CMakeLists.txt file: https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/Chapter10/cpp_convnet/CMakeLists.txt
30. Full code for the *Using ONNX to export PyTorch models* exercise: https://github.com/PacktPublishing/Mastering-PyTorch/blob/master/Chapter10/onnx.ipynb
31. AWS – *Getting Started*: https://aws.amazon.com/getting-started/
32. AWS account creation: https://aws.amazon.com/premiumsupport/knowledge-center/create-and-activate-aws-account/
33. The AWS console: https://aws.amazon.com/console/
34. AWS instances and AMIs guide: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instances-and-amis.html
35. AWS EC2 instance types: https://aws.amazon.com/ec2/instance-types/
36. Linux Python installation guide: https://docs.python-guide.org/starting/install3/linux/
37. PyTorch Linux documentation: https://pytorch.org/get-started/locally/#linux-prerequisites
38. AWS EC2 instance blog post: https://aws.amazon.com/blogs/machine-learning/get-started-with-deep-learning-using-the-aws-deep-learning-ami/
39. A guide to accessing EC2 instances: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AccessingInstances.html
40. Using PyTorch on AWS: https://pytorch.org/get-started/cloud-partners/#aws-quick-start
41. Deploying PyTorch models at scale: https://aws.amazon.com/blogs/machine-learning/deploying-pytorch-models-for-inference-at-scale-using-torchserve/
42. Amazon SageMaker use cases: https://docs.aws.amazon.com/sagemaker/latest/dg/pytorch.html
43. Google Cloud console: https://console.cloud.google.com
44. Google Cloud – *Getting Started*: https://console.cloud.google.com/getting-started
45. Google's deep learning VM: https://cloud.google.com/deep-learning-vm
46. Google's deep learning VM launcher: https://console.cloud.google.com/marketplace/product/click-to-deploy-images/deeplearning
47. Google Cloud GPU documentation: https://cloud.google.com/compute/docs/gpus
48. Google Cloud machine types: https://cloud.google.com/compute/docs/machine-types
49. Google Cloud blogs (PyTorch deep learning VM): https://cloud.google.com/ai-platform/deep-learning-vm/docs/pytorch_start_instance
50. Azure – *Getting Started*: https://azure.microsoft.com/en-us/get-started/
51. The Azure portal: https://portal.azure.com/
52. Azure DSVMs: https://azure.microsoft.com/en-us/services/virtual-machines/data-science-virtual-machines/

53. Azure DSVM Windows images: `https://azuremarketplace.microsoft.com/en-us/marketplace/apps/microsoft-dsvm.dsvm-win-2019?tab=Overview`
54. Azure DSVM Linux images: `https://azuremarketplace.microsoft.com/en-us/marketplace/apps/microsoft-dsvm.ubuntu-1804?tab=Overview`
55. Azure DSVM creation in Windows: `https://docs.microsoft.com/en-gb/azure/machine-learning/data-science-virtual-machine/provision-vm`
56. Azure DSVM creation in Linux: `https://docs.microsoft.com/en-gb/azure/machine-learning/data-science-virtual-machine/dsvm-ubuntu-intro`
57. Linux DSVM walkthrough: `https://docs.microsoft.com/en-gb/azure/machine-learning/data-science-virtual-machine/linux-dsvm-walkthrough`
58. Windows DSVM walkthrough: `https://docs.microsoft.com/en-gb/azure/machine-learning/data-science-virtual-machine/vm-do-ten-things`
59. Azure ML VMs vs. DSVMs: `https://docs.microsoft.com/en-gb/azure/machine-learning/data-science-virtual-machine/overview`
60. Training PyTorch models on AMLS: `https://docs.microsoft.com/en-us/azure/machine-learning/how-to-train-pytorch`
61. Deploying PyTorch models on AMLS: `https://docs.microsoft.com/en-us/azure/machine-learning/how-to-deploy-and-where?tabs=azcli`
62. Azure ML further reading: `https://docs.microsoft.com/en-us/azure/machine-learning/overview-what-is-azure-ml`
63. Guidance for working with PyTorch and Azure: `https://azure.microsoft.com/en-us/develop/pytorch/`

Learn more on Discord

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

`https://packt.link/mastorch`

14

PyTorch on Mobile Devices

In the previous chapter, we learned extensively about operationalizing PyTorch models as services in production systems. While productionizing **machine learning** (**ML**) models as services in the cloud remains the most popular form of ML deployment, several use cases require models to be deployed on mobile devices, such as:

- **User data protection** – mobile models do not require third-party data transfer, as the processing is done where the data is first acquired
- **Reduced latency –** mobile models save us the cloud network I/O time
- **Better user experience** – mobile models can provide real-time user interaction with lower latency compared to running models remotely from the cloud
- **Leveraging dedicated mobile hardware and software** for ML (eg., coreML) that mobile phone makers are increasingly adding to their products

In this chapter, we will learn how to deploy PyTorch models on mobile devices using PyTorch Mobile [1]. PyTorch Mobile is a subset of PyTorch designed for mobile and embedded platforms. It allows developers to run PyTorch models on edge devices such as smartphones, tablets, and IoT devices. Under the hood, PyTorch Mobile optimizes model execution and memory usage for efficient and fast performance on mobile and embedded hardware.

> **Note**
>
> Running ML models on mobile devices faces challenges such as limited computing power, reduced memory capacity, and energy constraints, which necessitate significant model optimization and lightweight architecture adaptations compared to cloud-based solutions. These constraints make it crucial to optimize models for efficient on-device performance to ensure responsive and sustainable application functionality.

We will use PyTorch Mobile to optimize the MNIST model that we trained in *Chapter 1* of this book, and we will then deploy this optimized model on two major mobile operating systems – Android and iOS. Depending on the mobile device you have (Android and/or iOS), you will be able to try out these deployments on your device. In this process, we will also learn how to build a camera-based app, both on Android as well as iOS, that can capture images (of handwritten digits) and run predictions on those images using the MNIST model.

This chapter is broken down into the following topics:

- Deploying a PyTorch model on Android
- Building a PyTorch app on iOS

Deploying a PyTorch model on Android

In this section, we will create an Android app that allows you to capture an image using the phone camera and make a prediction (image classification) on the captured image. In *Chapter 1* of this book, we trained a **Modified National Institute of Standards and Technology** (**MNIST**) model to classify handwritten digits. In *Chapter 13*, we used tracing to convert the trained MNIST model from the original PyTorch format into an **intermediate representation** (**IR**). For our Android app, we will first optimize this traced MNIST model using PyTorch Mobile and then use the optimized model to make predictions (handwritten digit classification) on the captured image. All code for this section is available on GitHub [2].

Converting the PyTorch model to a mobile-friendly format

PyTorch Mobile provides a function, `optimize_for_mobile`, that converts a traced PyTorch model object into a mobile-friendly lightweight format. This can be done with the following lines of code:

```
import torch
from torch.utils.mobile_optimizer import optimize_for_mobile
traced_model = torch.jit.load('MasteringPyTorchV2/Chapter13/traced_convnet.pt')
optimized_traced_model = optimize_for_mobile(traced_model)
optimized_traced_model._save_for_lite_interpreter("./app/src/main/assets/
optimized_for_mobile_traced_model.pt")
```

You can find this code on GitHub [3]. The above code first loads the traced PyTorch model, converts it into a mobile-optimized model, and saves this model into an `assets` folder. Our Android app will load this optimized model to run predictions on the camera-captured images.

The `optimize_for_mobile` function in PyTorch is designed to enhance the performance of ML models on mobile devices. It performs a series of optimizations, such as reducing model size, improving memory usage, and increasing execution speed, by fusing operations and pruning unnecessary components. This ensures that the model runs efficiently on the limited hardware resources typical of mobile environments.

Let us next work on building the Android app.

Setting up the Android app development environment

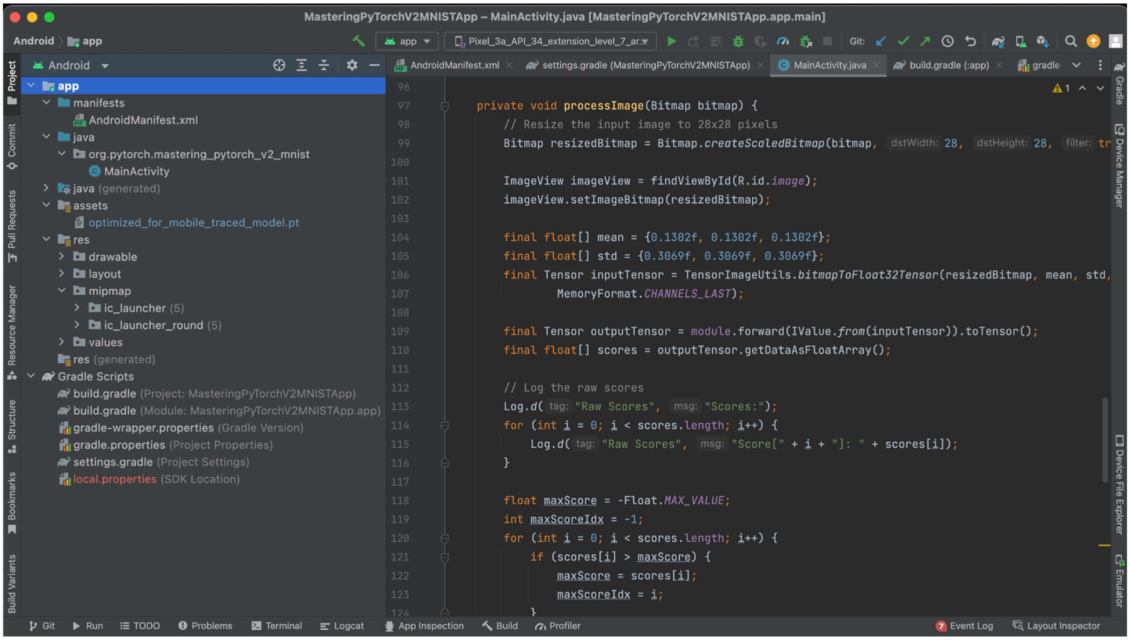
To build the Android app, we need to download Android Studio. Android Studio is the official **integrated development environment** (**IDE**) for Android app development. Developed by Google, it has a powerful and user-friendly interface. Android Studio provides tools and features for designing, coding, testing, and debugging Android applications. You can download it from their official website [4].

> **Note**
>
> Our Android app will need version 3.5.1 (or higher) of Android Studio to run successfully. This version of Android Studio also facilitates the installation of the Android **SDK** (**Software Development Kit**) and Android **NDK** (**Native Development Kit**) via the Android Studio UI, which are necessary for this project.

Inside Android Studio, we need to open a new project using the `Android` folder [2] as the project path.

> **Note**
>
> The Android SDK requires Java and hence the **Java Development Kit** (**JDK**). If you don't have it installed, please follow the instructions here [5].

You shall see an IDE window similar to *Figure 14.1*:


MasteringPyTorchV2MNISTApp – MainActivity.java [MasteringPyTorchV2MNISTApp.app.main]
Android 〉 app
app
Pixel_3a_API_34_extension_level_7_ar
Git:
Android
AndroidManifest.xml
settings.gradle (MasteringPyTorchV2MNISTApp)
MainActivity.java
build.gradle (:app)
gradle
app
manifests
AndroidManifest.xml
java
org.pytorch.mastering_pytorch_v2_mnist
MainActivity
java (generated)
assets
optimized_for_mobile_traced_model.pt
res
drawable
layout
mipmap
ic_launcher (5)
ic_launcher_round (5)
values
res (generated)
Gradle Scripts
build.gradle (Project: MasteringPyTorchV2MNISTApp)
build.gradle (Module: MasteringPyTorchV2MNISTApp.app)
gradle-wrapper.properties (Gradle Version)
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)
Project
Commit
Pull Requests
Resource Manager
Structure
Bookmarks
Build Variants
Gradle
Device Manager
Device File Explorer
Emulator
Git
Run
TODO
Problems
Terminal
Logcat
App Inspection
Build
Profiler
Event Log
Layout Inspector

```java

    private void processImage(Bitmap bitmap) {
        // Resize the input image to 28x28 pixels
        Bitmap resizedBitmap = Bitmap.createScaledBitmap(bitmap,  dstWidth: 28,  dstHeight: 28,  filter: tr

        ImageView imageView = findViewById(R.id.image);
        imageView.setImageBitmap(resizedBitmap);

        final float[] mean = {0.1302f, 0.1302f, 0.1302f};
        final float[] std = {0.3069f, 0.3069f, 0.3069f};
        final Tensor inputTensor = TensorImageUtils.bitmapToFloat32Tensor(resizedBitmap, mean, std,
                MemoryFormat.CHANNELS_LAST);

        final Tensor outputTensor = module.forward(IValue.from(inputTensor)).toTensor();
        final float[] scores = outputTensor.getDataAsFloatArray();

        // Log the raw scores
        Log.d( tag: "Raw Scores",  msg: "Scores:");
        for (int i = 0; i < scores.length; i++) {
            Log.d( tag: "Raw Scores",  msg: "Score[" + i + "]: " + scores[i]);
        }

        float maxScore = -Float.MAX_VALUE;
        int maxScoreIdx = -1;
        for (int i = 0; i < scores.length; i++) {
            if (scores[i] > maxScore) {
                maxScore = scores[i];
                maxScoreIdx = i;
            }
```

Figure 14.1: Android Studio window showing our Android project – MasteringPyTorchV2MNISTApp

In *Figure 14.1*, you may notice that the name of our project is `MasteringPyTorchV2MNISTApp`. We have set this under the `settings.gradle` file [6], which contains the following lines of code:

```
include ':app'
rootProject.name='MasteringPyTorchV2MNISTApp'
```

The above configuration code essentially defines the name of our Android app (or project) and points to the source code of the Android app, which is located inside the app folder.

Note

Gradle is an open-source build automation tool used for building, automating, and managing the build process of software projects. It is particularly popular in the Java and Android development communities, although it can be used for building projects in other programming languages as well.

In *Figure 14.1*, we can see a `build.gradle` file [7], which acts as a *MakeFile* containing all the build instructions for our app. The contents of this file are as follows:

```
apply plugin: 'com.android.application'
android {
    compileSdkVersion 28
    buildToolsVersion "29.0.2"
    defaultConfig {
        applicationId "org.pytorch.mastering_pytorch_v2_mnist"
        minSdkVersion 21
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"
    }
...
dependencies {
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'org.pytorch:pytorch_android_lite:1.12.2'
    implementation 'org.pytorch:pytorch_android_torchvision_lite:1.12.2'
}
```

As we can see, this file contains various pieces of configuration information, such as the name and version of the Android app, the minimum supported and targeted SDK version (API level), and different dependencies (libraries that the app requires to function). You will notice that PyTorch Android is listed as a dependency (alongside AndroidX). While `org.pytorch:pytorch_android_lite` is the main PyTorch Android API dependency, containing the libtorch native library for Android, `org.pytorch:pytorch_android_torchvision` provides functions to convert images captured by the Android app (in formats such as `android.media.Image` and `android.graphics.Bitmap`) to tensors.

Note

AndroidX is an open-source Android software library and development platform that provides a set of libraries, tools, and architectural components to simplify Android app development. It's the modern replacement for the Android Support Library, with several enhancements and additional features.

In *Figure 14.1*, we can also see that the optimized MNIST model file is displayed under the `assets` folder. While we already have the ML model, we need to build the following two components to complete the Android app:

- Camera capture
- Using captured images for ML model inference

Before we work on running model inference on captured images (as indicated under the `processImage` function in *Figure 14.1*), in the next section, we will discuss the important aspects of our Android code that enable our app to use the phone camera for capturing images and rendering those captured images for further use.

Using the phone camera in the Android app to capture images

When we build an Android app, we need to decide what aspects of the phone hardware and software our app needs access to. Before starting to write any app code, we first need to fill in these access requirements inside the `AndroidManifest.xml` file, which is located inside the `src/main/` subfolder within the app folder. Our manifest file [8] looks like the following:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.pytorch.mastering_pytorch_v2_mnist">
...
    <uses-permission android:name="android.permission.CAMERA" />
    <uses-feature android:name="android.hardware.camera" />
    <uses-feature android:name="android.hardware.camera.autofocus" />
</manifest>
```

In the initial lines, we specify the name and version of the Android app, and toward the end of the file, we request the camera access permission for this app to function. Besides the permission, the file specifies the extra phone features used by this app – in this case, the phone camera and the autofocus functionality within the phone camera.

Now that we have set the access permissions in our manifest file, it is time to head over to the main file of our Android app source code – `MainActivity.java`. This file [9] is located inside the `src/main/java/org/pytorch/mastering_pytorch_v2_mnist` folder and contains all of the logic that runs inside the Android app under the `MainActivity` class, as shown in the following code:

```java
package org.pytorch.mastering_pytorch_v2_mnist;
import android.content.Context;
import android.Manifest;
...
public class MainActivity extends AppCompatActivity {
    private static final int CAMERA_PERMISSION_CODE = 101;
    private static final int CAMERA_REQUEST_CODE = 102;
    private Module module;
    @Override
    protected void onCreate(Bundle savedInstanceState) {...}
    @Override
    public void onRequestPermissionsResult(int requestCode, @NonNull String[]
permissions, @NonNull int[] grantResults) {...}
    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent
data) {...}
    private void openCamera() {...}
    private void processImage(Bitmap bitmap) {...}
    // Helper method to get asset file path
    private String assetFilePath(Context context, String assetName) throws
IOException {...}
```

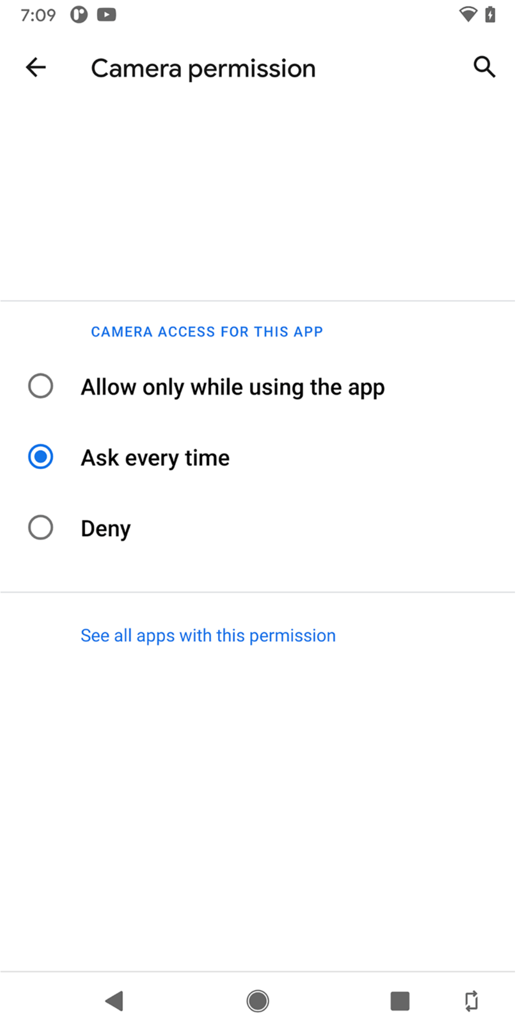
The code begins with the declaration of the package (app) followed by importing the required modules (dependencies), such as `android.content.Context` and `android.Manifest`. Then, we define the `MainActivity` class, which first initializes some constants and then defines a list of methods that handle important aspects and functionalities of our Android app. Let us look at some of these methods that are important for the camera capture functionality.

Enabling the camera during app startup

When the app starts, we need to check if the app has access to the phone camera. If yes, then the camera is opened and the image capture screen appears as soon as the app starts, as shown in the `onCreate` method below:

```java
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // Check for camera permission
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.CAMERA)
!= PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(this, new String[]{Manifest.
permission.CAMERA}, CAMERA_PERMISSION_CODE);
        } else {
            openCamera();
        }
    ...
    }
```

If the app doesn't have access to the phone camera, the above method asks the user to allow access to the camera, as shown in *Figure 14.2*.



Camera permission

CAMERA ACCESS FOR THIS APP

○ Allow only while using the app

● Ask every time

○ Deny

See all apps with this permission

Figure 14.2: Android app asking users to allow camera access for capturing images

If the user allows using the camera while using the app, this popup doesn't appear the next time the user starts the app. But if the user chooses the **Ask every time** option, then this popup appears every time the app starts. And, if the user chooses **Deny**, then the app will not function and display an error such as **Cannot connect to the camera**. Next, we look at another method in our code that handles the user response to the popup shown in *Figure 14.2*.

Handling camera permissions in Android

Once the user makes one of the choices seen in *Figure 14.2*, our code needs to handle that user action. This is done with the `onRequestPermissionsResult` method as shown in the following lines of code:

```
    public void onRequestPermissionsResult(int requestCode, @NonNull String[]
permissions, @NonNull int[] grantResults) {
        super.onRequestPermissionsResult(requestCode, permissions,
grantResults);
        if (requestCode == CAMERA_PERMISSION_CODE) {
            if (grantResults.length > 0 && grantResults[0] == PackageManager.
PERMISSION_GRANTED) {
                openCamera();
            } else {
                // Permission denied, handle accordingly
                Toast.makeText(this, "Camera permission denied. Cannot open the
camera.", Toast.LENGTH_SHORT).show();
            }
        }
    }
```

Essentially, we open the camera if permission to access the camera is granted by the user. Otherwise, we display an error message to the user stating that the camera cannot be opened.

Opening the camera for image capture

Once the app has ascertained that it has access to the phone camera, the following method opens the camera within the app to facilitate image capture:

```
private void openCamera() {
        Intent cameraIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
        if (cameraIntent.resolveActivity(getPackageManager()) != null) {
            startActivityForResult(cameraIntent, CAMERA_REQUEST_CODE);
        }
    }
```

At this point, you should be able to see a screen similar to *Figure 14.3*.

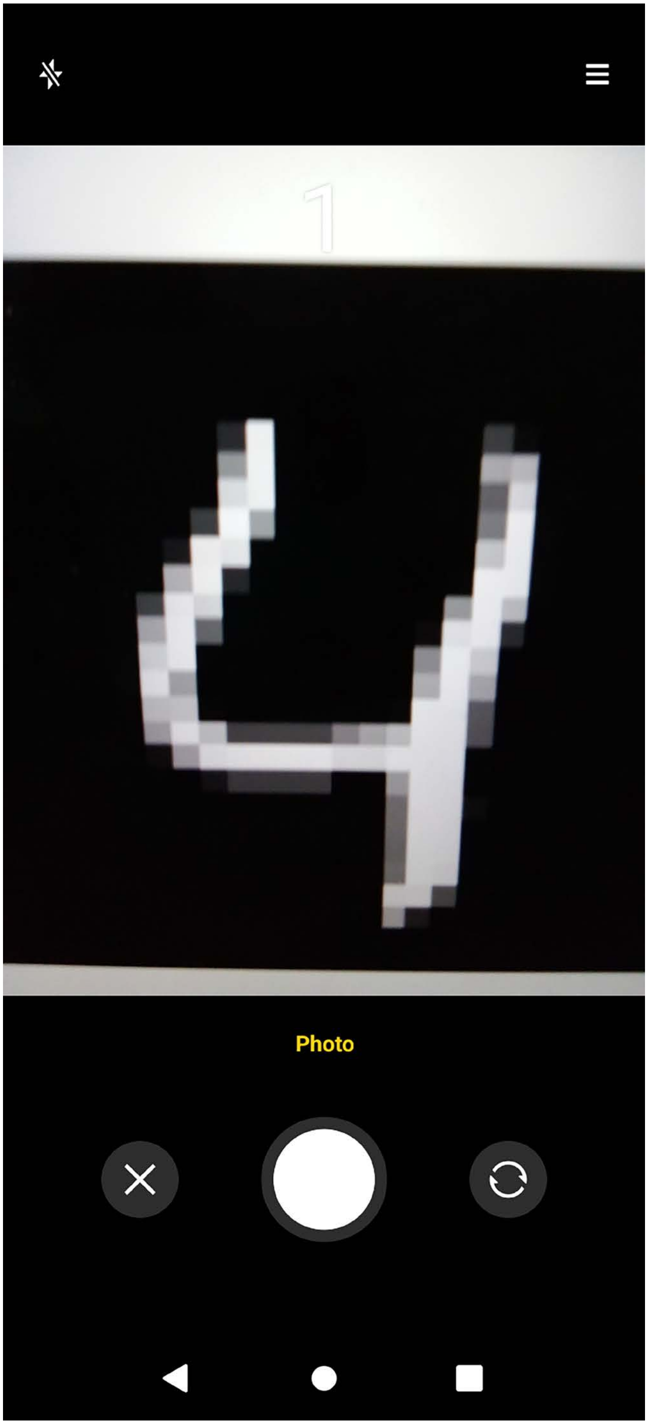


Figure 14.3: The image capture screen, where the app accesses the camera, which features different buttons, such as flash (top left), exit screen (bottom left), capture image (bottom center), switch camera (bottom right), and more options (top right)

The image capture screen allows you to capture a photo that you would like your ML model to make inferences on. In the example shown in *Figure 14.3*, the camera is pointing to a handwritten digit. Once captured, this image will then be analyzed by our MNIST model. Next, we will learn about the final camera-related method in our code, where we handle the scenario when the user clicks a photo on the image capture screen.

Capturing images using the phone camera

When the user takes a photo using the image capture screen shown in *Figure 14.3*, we need to store the captured image in the form of an image object. In Android development, that object is the `android.graphics.Bitmap` object. The following method converts the image captured by the camera into a `Bitmap` object:

```
    protected void onActivityResult(int requestCode, int resultCode, Intent 
data) {
        super.onActivityResult(requestCode, resultCode, data);
        if (requestCode == CAMERA_REQUEST_CODE && resultCode == RESULT_OK) {
            if (data != null && data.getExtras() != null) {
                Bitmap capturedBitmap = (Bitmap) data.getExtras().get("data");
                if (capturedBitmap != null) {
                    processImage(capturedBitmap);
                }
            }
        }
    }
```

This code checks if the image capture has returned a non-null stream of data. If so, that data is converted into a `Bitmap` object – `capturedBitmap`. Finally, if this object is not null, then the image is passed through the `processImage` method, where we run the model inference on the captured image. In the next section, we discuss the details of the mobile ML model inference in detail.

Running ML model inference on camera-captured images

In this section, we will focus on getting the ML model predictions within our Android app given that we have already captured an image using the phone camera. We ensure that we have the ML model file in the required place within our project code, load the ML model, and then finally, process the captured image to produce ML model prediction.

Validating the ML model binary path

Before we discuss the `processImage` method where the ML model inference takes place, let us rewind a little to the `onCreate` method where we check for camera permissions during app startup. Besides checking for camera access, this method also ascertains that the mobile-optimized ML model binary is available inside the `src/main/assets` folder, as shown in the following piece of code:

```
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
...
        try {
            module = LiteModuleLoader.load(assetFilePath(this, "optimized_for_
mobile_traced_model.pt"));
        } catch (IOException e) {
            Log.e("MasteringPyTorchV2MNIST", "Error reading assets", e);
            finish();
        }
    }
```

The ML model is loaded under the `module` variable. The above code uses a helper function, `assetFilePath`, which is also defined within the `MainActivity.java` file, as shown below:

```
// Helper method to get asset file path
    private String assetFilePath(Context context, String assetName) throws
IOException {
        File file = new File(context.getFilesDir(), assetName);
        if (!file.exists()) {
            try (InputStream is = context.getAssets().open(assetName)) {
                try (OutputStream os = new FileOutputStream(file)) {
                    byte[] buffer = new byte[4 * 1024];
                    int read;
                    while ((read = is.read(buffer)) != -1) {
                        os.write(buffer, 0, read);
                    }
                    os.flush();
                }
            }
        }
        return file.getAbsolutePath();
    }
```

Now that the ML model is verified to be present at the desired location, we process the image captured using the phone camera to produce MNIST model predictions.

Performing image classification on camera-captured images

We have finally arrived at the most important and relevant part of the Android app source code, where we write the logic for producing ML model predictions on the captured image using the `processImage` method, as shown below:

```
private void processImage(Bitmap bitmap) {
        // Resize the input image to 28x28 pixels
        Bitmap resizedBitmap = Bitmap.createScaledBitmap(bitmap, 28, 28, true);
        ImageView imageView = findViewById(R.id.image);
        imageView.setImageBitmap(resizedBitmap);
        // declare MNIST training dataset mean and std pixel values as we
        // did while training MNIST model in chapter 1
        // (https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter01/
        // mnist_pytorch.ipynb)
        final float[] mean = {0.1302f, 0.1302f, 0.1302f};
        final float[] std = {0.3069f, 0.3069f, 0.3069f};
        final Tensor inputTensor = TensorImageUtils.
bitmapToFloat32Tensor(resizedBitmap, mean, std,
                MemoryFormat.CHANNELS_LAST);
        final Tensor outputTensor = module.forward(IValue.from(inputTensor)).
toTensor();
        final float[] scores = outputTensor.getDataAsFloatArray();
        // Log the raw scores
        Log.d("Raw Scores", "Scores:");
        for (int i = 0; i < scores.length; i++) {
            Log.d("Raw Scores", "Score[" + i + "]: " + scores[i]);
        }
        float maxScore = -Float.MAX_VALUE;
        int maxScoreIdx = -1;
        for (int i = 0; i < scores.length; i++) {
            if (scores[i] > maxScore) {
                maxScore = scores[i];
                maxScoreIdx = i;
            }
        }
        String className = String.valueOf(maxScoreIdx);
        TextView textView = findViewById(R.id.text);
        textView.setText(className);
        // Add "Retake Photo" button logic here
        Button retakeButton = findViewById(R.id.retake_button);
        retakeButton.setVisibility(View.VISIBLE); // Show the retake button
        retakeButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                openCamera();
                // Call the openCamera method again to capture a new image
            }
        });
    }
```

There are three important elements within this method:

1. **Pre-processing the captured image (Bitmap object) into a tensor:** the bitmap image is first resized to (`28`, `28`) pixels, and the pixel values are normalized as per the MNIST dataset's mean and standard deviation values for `R`, `G`, and `B` channels. Then, the normalized `Bitmap` object is converted to a `float32` tensor.
2. **Running ML model inference on the image tensor:** the produced tensor is passed into the loaded ML model for inference, which produces class logits for all of the 10 classes (digits `0` to `9`). The class (digit) with the highest logit is displayed as the model prediction for the captured image. All class logits are logged for debugging.
3. **Asking the user to retake a photo for another model prediction:** once the model produces a prediction, a button pops up at the bottom of the screen offering the user the opportunity to retake the photo. If the user clicks that button, the app reopens the camera and goes back to the image capture screen.

Figure 14.4 shows the end-to-end flow, from the user capturing and confirming an image to the ML model returning the digit prediction for that image and the app asking the user if they want to retake the photo.

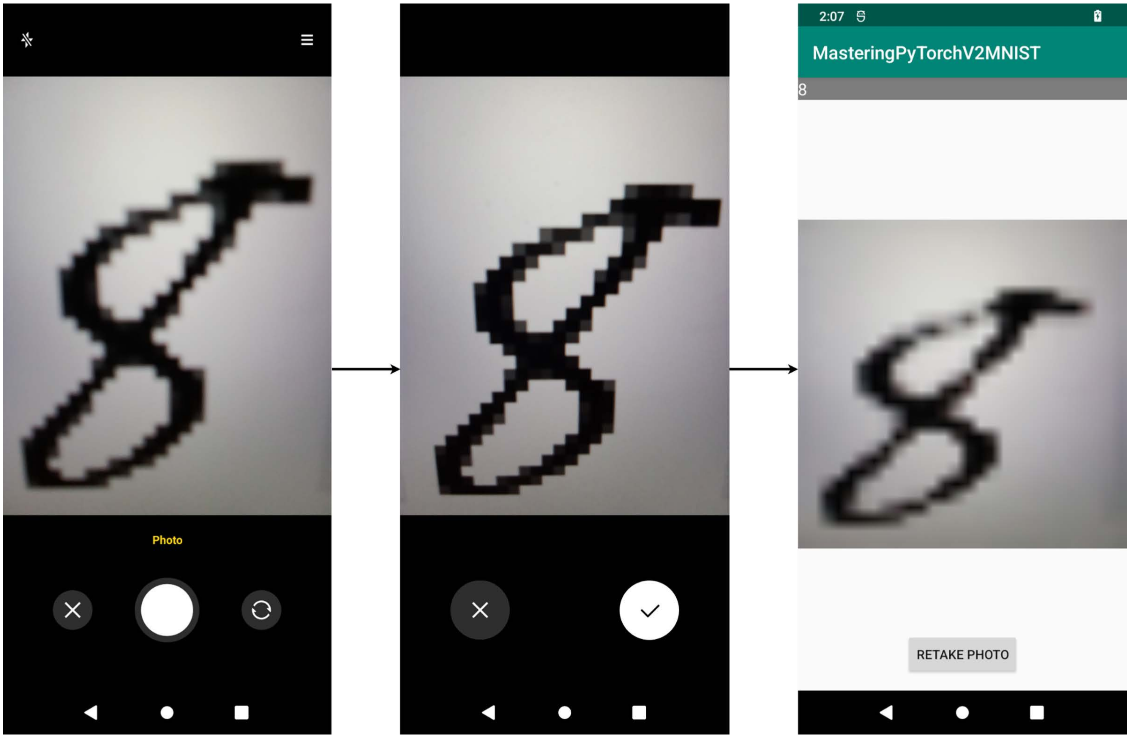


Figure 14.4: End-to-end MasteringPyTorchV2MNISTApp flow, from the image capture screen to the ML model inference screen

In the rightmost screen in *Figure 14.4*, when the ML model is triggered, the class logits are logged at the bottom of the Android Studio IDE, and the logs look like the following for this example of the digit 8:

```
D/Raw Scores: Scores:
D/Raw Scores: Score[0]: -2.8765326
D/Raw Scores: Score[1]: -7.383335
D/Raw Scores: Score[2]: -2.4631808
D/Raw Scores: Score[3]: -3.6827347
D/Raw Scores: Score[4]: -5.9684258
D/Raw Scores: Score[5]: -3.177271
D/Raw Scores: Score[6]: -4.41477
D/Raw Scores: Score[7]: -8.842412
D/Raw Scores: Score[8]: -0.26049972
D/Raw Scores: Score[9]: -5.1912117
```

As we can see, the logit value is the highest for the digit 8 and it is hence displayed as the model prediction, as shown in the rightmost screen in *Figure 14.4*.

Note

The higher the logit value, the higher the probability, and vice versa.

It is important to note that while transitioning from the middle screen to the rightmost screen in *Figure 14.4*, the captured image gets slightly distorted because of the reshaping (square image) and resizing (sizing down to 28 pixels on each side) of the captured image. What goes into the model is the image shown on the rightmost screen.

Having discussed all the necessary methods within the MainActivity.java file, we are now ready to launch this app on a device and use it in real time.

Launching the app on an Android mobile device

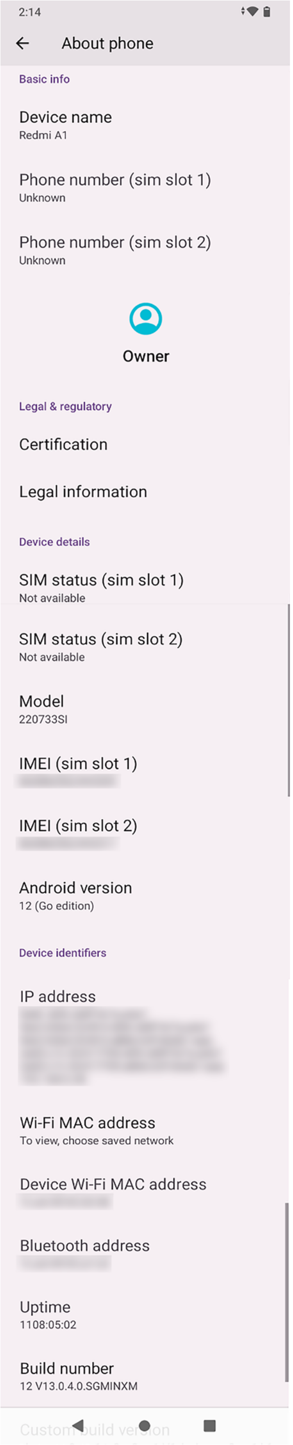
In this section, we will run this app on an Android device. You should plug an Android device (version 10 or higher) into your laptop via a USB cable and enable **Developer Options** in your device settings.

> **Note**
>
> If you have an iOS device as opposed to an Android device, you can simply read through this section, and then get hands-on in the next section, which is dedicated to iOS.

Once connected, you should be able to select your device under the **Available devices** dropdown in Android Studio (see the **Pixel_3a_API_34_extension_level_7_** section in *Figure 14.1*). After selecting your device, press the **Run** button. You shall see an output in the logs (at the bottom of the Android Studio screen) similar to the following:

```
11/02 13:44:44: Launching 'app' on Xiaomi 220733SI.
Install successfully finished in 703 ms.
$ adb shell am start -n "org.pytorch.mastering_pytorch_v2_mnist/org.pytorch.
mastering_pytorch_v2_mnist.MainActivity" -a android.intent.action.MAIN -c
android.intent.category.LAUNCHER
Connected to process 5857 on device 'xiaomi-220733si-ABCDEFGHIJKLMNOP'.
Capturing and displaying logcat messages from application. This behavior can be
disabled in the "Logcat output" section of the "Debugger" settings page.
```

The above logs are produced for the *Redmi A1* phone that I have used throughout this chapter for building our Android app. *Figure 14.5* shows the phone configuration for my device.



2:14

About phone

Basic info

Device name
Redmi A1

Phone number (sim slot 1)
Unknown

Phone number (sim slot 2)
Unknown

Owner

Legal & regulatory

Certification

Legal information

Device details

SIM status (sim slot 1)
Not available

SIM status (sim slot 2)
Not available

Model
220733SI

IMEI (sim slot 1)
[redacted]

IMEI (sim slot 2)
[redacted]

Android version
12 (Go edition)

Device identifiers

IP address
[redacted]

Wi-Fi MAC address
To view, choose saved network

Device Wi-Fi MAC address
[redacted]

Bluetooth address
[redacted]

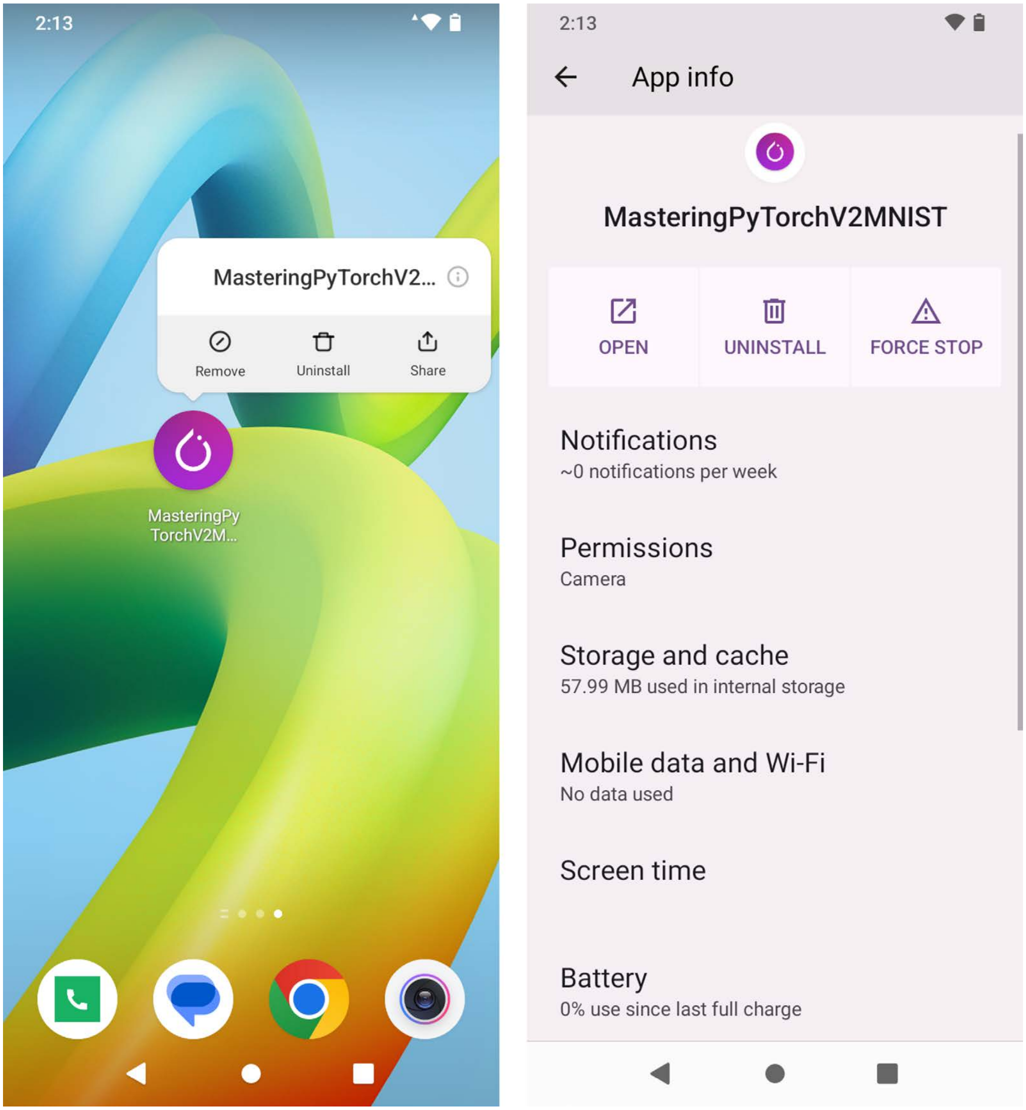
Uptime
1108:05:02

Build number
12 V13.0.4.0.SGMINXM

Custom build version

Figure 14.5: Device information for a Redmi A1 Android phone

Once you successfully launch the app on your device, you shall see the app startup screen, similar to the leftmost screen shown in *Figure 14.4*. You may see the screen shown in *Figure 14.2*, asking for your camera permissions, if you are launching the app for the first time. Once launched, the app gets permanently installed on your phone and can be accessed even if your phone is disconnected from your laptop. *Figure 14.6* shows the app, with its logo and information, on my phone's home screen.



MasteringPyTorchV2MNIST

Notifications
~0 notifications per week

Permissions
Camera

Storage and cache
57.99 MB used in internal storage

Mobile data and Wi-Fi
No data used

Screen time

Battery
0% use since last full charge

Figure 14.6: MasteringPyTorchV2MNIST installed on an Android device, available on the home screen

As we can see, the app needs camera permission and has some cache storage in the form of captured images in a given app session. *Figure 14.7* shows some examples of correct predictions made by the mobile ML model on various captured handwritten digit images.

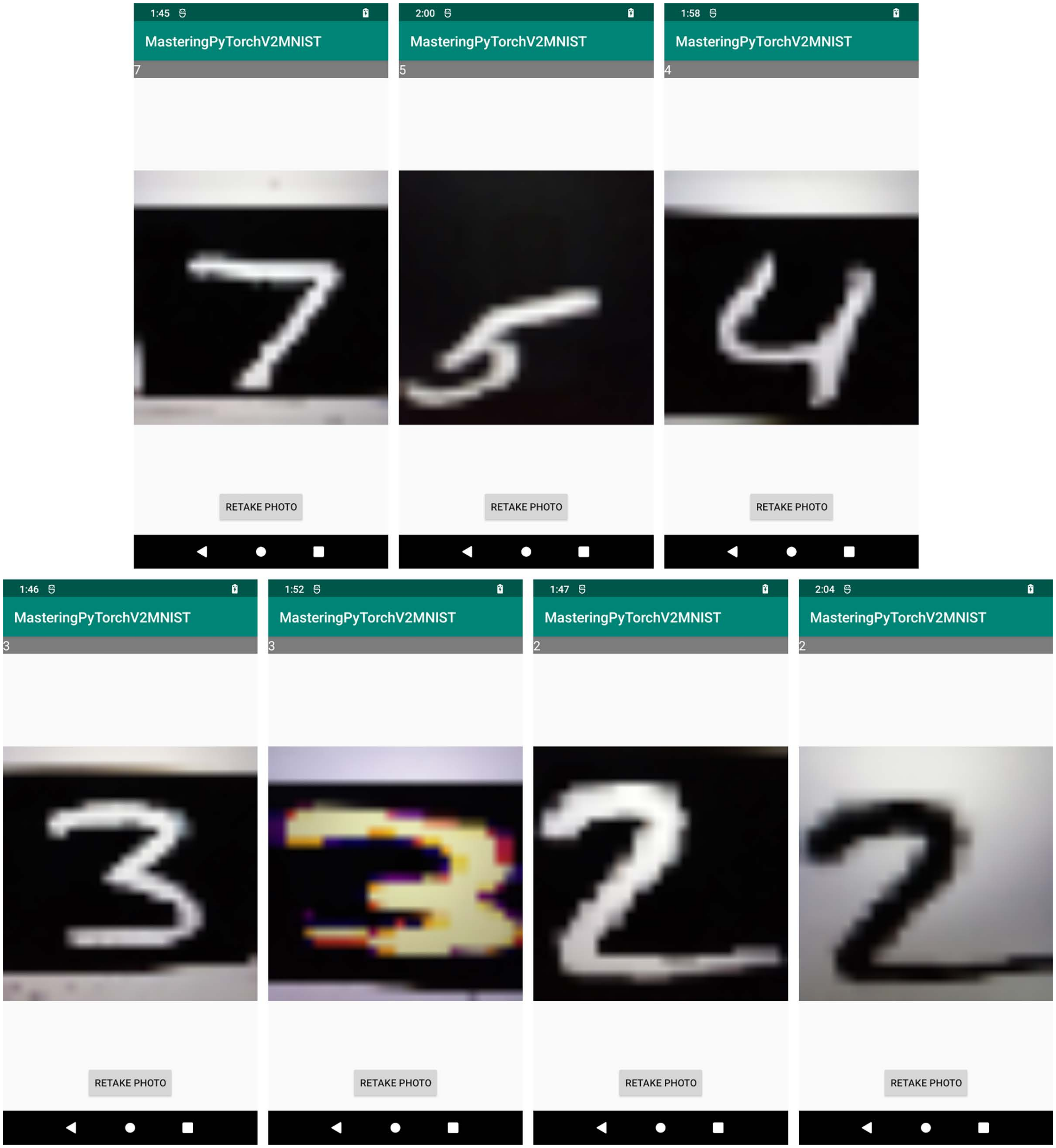


Figure 14.7: Examples of correct predictions made by the MNIST mobile model

Interestingly, the model can perform on different color combinations – black ink, white ink, yellow ink, black background, and white background. However, this model is not perfect, as is revealed by some mistakes made by the app for some borderline scenarios shown in *Figure 14.8*. In these images, the digit 1 is written in a way that coincides with parts of the digit 7 as well as the digit 4, hence the understandable mistakes that the model makes with this image. In such cases, it is useful to look at raw probability outputs from the model to see how confident the model is when making mistakes – the higher the confidence, the more effort is needed to retrain/fine-tune it to fix such errors.

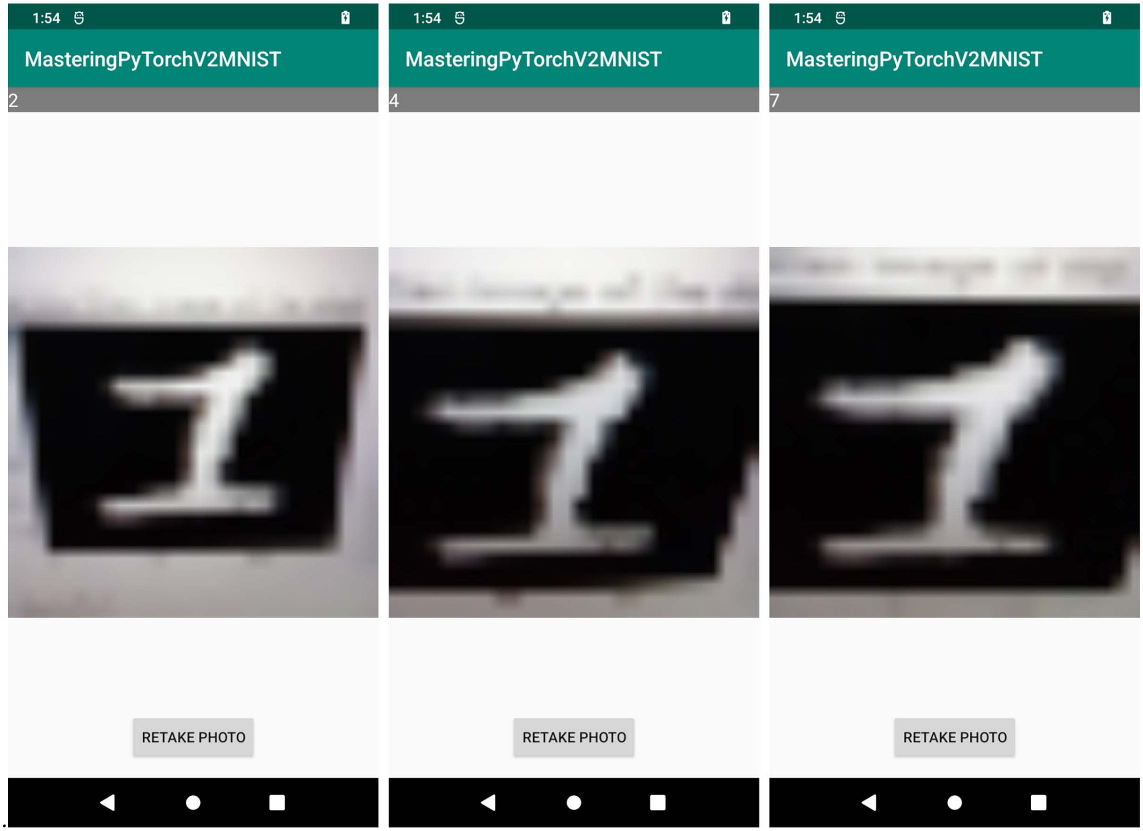


Figure 14.8: Examples of incorrect predictions made by the MNIST mobile model

This brings us to the end of building an Android app for handwritten digit classification on camera-captured images using PyTorch Mobile. The PyTorch Android webpage [10] is a great resource to read more about building and deploying different kinds of ML models to Android apps. In the next section, we will redo the same exercise but for iOS, once again with the help of PyTorch Mobile.

Building PyTorch apps on iOS

In this section, we will create an iOS app that allows you to perform handwritten digit classification on camera-captured images of handwritten digits. We will reuse the mobile-optimized MNIST model from the Android app development section. All code for this section is available on GitHub [11].

Setting up the iOS development environment

To build iOS apps, you need to download Xcode [12] on your MacBook. Xcode is an IDE created by Apple for developing software on iOS (and also on macOS, iPadOS, watchOS, and tvOS). It is the primary tool used by developers to create applications and software for Apple's various platforms.

Note

You will need Xcode version 11 or higher to build this app. Xcode is the IDE to build iOS apps. You must have a MacBook to build iOS apps.

Before we open Xcode, from the command line, we need to first set the current working directory to the `iOS/HelloWorld` folder [13]. From this directory, we need to run the following command:

```
pod install
```

The `pod` command refers to CocoaPods [14], which is an open-source dependency manager for iOS projects. It simplifies the process of incorporating third-party libraries and frameworks into an Xcode project. Rather than manually downloading, configuring, and adding external libraries to your project, CocoaPods automates the process.

Note

You can install CocoaPods on your MacBook using the following command: `sudo gem install cocoapods`.

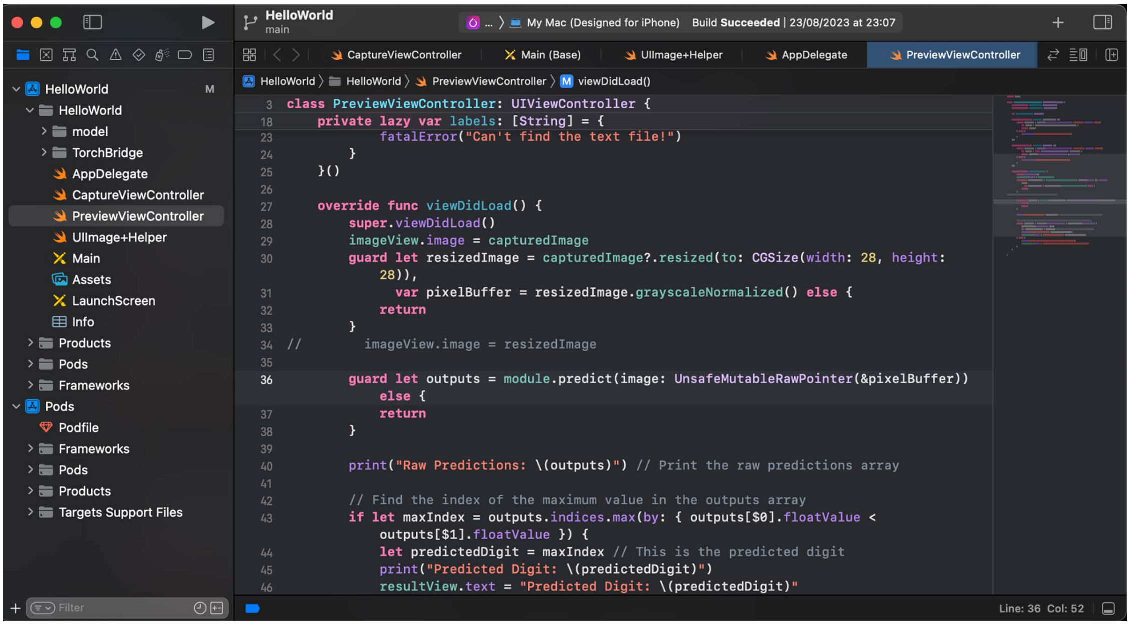
The above command essentially installs dependencies from a Podfile (much like how `pip install -r requirements.txt` installs dependencies from a `requirements.txt` file). In the current working directory, we have a Podfile [15] that has the following contents:

```
platform :ios, '12.0'
target 'HelloWorld' do
    pod 'LibTorch-Lite', '~> 1.13.0.1'
end
```

The `pod install` command therefore installs the PyTorch C++ library (libtorch), which is necessary to run ML predictions in our iOS app. This command should produce an output similar to the following on your terminal screen:

```
Analyzing dependencies
Adding spec repo 'trunk' with CDN 'https://cdn.cocoapods.org/'
Downloading dependencies
Installing LibTorch-Lite (1.10.0)
Generating Pods project
Integrating client project
[!] Please close any current Xcode sessions and use 'HelloWorld.xcworkspace'
for this project from now on.
Pod installation complete! There is 1 dependency from the Podfile and 1 total
pod installed.
```

As the above output says, let's open `HelloWorld.xcworkspace` (from the current working directory) in Xcode. You shall see the Xcode IDE window looking similar to *Figure 14.9*.



```
class PreviewViewController: UIViewController {
    private lazy var labels: [String] = {
            fatalError("Can't find the text file!")
        }
    }()

    override func viewDidLoad() {
        super.viewDidLoad()
        imageView.image = capturedImage
        guard let resizedImage = capturedImage?.resized(to: CGSize(width: 28, height:
            28)),
             var pixelBuffer = resizedImage.grayscaleNormalized() else {
            return
        }
//          imageView.image = resizedImage

        guard let outputs = module.predict(image: UnsafeMutableRawPointer(&pixelBuffer))
            else {
            return
        }

        print("Raw Predictions: \(outputs)") // Print the raw predictions array

        // Find the index of the maximum value in the outputs array
        if let maxIndex = outputs.indices.max(by: { outputs[$0].floatValue <
            outputs[$1].floatValue }) {
            let predictedDigit = maxIndex // This is the predicted digit
            print("Predicted Digit: \(predictedDigit)")
            resultView.text = "Predicted Digit: \(predictedDigit)"
```

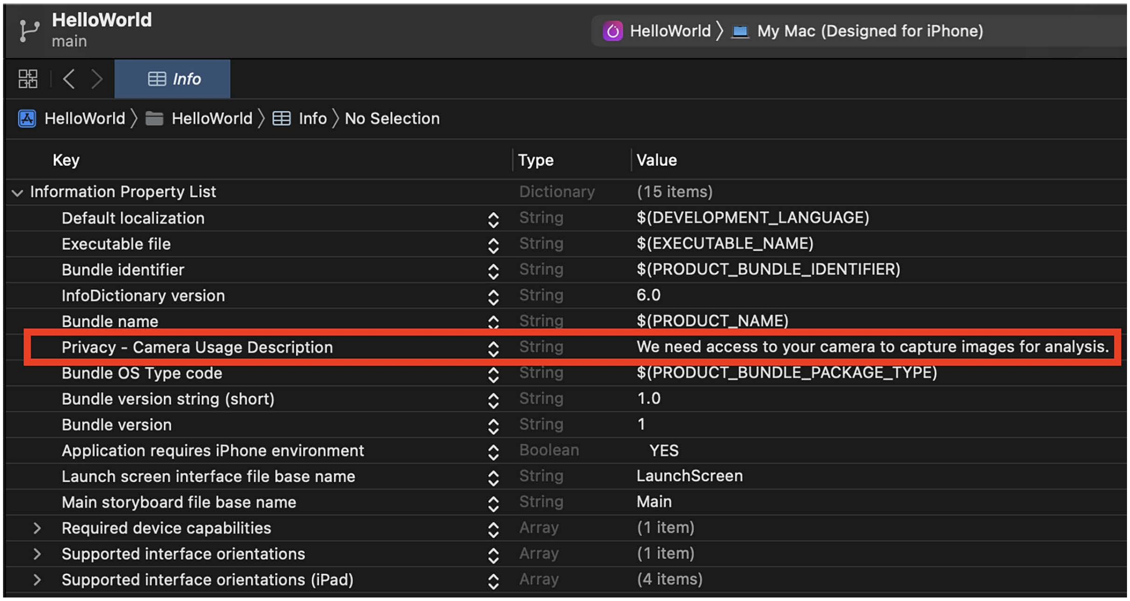
Figure 14.9: Xcode window showing our HelloWorld project

On the leftmost side of *Figure 14.9*, we can see the `model` folder, which contains the mobile-optimized MNIST model object [16]. We have copied this model from the Android folder to the iOS folder with the following command:

```
cp ../../Android/app/src/main/assets/optimized_for_mobile_traced_model.pt
HelloWorld/model/model.pt
```

In the leftmost part of *Figure 14.9*, we can also see an `Info.plist` file [17]. This file is a configuration file that contains essential metadata and configuration information about your iOS app. It's written in XML format and is used to provide details to the operating system and the App Store about how your app should behave.

To enable camera access, we have added an entry to this file as shown in *Figure 14.10*.


HelloWorld
main
HelloWorld 〉 My Mac (Designed for iPhone)
Info
HelloWorld 〉 HelloWorld 〉 Info 〉 No Selection

Key	Type	Value
Information Property List	Dictionary	(15 items)
Default localization	String	$(DEVELOPMENT_LANGUAGE)
Executable file	String	$(EXECUTABLE_NAME)
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	$(PRODUCT_NAME)
Privacy - Camera Usage Description	String	We need access to your camera to capture images for analysis.
Bundle OS Type code	String	$(PRODUCT_BUNDLE_PACKAGE_TYPE)
Bundle version string (short)	String	1.0
Bundle version	String	1
Application requires iPhone environment	Boolean	YES
Launch screen interface file base name	String	LaunchScreen
Main storyboard file base name	String	Main
Required device capabilities	Array	(1 item)
Supported interface orientations	Array	(1 item)
Supported interface orientations (iPad)	Array	(4 items)

Figure 14.10: The Info.plist file displaying various app configuration settings and highlighting the camera access request, along with the justification

In iOS app development, we need to justify accessing the phone camera, as can be seen on the right side of the highlighted portion of *Figure 14.10*. Now that we have set up our Xcode project and taken care of the phone camera access, let us look into the code that handles the image capture functionality of our app in the next section.

Using a phone camera in the iOS app to capture images

In *Figure 14.9*, on the left side, we can see a few different files containing source code. These are Swift files, where Swift is a programming language that is typically used to write source code for iOS apps. The file that takes care of handling camera capture for our app is (as the name suggests) `CaptureViewController.swift` [18]. This file contains the `CaptureViewController` class, which lists various objects and methods needed to handle the camera capture flow, as shown in the high-level code snippet below:

```
class CaptureViewController: UIViewController, AVCapturePhotoCaptureDelegate {
    @IBOutlet var captureButton: UIButton!
    @IBOutlet var imageView: UIImageView!
    private var captureSession: AVCaptureSession!
    private var photoOutput: AVCapturePhotoOutput!
    private var previewLayer: AVCaptureVideoPreviewLayer!
    private var capturedImage: UIImage?
    func viewDidLoad() {...}
    func setupCamera() {...}
    @IBAction func captureButtonTapped(_ sender: UIButton) {...}
    func photoOutput(_ output: AVCapturePhotoOutput, didFinishProcessingPhoto
photo: AVCapturePhoto, error: Error?) {...}
    ...
}
```

In the above code:

1. First, we define an object linked to the capture button on the camera capture screen – `captureButton`.
2. Then, there is an object linked to the dynamic camera output display on the screen – `imageView`.
3. Next, we have a session object – `captureSession` – which manages real-time capture of incoming image streams from the phone camera.
4. We then have a `photoOutput` object, which stores the raw captured photo.
5. Then, there is a `previewLayer` object, which renders the captured photo to the user.
6. And then, we have a `capturedImage` object, which stores the processed form of the raw captured photo (stored in `photoOutput`).

These objects are followed by several declared methods, starting with `viewDidLoad`. The `viewDidLoad` method is used for the initial setup and configuration of a given view, in this case, the camera capture view (or screen). This method contains the following code:

```
override func viewDidLoad() {
        super.viewDidLoad()
        setupCamera()
    }
```

So, this method in turn calls another function within the `CaptureViewController` class – `setupCamera`. The `setupCamera` method first ascertains if the app indeed has access to the phone camera. Next, it initializes the `captureSession`, `previewLayer`, and `photoOutput` objects necessary to run the camera capture flow. The function ends with running the camera capture session, as shown in the following lines of code:

```
func setupCamera() {
        captureSession = AVCaptureSession()
        guard let captureDevice = AVCaptureDevice.default(for: .video) else {
            fatalError("Cannot access camera.")
        }
        do {
            let input = try AVCaptureDeviceInput(device: captureDevice)
            captureSession.addInput(input)
            photoOutput = AVCapturePhotoOutput()
            captureSession.addOutput(photoOutput)
            previewLayer = AVCaptureVideoPreviewLayer(session: captureSession)
            previewLayer.videoGravity = .resizeAspectFill
            // Maintain aspect ratio
            // Calculate the square frame that fits within the screen bounds
            let previewFrame = ...
            previewLayer.frame = previewFrame
            view.layer.addSublayer(previewLayer)
            captureSession.startRunning()
        } catch {
            fatalError("Cannot set up camera.")
        }
    }
```

The above code defines the logic for capturing an incoming image stream into the `photoOutput` object and displaying the camera capture in the `previewLayer`. Next, we have the `captureButtonTapped` method, which connects the pressing of the camera capture button to the storing of the captured photo in the `photoOutput` variable:

```
@IBAction func captureButtonTapped(_ sender: UIButton) {
        let settings = AVCapturePhotoSettings()
        photoOutput.capturePhoto(with: settings, delegate: self)
    }
```

Next, we have the `photoOutput` method, which converts the raw captured `photoOutput` object to the final `capturedImage` object and shows the captured image on the `previewLayer`:

```
func photoOutput(_ output: AVCapturePhotoOutput, didFinishProcessingPhoto
photo: AVCapturePhoto, error: Error?) {
        if let imageData = photo.fileDataRepresentation(), let image =
UIImage(data: imageData) {
            capturedImage = cropImage(image, to: previewLayer.frame)
            performSegue(withIdentifier: "showImagePreview", sender: self)
        }
    }
```

The above code essentially converts the captured photo stream to an image data representation and performs cropping on this image to fit the limits of the phone screen.

This completes the work needed for handling camera capture logic for our app. In the next section, we will learn how to use the camera-captured image to perform ML model inference and display the model prediction on the iOS app.

Running ML model inference on camera-captured images

The most important and relevant source code for our AI-powered iOS app lives in the `PreviewViewController.swift` file [19].

This file contains the PreviewViewController class, which lists several key objects and methods that are crucial for our app to run and display ML model prediction on the captured image. This class contains the following high-level components:

```
class PreviewViewController: UIViewController {
    @IBOutlet var imageView: UIImageView!
    @IBOutlet var resultView: UITextView!
    var capturedImage: UIImage?
    private lazy var module: TorchModule = {...}
    private lazy var labels: [String] = {...}
    func viewDidLoad() {...}
```

First, we have the imageView object, which displays the captured image in the **Preview** screen. In the same *Preview* screen, we display the ML model prediction under the resultView object. The capturedImage object initialized here is the same object that we have stored the final captured image in, in the Capture view. We will use the capturedImage object as input to our ML model for inference.

Next, we define a module variable where we load our mobile-optimized MNIST model object:

```
private lazy var module: TorchModule = {
        if let filePath = Bundle.main.path(forResource: "model", ofType: "pt"),
            let module = TorchModule(fileAtPath: filePath) {
            return module
        } else {
            fatalError("Can't find the model file!")
        }
    }()
```

We first check if the model file is available at the desired location, and then load the model as a TorchModule object.

Note

Using TorchModule in this step is the reason why we needed to install libtorch for this project, using pod install. TorchModule offers benefits such as simplified serialization, efficient device-agnostic computation, and straightforward integration with the broader PyTorch ecosystem for streamlined model deployment and execution.

Then, we define a labels variable of type String, which is used for mapping the raw numerical output from our ML model into one of the 10 classes (digits 0 to 9) and displaying the digit as a string on the app screen:

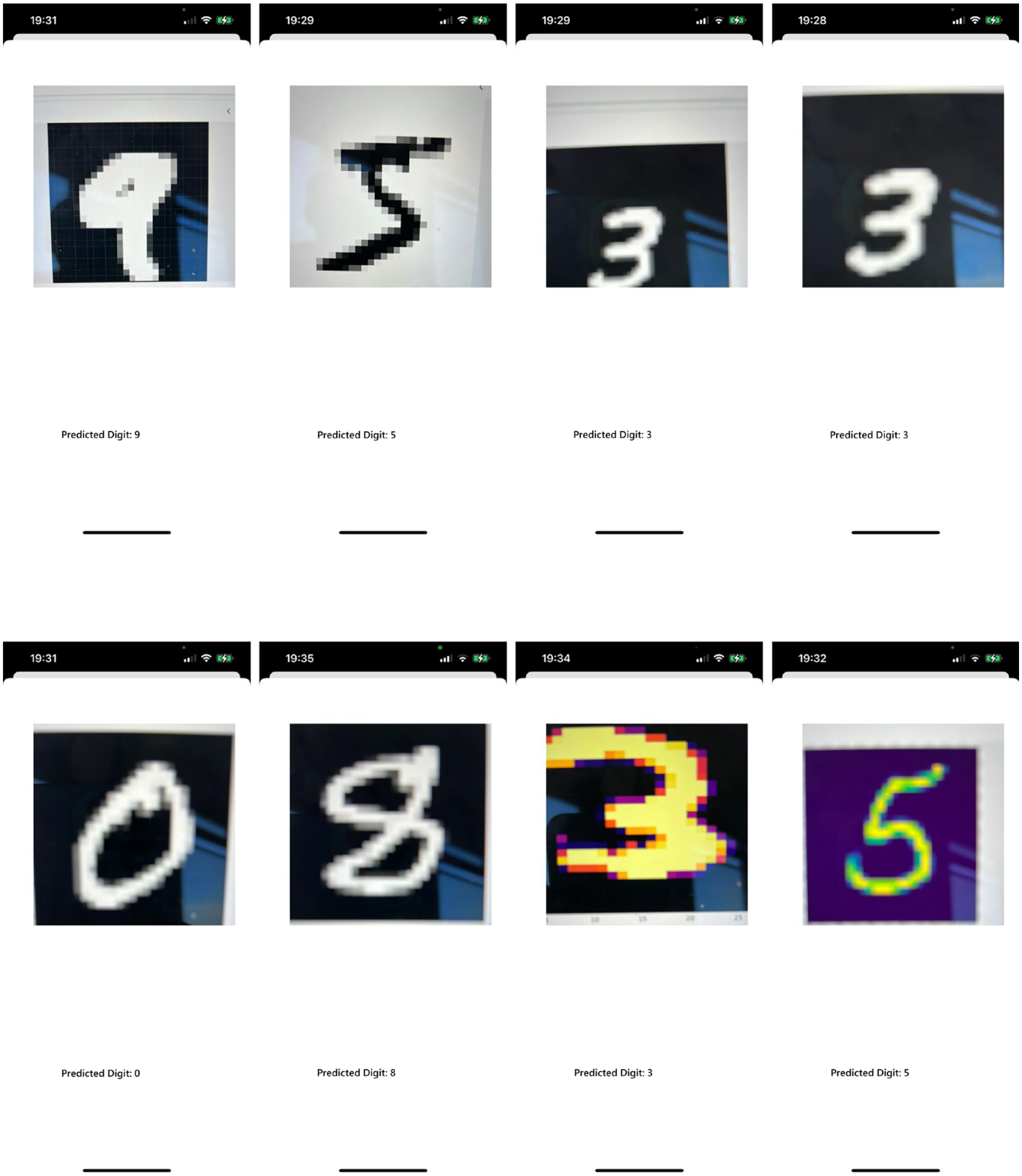
```
private lazy var labels: [String] = {
        if let filePath = Bundle.main.path(forResource: "digits", ofType:
"txt"),
      let labels = try? String(contentsOfFile: filePath) {
      return labels.components(separatedBy: .newlines)
    } else {
      fatalError("Can't find the text file!")
    }
  }()
```

The above code loads a `labels.txt` file [20] (located within the same `model` folder that contains the `model.pt` file), which simply lists the digits `0` to `9` (one digit per line). Finally, we have the `viewDidLoad` method, where we place the logic for running model inference on the captured image and displaying the prediction on screen:

```
override func viewDidLoad() {
    super.viewDidLoad()
    imageView.image = capturedImage
    guard let resizedImage = capturedImage?.resized(to: CGSize(width: 28,
height: 28)),
       var pixelBuffer = resizedImage.grayscaleNormalized() else {
      return
    }
    guard let outputs = module.predict(image:
UnsafeMutableRawPointer(&pixelBuffer)) else {
      return
    }
    print("Raw Predictions: \(outputs)") // Print the raw predictions array
    // Find the index of the maximum value in the outputs array
    if let maxIndex = outputs.indices.max(by: { outputs[$0].floatValue <
outputs[$1].floatValue }) {
      let predictedDigit = maxIndex // This is the predicted digit
      print("Predicted Digit: \(predictedDigit)")
      resultView.text = "Predicted Digit: \(predictedDigit)"
    } else {
      print("Unable to determine predicted digit")
      resultView.text = "Unable to determine predicted digit"
    }
  }
```

In this method, first, the image is resized to (28, 28) pixels using the `resized` method, and then, using the `grayscaleNormalized` method, the resized image is converted to a grayscale image and its pixel values are normalized using the mean and standard deviation values based on the MNIST dataset. Both the `resized` and `grayscaleNormalized` methods are defined in the `UIImage+Helper.swift` file [21].

We then run `module.predict` (model inference) on the normalized image to produce the class probabilities of our MNIST model. The raw probabilities are logged for debugging and the class (digit) with the highest probability is converted into the corresponding string value using the `labels` variable. This string value is then shown under the `resultView` object on the *Preview* screen. *Figure 14.11* shows examples of the *Preview* screen where the ML model correctly predicts the digit from the captured image.



Predicted Digit: 9

Predicted Digit: 5

Predicted Digit: 3

Predicted Digit: 3

Predicted Digit: 0

Predicted Digit: 8

Predicted Digit: 3

Predicted Digit: 5

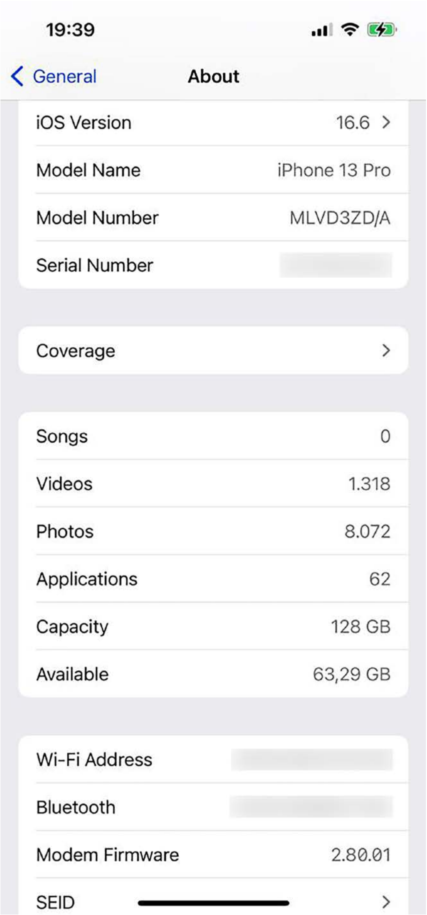
Figure 14.11: Demonstration of the Preview screen of the iOS app with different examples of handwritten digit images captured by the phone camera, where the MNIST model makes correct digit predictions

As we can see in *Figure 14.11*, the iOS app can produce correct results for different images using the underlying mobile-optimized MNIST model.

Note

Please use iOS version 12.0 or higher to work on this app.

The app can be launched on an iPhone by connecting the iPhone either wirelessly or via a USB cable to the MacBook where you are building the app on Xcode. Once connected, you should be able to see your device in the dropdown at the top-center area of the Xcode window (see the tab in *Figure 14.9* that reads **My Mac (Designed for iPhone)**). Select your device and press the **Run** button in the top-left panel of the Xcode IDE. This should first build the project and thereafter launch the app on your iPhone. For the visuals shown in *Figure 14.11*, I have used an iPhone 13 Pro with iOS version 16.6. You can find details of the device used for this exercise in *Figure 14.12*.



iOS Version	16.6 >
Model Name	iPhone 13 Pro
Model Number	MLVD3ZD/A
Serial Number	[redacted]
Coverage	>
Songs	0
Videos	1.318
Photos	8.072
Applications	62
Capacity	128 GB
Available	63,29 GB
Wi-Fi Address	[redacted]
Bluetooth	[redacted]
Modem Firmware	2.80.01
SEID	>

Figure 14.12: Details of the iPhone used for deploying the handwritten digits classification app

This concludes the development and deployment of a PyTorch-powered iOS app that performs image classification on camera-captured images. The PyTorch iOS webpage [22] is an excellent resource for learning how to build other types of ML models into your iOS apps. I hope you will now be able to use PyTorch more easily and effectively to develop AI-powered mobile applications.

Summary

In this chapter, we first learned about PyTorch Mobile and how it can be used to convert traced PyTorch model artifacts into optimized model objects that can run on mobile devices. We then learned how to build an Android app that uses PyTorch Mobile to classify images of handwritten digit captured by a phone camera using a pre-trained MNIST model. We then repeated this exercise for iOS, where we built an iOS app, again from scratch, to classify images of handwritten digits into one of 10 classes. In the next chapter, we will discuss various tools and libraries such as fastai and PyTorch Lightning that speed up and simplify the process of model training in PyTorch. We will also learn how to profile PyTorch code to understand resource utilization, using PyTorch profiler.

Reference list

1. GitHub 1: `https://pytorch.org/mobile/home/`
2. GitHub 2: `https://github.com/arj7192/MasteringPyTorchV2/tree/main/Chapter14/Android`
3. GitHub 3: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter14/Android/mobile_optimized_model.py`
4. Android Studio: `https://developer.android.com/studio`
5. Install JDK 21: `https://www3.ntu.edu.sg/home/ehchua/programming/howto/jdk_howto.html`
6. GitHub 4: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter14/Android/settings.gradle`
7. GitHub 5: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter14/Android/app/build.gradle`
8. GitHub 6: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter14/Android/app/src/main/AndroidManifest.xml`
9. GitHub 7: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter14/Android/app/src/main/java/org/pytorch/mastering_pytorch_v2_mnist/MainActivity.java`
10. PyTorch Mobile for Android: `https://pytorch.org/mobile/android/`
11. GitHub 8: `https://github.com/arj7192/MasteringPyTorchV2/tree/main/Chapter14/iOS`
12. Xcode 15: `https://developer.apple.com/xcode/`
13. GitHub 9: `https://github.com/arj7192/MasteringPyTorchV2/tree/main/Chapter14/iOS/HelloWorld`
14. CocoaPods: `https://cocoapods.org/`
15. GitHub 10: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter14/iOS/HelloWorld/Podfile`

16. GitHub 11: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter14/iOS/HelloWorld/HelloWorld/model/model.pt`
17. GitHub 12: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter14/iOS/HelloWorld/HelloWorld/Info.plist`
18. GitHub 13: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter14/iOS/HelloWorld/HelloWorld/CaptureViewController.swift`
19. GitHub 14: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter14/iOS/HelloWorld/HelloWorld/PreviewViewController.swift`
20. GitHub 15: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter14/iOS/HelloWorld/HelloWorld/model/digits.txt`
21. GitHub 16: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter14/iOS/HelloWorld/HelloWorld/UIImage%2BHelper.swift`
22. PyTorch Mobile for iOS: `https://pytorch.org/mobile/ios/`

Learn more on Discord

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

`https://packt.link/mastorch`

15

Rapid Prototyping with PyTorch

In the preceding chapters, we saw multiple facets of PyTorch as a Python library. We saw its use to train vision and text models. We learned about its extensive **application programming interfaces** (**APIs**) to load and process datasets. We explored the model inference support provided by PyTorch. We also noticed the interoperability of PyTorch across programming languages such as C++ as well as with other deep learning libraries (such as TensorFlow).

To accommodate all of these features, PyTorch provides a rich and extensive family of APIs, which makes it one of the best deep learning libraries of all time. However, the vast expanse of those features also makes PyTorch a heavy library, and this can sometimes intimidate users when performing streamlined or simple model training and testing tasks.

This chapter is focused on introducing some of the libraries that are built on top of PyTorch and aimed at providing intuitive and easy-to-use APIs, helping us build quick model training and testing pipelines with just a few lines of code. We will first discuss fastai, which is one of the most popular high-level deep learning libraries.

We will demonstrate how fastai helps speed up the deep learning research process as well as make deep learning accessible to all levels of expertise. Next, we will look at **PyTorch Lightning**, which provides the ability to use the exact same code for training on any hardware configuration, be it multiple **central processing units** (**CPUs**), **graphics processing units** (**GPUs**), or even **tensor processing units** (**TPUs**).

There are other such libraries too—such as **Poutyne**, **PyTorch Ignite**, and more—that aim to achieve similar goals. We won't fully cover them here, but we will provide GitHub code for Poutyne as an exercise to train PyTorch models with just a few lines of code.

Finally, we will explore PyTorch Profiler and use it to understand CPU processing, GPU processing, and memory consumption of a PyTorch model during inference. This chapter should familiarize you with the high-level deep learning libraries that can be extremely useful to rapidly prototype your deep learning models, as well as enable you to dissect your model's performance during inference.

By the end of this chapter, you will be able to use fastai, PyTorch Lightning, Poutyne, and PyTorch Profiler in your own deep learning projects, and you will hopefully see a significant reduction in the amount of time spent on model training and testing.

This chapter is broken down into the following topics:

- Using fastai, to set up model training in a few minutes
- Training models on any hardware using PyTorch Lightning
- Profiling MNIST model inference using PyTorch Profiler

Using fastai to set up model training in a few minutes

In this section, we will use the fastai library [1] to train and evaluate a handwritten digit classification model in fewer than 10 lines of code. We will also use fastai's **interpretability** module to understand where the trained model fails to perform well. The full code for the exercise can be found in our GitHub repository [2].

Setting up fastai and loading data

In this section, we will first import the fastai library, then load the `MNIST` dataset, and finally, preprocess the dataset for model training. We'll proceed as follows:

1. First, we will import fastai in the recommended way, as shown here:

   ```
   import os
   from fast.ai.vision.all import *
   ```

   Although `import *` is not the recommended way of importing libraries in Python, the fastai documentation suggests this format [3].

   Basically, this line of code imports some of the key modules from the fastai library that are usually necessary and mostly sufficient for a user to perform model training and evaluation.

2. Next, by using fastai's ready-to-use data modules, we will load the `MNIST` dataset, which is among the provided list of datasets under the fastai library [4], as shown below:

   ```
   path = untar_data(URLs.MNIST)
   print(path)
   ```

   This code should produce an output similar to the following:

   ```
   /Users/ashish.jha/.fastai/data/mnist_png
   ```

   This is where the dataset will be stored, just so we know for future purposes.

3. We can now look at a sample image path under the stored dataset, so as to understand how the dataset is laid out, using the following code:

```
files = get_image_files(path/"training")
print(len(files))
print(files[0])
```

   This should output as follows:

```
60000
/Users/ashish.jha/.fastai/data/mnist_png/training/9/36655.png
```

   There are a total of 60,000 images in the training dataset. As we can see, inside the `training` folder, there is a `9` subfolder that refers to the digit 9, and inside that subfolder are images of the digit 9.

   Note

   fastai functions such as `get_image_path`, `untar_data`, and so on are specific to the fastai library and are not available under PyTorch. In that regard, fastai's APIs are quite isolated from PyTorch and other PyTorch frameworks such as PyTorch Lightning.

4. Using the information gathered in the preceding step, we can generate labels for the `MNIST` dataset. We first declare a function that takes an image path and uses its parent folder's name to derive the digit (class) that the image belongs to. Using this function and the `MNIST` dataset path, we instantiate a `DataLoader`, as shown in the following piece of code:

```
def label_func(f): return f.parent.name
dls = ImageDataLoaders.from_path_func(
    path, fnames=files, label_func=label_func, num_workers=0)
dls.show_batch()
```

We set num_workers to 0 while creating the DataLoader object. This means that the data will be loaded in the main process, which ensures stable memory consumption across training epochs. The above code should output something like this:

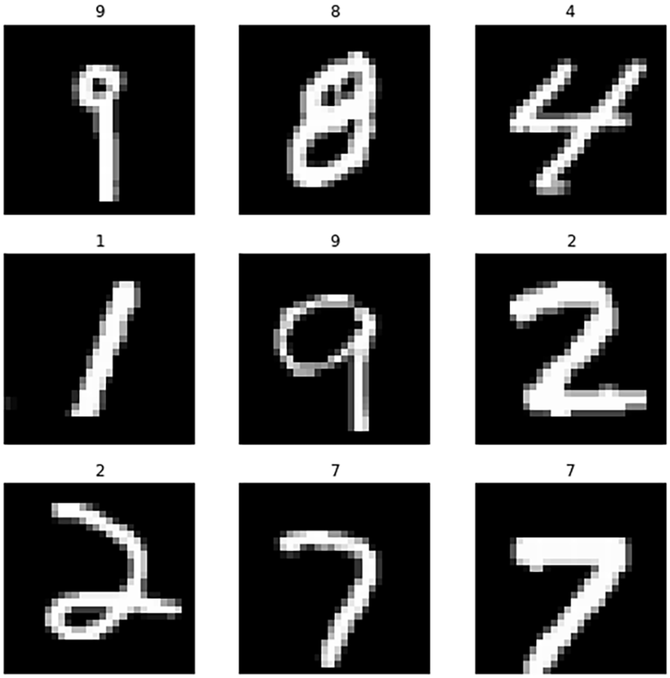


Figure 15.1: fastai batch display

As we can see, the Dataloader is correctly set up, and we are now ready to move on to model training, which we will do in the next section.

Training an MNIST model using fastai

Using the DataLoader created in the preceding section, we will now train a model with fastai using three lines of code, as follows:

1. First, we use fastai's vision_learner module to instantiate our model. Instead of defining the model architecture from scratch, we use resnet18 as the base architecture. fastai provides an extensive list of base architectures for computer vision tasks [5].

   Also, feel free to review the CNN model architecture details provided in *Chapter 2*, *Deep CNN Architectures*.

2. Next, we also define the metric that the model training logs should contain. Before actually training the model, we use fastai's **learning rate finder** to suggest a good learning rate for this model architecture and dataset combination [6]. The code for this step is shown here:

```
learn = vision_learner(dls, arch=resnet18, metrics=accuracy)
learn.lr_find()
```

It should output something like this:

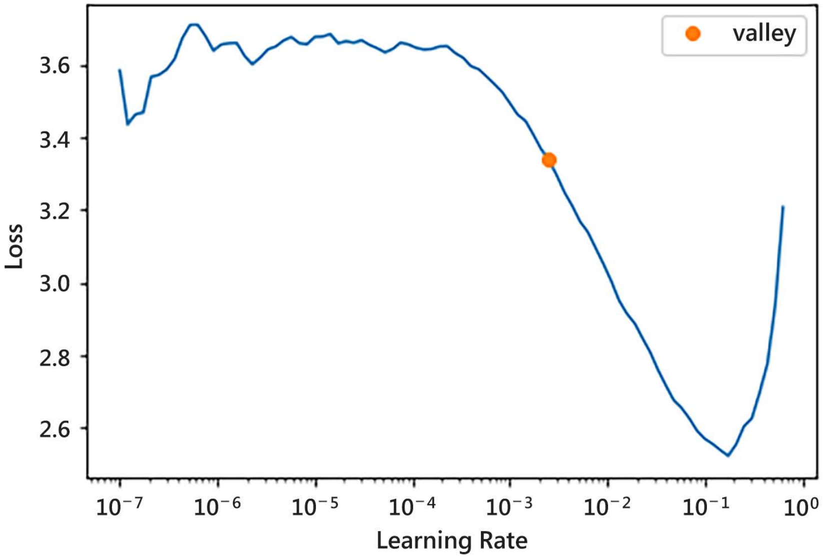


Figure 15.2: Learning rate finder output

The learning rate finder essentially does model training with varying learning rates per iteration, starting from a low value and ending with a high value. It then plots the loss for each of those iterations against the corresponding learning rate value. As we can see in this plot, a learning rate of `0.0025` is optimal. Hence, we will choose this as our base learning rate value for model training.

3. We are now ready to train our model. We could use `learn.fit` to train the model from scratch, but to aim for a better performance, we will fine-tune a pre-trained `resnet18` model using the `learn.fine_tune` method, as shown in the following line of code:

```
learn.fine_tune(epochs=2, base_lr=0.0025, freeze_epochs=1)
```

Here, `freeze_epochs` refers to the number of epochs the model is trained on initially with a frozen network, where only the last layer is unfrozen. `epochs` refers to the number of epochs the model is trained on thereafter, by unfreezing the entire `resnet18` network. The code should output something like this:

```
epoch    train_loss    valid_loss    accuracy    time
0    0.748856    0.509900    0.838667    06:12
epoch    train_loss    valid_loss    accuracy    time
0    0.107178    0.077484    0.977583    12:40
1    0.058555    0.044029    0.987167    12:38
```

As we can see, there is a first epoch of training with the frozen network, and then there are two subsequent epochs of training with the unfrozen network. We also see the accuracy metric in the logs, which we declared as our metric in *step 2*. The training logs look reasonable, and it looks like the model is indeed learning the task. In the next and final part of this exercise, we will look at the performance of this model on some samples and try to understand where it fails.

Evaluating and interpreting the model using fastai

We will first look at how the trained model performs on some of the sample images and, finally, explore the top mistakes made by the model in order to understand the scope for improvement. We'll proceed as follows:

1. With the trained model, we can use the `show_results` method to look at some of the model's predictions, as shown in the following line of code:

   ```
   learn.show_results()
   ```

   It should output something like this:

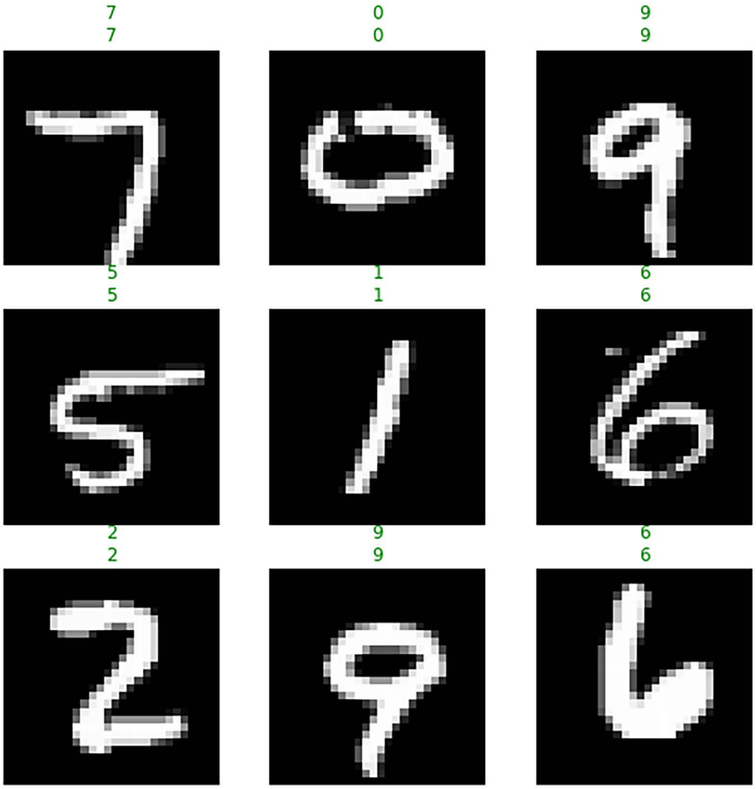


Figure 15.3: fastai sample predictions

In the preceding screenshot, we can see that the model has got all nine images right. Because the accuracy of the trained model is already at 99%, we will need 100 images to look at a wrong prediction. We will instead look exclusively at the mistakes made by the model in the next step.

2. In *Chapter 17*, *PyTorch and Explainable AI*, we will learn in detail about **model interpretability**. One of the ways of trying to understand how a trained model works is to look at where it fails the most. Using fastai's `Interpretation` module, we can do that in two lines of code, as shown here:

   ```
   interp = Interpretation.from_learner(learn)
   interp.plot_top_losses(9, figsize=(15,10))
   ```

   This should output the following:

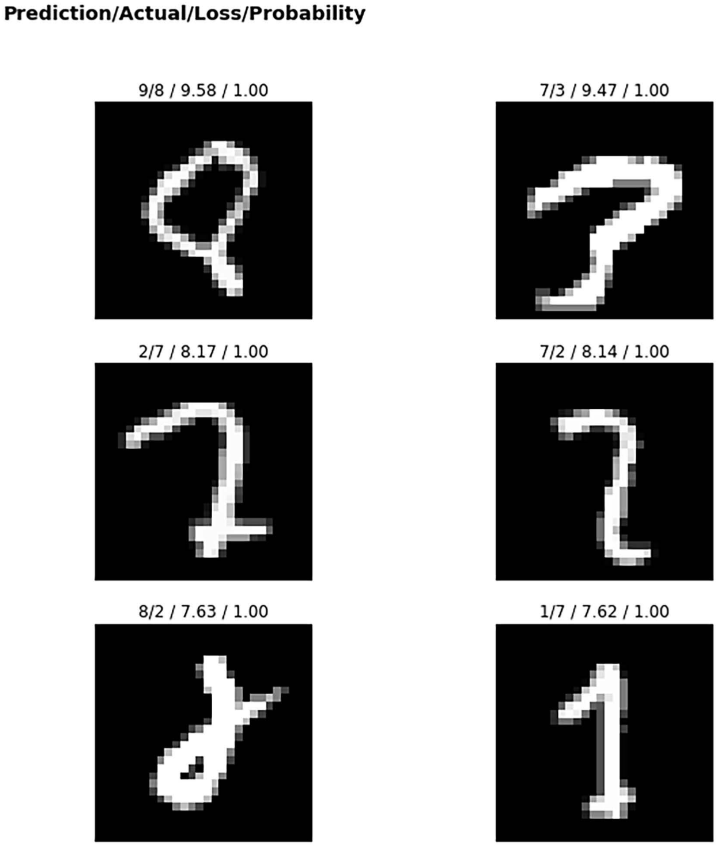


Figure 15.4: fastai top model mistakes

In *Figure 15.4*, we can see that each image is titled with the prediction, ground truth, cross-entropy loss, and prediction probability. Most of these cases are hard/wrong even for humans, and hence it is acceptable for the model to make mistakes. But for cases such as the one at the bottom right, the model is just plain wrong. This type of analysis can then be followed up by further dissecting the model for such curious cases, as we did in the previous chapter.

This concludes the exercise and our discussion on fastai. fastai has a lot to offer for machine learning engineers and researchers—both beginners and advanced users. This exercise was aimed at demonstrating fastai's speediness and ease of use. Lessons from this section can be used to work on other machine learning tasks with fastai. Under the hood, fastai uses PyTorch's functionalities, and therefore, it is always possible to switch between these two frameworks.

In the next section, we will explore another such library that sits on top of PyTorch and facilitates users to train models with relatively few lines of code, rendering the code hardware-agnostic.

Training models on any hardware using PyTorch Lightning

PyTorch Lightning (now known as Lightning) [7] is yet another library that is built on top of PyTorch to abstract out the boilerplate code needed for model training and evaluation. A special feature of this library is that any model training code written using PyTorch Lightning can be run without changes on any hardware configuration, such as multiple CPUs, multiple GPUs, or even multiple TPUs.

In the following exercise, we will train and evaluate a handwritten digit classification model using PyTorch Lightning on CPUs. You can use the same code for training on GPUs or TPUs. The full code for the following exercise can be found in our GitHub repository [8].

Defining the model components in PyTorch Lightning

In this part of the exercise, we will demonstrate how to initialize the `model` class in PyTorch Lightning. This library works on the philosophy of *self-contained model systems*—that is, the `model` class contains not only the model architecture definition but also the optimizer definition and dataset loaders, as well as the training, validation, and test set performance computation functions, all in one place.

We'll proceed as follows:

1. First, we need to import the relevant modules, as follows:

```
import torch
import torch.nn as nn
from torch.nn import functional as F
from torch.utils.data import DataLoader
from torchvision.datasets import MNIST
from torchvision import transforms
import pytorch_lightning as pl
```

   As we can see, PyTorch Lightning still uses a lot of native PyTorch modules for the model class definition. We have additionally imported the `MNIST` dataset straight from the `torchvision.datasets` module to train the handwritten digit classifier on.

2. Next, we define the PyTorch Lightning model class, which contains everything that is needed to train and evaluate our model. Let's first look at the model architecture-related methods of the class, as follows:

   ```
   class ConvNet(pl.LightningModule):
       def __init__(self):
           super(ConvNet, self).__init__()
           self.cn1 = nn.Conv2d(1, 16, 3, 1)
           ...
           self.fc2 = nn.Linear(64, 10)
       def forward(self, x):
           x = self.cn1(x)
           ...
           op = F.log_softmax(x, dim=1)
           return op
   ```

   These two methods—`__init__` and `forward`—work in the same manner as they did with the native PyTorch code.

3. Next, let's look at the other methods of the model class, as follows:

   ```
       def training_step(self, batch, batch_num):
           ...
       def validation_step(self, batch, batch_num):
           ...
       def validation_epoch_end(self, outputs):
           ...
       def test_step(self, batch, batch_num):
           ...
       def test_epoch_end(self, outputs):
           ...
       def configure_optimizers(self):
           return torch.optim.Adadelta(self.parameters(), lr=0.5)
       def train_dataloader(self):
           ...
       def val_dataloader(self):
           ...
       def test_dataloader(self):
           ...
   ```

While methods such as training_step, validation_step, and test_step are meant to evaluate per-iteration performances on the training, validation, and test sets, the *_epoch_end methods compute the per-epoch performances. There are the *_dataloader methods for the training, validation, and test sets. And finally, there is the configure_optimizer method, which defines the optimizer to be used to train the model.

Training and evaluating the model using PyTorch Lightning

Having set up the model class, we will now train the model in this part of the exercise. Then, we will evaluate the performance of the trained model on the test set.

We'll proceed as follows:

1. **Instantiating the model object:** Here, we will first instantiate the model object using the model class defined in *step 2* of the previous section—*Defining the model components in PyTorch Lightning*. We will then use the Trainer module from PyTorch Lightning to define a trainer object.

   Note that we rely on CPUs only for model training. However, you can easily switch to GPUs or TPUs. The beauty of PyTorch Lightning lies in the fact that you can add an argument such as gpus=8 or tpus=2 in the trainer definition code depending on your hardware settings, and the entire code will still run without any further modifications.

   We will begin the model training process with the following lines of code:

   ```
   model = ConvNet()
   trainer = pl.Trainer(progress_bar_refresh_rate=20, max_epochs=10)
   trainer.fit(model)
   ```

   It should output something like this:

   ```
   GPU available: False, used: False
   TPU available: False, using: 0 TPU cores
   IPU available: False, using: 0 IPUs
   HPU available: False, using: 0 HPUs
     | Name | Type      | Params
   0 | cn1  | Conv2d    | 160
   1 | cn2  | Conv2d    | 4.6 K
   2 | dp1  | Dropout2d | 0
   3 | dp2  | Dropout2d | 0
   4 | fc1  | Linear    | 294 K
   5 | fc2  | Linear    | 650
   300 K     Trainable params
   ```

```
Non-trainable params
300 K     Total params
1.202     Total estimated model params size (MB)
Epoch 9: 100%
3750/3750 [01:19<00:00, 47.07it/s, loss=0.0642, v_num=2, train_loss_
step=0.000264, val_loss_step=4.39e-5, val_loss_epoch=0.0122, train_loss_
epoch=0.029]
```

First, the `trainer` object assesses the available hardware, and then it logs the entire model architecture that is to be trained, along with the number of parameters per layer in the architecture. Thereafter, it begins the model training epoch by epoch. It trains until `10` epochs, as specified using the `max_epochs` argument while defining the `trainer` object. We can also see that the training and validation losses are logged at every epoch.

2. **Testing the model:** Having trained the model for `10` epochs, we can now test it. Using the `.test` method, we request the `trainer` object, which was defined in *step 1* of this section, to run inference on the test set, as follows:

```
trainer.test()
```

It should output something like this:

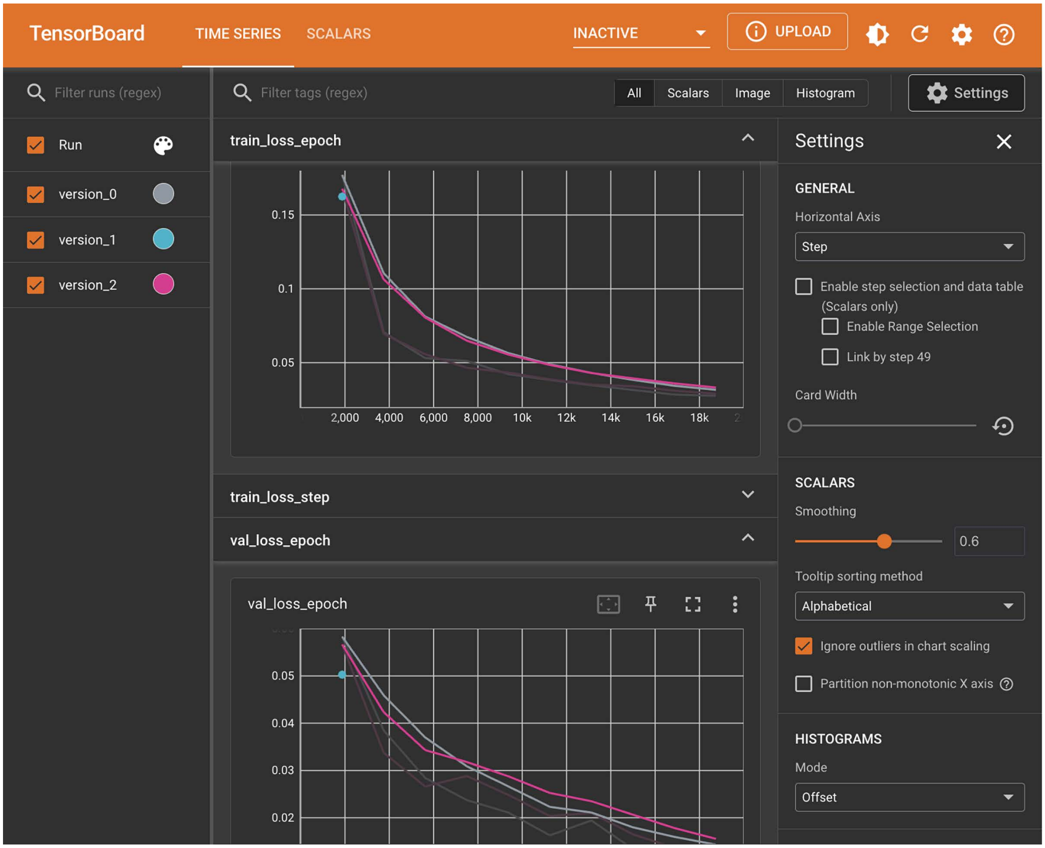
```
Testing DataLoader 0: 100%
313/313 [00:03<00:00, 92.68it/s]
────────────────────────────────────────────────────────────

       Test metric             DataLoader 0
────────────────────────────────────────────────────────────

     test_loss_epoch        0.03530178219079971
───────────────────────────────────────────────────────────

[{'test_loss_epoch': 0.03530178219079971}]
```

We can see that the model outputs the test loss using the trained model.

3. **Exploring the trained model:** Finally, PyTorch Lightning also provides a neat interface with TensorBoard [9], which is a great visualization toolkit made originally for TensorFlow. By running the following lines of code, we can explore the training, validation, and test set performance of the trained model interactively:

```
# Start TensorBoard.
%reload_ext tensorboard
%tensorboard --logdir lightning_logs/
```

This should produce an output in the notebook as shown in *Figure 15.5*.



Settings

GENERAL

Horizontal Axis

Step

☐ Enable step selection and data table (Scalars only)

☐ Enable Range Selection

☐ Link by step 49

Card Width

SCALARS

Smoothing

0.6

Tooltip sorting method

Alphabetical

☑ Ignore outliers in chart scaling

☐ Partition non-monotonic X axis

HISTOGRAMS

Mode

Offset

Figure 15.5: PyTorch Lightning TensorBoard output

Within this interactive visualization toolkit, we can look at the epoch-wise model training progress in terms of loss, accuracy, and various other metrics. This is another neat feature of PyTorch Lightning that enables us to have a rich model evaluation and debugging experience with just a few lines of code.

Note

Regular PyTorch code also provides an interface with TensorBoard, although the code is lengthier [10].

This brings us to the end of this exercise and this section. Although it is a brief overview of the PyTorch Lightning library, it should be enough to get an idea of the library, how it works, and how it can work for your projects. There are plenty more examples and tutorials available on PyTorch Lightning's documentation page [11].

If you are in the process of rapidly experimenting with various models or want to reduce the scaffolding code in your model training pipeline, this library is well worth a try.

Another useful library to rapidly protoype PyTorch models is Poutyne. Instead of diving deeper into this library in this book, we provided the MNIST model training exercise using Poutyne, similar to the previous exercises in this chapter, in our GitHub repository [12]. I encourage you to try running this exercise and explore this library.

This concludes our exploration of deep learning prototyping libraries for PyTorch. While these libraries are great for quick prototyping, they abstract away a lot of lower-level implementation details. Such lower-level details need to be tweaked when working on custom research workflows (for example, if you need to implement a custom loss function that might not be provided out of the box in such prototyping libraries). In the next section, we will switch tracks a bit, and profile PyTorch model inference code to understand our model's consumption of hardware resources during runtime, such as CPU processing, GPU processing, as well as memory.

Profiling MNIST model inference using PyTorch Profiler

The profiling of programming code is the analysis of its performance in terms of its space (memory) and time complexity, providing us with a breakdown of the time and memory consumed by the various sub-modules or functions called within the code. When we run inference using a PyTorch deep learning model, a series of such function calls are made in order to produce the output (*y*) from the input (*X*). In this section, we will learn how to profile PyTorch model inference using the PyTorch Profiler.

We will infer the MNIST model that was trained in *Chapter 1*, *Overview of Deep Learning Using PyTorch* [13], and deployed in *Chapter 13*, *Operationalizing PyTorch Models into Production* [14]. First we will run the model inference on a CPU and profile the inference to examine the CPU time and memory consumption by its various internal operations. Next, we will run model inference on the GPU and repeat the profiling exercise. Finally, we will visualize the results of profiling in the form of a chart. The code for this exercise is available on GitHub [15] in the form of a notebook, which starts by training the MNIST model. Our focus will be on the profiling section of the notebook.

Profiling on a CPU

We have with us a trained MNIST model that has the following architecture:

```
print(model)
```

```
ConvNet(
  (cn1): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1))
  (cn2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (dp1): Dropout2d(p=0.1, inplace=False)
  (dp2): Dropout2d(p=0.25, inplace=False)
  (fc1): Linear(in_features=4608, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=10, bias=True)
)
```

And we have an input data sample (*X*) to be used as input for inference:

```
print(sample_data.shape)
torch.Size([500, 1, 28, 28])
```

The input sample is essentially a batch of 500 grayscale images, each sized 28x28 pixels. Now, in the form of an exercise, we will use this model and data sample for inference and profile the model inference code using PyTorch Profiler:

1. First, we will import the relevant PyTorch Profiler libraries:

   ```
   from torch.profiler import profile, record_function, ProfilerActivity
   ```

   While `profile` is the PyTorch Profiler global context manager [16], `record_function` is used as the labeled context manager to profile each sub-task. `ProfilerActivity` is the activity class, with `CPU` and `CUDA` as the two supported activity groups to be used for profiling.

2. With libraries imported, its time to use PyTorch Profiler to analyze CPU utilization during model inference:

   ```
   with profile(activities=[ProfilerActivity.CPU],
                record_shapes=True) as prof:
       with record_function("model_inference"):
           model(sample_data)
   ```

   Because we currently focus on the CPU, the activity group is limited to it. `record_shapes` is set to `True` in order to retain the tensor shapes involved in each of the analyzed operations.

3. Finally, we can print the results of model inference profiling:

   ```
   print(prof.key_averages().table(sort_by="cpu_time_total"))
   ```

   This should produce the following output:

   ```
   ---------------------------------  ------------  ------------  ----------
   --  ------------  ------------  ------------
                                Name    Self CPU %      Self CPU   CPU total
   %     CPU total  CPU time avg    # of Calls
   ---------------------------------  ------------  ------------  ----------
   --  ------------  ------------  ------------
                     model_inference         6.06%       6.256ms
   99.99%     103.238ms     103.238ms             1
                        aten::conv2d         0.01%       8.000us
   45.74%       47.229ms      23.614ms             2
                   aten::convolution         0.03%      31.000us
   45.73%       47.221ms      23.610ms             2
   ```

```
               aten::_convolution         0.02%      17.000us
45.70%      47.190ms      23.595ms             2
         aten::mkldnn_convolution        45.67%      47.160ms
45.69%      47.173ms      23.587ms             2
                 aten::max_pool2d         0.00%       5.000us
36.06%      37.232ms      37.232ms             1
    aten::max_pool2d_with_indices        36.05%      37.227ms
36.05%      37.227ms      37.227ms             1
...
                      aten::zero_         0.00%       0.000us
0.00%       0.000us       0.000us             1
            aten::feature_dropout         0.00%       0.000us
0.00%       0.000us       0.000us             2
               aten::resolve_conj         0.00%       0.000us
0.00%       0.000us       0.000us             4
---------------------------------  ------------  ------------  ----------
--  ------------  ------------  ------------
Self CPU time total: 103.253ms
```

As we can see, the total time taken by the model to infer 500 images is 103 ms. What is more interesting is the breakdown of this runtime. Ordered in descending order, we can see that most time is taken by convolution operations—a total of 47 ms, followed by max pool operations with a total of 37 ms.

We can see multiple rows stating convolution. These rows reflect the internal (children) calls made by the public `conv_2d` class. While the `CPU total` column shows us the total time taken by a function inclusive of underlying children calls, `Self CPU` excludes the internal call times. Hence, the CPU total appears as a cumulative sum of the `Self CPU` values. The same logic applies to the CPU utilization percentage, which contains normalized values for the CPU times.

Finally, because there are two convolutional layers in the model, there are two calls made, and hence, the CPU time average is halved (23.5 ms) for convolution. In reality though, the two convolutional layers do not consume 23.5 ms each. How can we know their exact times? We will see in the next step.

4. Instead of grouping operations broadly (convolution, max pool, ReLU, and so on), which was done in the previous step by default, we can group operations more finely, by their input tensor shape:

```
print(prof.key_averages(group_by_input_shape=True).table(
    sort_by="cpu_time_total"))
```

This should produce the following output:

```
---------------------------------  ------------  ------------  ----------
--  ------------  ------------  ------------  ---------------------------
-----------------------------------------------------
                             Name    Self CPU %      Self
CPU   CPU total %     CPU total  CPU time avg    # of Calls
Input Shapes
---------------------------------  ------------  ------------  ----------
--  ------------  ------------  ------------  ---------------------------
-----------------------------------------------------
                  model_inference         6.06%       6.256ms
99.99%     103.238ms     103.238ms             1
[]
                     aten::conv2d         0.00%       4.000us
39.62%      40.910ms      40.910ms             1
[[500, 16, 26, 26], [32, 16, 3, 3], [32], [], [], [], []]
                aten::convolution         0.01%      13.000us
39.62%      40.906ms      40.906ms             1                 [[500,
16, 26, 26], [32, 16, 3, 3], [32], [], [], [], [], [], []]
               aten::_convolution         0.01%      10.000us
39.60%      40.893ms      40.893ms             1  [[500, 16, 26, 26],
[32, 16, 3, 3], [32], [], [], [], [], [], [], [], [], [], []
         aten::mkldnn_convolution        39.59%
40.875ms        39.59%      40.883ms      40.883ms             1
[[500, 16, 26, 26], [32, 16, 3, 3], [32], [], [], [], []]
                 aten::max_pool2d         0.00%       5.000us
36.06%      37.232ms      37.232ms             1
[[500, 32, 24, 24], [], [], [], [], []]
    aten::max_pool2d_with_indices        36.05%
37.227ms        36.05%      37.227ms      37.227ms             1
[[500, 32, 24, 24], [], [], [], [], []]
                     aten::conv2d         0.00%
4.000us         6.12%       6.319ms       6.319ms             1
[[500, 1, 28, 28], [16, 1, 3, 3], [16], [], [], [], []]
                aten::convolution         0.02%      18.000us
6.12%       6.315ms       6.315ms             1                   [[500,
1, 28, 28], [16, 1, 3, 3], [16], [], [], [], [], [], []]
...
                 aten::as_strided         0.00%
0.000us         0.00%       0.000us       0.000us             1
[[10], [], [], []]
```

```
               aten::resolve_conj         0.00%
0.000us         0.00%       0.000us       0.000us             1
[[500, 10]]
---------------------------------  ------------  ------------  ----------
--  ------------  ------------  ------------  --------------------------
-----------------------------------------------------
Self CPU time total: 103.253ms
```

Now, we can see that two convolutional layers are represented separately, where the second convolutional layer consumes 40 ms of CPU time in total and the first convolutional layer consumes 6 ms. This is reasonable as the input to the first convolutional layer has only 1 feature map compared to 16 feature maps as input to the second convolutional layer.

5. So far, we have focused on CPU time. But what about memory consumption? Thanks to the sleek API provided by PyTorch Profiler, we can simply add the `profile_memory=True` argument to our profiling statement, like so:

```
with profile(activities=[ProfilerActivity.CPU],
             profile_memory=True,
             record_shapes=True) as prof:
        model(sample_data)
print(prof.key_averages().table(
        sort_by="self_cpu_memory_usage"))
```

This should produce the following output:

```
---------------------------------  ------------  ------------  ----------
--  ------------  ------------  ------------  ------------  ------------
                             Name    Self CPU %      Self CPU   CPU total
%     CPU total  CPU time avg       CPU Mem   Self CPU Mem    # of Calls
---------------------------------  ------------  ------------  ----------
--  ------------  ------------  ------------  ------------  ------------
                       aten::relu         0.04%      37.000us
8.04%       8.232ms       2.744ms      55.91 Mb           0 b
3
                  aten::clamp_min         8.01%       8.195ms
8.01%       8.195ms       2.732ms      55.91 Mb      55.91 Mb
3
                     aten::conv2d         0.01%       8.000us
51.31%      52.507ms      26.253ms      55.79 Mb           0 b
2
                aten::convolution         0.04%      37.000us
51.30%      52.499ms      26.250ms      55.79 Mb           0 b
2
```

```
               aten::_convolution         0.02%      17.000us
51.27%      52.462ms      26.231ms      55.79 Mb           0 b
2
...
                      aten::copy_         0.02%      20.000us
0.02%      20.000us      10.000us           0 b           0 b
2
               aten::resolve_conj         0.00%       0.000us
0.00%       0.000us       0.000us           0 b           0 b
4
                         [memory]         0.00%       0.000us
0.00%       0.000us       0.000us    -138.22 Mb    -138.22 Mb
10
---------------------------------  ------------  ------------  ----------
--  ------------  ------------  ------------  ------------  ------------
Self CPU time total: 102.329ms
```

Interestingly, ReLU consumes the most memory. This is because the ReLU layer allocates new memory for output post-activation. This can be circumvented by using `nn.ReLU(inplace=True)` instead of `nn.ReLU()`. These are the insights and gains we get by profiling our code!

Next, we will run the model inference on the GPU and profile the code to understand GPU utilization.

Profiling model inference on the GPU

To profile model inference on the GPU, we first need to load the model and input data onto it:

```
model=model.cuda()
sample_data=sample_data.cuda()
```

And then, we simply need to add `ProfilerActivity.CUDA` to the list of activity groups to be profiled, as shown below:

```
with profile(activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA],
             record_shapes=True) as prof:
    with record_function("model_inference"):
        model(sample_data)
print(prof.key_averages().table(sort_by="cuda_time_total"))
```

This should produce the following output:

```
-------------------------------------------------------  ------------  --------
----  ------------  ------------  ------------  ------------  ------------  ---
---------  ------------  ------------
                                                   Name    Self CPU %      Self
CPU   CPU total %     CPU total  CPU time avg     Self CUDA   Self CUDA %
CUDA total  CUDA time avg    # of Calls
```

```
-------------------------------------------------------  ------------  --------
----  ------------  ------------  ------------  ------------  ------------  ---
---------  ------------  ------------
                                        model_inference         6.24%
506.000us        61.60%       4.999ms       4.999ms       0.000us         0.00%
4.406ms       4.406ms             1
                                           aten::conv2d         0.14%
11.000us        49.16%       3.989ms       1.994ms       0.000us         0.00%
3.460ms       1.730ms             2
                                      aten::convolution         0.68%
55.000us        49.02%       3.978ms       1.989ms       0.000us         0.00%
3.460ms       1.730ms             2
                                     aten::_convolution         0.55%
45.000us        48.34%       3.923ms       1.962ms       0.000us         0.00%
3.460ms       1.730ms             2
                                aten::cudnn_convolution         7.71%
626.000us        46.85%       3.802ms       1.901ms       2.898ms        65.77%
2.898ms       1.449ms             2
                                   volta_cgemm_32x32_tn         0.00%
0.000us         0.00%       0.000us       0.000us       1.186ms        26.92%
1.186ms     593.000us             2
...
                                         cudaEventQuery         0.06%
5.000us         0.06%       5.000us       5.000us       0.000us         0.00%
0.000us       0.000us             1
                                  cudaDeviceSynchronize        38.09%
3.091ms        38.09%       3.091ms       3.091ms       0.000us         0.00%
0.000us       0.000us             1
-------------------------------------------------------  ------------  --------
----  ------------  ------------  ------------  ------------  ------------  ---
---------  ------------  ------------
Self CPU time total: 8.115ms
Self CUDA time total: 4.406ms
```

Firstly, we can notice that the inference time is substantially smaller when using the GPU—here, I use one NVIDIA Tesla T4 for inference. Profiling provides a breakdown of the 4.4 ms GPU inference time. As expected, the majority of time is taken by the convolutional operation (65%).

Notice the use of the `cudnn` [17] backend by the `torch  conv2d` operation for the internal/lower-level optimized convolution operation when working with the GPU. Profiling with the CPU earlier, we saw that torch used `mkl` [18] as the backend for lower-level convolutional operation calls. Profiling can help us understand such lower-level details related to operations optimized by each hardware (CPU/GPU).

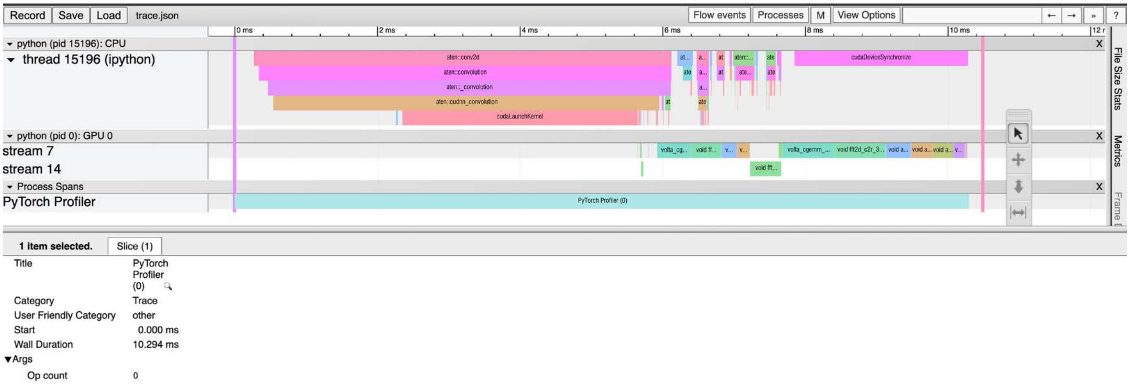
In the next and final part of the exercise we will learn to visualize the results of profiling model inference.

Visualizing model profiling results

PyTorch Profiler allows us to save the results of profiling as a `trace.json` file, which can be opened in Google Chrome and visualized as a chart. Having the model and input data loaded in the GPU, all we need to do is the following:

```
with profile(activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA]) as prof:
    model(inputs)
prof.export_chrome_trace("trace.json")
```

We can then open the JSON file in a new tab in Google Chrome at `chrome://tracing`. You should see a chart as shown in *Figure 15.6*:



1 item selected. Slice (1)

Title PyTorch Profiler (0)
Category Trace
User Friendly Category other
Start 0.000 ms
Wall Duration 10.294 ms
▾Args
Op count 0

Figure 15.6: PyTorch Profiler trace results

This concludes our discussions on using PyTorch Profiler to better understand the performance of our MNIST model during inference on the CPU as well as the GPU. Once again, PyTorch's website [19] is an excellent resource to dig deeper into the Profiler APIs.

Summary

In this chapter, we focused on both abstracting out the noisy details involved in model training code and the core components to facilitate the rapid prototyping of models. As PyTorch code can often be cluttered with a lot of such noisy detailed code components, we looked at some of the high-level libraries that are built on top of PyTorch. We also learned how to profile PyTorch code during model inference to better benchmark model performance on the CPU and GPU.

In the next chapter, we will move on to another important and promising aspect of applied machine learning that we already touched upon in both *Chapter 2*, *Deep CNN Architectures*, and *Chapter 5*, *Advanced Hybrid Models*: we will learn how to effectively use PyTorch for **Automated Machine Learning** (**AutoML**). By doing this, we will be able to use AutoML to train machine learning models automatically—that is, without having to decide on and define the model architecture.

Reference list

1. fastai library: https://docs.fast.ai/
2. fastai exercise (GitHub): https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter15/fastai.ipynb
3. Importing the fastai library: https://docs.fast.ai/quick_start.html
4. fastai library datasets: https://docs.fast.ai/data.external.html
5. fastai vision models: https://fastai1.fast.ai/vision.models.html
6. Learning rate finder: https://docs.fast.ai/callback.schedule.html#lrfinder
7. PyTorch Lightning: https://github.com/Lightning-AI/lightning
8. Training models on PyTorch Lightning (GitHub): https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter15/pytorch_lightning.ipynb
9. TensorBoard: https://www.tensorflow.org/tensorboard
10. PyTorch TensorBoard docs: https://pytorch.org/docs/stable/tensorboard.html
11. Lightning docs: https://pytorch-lightning.readthedocs.io/en/stable/
12. Poutyne exercise: https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter15/poutyne.ipynb
13. Chapter 1 GitHub: https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter01/mnist_pytorch.ipynb
14. Chapter 13 GitHub: https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter13/mnist_pytorch.ipynb
15. Profiling exercise: https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter15/pytorch_profiler.ipynb
16. PyTorch Profiler: https://pytorch.org/docs/stable/profiler.html#torch.profiler.profile
17. cuDNN: https://pytorch.org/docs/stable/backends.html#module-torch.backends.cudnn
18. MKL: https://pytorch.org/docs/stable/backends.html#module-torch.backends.mkl
19. Profiler recipes: https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html

Learn more on Discord

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

https://packt.link/mastorch

16

PyTorch and AutoML

Automated machine learning (**AutoML**) provides methods to find the optimal neural architecture and the best hyperparameter settings for a given neural network. We have already covered neural architecture search in detail while discussing the `RandWireNN` model in *Chapter 5, Hybrid Advanced Models*.

In this chapter, we will look more broadly at the AutoML tool for PyTorch—**Auto-PyTorch**—which performs both neural architecture search and hyperparameter search. We will also look at another AutoML tool called **Optuna** that performs hyperparameter search for a PyTorch model.

By the end of this chapter, non-experts will be able to design machine learning models with little domain experience, and experts will be able to drastically speed up their model selection process.

This chapter is broken down into the following topics:

- Finding the best neural architectures with AutoML
- Using Optuna for hyperparameter search

All code files for this chapter can be found at `https://github.com/arj7192/MasteringPyTorchV2/tree/main/Chapter16`.

Finding the best neural architectures with AutoML

One way to think of machine learning algorithms is that they automate the process of learning relationships between given inputs and outputs. In traditional software engineering, we would have to explicitly write/code these relationships in the form of functions that take in input and return output. In the machine learning world, machine learning models find such functions for us. Although this automation speeds the process up, there is still a lot to be done. Besides mining and cleaning data, here are a few routine tasks to be performed to get those functions:

- Choosing a machine learning model (or a model family and then a model)
- Deciding on the model architecture (especially in the case of deep learning)
- Choosing hyperparameters

- Adjusting hyperparameters based on validation set performance
- Trying different models (or model families)

These are the kinds of tasks that justify the requirement of a human machine learning expert. Most of these steps are manual and either take a lot of time or need a lot of expertise, and we have far too few machine learning experts than needed to create and deploy all the machine learning models that are becoming increasingly popular, valuable, and useful across both industry and academia.

This is where AutoML comes to the rescue. AutoML has become a discipline within the field of machine learning that aims to automate the previously listed steps and beyond.

In this section, we will take a look at Auto-PyTorch [1]—an AutoML tool created to work with PyTorch. In the form of an exercise, we will find an optimal neural network along with the hyperparameters to perform handwritten digit classification—a task that we worked on in *Chapter 1*, *Overview of Deep Learning Using PyTorch*.

The difference from the first chapter will be that, this time, we do not decide on the architecture or the hyperparameters, and instead let Auto-PyTorch figure that out for us. We will first load the dataset, then define an Auto-PyTorch model search instance, and finally, run the model searching routine, which will provide us with a best-performing model [2].

Using Auto-PyTorch for optimal MNIST model search

We will execute the model search in the form of a Jupyter notebook. In the text, we only show the important parts of the code. The full code can be found in our GitHub repository [3].

Loading the MNIST dataset

We will now discuss the code for loading the dataset step by step, as follows:

1. First, we import the relevant libraries, like this:

```
import torch
from autoPyTorch import AutoNetClassification
```

   The last line is crucial, as we import the relevant Auto-PyTorch module here. This will help us set up and execute a model search session.

2. Next, we load the training and test datasets using Torch **application programming interfaces** (**APIs**), as follows:

```
train_ds = datasets.MNIST(...)
test_ds = datasets.MNIST(...)
```

3. We then convert these dataset tensors into training and testing input (X) and output (y) arrays, like this:

```
X_train, X_test, y_train, y_test = train_ds.data.numpy().reshape(-1,
28*28), test_ds.data.numpy().reshape(-1, 28*28) ,train_ds.targets.
numpy(), test_ds.targets.numpy()
```

Note that we are reshaping the images into flattened vectors of size 784. In the next section, we will be defining an Auto-PyTorch model searcher that expects a flattened feature vector as input, and hence we do this reshaping.

Auto-PyTorch (at the time of writing) only provides support for featurized and image data in the form of `AutoNetClassification` and `AutoNetImageClassification`, respectively. While we are using featurized data in this exercise, we leave it as an exercise for you to use image data instead. A relevant image tutorial can be found here [4].

Running a neural architecture search with Auto-PyTorch

Having loaded the dataset in the preceding section, we will now use Auto-PyTorch to define a model search instance and use it to perform the tasks of neural architecture search and hyperparameter search. We'll proceed as follows:

1. This is the most important step of the exercise, where we define an `autoPyTorch` model search instance, like this:

   ```
   autoPyTorch = AutoNetClassification("tiny_cs",  # config preset
                log_level='info', max_runtime=2000, min_budget=100,
                max_budget=1500)
   ```

   The configs here are derived from the examples provided in the Auto-PyTorch repository [1]. Generally, `tiny_cs` is used for faster searches with fewer hardware requirements.

   The budget argument is all about setting constraints on resource consumption by the Auto-PyTorch process. As a default, the unit of a budget is time—that is, how much **central processing unit/graphics processing unit** (CPU/GPU) time we are comfortable spending on the model search.

2. After instantiating an Auto-PyTorch model search instance, we execute the search by trying to fit the instance on the training dataset, as follows:

   ```
   autoPyTorch.fit(X_train, y_train, validation_split=0.1)
   ```

   Internally, Auto-PyTorch will run several trials of different model architectures and hyperparameter settings based on methods mentioned in the original paper [2].

   The different trials will be benchmarked against the 10% validation dataset, and the best-performing trial will be returned as output. The command in the preceding code snippet should output the following:

   ```
   {'optimized_hyperparameter_config': ('CreateDataLoader: batch_size": 125,
   'Imputation: strategy': 'median',
   'InitializationSelector:initialization_method': 'default',
   'InitializationSelector: initializer: initialize_bias': 'No',
   'LearningrateSchedulerSelector:1r_scheduler':'cosine_annealing',
   ...
   ```

```
'OptimizerSelector: sgd: learning_rate": 0.06829146967649465,
'OptimizerSelector: sgd:momentum': 0.9343847098348538,
'OptimizerSelector: sgd:weight_decay': 0.00024250667352118845,
'PreprocessorSelector:truncated_svd: target_dim": 100),

'budget': 40.0,

'loss': -96.45,
'info': ('loss': 0.12337125303244502,
'model parameters: 176110.0,
'train_accuracy': 96.28550185873605,
'lr_scheduler_converged': 0.0,
'1r': 0.06829146967649465,
'val_accuracy': 96.45)}
```

The output basically shows the hyperparameter setting that Auto-PyTorch finds optimal for the given task—for example, the learning rate is `0.068`, momentum is `0.934`, and so on. The preceding screenshot also shows the training and validation set accuracy for the chosen optimal model configuration.

3. Having converged to an optimal trained model, we can now make predictions on our test set using that model, as follows:

```
y_pred = autoPyTorch.predict(X_test)
print("Accuracy score", np.mean(y_pred.reshape(-1) == y_test))
```

It should output something like this:

```
Accuracy score 0.964
```

As we can see, we have obtained a model with a decent test set performance of 96.4%. For context, a random choice on this task would lead to a performance rate of 10%. We have obtained this good performance without defining either the model architecture or the hyperparameters. Upon setting a higher budget, a more extensive search could lead to an even better performance.

Also, the performance will vary based on the hardware (machine) on which the search is being performed. Hardware with more compute power and memory can run more searches in the same time budget and hence can lead to better performance.

Visualizing the optimal AutoML model

In this section, we will look at the best-performing model that we have obtained by running the model search routine in the previous section. We'll proceed as follows:

1. Having already looked at the hyperparameters in the preceding section, let's look at the optimal model architecture that Auto-PyTorch has devised for us, as follows:

```
pytorch_model = autoPyTorch.get_pytorch_model()
print(pytorch_model)
```

It should output something like this:

```
Sequential(
(0) Linear(in_features-100, out features-100, bias-True)
(1): Sequential(
  (0): ResBlock(
    (layers): Sequential(
      (0): BatchNormid (100, eps-le-05, momentum-0.1, affine-True, track_
running_stats-True)
      (1): ReLU()
      (2) Linear(in_features-100, out features-100, bias-True)
      (3): BatchNormid (100, eps-le-05, momentum-0.1, affine-True, track
running_stats=True)
      (4) ReLU()
      (5): Linear(in features-100, out features-100, bias-True)
  (1) ResBlock(
...
  (3): ResBlock(
    (layers): Sequential(
      (0): BatchNormid (100, eps-le-05, momentum-0.1, affine-True, track_
running_stats-True)
      (1): ReLU()
      (2): Linear(in_features-100, out features-100, bias-True)
      (3): BatchNormid (100, eps-le-05, momentum-0.1, affine-True, track
running_stats-True)
      (4): ReLU()
      (5) Linear(in features-100, out_features-100, bias-True)
...
(3) BatchNormid (100, eps-le-05, momentum-0.1, affine-True, track_
running_stats=True)
(4): ReLU()
(5) Linear(in features-100, out features-10, bias-True)
```

The model consists of some structured residual blocks containing fully connected layers, batch normalization layers, and ReLU activations. At the end, we see a final fully connected layer with 10 outputs—one for each digit from 0 to 9.

2. We can also visualize the actual model graph using `torchviz`, as shown in the next code snippet:

```
x = torch.randn(1, pytorch_model[0].in_features)
y = pytorch_model(x)
arch = make_dot(y.mean(), params=dict(pytorch_model.named_parameters()))
```

This should save a `convnet_arch.pdf` file in the current working directory, which should look like this upon opening:

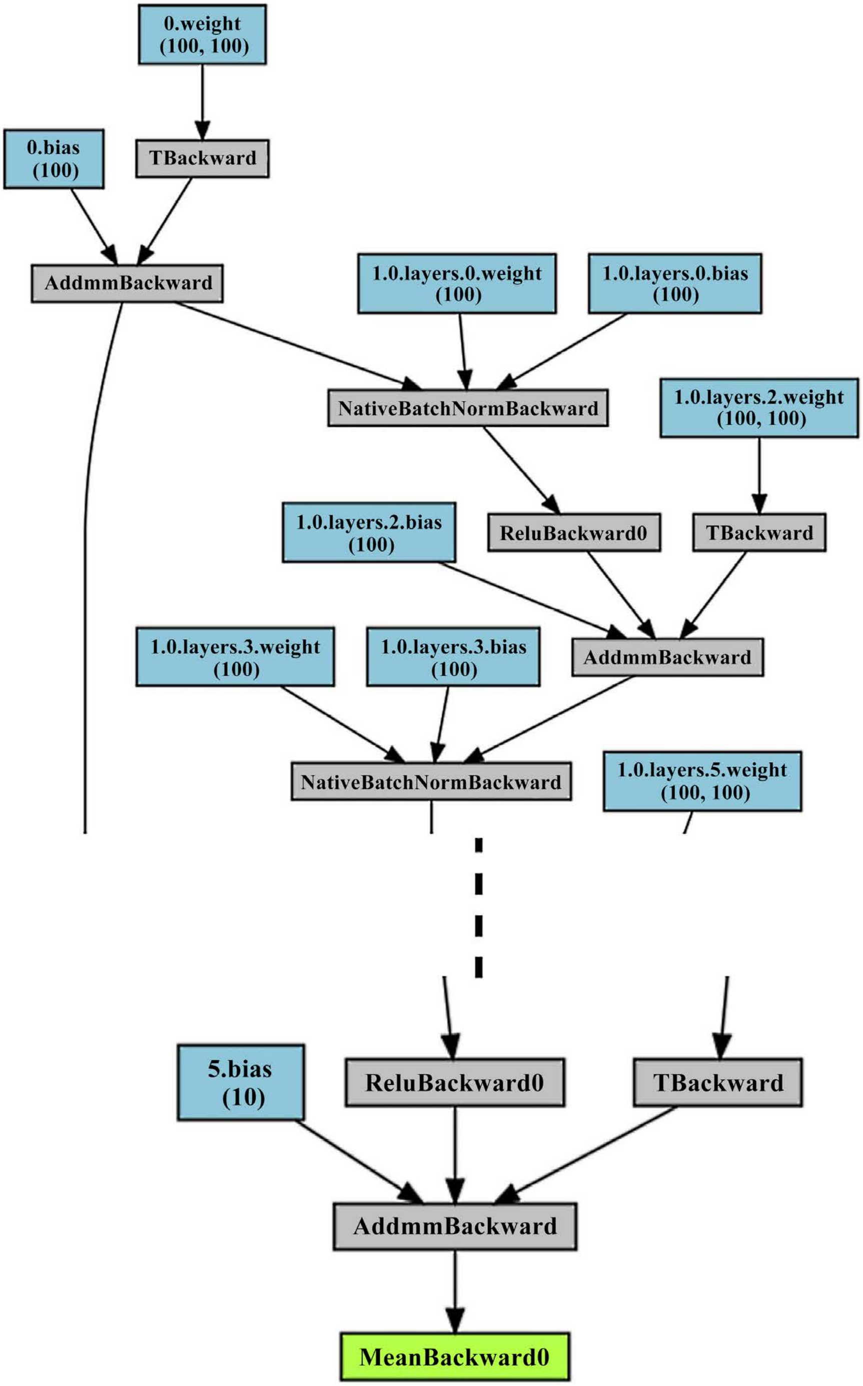


Figure 16.1: Auto-PyTorch model diagram

3. To peek into how the model converged to this solution, we can look at the search space that was used during the model-searching process with the following code:

   ```
   autoPyTorch.get_hyperparameter_search_space()
   ```

   This should output the following:

   ```
   Configuration space object:
     Hyperparameters:
       CreateDataLoader:batch_size, Type: Constant, Value: 125
       Imputation:strategy, Type: Categorical, Choices: {median}, Default:
   median
       InitializationSelector:initialization_method, Type: Categorical,
   Choices: {default}, Default: default
       InitializationSelector:initializer:initialize_bias, Type: Constant,
   Value: No
       ...
       ResamplingStrategySelector:target_size_strategy, Type: Categorical,
   Choices: {none}, Default: none
       ResamplingStrategySelector:under_sampling_method, Type: Categorical,
   Choices: {none}, Default: none
       TrainNode:batch_loss_computation_technique, Type: Categorical,
   Choices: {standard}, Default: standard
     Conditions:
       LearningrateSchedulerSelector:cosine_annealing:T_max |
   LearningrateSchedulerSelector:lr_scheduler == 'cosine_annealing'
       LearningrateSchedulerSelector:cosine_annealing:eta_min |
   LearningrateSchedulerSelector:lr_scheduler == 'cosine_annealing'
       NetworkSelector:shapedresnet:activation | NetworkSelector:network ==
   'shapedresnet'
       ...
       OptimizerSelector:sgd:learning_rate | OptimizerSelector:optimizer ==
   'sgd'
       OptimizerSelector:sgd:momentum | OptimizerSelector:optimizer == 'sgd'
       OptimizerSelector:sgd:weight_decay | OptimizerSelector:optimizer ==
   'sgd'
       PreprocessorSelector:truncated_svd:target_dim |
   PreprocessorSelector:preprocessor == 'truncated_svd'
   ```

It essentially lists the various ingredients required to build the model, with an allocated range per ingredient. For instance, the learning rate is allocated a range of **0.0001** to **0.1** and this space is sampled in a log scale—this is not linear but logarithmic sampling.

While running trials, we have already seen the exact hyperparameter values that Auto-PyTorch samples from these ranges as the optimal values for the given task. We can also alter these hyperparameter ranges manually, or even add more hyperparameters, using the `HyperparameterSearchSpaceUpdates` sub-module under the Auto-PyTorch module [5].

This concludes our exploration of Auto-PyTorch. We successfully built an MNIST digit classification model using Auto-PyTorch, without specifying either the model architecture or the hyperparameters. This MNIST exercise will help you get started with using this and other AutoML tools to build PyTorch models in an automated fashion. Some other similar tools are listed here: Hyperopt [6], Tune [7], skorch [8], BoTorch [9], and Optuna [10].

While we cannot cover all these tools in this chapter, in the next section, we will discuss Optuna, which is a tool focused exclusively on finding an optimal set of hyperparameters, and a tool that works well with PyTorch.

Using Optuna for hyperparameter search

Optuna is one of the hyperparameter search tools that supports PyTorch. You can read in detail about the search strategies used by the tool, such as **Tree-Structured Parzen Estimation** (**TPE**) and **Covariance Matrix Adaptation Evolution Strategy** (**CMA-ES**), in the *Optuna* paper [11]. Besides the advanced hyperparameter search methodologies, the tool also provides a sleek API, which we will explore in a moment.

In this section, we will once again build and train the MNIST model, but this time, using Optuna to figure out the optimal hyperparameter setting. We will discuss important parts of the code step by step in the form of an exercise. The full code can be found on our GitHub [12].

Defining the model architecture and loading the dataset

First, we will define an Optuna-compliant model object. By Optuna-compliant, we mean adding APIs within the model definition code that are provided by Optuna to enable the parameterization of the model hyperparameters. To do this, we'll proceed as follows:

1. First, we import the necessary libraries, as follows:

   ```
   import torch
   import optuna
   ```

   The `optuna` library will manage the hyperparameter search for us throughout the exercise.

2. Next, we define the model architecture. Because we want to be flexible with some of the hyperparameters—such as the number of layers and the number of units in each layer—we need to include some Optuna logic in the model definition code. So, first, we have declared that we need anywhere between 1 to 4 convolutional layers and 1 to 2 fully connected layers thereafter, as illustrated in the following code snippet:

   ```
   class ConvNet(nn.Module):
       def __init__(self, trial):
   ```

```
        super(ConvNet, self).__init__()
        num_conv_layers = trial.suggest_int(
            "num_conv_layers", 1, 4)
        num_fc_layers = trial.suggest_int(
            "num_fc_layers", 1, 2)
```

3. We then successively append the convolutional layers, one by one. Each convolutional layer is instantly followed by a ReLU activation layer, and for each convolutional layer, we declare the depth of that layer to be between `16` and `64`.

   The stride and padding are fixed to `3` and `True`, respectively, and the whole convolutional block is then followed by a `MaxPool` layer, then a `Dropout` layer, with dropout probability ranging anywhere between `0.1` to `0.4` (another hyperparameter), as illustrated in the following code snippet:

```
        self.layers = []
        input_depth = 1 # grayscale image
        for i in range(num_conv_layers):
            output_depth = trial.suggest_int(
                f"conv_depth_{i}", 16, 64)
            self.layers.append(nn.Conv2d(
                input_depth, output_depth, 3, 1))
            self.layers.append(nn.ReLU())
            input_depth = output_depth
        self.layers.append(nn.MaxPool2d(2))
        p = trial.suggest_float(f"conv_dropout_{i}", 0.1, 0.4)
        self.layers.append(nn.Dropout(p))
        self.layers.append(nn.Flatten())
```

4. Next, we add a flattening layer so that fully connected layers can follow. We have to define a `_get_flatten_shape` function to derive the shape of the flattening layer output. We then successively add fully connected layers, where the number of units is declared to be between `16` and `64`. A `Dropout` layer follows each fully connected layer, again with a probability range of `0.1` to `0.4`.

   Finally, we append a fixed fully connected layer that outputs `10` numbers (one for each class/digit), followed by a `LogSoftmax` layer. Having defined all the layers, we then instantiate our model object, as follows:

```
        input_feat = self._get_flatten_shape()
        for i in range(num_fc_layers):
            output_feat = trial.suggest_int(
                f"fc_output_feat_{i}", 16, 64)
            self.layers.append(nn.Linear(
                input_feat, output_feat))
            self.layers.append(nn.ReLU())
            p = trial.suggest_float(f"fc_dropout_{i}", 0.1, 0.4)
            self.layers.append(nn.Dropout(p))
            input_feat = output_feat
        self.layers.append(nn.Linear(input_feat, 10))
        self.layers.append(nn.LogSoftmax(dim=1))
        self.model = nn.Sequential(*self.layers)
    def _get_flatten_shape(self):
        conv_model = nn.Sequential(*self.layers)
        op_feat = conv_model(torch.rand(1, 1, 28, 28))
        n_size = op_feat.data.view(1, -1).size(1)
        return n_size
```

This model initialization function is conditioned on the `trial` object, which is facilitated by Optuna and will decide on the hyperparameter setting for our model. Finally, the `forward` method is quite straightforward, as can be seen in the following code snippet:

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

Thus, we have defined our model object and we can now move on to loading the dataset.

5. The code for dataset loading is the same as in *Chapter 1*, *Overview of Deep Learning Using PyTorch*, and is shown again in the following snippet:

```
train_dataloader = torch.utils.data.DataLoader(...)
test_dataloader = ...
```

In this section, we have successfully defined our parameterized model object as well as loaded the dataset. We will now define the model training and testing routines, along with the optimization schedule.

Defining the model training routine and optimization schedule

Model training itself involves hyperparameters such as optimizer, learning rate, and so on. In this part of the exercise, we will define the model training procedure while utilizing Optuna's parameterization capabilities. We'll proceed as follows:

1. First, we define the training routine. Once again, the code is the same as the training routine code we had for this model in the exercise found in *Chapter 1*, *Overview of Deep Learning Using PyTorch*, and is shown again here:

```
def train(model, device, train_dataloader, optim, epoch):
    for b_i, (X, y) in enumerate(train_dataloader):
        ...
```

2. The model testing routine needs to be slightly augmented. To operate as per Optuna API requirements, the test routine needs to return a model performance metric—accuracy, in this case—so that Optuna can compare different hyperparameter settings based on this metric, as illustrated in the following code snippet:

   ```
   def test(model, device, test_dataloader):
       with torch.no_grad():
           for X, y in test_dataloader:
               ...
       accuracy = 100. * success/ len(test_dataloader.dataset)
       return accuracy
   ```

3. Previously, we would instantiate the model and the optimization function with the learning rate, and start the training loop outside of any function. But to follow the Optuna API requirements, we do all that under an `objective` function, which takes in the same `trial` object that was fed as an argument to the `__init__` method of our model object.

   The `trial` object is needed here too because there are hyperparameters associated with deciding on the learning rate value and choosing an optimizer, as illustrated in the following code snippet:

   ```
   def objective(trial):
       model = ConvNet(trial)
       opt_name = trial.suggest_categorical(
           "optimizer", ["Adam", "Adadelta", "RMSprop", "SGD"])
       lr = trial.suggest_float("lr", 1e-1, 5e-1, log=True)
       optimizer = getattr(optim,opt_name)(model.parameters(), lr=lr)
       for epoch in range(1, 3):
           train(model, device, train_dataloader, optimizer, epoch)
           accuracy = test(model, device,test_dataloader)
           trial.report(accuracy, epoch)
           if trial.should_prune():
               raise optuna.exceptions.TrialPruned()
       return accuracy
   ```

For each epoch, we record the accuracy returned by the model-testing routine. Additionally, at each epoch, we check whether we will prune—that is, whether we will skip—the current epoch. This is another feature offered by Optuna to speed up the hyperparameter search process so that we don't waste time on poor hyperparameter settings.

Running Optuna's hyperparameter search

In this final part of the exercise, we will instantiate what is called an **Optuna study**, and, using the model definition and the training routine, we will execute Optuna's hyperparameter search process for the given model and the given dataset. We'll proceed as follows:

1. Having prepared all the necessary components in the preceding sections, we are ready to start the hyperparameter search process—something that is called a `study` in Optuna terminology. A `trial` is one hyperparameter-search iteration in a `study`. The code can be seen in the following snippet:

```
study = optuna.create_study(study_name="mastering_pytorch",
                            direction="maximize")
study.optimize(objective, n_trials=10, timeout=2000)
```

The `direction` argument helps Optuna compare different hyperparameter settings. Because our metric is accuracy, we will need to maximize the metric. We allow a maximum of `2000` seconds for the `study` or a maximum of `10` different searches—whichever finishes first. The preceding command should output the following:

```
A new study created in memory with name: mastering pytorch

epochs 1 [0/60000 (0V)] training loss: 2.314928

epoch: 1 [16000/60000 (27%)) training loss: 2.339143

epoch: 1 [32000/60000 (53%) training loss: 2.354311

epoch: 1 [48000/60000 (80%)) training loss: 2.392770
Test dataset: Overall Loss: 2.4598, Overall Accuracy: 974/10000 (10%)
epochs 2 [0/60000 (0%)) training loss: 2.352018
epoch: 2 [16000/60000 (27%)) training loss: 2.425988
epoch: 2 [32000/60000 (53%)) training loss: 2.432955
epochs 2 [48000/60000 (80%)) training loss: 2.497166
Trial 0 finished with value: 9.82 and parameters: ('num conv_layers': 4,
'num fc layers': 2, 'conv_depth 0': 20, 'conv_depth 1': 18, "conv_depth
2': 38, 'conv_depth 3°: 27, 'conv _dropout_3': 0.18560304003563008, 'fc_
output_feat 0': 54, 'fc_dropout 19233257074201586, 'fc_output_feat_1':
33, 'fc_dropout_1': 0.10418259777735323, 'optimizer': 'RMSprop', 'lr':
431360836333). Best is trial 0 with value: 9.83.
Trial 1 finished with value: 95.68 and parameters: ('num conv_
layers': 1, 'num fc_layers': 2, 'conv_depth 0': 39, 'conv_dropout_0':
0.3950204757059781, 'fc_output feat 0': 17, 'fc_dropout_0':
0.3760852329345368, 'fc_output_feat_1': 40, "fc_dropout_1':
0.29727560678671294, 'optimizer': 'Adadelta', 'lr': 0.25498429405323125).
Best is trial 1 with value: 95.68.
Trial 2 finished with value: 98.77 and parameters: ('num conv_layers': 3,
'num fc_layers': 2, 'conv_depth 0': 27, 'conv_depth 1': 28, 'conv_depth
2': 42,, 'conv_dropout_0': 0.327456511733, 'fc_output feat 0': 57, 'fc_
dropout_0': 0.1234849615378501, 'fc_output_feat_1': 54, "fc_dropout_1':
0.36784682560478876, 'optimizer': 'Adadelta', 'lr': 0. 4290610978292583).
Best is trial 2 with value: 98.77.
Trial 3 finished with value: 98.28 and parameters: ('num conv_layers':
2, 'num _fc_layers':11, 'conv _depth 0': 38, 'conv_depth_1': 40, 'conv_
dropout_1': 0.359274403082444), 'fc_output_feat 0': 20, 'fc_dropout _0':
0.22476024022504099, 'optimizer': 'Adadelta', 'lr': 0.3167220174336792).
Best is trial 2 with value: 98.77.
...
Trial 7 pruned.
Trial 8 pruned.
Trial 9 pruned.
```

As we can see, the third trial is the most optimal, producing a test set accuracy of 98.77%, and the last three trials are pruned. In the logs, we also see the hyperparameters for each non-pruned trial. For the most optimal trial, for example, there are three convolutional layers with 27, 28, and 46 feature maps, respectively, and then there are two fully connected layers with 57 and 54 units/neurons, respectively, and so on.

2. Each trial is given a completed or a pruned status. We can demarcate those with the following code:

```
pruned_trials = [t for t in study.trials if t.state == optuna.trial.
TrialState.PRUNED]
complete_trials = [t for t in study.trials if t.state == optuna.trial.
TrialState.COMPLETE]
```

3. Finally, we can specifically look at all the hyperparameters of the most successful trial with the following code:

```
print("results: ")
trial = study.best_trial
for key, value in trial.params.items():
    print("{}: {}".format(key, value))
```

You will see the following output:

```
results:
num_trials_conducted: 10
num_trials_pruned: 3
num_trials_completed: 7
results from best trial:
accuracy: 98.77
```

```
hyperparameters:
num_conv_layers: 3
num_fc_layers: 2
conv_depth_0: 27
conv_depth_1: 28
conv_depth_2: 46
conv_dropout_2: 0.3274565117338556
fc_output_feat 0: 57
fc_dropout_0: 0.12348496153785013
fc_output_feat_1: 54
fc_dropout_1: 0.36784682560478876
optimizer: Adadelta
lr: 0.4290610978292583
```

As we can see, the output shows us the total number of trials and the number of successful trials performed. It further shows us the model hyperparameters for the most successful trial, such as the number of layers, the number of neurons in layers, the learning rate, the optimization schedule, and so on.

This brings us to the end of the exercise. We have managed to use Optuna to define a range of hyperparameter values for different kinds of hyperparameters for a handwritten digit classification model. Using Optuna's hyperparameter search algorithm, we ran 10 different trials and managed to obtain the highest accuracy of 98.77% in one of those trials. The model (architecture and hyperparameters) from the most successful trial can be used for training with larger datasets, thereby serving in a production system.

Using the lessons from this section, you can use Optuna to find the optimal hyperparameters for any neural network model written in PyTorch. Optuna can also be used in a distributed fashion if the model is extremely large and/or there are way too many hyperparameters to tune. Optuna has well-written documentation on distributed search, which can be found here [13].

Lastly, Optuna supports not only PyTorch but other popular machine learning libraries too, such as TensorFlow, scikit-learn, MXNet, and so on.

Summary

In this chapter, we discussed AutoML, which aims to provide methods for model selection and hyperparameter optimization. AutoML is useful for beginners who have little expertise in making decisions such as how many layers to put in a model, which optimizer to use, and so on. AutoML is also useful for experts to both speed up the model training process and discover superior model architectures for a given task that would be nearly impossible to figure out manually.

In the next chapter, we will study another increasingly important and crucial aspect of machine learning, especially deep learning. We will closely look at how to interpret output produced by PyTorch models—a field popularly known as model interpretability or explainability.

Reference list

1. Auto-PyTorch GitHub repository: `https://github.com/automl/Auto-PyTorch`
2. *Auto-PyTorch Tabular: Multi-Fidelity MetaLearning for Efficient and Robust AutoDL*: `https://arxiv.org/abs/2006.13799`
3. Using Auto-PyTorch for optimal MNIST model search (full code): `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter16/automl-pytorch.ipynb`
4. Image data for a supplementary exercise: `https://github.com/automl/Auto-PyTorch/blob/master/examples/20_basics/example_image_classification.py`
5. Auto-PyTorch GitHub documentation: `https://github.com/automl/Auto-PyTorch#configuration`
6. Hyperopt: `https://github.com/hyperopt/hyperopt`
7. Tune: `https://docs.ray.io/en/latest/tune/index.html`
8. skorch: `https://github.com/skorch-dev/skorch`
9. BoTorch: `https://botorch.org/`
10. Optuna: `https://optuna.org/`
11. *Optuna: A Next-generation Hyperparameter Optimization Framework*: `https://arxiv.org/pdf/1907.10902.pdf`
12. Optuna exercise full code: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter16/optuna_pytorch.ipynb`
13. Distributed tuning with Optuna: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter16/optuna_pytorch.ipynb`

Learn more on Discord

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

`https://packt.link/mastorch`

17

PyTorch and Explainable AI

Throughout this book, we have built several deep learning models that can perform different kinds of tasks for us, such as a handwritten digit classifier, an image-caption generator, and a sentiment classifier. Although we have mastered how to train and evaluate these models using PyTorch, we do not know precisely what is happening inside these models while they make predictions. Model interpretability or explainability is a field of machine learning where we aim to answer the question, "Why did the model make that prediction?" Put differently, "What did the model see in the input data to make that particular prediction?" The answers to such questions become essential when such models are used for sensitive applications such as cancer diagnosis and legal aid.

In this chapter, we will use the handwritten digit classification model from *Chapter 1, Overview of Deep Learning Using PyTorch*, examine its inner workings, and thereby explain why the model makes a certain prediction for a given input. We will first dissect the model using only PyTorch code. Then, we will use a specialized model interpretability toolkit, called **Captum** [1], to further investigate what is happening inside the model. Captum is a dedicated third-party library for PyTorch that provides model interpretability tools for deep learning models, including image- and text-based models.

This chapter should provide you with the skills that are required to uncover the internals of a deep learning model. Looking inside a model this way can help you to reason about the model's predictive behavior. At the end of this chapter, you will be able to use this hands-on experience to start interpreting your own deep learning models using PyTorch (and Captum).

This chapter is broken down into the following topics:

- Model interpretability in PyTorch
- Using Captum to interpret models

Model interpretability in PyTorch

In this section, we will dissect a trained handwritten digits classification model using PyTorch in the form of an exercise. More precisely, we will be looking at the details of the convolutional layers of the trained handwritten digits classification model to understand what visual features the model is learning from the handwritten digit images.

We will look at the convolutional filters/kernels along with the feature maps produced by those filters.

These details will help us to understand how the model is processing input images and, therefore, making predictions. The full code for the exercise can be found in our GitHub repository [2].

Training the handwritten digits classifier – a recap

We will quickly revisit the steps from *Chapter 1*, *Overview of Deep Learning Using PyTorch*, involved in training the handwritten digits classification model. After completing these steps, we will have a trained CNN model with decent classification accuracy. The steps are as follows:

1. First, we import the relevant libraries, and then set the random seeds to be able to reproduce the results of this exercise:

    ```
    import torch
    np.random.seed(123)
    torch.manual_seed(123)
    ```

2. Next, we will define the model's architecture:

    ```
    class ConvNet(nn.Module):
        def __init__(self):
        def forward(self, x):
    ```

3. Next, we will define the model's training and testing routine:

    ```
    def train(model, device, train_dataloader, optim,  epoch):
    def test(model, device, test_dataloader):
    ```

4. We then define the training and testing dataset loaders:

    ```
    train_dataloader = torch.utils.data.DataLoader(...)
    test_dataloader = torch.utils.data.DataLoader(...)
    ```

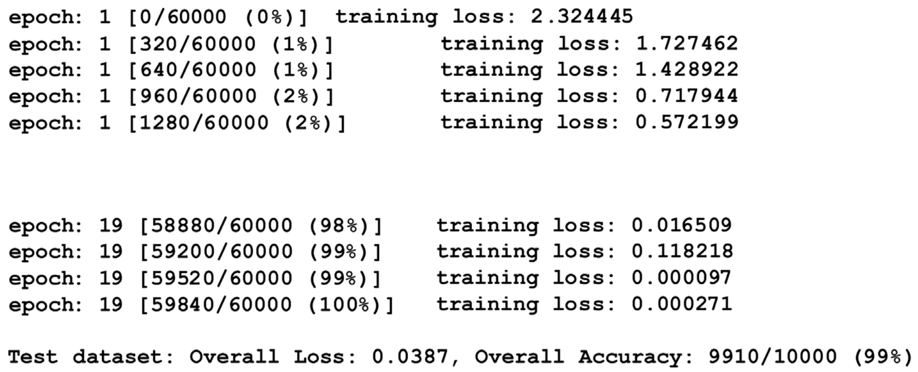
5. Next, we instantiate our model and define the optimization schedule:

    ```
    device = torch.device("cpu")
    model = ConvNet()
    optimizer = optim.Adadelta(model.parameters(), lr=0.5)
    ```

6. Finally, we start the model training loop, where we train our model for 20 epochs:

    ```
    for epoch in range(1, 20):
        train(model, device, train_dataloader, optimizer, epoch)
        test(model, device, test_dataloader)
    ```

    This should output the following:



```
epoch: 1 [0/60000 (0%)]  training loss: 2.324445
epoch: 1 [320/60000 (1%)]        training loss: 1.727462
epoch: 1 [640/60000 (1%)]        training loss: 1.428922
epoch: 1 [960/60000 (2%)]        training loss: 0.717944
epoch: 1 [1280/60000 (2%)]       training loss: 0.572199


epoch: 19 [58880/60000 (98%)]    training loss: 0.016509
epoch: 19 [59200/60000 (99%)]    training loss: 0.118218
epoch: 19 [59520/60000 (99%)]    training loss: 0.000097
epoch: 19 [59840/60000 (100%)]   training loss: 0.000271

Test dataset: Overall Loss: 0.0387, Overall Accuracy: 9910/10000 (99%)
```

Figure 17.1: Model training logs

7. Finally, we can test the trained model on a sample test image. The sample test image is loaded as follows:

```
test_samples = enumerate(test_dataloader)
b_i, (sample_data, sample_targets) = next(test_samples)
plt.imshow(sample_data[0][0], cmap='gray', interpolation='none')
plt.show()
```

This should output the following:

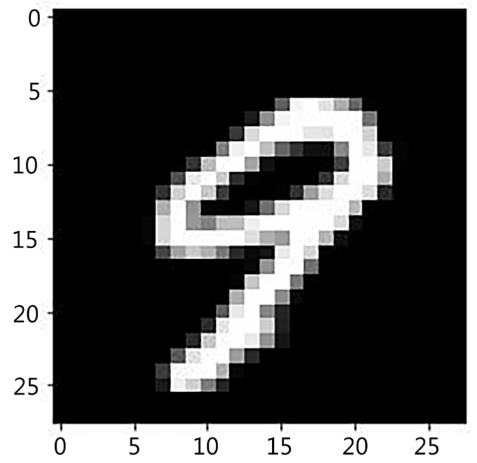
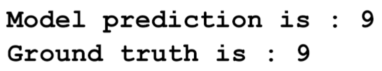


Figure 17.2: An example of a handwritten image

8. Then, we use this sample test image to make a model prediction, as follows:

```
print(f"Model prediction is : {model(sample_data).data.max(1)[1][0]}")
print(f"Ground truth is : {sample_targets[0]}")
```

This should output the following:



```
Model prediction is : 9
Ground truth is : 9
```

Figure 17.3: Model prediction

Therefore, we have trained a handwritten digits classification model and used it to make inferences on a sample image. We will now look at the internals of the trained model. We will also investigate what convolutional filters have been learned by this model.

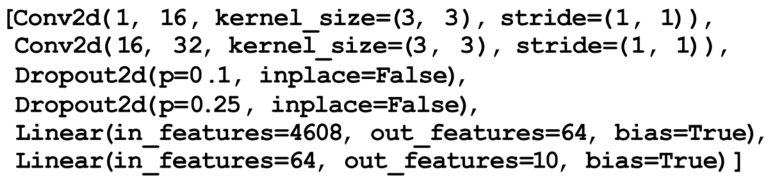
Visualizing the convolutional filters of the model

In this section, we will go through the convolutional layers of the trained model and look at the filters that the model has learned during training. This will tell us how the convolutional layers are operating on the input image, what kinds of features are being extracted, and more:

1. First, we need to obtain a list of all the layers in the model, as follows:

```
model_children_list = list(model.children())
convolutional_layers = []
model_parameters = []
model_children_list
```

This should output the following:



```
[Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1)),
 Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1)),
 Dropout2d(p=0.1, inplace=False),
 Dropout2d(p=0.25, inplace=False),
 Linear(in_features=4608, out_features=64, bias=True),
 Linear(in_features=64, out_features=10, bias=True)]
```

Figure 17.4: Model layers

As you can see, there are two convolutional layers that both have 3x3 filters. The first convolutional layer uses **16** such filters, whereas the second convolutional layer uses **32**. We are focusing on visualizing convolutional layers in this exercise because they are visually more intuitive. However, you can similarly explore the other layers, such as linear layers, by visualizing their learned weights.

2. Next, we select only the convolutional layers from the model and store them in a separate list:

```
for i in range(len(model_children_list)):
    if type(model_children_list[i]) == nn.Conv2d:
        model_parameters.append(model_children_list[i].w      eight)
        convolutional_layers.append(model_children_list[i])
```

In this process, we also make sure to store the parameters or weights learned in each convolutional layer.

3. We are now ready to visualize the learned filters of the convolutional layers. We begin with the first layer, which has 16 3x3 filters. The following code visualizes those filters for us:

```
plt.figure(figsize=(5, 4))
for i, flt in enumerate(model_parameters[0]):
    # add 4x4 subplots to plot the 16 kernels
    plt.subplot(4, 4, i+1)
    plt.imshow(flt[0, :, :].detach(), cmap='gray')
    plt.axis('off')
plt.show()
```

This should output the following:

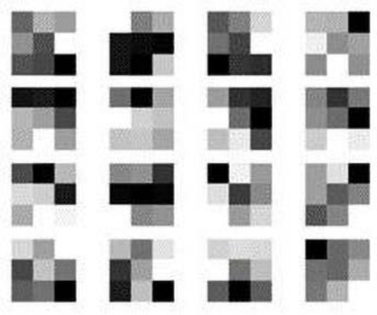


Figure 17.5: The first convolutional layer's filters

Firstly, we can see that all the learned filters are slightly different from each other, which is a good sign. These filters usually have contrasting values inside them so that they can extract some types of gradients when convolved around an image. During model inference, each of these 16 filters operates independently on the input grayscale image and produces 16 different feature maps, which we will visualize in the next section.

4. Similarly, we can visualize the 32 filters learned in the second convolutional layer using the same code, as in the preceding step, but with the following change:

```
plt.figure(figsize=(5, 8))
for i, flt in enumerate(model_parameters[1]):
plt.show()
```

This should output the following:

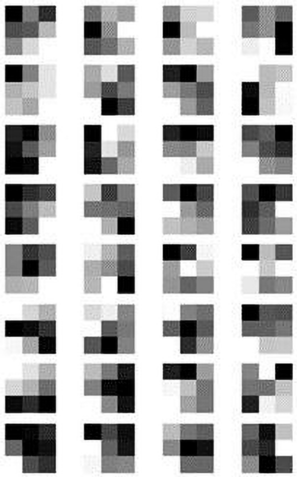


Figure 17.6: The second convolutional layer's filters

Once again, we have 32 different filters/kernels that have contrasting values aimed at extracting gradients from the image. These filters are already applied to the output of the first convolutional layer, and hence produce even higher levels of output feature maps.

The usual goal of CNN models with multiple convolutional layers is to keep producing more and more complex, or higher-level, features that can represent complex visual elements such as a nose on a face, traffic lights on the road, and more.

Next, we will take a look at what comes out of these convolutional layers as these filters operate/convolve on their given inputs.

Visualizing the feature maps of the model

In this section, we will run a sample handwritten image through the convolutional layers and visualize the outputs of these layers. For different layers, we expect the outputs to capture different visual features of the image such as detecting edges, color, curves, and circles. Let's get going:

1. First, we need to gather the results of every convolutional layer output in the form of a list, which is achieved using the following code:

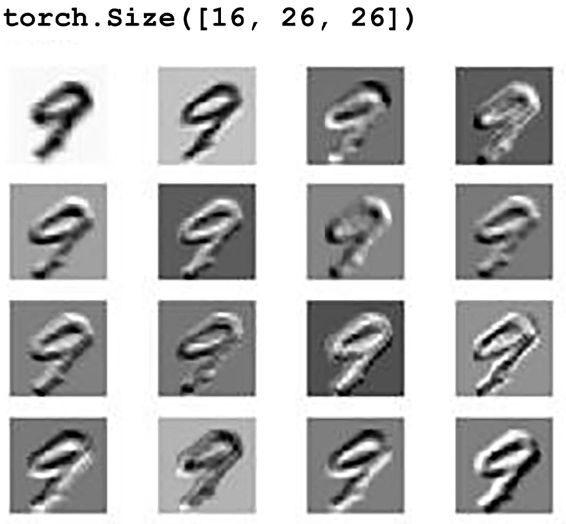
   ```
   per_layer_results = [convolutional_layers[0](sample_data)]
   for i in range(1, len(convolutional_layers)):
       per_layer_results.append(
           convolutional_layers[i](per_layer_results[-1]))
   ```

   Notice that we call the forward pass for each convolutional layer separately while ensuring that the *n*th convolutional layer receives as input the output of the (*n-1*)th convolutional layer.

2. We can now visualize the feature maps produced by the two convolutional layers. We will begin with the first layer by running the following code:

```
plt.figure(figsize=(5, 4))
layer_visualisation = per_layer_results[0][0, :, :, :]
layer_visualisation = layer_visualisation.data
print(layer_visualisation.size())
for i, flt in enumerate(layer_visualisation):
    plt.subplot(4, 4, i + 1)
    plt.imshow(flt, cmap='gray')
    plt.axis("off")
plt.show()
```

This should output the following:



```
torch.Size([16, 26, 26])
```

Figure 17.7: The first convolutional layer's feature maps

The numbers **(16, 26, 26)** represent the output dimensions of the first convolution layer. Essentially, the sample image size is (28, 28), the filter size is (3,3), and there is no padding. Therefore, the resulting feature map size will be (26, 26). Because there are 16 such feature maps produced by the 16 filters (please refer to *Figure 17.5*), the overall output dimension is (16, 26, 26).

As you can see, each filter produces a feature map from the input image. Additionally, each feature map represents a different visual feature in the image. For example, the top-left feature map essentially inverts the pixel values in the image (please refer to *Figure 17.2*), whereas the bottom-right feature map represents some form of edge detection.

These 16 feature maps are then passed on to the second convolutional layer, where yet another 32 filters convolve separately on these 16 feature maps to produce 32 new feature maps. We will look at these next.

3. We can use the same code as the preceding one with minor changes (as highlighted in the following code) to visualize the 32 feature maps produced by the next convolutional layer:

```
plt.figure(figsize=(5, 8))
layer_visualisation = per_layer_results[1][0, :, :, :]
    plt.subplot(8, 4, i + 1)
plt.show()
```

This should output the following:

```
torch.Size([32, 24, 24])
```

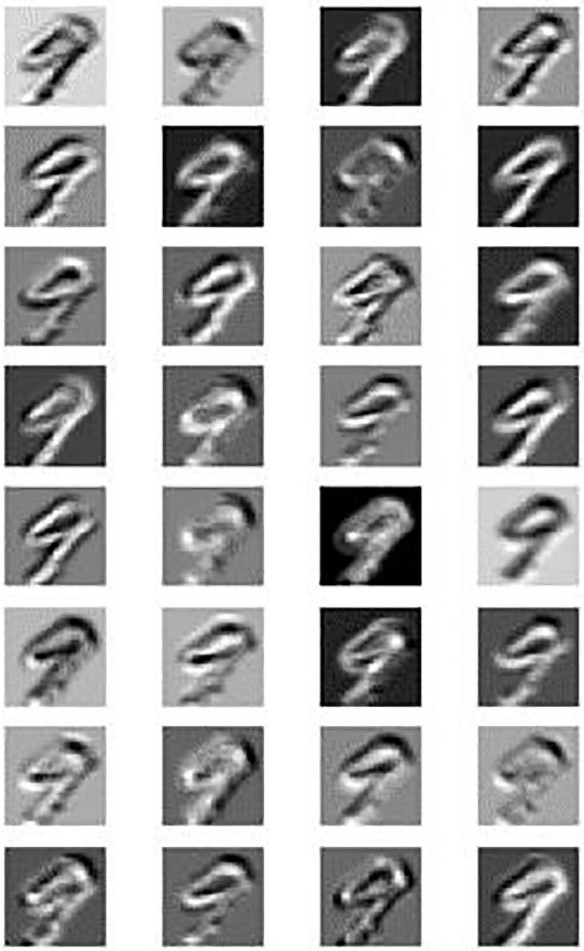


Figure 17.8: The second convolutional layer's feature maps

Compared to the earlier 16 feature maps, these 32 feature maps are visibly more complex. They seem to be doing more than just edge detection, and this is because they are already operating on the outputs of the first convolutional layer instead of the raw input image.

In this model, the two convolutional layers are followed by two linear layers with (4,608x64) and (64x10) number of parameters, respectively. Although the linear layer weights are also useful to visualize, the sheer number of parameters (4,608x64) is, visually, a lot to get your head around. Therefore, in this section, we will restrict our visual analysis to convolutional weights only.

And thankfully, we have more sophisticated ways of interpreting model prediction without having to look at such a large number of parameters. In the next section, we will explore Captum, which is a machine learning model interpretability toolkit that works with PyTorch and helps us explain model decisions with a few lines of code.

Using Captum to interpret models

Captum [3] is an open source model interpretability library built by Meta on top of PyTorch, and it is currently (at the time of writing) under active development. In this section, we will use the handwritten digits classification model that we had trained in the preceding section. We will also use some of the model interpretability tools offered by Captum to explain the predictions made by this model. The full code for the following exercise can be found in our GitHub repository [4].

Setting up Captum

The model training code is similar to the code shown in the *Training the handwritten digits classifier – a recap* section of this chapter. In the following steps, we will use the trained model and a sample image to understand what happens inside the model while making a prediction for the given image:

1. There are a few extra imports related to Captum that we need to perform in order to use Captum's built-in model interpretability functions:

```
from captum.attr import IntegratedGradients
from captum.attr import Saliency
from captum.attr import DeepLift
from captum.attr import visualization as viz
```

2. In order to do a model forward pass with the input image, we reshape the input image to match the model input size, that is (1, 28, 28):

```
captum_input = sample_data[0].unsqueeze(0)
captum_input.requires_grad = True
```

   As per Captum's requirements, the input tensor (image) needs to be involved in gradient computation. Therefore, we set the `requires_grad` flag for `input` to `True`.

3. Next, we prepare the sample image to be processed by the model interpretability methods using the following code:

```
orig_image = np.tile(np.transpose((sample_data[0].cpu().detach().numpy()
/ 2) + 0.5, (1, 2, 0)), (1,1,3))
_ = viz.visualize_image_attr(None, orig_image, cmap='gray',
method="original_image", title="Original Image")
```

This should output the following:

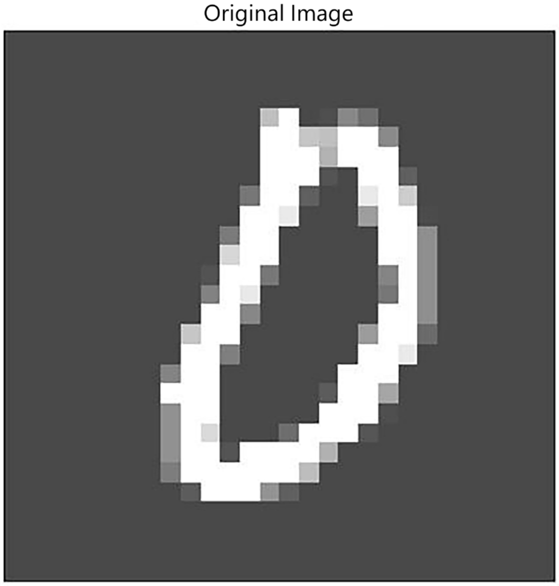


Figure 17.9: The original image

The above code simply outputs the original handwritten digit image. We have tiled the grayscale image across the depth dimension so that it can be consumed by the Captum method, which expects a three-channel image.

Next, we will actually apply some of Captum's interpretability methods to the forward pass of the prepared grayscale image through the pretrained handwritten digits classification model.

Exploring Captum's interpretability tools

In this section, we will be looking at some of the model interpretability methods provided by Captum.

One of the most fundamental methods of interpreting model results is by looking at saliency, which represents the gradients of the output (class `0`, in this example) with respect to the input (that is, the input image pixels). The larger the gradients with respect to a particular input, the more important that input is. These gradients tell us how much a small change in each input feature would affect the output. You can read more about how these gradients are exactly calculated in the original saliency paper [5]. Captum provides an implementation of the saliency method:

1. In the following code, we use Captum's `Saliency` module to compute the gradients:

```
saliency = Saliency(model)
gradients = saliency.attribute(
    captum_input, target=sample_targets[0].item())
gradients = np.reshape(
    gradients.squeeze().cpu().detach().numpy(), (28, 28, 1))
_ = viz.visualize_image_attr(
    gradients, orig_image, method="blended_heat_map",
    sign="absolute_value",
    show_colorbar=True, title="Overlayed Gradients")
```

This should output the following:

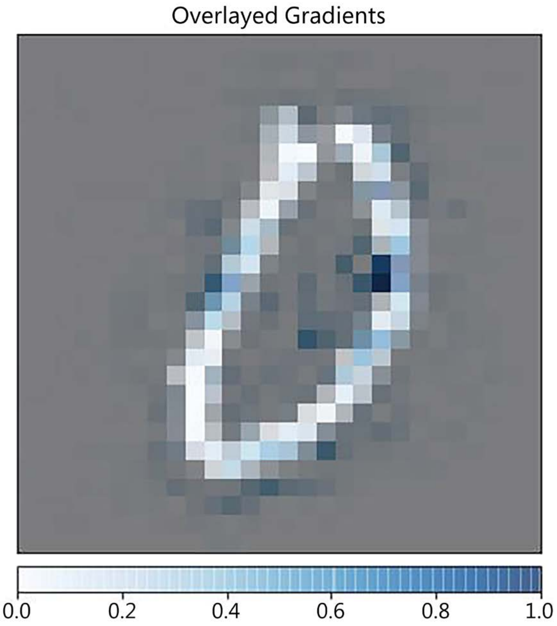


Figure 17.10: Overlayed gradients

In the preceding code, we reshaped the obtained gradients to size `(28,28,1)` in order to overlay them on the original image, as shown in the preceding diagram. Captum's `viz` module takes care of the visualizations for us. We can further visualize only the gradients, without the original image, using the following code:

```
plt.imshow(np.tile(gradients/(np.max(gradients)), (1,1,3)));
```

We will get the following output:

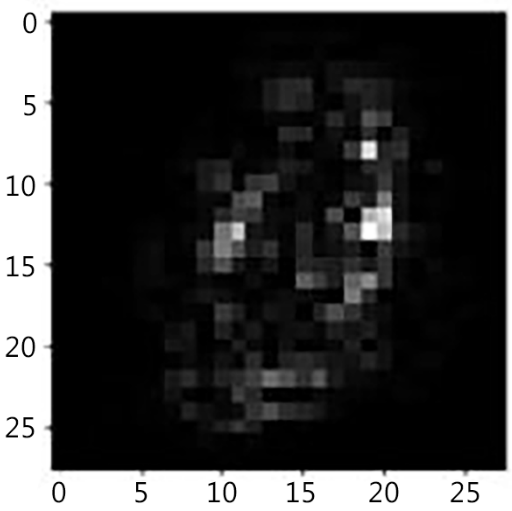


Figure 17.11: Gradients

As you can see, the gradients are spread across those pixel regions in the image that are likely to contain the digit `0`.

2. Next, using similar code, we will look at another interpretability method – integrated gradients. With this method, we will look for **feature attribution**, or **feature importance**. That is, we'll look for what pixels are important to use when making predictions. Under the integrated gradients technique, apart from the input image, we also need to specify a baseline image, which is usually set to an image with all of the pixel values set to zero. The zero baseline acts as a blank canvas to measure the impact of each pixel from a state where it would have no influence on the output, allowing for a clear depiction of how the pixels progressively influence the model's decision.

   An integral of gradients is then calculated with respect to the input image along the path from the baseline image to the input image. Details of the implementation of the integrated gradients technique can be found in the original paper [6]. The following code uses Captum's `IntegratedGradients` module to derive the importance of each input image pixel:

```
integ_grads = IntegratedGradients(model)
attributed_ig, delta = integ_grads.attribute(
    captum_input, target = sample_targets[0],
    baselines = captum_input * 0, return_convergence_delta=True)
attributed_ig = np.reshape(
    attributed_ig.squeeze().cpu().detach().numpy(), (28, 28, 1))
_ = viz.visualize_image_attr(
    attributed_ig, orig_image, method = "blended_heat_map",
    sign="all", show_colorbar=True,
    title="Overlayed Integrated Gradients")
```

This should output the following:

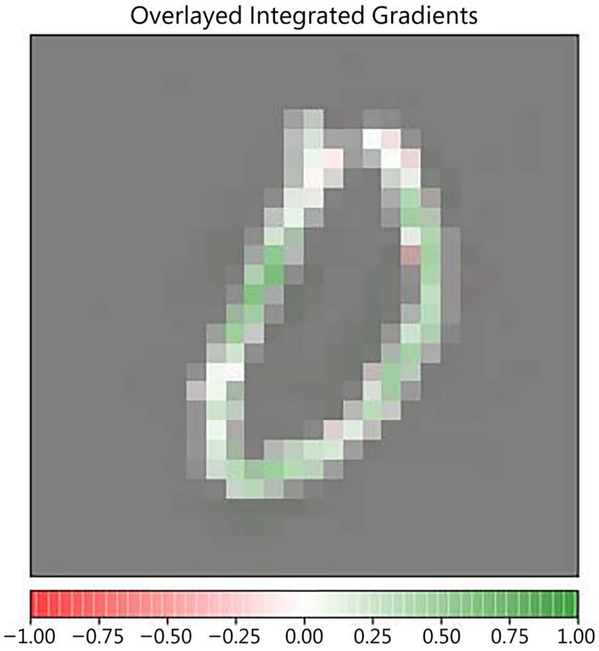


Figure 17.12: Overlayed integrated gradients

As expected, the gradients are high in the pixel regions that contain the digit `0`.

3. Finally, we will look at yet another gradient-based attribution technique, called **deeplift**. Deeplift also requires a baseline image besides the input image. Once again for the baseline, we use an image with all the pixel values set to zero. Deeplift computes the change in non-linear activation outputs with respect to the change in input from the baseline image to the input image (*Figure 17.9*). The following code uses the `DeepLift` module provided by Captum to compute the gradients and displays these gradients overlayed on the original input image:

```
deep_lift = DeepLift(model)
attributed_dl = deep_lift.attribute(
    captum_input, target=sample_targets[0],
    baselines=captum_input * 0, return_convergence_delta=False)
attributed_dl = np.reshape(
    attributed_dl.squeeze(0).cpu().detach().numpy(), (28, 28, 1))
_ = viz.visualize_image_attr(
    attributed_dl, orig_image, method="blended_heat_map",
    sign="all",show_colorbar=True, title="Overlayed DeepLift")
```

You should see the following output:

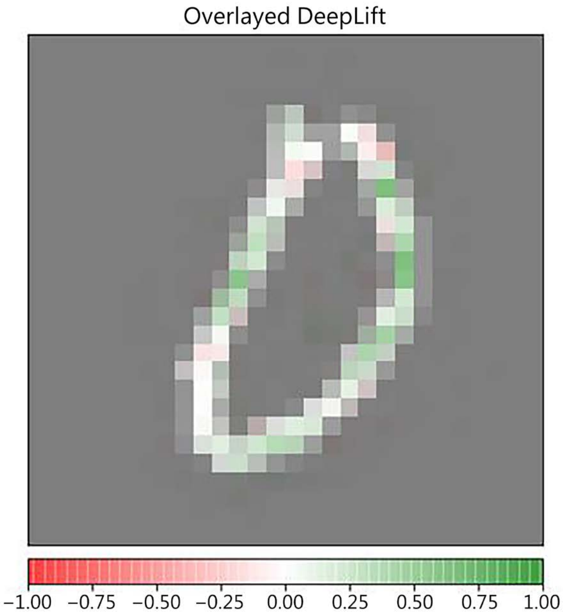


Figure 17.13: Overlayed deeplift

Once again, the gradient values are extreme around the pixels that contain the digit `0`.

This brings us to the end of this exercise and this section. There are more model interpretability techniques provided by Captum, such as *LayerConductance*, *GradCAM*, and *SHAP* [7]. Model interpretability is an active area of research, and hence libraries such as Captum are likely to evolve rapidly. More such libraries are likely to be developed in the near future, which will enable us to make model interpretability a standard component of the machine learning life cycle.

Summary

In this chapter, we have briefly explored how to explain or interpret the decisions made by deep learning models using PyTorch.

In the next chapter of this book, we will learn how to build a recommendation system from scratch, using PyTorch.

Reference List

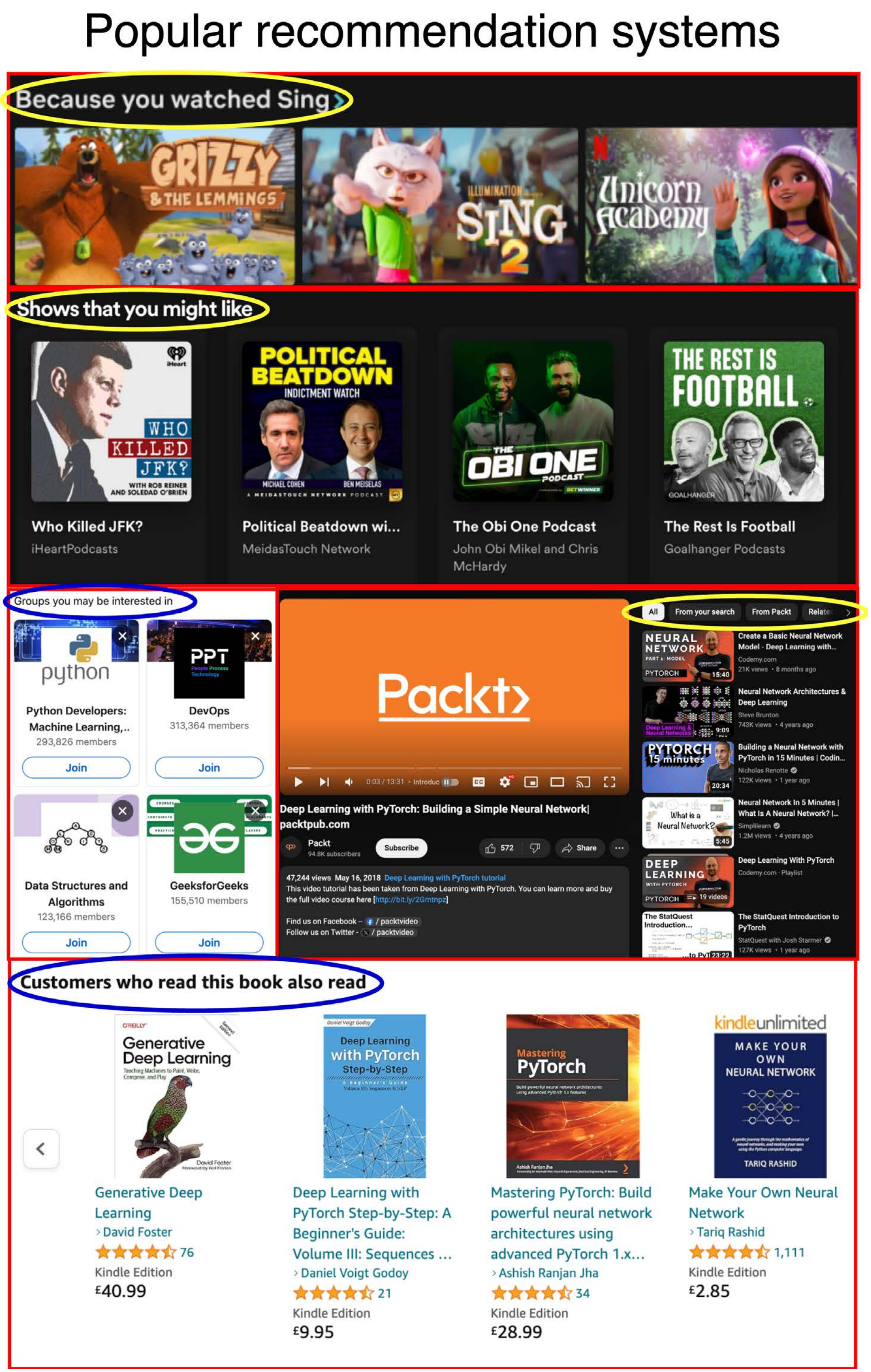
1. Captum GitHub: `https://github.com/pytorch/captum`
2. GitHub 1: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter17/pytorch_interpretability.ipynb`
3. Captum Website: `https://captum.ai/`

4. GitHub 2: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter17/captum_interpretability.ipynb`
5. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps: `https://arxiv.org/pdf/1312.6034`
6. Axiomatic Attribution for Deep Networks: `https://arxiv.org/pdf/1703.01365`
7. A Unified Approach to Interpreting Model Predictions: `https://arxiv.org/pdf/1705.07874`

18

Recommendation Systems with PyTorch

Recommendation systems are everywhere, like Netflix and YouTube, which recommend what to watch; Spotify, which recommends what to listen to; LinkedIn, which suggests jobs; and Amazon, which recommends which products to buy.



Because you watched Sing

Shows that you might like

Who Killed JFK?
iHeartPodcasts

Political Beatdown wi...
MeidasTouch Network

The Obi One Podcast
John Obi Mikel and Chris McHardy

The Rest Is Football
Goalhanger Podcasts

Groups you may be interested in

Python Developers: Machine Learning,..
293,826 members

DevOps
313,364 members

Data Structures and Algorithms
123,166 members

GeeksforGeeks
155,510 members

0:03 / 13:31 · Introduc

Deep Learning with PyTorch: Building a Simple Neural Network| packtpub.com

Packt
94.8K subscribers

47,244 views May 16, 2018 Deep Learning with PyTorch tutorial
This video tutorial has been taken from Deep Learning with PyTorch. You can learn more and buy the full video course here [http://bit.ly/2Gmtnpz]

Find us on Facebook – / packtvideo
Follow us on Twitter - / packtvideo

Create a Basic Neural Network Model - Deep Learning with...
Codemy.com
21K views · 8 months ago

Neural Network Architectures & Deep Learning
Steve Brunton
743K views · 4 years ago

Building a Neural Network with PyTorch in 15 Minutes | Codin...
Nicholas Renotte
122K views · 1 year ago

Neural Network In 5 Minutes | What Is A Neural Network? |...
Simplilearn
1.2M views · 4 years ago

Deep Learning With PyTorch
Codemy.com · Playlist

The StatQuest Introduction to PyTorch
StatQuest with Josh Starmer
127K views · 1 year ago

Customers who read this book also read

Generative Deep Learning
› David Foster
★★★★☆ 76
Kindle Edition
£40.99

Deep Learning with PyTorch Step-by-Step: A Beginner's Guide: Volume III: Sequences ...
› Daniel Voigt Godoy
★★★★☆ 21
Kindle Edition
£9.95

Mastering PyTorch: Build powerful neural network architectures using advanced PyTorch 1.x...
› Ashish Ranjan Jha
★★★★☆ 34
Kindle Edition
£28.99

Make Your Own Neural Network
› Tariq Rashid
★★★★☆ 1,111
Kindle Edition
£2.85

Figure 18.1: Examples of different recommendation systems. Starting from top to bottom, followed by left to right – Netflix, Spotify, LinkedIn, YouTube, and Amazon

A recommendation system is an algorithm that provides personalized suggestions to users. The main goal is to predict what product(s) a user may be interested in, based on their preferences, behaviors, similarity with existing users, and interactions with the system. Most of today's recommendation systems are powered by an underlying deep learning model. Such models predict if a user will like a product (a movie, a book, a podcast, a person on the web, etc.) based on the existing consumption patterns of this and the other users in the system.

In this chapter, we use PyTorch to build such a recommendation system from scratch. We will build a movie recommendation system, like Netflix's, although much smaller in size. We will train a deep learning model on the MovieLens dataset [1] and use the trained model to recommend the most suitable (not yet seen or partially watched) movies for a given user. By the end of this chapter, you will be able to build your recommendation system using PyTorch. Although the example used in the chapter is on movies, the idea can be extrapolated to all kinds of recommendation systems.

This chapter is broken down into the following topics:

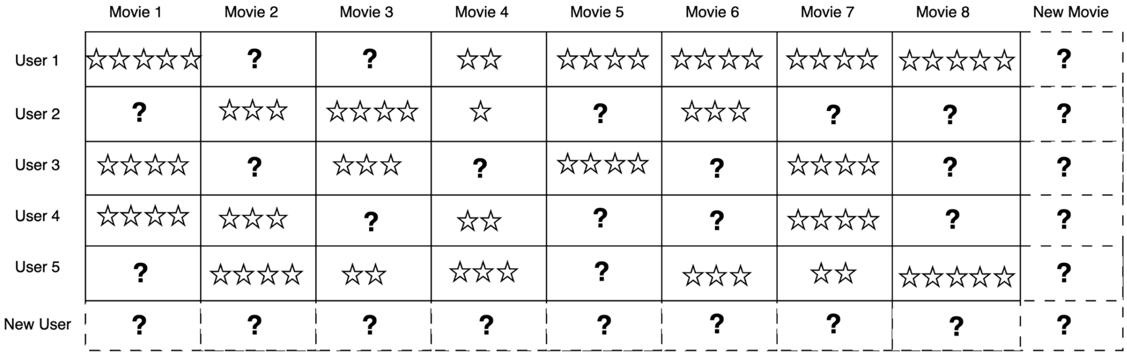
- Using deep learning for **recommendation systems**
- Understanding and loading the **MovieLens** dataset
- Training and evaluating an **embedding-based** recommendation system model
- Building a recommendation system using the trained model

Using deep learning for recommendation systems

In this section, we will learn how to use deep learning to create a recommendation system. We will use the example of a movie recommendation system to understand the underlying concepts. First, we will review what movie recommendation system data looks like, and we will then discuss a deep learning-based solution to build a recommendation system.

Understanding a movie recommendation system dataset

The dataset for building a recommendation system looks like the toy example shown in *Figure 18.2*. On the vertical axis, we have 5 users in our database. And on the horizontal axis, we have 8 movies.



	Movie 1	Movie 2	Movie 3	Movie 4	Movie 5	Movie 6	Movie 7	Movie 8	New Movie
User 1	☆☆☆☆☆	?	?	☆☆	☆☆☆☆	☆☆☆☆	☆☆☆☆	☆☆☆☆☆	?
User 2	?	☆☆☆	☆☆☆☆	☆	?	☆☆☆	?	?	?
User 3	☆☆☆☆	?	☆☆☆	?	☆☆☆☆	?	☆☆☆☆	?	?
User 4	☆☆☆☆	☆☆☆	?	☆☆	?	?	☆☆☆☆	?	?
User 5	?	☆☆☆☆	☆☆	☆☆☆	?	☆☆☆	☆☆	☆☆☆☆☆	?
New User	?	?	?	?	?	?	?	?	?

Figure 18.2: Example of a movie database presented as a user-movie matrix, where the entries of the matrix represent the rating given by the user to the movie

For a given user and a given movie, we find a rating ranging from 1 to 5 stars, 1 being the worst and 5 being the best. This rating is used as a target to train a deep learning model, which can then predict user ratings for movies not yet seen by the given user.

Not all movies are watched by every user, and hence we see a lot of question marks (read as null) in the data. Such sparsity in recommendation systems data is quite normal. Also, we have a new user who has not watched anything yet and a new movie that has not been watched by anyone yet. These again add to the data sparsity. Hence, a list of (user, movie, rating) is typically used as a training dataset for building a recommendation system instead of the entire user-movie matrix.

An intermediate goal of a trained deep learning-based recommendation system is to fill in those question marks (in *Figure 18.2*) with a predicted rating. Once we have those predicted ratings, we can recommend products (movies) to users in decreasing order of the predicted ratings. This begs the question – how do we predict the ratings? While there can be many ways (including non-deep learning methods) to do this, a prominent solution is an embedding-based recommender system.

Understanding embedding-based recommender systems

In this section, we will learn how an embedding-based recommender system works. Later in the chapter, we will build this system from scratch using PyTorch. As the name suggests, such a system works on embeddings. In a recommendation system, we have two main entities – products (movies) and users.

The idea of an embedding-based recommendation system is to convert products (movies) and users into vectors of real numbers in a lower-dimensional space. These vectors, also known as embeddings, represent the features or characteristics of products and users in a way that preserves their relationships and patterns, bringing together similar movies in the embedding space, and ensuring users are *closer* to the movies they like. How do we learn these embeddings? We train an end-to-end deep neural network model (EmbeddingNet) that takes in a user and a movie as input and predicts the user rating for that movie as output, as shown in *Figure 18.3*.

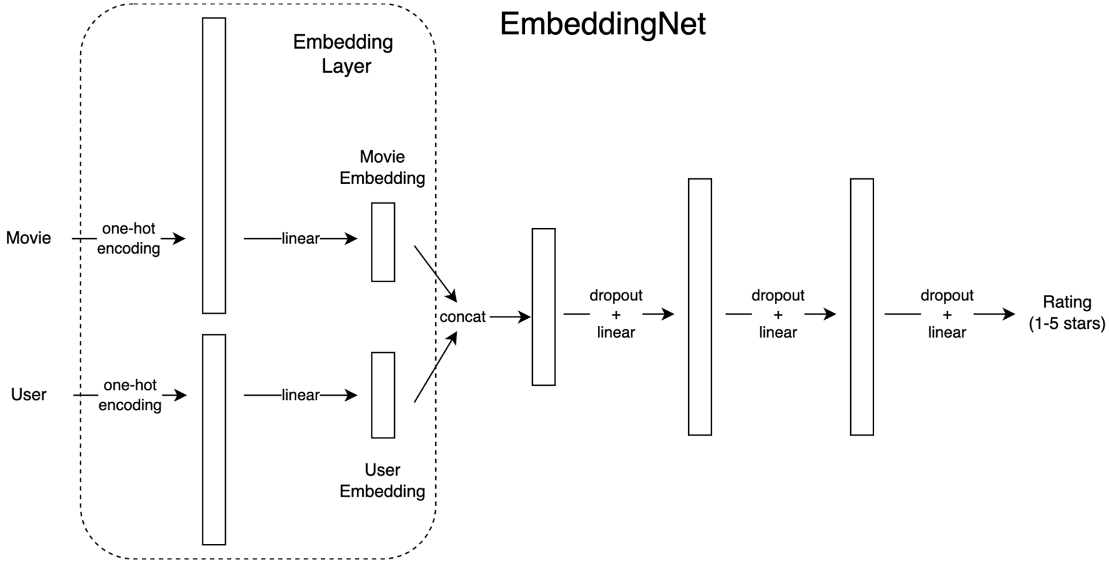


Figure 18.3: Architecture of an EmbeddingNet model with user and movie as input and rating as output. First, we have an embedding layer that transforms movie and user into fixed-sized embeddings, and then we have a series of linear layers, ultimately producing a single (rating) value

First, the movie and the user are converted into their respective one-hot encodings. If there are 8 movies and 5 users in the database, then the one-hot-encoding vectors are of sizes 8 and 5 for movies and users, respectively.

Note

In the case of a huge number of movies and/or users, one-hot encoding is inefficient as it results in large vectors where all elements (except one) are 0. In such cases, techniques such as feature hashing are used where a movie or user ID is first passed through a hash function to produce a number k that lies between 0 and N (where N is a fixed number much smaller than the total number of movies or users). The hash encoding is hence a vector of size N where the kth element is 1 and the other elements are 0. Compared to one-hot encoding, feature hashing produces smaller vectors with a (manageable) risk of mapping multiple movies or users to the same hash encoding.

These one-hot encodings are then passed through two separate linear layers of the network, producing movie and user embedding vectors. These vectors are concatenated and further passed through a series of dropout and linear layers, ultimately producing a rating value between 1 and 5 (normalized).

Note

Besides the one-hot encoding of users and movies into vectors, recommendation systems also typically use user and movie metadata, such as age, gender, and language, of the user, and genre, length, language of the movie. The extra user metadata helps recommend movies to new users by looking up existing users most similar to the new user. The extra movie metadata helps recommend new movies to users by looking up existing movies most similar to the new movie.

The EmbeddingNet is trained from end to end on a training set that contains available ratings from existing users on existing movies. To train the model, we use mean squared error between actual and predicted ratings (values between 1 and 5) as the loss function. Once trained, the model can then be used to predict ratings (filling the question marks in *Figure 18.2*). The user and movie embeddings produced in the process can also be used to find similar users and similar movies.

To train the EmbeddingNet using PyTorch, we first need a dataset. In the next section, we will explore such a dataset – the MovieLens dataset. We download a small version of this dataset, analyze it, and process it to use for training EmbeddingNet.

Understanding and processing the MovieLens dataset

In this section, we dive into the code for creating our recommendation system. As with most ML projects, it all starts with data. We use the MovieLens dataset to create a movie recommendation system.

The MovieLens dataset is a widely used benchmark dataset in the field of recommender systems. It consists of user ratings and movie metadata, providing a rich source for training and evaluating recommendation algorithms. The dataset includes various versions, with MovieLens 100K, 1M, 10M, and 20M being some of the commonly used subsets, differing in the number of ratings and movies. In this chapter, we use the MovieLens 100K dataset, which contains over 100K movie ratings.

As shown in *Figure 18.4*, we first begin by downloading the dataset. We then load the dataset files as DataFrames, analyze the different DataFrames, and clean the dataset if needed.

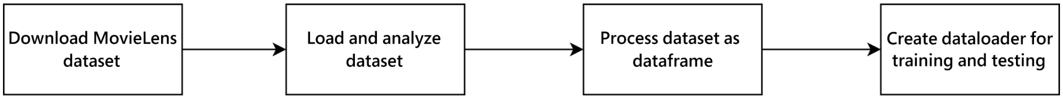

Download MovieLens dataset
Load and analyze dataset
Process dataset as dataframe
Create dataloader for training and testing

Figure 18.4: Steps in exploring, analyzing, and processing the MovieLens dataset

Next, we process the dataset in a format that is usable for training an EmbeddingNet model. Finally, we create training and testing dataloaders from the processed dataset. While we discuss most of the code in the chapter, some verbose code is omitted. You can find the full code for this chapter on GitHub [2].

Downloading the MovieLens dataset

The MovieLens 100K dataset is available at the following URL:

```
DATASET_LINK="https://files.grouplens.org/datasets/movielens/ml-latest-small.
zip"
```

The following lines of code download and extract the dataset into our local disk:

```
!wget -nc $DATASET_LINK
!unzip -n ml-latest-small.zip
```

This should produce an output similar to the following:

```
https://files.grouplens.org/datasets/movielens/ml-latest-small.zip
...
Saving to: 'ml-latest-small.zip'
ml-latest-small.zip 100%[===================>] 955.28K  1.52MB/s    in 0.6s

Archive:  ml-latest-small.zip
   creating: ml-latest-small/
  ...
  inflating: ml-latest-small/ratings.csv
  inflating: ml-latest-small/movies.csv
```

In two stages, we download a 1.5 MB ZIP file containing the dataset and then unzip the file into several CSV files. Two of the most important files are `ratings.csv` and `movies.csv`, which contain information about user ratings and movie metadata, respectively. Let us load these CSV files and understand their content.

Loading and analyzing the MovieLens dataset

We load the movie and user ratings data as pandas DataFrames using a defined utility function `read_data`:

```
ratings, movies = read_data('./ml-latest-small/')
```

Let us examine the contents of the `ratings` dataframe:

```
ratings.head()
```

This should produce an output similar to *Table 18.1*:

userId	movieId	rating	timestamp
1	1	4.0	964982703
1	3	4.0	964981247
1	6	4.0	964982224
1	47	5.0	964983815
1	50	5.0	964982931

Table 18.1: Dataframe showing timestamped user ratings for different movies

This dataframe contains users, the movies they rated, the ratings, and the timestamp when the rating was done. Let us examine the range of rating values with the following line of code:

```
minmax = ratings.rating.min(), ratings.rating.max()
```

The `minmax` variable contains the values (0.5, 5.0), indicating that the ratings vary from 0.5 to 5.0 (with a step of 0.5). We will use this information later to normalize the rating values between 0 and 1 to create our model training target. Next, let us look at the `movies` dataframe:

```
movies.head()
```

This should produce an output similar to *Table 18.2*.

movieId	title	genres
1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
2	Jumanji (1995)	Adventure|Children|Fantasy
3	Grumpier Old Men (1995)	Comedy|Romance
4	Waiting to Exhale (1995)	Comedy|Drama|Romance
5	Father of the Bride Part II (1995)	Comedy

Table 18.2: Dataframe showing the title and genres for each movie of the MovieLens database

This dataframe contains the information about each movie – the movie title and the different genres it belongs to. We can join this dataframe with the `ratings` dataframe to capture the movie name (title) inside the `ratings` dataframe:

```
ratings = ratings.merge(movies[["movieId", "title"]], on="movieId")
ratings.head()
```

This should produce an output similar to *Table 18.3*.

userId	movieId	rating	timestamp	title
1	1	4.0	964982703	Toy Story (1995)
5	1	4.0	847434962	Toy Story (1995)
7	1	4.5	1106635946	Toy Story (1995)
15	1	2.5	1510577970	Toy Story (1995)
17	1	4.5	1305696483	Toy Story (1995)

Table 18.3: Dataframe showing timestamped user ratings for different movies, along with the respective movie titles.

This dataframe looks more complete as it contains both the ratings as well as the movie information. We will use this dataframe to build our movie recommender system, which takes in a user (`userId`) as input and produces the top-k movie titles as recommendations.

To recreate the visualization presented in *Figure 18.2*, we can also look at the ratings dataframe in the form of a user-movie matrix, using a pre-defined function `tabular_preview`:

```
tabular_preview(ratings, 10)
```

The above function takes an additional argument `n` (10 in this case), where `n` refers to the number of most active users and most watched movies to be displayed using this function. This line of code should produce an output similar to *Table 18.4*.

movieId	**110**	**260**	**296**	**318**	**356**	**480**	**527**	**589**	**593**	**2571**
userId										
68	2.5	5.0	2.0	3.0	3.5	3.5	4.0	3.5	3.5	4.5
274	4.5	3.0	5.0	4.5	4.5	3.5	4.0	4.5	4.0	4.0
288	5.0	5.0	5.0	5.0	5.0	2.0	5.0	4.0	5.0	3.0
380	4.0	5.0	5.0	3.0	5.0	5.0	NaN	5.0	5.0	4.5
414	5.0	5.0	5.0	5.0	5.0	4.0	4.0	5.0	4.0	5.0
448	NaN	5.0	5.0	NaN	3.0	3.0	NaN	3.0	5.0	2.0
474	3.0	4.0	4.0	5.0	3.0	4.5	5.0	4.0	4.5	4.5
599	3.5	5.0	5.0	4.0	3.5	4.0	NaN	4.5	3.0	5.0
606	3.5	4.5	5.0	3.5	4.0	2.5	5.0	3.5	4.5	5.0
610	4.5	5.0	5.0	3.0	3.0	5.0	3.5	5.0	4.5	5.0

Table 18.4: User-movie matrix for the top 10 most active users and top 10 most watched movies from the MovieLens 100K dataset

Notice the resemblance between *Table 18.4* and *Figure 18.2*. The NaNs in *Table 18.4* are equivalent to the question marks in *Figure 18.2*, both referring to the absence of a rating value, meaning that the given user hasn't rated the given movie. *Table 18.4* lists the top 10 most active users and the top 10 most watched movies, represented by their IDs (`userId` and `movieId`) from the dataset.

The rating values are either floating-point numbers ranging between 0.5 and 5, with values such as 2.0, 3.5, 3.5, and so on, or null (NaN). Our goal is to train a recommendation system using non-null ratings and replace the null ratings with predicted ratings to thereby compile movie recommendations for every user.

This concludes our exploration of the dataset. In the next section, we process this dataset to convert it into a format that is suitable and usable for training a PyTorch model.

Processing the MovieLens dataset

We define a `create_dataset` function that takes in the `ratings` dataframe as input and produces a formatted training dataset as output:

```
def create_dataset(ratings, top=None):
    if top is not None:
        ratings.groupby('userId')['rating'].count()
    unique_users = ratings.userId.unique()
    user_to_index = {old: new for new, old in enumerate(unique_users)}
    new_users = ratings.userId.map(user_to_index)
    unique_movies = ratings.movieId.unique()
    movie_to_index = {
        old: new for new, old in enumerate(unique_movies)}
    new_movies = ratings.movieId.map(movie_to_index)
    n_users = unique_users.shape[0]
    n_movies = unique_movies.shape[0]
    X = pd.DataFrame({'user_id': new_users, 'movie_id': new_movies})
    y = ratings['rating'].astype(np.float32)
    return (n_users, n_movies), (X, y), (user_to_index, movie_to_index)
```

This function reindexes the users and movies and stores a mapping (under the variables `user_to_index` and `movie_to_index`, respectively) between their new indices and the original IDs (`userId` and `movieId`, respectively). It then creates the input data for training, which contains the user and movie indices, and the output data, which contains the corresponding rating value (not normalized) for the given user and the given movie. Let us now execute this function to create our dataset:

```
(n, m), (X, y), (user_to_index, movie_to_index) = create_dataset(ratings)
print(f'Embeddings: {n} users, {m} movies')
print(f'Dataset shape: {X.shape}')
print(f'Target shape: {y.shape}')
```

This should produce the following output.

```
Embeddings: 610 users, 9724 movies
Dataset shape: (100836, 2)
Target shape: (100836,)
```

The dataset contains 610 users, 9,724 movies, and over 100K ratings (because we are using the MovieLens 100K dataset). So far, we have partitioned the data into input and output datasets. We further split the dataset into training and validation sets using the following lines of code:

```
X_train, X_valid, y_train, y_valid = train_test_split(
    X, y, test_size=0.2, random_state=RANDOM_STATE)
datasets = {'train': (X_train, y_train), 'val': (X_valid, y_valid)}
dataset_sizes = {'train': len(X_train), 'val': len(X_valid)}
```

This concludes the creation of a formatted dataset for training a deep learning model in PyTorch.

Note

Instead of splitting training and validation sets randomly, we can also split them based on the rating timestamps. That way, we retain older ratings as the training set, and predict future ratings belonging to the validation set. This also opens opportunities to build user features such as movies liked by user A so far.

In the next section, we will use this formatted dataset (with input and output DataFrames) to create dataloaders.

Creating the MovieLens dataloader

To train the EmbeddingNet model using PyTorch, we create a dataset iterator that can iterate over the several instances (ratings) of our dataset:

```
class ReviewsIterator:
    def __init__(self, X, y, batch_size=32, shuffle=True):
        ...
        self.n_batches = int(math.ceil(X.shape[0] // batch_size))
        self._current = 0
    def __iter__(self):
        return self
    def __next__(self):
        return self.next()
    def next(self):
        ...
        return self.X[k*bs:(k + 1)*bs], self.y[k*bs:(k + 1)*bs]
```

The iterator simply takes in the batch size as input and iterates over batches of the dataset. We create the following function to facilitate iterating over the dataset using the `ReviewsIterator` class:

```
def batches(X, y, bs=32, shuffle=True):
    for xb, yb in ReviewsIterator(X, y, bs, shuffle):
        xb = torch.LongTensor(xb)
        yb = torch.FloatTensor(yb)
        # yield inputs (xb) and targets (yb) reshaped to have 1 column.
        yield xb, yb.view(-1, 1)
```

We can test the above function with the following lines of code:

```
for x_batch, y_batch in batches(X, y, bs=4):
    print(x_batch)
    print(y_batch)
    break
```

This should produce an output similar to the following:

```
tensor([[ 341, 1891],
        [  83,  907],
        [ 106, 5749],
        [ 146,   61]])
tensor([[4.],
        [2.],
        [4.],
        [5.]])
```

As we can see, the iterator produces 1 batch of data (with a batch size of 4 samples), with inputs containing user and movie indices, and output containing raw ratings values. We are now ready to use the MovieLens data to train an EmbeddingNet model. In the next section, we train a movie recommender system using PyTorch.

Training and evaluating a recommendation system model

In this section, we first define an EmbeddingNet model using PyTorch.

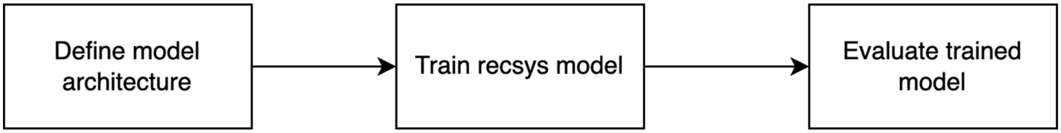


Figure 18.5: Steps in building, training, and evaluating an EmbeddingNet model

We train this model on the MovieLens dataset to thereafter predict user ratings for unseen movies. We finally evaluate the trained model on the validation set.

Defining the EmbeddingNet architecture

We define the EmbeddingNet model with the following lines of PyTorch code:

```
class EmbeddingNet(nn.Module):
    def __init__(self, n_users, n_movies,
                 n_factors=50, embedding_dropout=0.02,
                 hidden=10, dropouts=0.2):
        ...
        n_last = hidden[-1]
        def gen_layers(n_in):
            nonlocal hidden, dropouts
            for n_out, rate in zip_longest(hidden, dropouts):
                yield nn.Linear(n_in, n_out)
                yield nn.ReLU()
                if rate is not None and rate > 0.:
                    yield nn.Dropout(rate)
                n_in = n_out
        self.u = nn.Embedding(n_users, n_factors)
        self.m = nn.Embedding(n_movies, n_factors)
        self.drop = nn.Dropout(embedding_dropout)
        self.hidden = nn.Sequential(*list(gen_layers(n_factors * 2)))
        self.fc = nn.Linear(n_last, 1)
        self._init()
```

The initialization of the model includes the following parameters:

- Total number of users (`n_users`): This is needed to convert user indices into one-hot vectors
- Total number of movies (`n_movies`): This is needed to convert movie indices into one-hot vectors
- `n_factors`: This is the size of the user as well as movie embeddings
- `embedding_dropout`: Dropout to be applied at the embedding layer, right after the linear layer that produces embeddings
- `hidden`: This can be a number or a list of numbers reflecting the size(s) of the hidden layer(s) that follow the embedding layer (see *Figure 18.3*)
- `dropouts`: This is the dropout or list of dropouts applied to the hidden layer(s)

The above code uses the listed arguments to generate a model as per the architecture shown in *Figure 18.3*. In the last two lines of the above code, the final layer of the model is declared with a single output that produces the rating value, and an `_init` method is called. Let us look at how this method is defined:

```
    def _init(self):
        def init(m):
            if type(m) == nn.Linear:
               torch.nn.init.xavier_uniform_(m.weight)
               m.bias.data.fill_(0.01)
        self.u.weight.data.uniform_(-0.05, 0.05)
        self.m.weight.data.uniform_(-0.05, 0.05)
        self.hidden.apply(init)
        init(self.fc)
```

The `_init` method essentially initializes the weights of the hidden linear layers of the EmbeddingNet with Xavier initialization [3] and initializes the embedding layer weights with uniform (random) initialization. Having defined the model architecture, let us look at how the forward pass to this model is defined:

```
def forward(self, users, movies, minmax=None):
        features = torch.cat([self.u(users), self.m(movies)], dim=1)
        x = self.drop(features)
        x = self.hidden(x)
        out = torch.sigmoid(self.fc(x))
        if minmax is not None:
            min_rating, max_rating = minmax
            out = out*(max_rating - min_rating + 1) + min_rating - 0.5
        return out
```

The `forward` method takes in the following arguments:

- List of user indices (`users`)
- List of movie indices (`movies`) with the same length as `users`
- `minmax`: Minimum and maximum rating values possible, to de-normalize the predicted (normalized) rating values

In the forward pass, the model follows the sequence of steps as demonstrated in the network architecture in *Figure 18.3*. The user and movie embeddings are concatenated and passed through a dropout layer followed by linear layer(s), and finally followed by a sigmoid activation resulting in a single value between 0 and 1, which is then de-normalized to a rating value between 0.5 and 5. Having defined the EmbeddingNet model architecture using PyTorch, let us instantiate such a model object:

```
net = EmbeddingNet(
    n_users=n, n_movies=m,
    n_factors=150, hidden=[500, 500, 500],
    embedding_dropout=0.05, dropouts=[0.5, 0.5, 0.25])
```

We define the size of user and movie embedding vectors as `150`.

Note

The embedding vector size is one of the hyperparameters of an embeddings-based recommendation system model. A high value increases the number of model parameters and can lead to overfitting, except if you have large amounts of data. In general, different values of embedding vector size should be tried to get the value that produces the most accurate recommendations. We have used `150` in this chapter for demonstration.

These are concatenated to form a vector of size `300`, which is followed by a dropout of `0.05`, which is then followed by 3 hidden layers each of size `500`, each followed by dropouts of values `0.5`, `0.5`, and `0.25`, respectively. The network should like the following:

```
EmbeddingNet(
  (u): Embedding(610, 150)
  (m): Embedding(9724, 150)
  (drop): Dropout(p=0.05, inplace=False)
  (hidden): Sequential(
    (0): Linear(in_features=300, out_features=500, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=500, out_features=500, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=500, out_features=500, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.25, inplace=False)
  )
  (fc): Linear(in_features=500, out_features=1, bias=True)
)
```

This concludes the definition of the EmbeddingNet model. In the next section, we will train this model on the MovieLens dataset.

Training EmbeddingNet

Before we start the model training loop, let us first define the hyperparameters and other configuration parameters:

```
lr = 1e-5  # learning rate
wd = 1e-5  # weight decay
bs = 200  # batch size
n_epochs = 200
patience = 10
no_improvements = 0
```

```
best_loss = np.inf
best_weights = None
history = []
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
net.to(device)
criterion = nn.MSELoss(reduction='sum')
optimizer = optim.Adam(net.parameters(), lr=lr, weight_decay=wd)
iterations_per_epoch = int(math.ceil(dataset_sizes['train'] // bs))
```

We define a learning rate of `1e-5`, weight decay of `1e-5`, batch size of `200`, training for `200` epochs, and a `patience` of `10`, where patience is the number of consecutive epochs to wait with no further improvements before we stop training. We define a `no_improvements` flag to reflect the *no further improvements* phase of the model.

We define the best loss so far as infinity, and the best model weights so far as null. We define a `history` variable that keeps track of the training and validation losses along the epochs. We define the device to train the PyTorch model on, the loss function as the mean squared error loss, optimizer as the Adam optimizer, and `iterations_per_epoch` as the number of batches (of 200 samples) of the training dataset that can be processed within an epoch.

With all the hyperparameters defined, we can now run the model training loop, where we train and validate the model on the training and validation datasets, respectively, using gradient descent with backpropagation:

```
for epoch in range(n_epochs):
    for phase in ('train', 'val'):
        ...
        for batch in batches(*datasets[phase], shuffle=training, bs=bs):
            ...
            with torch.set_grad_enabled(training):
                outputs = net(x_batch[:, 0], x_batch[:, 1], minmax)
                loss = criterion(outputs, y_batch)
                if training:
                    loss.backward()
                    optimizer.step()
            running_loss += loss.item()
        epoch_loss = running_loss / dataset_sizes[phase]
        if phase == 'val':
            if epoch_loss < best_loss:
                best_loss = epoch_loss
                best_weights = copy.deepcopy(net.state_dict())
                no_improvements = 0
            else:
                no_improvements += 1
    history.append(stats)
    if no_improvements >= patience:
        break
```

This should produce an output similar to the following:

```
loss improvement on epoch: 1
[001/200] train: 1.1996 - val: 1.0651
loss improvement on epoch: 2
[002/200] train: 1.0806 - val: 1.0494
...
[048/200] train: 0.6331 - val: 0.7778
[049/200] train: 0.6326 - val: 0.7714
early stopping after epoch 049
```

In the above code, we first run a forward pass through the model to obtain de-normalized predicted rating values for pairs of users and movies in the training batches. These predicted ratings are compared to the actual user ratings (ground truth) to obtain the mean squared error. Using gradient descent with backpropagation, this error is backpropagated to tune the weights of all layers of the EmbeddingNet. After 200 epochs, we get the trained model. We can check the learning curve with the following line of code:

```
ax = pd.DataFrame(history).drop(columns='total').plot(x='epoch')
```

This should produce an output as shown in *Figure 18.6*.

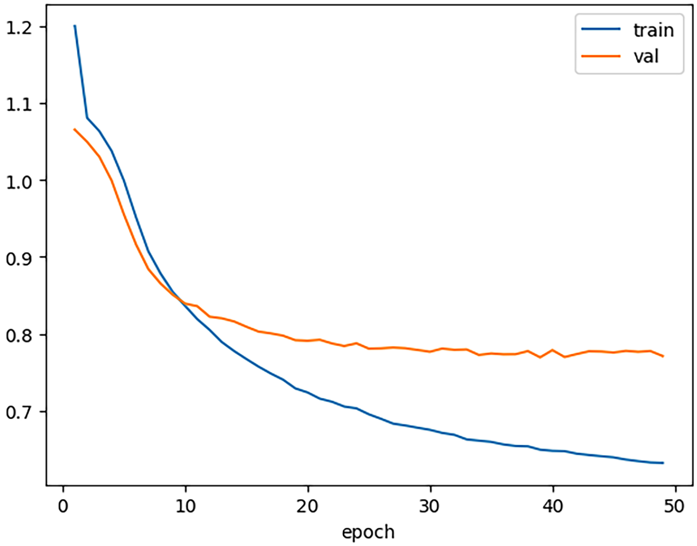


Figure 18.6: Learning curve of EmbeddingNet showing a stable training trajectory with validation loss going down along with training loss, until early stopping

The training stops at around 50 epochs due to the early stopping mechanism (using `patience`). The training curve indicates that the model learns user rating patterns that are generalizable from the training set to the validation set, indicating a successful model training. In the next section, we evaluate the performance of the trained model on the validation set using mean squared error as the metric.

Evaluating the trained EmbeddingNet model

We run the validation set user-movie pairs through the trained model to generate predicted ratings with the following lines of code:

```
groud_truth, predictions = [], []
with torch.no_grad():
    for batch in batches(*datasets['val'], shuffle=False, bs=bs):
        x_batch, y_batch = [b.to(device) for b in batch]
        outputs = net(x_batch[:, 0], x_batch[:, 1], minmax)
        groud_truth.extend(y_batch.tolist())
        predictions.extend(outputs.tolist())
groud_truth = np.asarray(groud_truth).ravel()
predictions = np.asarray(predictions).ravel()
```

We can now compare the predicted ratings (`predictions`) with the actual ratings (`ground_truth`) with the following lines of code:

```
final_loss = np.sqrt(
    np.mean((np.array(predictions) - np.array(groud_truth))**2))
print(f'Final RMSE: {final_loss:.4f}')
```

This should produce an output similar to the following:

```
Final RMSE: 0.8816
```

This indicates that on average the trained model is less than 1 rating away from the ground truth in predicting the correct user rating for unseen movies. We can further check some of the predicted ratings with the following line of code:

```
np.array(predictions)
```

This should produce an output similar to the following:

```
array([3.38753653, 2.63408899, 3.20616698, ...])
```

We can check the corresponding ground-truth ratings with the following line of code:

```
datasets["val"][1][:5]
```

This should produce an output similar to *Table 18.5*.

67037	4.0
42175	2.0
93850	4.0
6187	5.0
12229	4.0

Table 18.5: Dataframe showing rating values (between 0.5 and 5.0) from the validation set of the MovieLens dataset

As we can see, the predictions 3.38, 2.63, and 3.2 are within 1 unit of rating of the ground-truth values 4, 2, and 4, respectively, reaffirming the obtained mean squared error metric score of 0.8816.

This concludes the training and evaluation of the EmbeddingNet. In the next and final section of this chapter, we will use EmbeddingNet to create a recommendation system, where we provide a user as input, and get the top-k movie recommendations as output.

Building a recommendation system using the trained model

In this section, we create a recommendation system in the form of a function that returns a list of movie titles as recommendations for a given user. As shown in *Figure 18.7*, we first fetch all movies not seen by the user. Each of these movies is fed along with the given user as input to the EmbeddingNet, producing the respective predicted ratings. The movies are then sorted in decreasing order of rating, and the top-k movies are returned as recommendations.

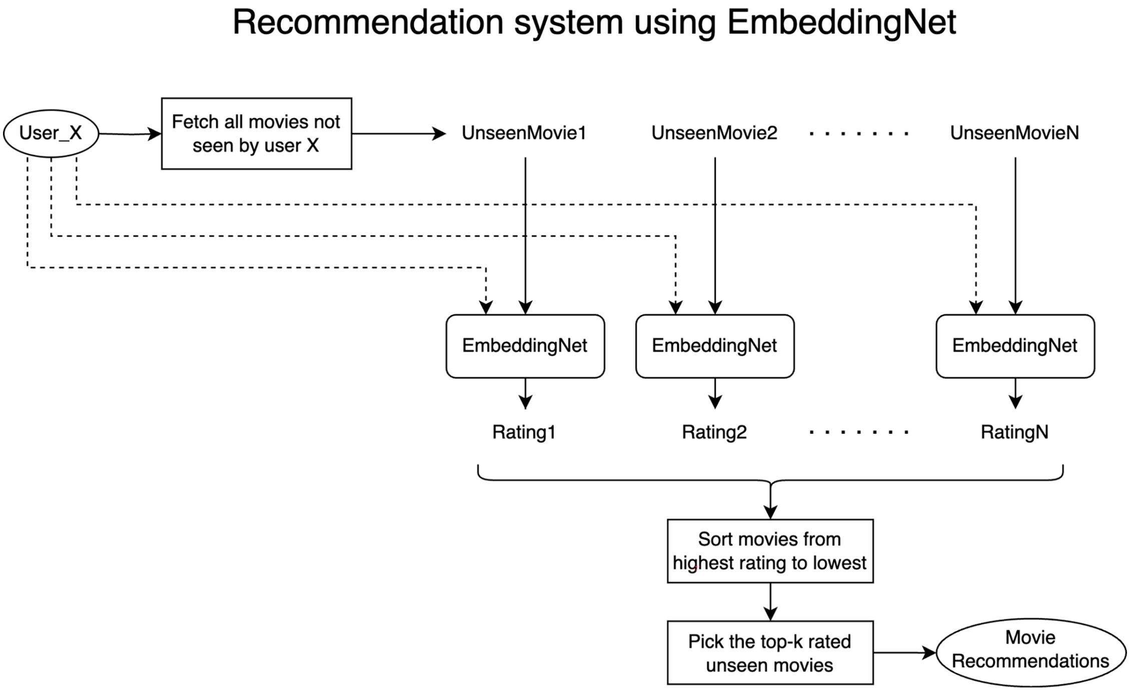

Recommendation system using EmbeddingNet
User_X
Fetch all movies not seen by user X
UnseenMovie1
UnseenMovie2
. . . .
UnseenMovieN
EmbeddingNet
EmbeddingNet
EmbeddingNet
Rating1
Rating2
. . . .
RatingN
Sort movies from highest rating to lowest
Pick the top-k rated unseen movies
Movie Recommendations

Figure 18.7: Schematic representation of a movie recommendation system that uses an EmbeddingNet under the hood to generate ratings on movies not seen by a user

The following function performs the steps outlined in *Figure 18.7*:

```
def recommender_system(user_id, model, n_movies):
    seen_movies = set(X[X['user_id'] == user_id]['movie_id'])
    user_ratings = y[X['user_id'] == user_id]
    top_rated_movie_ids = X.loc[(X['user_id'] == user_id) &
                         (y == user_ratings.max()), "movie_id"]
print("\n".join(
    movies[movies.movieId.isin(top_rated_movie_ids)].title.iloc[:10].tolist()))
    unseen_movies = list(set(ratings.movieId) - set(seen_movies))
    unseen_movies_index = [movie_to_index[i] for i in unseen_movies]
    model_input = (torch.tensor([user_id]*len(unseen_movies_index),
                                device=device),
                   torch.tensor(unseen_movies_index, device=device))
    with torch.no_grad():
        predicted_ratings = model(*model_input, minmax).detach().numpy()
    zipped_pred = zip(unseen_movies, predicted_ratings)
    sorted_movie_index = list(zip(
        *sorted(zipped_pred, key=lambda c: c[1], reverse=True)))[0]
    recommended_movies = movies[
        movies.movieId.isin(sorted_movie_index)].title.tolist()
    print("\n".join(recommended_movies[:n_movies]))
```

The `recommender_system` function takes the following inputs:

- `user_id`: Index of the given user who we want to recommend movies to
- `model`: Trained EmbeddingNet model object
- `n_movies`: Number of movies to be returned by the recommendation system

We call the above function with some sample values, as shown below:

```
recommender_system(32, net, 10)
```

We choose user number 32 and ask for 10 movie recommendations. This should produce an output similar to the following:

```
Total movies seen by the user: 575
======================================================
Some top rated movies (rating = 5.0) seen by the user:
======================================================
Jumanji (1995)
GoldenEye (1995)
...
Mighty Morphin Power Rangers: The Movie (1995)
======================================================
Top 10 Movie recommendations for the user 32 are:
======================================================
Father of the Bride Part II (1995)
Heat (1995)
Sudden Death (1995)
American President, The (1995)
Four Rooms (1995)
Get Shorty (1995)
Assassins (1995)
Powder (1995)
Persuasion (1995)
It Takes Two (1995)
```

The function first returns the total number of movies already watched by user 32. It then shows some (10) of the already watched movies by user 32, to give an idea of the user's taste. It then lists the 10 best movie recommendations for this user, based on the highest predicted ratings by this user according to the trained EmbeddingNet.

This concludes our discussions on building a recommendation system from scratch using PyTorch and the MovieLens dataset. The concepts from this chapter can be used in different kinds of recommendation systems, with different underlying models and different data modalities.

Building a recommendation system often involves working on a large scale of data, with millions of users and products, and over billions of ratings. To tackle such massive-scale problems, PyTorch has introduced the **TorchRec** library [4]. This library provides common sparsity and parallelism functionalities needed for large-scale recommender systems. It also allows training models with large embedding tables shared across many GPUs. This library is in the early stages of development (at the time of writing) and hence we do not cover **TorchRec** in this chapter in detail. While the TorchRec webpage has a tutorial [5] to get started, I recommend keeping an eye on this space for exciting developments soon.

Summary

In this chapter, we built a recommendation system from scratch using PyTorch. We first learned how to use deep learning to power a recommendation system. We then explored and analyzed the MovieLens dataset. We then defined an EmbeddingNet model using PyTorch and trained and evaluated it on the MovieLens dataset.

Finally, we used the trained EmbeddingNet model to create a movie recommendation system. In the next and final chapter of this book, we will learn more about the Hugging Face ecosystem and how PyTorch users can benefit from different Hugging Face products, components, and libraries.

Reference list

1. MovieLens datasets: `https://grouplens.org/datasets/movielens/latest/`
2. GitHub 1: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter18/torch-recsys.ipynb`
3. Understanding the difficulty of training deep feedforward neural networks: `https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf`
4. TorchRec documentation: `https://pytorch.org/torchrec/`
5. TorchRec GitHub: `https://github.com/pytorch/torchrec/blob/main/Torchrec_Introduction.ipynb`

Learn more on Discord

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

`https://packt.link/mastorch`

19

PyTorch and Hugging Face

We learned about parts of Hugging Face in *Chapter 7*, *Music and Text Generation with PyTorch*, as well as *Chapter 10*, *Text-to-Image Generation*. Hugging Face is an open-source platform and community-driven library that provides a comprehensive suite of AI tools, pre-trained models, and a collaborative ecosystem for developing and sharing state-of-the-art models. It has become one of the foundational platforms in the current AI landscape. We dedicate this chapter to learning more about Hugging Face and how PyTorch users can benefit from Hugging Face in researching, training, evaluating, optimizing, and deploying deep learning models.

By the end of this chapter, you will be able to use Hugging Face in your deep learning projects. You will be able to use pre-trained models from the Hugging Face Hub, use the **Transformers** library with PyTorch, speed up model training using **Accelerate**, and optimize your trained PyTorch models for deployment using **Optimum**.

This chapter is broken down into the following topics:

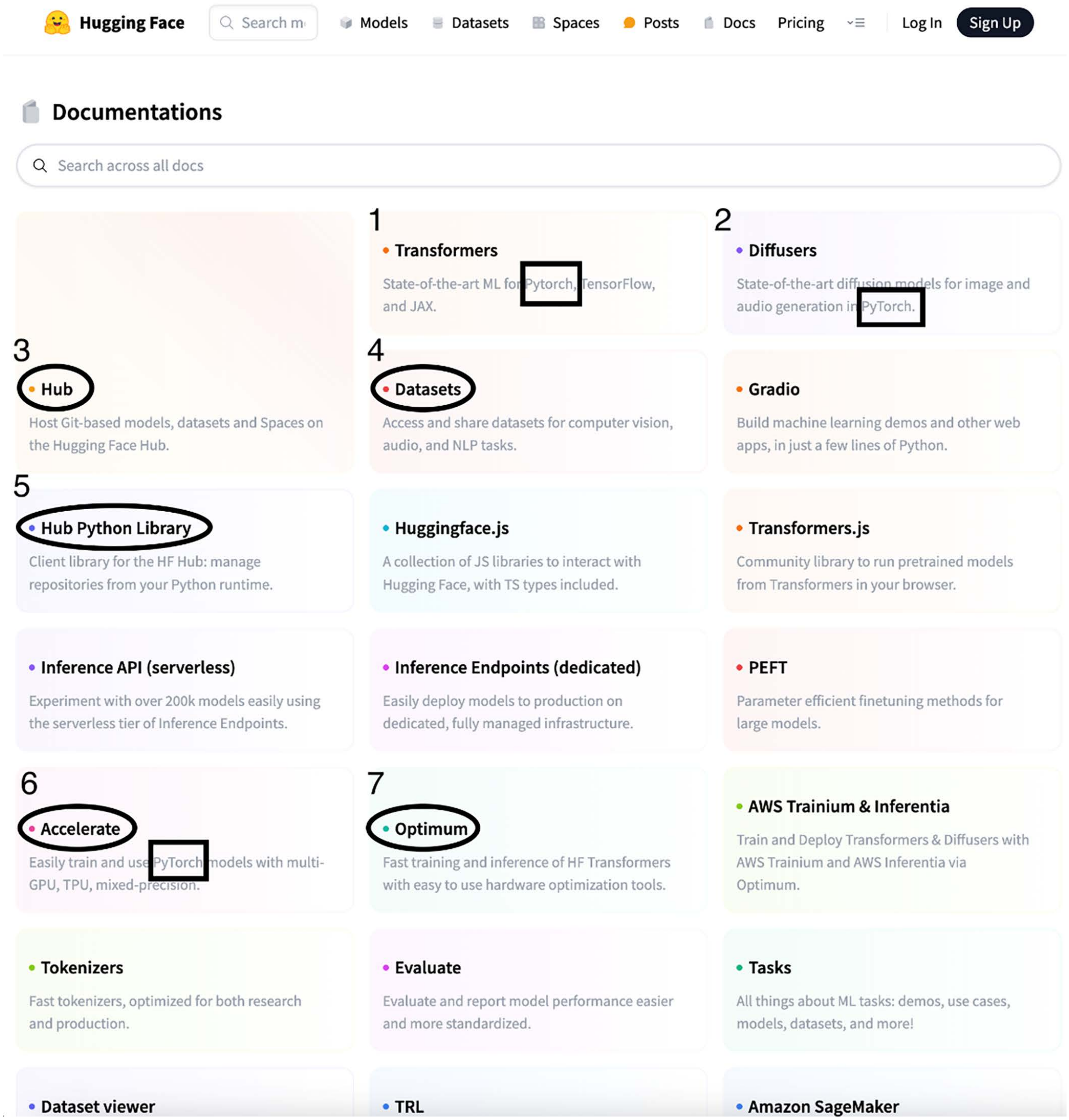
- Understanding Hugging Face within the PyTorch context
- Using the Hugging Face Hub for pre-trained models
- Using Hugging Face datasets with PyTorch
- Using Accelerate to speed up PyTorch model training
- Using Optimum to optimize PyTorch model deployment

Understanding Hugging Face within the PyTorch context

Hugging Face [1] is a rapidly growing multi-faceted AI company. On the one hand, it provides a host of libraries related to training, evaluating, optimizing, and deploying AI models. On the other hand, it is a hub of various AI models, datasets, and live AI demos (referred to as spaces in Hugging Face jargon). Hugging Face is quickly evolving into an AI community where developers are sharing cutting-edge AI work and having discussions that push the frontier of AI.

Exploring Hugging Face components relevant to PyTorch

We can view Hugging Face as a platform that contains various components, as shown in *Figure 19.1*. You can access the page shown in the figure on the Hugging Face website [2]. The first thing to note in this figure is the mention of PyTorch in various Hugging Face components. Hugging Face's libraries, models, and datasets are fully compatible with PyTorch, so it is wise to discuss Hugging Face in detail in a PyTorch book.



Documentations

Search across all docs

- **Transformers** State-of-the-art ML for Pytorch, TensorFlow, and JAX.
- **Diffusers** State-of-the-art diffusion models for image and audio generation in PyTorch.
- **Hub** Host Git-based models, datasets and Spaces on the Hugging Face Hub.
- **Datasets** Access and share datasets for computer vision, audio, and NLP tasks.
- **Gradio** Build machine learning demos and other web apps, in just a few lines of Python.
- **Hub Python Library** Client library for the HF Hub: manage repositories from your Python runtime.
- **Huggingface.js** A collection of JS libraries to interact with Hugging Face, with TS types included.
- **Transformers.js** Community library to run pretrained models from Transformers in your browser.
- **Inference API (serverless)** Experiment with over 200k models easily using the serverless tier of Inference Endpoints.
- **Inference Endpoints (dedicated)** Easily deploy models to production on dedicated, fully managed infrastructure.
- **PEFT** Parameter efficient finetuning methods for large models.
- **Accelerate** Easily train and use PyTorch models with multi-GPU, TPU, mixed-precision.
- **Optimum** Fast training and inference of HF Transformers with easy to use hardware optimization tools.
- **AWS Trainium & Inferentia** Train and Deploy Transformers & Diffusers with AWS Trainium and AWS Inferentia via Optimum.
- **Tokenizers** Fast tokenizers, optimized for both research and production.
- **Evaluate** Evaluate and report model performance easier and more standardized.
- **Tasks** All things about ML tasks: demos, use cases, models, datasets, and more!
- **Dataset viewer**
- **TRL**
- **Amazon SageMaker**

Figure 19.1: Various Hugging Face components. Notice the mention of PyTorch under various components. Numbered are some of the components of interest to us. Components numbered 1 and 2 have already been discussed in Chapters 7 and 10 respectively. The other numbered components are covered in this chapter.

The second thing to note is that, of the various components shown in *Figure 19.1*, we highlight some of the components that will especially interest you. While components 1 (the **Transformers** library) and 2 (the **Diffusers** library) were mentioned in *Chapter 7*, *Music and Text Generation with PyTorch*, and *Chapter 10*, *Text-to-Image Generation*, respectively, we are interested in discussing the following components:

- **Hub** (or **Hugging Face Hub**) and **Hub library:** This is where all AI models, datasets, and spaces are accessed. This is one of the most widely used features of Hugging Face as it enables developers to easily access cutting-edge AI models and datasets to either build models on top of, or for inference. The Hub library enables us to access the Hub via Python code.
- **Datasets:** Besides the dataset library inside PyTorch, the Hugging Face Datasets library enables PyTorch users to easily use Hugging Face datasets and even non-Hugging Face datasets.
- **Accelerate:** This is an engineering-focused library that enables us to effectively utilize multi-**Graphical Processing Unit** (**GPU**) or **Tensor Processing Unit** (**TPU**) settings during training and inferencing heavy AI models, as well as utilizing optimization techniques such as mixed precision (discussed in *Chapter 12*, *Model Training Optimizations*).
- **Optimum:** This is another engineering-focused library that enables optimizations on the model deployment front to maximize model performance on specified target hardware. This ensures optimal efficiency, speed, and resource utilization, which can significantly reduce operational costs and improve the user experience.

Integrating Hugging Face with PyTorch

In terms of integration with PyTorch, Hugging Face provides seamless compatibility between its libraries and PyTorch tensors, dataloaders, and GPU support. This allows developers familiar with PyTorch to leverage the benefits of pre-trained models without having to learn new APIs completely.

As a PyTorch user, you can get started with using a Hugging Face library by installing the library. For example, let's install the most popular Hugging Face library, `transformers`, with the following command:

```
pip install torch transformers
```

The `transformers` library is popular due to its extensive collection of pre-trained models, its ease of use in implementing state-of-the-art NLP techniques, and its broad support for tasks such as text classification, translation, and generation, all with consistent and user-friendly APIs. We load a pre-trained model using the `transformers` library with the following snippet of code:

```
from transformers import AutoModelForSequenceClassification, AutoTokenizer

# a pre-trained model from HuggingFace
model_name = "bert-base-uncased"
# Load the pre-trained model and tokenizer
model = AutoModelForSequenceClassification.from_pretrained(model_name)
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

All code for this section is available on GitHub [3]. In the above code, we imported the AutoTokenizer and AutoModelForSequenceClassification classes from the transformers package and initialized them with pre-trained weights (bert-base-uncased) of a **Bidirectional Encoder Representations from Transformers** (**BERT**) model [4]. After encoding our example sentence into tokens, we next run the model on an example input with the following snippet of code:

```
import torch
input_text = "I love PyTorch!"

# Tokenize the input text using the tokenizer
inputs = tokenizer(input_text, return_tensors="pt")

# Perform inference using the pre-trained model
with torch.no_grad():
    outputs = model(**inputs)

# Access the model predictions or outputs
predicted_class = torch.argmax(outputs.logits, dim=1).item()
```

Feeding the input to the BERT model returns hidden states (inside the outputs variable) that capture meaningful representations of words based on their context. We retrieve the logits from the model output and perform argmax on it to get the class with the highest probability in the predicted_class variable.

What this code produces as output is not important. What is important is the demonstration of using the torch library together with the transformers library. In general, torch works well with transformers in making effective use of deep learning models with powerful tensor operations, enabling robust, scalable, and customizable machine learning applications.

Let's take the above code a step further. We can use the parameters of this pre-trained model to instantiate a new training optimizer to, say, fine-tune the pre-trained model, with the following line of code:

```
# Example: Fine-tuning a pre-trained model
# (Assuming 'train_dataset' and 'validation_dataset' are already prepared)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-5)
# Train the model with your dataset and optimizer
```

The above piece of code again shows the interplay between the transformers library and the torch library. The parameters of the pre-trained model from Hugging Face are directly accessible by the Adam optimizer from PyTorch.

We have used the CPU so far to perform model inference, but what if we want to utilize the idle GPU on our machine? We do so with the following lines of code:

```
# Example: Using GPU for model inference
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
model.to(device)
inputs.to(device)
outputs = model(**inputs)
```

Once again, the above code shows the interoperability between `transformers` and `torch` objects. This concludes our initial exploration of using the `torch` library together with `transformers`, a Hugging Face library. Similar interoperability with torch is applicable to other Hugging Face libraries, such as Diffusers (as seen in *Chapter 10*, *Text-to-Image Generation*), Accelerate, and Optimum, as we will see in the later sections in this chapter.

While we were swiftly able to use a pre-trained model, `bert-base-uncased` [4], in this section, in the next section, we will explore a much wider range of resources, including models, datasets, and demos, that are available for use in the **Hugging Face Hub**.

Using the Hugging Face Hub for pre-trained models

In the previous section, we used a pre-trained BERT model as an example to demonstrate the interface between the `transformers` and `torch` libraries. In that example, we used the **Hugging Face Hub** [5] to download and then load a pre-trained BERT model. In this section, we will explore using the Hugging Face Hub to load pre-trained models further and demonstrate how to use the Hugging Face Hub via a Python library and via the Hugging Face website. All the code for this section is available on GitHub [6].

Note

You need to use an API token to access many of the models available on the Hugging Face Hub. You can access your API token from the following page: `https://huggingface.co/settings/tokens`.

To use the Hugging Face Hub Python library, we need to install it with the following command:

```
pip install huggingface_hub
```

Once it is installed, we can run the following command to import the library and fetch all pre-trained models available on the Hugging Face Hub:

```
from huggingface_hub import hf_api
models = hf_api.list_models()
```

At the time of writing this book, there are over 650,000 pre-trained models available on the Hugging Face Hub! You can check this number with the following command (it takes a while to run):

```
len([t for t in models])
```

While we can't look at all of the models, let's look at some of the most popular text-generation models available on the Hugging Face Hub with the following lines of code:

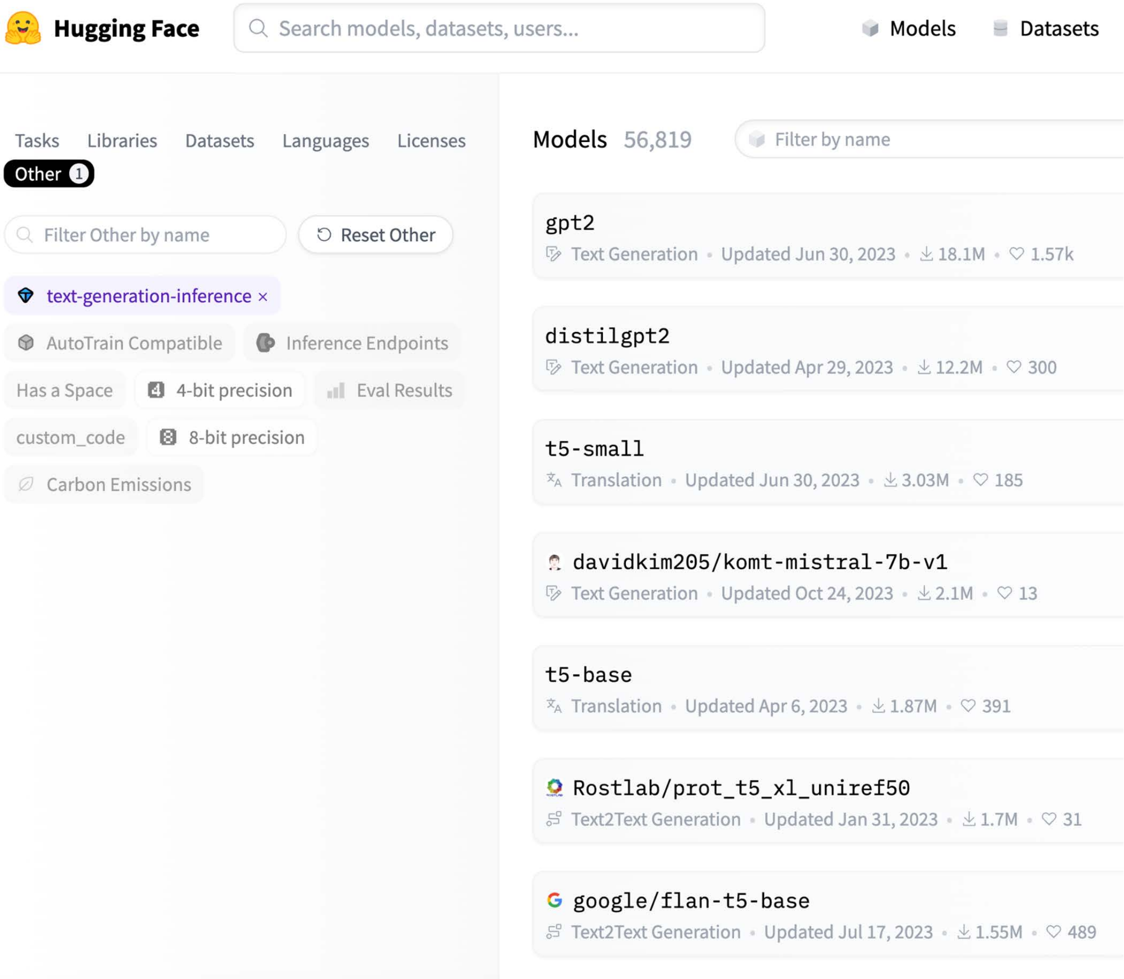
```
text_gen_models = [model.id for model in models
                   if "text-generation-inference" in model.tags
```

```
                        and model.downloads>1000000]
print(text_gen_models)
```

This should produce an output similar to the following:

```
['distilgpt2', 'gpt2', 't5-base', 't5-small', 'Rostlab/prot_t5_xl_uniref50',
'bigscience/bloom-560m', 'google/flan-t5-base', 'tiiuae/falcon-40b-instruct',
'davidkim205/komt-mistral-7b-v1']
```

This list of models can also be achieved via the Hugging Face **Models** page [7], as shown in *Figure 19.2*:



Models 56,819

gpt2
Text Generation • Updated Jun 30, 2023 • 18.1M • 1.57k

distilgpt2
Text Generation • Updated Apr 29, 2023 • 12.2M • 300

t5-small
Translation • Updated Jun 30, 2023 • 3.03M • 185

davidkim205/komt-mistral-7b-v1
Text Generation • Updated Oct 24, 2023 • 2.1M • 13

t5-base
Translation • Updated Apr 6, 2023 • 1.87M • 391

Rostlab/prot_t5_xl_uniref50
Text2Text Generation • Updated Jan 31, 2023 • 1.7M • 31

google/flan-t5-base
Text2Text Generation • Updated Jul 17, 2023 • 1.55M • 489

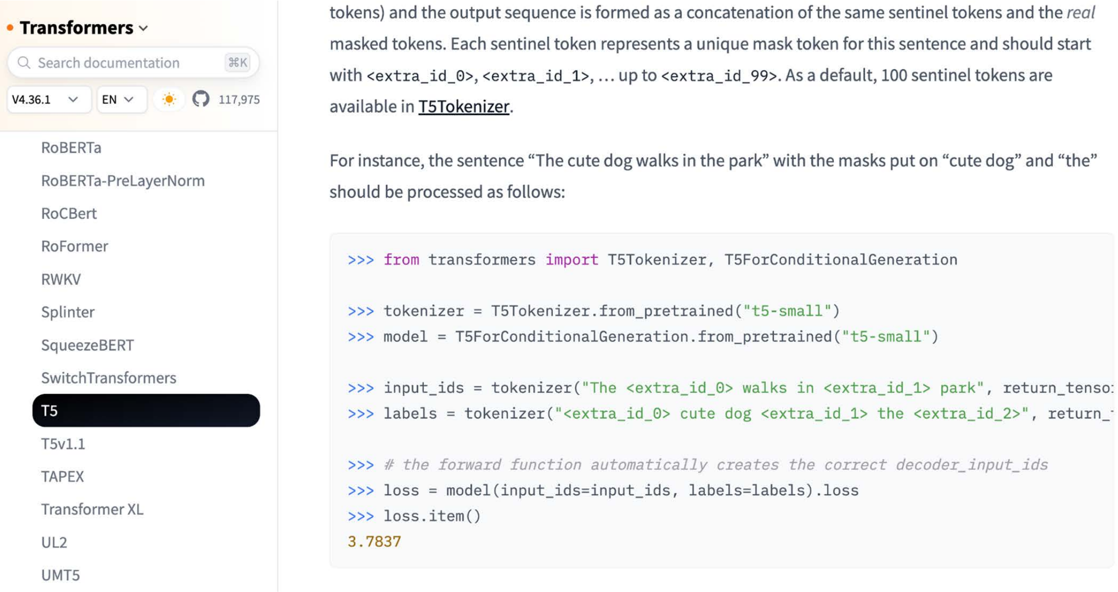
Figure 19.2: Hugging Face Models page listing all text-generation models sorted by number of downloads (in decreasing order)

You might recognize some of these popular models, and we used the gpt2 model from the Hugging Face Hub in *Chapter 7*, *Music and Text Generation with PyTorch*. For the current demonstration, let's use another popular text-generation model called the **Text-to-Text Transfer Transformer** (**T5**) model [8] developed by Google.

We load the `t5-small` model from the Hugging Face Hub with the following lines of code:

```
from transformers import T5Tokenizer, T5ForConditionalGeneration
tokenizer = T5Tokenizer.from_pretrained("t5-small")
model = T5ForConditionalGeneration.from_pretrained("t5-small")
```

If you are wondering how I know which transformers objects (`T5Tokenizer` or `T5ForConditionalGeneration`) to import to load the T5 model, let me tell you that Hugging Face provides usage examples on the T5 model page [7], as shown in *Figure 19.3*. This helps me decide which transformers objects to import.



tokens) and the output sequence is formed as a concatenation of the same sentinel tokens and the *real* masked tokens. Each sentinel token represents a unique mask token for this sentence and should start with `<extra_id_0>`, `<extra_id_1>`, … up to `<extra_id_99>`. As a default, 100 sentinel tokens are available in T5Tokenizer.

For instance, the sentence "The cute dog walks in the park" with the masks put on "cute dog" and "the" should be processed as follows:

```
>>> from transformers import T5Tokenizer, T5ForConditionalGeneration

>>> tokenizer = T5Tokenizer.from_pretrained("t5-small")
>>> model = T5ForConditionalGeneration.from_pretrained("t5-small")

>>> input_ids = tokenizer("The <extra_id_0> walks in <extra_id_1> park", return_tenso
>>> labels = tokenizer("<extra_id_0> cute dog <extra_id_1> the <extra_id_2>", return_

>>> # the forward function automatically creates the correct decoder_input_ids
>>> loss = model(input_ids=input_ids, labels=labels).loss
>>> loss.item()
3.7837
```

Figure 19.3: Hugging Face model page for the T5 model showing a usage example

Now that we have loaded the T5 model, in the below code, we use the `t5-small` model to translate an English sentence to German:

```
from transformers import DistilBertTokenizerFast,
DistilBertForSequenceClassification
input_ids = tokenizer("translate English to German: I love PyTorch.",
                      return_tensors="pt").input_ids
outputs = model.generate(input_ids)
print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

This should produce the following output:

```
Ich liebe PyTorch.
```

There we go; the model seems to work perfectly! The process shown here can easily be adapted for the different models on the Hugging Face Hub for tasks ranging from **Natural Language Processing** (**NLP**) to computer vision and even to multimodal models.

In the next section, we look at another Hugging Face product that can be used interoperably with PyTorch – the Hugging Face **Datasets** library.

Using the Hugging Face Datasets library with PyTorch

Using the Hugging Face `datasets` library with PyTorch enables easy access to thousands of public datasets and simplifies handling custom ones. There are over 144,000 (in May 2024) datasets available on Hugging Face, which can be checked with the following lines of code:

```
from huggingface_hub import hf_api
datasets = hf_api.list_datasets()
len([d for d in datasets])
```

To get started with the Hugging Face `datasets` library, make sure you have installed the following dependencies:

```
pip install torch datasets transformers
```

All code for this section is available on GitHub [9]. First, we should import the required libraries and set up the environment:

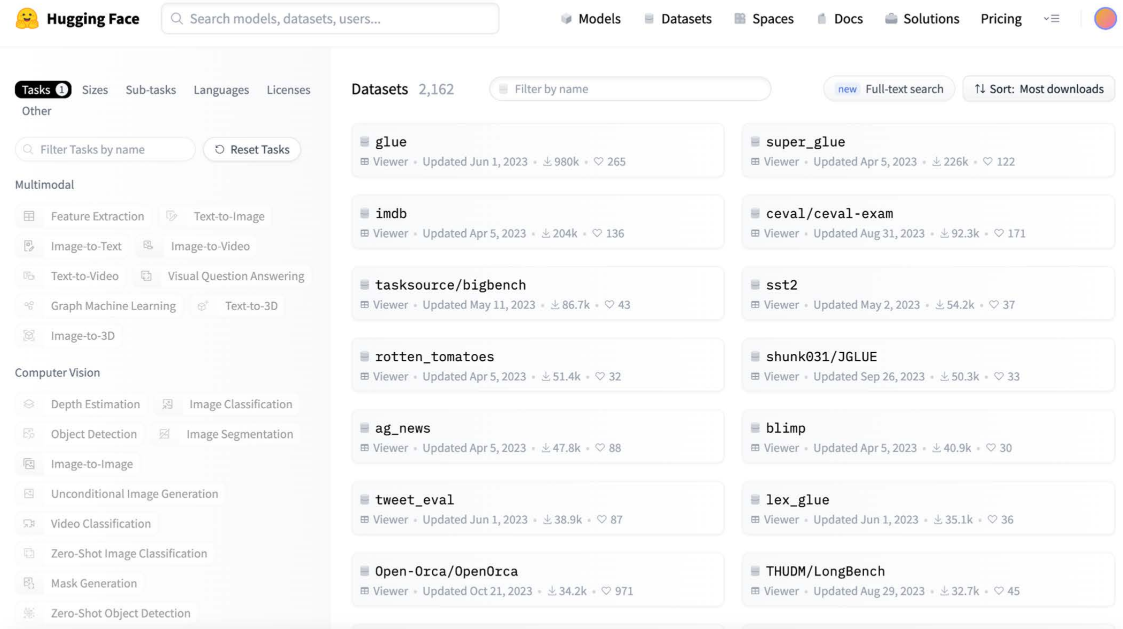
```
import torch
from datasets import load_dataset
from transformers import BertTokenizer
```

We import the `load_dataset` function from the datasets library. We plan to use the BERT model for our demonstration, hence we import the `BertTokenizer` to convert text into tokens.

Next, with just one line of code, we can load a dataset:

```
# Loading a dataset from HuggingFace Datasets library
dataset = load_dataset("rotten_tomatoes")
```

For our demonstration, we use the `rotten_tomatoes` dataset [10] from Hugging Face. Why did we choose the `rotten_tomatoes` dataset? Well, similar to the process of selecting the T5 model in the previous section, we simply look at the Hugging Face `Datasets` page [11] for datasets that belong to a particular task, say, text classification, and then sort the datasets in decreasing order of their downloads to get the most popular datasets, as shown in *Figure 19.4*:



Datasets 2,162

glue
Viewer • Updated Jun 1, 2023 • 980k • 265

super_glue
Viewer • Updated Apr 5, 2023 • 226k • 122

imdb
Viewer • Updated Apr 5, 2023 • 204k • 136

ceval/ceval-exam
Viewer • Updated Aug 31, 2023 • 92.3k • 171

tasksource/bigbench
Viewer • Updated May 11, 2023 • 86.7k • 43

sst2
Viewer • Updated May 2, 2023 • 54.2k • 37

rotten_tomatoes
Viewer • Updated Apr 5, 2023 • 51.4k • 32

shunk031/JGLUE
Viewer • Updated Sep 26, 2023 • 50.3k • 33

ag_news
Viewer • Updated Apr 5, 2023 • 47.8k • 88

blimp
Viewer • Updated Apr 5, 2023 • 40.9k • 30

tweet_eval
Viewer • Updated Jun 1, 2023 • 38.9k • 87

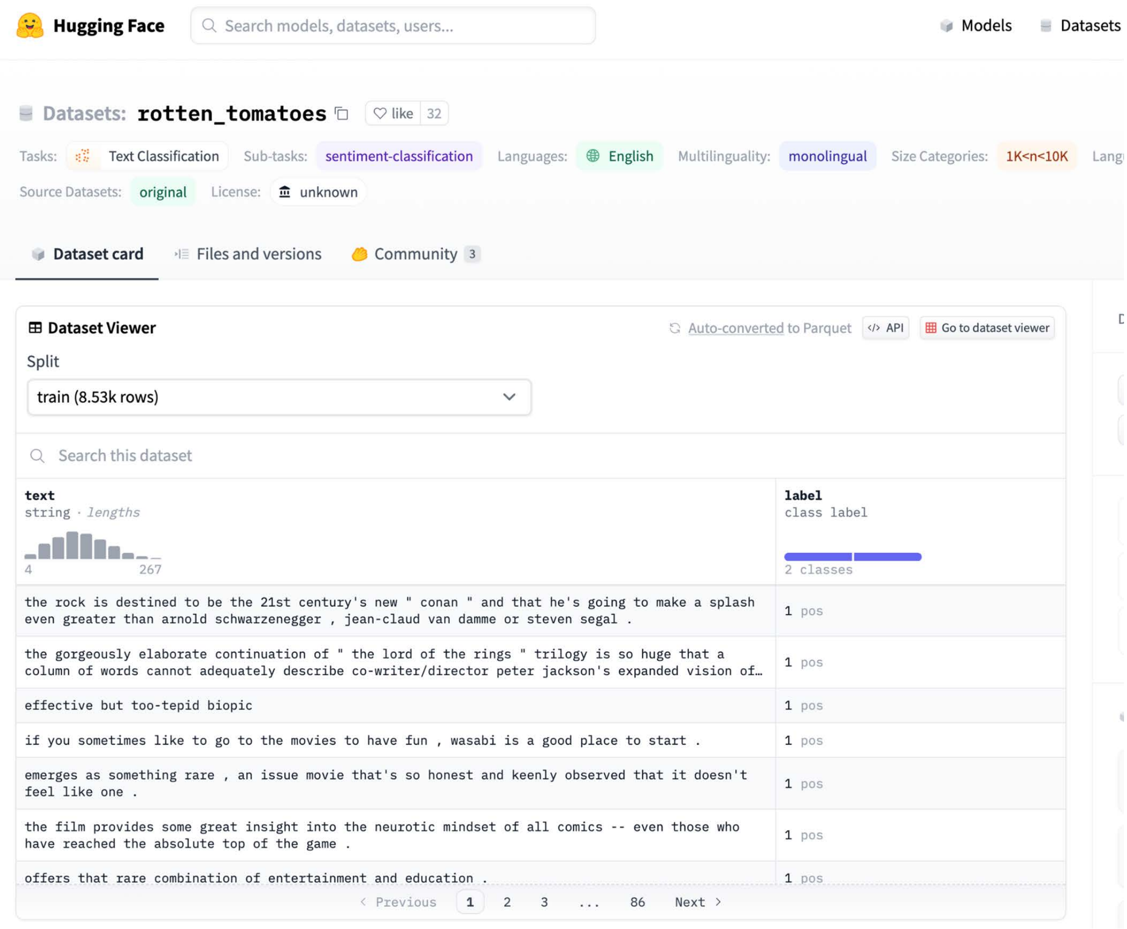
lex_glue
Viewer • Updated Jun 1, 2023 • 35.1k • 36

Open-Orca/OpenOrca
Viewer • Updated Oct 21, 2023 • 34.2k • 971

THUDM/LongBench
Viewer • Updated Aug 29, 2023 • 32.7k • 45

Figure 19.4: Hugging Face datasets page listing all the text-classification datasets sorted by number of downloads (in decreasing order)

From the displayed popular datasets, we select the `rotten_tomatoes` dataset, which is a dataset of movie reviews labeled with positive or negative sentiments (classes 1 and 0 respectively), as shown in *Figure 19.5*:



Datasets: rotten_tomatoes

Tasks: Text Classification Sub-tasks: sentiment-classification Languages: English Multilinguality: monolingual Size Categories: 1K<n<10K Lang

Source Datasets: original License: unknown

Split

train (8.53k rows)

text string · lengths 4 267	label class label 2 classes
the rock is destined to be the 21st century's new " conan " and that he's going to make a splash even greater than arnold schwarzenegger , jean-claud van damme or steven segal .	1 pos
the gorgeously elaborate continuation of " the lord of the rings " trilogy is so huge that a column of words cannot adequately describe co-writer/director peter jackson's expanded vision of…	1 pos
effective but too-tepid biopic	1 pos
if you sometimes like to go to the movies to have fun , wasabi is a good place to start .	1 pos
emerges as something rare , an issue movie that's so honest and keenly observed that it doesn't feel like one .	1 pos
the film provides some great insight into the neurotic mindset of all comics -- even those who have reached the absolute top of the game .	1 pos
offers that rare combination of entertainment and education .	1 pos

Figure 19.5: Hugging Face dataset page for the rotten_tomatoes dataset

Having loaded the dataset, we need to tokenize the text to use it inside the BERT model. We do so with the following lines of code:

```
# Initializing a tokenizer
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")

# Tokenizing and preparing the dataset for PyTorch
def tokenize_function(example):
    # Tokenizes the text, applies padding/truncation, and returns tensors
    # including the attention mask.
    # return tokenizer(example["text"], padding="max_length", truncation=True)
```

```
tokenized_dataset = dataset.map(tokenize_function, batched=True)
# The attention mask helps the model distinguish between
# actual data and padding.
tokenized_dataset.set_format(type='torch', columns=['input_ids', 'attention_
mask', 'label'])
```

The `tokenize_dataset` function is applied to each text sample (movie review) of the dataset using the `map` method of the `dataset` object. This tokenized dataset is then converted to PyTorch format. Next, we create training and evaluation dataloaders using the tokenized dataset with the following lines of code:

```
# Creating PyTorch DataLoader
train_dataloader = torch.utils.data.DataLoader(
    tokenized_dataset['train'], batch_size=8, shuffle=True)
eval_dataloader = torch.utils.data.DataLoader(
    tokenized_dataset['test'], batch_size=8)
```

Now that the data is ready, we load the pre-trained BERT model (`bert-base-uncased`) from the Hugging Face Hub and initialize an `AdamW` optimizer to fine-tune the BERT model. `AdamW` modifies the Adam optimizer (discussed in *Chapter 1*, *Overview of Deep Learning Using PyTorch*) by decoupling the weight decay from the gradient updates, leading to more effective regularization and often better generalization in deep learning models:

```
from transformers import BertForSequenceClassification, AdamW

# Load pre-trained BERT model for sequence classification
model = BertForSequenceClassification.from_pretrained("bert-base-uncased")

# Optimizer and learning rate scheduler setup
optimizer = AdamW(model.parameters(), lr=5e-5)
```

Finally, we can start fine-tuning the pre-trained BERT model with the following PyTorch training and validation code:

```
# Training loop using PyTorch
for epoch in range(3):  # Train for 3 epochs as an example
    model.train()
    for batch in tqdm(train_dataloader):
        optimizer.zero_grad()
        input_ids = batch['input_ids']
        attention_mask = batch['attention_mask']
        labels = batch['label']
        outputs = model(input_ids=input_ids,
                        attention_mask=attention_mask,
                        labels=labels)
        loss = outputs.loss
        loss.backward()
        optimizer.step()
    # Evaluation loop
    model.eval()
    ...
    for batch in tqdm(eval_dataloader):
        with torch.no_grad():
            ...
            total_correct += (predictions == labels).sum().item()
            total_samples += len(labels)
    accuracy = total_correct / total_samples
    print(f"Epoch {epoch + 1} - Evaluation Accuracy: {accuracy}")
```

This should fine-tune the BERT model on the text classification task of classifying movie reviews into positive and negative sentiments using the `rotten_tomatoes` dataset, producing an output similar to the following:

```
Epoch 1 - Evaluation Accuracy: 0.800187617260788
Epoch 2 - Evaluation Accuracy: 0.8292682926829268
Epoch 3 - Evaluation Accuracy: 0.8461538461538461
```

In conclusion, the Hugging Face `datasets` library provides convenient functionality that allows you to work with numerous publicly accessible datasets right inside your PyTorch projects. Using the `datasets` library can save time and resources, allowing us to concentrate on building robust machine learning models. In the next section, we will look into how Hugging Face can optimize training PyTorch models through the `accelerate` library.

Using Accelerate to speed up PyTorch model training

Accelerate is a powerful tool developed by Hugging Face that's designed to manage distributed training across multiple CPUs, GPUs, and TPUs, or even cloud services such as **Amazon Web Services** (**AWS**), **Google Cloud Platform** (**GCP**), and Microsoft Azure. It abstracts data and model parallelism and efficiently distributes computations across multiple CPUs, GPUs, or TPUs, reducing overhead and streamlining execution, thereby making scaling effortless. The best part is that you need to add just **five** lines of `accelerate` code into the existing PyTorch code to optimally utilize the hardware, as shown in *Figure 19.6*:

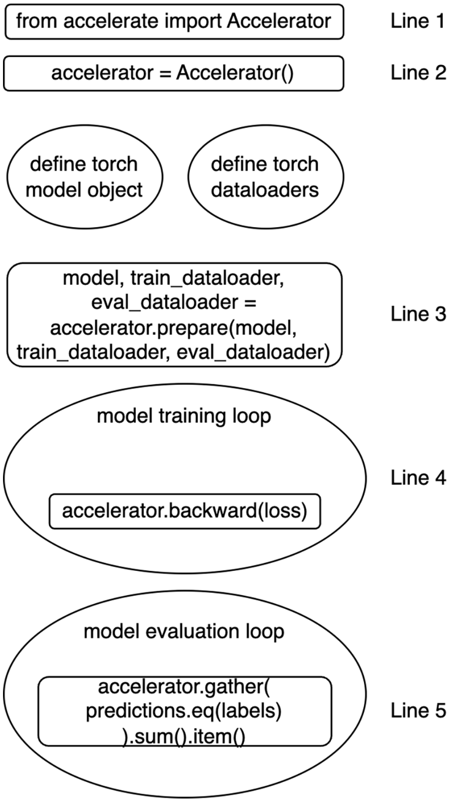

from accelerate import Accelerator
Line 1
accelerator = Accelerator()
Line 2
define torch model object
define torch dataloaders
model, train_dataloader, eval_dataloader = accelerator.prepare(model, train_dataloader, eval_dataloader)
Line 3
model training loop
accelerator.backward(loss)
Line 4
model evaluation loop
accelerator.gather(predictions.eq(labels)).sum().item()
Line 5

Figure 19.6: Schematic representation of accelerating PyTorch model training code with just five lines of accelerate code

To illustrate the usage of `accelerate`, we will continue the previous example of fine-tuning a BERT model for text classification and use accelerate inside the training code to optimize the training process. All code for this section is available on GitHub [12]. Before proceeding, ensure you have the latest version of `accelerate` installed:

```
pip install accelerate
```

The first two lines of code we need to add to our PyTorch code are simply importing the `accelerate` library and instantiating the `accelerate` object, as shown below:

```
from accelerate import Accelerator
# Initialize the accelerator
accelerator = Accelerator(cpu=False, mixed_precision="fp16")
```

While we feed only a couple of arguments into the `accelerator` object instantiation, you can find the full list of arguments on the Hugging Face website [13]. The next additional line of code comes right after instantiating the model object and the dataloader objects:

```
# Move model to the device managed by Accelerator
model, train_dataloader, eval_dataloader = accelerator.prepare(
    model, train_dataloader, eval_dataloader)
```

This line of code ensures that the model and the data are placed on the available hardware, such as GPUs or TPUs. The fourth line of code is a replacement rather than an addition. Originally, the following line of code inside the model training loop performs backpropagation:

```
loss.backward()
```

We need to replace this with the following:

```
accelerator.backward(loss)
```

This is done to manage gradients in distributed training across multiple GPUs, as the accelerator efficiently handles gradient accumulation and synchronization across devices, optimizing the backward pass in a multi-device setup.

The fifth and final line of code is another replacement, this time within the evaluation part of the model training loop. In the original training code, we calculate the total number of correct predictions on the test set with the following line of code:

```
total_correct += (predictions == labels).sum().item()
```

We need to replace it with this:

```
total_correct += accelerator.gather(predictions.eq(labels)).sum().item()
```

Again, this is needed because the `gather` method aggregates the correct predictions across distributed computations, ensuring the accurate calculation of the total correct predictions across multiple devices during training in a distributed setup.

Note

The `predictions == labels` operation can be slow in distributed settings as it involves processing and synchronizing data across multiple devices, whereas `accelerator.gather(predictions.eq(labels))` optimizes this by reducing cross-device communication and efficiently aggregating results.

This concludes our brief overview of the accelerate library and how it can be easily plugged into PyTorch code to make it more hardware efficient. Hugging Face's page on the `accelerate` library [14] is a perfect place to dig deeper into this subject to get the best out of your hardware while running PyTorch.

In the next and final section of this chapter, we will look at yet another Hugging Face product, **Optimum**, which enables us to get the most out of our hardware when it comes to running inference on PyTorch models.

Using Optimum to optimize PyTorch model deployment

A crucial aspect of the machine learning life cycle is model deployment. Hugging Face's **Optimum** aims to reduce the complexity involved in deploying AI models across diverse platforms, languages, frameworks, and devices. As the name indicates, Optimum also helps optimize the model before deployment.

In this section, we will take a pre-trained model (trained using PyTorch) from the Hugging Face Hub, and convert that PyTorch model into an **Open Neural Network Exchange** (**ONNX**) model to use it for inference with ONNX Runtime, as shown in *Figure 19.7*.

> **Note**
>
> ONNX Runtime is an open-source, high-performance inference engine developed by Microsoft, designed to efficiently execute models that are compliant with the ONNX format across various hardware platforms, such as Intel CPUs, NVIDIA GPUs, Jetson Nano, Android phones, and so on.

We discussed ONNX in *Chapter 13*, *Operationalizing PyTorch Models into Production*. Converting a PyTorch model into ONNX allows cross-platform, framework-independent deployment, optimizing performance across different hardware accelerators using libraries such as Optimum.

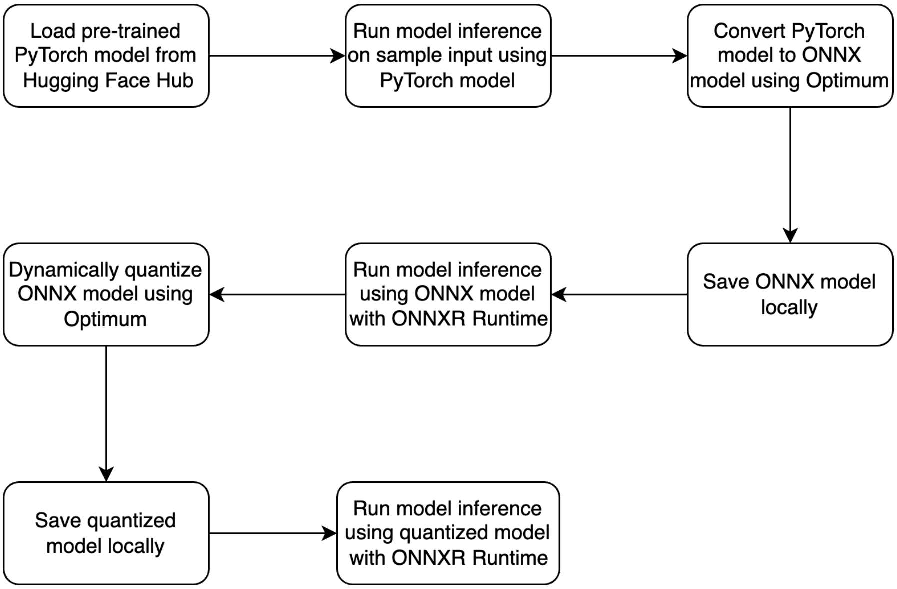

Load pre-trained PyTorch model from Hugging Face Hub
Run model inference on sample input using PyTorch model
Convert PyTorch model to ONNX model using Optimum
Save ONNX model locally
Run model inference using ONNX model with ONNXR Runtime
Dynamically quantize ONNX model using Optimum
Save quantized model locally
Run model inference using quantized model with ONNXR Runtime

Figure 19.7: Schematic representation of accelerating PyTorch model training code with just five lines of accelerate code

We will also learn about quantizing the ONNX model using Optimum to reduce the model size at the expense of slightly reduced accuracy. At each step, we run model inference on a fixed sample input to ensure that the optimizations performed on the model do not alter the model's output on a given fixed input.

All code for this section is available on GitHub [15].

Note

Quantization is the process of reducing the memory footprint and computational complexity of a neural network model by reducing the precision of its parameters and activations while aiming to maintain the model's performance and inference accuracy.

Before we delve into the code, we need to install the following packages:

```
pip install onnx onnxruntime optimum
```

First, we need to import the libraries required to load the pre-trained `bert-base-uncased` text-classification model in PyTorch format, and the library to convert the BERT model into an ONNX model, `ORTModelForSequenceClassification`:

```
from transformers import AutoModelForSequenceClassification, AutoTokenizer
from optimum.onnxruntime import ORTModelForSequenceClassification
```

Next, we define the model name that we want to download from the Hugging Face Hub and name the directory where we wish to store the ONNX model artifacts:

```
# a pre-trained model from HuggingFace
model_name = "bert-base-uncased"
onnx_directory = "bert-base-uncased_onnx"
```

We load the pre-trained BERT model in PyTorch format from the Hugging Face Hub with the following lines of code:

```
# Load the pre-trained model and tokenizer
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased", export=True)
model = AutoModelForSequenceClassification.from_pretrained(model_name)
```

Note that we have added the `export=True` argument to the tokenizer initialization statement as we intend to save the tokenizer loaded from the Hugging Face Hub onto our local storage. Let's now check if the loaded model works on a sample input with the following lines of code:

```
input_ids = tokenizer("I love PyTorch!", return_tensors="pt")
model(**input_ids)
```

This should produce an output that looks like the following:

```
SequenceClassifierOutput(loss=None, logits=tensor([[-0.0403, 0.0209]], grad_
fn=<AddmmBackward0>), hidden_states=None, attentions=None)
```

Next, we convert the loaded PyTorch model into an ONNX model:

```
model_onnx = ORTModelForSequenceClassification.from_pretrained(
    "bert-base-uncased", export=True)
```

Notice that we are converting the model from PyTorch format to ONNX format in just one line of code, using the `ORTModelForSequenceClassification` class imported from the `optimum.onnxruntime` library. In contrast, in *Chapter 13*, *Operationalizing PyTorch Models into Production*, we wrote several lines of code to convert the PyTorch model to ONNX format.

Now that we have the ONNX model, let's check if the model works on the same sample input with the following line of code:

```
model_onnx(**input_ids)
```

This should produce an output that looks like the following:

```
SequenceClassifierOutput(loss=None, logits=tensor([[-0.0403,  0.0272]]),
hidden_states=None, attentions=None)
```

As we can see, the ONNX model is also functioning as expected. We now save the ONNX model into the defined ONNX model directory with the following lines of code:

```
model_onnx.save_pretrained(onnx_directory)
tokenizer.save_pretrained(onnx_directory)
```

It is worth checking the contents of the ONNX model folder with the following command on your terminal:

```
du -sh bert-base-uncased_onnx/*
```

This should produce the following output:

```
4.0K bert-base-uncased_onnx/config.json
418M bert-base-uncased_onnx/model.onnx
4.0K bert-base-uncased_onnx/ort_config.json
4.0K bert-base-uncased_onnx/special_tokens_map.json
696K bert-base-uncased_onnx/tokenizer.json
4.0K bert-base-uncased_onnx/tokenizer_config.json
228K bert-base-uncased_onnx/vocab.txt
```

As we can see the ONNX model is around 400 MB in size. Can we somehow make it smaller? The answer is yes. Thanks to the `optimum` library, we can convert the ONNX model to a quantized ONNX model, which is much smaller and also faster. We need to import the following dependencies to perform quantization:

```
from optimum.onnxruntime.configuration import AutoQuantizationConfig
from optimum.onnxruntime import ORTQuantizer
```

The following lines of code quantize the ONNX model:

```
qconfig = AutoQuantizationConfig.arm64(is_static=False, per_channel=False)
quantizer = ORTQuantizer.from_pretrained(model_onnx)
quantizer.quantize(save_dir=onnx_directory, quantization_config=qconfig)
```

First, we specify some configurations or settings related to the quantization process. Hugging Face's quantization page [16] is an excellent reference to learn more about the various configuration options. In our setting, we set `is_static` to `False`, as we are performing dynamic quantization.

Note

Dynamic quantization involves quantizing the weights and activations of a neural network during inference without retraining the model, optimizing memory and inference speed by dynamically adjusting the precision of tensors based on the observed input ranges. Static quantization, on the other hand, quantizes the model's weights and activations before inference, typically during or after training, fixing the precision levels beforehand to achieve memory and computational efficiency.

The above lines of code save the quantized model inside the same ONNX model directory where we have previously saved the ONNX model. Let's check the contents of that folder once again with the following terminal command:

```
du -sh bert-base-uncased_onnx/*
```

This should now produce the following output:

```
4.0K bert-base-uncased_onnx2/config.json
418M bert-base-uncased_onnx2/model.onnx
106M bert-base-uncased_onnx2/model_quantized.onnx
4.0K bert-base-uncased_onnx2/ort_config.json
4.0K bert-base-uncased_onnx2/special_tokens_map.json
696K bert-base-uncased_onnx2/tokenizer.json
4.0K bert-base-uncased_onnx2/tokenizer_config.json
228K bert-base-uncased_onnx2/vocab.txt
```

As we can see, we have the `model_quantized.onnx` file, which is roughly 100 MB in size, a quarter of the original ONNX model. Dynamic quantization from Optimum has therefore reduced model size by 4x. Let's load the quantized model into a new ONNX model object with the following line of code:

```
model_quantized = ORTModelForSequenceClassification.from_pretrained(
    onnx_directory, file_name="model.onnx")
```

Finally, we check whether the quantized model works on the sample input, with the following lines of code:

```
model_quantized(**input_ids)
```

This should produce an output that looks like the following:

```
SequenceClassifierOutput(loss=None, logits=tensor([[-0.0403,  0.0272]]),
hidden_states=None, attentions=None)
```

Voilà! We get the same output from a 4x smaller model as we did from the ONNX model. This brings us to the end of exploring the `optimum` library. Please refer to Hugging Face's Optimum page [17] for a detailed understanding of this product. When combined with Accelerate for distributed training, Optimum becomes a valuable ally in streamlining machine learning life cycle management, enabling rapid prototyping, experimentation, and delivery of high-quality products powered by cutting-edge research.

Hugging Face is a rapidly growing and evolving platform, and it is an important pillar of the new AI story that we are witnessing in the post-transformers era. To succeed in AI, it is extremely advisable to stay in touch with the latest Hugging Face updates, especially the ones related to PyTorch. While this chapter should help you get started with using Hugging Face in your PyTorch or other deep learning projects, the Hugging Face website [1] is the best resource if you want to dive deeper into the topics, datasets, models, or AI demos of your choice!

Summary

In this chapter, we first discussed the different relevant Hugging Face components for PyTorch users. We then established how to use the transformers library (the most important Hugging Face library) together with PyTorch. Next, we learned about the Hugging Face Hub, which provides a wide range of over 650,000 pre-trained models, and used the Hub to load the BERT model for inference. We then explored the Hugging Face datasets library, which gives us access to over 144,000 datasets. We learned how to use it with PyTorch through an example of fine-tuning a pre-trained model.

Next, we learned about the accelerate library from Hugging Face, and how it can be used to speed up PyTorch training code with just five lines of code changes. We then explored the Optimum library from Hugging Face and used it to convert a PyTorch model to an ONNX model. We used the ONNX model for inference using ONNX Runtime. Finally, we used Optimum to quantize the ONNX model into a 4x smaller model and used that model for inference with ONNX Runtime. This brings us to the end of this book. Thank you so much for reading. I hope the book helps you become better at using PyTorch. And, I wish you all the best!

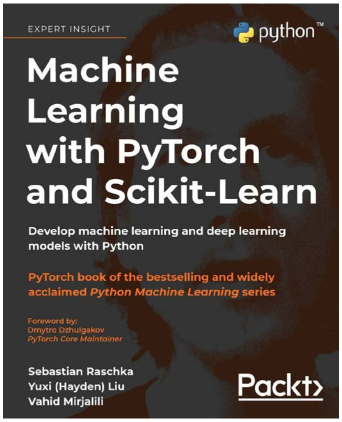
Reference list

1. Hugging Face: `https://huggingface.co/`
2. Hugging Face documentation: `https://huggingface.co/docs`
3. GitHub 1: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter19/HuggingFacePyTorch.ipynb`
4. Hugging Face `bert-base-uncased`: `https://huggingface.co/google-bert/bert-base-uncased`
5. Hugging Face Hub: `https://huggingface.co/docs/hub/index`
6. GitHub 2: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter19/HuggingFaceHub.ipynb`
7. Hugging Face Models page: `https://huggingface.co/models?other=text-generation-inference&sort=downloads`
8. Hugging Face T5: `https://huggingface.co/docs/transformers/model_doc/t5`
9. GitHub 3: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter19/HuggingFaceDatasets.ipynb`
10. Hugging Face `rotten_tomatoes`: `https://huggingface.co/datasets/rotten_tomatoes`
11. Hugging Face Datasets: `https://huggingface.co/datasets?task_categories=task_categories:text-classification&sort=downloads`
12. GitHub 4: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter19/HuggingFaceAccelerate.ipynb`
13. Hugging Face Accelerator: `https://huggingface.co/docs/accelerate/package_reference/accelerator#accelerate.Accelerator`
14. Hugging Face Accelerate library: `https://huggingface.co/docs/accelerate/index`
15. GitHub 5: `https://github.com/arj7192/MasteringPyTorchV2/blob/main/Chapter19/HuggingFaceOptimum.ipynb`
16. Hugging Face quantization: `https://huggingface.co/docs/optimum/onnxruntime/usage_guides/quantization`
17. Hugging Face Optimum page: `https://huggingface.co/docs/optimum/index`

Leave a review!

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



EXPERT INSIGHT

python™

Machine Learning with PyTorch and Scikit-Learn

Develop machine learning and deep learning models with Python

PyTorch book of the bestselling and widely acclaimed *Python Machine Learning* series

Foreword by:
Dmytro Dzhulgakov
PyTorch Core Maintainer

Sebastian Raschka
Yuxi (Hayden) Liu
Vahid Mirjalili

Packt>

Machine Learning with PyTorch and Scikit-Learn

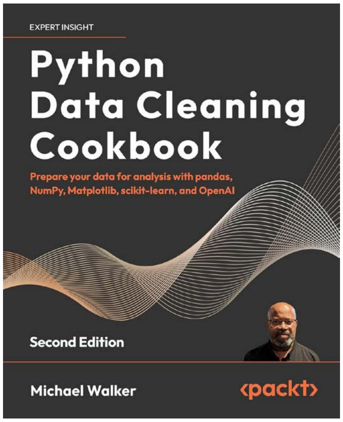
Sebastian Raschka

Yuxi (Hayden) Liu

Vahid Mirjalili

ISBN: 978-1-80181-931-2

- Explore frameworks, models, and techniques for machines to 'learn' from data
- Use scikit-learn for machine learning and PyTorch for deep learning
- Train machine learning classifiers on images, text, and more
- Build and train neural networks, transformers, and boosting algorithms
- Discover best practices for evaluating and tuning models
- Predict continuous target outcomes using regression analysis
- Dig deeper into textual and social media data using sentiment analysis


EXPERT INSIGHT
Python Data Cleaning Cookbook
Prepare your data for analysis with pandas, NumPy, Matplotlib, scikit-learn, and OpenAI
Second Edition
Michael Walker
‹packt›

Python Data Cleaning Cookbook – Second Edition

Michael Walker

ISBN: 978-1-80323-987-3

- Using OpenAI tools for various data cleaning tasks
- Producing summaries of the attributes of datasets, columns, and rows
- Anticipating data-cleaning issues when importing tabular data into pandas
- Applying validation techniques for imported tabular data
- Improving your productivity in pandas by using method chaining
- Recognizing and resolving common issues like dates and IDs
- Setting up indexes to streamline data issue identification
- Using data cleaning to prepare your data for ML and AI models

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Mastering Pytorch, Second Edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

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

Index

Download a free PDF copy of this book