EXPERT INSIGHT

Mastering PyTorch

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

Second Edition

Ashish Ranjan Jha

packt

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

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`

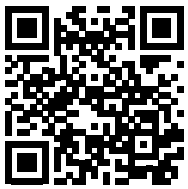

Table of Contents

Chapter 6: Graph Neural Networks 167

Chapter 7: Music and Text Generation with PyTorch 203

Chapter 12: Model Training Optimizations 325

Chapter 13: Operationalizing PyTorch Models into Production 347

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.)

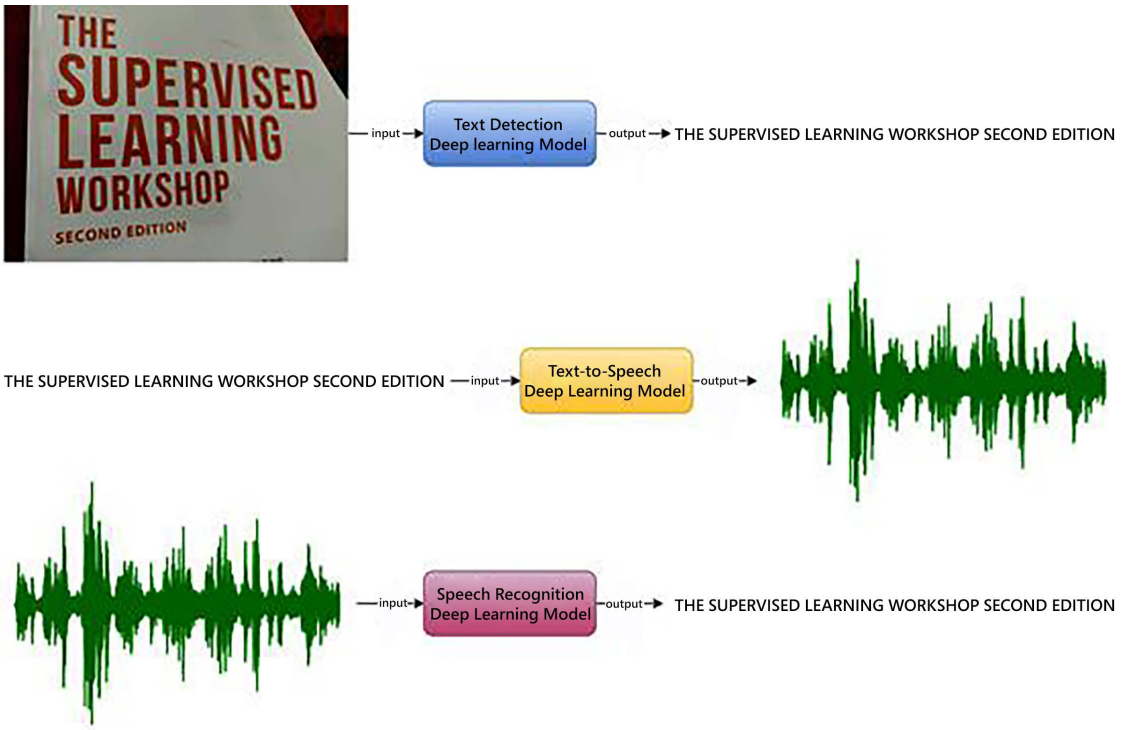

Figure 1.1: Deep learning model examples

Deep neural networks involve a lot of mathematical computations, linear algebraic equations, non-linear 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.

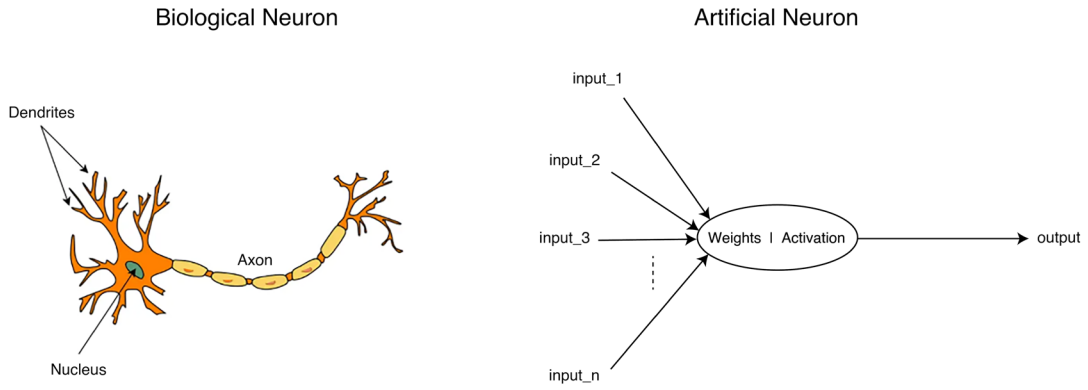

Figure 1.2: Artificial neuron inspired by biological neuron. (Biological neuron image by: https://pixabay.com/users/clker-free-vector-images-3736)

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:

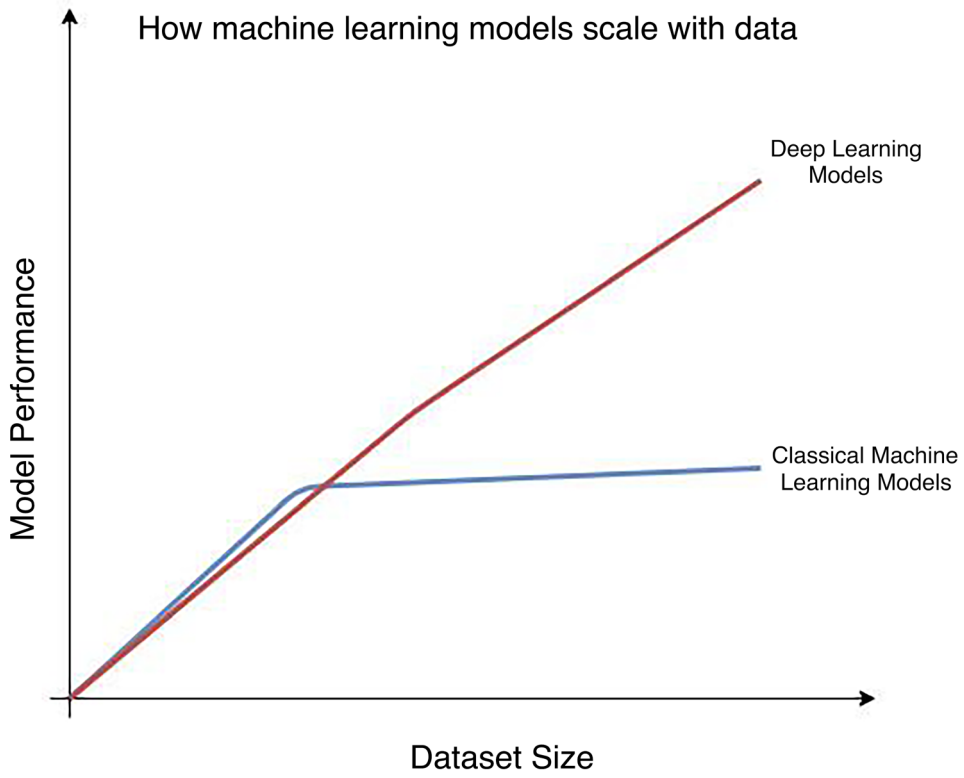

Figure 1.3: Model performance versus dataset size

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:

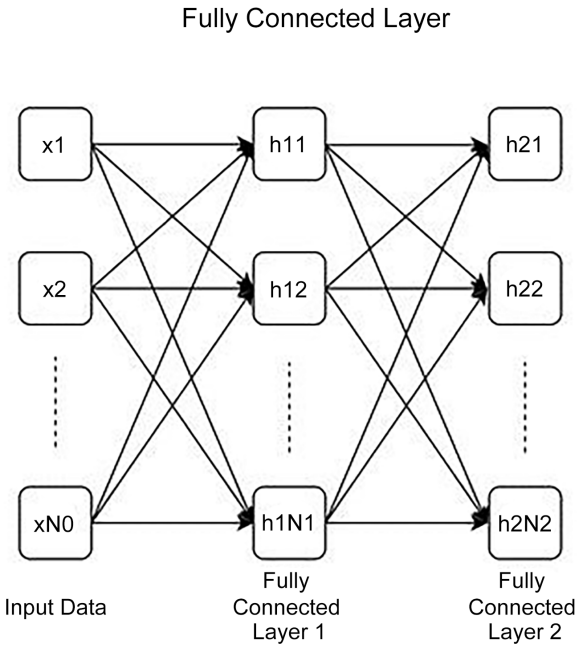

Figure 1.4: Fully connected layer

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:

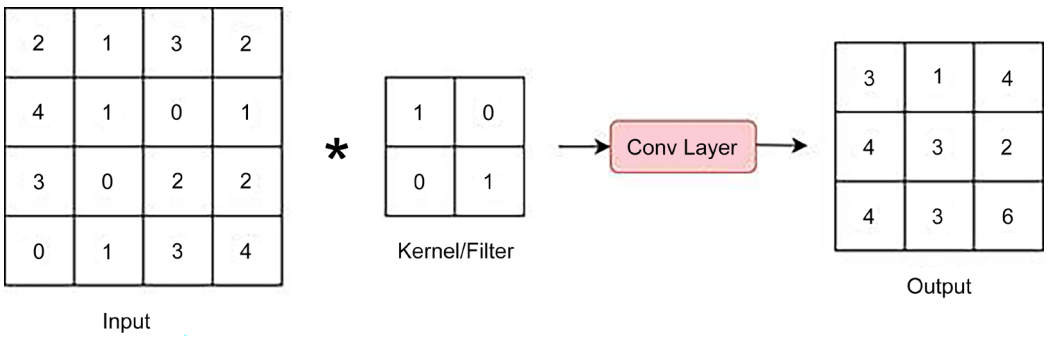

Figure 1.5: Convolutional layer

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):

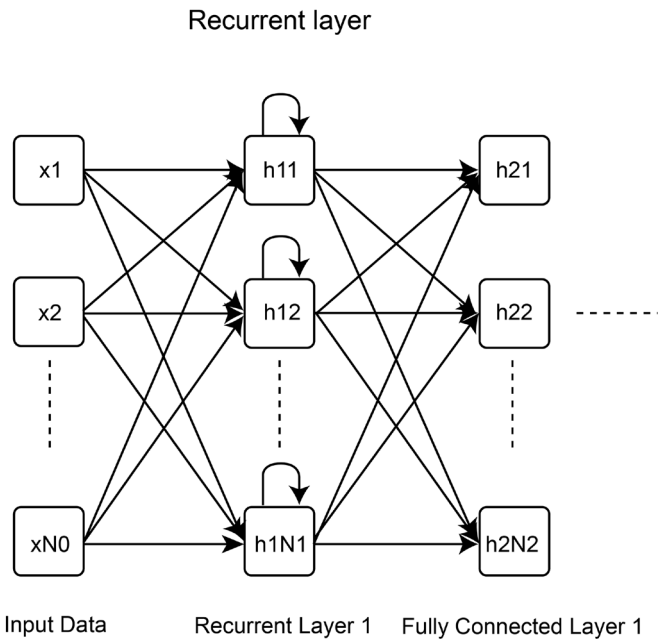

Figure 1.6: Recurrent layer

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:

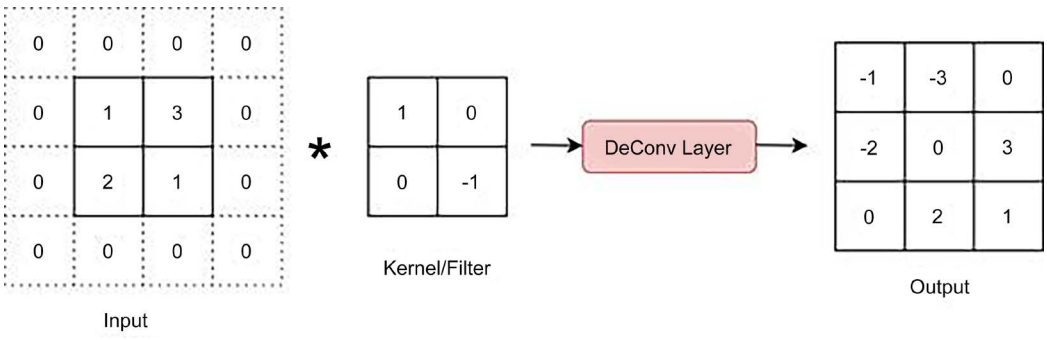

Figure 1.7: DeConvolutional Layer

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:

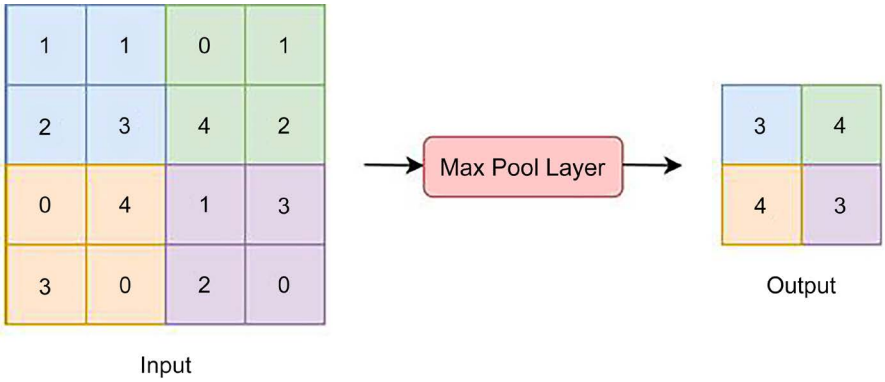

Figure 1.8: Pooling layer

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:

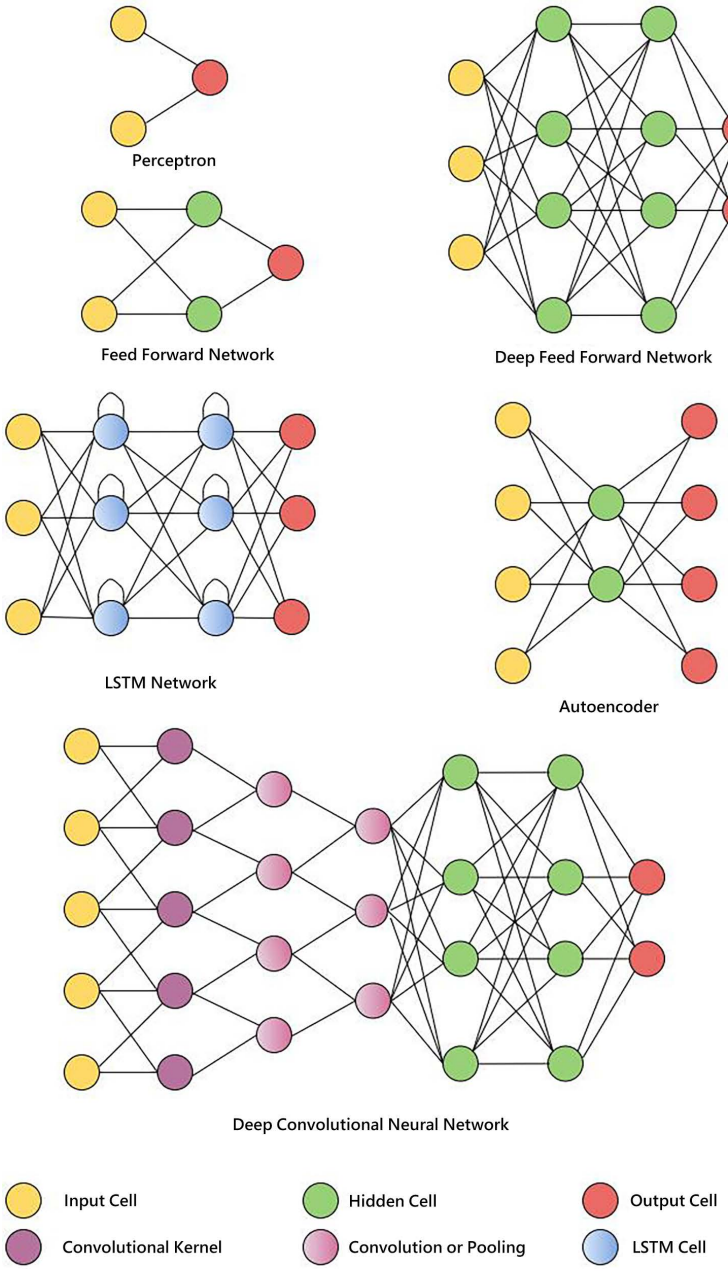

Figure 1.9: Different neural network architectures

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:

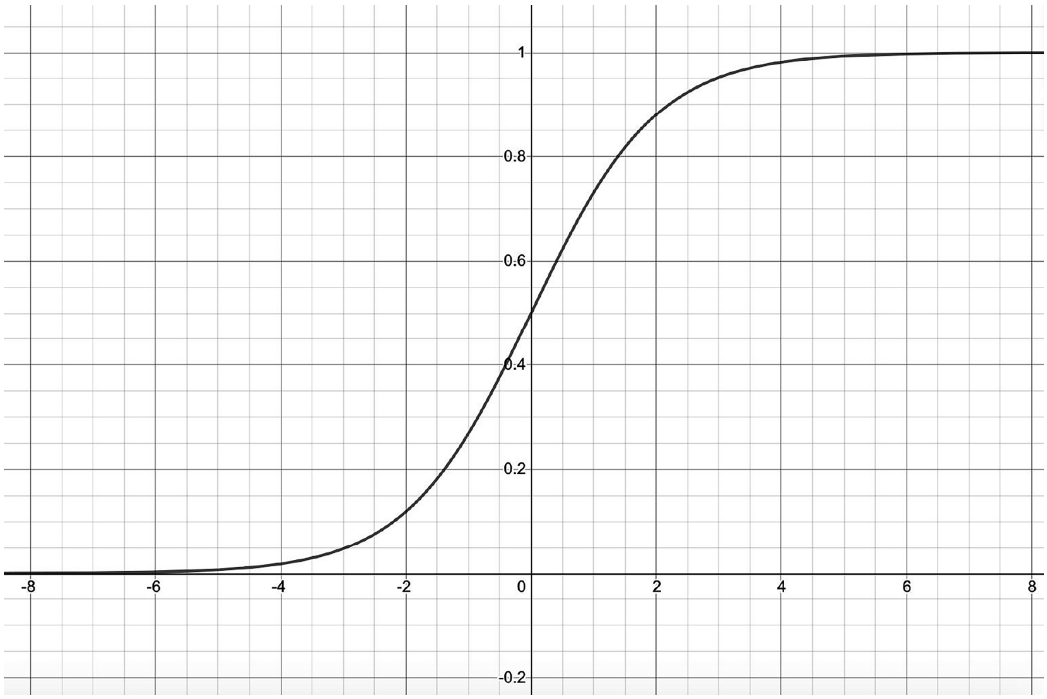

Figure 1.10: Sigmoid function

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:

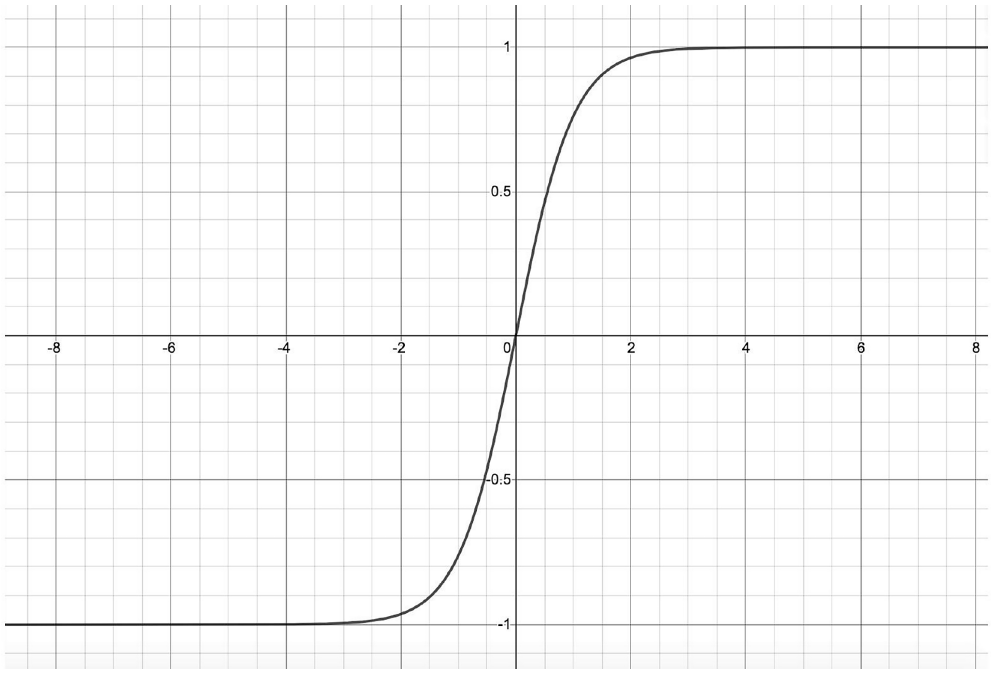

Figure 1.11: TanH function

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:

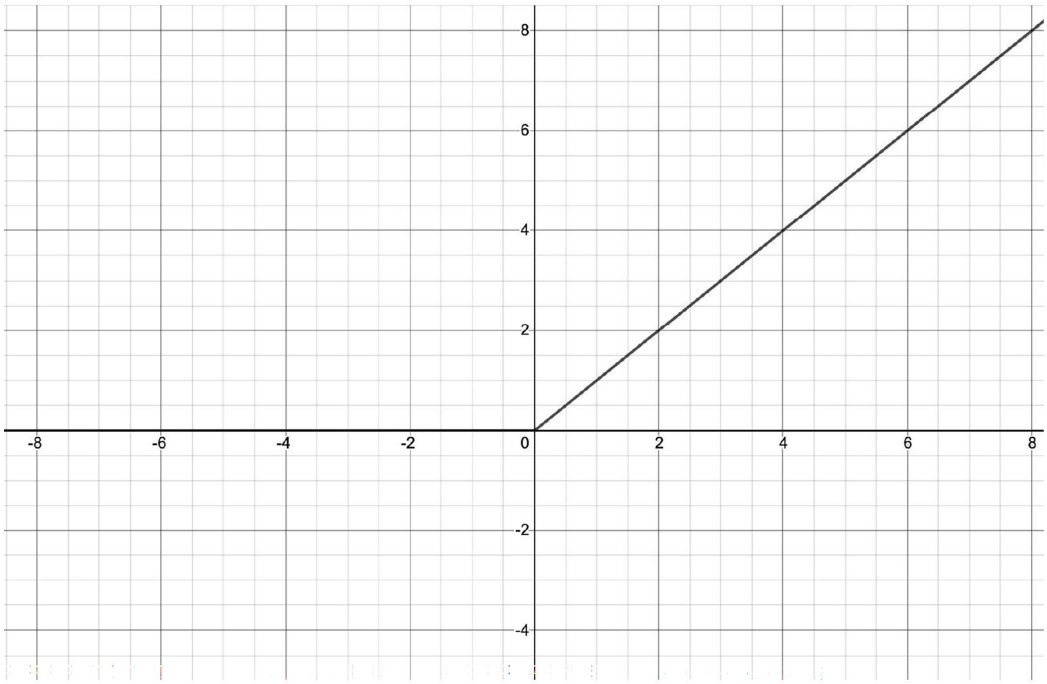

Figure 1.12: ReLU function

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:

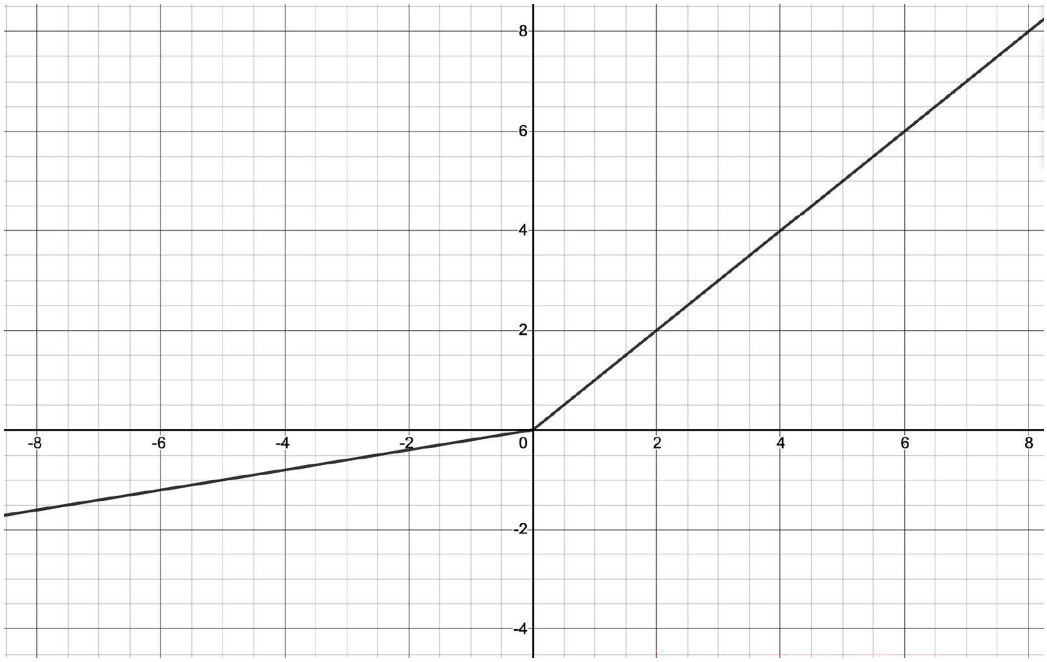

Figure 1.13: Leaky ReLU function

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:

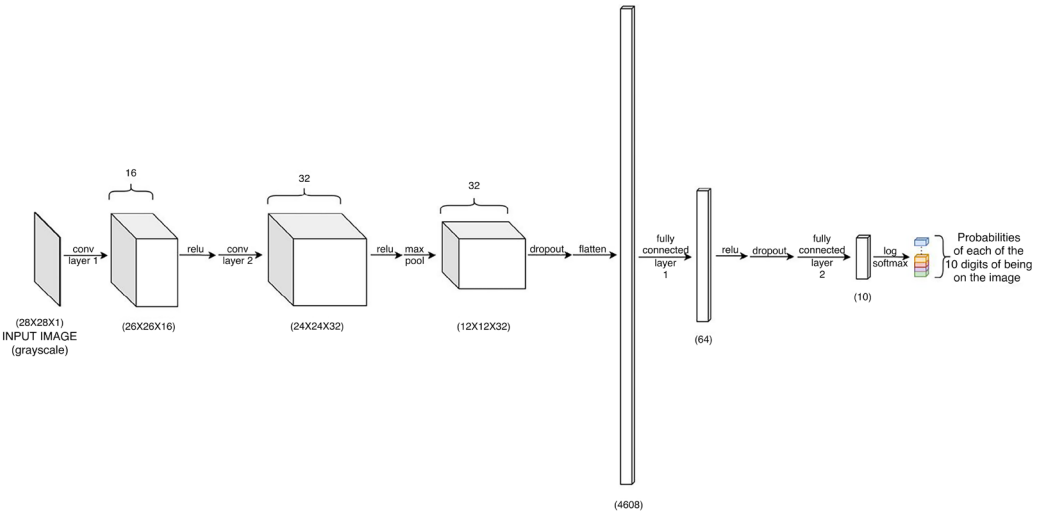

Figure 1.14: Neural network architecture

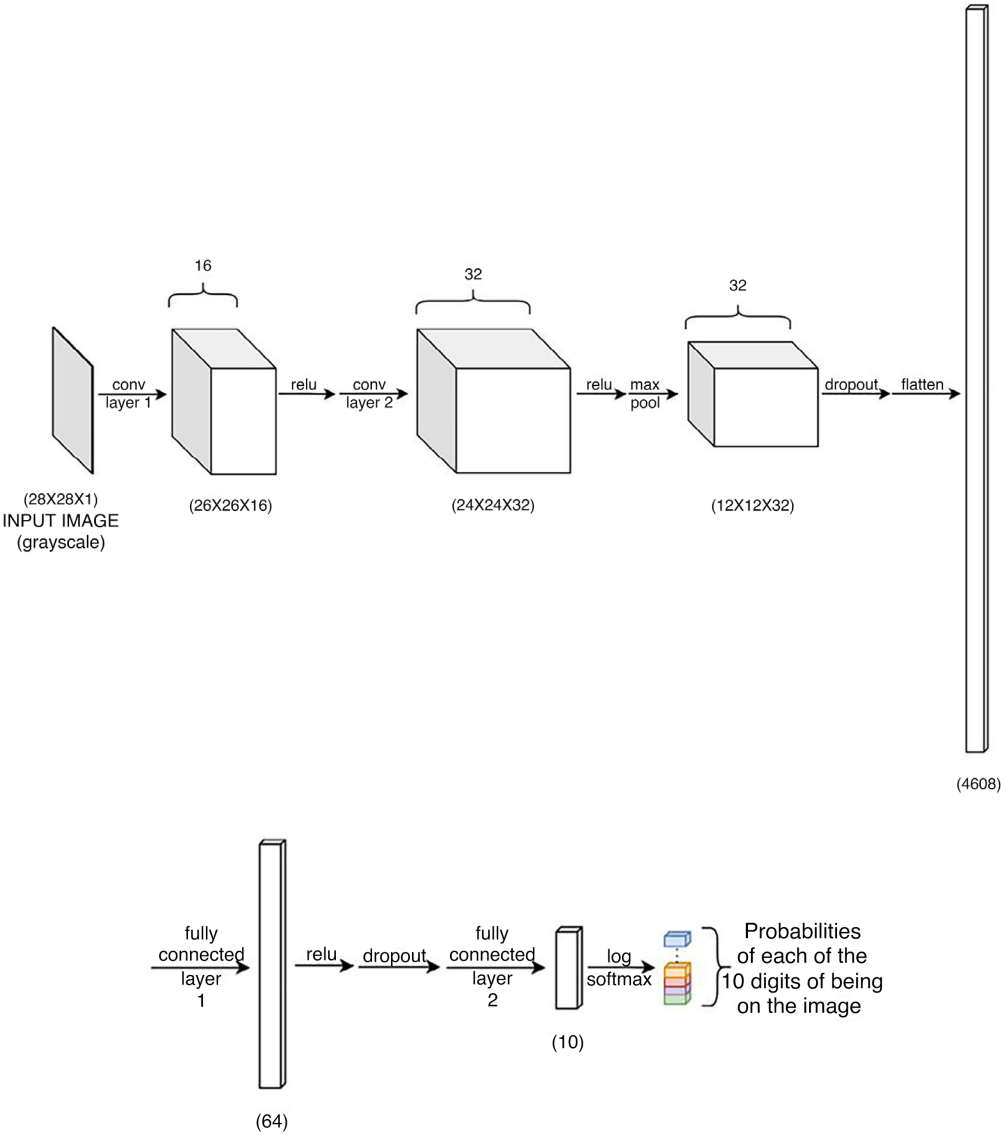

Figure 1.14: Neural network architecture

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:

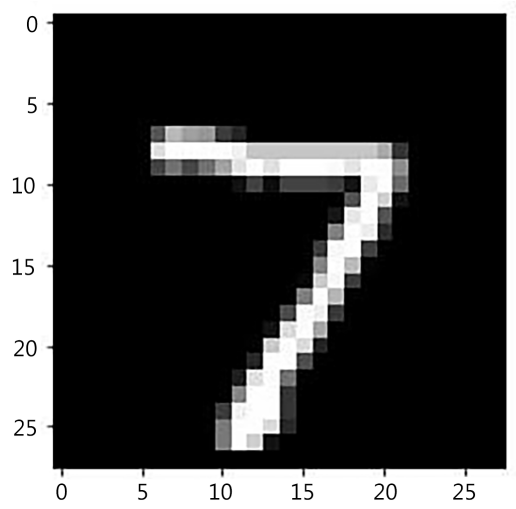

Figure 1.15: Sample handwritten image

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]`.