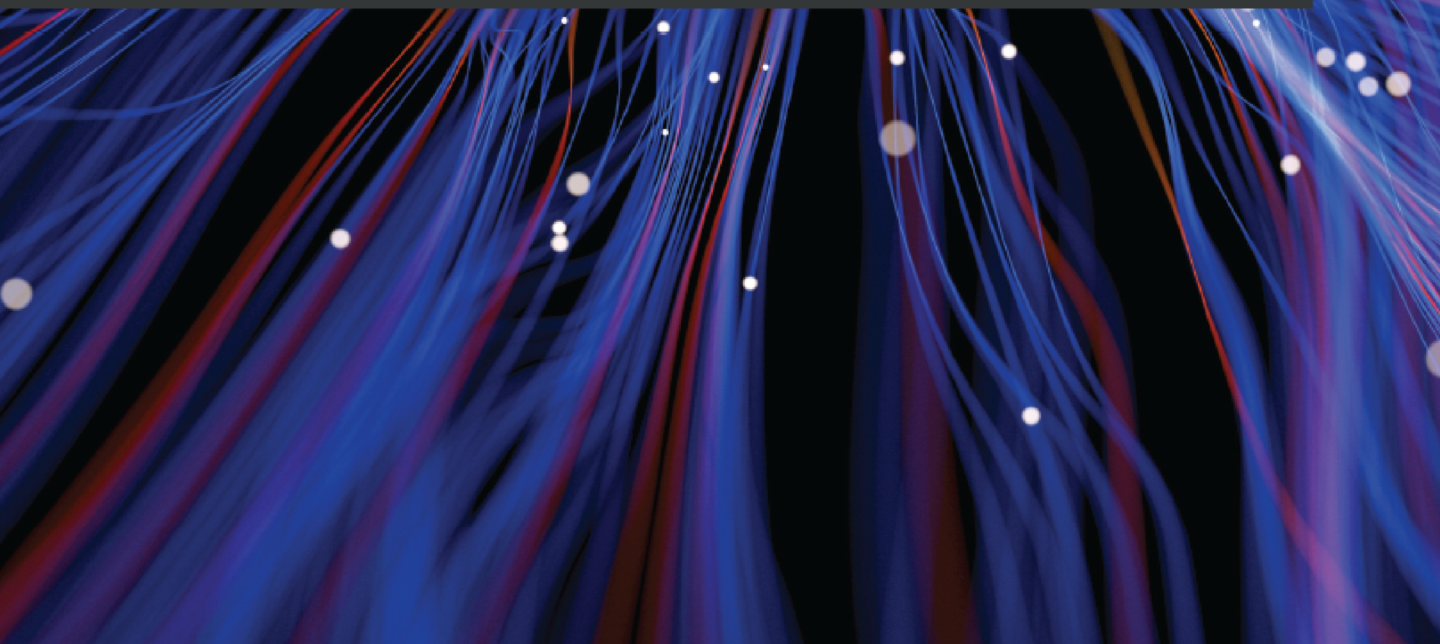
Modern Computer Vision with PyTorch

Explore deep learning concepts and implement over 50 real-world image applications



V Kishore Ayyadevara | Yeshwanth Reddy

Modern Computer Vision with PyTorch

Explore deep learning concepts and implement over 50 real-world image applications

V Kishore Ayyadevara
Yeshwanth Reddy

Packt

BIRMINGHAM - MUMBAI

Modern Computer Vision with PyTorch

Commissioning Editor: Sunith Shetty
Acquisition Editor: Siddharth Mandal
Content Development Editor: Joseph Sunil
Senior Editor: Roshan Kumar
Technical Editor: Manikandan Kurup
Copy Editor: Safis Editing
Project Coordinator: Aishwarya Mohan
Proofreader: Safis Editing
Indexer: Rekha Nair
Production Designer: Prashant Ghare

First published: November 2020

Production reference: 1261120

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-83921-347-2

www.packt.com

I am grateful to my wife, Sindhura, for her love, constant support, and for being a source of inspiration all throughout. Sincere thanks to the Packt team, Joseph Sunil, Kirti Pisat, Gebin George, and Tushar Gupta, for their support and belief in me.

Special thanks to the reviewers for their helpful feedback. This book would not have been in this shape without the great support and feedback I received from Kiran Kumar Meetakoti, Raghav Bali, Subhadip Maji, and Shresth Verma.

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the authors

V Kishore Ayyadevara leads a team focused on using AI to solve problems in the healthcare space. He has more than 10 years' experience in the field of data science with prominent technology companies. In his current role, he is responsible for developing a variety of cutting-edge analytical solutions that have an impact at scale while building strong technical teams.

Kishore has filed 8 patents at the intersection of machine learning, healthcare, and operations. Prior to this book, he authored four books in the fields of machine learning and deep learning. Kishore got his MBA from IIM Calcutta and his engineering degree from Osmania University.

I would like to dedicate this book to my dear parents, Hema and Subrahmanyeswara Rao, my lovely wife, Sindhura, and my dearest daughter, Hemanvi. This would not have been possible without their patience, support, and encouragement.

Yeshwanth Reddy is a senior data scientist with a strong focus on the research and implementation of cutting-edge technologies to solve problems in the health and computer vision domains. He has filed four patents in the field of OCR. He also has 2 years of teaching experience, where he delivered sessions to thousands of students in the fields of statistics, machine learning, AI, and natural language processing. He has completed his MTech and BTech at IIT Madras.

I would like to thank and dedicate this book to my dear parents Lalitha and Ravi, and my brother Sumanth - for whom I am ever grateful. My gratitude also goes to my extended family and all my friends who directly or indirectly helped me through the past year.

About the reviewer

Jamshaid Sohail is passionate about data science, machine learning, computer vision, and natural language processing and has more than 2 years of experience in the industry. He has worked in a Silicon Valley-based start-up named FunnelBeam, as a data scientist. He has worked with the founders of FunnelBeam from Stanford University. Currently, he is working as a data scientist at Systems Limited. He has completed over 66 online courses from different platforms. He authored the book *Data Wrangling with Python 3.X* for Packt Publishing and has reviewed multiple books and courses. He is also developing a comprehensive course on data science at Educative and is in the process of writing books for multiple publishers.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Section 2 - Object Classification and Detection

Section 4 - Combining Computer Vision with Other Techniques

Preface

Artificial Intelligence (**AI**) is here, and has become a powerful force and is fuelling some of the modern applications that are used on a daily basis. Much like the discovery/invention of fire, wheel, oil, electricity, and electronics - Artificial Intelligence is reshaping our world in ways that we could only fantasize about. AI has been historically a niche computer science subject, offered by a handful of labs. But because of the explosion of excellent theory, increase in computing power, and availability of data, the field started growing exponentially since the 2000s and has shown no sign of slowing down anytime soon.
AI has proven again and again that given the right algorithm and enough amount of data, it can learn the task by itself with limited human intervention and produce results that rival human judgement and sometimes even surpass them. Whether you are a rookie learning the ropes or a veteran driving large organizations, there is every reason to understand how AI works. Neural networks are some of the most flexible classes of Artificial Intelligence algorithms that have been adapted to a vast range of applications including structured data, text, and vision domains.

This book starts with the basics of neural networks and covers over 50 applications of computer vision. First, you will build a **neural network** (**NN**) from scratch using both NumPy, PyTorch, and then learn the best practices of tweaking a NN's hyper-parameters. As we progress, you will learn about CNNs, transfer-learning with a focus on classifying images. You will also learn about the practical aspects to take care of while building a NN model.

Next, you will learn about multi-object detection, segmentation, and implement them using R-CNN family, SSD, YOLO, U-Net, Mask-RCNN architectures. You will then learn to use the Detectron2 framework to simplify the process of building a NN for object detection and human-pose-estimation. Finally, you will implement 3-D object detection.

Subsequently, you will learn about auto-encoders and GANs with a strong focus on image manipulation and generation. Here, you will implement VAE, DCGAN, CGAN, Pix2Pix, CycleGan, StyleGAN2, SRGAN, Style-Transfer to manipulate images on a variety of tasks.

You will then learn to combine NLP and CV techniques while performing OCR, Image Captioning, object detection with transformers. Next, you will learn to combine RL with CV techniques to implement a self-driving car agent. Finally, you'll wrap up with moving a NN model to production and learn conventional CV techniques using the OpenCV library.

Who this book is for

This book is for newcomers to PyTorch and intermediate-level machine learning practitioners who are looking to become well versed in CV techniques using deep learning and PyTorch. Those who are just getting started with NNs will also find this book useful. Basic knowledge of the Python programming language and machine learning is all you need to get started with this book.

What this book covers

`Chapter 1`, *Artificial Neural Network Fundamentals*, gives you the complete details of how a neural network works. You will start by learning the key terminology associated with neural networks. Next, you will understand the working details of the building blocks and build a neural network from scratch on a toy dataset. By the end of this chapter, you will be confident about how a neural network works.

`Chapter 2`, *PyTorch Fundamentals*, introduces you to working with PyTorch. You will learn about the ways of creating and manipulating tensor objects before learning about the different ways of building a neural network model using PyTorch. You will still work with a toy dataset so that you understand the specifics of working with PyTorch.

`Chapter 3`, *Building a Deep Neural Network with PyTorch*, combines all that has been covered in the previous chapters to understand the impact of various neural network hyperparameters on model accuracy. By the end of this chapter, you will be confident about working with neural networks on a realistic dataset.

`Chapter 4`, *Introducing Convolutional Neural Networks*, details the challenges of using a vanilla neural network and you will be exposed to the reason why convolutional neural networks overcome the various limitations of traditional neural networks. You will dive deep into the working details of CNN and understand the various components in it. Next, you will learn the best practices of working with images. In this chapter, you will start working with real-world images and learn the intricacies of how CNNs help in image classification.

`Chapter 5`, *Transfer Learning for Image Classification*, exposes you to solving image classification problems in real-world. You will learn about multiple transfer learning architectures and also understand how it helps in significantly improving the image classification accuracy. Next, you will leverage transfer learning to implement the use cases of facial keypoint detection and age, gender estimation.

`Chapter 6`, *Practical Aspects of Image Classification*, provides insight into the practical aspects to take care of while building and deploying image classification models. You will practically see the advantages of leveraging data augmentation and batch normalization on real-world data. Further, you will learn about how class activation maps help in explaining the reason why CNN model predicted a certain outcome. By the end of this chapter, you can confidently tackle a majority of image classification problems and leverage the models discussed in the previous 3 chapters on your custom dataset.

`Chapter 7`, *Basics of Object Detection*, lays the foundation for object detection where you will learn about the various techniques that are used to build an object detection model. Next, you will learn about region proposal-based object-detection techniques through a use case where you will implement a model to locate trucks and buses in an image.

`Chapter 8`, *Advanced Object Detection*, exposes you to the limitations of the region-proposal based architectures. You will then learn about the working details of more advanced architectures that address the issues of region proposal-based architectures. You will implement all the architectures on the same dataset (trucks vs buses detection) so that you can contrast how each architecture works.

`Chapter 9`, *Image Segmentation*, builds upon the learnings in previous chapters and will help you build models that pin-point the location of the objects of various classes as well as instances of objects in an image. You will implement the use cases on images of a road and also on images of common household. By the end of this chapter, you will confidently tackle any image classification, object detection/segmentation problem and solve it by building a model using PyTorch.

`Chapter 10`, *Applications of Object Detection and Segmentation*, sums up the learnings of all the previous chapters where you will implement object detection, segmentation in a few lines of code, implement models to perform human crowd counting and image colorization. Finally, you will also learn about how 3D object detection on a real-world dataset.

`Chapter 11`, *Autoencoders and Image Manipulation*, , lays the foundation for modifying an image. You will start by learning about various autoencoders that help in compressing an image and also generating novel images. Next, you will learn about adversarial attack that fools a model before implementing neural style transfer. Finally, you will implement an autoencoder to generate deep fake images.

`Chapter 12`, *Image Generation Using GANs*, starts by giving you a deep dive into how GANs work. Next, you will implement fake facial image generation as well as generating images of interest using GANs.

`Chapter 13`, *Advanced GANs to Manipulate Images*, takes image manipulation to the next level. You will implement GANs to convert objects from one class to another, generate images from sketches, and manipulate custom images so that we can generate an image in a specific style. By the end of this chapter, you can confidently perform image manipulation using a combination of autoencoders and GANs.

`Chapter 14`, *Training with Minimal Data Points*, lays the foundation where you will learn about leveraging other techniques in combination with computer vision techniques. You will also learn about classifying images from minimal and also zero training data points.

`Chapter 15`, *Combining Computer Vision and NLP Techniques*, gives you the working details of various NLP techniques like word embedding, LSTM, transformer, using which you will implement applications like image captioning, OCR, and object detection with transformers.

`Chapter 16`, *Combining Computer Vision and Reinforcement Learning*, starts by exposing you to the terminology of RL and also the way to assign value to a state. You will appreciate how RL and neural networks can be combined as you learn about Deep Q-Learning. With this learning, you will implement an agent to play the game of Pong and also an agent to implement a self-driving car.

`Chapter 17`, *Moving a Model to Production*, describes the best practices of moving a model to production. You will first learn about deploying a model on a local server before moving it to the AWS public cloud.

`Chapter 18`, *Using OpenCV Utilities for Image Analysis*, details the various OpenCV utilities to create 5 interesting applications. Through this chapter, you will learn about utilities that aid deep learning as well as utilities that can substitute deep learning in scenarios where there are considerable constraints on memory or speed of inference.

To get the most out of this book

Software/hardware covered in the book	OS requirements
Minimum 128 GB storage Minimum 8 GB RAM Intel i5 processor or better NVIDIA 8+ GB graphics card – GTX1070 or better Minimum 50 Mbps internet speed	Windows, Linux, and macOS
Python 3.6 and above	Windows, Linux, and macOS
PyTorch 1.7	Windows, Linux, and macOS
Google Colab (can run in any browser)	Windows, Linux, and macOS

Do note that almost all the code in the book can be run using Google Colab by clicking the **Open Colab** button in each of the notebooks for the chapters on GitHub.

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Modern-Computer-Vision-with-PyTorch`. In case there's an update to the code, it will be updated on the existing GitHub repository.

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://static.packt-cdn.com/downloads/9781839213472_ColorImages.pdf`.

Conventions used

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

`CodeInText`: Indicates code words in the text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "We are creating an object of the `FMNISTDataset` class named `val`, in addition to the `train` object that we saw earlier."

A block of code is set as follows:

```
# Crop image
img = img[50:250,40:240]
# Convert image to grayscale
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# Show image
plt.imshow(img_gray, cmap='gray')
```

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 accuracy(x, y, model):
    model.eval() # <- let's wait till we get to dropout section
    # get the prediction matrix for a tensor of `x` images
    prediction = model(x)
    # compute if the location of maximum in each row coincides
    # with ground truth
    max_values, argmaxes = prediction.max(-1)
    is_correct = argmaxes == y
    return is_correct.cpu().numpy().tolist()
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "We will apply gradient descent (after a feedforward pass) using one **batch** at a time until we exhaust all data points within **one epoch of training**."

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: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@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 would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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@packt.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 `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Section 1 - Fundamentals of Deep Learning for Computer Vision

In this section, we will learn what the basic building blocks of a neural network are, and what the role of each block is, in order to successfully train a network. In this part, we will first briefly look at the theory of neural networks, before moving on to building and training neural networks with the PyTorch library.

This section comprises the following chapters:

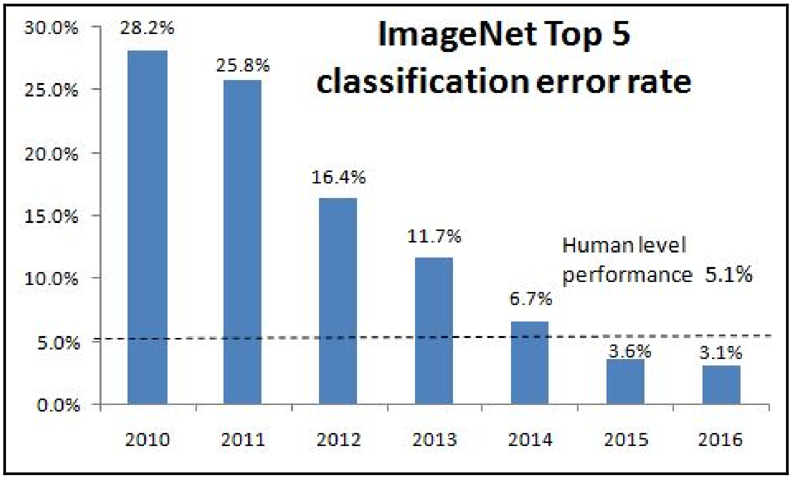
- `Chapter 1`, *Artificial Neural Network Fundamentals*
- `Chapter 2`, *PyTorch Fundamentals*
- `Chapter 3`, *Building a Deep Neural Network with PyTorch*

1
Artificial Neural Network Fundamentals

An **Artificial Neural Network** (**ANN**) is a supervised learning algorithm that is loosely inspired by the way the human brain functions. Similar to the way neurons are connected and activated in the human brain, a neural network takes input and passes it through a function, resulting in certain subsequent neurons getting activated, and consequently producing the output.

There are several standard ANN architectures. The universal approximation theorem says that we can always find a large enough neural network architecture with the right set of weights that can exactly predict any output for any given input. This means, for a given dataset/task we can create an architecture and keep adjusting its weights until the ANN predicts what we want it to predict. Adjusting the weights until this happens is called training the neural network. Successful training on large datasets and customized architecture is how ANNs have gained prominence in solving various relevant tasks.

One of the prominent tasks in computer vision is to recognize the class of the object present in an image. ImageNet was a competition held to identify the class of objects present in an image. The reduction in classification error rate over the years is as follows:



The year 2012 was when a neural network (AlexNet) was used in the winning solution of the competition. As you can see from the preceding chart, there was a considerable reduction in errors from the year 2011 to the year 2012 by leveraging neural networks. Over time since then, with more deep and complex neural networks, the classification error kept reducing and has beaten human-level performance. This gives a solid motivation for us to learn and implement neural networks for our custom tasks, where applicable.

In this chapter, we will create a very simple architecture on a simple dataset and mainly focus on how the various building blocks (feedforward, backpropagation, learning rate) of an ANN help in adjusting the weights so that the network learns to predict the expected outputs from given inputs. We will first learn, mathematically, what a neural network is, and then build one from scratch to have a solid foundation. Then we will learn about each component responsible for training the neural network and code them as well. Overall, we will cover the following topics:

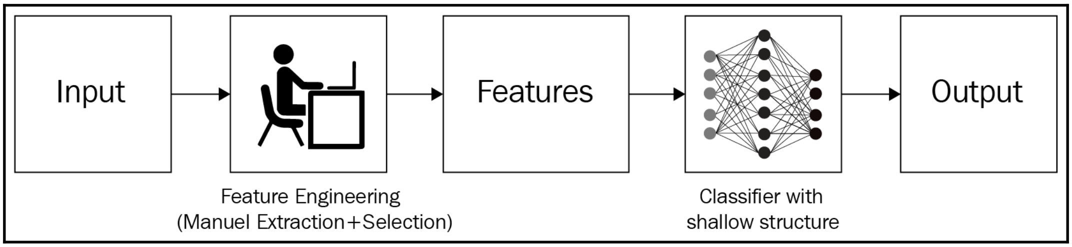
- Comparing AI and traditional machine learning
- Learning about the artificial neural network building blocks
- Implementing feedforward propagation
- Implementing backpropagation

- Putting feedforward propagation and backpropagation together
- Understanding the impact of the learning rate
- Summarizing the training process of a neural network

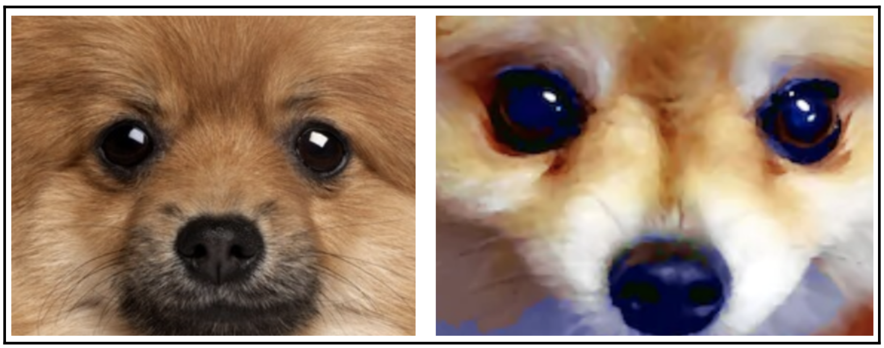
Comparing AI and traditional machine learning

Traditionally, systems were made intelligent by using sophisticated algorithms written by programmers.

For example, say you are interested in recognizing whether a photo contains a dog or not. In the traditional **Machine Learning** (**ML**) setting, an ML practitioner or a subject matter expert first identifies the features that need to be extracted from images. Then they extract those features and pass them through a well-written algorithm that deciphers the given features to tell whether the image is of a dog or not. The following diagram illustrates the same idea:



Take the following samples:



From the preceding images, a simple rule might be that if an image contains three black circles aligned in a triangular shape, it can be classified as a dog. However, this rule would fail against this deceptive close-up of a muffin:

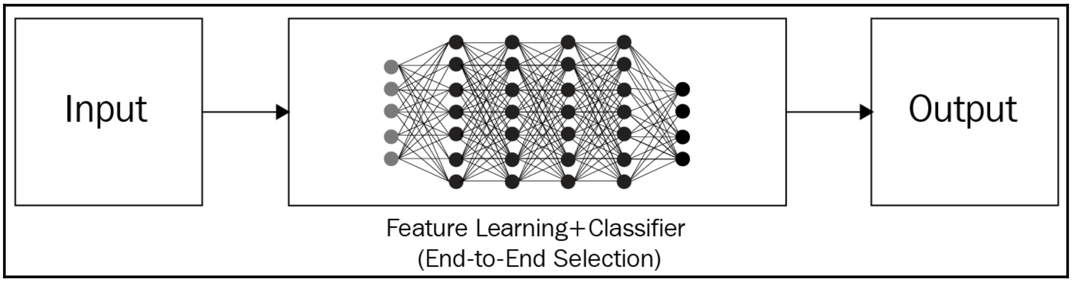
Of course, this rule also fails when shown an image with anything other than a dog's face close up. Naturally, therefore, the number of manual rules we'd need to create for the accurate classification of multiple types can be exponential, especially as images become more complex. Therefore, the traditional approach works well in a very constrained environment (say, taking a passport photo where all the dimensions are constrained within millimeters) and works badly in an unconstrained environment, where every image varies a lot.

We can extend the same line of thought to any domain, such as text or structured data. In the past, if someone was interested in programming to solve a real-world task, it became necessary for them to understand everything about the input data and write as many rules as possible to cover every scenario. This is tedious and there is no guarantee that all new scenarios would follow said rules.

However, by leveraging artificial neural networks, we can do this in a single step.

Neural networks provide the unique benefit of combining feature extraction (hand-tuning) and use those features for classification/regression in a single shot with little manual feature engineering. Both these subtasks only require labeled data (for example, which pictures are dogs and which pictures are not dogs) and neural network architecture. It does not require a human to come up with rules to classify an image, which takes away the majority of the burden traditional techniques impose on the programmer.

Notice that the main requirement is that we provide a considerable amount of examples for the task that needs a solution. For example, in the preceding case, we need to provide lots and lots of *dog* and *not-dog* pictures to the model so it learns the features. A high-level view of how neural networks are leveraged for the task of classification is as follows:



Now that we have gained a very high-level overview of the fundamental reason why neural networks perform better than traditional computer vision methods, let's gain a deeper understanding of how neural networks work throughout the various sections in this chapter.

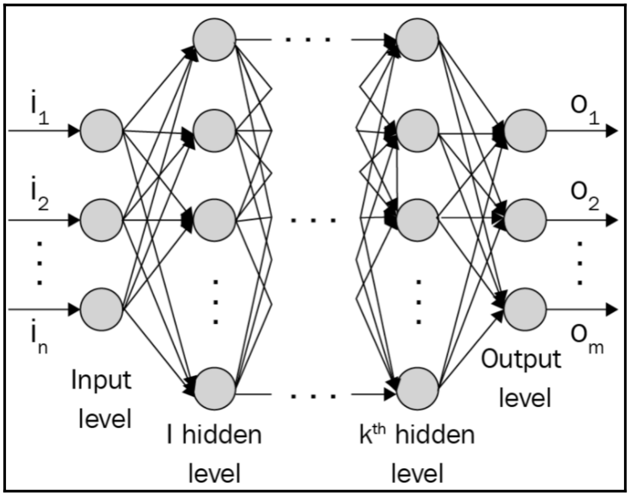
Learning about the artificial neural network building blocks

An ANN is a collection of tensors (weights) and mathematical operations, arranged in such a way to loosely replicate the functioning of a human brain. It can be viewed as a mathematical function that takes in one or more tensors as inputs and predicts one or more tensors as outputs. The arrangement of operations that connects these inputs to outputs is referred to as the architecture of the neural network – which we can customize based on the task at hand, that is, based on whether the problem contains structured (tabular) or unstructured (image, text, audio) data (which is the list of input and output tensors).

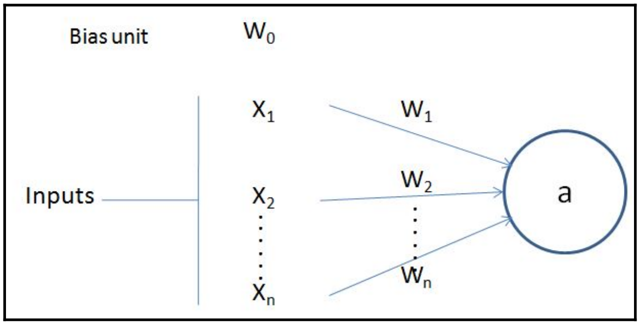
An ANN is made up of the following:

- **Input layers**: These layers take the independent variables as input.
- **Hidden (intermediate) layers**: These layers connect the input and output layers while performing transformations on top of input data. Furthermore, the hidden layers contain **nodes** (units/circles in the following diagram) to modify their input values into higher-/lower-dimensional values. The functionality to achieve a more complex representation is achieved by using various activation functions that modify the values of the nodes of intermediate layers.
- **Output layer**: This contains the values the input variables are expected to result in.

With this in mind, the typical structure of a neural network is as follows:



The number of **nodes** (circles in the preceding diagram) in the output layer depends on the task at hand and whether we are trying to predict a continuous variable or a categorical variable. If the output is a continuous variable, the output has one node. If the output is categorical with *m* possible classes, there will be *m* nodes in the output layer. Let's zoom into one of the nodes/neurons and see what's happening. A neuron transforms its inputs as follows:



In the preceding diagram, x_1,x_2, ..., x_n are the input variables, and w_0 is the bias term (similar to the way we have a bias in linear/logistic regression).

Note that w_1,w_2, ..., w_n are the weights given to each of the input variables and w_0 is the bias term. The output value *a* is calculated as follows:

$$a = f(w_0 + \sum_{i=1}^{n} w_i x_i)$$

As you can see, it is the sum of the products of *weight and input* pairs followed by an additional function *f* (the bias term + sum of products). The function *f* is the activation function that is used to apply non-linearity on top of this sum of products. More details on the activation functions will be provided in the next section, on feedforward propagation. Further, higher nonlinearity can be achieved by having more than one hidden layer, stacking multitudes of neurons.

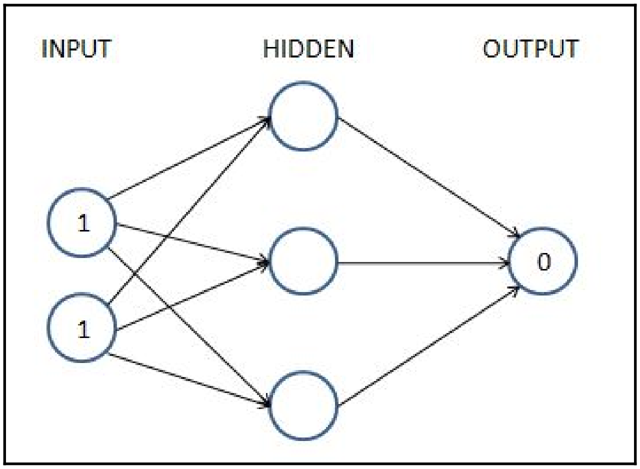
At a high level, a neural network is a collection of nodes where each node has an adjustable float value and the nodes are interconnected as a graph to return outputs in a format that is dictated by the architecture of the network. The network constitutes three main parts: the input layer, the hidden layer(s), and the output layer. Note that you can have a higher *number (n) of* hidden layers, with the term *deep* learning referring to the greater number of hidden layers. Typically, more hidden layers are needed when the neural network has to comprehend something complicated such as image recognition.

With the architecture of a neural network understood, in the next section, we will learn about feedforward propagation, which helps in estimating the amount of error (loss) the network architecture has.

Implementing feedforward propagation

To build a strong foundational understanding of how feedforward propagation works, we'll go through a toy example of training a neural network where the input to the neural network is (1, 1) and the corresponding (expected) output is 0. Here, we are going to find the optimal weights of the neural network based on this single input-output pair. However, you should note that in reality, there will be thousands of data points on which an ANN is trained.

Our neural network architecture for this example contains one hidden layer with three nodes in it, as follows:



Every arrow in the preceding diagram contains exactly one float value (**weight**) that is adjustable. There are 9 (6 in the first hidden layer and 3 in the second) floats that we need to find, so that when the input is (1,1), the output is as close to (0) as possible. This is what we mean by training the neural network. We have not introduced a bias value yet, for simplicity purposes only – the underlying logic remains the same.

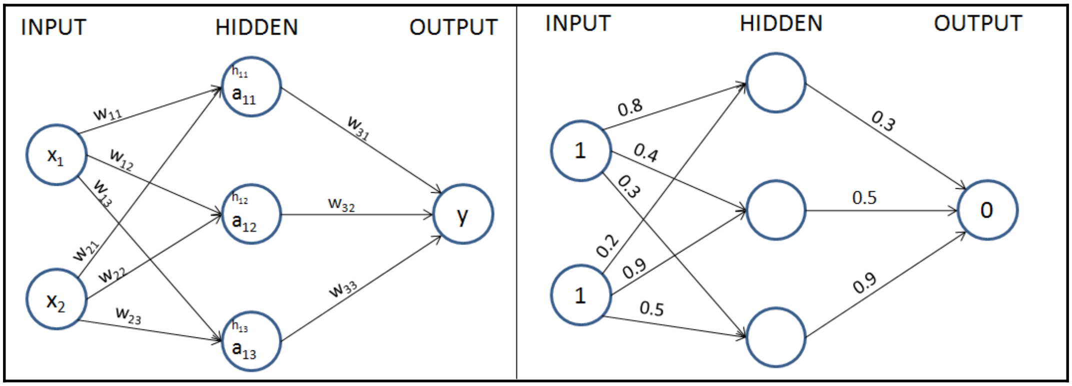
In the subsequent sections, we will learn the following about the preceding network:

- Calculating hidden layer values
- Performing non-linear activations
- Estimating the output layer value
- Calculating the loss value corresponding to the expected value

Calculating the hidden layer unit values

We'll now assign weights to all of the connections. In the first step, we assign weights randomly across all the connections. And in general, neural networks are initialized with random weights before the training starts. Again, for simplicity, while introducing the topic, we will **not** include the bias value while learning about feedforward propagation and backpropagation. But we will have it while implementing both feedforward propagation and backpropagation from scratch.

Let's start with initial weights that are randomly initialized between 0 and 1, but note that the final weights after the training process of a neural network don't need to be between a specific set of values. A formal representation of weights and values in the network is provided in the following diagram (left half) and the randomly initialized weights are provided in the network in the right half.



In the next step, we perform the multiplication of the input with weights to calculate the values of hidden units in the hidden layer.

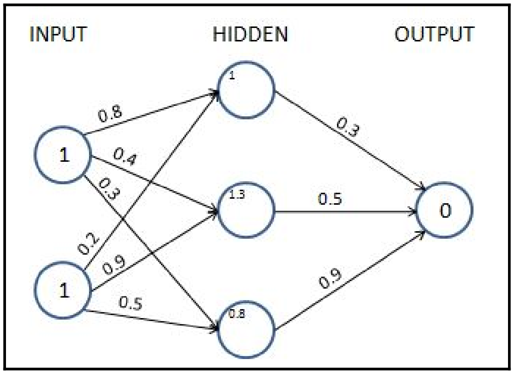
The hidden layer's unit values before activation are obtained as follows:

$$h_{11} = X_1 * w_{11} + X_2 * w_{21} = 1 * 0.8 + 1 * 0.2 = 1$$

$$h_{12} = X_1 * w_{12} + X_2 * w_{22} = 1 * 0.4 + 1 * 0.9 = 1.3$$

$$h_{13} = X_1 * w_{13} + X_2 * w_{23} = 1 * 0.3 + 1 * 0.5 = 0.8$$

The hidden layer's unit values (before activation) that are calculated here are also shown in the following diagram:



Now, we will pass the hidden layer values through a non-linearity activation. Note that, if we do not apply a non-linear activation function in the hidden layer, the neural network becomes a giant linear connection from input to output, no matter how many hidden layers exist.

Applying the activation function

Activation functions help in modeling complex relations between the input and the output.

Some of the frequently used activation functions are calculated as follows (where *x* is the input):

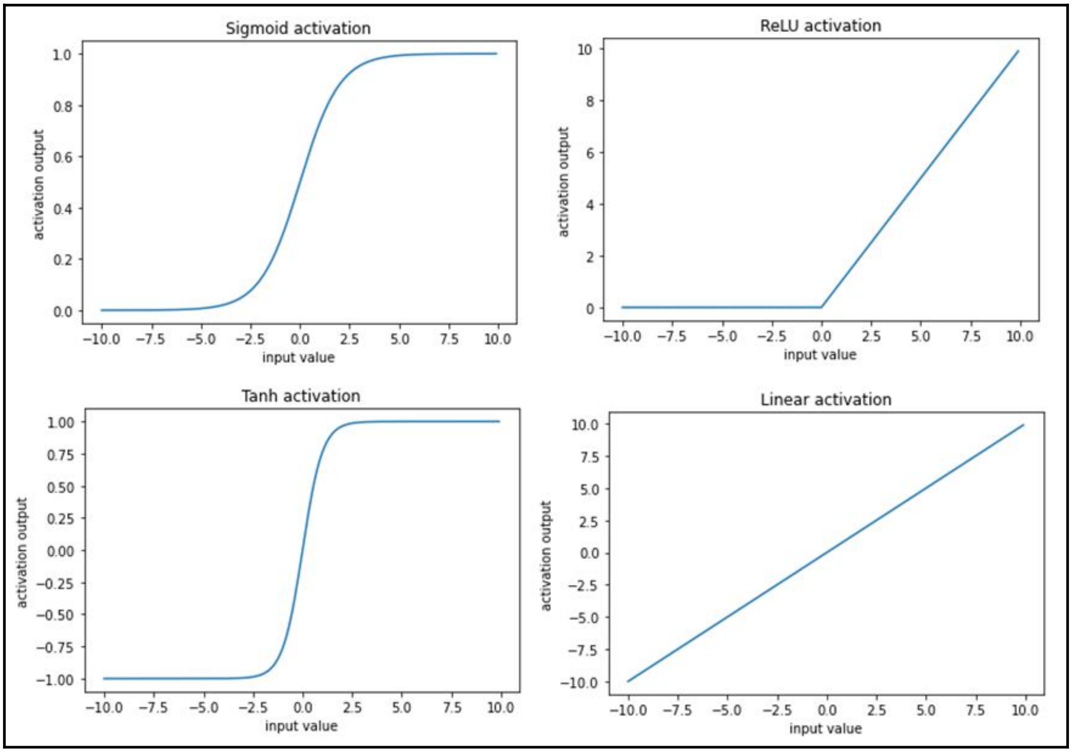
$$Sigmoid\ activation(x) = \frac{1}{1+e^{-x}}$$

$$ReLU\ activation(x) = \begin{cases} x & if\ x > 0 \\ 0 & if\ x \leqslant\ 0 \end{cases}$$

$$Tanh\ activation(x)\ = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

$$Linear\ activation(x) = x$$

Visualizations of each of the preceding activations for various input values are as follows:



For our example, let's use the sigmoid (logistic) function for activation.

By applying sigmoid (logistic) activation, *S(x)*, to the three hidden layer *sums*, we get the following values after sigmoid activation:

$$a_{11} = S(1.0) = \frac{1}{(1+e^{-1})} = 0.73$$

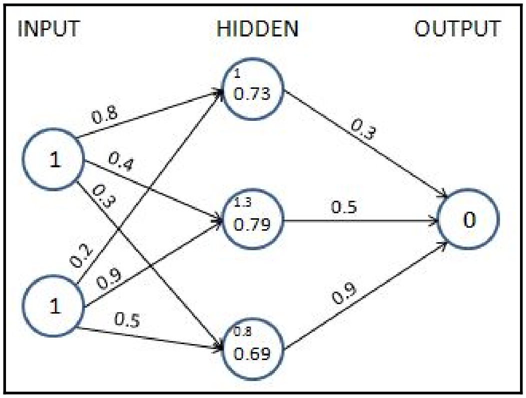
$$a_{12} = S(-1.3) = \frac{1}{(1+e^{-1.3})} = 0.79$$

$$a_{13} = S(-0.8) = \frac{1}{(1+e^{-0.8})} = 0.69$$

Now that we have obtained the hidden layer values after activation, in the next section, we will obtain the output layer values.

Calculating the output layer values

So far, we have calculated the final hidden layer values after applying the sigmoid activation. Using the hidden layer values after activation, and the weight values (which are randomly initialized in the first iteration), we will calculate the output value for our network:



We perform the sum of products of the hidden layer values and weight values to calculate the output value. Another reminder: we excluded the bias terms that need to be added at each unit(node), only to simplify our understanding of the working details of feedforward propagation and backpropagation for now and will include it while coding up feedforward propagation and backpropagation:

$$output\ node\ value\ (\hat{y}) = \ 0.73 * 0.3 + 0.79 * 0.5 + 0.69 * 0.9 \ = \ 1.235$$

Because we started with a random set of weights, the value of the output node is very different from the target. In this case, the difference is *1.235* (remember, the target is 0). In the next section, we will learn about calculating the loss value associated with the network in its current state.

Calculating loss values

Loss values (alternatively called cost functions) are the values that we optimize for in a neural network. To understand how loss values get calculated, let's look at two scenarios:

- Categorical variable prediction
- Continuous variable prediction

Calculating loss during continuous variable prediction

Typically, when the variable is continuous, the loss value is calculated as the mean of the square of the difference in actual values and predictions, that is, we try to minimize the mean squared error by varying the weight values associated with the neural network. The mean squared error value is calculated as follows:

$$J_\theta = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

$$\hat{y}_i = \eta_\theta(x_i)$$

In the preceding equation, y_i is the actual output. $\hat{y}_i$ is the prediction computed by the neural network η (whose weights are stored in the form of θ), where its input x_i is , and m is the number of rows in the dataset.

> The key takeaway should be the fact that for every unique set of weights, the neural network would predict a different loss and we need to find the golden set of weights for which the loss is zero (or, in realistic scenarios, as close to zero as possible).

In our example, let's assume that the outcome that we are predicting is continuous. In that case, the loss function value is the mean squared error, which is calculated as follows:

$$loss(error) = 1.235^2 = 1.52$$

Now that we understand how to calculate the loss value for a continuous variable, in the next section, we will learn about calculating the loss value for a categorical variable.

Calculating loss during categorical variable prediction

When the variable to predict is discrete (that is, there are only a few categories in the variable), we typically use a categorical cross-entropy loss function. When the variable to predict has two distinct values within it, the loss function is binary cross-entropy.

Binary cross-entropy is calculated as follows:

$$-\frac{1}{m}\sum_{i=1}^{m}(y_i \log(p_i) + (1 - y_i)\log(1 - p_i))$$

y is the actual value of the output, *p* is the predicted value of the output, and *m* is the total number of data points.

Categorical cross-entropy is calculated as follows:

$$-\frac{1}{m}\sum_{j=1}^{C}\sum_{i=1}^{m} y_i log(p_i)$$

y is the actual value of the output, *p* is the predicted value of the output, *m* is the total number of data points, and *C* is the total number of classes.

A simple way of visualizing cross-entropy loss is to look at the prediction matrix itself. Say you are predicting five classes – Dog, Cat, Rat, Cow, and Hen – in an image recognition problem. The neural network would necessarily have five neurons in the last layer with softmax activation (more on softmax in the next section). It will be thus forced to predict a probability for every class, for every data point. Say there are five images and the prediction probabilities look like so (the highlighted cell in each row corresponds to the target class):

Target (Correct class)	Prediction probabilities					Cross entropy loss	
	Dog	Cat	Cow	Hen	Rat		
Dog	0.88	0.02	0.04	0.04	0.02	-log(0.88)	= 0.128
Cat	0.26	0.21	0.17	0.18	0.18	-log(0.21)	= 1.56
Cow	0.01	0.01	0.96	0.01	0.01	-log(0.96)	= 0.04
Hen	0.14	0.09	0.01	0.57	0.19	-log(0.57)	= 0.56
Rat	0.21	0.02	0.05	0.17	0.55	-log(0.55)	= 0.597

Note that each row sums to 1. In the first row, when the target is **Dog** and the prediction probability is **0.88**, the corresponding loss is **0.128** (which is the negative of the log of **0.88**). Similarly, other losses are computed. As you can see, the loss value is less when the probability of the correct class is high. As you know, the probabilities range between 0 and 1. So, the minimum possible loss can be 0 (when the probability is 1) and the maximum loss can be infinity when the probability is 0.

The final loss within a dataset is the mean of all individual losses across all rows.

Now that we have a solid understanding of calculating mean squared error loss and cross-entropy loss, let's get back to our toy example. Assuming our output is a continuous variable, we will learn how to minimize the loss value using backpropagation in a later section. We will update the weight values θ (which were initialized randomly earlier) to minimize the loss (J_θ). But, before that, let's first code feedforward propagation in Python using NumPy arrays to solidify our understanding of its working details.

Feedforward propagation in code

A high-level strategy of coding feedforward propagation is as follows:

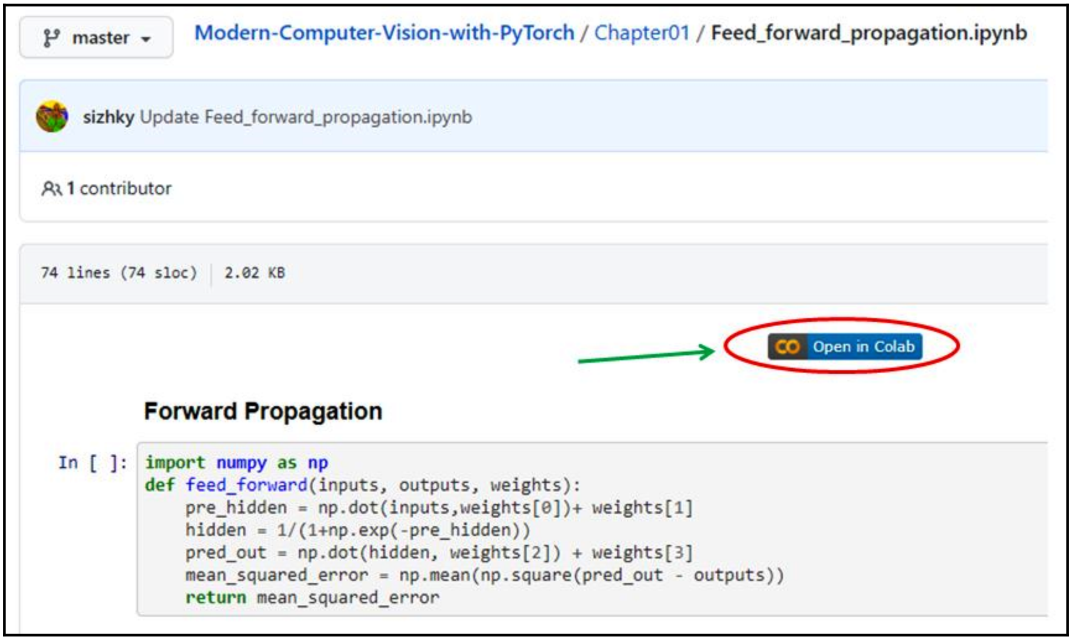
1. Perform a sum product at each neuron.
2. Compute activation.
3. Repeat the first two steps at each neuron until the output layer.
4. Compute the loss by comparing the prediction with the actual output.

It is going to be a function that takes in input data, current neural network weights, and output data as the inputs to the function and returns the loss of the current network state.

The feedforward function to calculate the mean squared error loss values across all data points is as follows:

The following code is available as `Feed_forward_propagation.ipynb` in the `Chapter01` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

We strongly encourage you to execute the code notebooks by clicking the **Open in Colab** button in each notebook. A sample screenshot is as follows:



Once you click on **Open in Colab** (highlighted in the preceding screenshot), you will be able to execute all the code without any hassle and should be able to replicate the results shown in this book.

With the way to execute code in place, let's go ahead and code feedforward propagation:

1. Take the input variable values (`inputs`), `weights` (randomly initialized if this is the first iteration), and the actual `outputs` in the provided dataset as the parameters of the `feed_forward` function:

```
import numpy as np
def feed_forward(inputs, outputs, weights):
```

To make this exercise a little more realistic, we will have bias associated with each node. Thus the weights array will contain not only the weights connecting different nodes but also the bias associated with nodes in hidden/ output layers.

2. Calculate hidden layer values by performing the matrix multiplication (`np.dot`) of `inputs` and weight values (`weights[0]`) connecting the input layer to the hidden layer and add the bias terms (`weights[1]`) associated with the hidden layer's nodes:

```
pre_hidden = np.dot(inputs,weights[0])+ weights[1]
```

3. Apply the sigmoid activation function on top of the hidden layer values obtained in the previous step – `pre_hidden`:

```
hidden = 1/(1+np.exp(-pre_hidden))
```

4. Calculate the output layer values by performing the matrix multiplication (`np.dot`) of hidden layer activation values (`hidden`) and weights connecting the hidden layer to the output layer (`weights[2]`) and summing the output with bias associated with the node in the output layer – `weights[3]`:

```
pred_out = np.dot(hidden, weights[2]) + weights[3]
```

5. Calculate the mean squared error value across the dataset and return the mean squared error:

```
mean_squared_error = np.mean(np.square(pred_out \
                                        - outputs))
return mean_squared_error
```

We are now in a position to get the mean squared error value as we forward-pass through the network.

Before we learn about backpropagation, let's learn about some constituents of the feedforward network that we built previously – the activation functions and loss value calculation – by implementing them in NumPy so that we have a detailed understanding of how they work.

Activation functions in code

While we applied the sigmoid activation on top of the hidden layer values in the preceding code, let's examine other activation functions that are commonly used:

- **Tanh**: The tanh activation of a value (the hidden layer unit value) is calculated as follows:

```
def tanh(x):
    return (np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
```

- **ReLU**: The **Rectified Linear Unit** (**ReLU**) of a value (the hidden layer unit value) is calculated as follows:

```
def relu(x):
    return np.where(x>0,x,0)
```

- **Linear**: The linear activation of a value is the value itself. This is represented as follows:

```
def linear(x):
    return x
```

- **Softmax**: Unlike other activations, softmax is performed on top of an array of values. This is generally done to determine the probability of an input belonging to one of the *m* number of possible output classes in a given scenario. Let's say we are trying to classify an image of a digit into one of the possible 10 classes (numbers from 0 to 9). In this case, there are 10 output values, where each output value should represent the probability of an input image belonging to one of the 10 classes.

  Softmax activation is used to provide a probability value for each class in the output and is calculated as follows:

```
def softmax(x):
    return np.exp(x)/np.sum(np.exp(x))
```

Notice the two operations on top of input `x` – `np.exp` will make all values positive, and the division by `np.sum(np.exp(x))` of all such exponents will force all the values to be in between 0 and 1. This range coincides with the probability of an event. And this is what we mean by returning a probability vector.

Now that we have learned about various activation functions, next, we will learn about the different loss functions.

Loss functions in code

Loss values (which are minimized during a neural network training process) are minimized by updating weight values. Defining the proper loss function is the key to building a working and reliable neural network model. The loss functions that are generally used while building a neural network are as follows:

- **Mean squared error**: The mean squared error is the squared difference between the actual and the predicted values of the output. We take a square of the error, as the error can be positive or negative (when the predicted value is greater than the actual value and vice versa). Squaring ensures that positive and negative errors do not offset each other. We calculate the **mean** of the squared error so that the error over two different datasets is comparable when the datasets are not of the same size.

  The mean squared error between an array of predicted output values (`p`) and an array of actual output values (`y`) is calculated as follows:

  ```
  def mse(p, y):
      return np.mean(np.square(p - y))
  ```

  The mean squared error is typically used when trying to predict a value that is continuous in nature.

- **Mean absolute error:** The mean absolute error works in a manner that is very similar to the mean squared error. The mean absolute error ensures that positive and negative errors do not offset each other by taking an average of the absolute difference between the actual and predicted values across all data points.

  The mean absolute error between an array of predicted output values (`p`) and an array of actual output values (`y`) is implemented as follows:

  ```
  def mae(p, y):
      return np.mean(np.abs(p-y))
  ```

  Similar to the mean squared error, the mean absolute error is generally employed on continuous variables. Further, in general, it is preferable to have a mean absolute error as a loss function when the outputs to predict have a value less than 1, as the mean squared error would reduce the magnitude of loss considerably (the square of a number between 1 and -1 is an even smaller number) when the expected output is less than 1.

- **Binary cross-entropy**: Cross-entropy is a measure of the difference between two different distributions: actual and predicted. Binary cross-entropy is applied to binary output data, unlike the previous two loss functions that we discussed (which are applied during continuous variable prediction).

  Binary cross-entropy between an array of predicted values (`p`) and an array of actual values (`y`) is implemented as follows:

  ```
  def binary_cross_entropy(p, y):
      return -np.mean(np.sum((y*np.log(p)+(1-y)*np.log(1-p))))
  ```

  Note that binary cross-entropy loss has a high value when the predicted value is far away from the actual value and a low value when the predicted and actual values are close.

- **Categorical cross-entropy**: Categorical cross-entropy between an array of predicted values (`p`) and an array of actual values (`y`) is implemented as follows:

  ```
  def categorical_cross_entropy(p, y):
      return -np.mean(np.sum(y*np.log(p)))
  ```

So far, we have learned about feedforward propagation, and various components, such as weight initialization, bias associated with nodes, activation, and loss functions, that constitute it. In the next section, we will learn about backpropagation, a technique to adjust weights so that they will result in a loss that is as small as possible.

Implementing backpropagation

In feedforward propagation, we connected the input layer to the hidden layer, which then was connected to the output layer. In the first iteration, we initialized weights randomly and then calculated the loss resulting from those weight values. In backpropagation, we take the reverse approach. We start with the loss value obtained in feedforward propagation and update the weights of the network in such a way that the loss value is minimized as much as possible.

The loss value is reduced as we perform the following steps:

1. Change each weight within the neural network by a small amount – one at a time.
2. Measure the change in loss (δL) when the weight value is changed (δW).
3. Update the weight by $-k.\frac{\delta L}{\delta W}$ (where k is a positive value and is a hyperparameter known as the **learning rate**).

> Note that the update made to a particular weight is proportional to the amount of loss that is reduced by changing it by a small amount. Intuitively, if changing a weight reduces the loss by a large value, then we can update the weight by a large amount. However, if the loss reduction is small by changing the weight, then we update it only by a small amount.

If the preceding steps are performed n number of times on the entire dataset (where we have done both the feedforward propagation and backpropagation), it essentially results in training for n **epochs**.

As a typical neural network contains thousands/millions (if not billions) of weights, changing the value of each weight, and checking whether the loss increased or decreased is not optimal. The core step in the preceding list is the measurement of "change of loss" when the weight is changed. As you might have studied in calculus, measuring this is the same as computing the **gradient** of loss concerning the weight. There's more on leveraging partial derivatives from calculus to calculate the gradient of the loss concerning the weight in the next section, on the chain rule for backpropagation.

In this section, we will implement gradient descent from scratch by updating one weight at a time by a small amount as detailed at the start of this section. However, before implementing backpropagation, let's understand one additional detail of neural networks: the **learning rate**.

Intuitively, the learning rate helps in building trust in the algorithm. For example, when deciding on the magnitude of the weight update, we would potentially not change the weight value by a big amount in one go but update it more slowly.

This results in obtaining stability in our model; we will look at how the learning rate helps with stability in the *Understanding the impact of the learning rate* section.

This whole process by which we update weights to reduce errors is called **gradient descent**.

Stochastic gradient descent is how errors are minimized in the preceding scenario. As mentioned earlier, **gradient** stands for the difference (which is the difference in loss values when the weight value is updated by a small amount) and **descent** means to reduce. **Stochastic** stands for the selection of random samples based on which a decision is taken.

Apart from stochastic gradient descent, many other similar optimizers help to minimize loss values; the different optimizers will be discussed in the next chapter.

In the next two sections, we will learn about coding the intuition for backpropagation from scratch in Python, and will also discuss in brief how backpropagation works using the chain rule.

Gradient descent in code

Gradient descent is implemented in Python as follows:

> The following code is available as `Gradient_descent.ipynb` in the `Chapter01` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Define the feedforward network and calculate the mean squared error loss value as we did in the *Feedforward propagation in code* section:

```
from copy import deepcopy
import numpy as np
def feed_forward(inputs, outputs, weights):
    pre_hidden = np.dot(inputs,weights[0])+ weights[1]
    hidden = 1/(1+np.exp(-pre_hidden))
    pred_out = np.dot(hidden, weights[2]) + weights[3]
    mean_squared_error = np.mean(np.square(pred_out \
                                            - outputs))
    return mean_squared_error
```

2. Increase each weight and bias value by a very small amount (0.0001) and calculate the overall squared error loss value one at a time for each of the weight and bias updates.

- In the following code, we are creating a function named `update_weights`, which performs the gradient descent process to update weights. The inputs to the function are the input variables to the network – `inputs`, expected `outputs`, `weights` (which are randomly initialized at the start of training the model), and the learning rate of the model – `lr` (more on the learning rate in a later section):

```
def update_weights(inputs, outputs, weights, lr):
```

- Ensure that you `deepcopy` the list of weights. As the weights will be manipulated in later steps, `deepcopy` ensures we can work with multiple copies of weights without disturbing actual weights. We will create three copies of the original set of weights that were passed as an input to the function – `original_weights`, `temp_weights`, and `updated_weights`:

```
    original_weights = deepcopy(weights)
    temp_weights = deepcopy(weights)
    updated_weights = deepcopy(weights)
```

- Calculate the loss value (`original_loss`) with the original set of weights by passing `inputs`, `outputs`, and `original_weights` through the `feed_forward` function:

```
    original_loss = feed_forward(inputs, outputs, \
                                 original_weights)
```

- We will loop through all the layers of the network:

```
    for i, layer in enumerate(original_weights):
```

- There are a total of four lists of parameters within our neural network – two lists for the weight and bias parameters that connect the input to the hidden layer and another two lists for the weight and bias parameters that connect the hidden layer to the output layer. Now, we loop through all the individual parameters and because each list has a different shape, we leverage `np.ndenumerate` to loop through each parameter within a given list:

```
        for index, weight in np.ndenumerate(layer):
```

- Now we store the original set of weights in `temp_weights`. We select its index weight present in the i^{th} layer and increase it by a small value. Finally, we compute the new loss with the new set of weights for the neural network:

```
temp_weights = deepcopy(weights)
temp_weights[i][index] += 0.0001
_loss_plus = feed_forward(inputs, outputs, \
                          temp_weights)
```

In the first line of the preceding code, we are resetting `temp_weights` to the original set of weights, as in each iteration, we update a different parameter to calculate the loss when a parameter is updated by a small amount within a given epoch.

- We calculate the gradient (change in loss value) due to the weight change:

```
grad = (_loss_plus - original_loss)/(0.0001)
```

This process of updating a parameter by a very small amount and then calculating the gradient is equivalent to the process of differentiation.

- Finally, we update the parameter present in the corresponding i^{th} layer and `index`, of `updated_weights`. The updated weight value will be reduced in proportion to the value of the gradient. Further, instead of completely reducing it by a value equal to the gradient value, we bring in a mechanism to build trust slowly by using the learning rate – `lr` (more on learning rate in the *Understanding the impact of the learning rate* section):

```
updated_weights[i][index] -= grad*lr
```

- Once the parameter values across all layers and indices within layers are updated, we return the updated weight values – `updated_weights`:

```
return updated_weights, original_loss
```

One of the other parameters in a neural network is the **batch size** considered in calculating the loss values.

In the preceding scenario, we considered all the data points to calculate the loss (mean squared error) value. However, in practice, when we have thousands (or in some cases, millions) of data points, the incremental contribution of a greater number of data points while calculating the loss value would follow the law of diminishing returns, and hence we would be using a batch size that is much smaller compared to the total number of data points we have. We will apply gradient descent (after feedforward propagation) using one **batch** at a time until we exhaust all data points within **one epoch of training**.

The typical batch size considered in building a model is anywhere between 32 and 1,024.

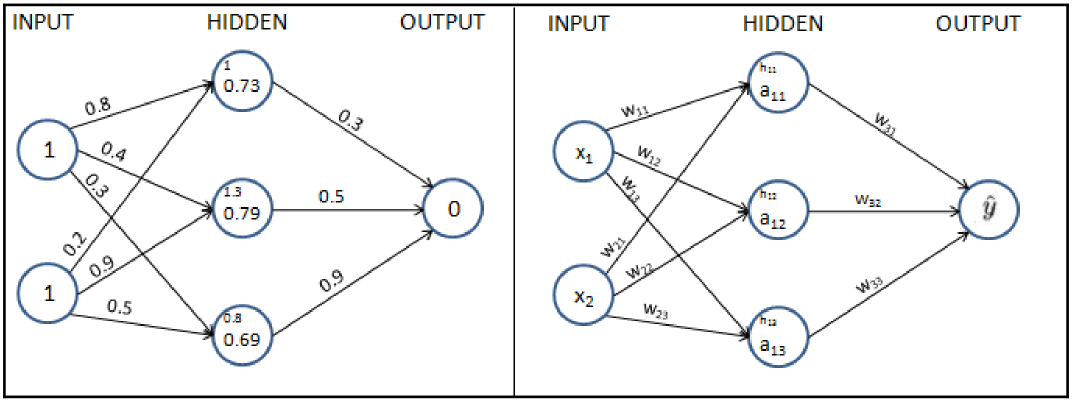
In this section, we learned about updating weight values based on the change in loss values when the weight values are changed by a small amount. In the next section, we will learn about how weights can be updated without computing gradients one gradient at a time.

Implementing backpropagation using the chain rule

So far, we have calculated gradients of loss concerning weight by updating the weight by a small amount and then calculating the difference between the feedforward loss in the original scenario (when the weight was unchanged) and the feedforward loss after updating weights. One drawback of updating weight values in this manner is that when the network is large, a large number of computations are needed to calculate loss values (and in fact, the computations are to be done twice – once where weight values are unchanged and again where weight values are updated by a small amount). This results in more computations and hence requires more resources and time. In this section, we will learn about leveraging the chain rule, which does not require us to manually compute loss values to come up with the gradient of the loss concerning the weight value.

In the first iteration (where we initialized weights randomly), the predicted value of the output is 1.235.

In order to get the theoretical formulation, let's denote the weights and hidden layer values and hidden layer activations as *w*, *h*, and *a* respectively as follows:



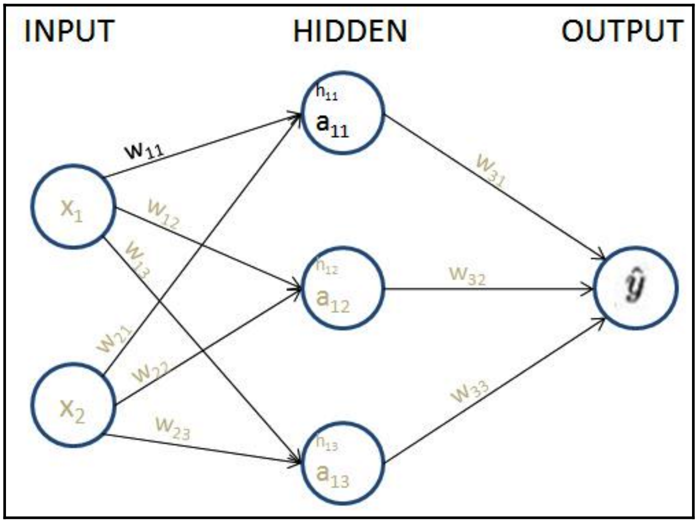
Note that, in the preceding diagrams, we have taken each component value of the left diagram and generalized it in the diagram on right.

In order to keep it easy to comprehend, in this section, we will understand how to use the chain rule to compute the gradient of loss value with respect to only w_{11}. The same learning can be extended to all the weights and biases of the neural network. We encourage you to practice and apply the chain rule calculation to the rest of the weights and bias values.

The `chain_rule.ipynb` notebook in the `Chapter01` folder of this book's GitHub repository contains the way to calculate gradients with respect to changes in weights and biases for all the parameters in a network using the chain rule.

Additionally, in order to keep this simple for our learning purposes, we will be working on only one data point, where the input is {1,1} and the expected output is {0}.

Given that we are calculating the gradient of loss value with w_{11}, let's understand all the intermediate components that are to be included while calculating the gradient through the following diagram (the components that do not connect the output to w_{11} are grayed out in the following diagram):



From the preceding diagram, we can see that w_{11} is contributing to the loss value through the path that is highlighted, – h_{11}, a_{11}, and $\hat{y}$.

Next, let's formulate how h_{11}, a_{11}, and $\hat{y}$ are obtained individually.

The loss value of the network is represented as follows:

$$MSE\ Loss\ (C) = \ (y - \hat{y})^2$$

The predicted output value $\hat{y}$ is calculated as follows:

$$\hat{y} = a_{11} * w_{21} \ + \ a_{12} * w_{22} \ + \ a_{13} * w_{23}$$

The hidden layer activation value (sigmoid activation) is calculated as follows:

$$a_{11} \ = \ \frac{1}{1 + e^{-h_{11}}}$$

The hidden layer value is calculated as follows:

$$h_{11} \ = \ x_1 * w_{11} \ + \ x_2 * w_{21}$$

Now that we have formulated all the equations, let's calculate the impact of the change in the loss value (*C*) with respect to the change in weight w_{11} as follows:

$$\frac{\partial C}{\partial w_{11}} = \frac{\partial C}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial a_{11}} * \frac{\partial a_{11}}{\partial h_{11}} * \frac{\partial h_{11}}{\partial w_{11}}$$

This is called a **chain rule**. Essentially, we are performing a chain of differentiations to fetch the differentiation of our interest.

Note that, in the preceding equation, we have built a chain of partial differential equations in such a way that we are now able to perform partial differentiation on each of the four components individually and ultimately calculate the derivative of the loss value with respect to weight value w_{11}.

The individual partial derivatives in the preceding equation are computed as follows:

- The partial derivative of the loss value with respect to the predicted output value $\hat{y}$ is as follows:

$$\frac{\partial C}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}}(y - \hat{y})^2 = -2 * (y - \hat{y})$$

- The partial derivative of the predicted output value $\hat{y}$ with respect to the hidden layer activation value a_{11} is as follows:

$$\frac{\partial \hat{y}}{\partial a_{11}} = \frac{\partial}{\partial a_{11}}(a_{11} * w_{21} + a_{12} * w_{22} + a_{13} * w_{23}) = w_{21}$$

- The partial derivative of the hidden layer activation value a_{11} with respect to the hidden layer value prior to activation h_{11} is as follows:

$$\frac{\partial a_{11}}{\partial h_{11}} = a_{11} * (1 - a_{11})$$

Note that the preceding equation comes from the fact that the derivative of the sigmoid function a is $a * (1 - a)$.

- The partial derivative of the hidden layer value prior to h_{11} activation with respect to the weight value w_{11} is as follows:

$$\frac{\partial h_{11}}{\partial w_{11}} = \frac{\partial}{\partial w_{11}}(x_1 * w_{11} + x_2 * w_{21}) = x_1$$

With this in place, the gradient of the loss value with respect to w_{11} is calculated by replacing each of the partial differentiation terms with the corresponding value as calculated in the previous steps as follows:

$$\frac{\partial C}{\partial w_{11}} = -2 * (y - \hat{y}) * w_{21} * a_{11} * (1 - a_{11}) * x_1$$

From the preceding formula, we can see that we are now able to calculate the impact on the loss value of a small change in the weight value (the gradient of the loss with respect to weight) without brute-forcing our way by recomputing the feedforward propagation again.

Next, we will go ahead and update the weight value as follows:

$$updated\ weight = original\ weight - learning\ rate * Gradient\ of\ loss\ with\ respect\ to\ weight$$

Working versions of the two methods, 1) identifying gradients using the chain rule and then updating weights, and 2) updating weight values by learning the impact a small change in weight value can have on loss values, resulting in the same values for updated weight values, are provided in the notebook `Chain_rule.ipynb` in the `Chapter01` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

In gradient descent, we performed the weight update process sequentially (one weight at a time). By leveraging the chain rule, we learned that there is an alternative way to calculate the impact of a change in weight by a small amount on the loss value, however, with an opportunity to perform computations in parallel.

Because we are updating parameters across all layers, the whole process of updating parameters can be parallelized. Further, given that in a realistic scenario, there can exist millions of parameters across layers, performing the calculation for each parameter on a different core of GPU results in the time taken to update weights is a much faster exercise than looping through each weight, one at a time.

Now that we have a solid understanding of backpropagation, both from an intuition perspective and also by leveraging the chain rule, in the next section, we will learn about how feedforward and backpropagation work together to arrive at the optimal weight values.

Putting feedforward propagation and backpropagation together

In this section, we will build a simple neural network with a hidden layer that connects the input to the output on the same toy dataset that we worked on in the *Feedforward propagation in code* section and also leverage the `update_weights` function that we defined in the previous section to perform backpropagation to obtain the optimal weight and bias values.

We define the model as follows:

1. The input is connected to a hidden layer that has three units/ nodes.
2. The hidden layer is connected to the output, which has one unit in the output layer.

The following code is available as `Back_propagation.ipynb` in the `Chapter01` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

We will create the network as follows:

1. Import the relevant packages and define the dataset:

```
from copy import deepcopy
import numpy as np
x = np.array([[1,1]])
y = np.array([[0]])
```

2. Initialize the weight and bias values randomly.

   The hidden layer has three units in it and each input node is connected to each of the hidden layer units. Hence, there are a total of six weight values and three bias values – one bias and two weights (two weights coming from two input nodes) corresponding to each of the hidden units. Additionally, the final layer has one unit that is connected to the three units of the hidden layer. Hence, a total of three weights and one bias dictate the value of the output layer. The randomly initialized weights are as follows:

```
W = [
    np.array([[-0.0053, 0.3793],
              [-0.5820, -0.5204],
              [-0.2723, 0.1896]], dtype=np.float32).T,
    np.array([-0.0140, 0.5607, -0.0628], dtype=np.float32),
    np.array([[ 0.1528,-0.1745,-0.1135]],dtype=np.float32).T,
    np.array([-0.5516], dtype=np.float32)
]
```

   In the preceding code, the first array of parameters correspond to the 2 x 3 matrix of weights that connect the input layer to the hidden layer. The second array of parameters represent the bias values associated with each node of the hidden layer. The third array of parameters correspond to the 3 x 1 matrix of weights joining the hidden layer to the output layer, and the final array of parameters represents the bias associated with the output layer.

3. Run the neural network through 100 epochs of feedforward propagation and backpropagation – the functions of which were already learned and defined as `feed_forward` and `update_weights` functions in the previous sections.

   - Define the `feed_forward` function:

```
def feed_forward(inputs, outputs, weights):
    pre_hidden = np.dot(inputs,weights[0])+ weights[1]
    hidden = 1/(1+np.exp(-pre_hidden))
    pred_out = np.dot(hidden, weights[2]) + weights[3]
    mean_squared_error = np.mean(np.square(pred_out \
                                           - outputs))
    return mean_squared_error
```

- Define the `update_weights` function:

```
def update_weights(inputs, outputs, weights, lr):
    original_weights = deepcopy(weights)
    temp_weights = deepcopy(weights)
    updated_weights = deepcopy(weights)
    original_loss = feed_forward(inputs, outputs, \
                                 original_weights)
    for i, layer in enumerate(original_weights):
        for index, weight in np.ndenumerate(layer):
            temp_weights = deepcopy(weights)
            temp_weights[i][index] += 0.0001
            _loss_plus = feed_forward(inputs, outputs, \
                                      temp_weights)
            grad = (_loss_plus - original_loss)/(0.0001)
            updated_weights[i][index] -= grad*lr
    return updated_weights, original_loss
```

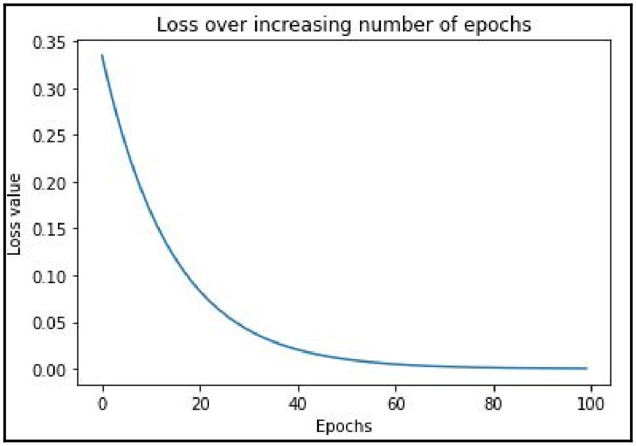
- Update weights over 100 epochs and fetch the loss value and the updated weight values:

```
losses = []
for epoch in range(100):
    W, loss = update_weights(x,y,W,0.01)
    losses.append(loss)
```

4. Plot the loss values:

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(losses)
plt.title('Loss over increasing number of epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss value')
```

The preceding code generates the following plot:



As you can see, the loss started at around 0.33 and steadily dropped to around 0.0001. This is an indication that weights are adjusted according to the input-output data and when an input is given, we can expect it to predict the output that we have been comparing it against in the loss function. The output weights are as follows:

```
[array([[ 0.01424004, -0.5907864 , -0.27549535],
        [ 0.39883757, -0.52918637, 0.18640439]], dtype=float32),
 array([ 0.00554004, 0.5519136 , -0.06599568], dtype=float32),
 array([[ 0.3475135 ],
        [-0.05529078],
        [ 0.03760847]], dtype=float32),
 array([-0.22443289], dtype=float32)]
```

The PyTorch version of the same code with the same weights is demonstrated in the GitHub notebook (`Auto_gradient_of_tensors.ipynb`). Revisit this section after understanding the core PyTorch concepts in the next chapter. Verify for yourself that the input and output are indeed the same whether the network is written in NumPy or PyTorch. Building a network from scratch using NumPy arrays, while sub-optimal, is done in this chapter to help you have a solid foundation of the working details of neural networks.

5. Once we have the updated weights, make the predictions for the input by passing the input through the network and calculate the output value:

```
pre_hidden = np.dot(x,W[0]) + W[1]
hidden = 1/(1+np.exp(-pre_hidden))
pred_out = np.dot(hidden, W[2]) + W[3]
# -0.017
```

The output of the preceding code is the value of `-0.017`, which is a value that is very close to the expected output of 0. As we train for more epochs, the `pred_out` value gets even closer to 0.

So far, we have learned about feedforward propagation and backpropagation. The key piece in the `update_weights` function that we defined here is the learning rate – which we will learn about in the next section.

Understanding the impact of the learning rate

In order to understand how learning rate impacts the training of a model, let's consider a very simple case, where we try to fit the following equation (note that the following equation is different from the toy dataset that we have been working on so far):

$$y = 3 \times x$$

Note that *y* is the output and *x* is the input. With a set of input and expected output values, we will try and fit the equation with varying learning rates to understand the impact of the learning rate.

> The following code is available as `Learning_rate.ipynb` in the `Chapter01` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. We specify the input and output dataset as follows:

```
x = [[1],[2],[3],[4]]
y = [[3],[6],[9],[12]]
```

2. Define the `feed_forward` function. Further, in this instance, we will modify the network in such a way that we do not have a hidden layer and the architecture is as follows:

$$y = w \times x + b$$

Note that, in the preceding function, we are estimating the parameters *w* and *b*:

```
from copy import deepcopy
import numpy as np
def feed_forward(inputs, outputs, weights):
    pred_out = np.dot(inputs,weights[0])+ weights[1]
    mean_squared_error = np.mean(np.square(pred_out \
                                            - outputs))
    return mean_squared_error
```

3. Define the `update_weights` function just like we defined it in the *Gradient descent in code* section:

```
def update_weights(inputs, outputs, weights, lr):
    original_weights = deepcopy(weights)
    org_loss = feed_forward(inputs, outputs,original_weights)
    updated_weights = deepcopy(weights)
    for i, layer in enumerate(original_weights):
        for index, weight in np.ndenumerate(layer):
            temp_weights = deepcopy(weights)
            temp_weights[i][index] += 0.0001
            _loss_plus = feed_forward(inputs, outputs, \
                                      temp_weights)
            grad = (_loss_plus - org_loss)/(0.0001)
            updated_weights[i][index] -= grad*lr
    return updated_weights
```

4. Initialize weight and bias values to a random value:

```
W = [np.array([[0]], dtype=np.float32),
     np.array([[0]], dtype=np.float32)]
```

Note that the weight and bias values are randomly initialized to values of 0. Further, the shape of the input weight value is 1 x 1, as the shape of each data point in the input is 1 x 1 and the shape of the bias value is 1 x 1 (as there is only one node in the output and each output has one value).

5. Let's leverage the `update_weights` function with a learning rate of 0.01, loop through 1,000 iterations, and check how the weight value (`W`) varies over increasing epochs:

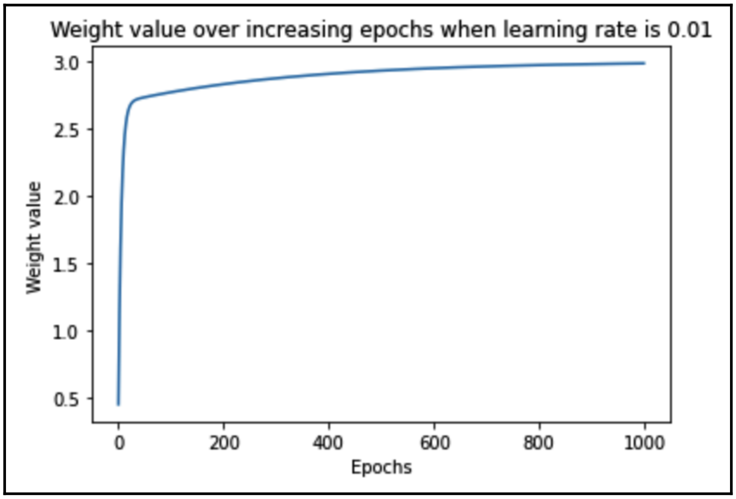
```
weight_value = []
for epx in range(1000):
    W = update_weights(x,y,W,0.01)
    weight_value.append(W[0][0][0])
```

Note that, in the preceding code, we are using a learning rate of 0.01 and repeating the `update_weights` function to fetch the modified weight at the end of each epoch. Further, in each epoch, we gave the most recent updated weight as an input to fetch the updated weight in the next epoch.

6. Plot the value of the weight parameter at the end of each epoch:

```
import matplotlib.pyplot as plt
%matplotlib inline
epochs = range(1, 1001)
plt.plot(epochs,weight_value)
plt.title('Weight value over increasing \
epochs when learning rate is 0.01')
plt.xlabel('Epochs')
plt.ylabel('Weight value')
```

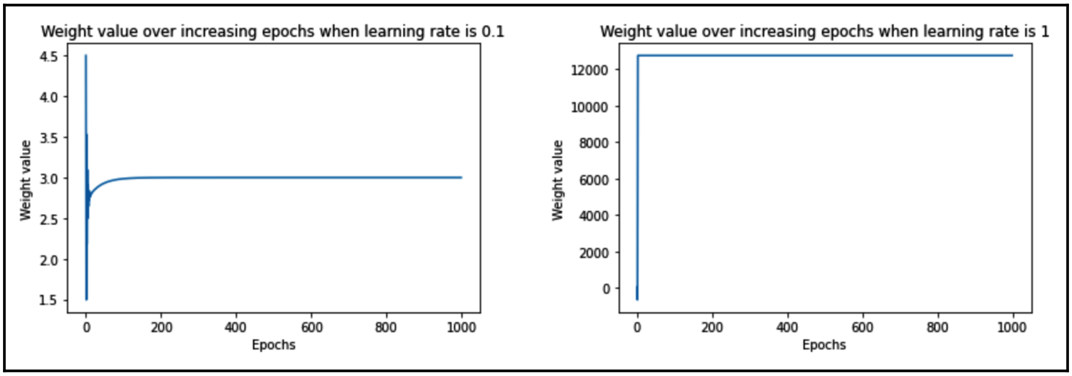
The preceding code results in a variation in the weight value over increasing epochs as follows:



Note that, in the preceding output, the weight value gradually increased in the right direction and then saturated at the optimal value of ~3.

In order to understand the impact of the value of the learning rate on arriving at the optimal weight values, let's understand how weight value varies over increasing epochs when the learning rate is 0.1 and when the learning rate is 1.

The following charts are obtained when we modify the corresponding learning rate value in step 5 and execute step 6 (the code to generate the following charts is the same as the code we learned earlier, with a change in the learning rate value, and is available in the associated notebook in GitHub):



Note that when the learning rate was very small (0.01), the weight value moved slowly (over a higher number of epochs) towards the optimal value. However, with a slightly higher learning rate (0.1), the weight value oscillated initially and then quickly saturated (in fewer epochs) to the optimal value. Finally, when the learning rate was high (1), the weight value spiked to a very high value and was not able to reach the optimal value.

The reason the weight value did not spike by a large amount when the learning rate was low is that we restricted the weight update by an amount that was equal to the *gradient * learning rate*, essentially resulting in a small amount of weight update when the learning rate was small. However, when the learning rate was high, weight update was high, after which the change in loss (when the weight was updated by a small value) was so small that the weight could not achieve the optimal value.

In order to have a deeper understanding of the interplay between the gradient value, learning rate, and weight value, let's run the `update_weights` function only for 10 epochs. Further, we will print the following values to understand how they vary over increasing epochs:

- Weight value at the start of each epoch
- Loss prior to weight update
- Loss when the weight is updated by a small amount
- Gradient value

We modify the `update_weights` function to print the preceding values as follows:

```
def update_weights(inputs, outputs, weights, lr):
    original_weights = deepcopy(weights)
    org_loss = feed_forward(inputs, outputs, original_weights)
    updated_weights = deepcopy(weights)
    for i, layer in enumerate(original_weights):
        for index, weight in np.ndenumerate(layer):
            temp_weights = deepcopy(weights)
            temp_weights[i][index] += 0.0001
            _loss_plus = feed_forward(inputs, outputs, \
                                      temp_weights)
            grad = (_loss_plus - org_loss)/(0.0001)
            updated_weights[i][index] -= grad*lr
            if(i % 2 == 0):
                print('weight value:', \
                      np.round(original_weights[i][index],2), \
                      'original loss:', np.round(org_loss,2), \
                      'loss_plus:', np.round(_loss_plus,2), \
                      'gradient:', np.round(grad,2), \
                      'updated_weights:', \
                      np.round(updated_weights[i][index],2))
    return updated_weights
```

The lines highlighted in bold font in the preceding code are where we modified the `update_weights` function from the previous section, where, first, we are checking whether we are currently working on the weight parameter by checking if (`i % 2 == 0`) as the other parameter corresponds to the bias value, and then we are printing the original weight value (`original_weights[i][index]`), loss (`org_loss`), updated loss value (`_loss_plus`), gradient (`grad`), and the resulting updated weight value (`updated_weights`).

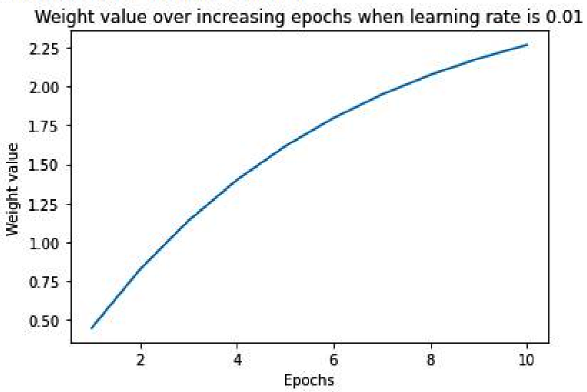
Let's now understand how the preceding values vary over increasing epochs across the three different learning rates that we are considering:

- **Learning rate of 0.01**: We will check the values using the following code:

```
W = [np.array([[0]], dtype=np.float32),
     np.array([[0]], dtype=np.float32)]
weight_value = []
for epx in range(10):
    W = update_weights(x,y,W,0.01)
    weight_value.append(W[0][0][0])
print(W)
import matplotlib.pyplot as plt
%matplotlib inline
epochs = range(1, 11)
plt.plot(epochs,weight_value)
plt.title('Weight value over increasing \
epochs when learning rate is 0.01')
plt.xlabel('Epochs')
plt.ylabel('Weight value')
```

The preceding code results in the following output:

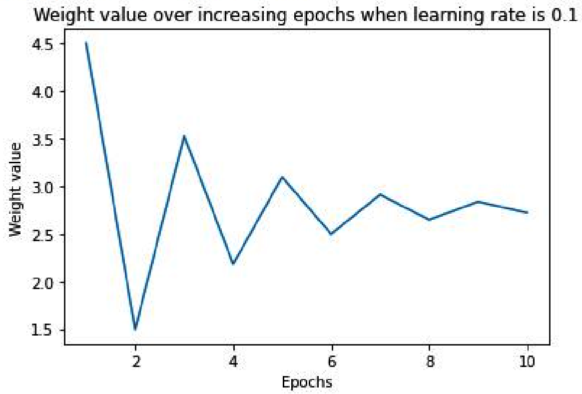
```
weight value: 0.0 original loss: 67.5 loss_plus: 67.5 gradient: -45.0 updated_weights: 0.45
weight value: 0.45 original loss: 46.88 loss_plus: 46.88 gradient: -37.49 updated_weights: 0.82
weight value: 0.82 original loss: 32.57 loss_plus: 32.57 gradient: -31.26 updated_weights: 1.14
weight value: 1.14 original loss: 22.64 loss_plus: 22.64 gradient: -26.05 updated_weights: 1.4
weight value: 1.4 original loss: 15.75 loss_plus: 15.75 gradient: -21.72 updated_weights: 1.62
weight value: 1.62 original loss: 10.97 loss_plus: 10.97 gradient: -18.1 updated_weights: 1.8
weight value: 1.8 original loss: 7.65 loss_plus: 7.65 gradient: -15.09 updated_weights: 1.95
weight value: 1.95 original loss: 5.35 loss_plus: 5.35 gradient: -12.59 updated_weights: 2.07
weight value: 2.07 original loss: 3.75 loss_plus: 3.75 gradient: -10.49 updated_weights: 2.18
weight value: 2.18 original loss: 2.64 loss_plus: 2.64 gradient: -8.75 updated_weights: 2.27
[array([[2.265477]], dtype=float32), array([[0.7404298]], dtype=float32)]
Text(0, 0.5, 'Weight value')
```



Note that, when the learning rate was 0.01, the loss value decreased slowly, and also the weight value updated slowly towards the optimal value. Let's now understand how the preceding varies when the learning rate is 0.1.

- **Learning rate of 0.1:** The code remains the same as in the learning rate of 0.01 scenario, however, the learning rate parameter would be 0.1 in this scenario. The output of running the same code with the changed learning rate parameter value is as follows:

```
weight value: 0.0 original loss: 67.5 loss_plus: 67.5 gradient: -45.0 updated_weights: 4.5
weight value: 4.5 original loss: 30.37 loss_plus: 30.38 gradient: 30.04 updated_weights: 1.5
weight value: 1.5 original loss: 13.79 loss_plus: 13.78 gradient: -20.31 updated_weights: 3.53
weight value: 3.53 original loss: 6.25 loss_plus: 6.26 gradient: 13.46 updated_weights: 2.18
weight value: 2.18 original loss: 2.85 loss_plus: 2.85 gradient: -9.14 updated_weights: 3.1
weight value: 3.1 original loss: 1.33 loss_plus: 1.33 gradient: 5.97 updated_weights: 2.5
weight value: 2.5 original loss: 0.65 loss_plus: 0.65 gradient: -4.12 updated_weights: 2.91
weight value: 2.91 original loss: 0.34 loss_plus: 0.34 gradient: 2.63 updated_weights: 2.65
weight value: 2.65 original loss: 0.2 loss_plus: 0.2 gradient: -1.88 updated_weights: 2.84
weight value: 2.84 original loss: 0.13 loss_plus: 0.13 gradient: 1.14 updated_weights: 2.72
[array([[2.7217765]], dtype=float32), array([[0.6589097]], dtype=float32)]
Text(0, 0.5, 'Weight value')
```



Let's contrast the learning rate scenarios of 0.01 and 0.1 – the major difference between the two is as follows:

When the learning rate was 0.01, the weight updated much slower when compared to a learning rate of 0.1 (from 0 to 0.45 in the first epoch when the learning rate is 0.01, to 4.5 when the learning rate is 0.1). The reason for the slower update is the lower learning rate as the weight is updated by the gradient times the learning rate.

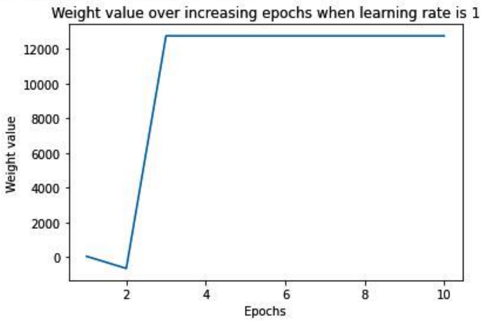
In addition to the weight update magnitude, we should note the direction of the weight update:

The gradient is negative when the weight value is smaller than the optimal value while it is positive when the weight value is larger than the optimal value. This phenomenon helps in updating weight values in the right direction.

Finally, we will contrast the preceding with a learning rate of 1:

- **Learning rate of 1**: The code remains the same as in the learning rate of 0.01 scenario, however, the learning rate parameter would be 1 in this scenario. The output of running the same code with the changed learning rate parameter is as follows:

```
weight value: 0.0 original loss: 67.5 loss_plus: 67.5 gradient: -45.0 updated_weights: 45.0
weight value: 45.0 original loss: 16604.45 loss_plus: 16604.52 gradient: 699.22 updated_weights: -654.22
weight value: -654.22 original loss: 4030779.1 loss_plus: 4030777.76 gradient: -13409.38 updated_weights: 12755.16
weight value: 12755.16 original loss: 1473935827.59 loss_plus: 1473935827.59 gradient: 0.0 updated_weights: 12755.16
weight value: 12755.16 original loss: 1473935827.59 loss_plus: 1473935827.59 gradient: 0.0 updated_weights: 12755.16
weight value: 12755.16 original loss: 1473935827.59 loss_plus: 1473935827.59 gradient: 0.0 updated_weights: 12755.16
weight value: 12755.16 original loss: 1473935827.59 loss_plus: 1473935827.59 gradient: 0.0 updated_weights: 12755.16
weight value: 12755.16 original loss: 1473935827.59 loss_plus: 1473935827.59 gradient: 0.0 updated_weights: 12755.16
weight value: 12755.16 original loss: 1473935827.59 loss_plus: 1473935827.59 gradient: 0.0 updated_weights: 12755.16
weight value: 12755.16 original loss: 1473935827.59 loss_plus: 1473935827.59 gradient: 0.0 updated_weights: 12755.16
[array([[12755.16]], dtype=float32), array([[3765.9717]], dtype=float32)]
Text(0, 0.5, 'Weight value')
```



From the preceding diagram, we can see that the weight has deviated to a very high value (as at the end of the first epoch, the weight value is 45, which further deviated to a very large value in later epochs). In addition to that, the weight value moved to a very large amount, so that a small change in the weight value hardly results in a change in the gradient, and hence the weight got stuck at that high value.

TIP

In general, it is better to have a low learning rate. This way, the model is able to learn slowly but will adjust the weights towards an optimal value. Typical learning rate parameter values range between 0.0001 and 0.01.

Now that we have learned about the building blocks of a neural network – feedforward propagation, backpropagation, and learning rate, in the next section, we will summarize a high-level overview of how these three are put together to train a neural network.

Summarizing the training process of a neural network

Training a neural network is a process of coming up with optimal weights for a neural network architecture by repeating the two key steps, forward-propagation and backpropagation with a given learning rate.

In forward-propagation, we apply a set of weights to the input data, pass it through the defined hidden layers, perform the defined nonlinear activation on the hidden layers' output, and then connect the hidden layer to the output layer by multiplying the hidden-layer node values with another set of weights to estimate the output value. Then, we finally calculate the overall loss corresponding to the given set of weights. For the first forward-propagation, the values of the weights are initialized randomly.

In backpropagation, we decrease the loss value (error) by adjusting weights in a direction that reduces the overall loss. Further, the magnitude of the weight update is the gradient times the learning rate.

The process of feedforward propagation and backpropagation is repeated until we achieve as minimal a loss as possible. This implies that, at the end of the training, the neural network has adjusted its weights θ such that it predicts the output that we want it to predict. In the preceding toy example, after training, the updated network will predict a value of 0 as output when *{1,1}* is fed as input as it is trained to achieve that.

Summary

In this chapter, we understood the need for a single network that performs both feature extraction and classification in a single shot, before we learned about the architecture and the various components of an artificial neural network. Next, we learned about how to connect the various layers of a network before implementing feedforward propagation to calculate the loss value corresponding to the current weights of the network. We next implemented backpropagation to learn about the way to optimize weights to minimize the loss value. Further, we learned about how the learning rate plays a role in achieving optimal weights for a network. In addition, we implemented all the components of a network – feedforward propagation, activation functions, loss functions, the chain rule, and gradient descent to update weights in NumPy from scratch so that we have a solid foundation to build upon in the next chapters.

Now that we understand how a neural network works, we'll implement one using PyTorch in the next chapter, and dive deep into the various other components (hyperparameters) that can be tweaked in a neural network in the third chapter.

Questions

1. What are the various layers in a neural network?
2. What is the output of feedforward propagation?
3. How is the loss function of a continuous dependent variable different from that of a binary dependent variable and also of a categorical dependent variable?
4. What is stochastic gradient descent?
5. What does a backpropagation exercise do?
6. How does a weight update of all the weights across layers happen during backpropagation?
7. Which functions of a neural network happen within each epoch of training a neural network?
8. Why is training a network on a GPU faster when compared to training it on a CPU?
9. How does the learning rate impact training a neural network?
10. What is the typical value of the learning rate parameter?

2
PyTorch Fundamentals

In the previous chapter, we learned about the fundamental building blocks of a neural network and also implemented forward and back-propagation from scratch in Python.

In this chapter, we will dive into the foundations of building a neural network using PyTorch, which we will leverage multiple times in subsequent chapters when we learn about various use cases in image analysis. We will start by learning about the core data type that PyTorch works on – tensor objects. We will then dive deep into the various operations that can be performed on tensor objects and how to leverage them when building a neural network model on top of a toy dataset (so that we strengthen our understanding before we gradually look at more realistic datasets, starting with the next chapter). This will allow us so to gain an intuition of how to build neural network models using PyTorch to map input and output values. Finally, we will learn about implementing custom loss functions so that we can customize based on the use case we are solving.

Specifically, this chapter will cover the following topics:

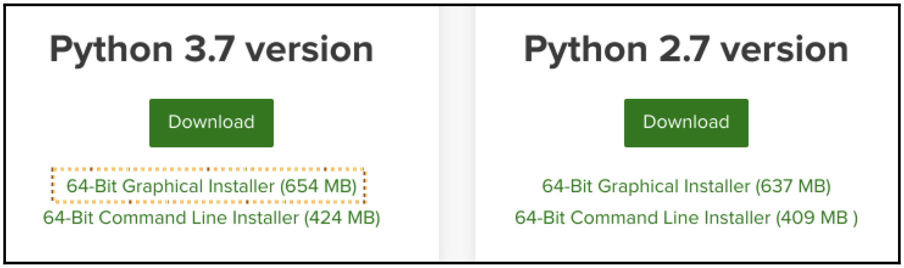
- Installing PyTorch
- PyTorch tensors
- Building a neural network using PyTorch
- Using a sequential method to build a neural network
- Saving and loading a PyTorch model

Installing PyTorch

PyTorch provides multiple functionalities that aid in building a neural network – abstracting the various components using high-level methods and also providing us with tensor objects that leverage GPUs to train a neural network faster.

Before installing PyTorch, we first need to install Python, as follows:

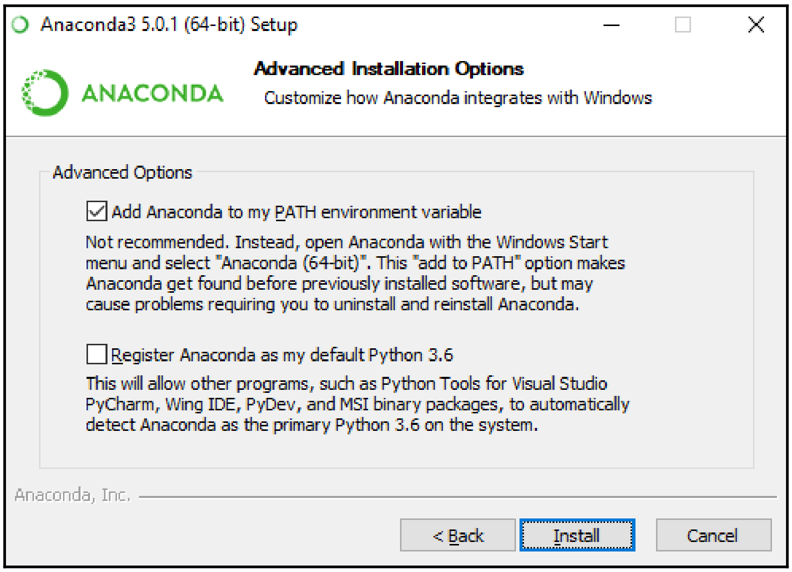
1. To install Python, we'll use the `anaconda.com/distribution/` platform to fetch an installer that will install Python as well as important deep learning-specific libraries for us automatically:



Choose the graphical installer of the latest Python version 3.xx (3.7, as of the time of writing this book) and let it download.

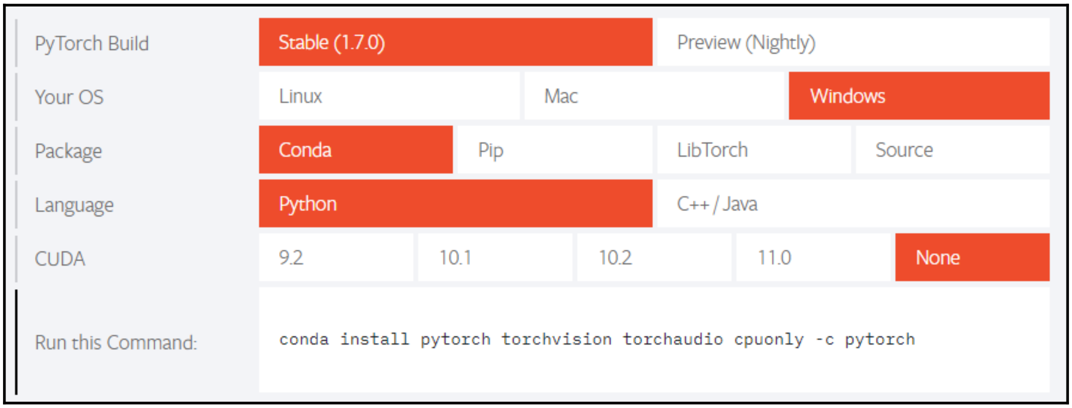
2. Install it using the downloaded installer:

Choose the **Add Anaconda to my PATH environment variable** option during installation as this will make it easy to invoke Anaconda's version of Python when we type `python` in Command Prompt/Terminal.



Next, we'll install PyTorch, which is equally simple.

3. Visit the **QUICK START LOCALLY** section on the `https://pytorch.org/` website and choose your operating system (**Your OS**), **Conda** for **Package**, **Python** for **Language**, and **None** for **CUDA**. If you have CUDA libraries, you may choose the appropriate version:



This will prompt you to run a command such as `conda install pytorch torchvision cpuonly -c pytorch` in your terminal.

4. Run the command in Command Prompt/Terminal and let Anaconda install PyTorch and the necessary dependencies.

If you own an NVIDIA graphics card as a hardware component, it is highly recommended to install CUDA drivers, which accelerate deep learning training by orders of magnitude. Do refer to the *Appendix* for instructions on how to install CUDA drivers. Once you have them installed, you can select **10.1** as the CUDA version and use that command instead to install PyTorch.

5. You can execute `python` in Command Prompt/Terminal and then type the following to verify that PyTorch is indeed installed:

```
>>> import torch
>>> print(torch.__version__)
# '1.7.0'
```

> **TIP** All the code in this book can be executed in Google Colab – `https://colab.research.google.com/`. Python and PyTorch are available by default in Google Colab. We highly encourage you to execute all code on Colab – which includes access to the GPU too, for free! Thanks to Google for providing such an excellent resource!

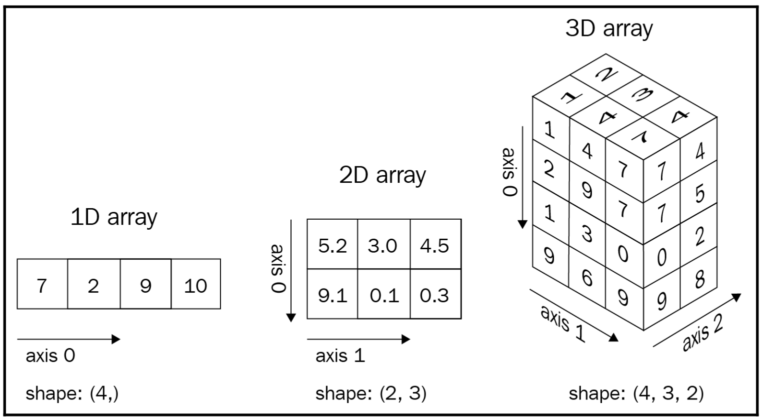
So, we have successfully installed Python and PyTorch. We will now perform some basic tensor operations in Python to help you get the hang of it.

PyTorch tensors

Tensors are the fundamental data types of PyTorch. A tensor is a multi-dimensional matrix similar to NumPy's ndarrays:

- A scalar can be represented as a zero-dimensional tensor.
- A vector can be represented as a one-dimensional tensor.
- A two-dimensional matrix can be represented as a two-dimensional tensor.
- A multi-dimensional matrix can be represented as a multi-dimensional tensor.

Pictorially, the tensors look as follows:



For instance, we can consider a color image as a three-dimensional tensor of pixel values, since a color image consists of `height x width x 3` pixels – where the three channels correspond to the RGB channels. Similarly, a grayscale image can be considered a two-dimensional tensor as it consists of `height x width` pixels.

By the end of this section, we will learn why tensors are useful and how to initialize them, as well as perform various operations on top of tensors. This will serve as a base for when we study leveraging tensors to build a neural network model in the following section.

Initializing a tensor

Tensors are useful in multiple ways. Apart from using them as base data structures for images, one more prominent use for them is when tensors are leveraged to initialize the weights connecting different layers of a neural network.

In this section, we will practice the different ways of initializing a tensor object:

The following code is available as `Initializing_a_tensor.ipynb` in the `Chapter02` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Import PyTorch and initialize a tensor by calling `torch.tensor` on a list:

```
import torch
x = torch.tensor([[1,2]])
y = torch.tensor([[1],[2]])
```

2. Next, access the tensor object's shape and data type:

```
print(x.shape)
# torch.Size([1,2]) # one entity of two items
print(y.shape)
# torch.Size([2,1]) # two entities of one item each
print(x.dtype)
# torch.int64
```

The data type of all elements within a tensor is the same. That means if a tensor contains data of different data types (such as a Boolean, an integer, and a float), the entire tensor is coerced to the most generic data type:

```
x = torch.tensor([False, 1, 2.0])
print(x)
# tensor([0., 1., 2.])
```

As you can see in the output of the preceding code, `False`, which was a Boolean, and `1`, which was an integer, were converted into floating-point numbers.

Alternatively, similar to NumPy, we can initialize tensor objects using built-in functions. Note that the parallels that we drew between tensors and weights of a neural network come to light now – where we are initializing tensors so that they represent the weight initialization of a neural network.

3. Generate a tensor object that has three rows and four columns filled with zeros:

```
torch.zeros((3, 4))
```

4. Generate a tensor object that has three rows and four columns filled with ones:

```
torch.ones((3, 4))
```

5. Generate three rows and four columns of values between 0 and 10 (including the low value but not including the high value):

```
torch.randint(low=0, high=10, size=(3,4))
```

6. Generate random numbers between 0 and 1 with three rows and four columns:

```
torch.rand(3, 4)
```

7. Generate numbers that follow a normal distribution with three rows and four columns:

```
torch.randn((3,4))
```

8. Finally, we can directly convert a NumPy array into a Torch tensor using `torch.tensor(<numpy-array>)`:

```
x = np.array([[10,20,30],[2,3,4]])
y = torch.tensor(x)
print(type(x), type(y))
# <class 'numpy.ndarray'> <class 'torch.Tensor'>
```

Now that we have learned about initializing tensor objects, we will learn about performing various matrix operations on top of them in the next section.

Operations on tensors

Similar to NumPy, you can perform various basic operations on tensor objects. Parallels to neural network operations are the matrix multiplication of input with weights, the addition of bias terms, and reshaping input or weight values when required. Each of these and additional operations are done as follows:

> The following code is available as `Operations_on_tensors.ipynb` in the `Chapter02` folder of this book's GitHub repository.

- Multiplication of all the elements present in `x` by `10` can be performed using the following code:

```
import torch
x = torch.tensor([[1,2,3,4], [5,6,7,8]])
print(x * 10)
# tensor([[10, 20, 30, 40],
#        [50, 60, 70, 80]])
```

- Adding `10` to the elements in `x` and storing the resulting tensor in `y` can be performed using the following code:

```
x = torch.tensor([[1,2,3,4], [5,6,7,8]])
y = x.add(10)
print(y)
# tensor([[11, 12, 13, 14],
#         [15, 16, 17, 18]])
```

- Reshaping a tensor can be performed using the following code:

```
y = torch.tensor([2, 3, 1, 0])
# y.shape == (4)
y = y.view(4,1)
# y.shape == (4, 1)
```

- Another way to reshape a tensor is by using the `squeeze` method, where we provide the axis index that we want to remove. Note that this is applicable only when the axis we want to remove has only one item in that dimension:

```
x = torch.randn(10,1,10)
z1 = torch.squeeze(x, 1) # similar to np.squeeze()
```

```
# The same operation can be directly performed on
# x by calling squeeze and the dimension to squeeze out
z2 = x.squeeze(1)
assert torch.all(z1 == z2)
# all the elements in both tensors are equal
print('Squeeze:\n', x.shape, z1.shape)

# Squeeze: torch.Size([10, 1, 10]) torch.Size([10, 10])
```

- The opposite of `squeeze` is `unsqueeze`, which means we add a dimension to the matrix, which can be performed using the following code:

```
x = torch.randn(10,10)
print(x.shape)
# torch.size(10,10)
z1 = x.unsqueeze(0)
print(z1.shape)

# torch.size(1,10,10)

# The same can be achieved using [None] indexing
# Adding None will auto create a fake dim
# at the specified axis
x = torch.randn(10,10)
z2, z3, z4 = x[None], x[:,None], x[:,:,None]
print(z2.shape, z3.shape, z4.shape)

# torch.Size([1, 10, 10])
# torch.Size([10, 1, 10])
# torch.Size([10, 10, 1])
```

Using `None` for indexing is a fancy way of unsqueezing, as shown, and will be used often in this book for creating fake channel/batch dimensions.

- Matrix multiplication of two different tensors can be performed using the following code:

```
x = torch.tensor([[1,2,3,4], [5,6,7,8]])
print(torch.matmul(x, y))

# tensor([[11],
#         [35]])
```

- Alternatively, matrix multiplication can also be performed by using the `@` operator:

```
print(x@y)

# tensor([[11],
#  [35]])
```

- Similar to `concatenate` in NumPy, we can perform concatenation of tensors using the `cat` method:

```
import torch
x = torch.randn(10,10,10)
z = torch.cat([x,x], axis=0) # np.concatenate()
print('Cat axis 0:', x.shape, z.shape)

# Cat axis 0:  torch.Size([10, 10, 10])
# torch.Size([20, 10, 10])
z = torch.cat([x,x], axis=1) # np.concatenate()
print('Cat axis 1:', x.shape, z.shape)

# Cat axis 1: torch.Size([10, 10, 10])
# torch.Size([10, 20, 10])
```

- Extraction of the maximum value in a tensor can be performed using the following code:

```
x = torch.arange(25).reshape(5,5)
print('Max:', x.shape, x.max())

# Max:  torch.Size([5, 5]) tensor(24)
```

- We can extract the maximum value along with the row index where the maximum value is present:

```
x.max(dim=0)

# torch.return_types.max(values=tensor([20, 21, 22, 23, 24]),
# indices=tensor([4, 4, 4, 4, 4]))
```

Note that, in the preceding output, we are fetching the maximum values across dimension `0`, which is the rows of the tensor. Hence, the maximum values across all rows are the values present in the 4th index and hence the `indices` output is all fours too. Furthermore, `.max` returns both the maximum values and the location (`argmax`) of the maximum values.

Similarly, the output when fetching the maximum value across columns is as follows:

```
m, argm = x.max(dim=1)
print('Max in axis 1:\n', m, argm)

# Max in axis 1: tensor([ 4, 9, 14, 19, 24])
# tensor([4, 4, 4, 4, 4])
```

The min operation is exactly the same as max but returns the minimum and arg-minimum where applicable.

- Permute the dimensions of a tensor object:

```
x = torch.randn(10,20,30)
z = x.permute(2,0,1) # np.permute()
print('Permute dimensions:', x.shape, z.shape)
# Permute dimensions:  torch.Size([10, 20, 30])
# torch.Size([30, 10, 20])
```

Note that the shape of the tensor changes when we perform permute on top of the original tensor.

TIP

Never reshape (that is, use tensor.view on) a tensor to swap the dimensions. Even though Torch will not throw an error, this is wrong and will create unforeseen results during training. If you need to swap dimensions, always use permute.

Since it is difficult to cover all the available operations in this book, it is important to know that you can do almost all NumPy operations in PyTorch with almost the same syntax as NumPy. Standard mathematical operations, such as abs, add, argsort, ceil, floor, sin, cos, tan, cumsum, cumprod, diag, eig, exp, log, log2, log10, mean, median, mode, resize, round, sigmoid, softmax, square, sqrt, svd, and transpose, to name a few, can be directly called on any tensor with or without axes where applicable. You can always run dir(torch.Tensor) to see all the methods possible for a Torch tensor and help(torch.Tensor.<method>) to go through the official help and documentation for that method.

Next, we will learn about leveraging tensors to perform gradient calculations on top of data – which is a key aspect of performing back-propagation in neural networks.

Auto gradients of tensor objects

As we saw in the previous chapter, differentiation and calculating gradients play a critical role in updating the weights of a neural network. PyTorch's tensor objects come with built-in functionality to calculate gradients.

In this section, we will understand how to calculate the gradients of a tensor object using PyTorch:

> The following code is available as `Auto_gradient_of_tensors.ipynb` in the `Chapter02` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Define a tensor object and also specify that it requires a gradient to be calculated:

```
import torch
x = torch.tensor([[2., -1.], [1., 1.]], requires_grad=True)
print(x)
```

In the preceding code, the `requires_grad` parameter specifies that the gradient is to be calculated for the tensor object.

2. Next, define the way to calculate the output, which in this specific case is the sum of the squares of all inputs:

$$out = \sum_{i=1}^{4} x_i^2$$

This is represented in code using the following line:

```
out = x.pow(2).sum()
```

We know that the gradient of the preceding function is *2*x*. Let's validate this using the built-in functions provided by PyTorch.

3. The gradient of a value can be calculated by calling the `backward()` method to the value. In our case, we calculate the gradient – change in `out` (output) for a small change in `x` (input) – as follows:

```
out.backward()
```

4. We are now in a position to obtain the gradient of `out` with respect to `x`, as follows:

```
x.grad
```

This results in the following output:

```
tensor([[ 4., -2.],
        [ 2.,  2.]])
```

Notice that the gradients obtained previously match with the intuitive gradient values (which are two times that of the value of x).

TIP

As an exercise, try recreating the scenario in `Chain rule.ipynb` in `Chapter 1`, *Artificial Neural Network Fundamentals*, with PyTorch. Compute the gradients after making a forward pass and make a single update. Verify that the updated weights match what we calculated in the notebook.

By now, we have learned about initializing, manipulating, and calculating gradients on top of a tensor object – which together constitute the fundamental building blocks of a neural network. Except for calculating auto gradients, initializing and manipulating data can also be performed using NumPy arrays. This calls for us to understand the reason why you should use tensor objects over NumPy arrays when building a neural network – which we will go through in the next section.

Advantages of PyTorch's tensors over NumPy's ndarrays

In the previous chapter, we saw that when calculating the optimal weight values, we vary each weight by a small amount and understand its impact on reducing the overall loss value. Note that the loss calculation based on the weight update of one weight does not impact the loss calculation of the weight update of other weights in the same iteration. Thus, this process can be optimized if each weight update is being made by a different core in parallel instead of updating weights sequentially. A GPU comes in handy in this scenario as it consists of thousands of cores when compared to a CPU (which, in general, could have <=64 cores).

A Torch tensor object is optimized to work with a GPU compared to NumPy. To understand this further, let's perform a small experiment, where we perform the operation of matrix multiplication using NumPy arrays in one scenario and tensor objects in another and compare the time taken to perform matrix multiplication in both scenarios:

> The following code is available as `Numpy_Vs_Torch_object_computation_speed_comparison.ipynb` in the `Chapter02` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Generate two different `torch` objects:

```
import torch
x = torch.rand(1, 6400)
y = torch.rand(6400, 5000)
```

2. Define the device to which we will store the tensor objects we created in *step 1*:

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

> Note that if you don't have a GPU device, the device will be `cpu` (furthermore, you would not notice the drastic difference in time taken to execute when using a CPU).

3. Register the tensor objects that were created in *step 1* with the device. Registering tensor objects means storing information in a device:

```
x, y = x.to(device), y.to(device)
```

4. Perform matrix multiplication of the Torch objects and also, time it so that we can compare the speed in a scenario where matrix multiplication is performed on NumPy arrays:

```
%timeit z=(x@y)
# It takes 0.515 milli seconds on an average to
# perform matrix multiplication
```

5. Perform matrix multiplication of the same tensors on `cpu`:

```
x, y = x.cpu(), y.cpu()
%timeit z=(x@y)
# It takes 9 milli seconds on an average to
# perform matrix multiplication
```

6. Perform the same matrix multiplication, this time on NumPy arrays:

```
import numpy as np
x = np.random.random((1, 6400))
y = np.random.random((6400, 5000))
%timeit z = np.matmul(x,y)
# It takes 19 milli seconds on an average to
# perform matrix multiplication
```

You will notice that the matrix multiplication performed on Torch objects on a GPU is ~18X faster than Torch objects on a CPU, and ~40X faster than the matrix multiplication performed on NumPy arrays. In general, `matmul` with Torch tensors on a CPU is still faster than NumPy. Note that you would notice this kind of speed up only if you have a GPU device. If you are working on a CPU device, you would not notice the dramatic increase in speed. This is why if you do not own a GPU, we recommend using Google Colab notebooks, as the service provides free GPUs.

Now that we have learned how tensor objects are leveraged across the various individual components/operations of a neural network and how using the GPU can speed up computation, in the next section, we will learn about putting this all together to build a neural network using PyTorch.

Building a neural network using PyTorch

In the previous chapter, we learned about building a neural network from scratch, where the components of a neural network are as follows:

- The number of hidden layers
- The number of units in a hidden layer
- Activation functions performed at the various layers
- The loss function that we try to optimize for
- The learning rate associated with the neural network
- The batch size of data leveraged to build the neural network
- The number of epochs of forward and back-propagation

However, for all of these, we built them from scratch using NumPy arrays in Python. In this section, we will learn about implementing all of these using PyTorch on a toy dataset. Note that we will leverage our learning so far regarding initializing tensor objects, performing various operations on top of them, and calculating the gradient values to update weights when building a neural network using PyTorch.

Note that, in this chapter, to gain the intuition of performing various operations, we will build a neural network on a toy dataset. Starting with the next chapter, we will deal with solving more realistic problems and datasets.

The toy problem we'll solve to understand the implementation of neural networks using PyTorch is a plain addition of two numbers, where we initialize the dataset as follows:

The following code is available as `Building_a_neural_network_using_PyTorch_on_a_toy_dataset.ipynb` in the `Chapter02` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Define the input (`x`) and output (`y`) values:

```
import torch
x = [[1,2],[3,4],[5,6],[7,8]]
y = [[3],[7],[11],[15]]
```

   Note that in the preceding input and output variable initialization, the input and output are a list of lists where the sum of values in the input list is the values in the output list.

2. Convert the input lists into tensor objects:

```
X = torch.tensor(x).float()
Y = torch.tensor(y).float()
```

   Note that in the preceding code, we have converted the tensor objects into floating-point objects. It is good practice to have tensor objects as floats or long ints, as they will be multiplied by decimal values (weights) anyway.

   Furthermore, we register the input (`X`) and output (`Y`) data points to the device – `cuda` if you have a GPU and `cpu` if you don't have a GPU:

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
X = X.to(device)
Y = Y.to(device)
```

3. Define the neural network architecture:

   - The `torch.nn` module contains functions that help in building neural network models:

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

   - We will create a class (`MyNeuralNet`) that can compose our neural network architecture. It is mandatory to inherit from `nn.Module` when creating a model architecture as it is the base class for all neural network modules:

```
class MyNeuralNet(nn.Module):
```

   - Within the class, we initialize all the components of a neural network using the `__init__` method. We should call `super().__init__()` to ensure that the class inherits `nn.Module`:

```
    def __init__(self):
        super().__init__()
```

   With the preceding code, by specifying `super().__init__()`, we are now able to take advantage of all the pre-built functionalities that have been written for `nn.Module`. The components that are going to be initialized in the `init` method will be used across different methods in the `MyNeuralNet` class.

   - Define the layers in the neural network:

```
        self.input_to_hidden_layer = nn.Linear(2,8)
        self.hidden_layer_activation = nn.ReLU()
        self.hidden_to_output_layer = nn.Linear(8,1)
```

   In the preceding lines of code, we specified all the layers of neural network – a linear layer (`self.input_to_hidden_layer`), followed by ReLU activation (`self.hidden_layer_activation`), and finally, a linear layer (`self.hidden_to_output_layer`). Note that, for now, the choice of the number of layers and activation is arbitrary. We'll learn about the impact of the number of units in layers and layer activations in more detail in the next chapter.

- Furthermore, let's understand what the functions in the preceding code are doing by printing the output of the `nn.Linear` method:

```
# NOTE - This line of code is not a part of model building,
# this is used only for illustration of Linear method
print(nn.Linear(2, 7))
Linear(in_features=2, out_features=7, bias=True)
```

In the preceding code, the linear method takes two values as input and outputs seven values, and also has a bias parameter associated with it. Furthermore, `nn.ReLU()` invokes the ReLU activation, which can then be used in other methods.

Some of the other commonly used activation functions are as follows:

- Sigmoid
- Softmax
- Tanh

Now that we have defined the components of a neural network, let's connect the components together while defining the forward propagation of the network:

```
def forward(self, x):
    x = self.input_to_hidden_layer(x)
    x = self.hidden_layer_activation(x)
    x = self.hidden_to_output_layer(x)
    return x
```

It is mandatory to use `forward` as the function name since PyTorch has reserved this function as the method for performing forward propagation. Using any other name in its place will raise an error.

By now, we have built the model architecture; let's inspect the randomly initialized weight values in the next step.

4. You can access the initial weights of each of the components by performing the following steps:

    - Create an instance of the `MyNeuralNet` class object that we defined earlier and register it to `device`:

```
mynet = MyNeuralNet().to(device)
```

- The weights and bias of each layer can be accessed by specifying the following:

```
# NOTE - This line of code is not a part of model building,
# this is used only for illustration of
# how to obtain parameters of a given layer
mynet.input_to_hidden_layer.weight
```

The output of the preceding code is as follows:

```
Parameter containing:
tensor([[ 0.5670,  0.2775],
        [-0.5525, -0.0506],
        [-0.1226, -0.0549],
        [-0.3667,  0.5775],
        [-0.2847, -0.7009],
        [-0.0449,  0.3303],
        [ 0.2479, -0.1501],
        [-0.4169, -0.0649]], requires_grad=True)
```

The values in your output will vary from the preceding, as the neural network is initialized with random values every time. If you wanted them to remain the same in multiple iterations of executing the same code, you would need to specify the seed using the `manual_seed` method in Torch as `torch.manual_seed(0)` just before creating the instance of the class object

- All the parameters of a neural network can be obtained by using the following code:

```
# NOTE - This line of code is not a part of model building,
# this is used only for illustration of
# how to obtain parameters of all layers in a model
mynet.parameters()
```

The preceding code returns a generator object.

- Finally, the parameters are obtained by looping through the generator, as follows:

```
# NOTE - This line of code is not a part of model building,
# this is used only for illustration of how to
# obtain parameters of all layers in a model
# by looping through the generator object
for par in mynet.parameters():
    print(par)
```

The preceding code results in the following output:

```
Parameter containing:
tensor([[ 0.5670,  0.2775],
        [-0.5525, -0.0506],
        [-0.1226, -0.0549],
        [-0.3667,  0.5775],
        [-0.2847, -0.7009],
        [-0.0449,  0.3303],
        [ 0.2479, -0.1501],
        [-0.4169, -0.0649]], requires_grad=True)
Parameter containing:
tensor([-0.7037,  0.4445, -0.4399,  0.6718,  0.2934, -0.6325,  0.2646, -0.5508],
       requires_grad=True)
Parameter containing:
tensor([[ 0.1219, -0.2936,  0.0820,  0.1212, -0.0885, -0.0113,  0.2657,  0.2921]],
       requires_grad=True)
Parameter containing:
tensor([0.0119], requires_grad=True)
```

The model has registered these tensors as special objects that are necessary for keeping track of forward and backward propagation. When defining any nn layers in the __init__ method, it will automatically create corresponding tensors and simultaneously register them. You can also manually register these parameters using the nn.Parameter(<tensor>) function. Hence, the following code is equivalent to the neural network class that we defined previously.

- An alternative way of defining the model using the nn.Parameter function is as follows:

```
# for illustration only
class MyNeuralNet(nn.Module):
     def __init__(self):
        super().__init__()
        self.input_to_hidden_layer = nn.Parameter(\
                                           torch.rand(2,8))
        self.hidden_layer_activation = nn.ReLU()
        self.hidden_to_output_layer = nn.Parameter(\
                                           torch.rand(8,1))

     def forward(self, x):
        x = x @ self.input_to_hidden_layer
        x = self.hidden_layer_activation(x)
        x = x @ self.hidden_to_output_layer
        return x
```

5. Define the loss function that we optimize for. Given that we are predicting for a continuous output, we'll optimize for mean squared error:

```
loss_func = nn.MSELoss()
```

The other prominent loss functions are as follows:

- CrossEntropyLoss (for multinomial classification)
- BCELoss (binary cross-entropy loss for binary classification)

- The loss value of a neural network can be calculated by passing the input values through the neuralnet object and then calculating MSELoss for the given inputs:

```
_Y = mynet(X)
loss_value = loss_func(_Y,Y)
print(loss_value)
# tensor(91.5550, grad_fn=<MseLossBackward>)
# Note that loss value can differ in your instance
# due to a different random weight initialization
```

In the preceding code, mynet(X) calculates the output values when the input is passed through the neural network. Furthermore, the loss_func function calculates the MSELoss value corresponding to the prediction of the neural network (_Y) and the actual values (Y).

As a convention, in this book, we will use **_**<variable> to associate prediction corresponding to the ground truth <variable>. Above this <variable> is Y.

Also note that when computing the loss, we *always* send the prediction first and then the ground truth. This is a PyTorch convention.

Now that we have defined the loss function, we will define the optimizer that tries to reduce the loss value. The input to the optimizer will be the parameters (weights and biases) corresponding to the neural network and the learning rate when updating the weights.

For this instance, we will consider the stochastic gradient descent (more on different optimizers and the impact of the learning rate in the next chapter).

6. Import the `SGD` method from the `torch.optim` module and then pass the neural network object (`mynet`) and learning rate (`lr`) as parameters to the `SGD` method:

```
from torch.optim import SGD
opt = SGD(mynet.parameters(), lr = 0.001)
```

7. Perform all the steps to be done in an epoch together:

   - Calculate the loss value corresponding to the given input and output.
   - Calculate the gradient corresponding to each parameter.
   - Update the weights based on the learning rate and gradient of each parameter.
   - Once the weights are updated, ensure that the gradients that have been calculated in the previous step are flushed before calculating the gradients in the next epoch:

```
# NOTE - This line of code is not a part of model building,
# this is used only for illustration of how we perform
opt.zero_grad() # flush the previous epoch's gradients
loss_value = loss_func(mynet(X),Y) # compute loss
loss_value.backward() # perform back-propagation
opt.step() # update the weights according to the gradients
computed
```

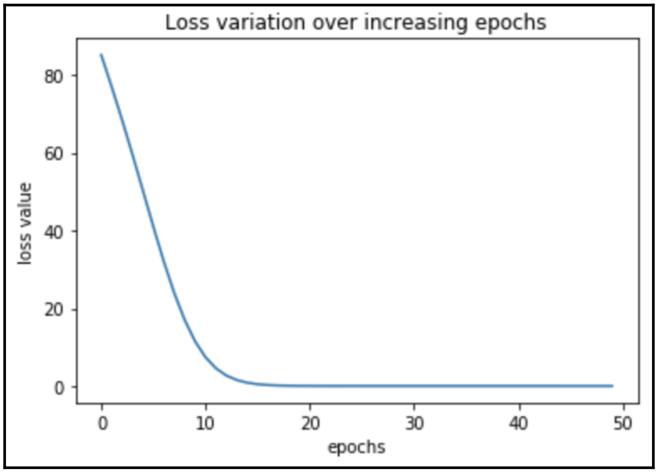
   - Repeat the preceding steps as many times as the number of epochs using a `for` loop. In the following example, we are performing the weight update process for a total of 50 epochs. Furthermore, we are storing the loss value in each epoch in the list – `loss_history`:

```
loss_history = []
for _ in range(50):
    opt.zero_grad()
    loss_value = loss_func(mynet(X),Y)
    loss_value.backward()
    opt.step()
    loss_history.append(loss_value)
```

- Let's plot the variation in loss over increasing epochs (as we saw in the previous chapter, we update weights in such a way that the overall loss value decreases with increasing epochs):

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(loss_history)
plt.title('Loss variation over increasing epochs')
plt.xlabel('epochs')
plt.ylabel('loss value')
```

The preceding code results in the following plot:



Note that, as expected, the loss value decreases over increasing epochs.

So far, in this section, we have updated the weights of a neural network by calculating the loss based on all the data points provided in the input dataset. In the next section, we will learn about the advantage of using only a sample of input data points per weight update.

Dataset, DataLoader, and batch size

One hyperparameter in a neural network that we have not considered yet is the batch size. Batch size refers to the number of data points considered to calculate the loss value or update weights.

This hyperparameter especially comes in handy in scenarios where there are millions of data points, and using all of them for one instance of weight update is not optimal, as memory is not available to hold so much information. In addition, a sample can be representative enough of the data. Batch size helps in fetching multiple samples of data that are representative enough, but not necessarily 100% representative of the total data.

In this section, we will come up with a way to specify the batch size to be considered when calculating the gradient of weights, to update weights, which is in turn used to calculate the updated loss value:

> The following code is available as `Specifying_batch_size_while_training_a_model.ipynb` in the `Chapter02` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Import the methods that help in loading data and dealing with datasets:

```
from torch.utils.data import Dataset, DataLoader
import torch
import torch.nn as nn
```

2. Import the data, convert the data into floating-point numbers, and register them to a device:

   - Provide the data points to work on:

```
x = [[1,2],[3,4],[5,6],[7,8]]
y = [[3],[7],[11],[15]]
```

   - Convert the data into floating-point numbers:

```
X = torch.tensor(x).float()
Y = torch.tensor(y).float()
```

   - Register data to the device – given that we are working on a GPU, we specify that the device is `'cuda'`. If you are working on a CPU, specify the device as `'cpu'`:

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
X = X.to(device)
Y = Y.to(device)
```

3. Instantiate a class of the dataset – `MyDataset`:

   ```
   class MyDataset(Dataset):
   ```

   Within the `MyDataset` class, we store the information to fetch one data point at a time so that a batch of data points can be bundled together (using `DataLoader`) and be sent through one forward and one back-propagation in order to update the weights:

   - Define an `__init__` method that takes input and output pairs and converts them into Torch float objects:

     ```
     def __init__(self,x,y):
         self.x = torch.tensor(x).float()
         self.y = torch.tensor(y).float()
     ```

   - Specify the length (`__len__`) of the input dataset:

     ```
     def __len__(self):
         return len(self.x)
     ```

   - Finally, the `__getitem__` method is used to fetch a specific row:

     ```
     def __getitem__(self, ix):
         return self.x[ix], self.y[ix]
     ```

   In the preceding code, `ix` refers to the index of the row that is to be fetched from the dataset.

4. Create an instance of the defined class:

   ```
   ds = MyDataset(X, Y)
   ```

5. Pass the dataset instance defined previously through `DataLoader` to fetch the `batch_size` number of data points from the original input and output tensor objects:

   ```
   dl = DataLoader(ds, batch_size=2, shuffle=True)
   ```

   In addition, in the preceding code, we also specify that we fetch a random sample (by mentioning that `shuffle=True`) of two data points (by mentioning `batch_size=2`) from the original input dataset (`ds`).

- To fetch the batches from `dl`, we loop through it:

```
# NOTE - This line of code is not a part of model building,
# this is used only for illustration of
# how to print the input and output batches of data
for x,y in dl:
    print(x,y)
```

This results in the following output:

```
tensor([[1., 2.],
        [3., 4.]]) tensor([[3.],
        [7.]])
tensor([[5., 6.],
        [7., 8.]]) tensor([[11.],
        [15.]])
```

Note that the preceding code resulted in two sets of input-output pairs as there were a total of four data points in the original dataset, while the batch size that was specified was `2`.

6. Now, we define the neural network class as we defined in the previous section:

```
class MyNeuralNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.input_to_hidden_layer = nn.Linear(2,8)
        self.hidden_layer_activation = nn.ReLU()
        self.hidden_to_output_layer = nn.Linear(8,1)
    def forward(self, x):
        x = self.input_to_hidden_layer(x)
        x = self.hidden_layer_activation(x)
        x = self.hidden_to_output_layer(x)
        return x
```

7. Next, we define the model object (`mynet`), loss function (`loss_func`), and optimizer (`opt`) too, as defined in the previous section:

```
mynet = MyNeuralNet().to(device)
loss_func = nn.MSELoss()
from torch.optim import SGD
opt = SGD(mynet.parameters(), lr = 0.001)
```

8. Finally, loop through the batches of data points to minimize the loss value, just like we did in *step 6* in the previous section:

```
import time
loss_history = []
start = time.time()
for _ in range(50):
    for data in dl:
        x, y = data
        opt.zero_grad()
        loss_value = loss_func(mynet(x),y)
        loss_value.backward()
        opt.step()
        loss_history.append(loss_value)
end = time.time()
print(end - start)
```

Note that while the preceding code seems very similar to the code that we went through in the previous section, we are performing 2X the number of weight updates per epoch when compared to the number of times the weights were updated in the previous section, as the batch size in this section is `2` whereas the batch size was `4` (the total number of data points) in the previous section.

Now that we have trained a model, in the next section, we will learn about predicting on a new set of data points.

Predicting on new data points

In the previous section, we learned how to fit a model on known data points. In this section, we will learn how to leverage the forward method defined in the trained `mynet` model from the previous section to predict on unseen data points. We will continue on from the code built in the previous section:

1. Create the data points that we want to test our model on:

```
val_x = [[10,11]]
```

Note that the new dataset (`val_x`) will also be a list of lists, as the input dataset was a list of lists.

2. Convert the new data points into a tensor float object and register to the device:

```
val_x = torch.tensor(val_x).float().to(device)
```

3. Pass the tensor object through the trained neural network – `mynet` – as if it were a Python function. This is the same as performing a forward propagation through the model that was built:

```
mynet(val_x)
# 20.99
```

The preceding code returns the predicted output values associated with the input data points.

By now, we have been able to train our neural network to map an input with output where we updated weight values by performing back-propagation to minimize the loss value (which is calculated using a pre-defined loss function).

In the next section, we will learn about building our own custom loss function instead of using a pre-defined loss function.

Implementing a custom loss function

In certain cases, we might have to implement a loss function that is customized to the problem we are solving – especially in complex use cases involving object detection/**generative adversial networks** (**GANs**). PyTorch provides the functionalities for us to build a custom loss function by writing a function of our own.

In this section, we will implement a custom loss function that does the same job as that of the `MSELoss` function that comes pre-built within `nn.Module`:

The following code is available as `Implementing_custom_loss_function.ipynb` in the `Chapter02` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Import the data, build the dataset and `DataLoader`, and define a neural network, as done in the previous section:

```
x = [[1,2],[3,4],[5,6],[7,8]]
y = [[3],[7],[11],[15]]
import torch
X = torch.tensor(x).float()
Y = torch.tensor(y).float()
import torch.nn as nn
device = 'cuda' if torch.cuda.is_available() else 'cpu'
X = X.to(device)
Y = Y.to(device)
```

```
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
class MyDataset(Dataset):
    def __init__(self,x,y):
        self.x = torch.tensor(x).float()
        self.y = torch.tensor(y).float()
    def __len__(self):
        return len(self.x)
    def __getitem__(self, ix):
        return self.x[ix], self.y[ix]
ds = MyDataset(X, Y)
dl = DataLoader(ds, batch_size=2, shuffle=True)
class MyNeuralNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.input_to_hidden_layer = nn.Linear(2,8)
        self.hidden_layer_activation = nn.ReLU()
        self.hidden_to_output_layer = nn.Linear(8,1)
    def forward(self, x):
        x = self.input_to_hidden_layer(x)
        x = self.hidden_layer_activation(x)
        x = self.hidden_to_output_layer(x)
        return x
mynet = MyNeuralNet().to(device)
```

2. Define the custom loss function by taking two tensor objects as input, take their difference, and square them up and return the mean value of the squared difference between the two:

```
def my_mean_squared_error(_y, y):
    loss = (_y-y)**2
    loss = loss.mean()
    return loss
```

3. For the same input and output combination that we had in the previous section, `nn.MSELoss` is used in fetching the mean squared error loss, as follows:

```
loss_func = nn.MSELoss()
loss_value = loss_func(mynet(X),Y)
print(loss_value)
# 92.7534
```

4. Similarly, the output of the loss value when we use the function that we defined in *step 2* is as follows:

```
my_mean_squared_error(mynet(X),Y)
# 92.7534
```

Notice that the results match. We have used the built-in `MSELoss` function and compared its result with the custom function that we built.

We can define a custom function of our choice, depending on the problem we are solving.

In the sections so far, we have learned about calculating the output at the last layer. The intermediate layer values have been a black box so far. In the next section, we will learn about fetching the intermediate layer values of a neural network.

Fetching the values of intermediate layers

In certain scenarios, it is helpful to fetch the intermediate layer values of the neural network (more on this when we discuss the style transfer and transfer learning use cases in later chapters).

PyTorch provides the functionality to fetch the intermediate values of the neural network in two ways:

The following code is available as `Fetching_values_of_intermediate_layers.ipynb` in the `Chapter02` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

- One way is by directly calling layers as if they are functions. This can be done as follows:

```
input_to_hidden = mynet.input_to_hidden_layer(X)
hidden_activation = mynet.hidden_layer_activation(\
                                        input_to_hidden)
print(hidden_activation)
```

Note that we had to call the `input_to_hidden_layer` activation prior to calling `hidden_layer_activation` as the output of `input_to_hidden_layer` is the input to the `hidden_layer_activation` layer.

- The other way is by specifying the layers that we want to look at in the `forward` method.

Let's look at the hidden layer values after activation for the model we have been working on in this chapter.

While all of the following code remains the same as what we saw in the previous section, we have ensured that the `forward` method returns not only the output but also the hidden layer values post-activation (`hidden2`):

```
class neuralnet(nn.Module):
    def __init__(self):
        super().__init__()
        self.input_to_hidden_layer = nn.Linear(2,8)
        self.hidden_layer_activation = nn.ReLU()
        self.hidden_to_output_layer = nn.Linear(8,1)
    def forward(self, x):
        hidden1 = self.input_to_hidden_layer(x)
        hidden2 = self.hidden_layer_activation(hidden1)
        output = self.hidden_to_output_layer(hidden2)
        return output, hidden2
```

We can now access the hidden layer values by specifying the following:

```
mynet = neuralnet().to(device)
mynet(X)[1]
```

Note that the 0^{th} index output of `mynet` is as we have defined – the final output of the forward propagation on the network – while the first index output is the hidden layer value post-activation.

So far, we have learned about implementing a neural network using the class of neural networks where we manually built each layer. However, unless we are building a complicated network, the steps to build a neural network architecture are straightforward, where we specify the layers and the sequence with which layers are to be stacked. In the next section, we will learn about a simpler way of defining neural network architecture.

Using a sequential method to build a neural network

So far, we have built a neural network by defining a class where we define the layers and how the layers are connected with each other. In this section, we will learn about a simplified way of defining the neural network architecture using the `Sequential` class. We will perform the same steps as we have done in the previous sections, except that the class that was used to define the neural network architecture manually will be substituted with a `Sequential` class for creating a neural network architecture.

Let's code up the network for the same toy data that we have worked on in this chapter:

> The following code is available as `Sequential_method_to_build_a_neural_network.ipynb` in the `Chapter02` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Define the toy dataset:

```
x = [[1,2],[3,4],[5,6],[7,8]]
y = [[3],[7],[11],[15]]
```

2. Import the relevant packages and define the device we will work on:

```
import torch
import torch.nn as nn
import numpy as np
from torch.utils.data import Dataset, DataLoader
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

3. Now, we define the dataset class (`MyDataset`):

```
class MyDataset(Dataset):
    def __init__(self, x, y):
        self.x = torch.tensor(x).float().to(device)
        self.y = torch.tensor(y).float().to(device)
    def __getitem__(self, ix):
        return self.x[ix], self.y[ix]
    def __len__(self):
        return len(self.x)
```

4. Define the dataset (`ds`) and dataloader (`dl`) object:

```
ds = MyDataset(x, y)
dl = DataLoader(ds, batch_size=2, shuffle=True)
```

5. Define the model architecture using the `Sequential` method available in the `nn` package:

```
model = nn.Sequential(
            nn.Linear(2, 8),
            nn.ReLU(),
            nn.Linear(8, 1)
        ).to(device)
```

Note that, in the preceding code, we defined the same architecture of the network as we defined in previous sections, but defined differently. `nn.Linear` accepts two-dimensional input and gives an eight-dimensional output for each data point. Furthermore, `nn.ReLU` performs ReLU activation on top of the eight-dimensional output and finally, the eight-dimensional input gives a one-dimensional output (which in our case is the output of the addition of the two inputs) using the final `nn.Linear` layer.

6. Print a summary of the model we defined in *step 5*:

    - Install and import the package that enables us to print the model summary:

```
!pip install torch_summary
from torchsummary import summary
```

    - Print a summary of the model, which expects the name of the model and also the input size of the model:

```
summary(model, torch.zeros(1,2))
```

The preceding code gives the following output:

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Linear: 1-1                            [-1, 8]                   24
├─ReLU: 1-2                              [-1, 8]                   --
├─Linear: 1-3                            [-1, 1]                   9
==========================================================================================
Total params: 33
Trainable params: 33
Non-trainable params: 0
Total mult-adds (M): 0.00
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.00
Estimated Total Size (MB): 0.00
==========================================================================================
```

Note that the output shape of the first layer is (-1, 8), where -1 represents that there can be as many data points as the batch size, and 8 represents that for each data point, we have an eight-dimensional output resulting in an output of the shape batch size x 8. The interpretation for the next two layers is similar.

7. Next, we define the loss function (`loss_func`) and optimizer (`opt`) and train the model, just like we did in the previous section. Note that, in this case, we need not define a model object; a network is not defined within a class in this scenario:

```
loss_func = nn.MSELoss()
from torch.optim import SGD
opt = SGD(model.parameters(), lr = 0.001)
import time
loss_history = []
start = time.time()
for _ in range(50):
    for ix, iy in dl:
        opt.zero_grad()
        loss_value = loss_func(model(ix),iy)
        loss_value.backward()
        opt.step()
        loss_history.append(loss_value)
end = time.time()
print(end - start)
```

8. Now that we have trained the model, we can predict values on a validation dataset that we define now:

   - Define the validation dataset:

   ```
   val = [[8,9],[10,11],[1.5,2.5]]
   ```

   - Predict the output of passing the validation list through the model (note that the expected value is the summation of the two inputs for each list within the list of lists). As defined in the dataset class, we first convert the list of lists into a float after converting them into a tensor object and registering them to the device:

   ```
   model(torch.tensor(val).float().to(device))
   # tensor([[16.9051], [20.8352], [ 4.0773]],
   # device='cuda:0', grad_fn=<AddmmBackward>)
   ```

Note that the output of the preceding code, as shown in the comment, is close to what is expected (which is the summation of the input values).

Now that we have learned about leveraging the sequential method to define and train a model, in the next section, we will learn about saving and loading a model to make an inference.

Saving and loading a PyTorch model

One of the important aspects of working on neural network models is to save and load back a model after training. Think of a scenario where you have to make inferences from an already-trained model. You would load the trained model instead of training it again.

> The following code is available as `save_and_load_pytorch_model.ipynb` in the `Chapter02` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

Before going through the relevant commands to do that, taking the preceding example as our case, let's understand what all the important components that completely define a neural network are. We need the following:

- A unique name (key) for each tensor (parameter)
- The logic to connect every tensor in the network with one or the other
- The values (weight/bias values) of each tensor

While the first point is taken care of during the `__init__` phase of a definition, the second point is taken care of during the `forward` method definition. By default, the values in a tensor are randomly initialized during the `__init__` phase. But what we want is to load a *specific* set of weights (or values) that were learned when training a model and associate each value with a specific name. This is what you obtain by calling a special method, described in the following sections.

state dict

The `model.state_dict()` command is at the root of understanding how saving and loading PyTorch models works. The dictionary in `model.state_dict()` corresponds to the parameter names (keys) and the values (weight and bias values) corresponding to the model. `state` refers to the current snapshot of the model (where the snapshot is the set of values at each tensor).

It returns a dictionary (`OrderedDict`) of keys and values:

```
model.state_dict()
```

```
OrderedDict([('0.weight', tensor([[ 0.5090,  0.6708],
                      [-0.5887, -0.2970],
                      [ 0.3078, -0.4445],
                      [-0.3859,  0.0028],
                      [-0.1816,  0.9181],
                      [ 0.1532,  0.6011],
                      [ 0.2814, -0.4834],
                      [-0.6280, -0.5868]])),
             ('0.bias',
              tensor([ 0.7432, -0.5181, -0.1400,  0.3236, -0.1791, -0.4466, -0.1104, -0.1615])),
             ('2.weight',
              tensor([[ 0.9044, -0.2407, -0.1512, -0.2253,  0.5417,  0.4821,  0.1548,  0.0964]])),
             ('2.bias', tensor([0.0956]))])
```

The keys are the names of the model's layers and the values correspond to the weights of these layers.

Saving

Running `torch.save(model.state_dict(), 'mymodel.pth')` will save this model in a Python serialized format on the disk with the name `mymodel.pth`. A good practice is to transfer the model to the CPU before calling `torch.save` as this will save tensors as CPU tensors and not as CUDA tensors. This will help in loading the model onto any machine, whether it contains CUDA capabilities or not.

We save the model using the following code:

```
torch.save(model.to('cpu').state_dict(), 'mymodel.pth')
```

Now that we understand saving a model, in the next section, we will learn about loading the model.

Loading

Loading a model would require us to initialize the model with random weights first and then load the weights from `state_dict`:

1. Create an empty model with the same command that was used in the first place when training:

```
model = nn.Sequential(
            nn.Linear(2, 8),
            nn.ReLU(),
            nn.Linear(8, 1)
        ).to(device)
```

2. Load the model from disk and unserialize it to create an `orderedDict` value:

```
state_dict = torch.load('mymodel.pth')
```

3. Load `state_dict` onto `model`, register to `device`, and make a prediction:

```
model.load_state_dict(state_dict)
# <All keys matched successfully>
model.to(device)
model(torch.tensor(val).float().to(device))
```

If all the weight names are present in the model, then you would get a message saying all the keys were matched. This implies we are able to load our model from disk, for all purposes, on any machine in the world.

Next, we can register the model to the device and perform inference on the new data points, as we learned in the previous section.

Summary

In this chapter, we learned about the building blocks of PyTorch – tensor objects and performing various operations on top of them. We proceeded further by building a neural network on a toy dataset where we started by building a class that initializes the feed-forward architecture, fetching data points from the dataset by specifying the batch size, and defining the loss function and the optimizer, looping through multiple epochs. Finally, we also learned about defining custom loss functions to optimize a metric of choice and leveraging the sequential method to simplify the process of defining the network architecture.

All the preceding steps form the foundation of building a neural network, which will be leveraged multiple times in the various use cases that we will build in subsequent chapters.

With this knowledge of the various components of building a neural network using PyTorch, we will proceed to the next chapter, where we will learn about the various practical aspects of dealing with the hyperparameters of a neural network on image datasets.

Questions

1. Why should we convert integer inputs into float values during training?
2. What are the various methods to reshape a tensor object?
3. Why is computation faster with tensor objects over NumPy arrays?
4. What constitutes the init magic function in a neural network class?
5. Why do we perform zero gradients before performing back-propagation?
6. What magic functions constitute the dataset class?
7. How do we make predictions on new data points?
8. How do we fetch the intermediate layer values of a neural network?
9. How does the sequential method help in simplifying defining the architecture of a neural network?

3
Building a Deep Neural Network with PyTorch

In the previous chapter, we learned how to code a neural network using PyTorch. We also learned about the various hyperparameters that are present in a neural network, such as its batch size, learning rate, and loss optimizer. In this chapter, we will shift gears and learn how to perform image classification using neural networks. Essentially, we will learn how to represent images and tweak the hyperparameters of a neural network to understand their impact.

For the sake of not introducing too much complexity and confusion, we only covered the fundamental aspects of neural networks in the previous chapter. However, there are many more inputs that we tweak in a network while training it. Typically, these inputs are known as **hyperparameters**. In contrast to the *parameters* in a neural network (which are learned during training), these inputs are hyperparameters that are provided by the person who is building the network. Changing different aspects of each hyperparameter is likely to affect the accuracy or speed of training a neural network. Furthermore, a few additional techniques such as scaling, batch normalization, and regularization help in improving the performance of a neural network. We will be learning about these concepts throughout this chapter.

However, before we get to that, we will learn about how an image is represented – only then will we do a deep dive into the details of hyperparameters. While learning about the impact of hyperparameters, we will restrict ourselves to one dataset – FashionMNIST – so that we can make a comparison of the impact of variation in various hyperparameters. Through this dataset, we will also be introduced to training and validation data and why it is important to have two separate datasets. Finally, we will learn about the concept of overfitting a neural network and then understand how certain hyperparameters help us avoid overfitting.

In summary, in this chapter, we will cover the following topics:

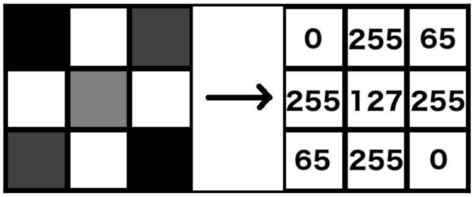
- Representing an image
- Why leverage neural networks for image analysis?
- Preparing data for image classification
- Training a neural network
- Scaling a dataset to improve model accuracy
- Understanding the impact of varying the batch size
- Understanding the impact of varying the loss optimizer
- Understanding the impact of varying the learning rate
- Understanding the impact of learning rate annealing
- Building a deeper neural network
- Understanding the impact of batch normalization
- The concept of overfitting

Let's get started!

Representing an image

A digital image file (typically associated with the extension "JPEG" or "PNG") is comprised of an array of pixels. A pixel is the smallest constituting element of an image. In a grayscale image, each pixel is a scalar (single) value between 0 and 255 – 0 is black, 255 is white, and anything in between is gray (the smaller the pixel value, the darker the pixel is). On the other hand, the pixels in color images are three-dimensional vectors that correspond to the scalar values that can be found in its red, green, and blue channels.

An image has *height x width x c* pixels, where *height* is the number of **rows** of pixels, *width* is the number of **columns** of pixels, and *c* is the number of **channels**. *c* is 3 for color images (one channel each for the *red, green,* and *blue* intensities of the image) and 1 for grayscale images. An example grayscale image containing 3 x 3 pixels and their corresponding scalar values is shown here:



Again, a pixel value of 0 means that it is pitch black, while 255 means it is pure luminance (tat is, pure white for grayscale and pure red/green/blue in the respective channel for a color image).

Converting images into structured arrays and scalars

Python can convert images into structured arrays and scalars as follows:

> The following code is available as `Inspecting_grayscale_images.ipynb` in the `Chapter03` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Download a sample image:

```
!wget https://www.dropbox.com/s/l98leemr7r5stnm/Hemanvi.jpeg
```

2. Import the `cv2` (for reading an image from disk) and `matplotlib` (for plotting the loaded image) libraries and read the downloaded image into the Python environment:

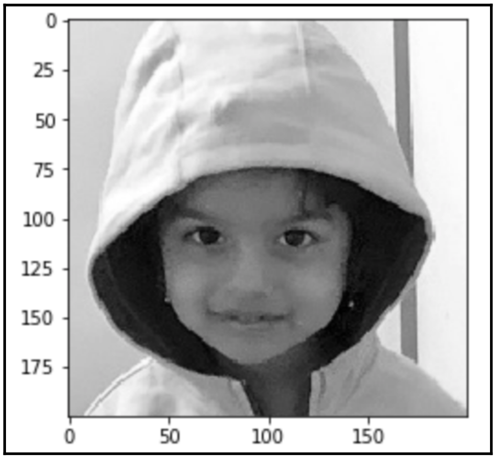
```
%matplotlib inline
import cv2, matplotlib.pyplot as plt
img = cv2.imread('Hemanvi.jpeg')
```

In the preceding line of code, we are leveraging the `cv2.imread` method to read the image. This converts an image into an array of pixel values.

3. We'll crop the image between 50th-250th rows, as well as 40th-240th columns. Finally, we'll convert the image into grayscale using the following code and plot it:

```
# Crop image
img = img[50:250,40:240]
# Convert image to grayscale
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# Show image
plt.imshow(img_gray, cmap='gray')
```

The output of the preceding sequence of steps is as follows:

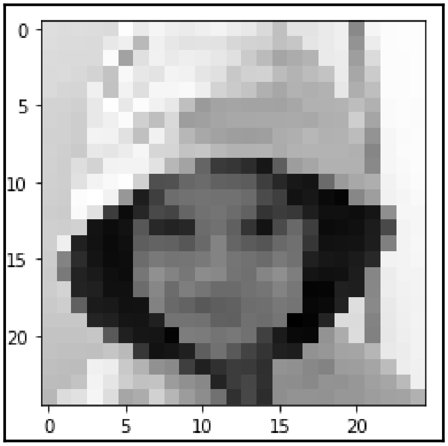


You might have noticed that the preceding image is represented as a 200 x 200 array of pixels. Now, let's reduce the number of pixels that are being used to represent the image so that we can overlay the pixel values on the image (this would be tougher to do if we were to visualize the pixel values over a 200 x 200 array compared to a 25 x 25 array).

4. Convert the image into a 25 x 25 array and plot it:

```
img_gray_small = cv2.resize(img_gray,(25,25))
plt.imshow(img_gray_small, cmap='gray')
```

This results in the following output:



Naturally, having fewer pixels to represent the same image results in a blurrier output.

5. Let's inspect the pixel values. Note that in the following output, due to space constraints, we have pasted only the first four rows of pixel values:

```
print(img_gray_small)
```

This results in the following output:

```
array([[222, 220, 221, 220, 218, 253, 234, 245, 238, 235, 239, 243, 236,
         232, 218, 193, 228, 228, 234, 239, 139, 245, 252, 253, 253],
       [221, 219, 219, 218, 232, 239, 186, 240, 231, 226, 227, 226, 215,
         212, 209, 193, 199, 229, 234, 239, 150, 236, 252, 253, 253],
       [219, 218, 218, 218, 251, 163, 224, 241, 234, 238, 236, 231, 224,
         204, 188, 166, 173, 180, 234, 236, 159, 219, 252, 252, 253],
       [218, 219, 216, 211, 196, 248, 231, 228, 243, 241, 229, 224, 201,
         209, 210, 189, 181, 189, 196, 235, 168, 204, 252, 252, 253],
```

The same set of pixel values, when copied and pasted into Excel and color-coded by pixel value, would look as follows:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y
1	222	220	221	220	218	253	234	245	238	235	239	243	236	232	218	193	228	228	234	239	139	245	252	253	253
2	221	219	219	218	232	239	186	240	231	226	227	226	215	212	209	193	199	229	234	239	150	236	252	253	253
3	219	218	218	218	251	163	224	241	234	238	236	231	224	204	188	166	173	180	234	236	159	219	252	252	253
4	218	219	216	211	196	248	231	228	243	241	229	224	201	209	210	189	181	189	196	235	168	204	252	252	253
5	218	214	213	240	195	242	223	246	246	249	238	211	203	196	177	168	179	176	179	231	175	191	252	252	253
6	212	212	208	232	254	232	252	241	232	192	155	164	166	165	164	163	168	178	178	181	190	178	250	252	251
7	211	209	205	232	240	251	208	191	217	158	161	166	169	169	170	170	171	169	176	177	206	166	250	252	251
8	209	209	205	243	242	225	193	241	215	184	169	163	159	158	160	173	176	184	178	179	189	150	246	250	252
9	210	207	203	232	229	236	246	214	213	196	199	185	179	179	181	172	179	180	180	181	177	136	246	251	252
10	209	206	222	212	242	243	244	226	184	165	104	61	57	48	27	97	158	167	178	178	178	139	246	249	252
11	208	206	225	243	249	254	209	82	85	105	109	100	98	95	65	43	28	24	109	156	169	175	242	248	251
12	208	205	252	255	242	153	33	66	111	116	117	116	115	109	78	66	22	27	14	9	137	159	241	245	249
13	205	204	250	225	63	15	42	77	71	104	115	118	110	101	56	64	60	34	20	20	17	25	145	246	246
14	208	206	209	23	16	22	90	45	39	43	110	115	99	56	23	78	107	65	15	17	20	32	76	244	246
15	208	239	37	22	14	19	97	102	100	90	108	133	104	94	88	108	114	57	21	22	23	33	130	243	246
16	205	133	48	24	15	17	110	124	118	119	124	134	119	116	109	123	116	36	27	25	31	44	242	243	246
17	204	124	38	30	19	16	120	146	133	121	142	138	118	114	135	145	128	24	20	21	33	136	236	243	244
18	205	212	39	37	20	20	110	137	110	128	119	109	109	115	119	133	121	8	9	36	34	137	237	241	243
19	204	206	101	31	29	21	22	132	108	102	91	97	101	111	113	122	118	11	14	38	222	139	233	240	242
20	200	200	200	41	36	25	24	37	117	117	116	101	99	114	111	119	4	8	41	220	219	134	232	239	244
21	197	196	196	199	92	37	26	25	8	127	125	118	122	116	67	31	13	11	150	173	220	131	231	238	242
22	195	193	193	192	198	187	58	22	25	37	97	115	93	70	55	36	33	148	153	165	166	183	233	236	242
23	192	190	189	240	237	202	180	147	140	66	36	52	64	61	51	150	146	138	134	157	159	166	189	237	240
24	189	188	197	244	229	206	194	196	157	146	138	39	63	73	53	144	143	139	150	148	153	161	167	187	239
25	184	220	225	245	221	167	173	209	183	157	143	116	53	74	49	144	150	150	148	153	153	158	162	165	178

As we mentioned previously, the pixels with a scalar value closer to 255 appear lighter, while those closer to 0 appear darker.

The preceding steps apply to color images too, which are represented as three-dimensional vectors. The brightest red pixel is denoted as (255,0,0). Similarly, a pure white pixel in a three-dimensional vector image is represented as (255,255,255). With this in mind, let's create a structured array of pixel values for a colored image:

The following code is available as `Inspecting_color_images.ipynb` in the `Chapter03` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Download a color image:

```
!wget https://www.dropbox.com/s/l98leemr7r5stnm/Hemanvi.jpeg
```

2. Import the relevant packages and load the image:

```
import cv2, matplotlib.pyplot as plt
%matplotlib inline
img = cv2.imread('Hemanvi.jpeg')
```

3. Crop the image:

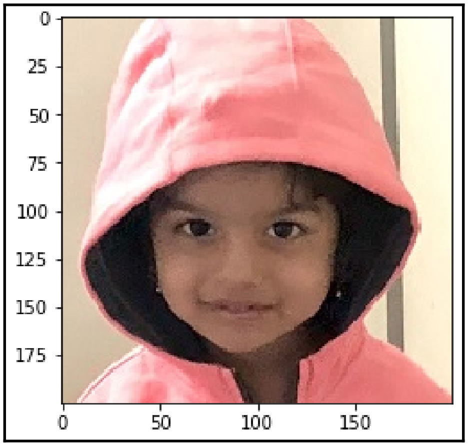
```
img = img[50:250,40:240,:]
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

Note that in the preceding code, we have reordered the channels using the `cv2.cvtcolor` method. We've done this because when we import images using cv2, the channels are ordered as Blue first, Green next, and finally Red; typically, we are used to looking at images in RGB channels, where the sequence is Red, Green, and then Blue.

4. Plot the image that's obtained (note that if you are reading the print book and haven't downloaded the color image bundle, it will appear in grayscale):

```
plt.imshow(img)
print(img.shape)
# (200,200,3)
```

This results in the following output:



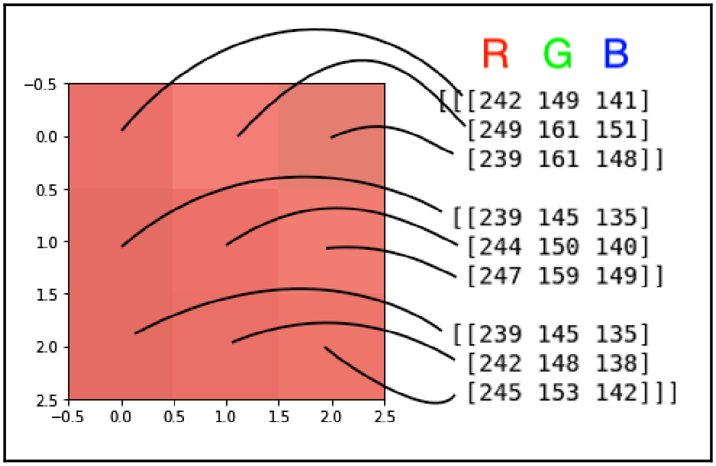
5. The bottom-right 3 x 3 array of pixels can be obtained as follows:

```
crop = img[-3:,-3:]
```

6. Print and plot the pixel values:

```
print(crop)
plt.imshow(crop)
```

The preceding code results in the following output:



Now that we can represent each image as an array of scalars (in the case of a grayscale image) or an array of arrays (in the case of a color image), we have essentially converted a file on disk into a structured array format that can now be processed mathematically using multiple techniques. Converting an image into a structured array of numbers (that is, reading an image into Python memory) enables us to perform mathematical operations on top of the images (which are represented as an array of numbers). We can leverage this data structure to perform various tasks such as classification, detection, and segmentation, all of which will be discussed in detail in later chapters.

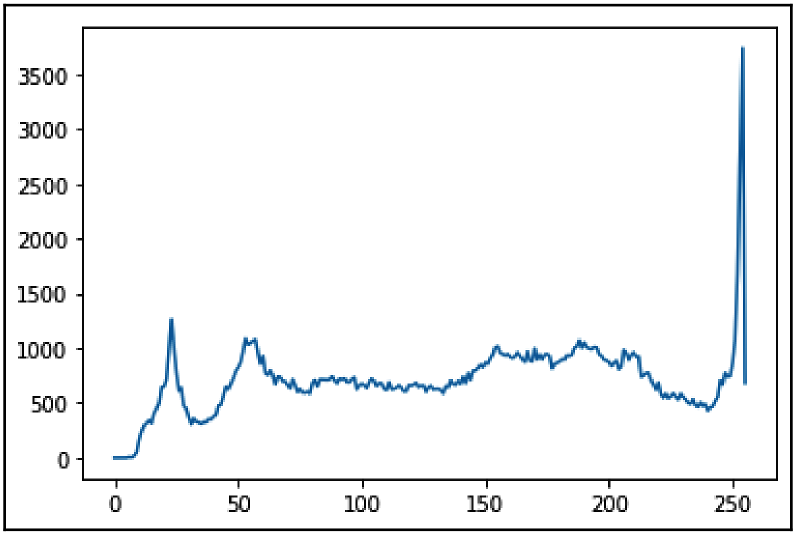
Now that we have an understanding of how images are represented, let's understand the reason for leveraging neural networks for image classification.

Why leverage neural networks for image analysis?

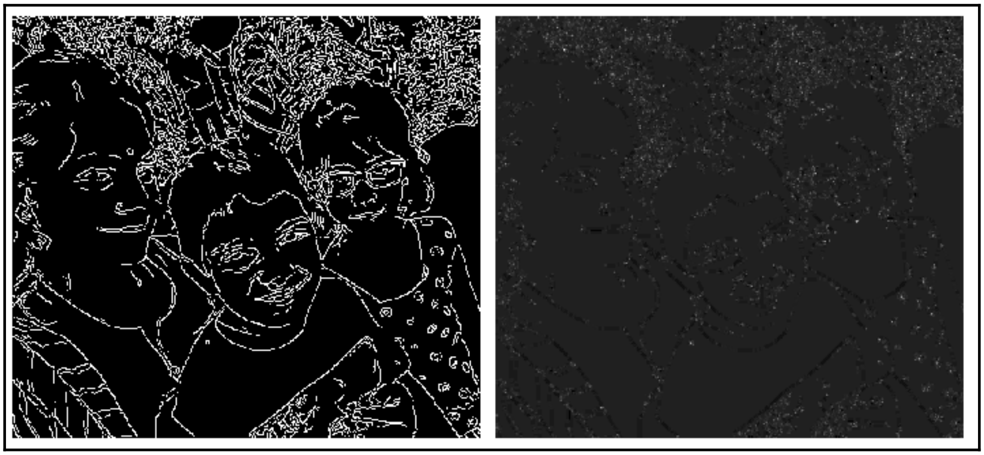
In traditional computer vision, we would create a few features for every image before using them as input. Let's take a look at a few such features based on the following sample image in order to appreciate the effort that we are avoiding going to by training a neural network:

Note that we will not walk you through how to get these features as the intention here is to help you realise why creating features manually is a sub-optimal exercise:

- **Histogram feature**: For some tasks, such as auto-brightness or night vision, it is important to understand the illumination in the picture; that is, the fraction of pixels that are bright or dark. The following graph shows a histogram for the example image. It depicts that the image is well illuminated since there is a spike at 255:

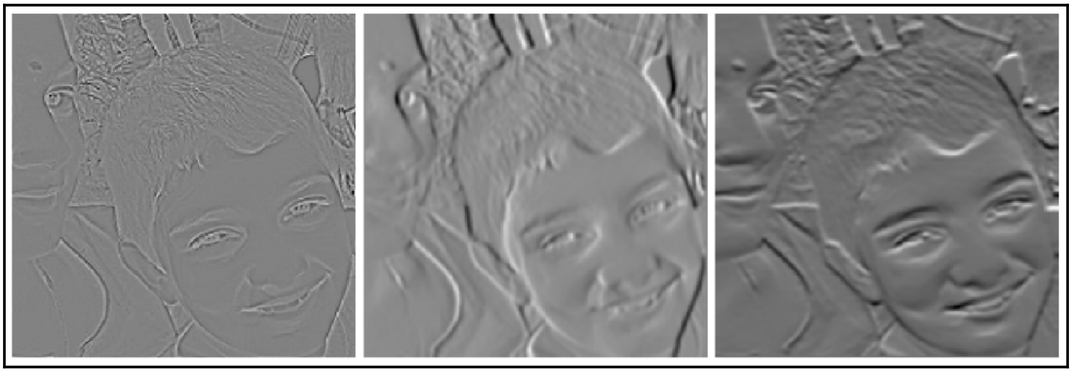


- **Edges and Corners feature**: For tasks such as image segmentation, where it is important to find the set of pixels corresponding to each person, it makes sense to extract the edges first because the border of a person is just a collection of edges. In other tasks, such as image registration, it is vital that key landmarks are detected. These landmarks will be a subset of all the corners in an image. The following image represents the edges and corners that can be found in our example image:



- **Color separation feature**: In tasks such as traffic light detection for a self-driving car, it is important that the system understands what color is being displayed on the traffic lights. The following image (best viewed in color) shows the exclusively red, green, and blue pixels for the example image:

- **Image gradients feature**: Taking this a step further, it might be important to understand how the colors are changing at the pixel level. Different textures can give us different gradients, which means they can be used as texture detectors. In fact, finding gradients is a prerequisite for edge detection. The following image shows the overall gradient, as well as the y and x components of the gradient, for a section of our example image:



These are just a handful of such features. There are so many more that it is difficult to cover all of them. The main drawback of creating these features is that you need to be an expert in image and signal analysis and should fully understand what features are best suited to solve a problem. Even if both constraints are satisfied, there is no guarantee that such an expert will be able to find the right combination of inputs, and even if they do, there is still no guarantee that such a combination will work in new, unseen scenarios.

Due to these drawbacks, the community has largely shifted to neural network-based models. These models not only find the right features automatically but also learn how to optimally combine them to get the job done. As we have already understood in the first chapter, neural networks act as both feature extractors and classifiers.

Now that we've had a look at some examples of historical feature extraction techniques and their drawbacks, let's learn how to train a neural network on images.

Preparing our data for image classification

Given that we are covering multiple scenarios in this chapter, in order for us to see the advantage of one scenario over the other, we will work on a single dataset throughout this chapter – the Fashion MNIST dataset. Let's prepare this dataset:

> The following code is available as `Preparing_our_data.ipynb` in the `Chapter03` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Start by downloading the dataset and importing the relevant packages. The `torchvision` package contains various datasets – one of which is the `FashionMNIST` dataset, which we will be working on in this chapter:

```
from torchvision import datasets
import torch
data_folder = '~/data/FMNIST' # This can be any directory
# you want to download FMNIST to
fmnist = datasets.FashionMNIST(data_folder, download=True, \
                                train=True)
```

In the preceding code, we are specifying the folder (`data_folder`) where we want to store the downloaded dataset. Next, we are fetching `fmnist` data from `datasets.FashionMNIST` and are storing it in `data_folder`. Furthermore, we are specifying that we only want to download the training images by specifying `train = True`

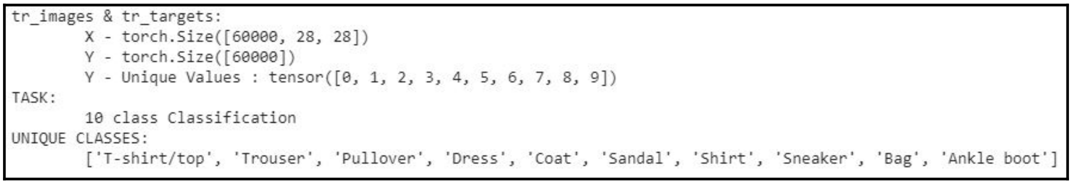
- Next, we must store the images that are available in `fmnist.data` as `tr_images` and the labels (targets) that are available in `fmnist.targets` as `tr_targets`:

```
tr_images = fmnist.data
tr_targets = fmnist.targets
```

2. Inspect the tensors that we are dealing with:

```
unique_values = tr_targets.unique()
print(f'tr_images & tr_targets:\n\tX -{tr_images.shape}\n\tY \
-{tr_targets.shape}\n\tY-Unique Values : {unique_values}')
print(f'TASK:\n\t{len(unique_values)} class Classification')
print(f'UNIQUE CLASSES:\n\t{fmnist.classes}')
```

The output of the preceding code is as follows:



```
tr_images & tr_targets:
        X - torch.Size([60000, 28, 28])
        Y - torch.Size([60000])
        Y - Unique Values : tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
TASK:
        10 class Classification
UNIQUE CLASSES:
        ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

Here, we can see that there are 60,000 images each of 28 x 28 in size and with 10 possible classes across all the images. Note that `tr_targets` contains the numeric values for each class, while `fmnist.classes` gives us the names that correspond to each numeric value in `tr_targets`.

3. Plot a random sample of 10 images for all the 10 possible classes:

- Import the relevant packages in order to plot a grid of images and so that you can also work on arrays:

```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
```

- Create a plot where we can show a 10 x 10 grid, where each row of the grid corresponds to a class and each column presents an example image belonging to the row's class. Loop through the unique class numbers (`label_class`) and fetch the indices of rows (`label_x_rows`) corresponding to the given class number:

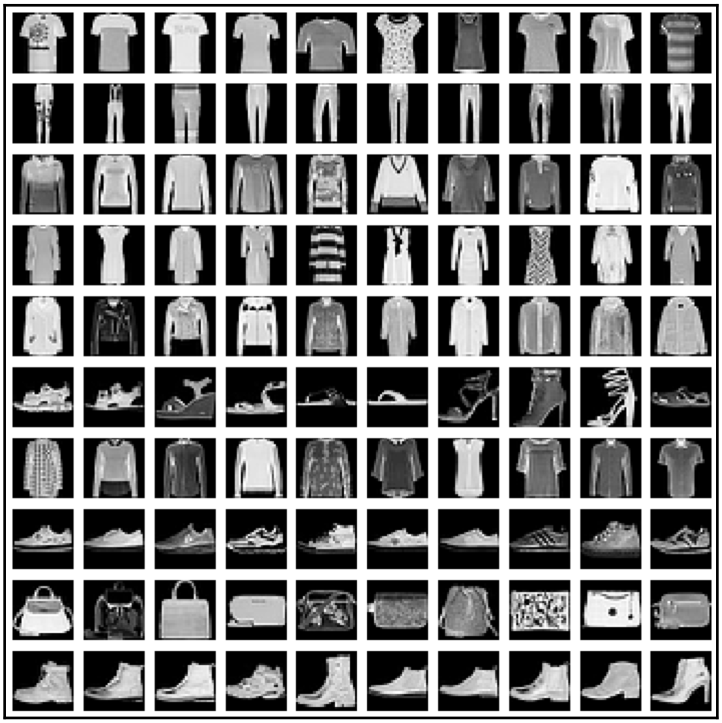
```
R, C = len(tr_targets.unique()), 10
fig, ax = plt.subplots(R, C, figsize=(10,10))
for label_class, plot_row in enumerate(ax):
    label_x_rows = np.where(tr_targets == label_class)[0]
```

Note that in the preceding code, we are fetching the 0th index as the output of the `np.where` condition as it has a length of 1. It contains an array of all the indices where the target value (`tr_targets`) is equal to `label_class`.

- Loop through 10 times to fill the columns of a given row. Furthermore, we need to select a random value (`ix`) from the indices corresponding to a given class that were obtained previously (`label_x_rows`) and plot them:

```
        for plot_cell in plot_row:
            plot_cell.grid(False); plot_cell.axis('off')
            ix = np.random.choice(label_x_rows)
            x, y = tr_images[ix], tr_targets[ix]
            plot_cell.imshow(x, cmap='gray')
    plt.tight_layout()
```

This results in the following output:



Note that in the preceding image, each row represents a sample of 10 different images all belonging to the same class.

Now that we have learned how to import a dataset, in the next section, we will learn how to train a neural network using PyTorch so that it takes in an image and predicts the class of that image. Furthermore, we will also learn about the impact that various hyperparameters have on the accuracy of prediction.

Training a neural network

To train a neural network, we must perform the following steps:

1. Import the relevant packages.
2. Build a dataset that can fetch data one data point at a time.
3. Wrap the DataLoader from the dataset.
4. Build a model and then define the loss function and the optimizer.
5. Define two functions to train and validate a batch of data, respectively.
6. Define a function that will calculate the accuracy of the data.
7. Perform weight updates based on each batch of data over increasing epochs.

In the following lines of code, we'll perform each of the following steps:

The following code is available as `Steps_to_build_a_neural_network_on_FashionMNIST.ipynb` in the `Chapter03` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Import the relevant packages and the FMNIST dataset:

```
from torch.utils.data import Dataset, DataLoader
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
device = "cuda" if torch.cuda.is_available() else "cpu"
from torchvision import datasets
data_folder = '~/data/FMNIST' # This can be any directory you
# want to download FMNIST to
fmnist = datasets.FashionMNIST(data_folder, download=True, \
                                                    train=True)
tr_images = fmnist.data
tr_targets = fmnist.targets
```

2. Build a class that fetches the dataset. Remember that it is derived from a `Dataset` class and needs three magic functions—`__init__`, `__getitem__`, and `__len__`—to **always** be defined:

```
class FMNISTDataset(Dataset):
    def __init__(self, x, y):
        x = x.float()
        x = x.view(-1,28*28)
        self.x, self.y = x, y
    def __getitem__(self, ix):
        x, y = self.x[ix], self.y[ix]
        return x.to(device), y.to(device)
    def __len__(self):
        return len(self.x)
```

   Note that in the `__init__` method, we have converted the input into a floating-point number and have also flattened each image into 28*28 = 784 numeric values (where each numeric value corresponds to a pixel value). We are also specifying the number of data points in the `__len__` method; here, it is the length of `x`. The `__getitem__` method contains logic for what should be returned when we ask for thee `ix`th data points (`ix` will be an integer between `0` and `__len__`).

3. Create a function that generates a training DataLoader – `trn_dl` from the dataset – called `FMNISTDataset`. This will sample 32 data points at random for the batch size:

```
def get_data():
    train = FMNISTDataset(tr_images, tr_targets)
    trn_dl = DataLoader(train, batch_size=32, shuffle=True)
    return trn_dl
```

   In the preceding lines of code, we created an object of the `FMNISTDataset` class named `train` and invoked the DataLoader so that it fetched 32 data points at random to return the training DataLoader; that is, `trn_dl`.

4. Define a model, as well as the loss function and the optimizer:

```
from torch.optim import SGD
def get_model():
    model = nn.Sequential(
                nn.Linear(28 * 28, 1000),
                nn.ReLU(),
                nn.Linear(1000, 10)
```

```
            ).to(device)
    loss_fn = nn.CrossEntropyLoss()
    optimizer = SGD(model.parameters(), lr=1e-2)
    return model, loss_fn, optimizer
```

The model is a network with one hidden layer containing 1,000 neurons. The output is a 10-neuron layer since there are 10 possible classes. Furthermore, we are calling the `CrossEntropyLoss` function since the output can belong to any of the 10 classes for each image. Finally, the key aspect to note in this exercise is that we have initialized the learning rate, `lr`, to a value of **0.01** and not the default of 0.001 to see how the model will learn for this exercise.

Note that we are not using "softmax" in the neural network at all. The range of outputs is unconstrained in that values can have an infinite range, whereas cross-entropy loss typically expects outputs as probabilities (each row should sum to 1). This still works in this setting because `nn.CrossEntropyLoss` actually expects us to send the raw logits (that is, unconstrained values). It performs softmax internally.

5. Define a function that will train the dataset on a batch of images:

```
def train_batch(x, y, model, opt, loss_fn):
    model.train() # <- let's hold on to this until we reach
    # dropout section
    # call your model like any python function on your batch
    # of inputs
    prediction = model(x)
    # compute loss
    batch_loss = loss_fn(prediction, y)
    # based on the forward pass in `model(x)` compute all the
    # gradients of 'model.parameters()'
    batch_loss.backward()
    # apply new-weights = f(old-weights, old-weight-gradients)
    # where "f" is the optimizer
    optimizer.step()
    # Flush gradients memory for next batch of calculations
    optimizer.zero_grad()
    return batch_loss.item()
```

The preceding code passes the batch of images through the model in the forward pass. It also computes the loss on batch and then passes the weights through backward propagation and updates them. Finally, it flushes the memory of the gradient so that it doesn't influence how the gradient is calculated in the next pass.

Now that we've done this, we can extract the loss value as a scalar by fetching `batch_loss.item()` on top of `batch_loss`.

6. Build a function that calculates the accuracy of a given dataset:

```
# since there's no need for updating weights,
# we might as well not compute the gradients.
# Using this '@' decorator on top of functions
# will disable gradient computation in the entire function
@torch.no_grad()
def accuracy(x, y, model):
    model.eval() # <- let's wait till we get to dropout
    # section
    # get the prediction matrix for a tensor of `x` images
    prediction = model(x)
    # compute if the location of maximum in each row
    # coincides with ground truth
    max_values, argmaxes = prediction.max(-1)
    is_correct = argmaxes == y
    return is_correct.cpu().numpy().tolist()
```

In the preceding lines of code, we are explicitly mentioning that we don't need to calculate the gradient by providing `@torch.no_grad()` and calculating the `prediction` values by feed-forwarding input through the model.

Next, we invoke `prediction.max(-1)` to identify the argmax index corresponding to each row.

Furthermore, we are comparing our `argmaxes` with the ground truth through `argmaxes == y` so that we can check whether each row is predicted correctly. Finally, we are returning the list of `is_correct` objects after moving it to a CPU and converting it into a numpy array.

7. Train the neural network using the following lines of code:

- Initialize the model, loss, optimizer, and DataLoaders:

```
trn_dl = get_data()
model, loss_fn, optimizer = get_model()
```

- Invoke the lists that contain the accuracy and loss values at the end of each epoch:

```
losses, accuracies = [], []
```

- Define the number of epochs:

```
for epoch in range(5):
    print(epoch)
```

- Invoke the lists that will contain the accuracy and loss values corresponding to each batch within an epoch:

```
    epoch_losses, epoch_accuracies = [], []
```

- Create batches of training data by iterating through the DataLoader:

```
    for ix, batch in enumerate(iter(trn_dl)):
        x, y = batch
```

- Train the batch using the `train_batch` function and store the loss value at the end of training on top of the batch as `batch_loss`. Furthermore, store the loss values across batches in the `epoch_losses` list:

```
        batch_loss = train_batch(x, y, model, optimizer, \
                                                  loss_fn)
        epoch_losses.append(batch_loss)
```

- We store the mean loss value across all batches within an epoch:

```
    epoch_loss = np.array(epoch_losses).mean()
```

- Next, we calculate the accuracy of the prediction at the end of training on all batches:

```
    for ix, batch in enumerate(iter(trn_dl)):
        x, y = batch
        is_correct = accuracy(x, y, model)
        epoch_accuracies.extend(is_correct)
    epoch_accuracy = np.mean(epoch_accuracies)
```

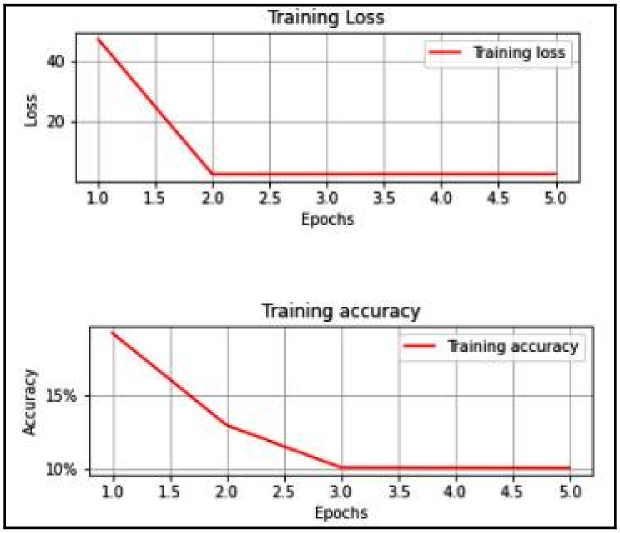
- Store the loss and accuracy values at the end of each epoch in a list:

```
losses.append(epoch_loss)
accuracies.append(epoch_accuracy)
```

The variation of the training loss and accuracy over increasing epochs can be displayed using the following code:

```
epochs = np.arange(5)+1
plt.figure(figsize=(20,5))
plt.subplot(121)
plt.title('Loss value over increasing epochs')
plt.plot(epochs, losses, label='Training Loss')
plt.legend()
plt.subplot(122)
plt.title('Accuracy value over increasing epochs')
plt.plot(epochs, accuracies, label='Training Accuracy')
plt.gca().set_yticklabels(['{:.0f}%'.format(x*100) \
                           for x in plt.gca().get_yticks()])
plt.legend()
```

The output of the preceding code is as follows:



Our training accuracy is at 12% at the end of the five epochs. Note that the loss value did not decrease considerably over an increasing number of epochs. In other words, no matter how long we wait, it is unlikely that the model is going to provide high accuracy (say, above 80%). This calls for us to understand how the various hyperparameters that were used impact the accuracy of our neural network.

Note that since we did not keep `torch.random_seed(0)`, the results might vary when you execute the code provided. However, the results you get should also get you to a similar conclusion.

Now that you have a complete picture of how to train a neural network, let's study some good practices we should follow to achieve good model performance and the reasons behind using them. This can be achieved by fine-tuning various hyperparameters, some of which we will look at in the upcoming sections.

Scaling a dataset to improve model accuracy

Scaling a dataset is the process of ensuring that the variables are confined to a finite range. In this section, we will confine the independent variables' values to values between 0 and 1 by dividing each input value by the maximum possible value in the dataset. This is a value of 255, which corresponds to white pixels:

> The following code is available as `Scaling_the_dataset.ipynb` in the `Chapter03` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Fetch the dataset, as well as the training images and targets, as we did in the previous section:

```
from torchvision import datasets
from torch.utils.data import Dataset, DataLoader
import torch
import torch.nn as nn
device = "cuda" if torch.cuda.is_available() else "cpu"
import numpy as np
data_folder = '~/data/FMNIST' # This can be any directory you
# want to download FMNIST to
fmnist = datasets.FashionMNIST(data_folder, download=True, \
                                    train=True)
tr_images = fmnist.data
tr_targets = fmnist.targets
```

2. Modify `FMNISTDataset`, which fetches data, so that the input image is divided by 255 (the maximum intensity/value of a pixel):

```
class FMNISTDataset(Dataset):
    def __init__(self, x, y):
        x = x.float()/255
        x = x.view(-1,28*28)
        self.x, self.y = x, y
    def __getitem__(self, ix):
        x, y = self.x[ix], self.y[ix]
        return x.to(device), y.to(device)
    def __len__(self):
        return len(self.x)
```

   Note that the only change we've made here compared to the previous section is that we're dividing the input data by the maximum possible pixel value – 255.

   Given that the pixel values range between 0 to 255, dividing them by 255 will result in values that are always between 0 to 1.

3. Train a model, just like we did in *steps 4*, *5*, *6*, and *7* of the previous section:

- Fetch the data:

```
def get_data():
    train = FMNISTDataset(tr_images, tr_targets)
    trn_dl = DataLoader(train, batch_size=32, shuffle=True)
    return trn_dl
```

- Define the model:

```
from torch.optim import SGD
def get_model():
    model = nn.Sequential(
                nn.Linear(28 * 28, 1000),
                nn.ReLU(),
                nn.Linear(1000, 10)
            ).to(device)
    loss_fn = nn.CrossEntropyLoss()
    optimizer = SGD(model.parameters(), lr=1e-2)
    return model, loss_fn, optimizer
```

- Define the functions for training and validating a batch of data:

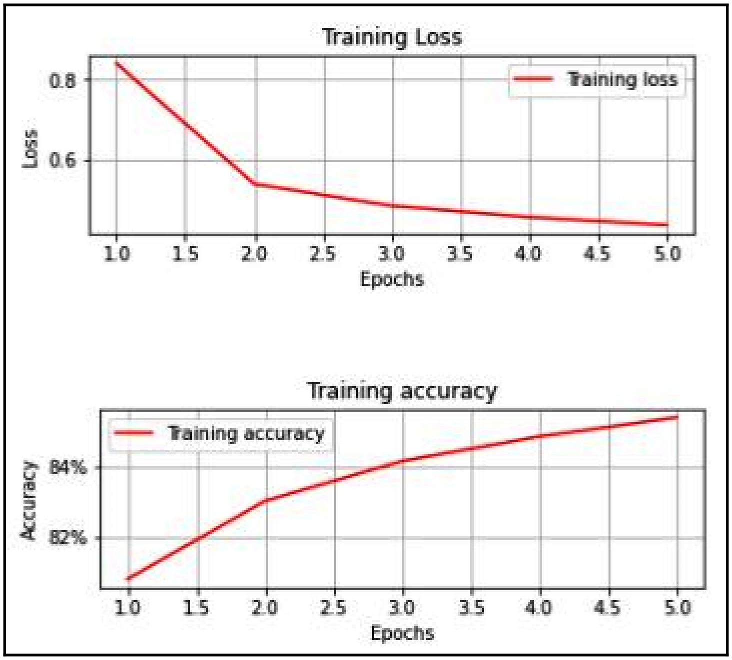
```
def train_batch(x, y, model, opt, loss_fn):
    model.train()
    # call your model like any python function on your batch
    # of inputs
    prediction = model(x)
    # compute loss
    batch_loss = loss_fn(prediction, y)
    # based on the forward pass in `model(x)` compute all the
    # gradients of 'model.parameters()'
    batch_loss.backward()
    # apply new-weights = f(old-weights, old-weight-gradients)
    # where "f" is the optimizer
    optimizer.step()
    # Flush memory for next batch of calculations
    optimizer.zero_grad()
    return batch_loss.item()
@torch.no_grad()
def accuracy(x, y, model):
    model.eval()
    # get the prediction matrix for a tensor of `x` images
    prediction = model(x)
    # compute if the location of maximum in each row
    # coincides with ground truth
    max_values, argmaxes = prediction.max(-1)
    is_correct = argmaxes == y
    return is_correct.cpu().numpy().tolist()
```

- Train the model over increasing epochs:

```
trn_dl = get_data()
model, loss_fn, optimizer = get_model()
losses, accuracies = [], []
for epoch in range(5):
    print(epoch)
    epoch_losses, epoch_accuracies = [], []
    for ix, batch in enumerate(iter(trn_dl)):
        x, y = batch
        batch_loss = train_batch(x, y, model, optimizer,
                                                loss_fn)
        epoch_losses.append(batch_loss)
    epoch_loss = np.array(epoch_losses).mean()
    for ix, batch in enumerate(iter(trn_dl)):
        x, y = batch
        is_correct = accuracy(x, y, model)
        epoch_accuracies.extend(is_correct)
    epoch_accuracy = np.mean(epoch_accuracies)
```

```
        losses.append(epoch_loss)
        accuracies.append(epoch_accuracy)
```

The variations for the training loss and accuracy values are as follows:



As we can see, the training loss consistently reduced and the training accuracy consistently increased, thus increasing the epochs to an accuracy of ~85%.

Contrast the preceding output with the scenario where input data is not scaled, where training loss did not reduce consistently, and the accuracy of the training dataset at the end of five epochs was only 12%.

Let's dive into the possible reason why scaling helps here.

Let's take the example of how a sigmoid value is calculated:

$$Sigmoid = 1/(1 + e^{-(Input*Weight)})$$

In the following table, we've calculated the **Sigmoid** column based on the preceding formula:

Input	Weight	Sigmoid
255	0.00001	0.501
255	0.0001	0.506
255	0.001	0.563
255	0.01	0.928
255	0.1	1.000
255	0.2	1.000
255	0.3	1.000
255	0.4	1.000
255	0.5	1.000
255	0.6	1.000
255	0.7	1.000
255	0.8	1.000
255	0.9	1.000
255	1	1.000

Input	Weight	Sigmoid
1	0.00001	0.500
1	0.0001	0.500
1	0.001	0.500
1	0.01	0.502
1	0.1	0.525
1	0.2	0.550
1	0.3	0.574
1	0.4	0.599
1	0.5	0.622
1	0.6	0.646
1	0.7	0.668
1	0.8	0.690
1	0.9	0.711
1	1	0.731

In the left-hand table, we can see that when the weight values are more than 0.1, the Sigmoid value does not vary with an increasing (changing) weight value. Furthermore, the Sigmoid value changed only by a little when the weight was extremely small; the only way to vary the sigmoid value is by having the weight change to a very, very small amount.

However, the Sigmoid value changed considerably in the right-hand table when the input value was small.

The reason for this is that the exponential of a large negative value (resulting from multiplying the weight value by a large number) is very close to 0, while the exponential value varies when the weight is multiplied by a scaled input, as seen in the right-hand table.

Now that we have understood that the Sigmoid value does not change considerably unless the weight values are very small, we will now learn about how weight values can be influenced toward an optimal value.

TIP

Scaling the input dataset so that it contains a much smaller range of values generally helps in achieving better model accuracy.

Next, we'll learn about the impact of one of the other major hyperparameters of any neural network: **batch size.**

Understanding the impact of varying the batch size

In the previous section, 32 data points were considered per batch in the training dataset. This resulted in a greater number of weight updates per epoch as there were 1,875 weight updates per epoch (60,000/32 is nearly equal to 1,875, where 60,000 is the number of training images).

Furthermore, we did not consider the model's performance on an unseen dataset (validation dataset). We will explore this in this section.

In this section, we will compare the following:

- The loss and accuracy values of the training and validation data when the training batch size is 32.
- The loss and accuracy values of the training and validation data when the training batch size is 10,000.

Now that we have brought validation data into the picture, let's rerun the code provided in the *Building a neural network* section with additional code to generate validation data, as well as to calculate the loss and accuracy values of the validation dataset.

The code for the *Batch size of 32* and *Batch size of 10,000* sections is available as `Varying_batch_size.ipynb` in the `Chapter03` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

Batch size of 32

Given that we have already built a model that uses a batch size of 32 during training, we will elaborate on the additional code that is used to work on the validation dataset. We'll skip going through the details of training the model since this is already present in the *Building a neural network* section. Let's get started:

1. Download and import the training images and targets:

```
from torchvision import datasets
import torch
data_folder = '~/data/FMNIST' # This can be any directory you
# want to download FMNIST to
```

```
fmnist = datasets.FashionMNIST(data_folder, download=True, \
                                                train=True)
tr_images = fmnist.data
tr_targets = fmnist.targets
```

2. In a similar manner to training images, we must download and import the validation dataset by specifying `train = False` while calling the `FashionMNIST` method in our datasets:

```
val_fmnist =datasets.FashionMNIST(data_folder,download=True, \
                                                train=False)
val_images = val_fmnist.data
val_targets = val_fmnist.targets
```

3. Import the relevant packages and define `device`:

```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
from torch.utils.data import Dataset, DataLoader
import torch
import torch.nn as nn
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

4. Define the dataset class (`FashionMNIST`), the functions that will be used to train on a batch of data (`train_batch`), calculate the accuracy (`accuracy`), and then define the model architecture, the loss function, and the optimizer (`get_model`). Note that the function for getting data will be the only function that will have a deviation from what we have seen in previous sections (as we are now working on training and validation datasets), so we will build it in the next step:

```
class FMNISTDataset(Dataset):
    def __init__(self, x, y):
        x = x.float()/255
        x = x.view(-1,28*28)
        self.x, self.y = x, y
    def __getitem__(self, ix):
        x, y = self.x[ix], self.y[ix]
        return x.to(device), y.to(device)
    def __len__(self):
        return len(self.x)

from torch.optim import SGD, Adam
def get_model():
    model = nn.Sequential(
                nn.Linear(28 * 28, 1000),
```

```
                nn.ReLU(),
                nn.Linear(1000, 10)
            ).to(device)

    loss_fn = nn.CrossEntropyLoss()
    optimizer = Adam(model.parameters(), lr=1e-2)
    return model, loss_fn, optimizer

def train_batch(x, y, model, opt, loss_fn):
    model.train()
    prediction = model(x)
    batch_loss = loss_fn(prediction, y)
    batch_loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    return batch_loss.item()

def accuracy(x, y, model):
    model.eval()
    # this is the same as @torch.no_grad
    # at the top of function, only difference
    # being, grad is not computed in the with scope
    with torch.no_grad():
        prediction = model(x)
    max_values, argmaxes = prediction.max(-1)
    is_correct = argmaxes == y
    return is_correct.cpu().numpy().tolist()
```

5. Define a function that will get data; that is, `get_data`. This function will return the training data with a batch size of 32 and the validation dataset with a batch size that's the length of the validation data (we will not use the validation data to train the model; we will only use it to understand the model's accuracy on unseen data):

```
def get_data():
    train = FMNISTDataset(tr_images, tr_targets)
    trn_dl = DataLoader(train, batch_size=32, shuffle=True)
    val = FMNISTDataset(val_images, val_targets)
    val_dl = DataLoader(val, batch_size=len(val_images),
                                            shuffle=False)
    return trn_dl, val_dl
```

In the preceding code, we created an object of the `FMNISTDataset` class named `val`, in addition to the `train` object that we saw earlier. Furthermore, the DataLoader for validation (`val_dl`) has been fetched with a batch size of `len(val_images)`, while the batch size of `trn_dl` is 32. This is because the training data is used to train the model while we are fetching the accuracy and loss metrics of the validation data. In this section and the next, we are trying to understand the impact of varying `batch_size` based on the model's training time and accuracy.

6. Define a function that calculates the loss of the validation data; that is, `val_loss`. Note that we are calculating this separately since loss of training data is getting calculated while training the model:

```
@torch.no_grad()
def val_loss(x, y, model):
    model.eval()
    prediction = model(x)
    val_loss = loss_fn(prediction, y)
    return val_loss.item()
```

As you can see, we are applying `torch.no_grad` because we are not training the model and only fetching predictions. Furthermore, we are passing our `prediction` through the loss function (`loss_fn`) and returning the loss value (`val_loss.item()`).

7. Fetch the training and validation DataLoaders. Also, initialize the model, loss function, and optimizer:

```
trn_dl, val_dl = get_data()
model, loss_fn, optimizer = get_model()
```

8. Train the model, as follows:

- Initialize the lists that contain training and validation accuracy, as well as loss values over increasing epochs:

```
train_losses, train_accuracies = [], []
val_losses, val_accuracies = [], []
```

- Loop through five epochs and initialize lists that contain accuracy and loss across batches of training data within a given epoch:

```
for epoch in range(5):
    print(epoch)
    train_epoch_losses, train_epoch_accuracies = [], []
```

- Loop through batches of training data and calculate the accuracy (`train_epoch_accuracy`) and loss value (`train_epoch_loss`) within an epoch:

```
for ix, batch in enumerate(iter(trn_dl)):
    x, y = batch
    batch_loss = train_batch(x, y, model, optimizer, \
                                         loss_fn)
    train_epoch_losses.append(batch_loss)
train_epoch_loss = np.array(train_epoch_losses).mean()

for ix, batch in enumerate(iter(trn_dl)):
    x, y = batch
    is_correct = accuracy(x, y, model)
    train_epoch_accuracies.extend(is_correct)
train_epoch_accuracy = np.mean(train_epoch_accuracies)
```

- Calculate the loss value and accuracy within the one batch of validation data (since the batch size of the validation data is equal to the length of the validation data):

```
for ix, batch in enumerate(iter(val_dl)):
    x, y = batch
    val_is_correct = accuracy(x, y, model)
    validation_loss = val_loss(x, y, model)
val_epoch_accuracy = np.mean(val_is_correct)
```

Note that in the preceding code, the loss value of the validation data is calculated using the `val_loss` function and is stored in the `validation_loss` variable. Furthermore, the accuracy of all the validation data points is stored in the `val_is_correct` list, while the mean of this is stored in the `val_epoch_accuracy` variable.

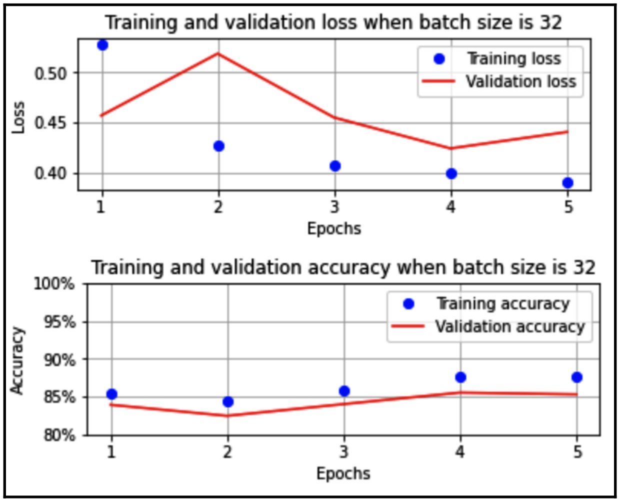
- Finally, we append the training and validation datasets' accuracy and loss values to the lists that contain the epoch level aggregate validation and accuracy values. We're doing this so that we can look at the epoch level's improvement in the next step:

```
train_losses.append(train_epoch_loss)
train_accuracies.append(train_epoch_accuracy)
val_losses.append(validation_loss)
val_accuracies.append(val_epoch_accuracy)
```

9. Visualize the improvements in the accuracy and loss values in the training and validation datasets over increasing epochs:

```
epochs = np.arange(5)+1
import matplotlib.ticker as mtick
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
%matplotlib inline
plt.subplot(211)
plt.plot(epochs, train_losses, 'bo', label='Training loss')
plt.plot(epochs, val_losses, 'r', label='Validation loss')
plt.gca().xaxis.set_major_locator(mticker.MultipleLocator(1))
plt.title('Training and validation loss \
when batch size is 32')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid('off')
plt.show()
plt.subplot(212)
plt.plot(epochs, train_accuracies, 'bo', \
         label='Training accuracy')
plt.plot(epochs, val_accuracies, 'r', \
         label='Validation accuracy')
plt.gca().xaxis.set_major_locator(mticker.MultipleLocator(1))
plt.title('Training and validation accuracy \
when batch size is 32')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.gca().set_yticklabels(['{:.0f}%'.format(x*100) \
                           for x in plt.gca().get_yticks()])
plt.legend()
plt.grid('off')
plt.show()
```

The preceding code gives us the following output:



As you can see, the training and validation accuracy are ~85% by the end of five epochs when the batch size is 32. Next, we will vary the `batch_size` parameter when training the DataLoader in the `get_data` function to see its impact on accuracy at the end of five epochs.

Batch size of 10,000

In this section, we'll use 10,000 data points per batch so that we can understand what impact varying the batch size has.

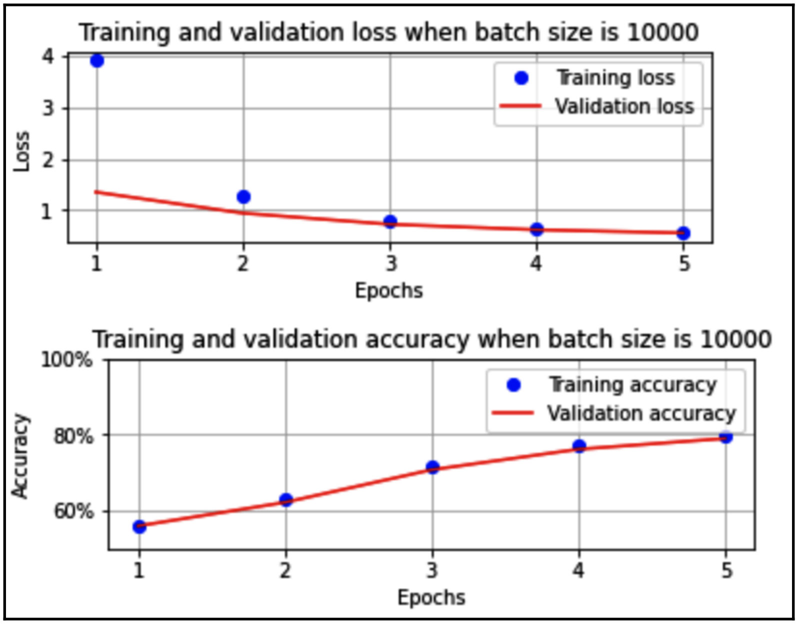
Note that the code provided in the *Batch size of 32* section remains exactly the same here, except for the code in *step 5*. Here, we will specify the DataLoaders for the training and validation datasets in the `get_data` function. **We encourage you to refer to the respective notebook that's available in this book's GitHub repository while executing the code.**

We will modify `get_data` so that it has a batch size of 10,000 while fetching the training DataLoader from the training dataset, as follows:

```
def get_data():
    train = FMNISTDataset(tr_images, tr_targets)
    trn_dl = DataLoader(train, batch_size=10000, shuffle=True)
    val = FMNISTDataset(val_images, val_targets)
```

```
    val_dl = DataLoader(val, batch_size=len(val_images), \
                                            shuffle=False)
    return trn_dl, val_dl
```

By making only this necessary change in *step 5* and after executing all the steps until *step 9*, the variation in the training and validation's accuracy and loss over increasing epochs when the batch size is 10,000 is as follows:



Here, we can see that the accuracy and loss values did not reach the same levels as that of the previous scenario, where the batch size was 32, because the time weights are updated fewer times when the batch size is 32 (1875). In the scenario where the batch size is 10,000, there were six weight updates per epoch since there were 10,000 data points per batch, which means that the total training data size was 60,000.

So far, we have learned how to scale a dataset, as well as the impact of varying the batch size on the model's training time to achieve a certain accuracy. In the next section, we will learn about the impact of varying the loss optimizer on the same dataset.

Having a lower batch size generally helps in achieving optimal accuracy when you have a small number of epochs, but it should not be so low that training time is impacted.

Understanding the impact of varying the loss optimizer

So far, we have been optimizing loss based on the Adam optimizer. In this section, we will do the following:

- Modify the optimizer so that it becomes a **Stochastic Gradient Descent** (**SGD**) optimizer
- Revert to a batch size of 32 while fetching data in the DataLoader
- Increase the number of epochs to 10 (so that we can compare the performance of SGD and Adam over a longer number of epochs)

Making these changes means that only one step in the *Batch size of 32* section will change (since the batch size is already 32 in the *Batch size of 32* section); that is, we will modify the optimizer so that it's the SGD optimizer.

Let's modify the `get_model` function in *step 4* of the *Batch size of 32* section in order to modify the optimzier so that we're using the SGD optimizer instead, as follows:

The following code is available as `Varying_loss_optimizer.ipynb` in the `Chapter03` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` . Note that we are not providing all the steps for brevity and that only the steps where we're making a change from the code we went through in the *Batch size of 32* section will be discussed in the following code. **We encourage you to refer to the respective notebooks in this book's GitHub repository while executing the code.**

1. Modify the optimizer so that you're using the SGD optimizer in the `get_model` function while ensuring that everything else remains the same:

```
from torch.optim import SGD, Adam
def get_model():
```

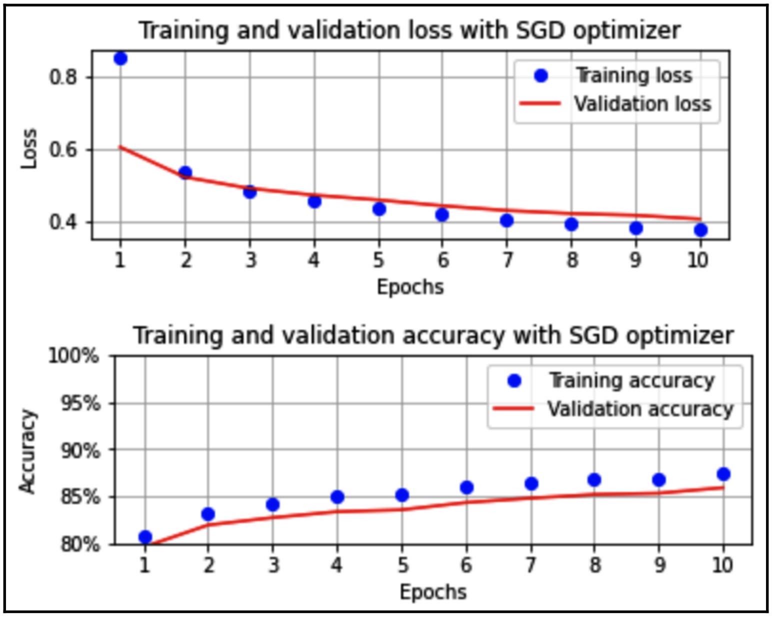
```
    model = nn.Sequential(
                nn.Linear(28 * 28, 1000),
                nn.ReLU(),
                nn.Linear(1000, 10)
            ).to(device)

    loss_fn = nn.CrossEntropyLoss()
    optimizer = SGD(model.parameters(), lr=1e-2)
    return model, loss_fn, optimizer
```

Now, let's increase the number of epochs in *step 8* while keeping every other step (except for *steps 4* and *8*) the same as they are in the *Batch size of 32* section.

2. Increase the number of epochs we'll be using to train the model:

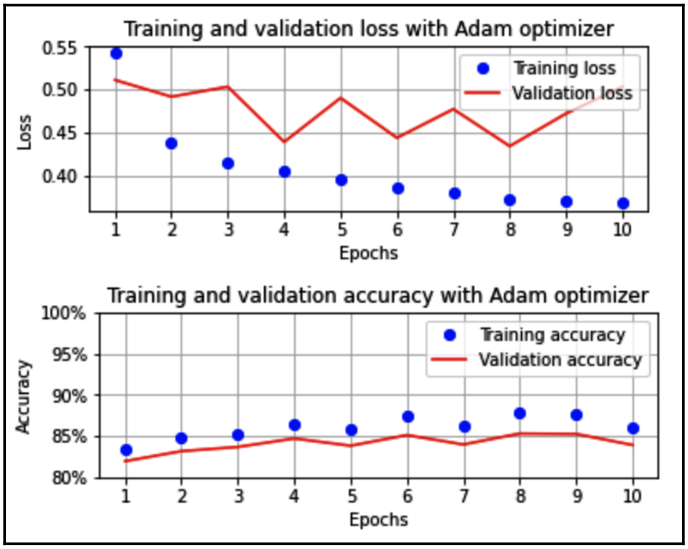
```
train_losses, train_accuracies = [], []
val_losses, val_accuracies = [], []
for epoch in range(10):
    train_epoch_losses, train_epoch_accuracies = [], []
    for ix, batch in enumerate(iter(trn_dl)):
        x, y = batch
        batch_loss = train_batch(x, y, model, optimizer, \
                                                    loss_fn)
        train_epoch_losses.append(batch_loss)
    train_epoch_loss = np.array(train_epoch_losses).mean()

    for ix, batch in enumerate(iter(trn_dl)):
        x, y = batch
        is_correct = accuracy(x, y, model)
        train_epoch_accuracies.extend(is_correct)
    train_epoch_accuracy = np.mean(train_epoch_accuracies)

    for ix, batch in enumerate(iter(val_dl)):
        x, y = batch
        val_is_correct = accuracy(x, y, model)
        validation_loss = val_loss(x, y, model)
    val_epoch_accuracy = np.mean(val_is_correct)

    train_losses.append(train_epoch_loss)
    train_accuracies.append(train_epoch_accuracy)
    val_losses.append(validation_loss)
    val_accuracies.append(val_epoch_accuracy)
```

After making these changes, once we execute all the remaining steps in the *Batch size of 32* section in order, the variation in the training and validation datasets' accuracy and loss values over increasing epochs will be as follows:



Let's fetch the same output for the training and validation loss and accuracy variation over increasing epochs where the optimizer is Adam. This requires us to change the optimizer in *step 4* to Adam.

The variation in the training and validation datasets' accuracy and loss values, once this change has been made and the code has been executed, is as follows:



As you can see, when we used the Adam optimizer, the accuracy was still very close to 85%. However, note that so far, the learning rate has been 0.01.

In the next section, we will learn about the impact the learning rate can have on the validation dataset's accuracy.

TIP

Certain optimizers achieve optimal accuracy faster compared to others. Adam generally achieves optimal accuracy faster. Some of the other prominent optimizers that are available include Adagrad, Adadelta, AdamW, LBFGS, and RMSprop.

Understanding the impact of varying the learning rate

So far, we have been using a learning rate of 0.01 while training our models. In Chapter 1, *Artificial Neural Network Fundamentals*, we learned that the learning rate plays a key role in attaining optimal weight values. Here, the weight values gradually move toward the optimal value when the learning rate is small, while the weight value oscillates at a non-optimal value when the learning rate is large. We worked on a toy dataset in Chapter 1, *Artificial Neural Network Fundamentals*, so we will work on a realistic scenario in this section.

To understand the impact of the varying learning rate, we'll go through the following scenario:

- Higher learning rate (0.1) on a scaled dataset
- Lower learning rate (0.00001) on a scaled dataset
- Lower learning rate (0.001) on a non-scaled dataset
- Higher learning rate (0.1) on a non-scaled dataset

Overall, in this section, we'll be learning about the impact that various learning rate values have on scaled and non-scaled datasets.

In this section, we are learning about the impact the learning rate has on non-scaled data, even though we have already established that it is helpful to scale a dataset. We're doing this again because we want you to gain an intuition of how the distribution of weights varies between the scenario where the model is able to fit to the data versus where the model isn't able to fit to the data.

Now, let's learn how the model learns on a scaled dataset.

Impact of the learning rate on a scaled dataset

In this section, we will contrast the accuracy of the training and validation datasets against the following:

- High learning rate
- Medium learning rate
- Low learning rate

The code for the following three subsections is available as `Varying_learning_rate_on_scaled_data.ipynb` in the `Chapter03` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` . Note that we are not providing all the steps for brevity; only the steps where there is a change from the code we went through in the *Batch size of 32* section will be discussed in the following code. **We encourage you to refer to the respective notebooks in this book's GitHub repository while executing the code.**

Let's get started!

High learning rate

In this section, we will adopt the following strategy:

- The steps we need to execute will be exactly the same as in the *Batch size of 32* section, when we used the Adam optimizer.
- The only change will be in the learning rate in `optimizer` while we define the `get_model` function. Here, we'll be changing the learning rate (`lr`) to a value of 0.1.

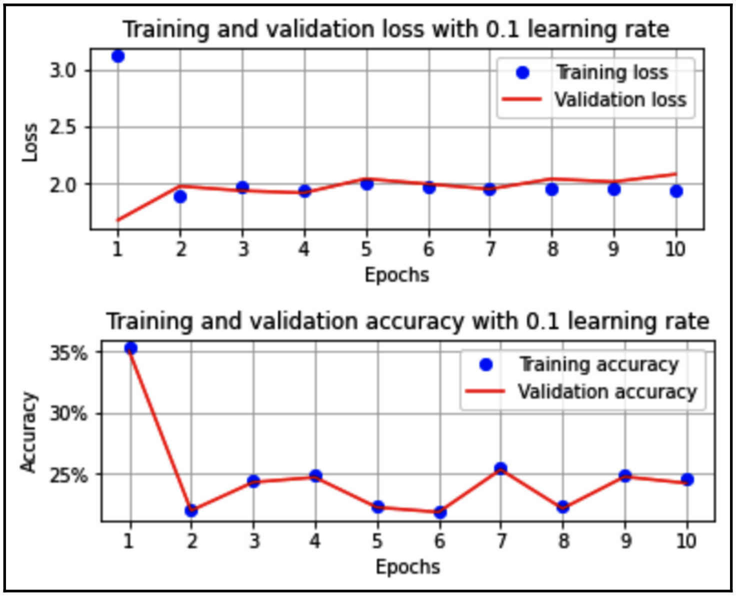
Note that all the code remains the same as in the *Batch size of 32* section, except for the modifications in the `get_model` function that we will be making in this section.

To modify the learning rate, we must change it in the definition of `optimizer`, which can be found in the `get_model` function, as follows:

```
def get_model():
    model = nn.Sequential(
                nn.Linear(28 * 28, 1000),
                nn.ReLU(),
                nn.Linear(1000, 10)
            ).to(device)

    loss_fn = nn.CrossEntropyLoss()
    optimizer = Adam(model.parameters(), lr=1e-1)
    return model, loss_fn, optimizer
```

Note that in the preceding code, we have modified the optimizer so that it has a learning rate of 0.1 (`lr=1e-1`).

Once we execute all the remaining steps, as provided in GitHub, the accuracy and loss values corresponding to the training and validation datasets will be as follows:



Note that the accuracy of the validation dataset is ~25% (contrast this accuracy with the ~85% accuracy we achieved when the learning rate was 0.01).

In the next section, we will understand the accuracy of the validation dataset when the learning rate is medium (0.001).

Medium learning rate

In this section, we'll reduce the learning rate of the optimizer to 0.001 by modifying the `get_model` function and retraining the model from scratch.

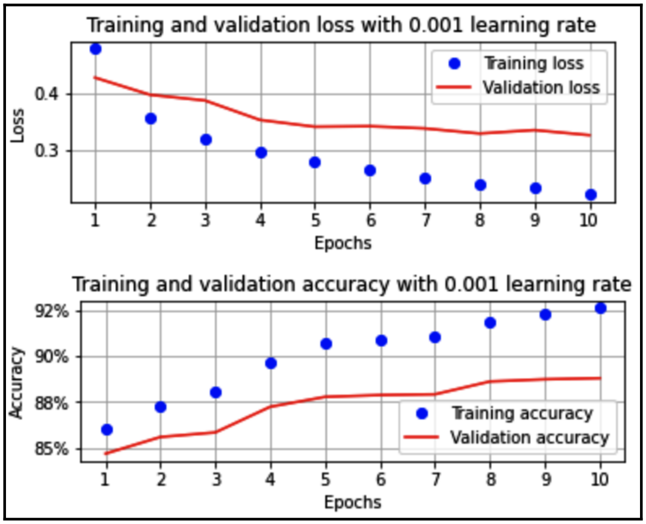
The modified code for the `get_model` function is as follows:

```
def get_model():
    model = nn.Sequential(
                nn.Linear(28 * 28, 1000),
                nn.ReLU(),
                nn.Linear(1000, 10)
            ).to(device)

    loss_fn = nn.CrossEntropyLoss()
```

```
    optimizer = Adam(model.parameters(), lr=1e-3)
    return model, loss_fn, optimizer
```

Note that in the preceding code, the learning rate has been reduced to a small value since we modified the `lr` parameter value.

Once we execute all the remaining steps, as provided in GitHub, the accuracy and loss values corresponding to the training and validation datasets will be as follows:



From the preceding output, we can see that the model was trained successfully when the learning rate (`or`) was reduced from 0.1 to 0.001

In the next section, we will reduce the learning rate even further.

Low learning rate

In this section, we'll reduce the learning rate of the optimizer to 0.00001 by modifying the `get_model` function and retraining the model from scratch. In addition, we will run the model for a longer number of epochs (100).

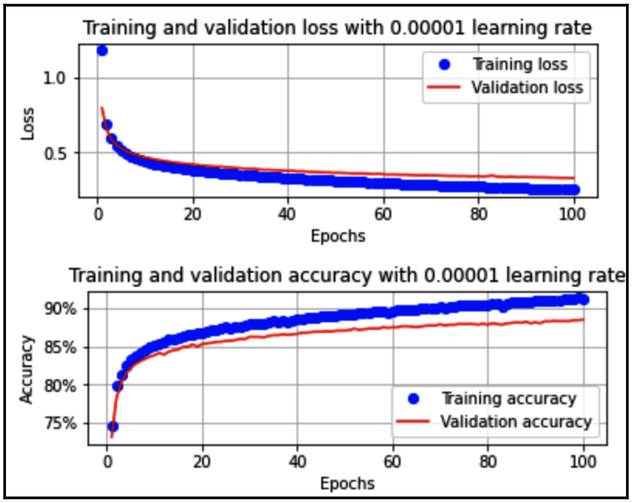
The modified code we'll be using for the `get_model` function is as follows:

```
def get_model():
    model = nn.Sequential(
                nn.Linear(28 * 28, 1000),
                nn.ReLU(),
```

```
            nn.Linear(1000, 10)
        ).to(device)

    loss_fn = nn.CrossEntropyLoss()
    optimizer = Adam(model.parameters(), lr=1e-5)
    return model, loss_fn, optimizer
```

Note that in the preceding code, the learning rate has been reduced to a very small value due to us modifying the `lr` parameter value.

Once we execute all the remaining steps, as provided in GitHub, the accuracy and loss values corresponding to the training and validation datasets will be as follows:



From the preceding image, we can see that the model learned far slower compared to the previous scenario (medium learning rate). Here, it took ~100 epochs to reach an accuracy of ~89% compared to eight epochs when the learning rate was 0.001.

In addition, we should also note that the gap between the training and validation loss is much lower when the learning rate is low compared to the previous scenario (where a similar gap existed at the end of epoch 4). The reason for this is that the weight update is much lower when the learning rate is low, which means that the gap between the training and validation loss does not widen quickly.

So far, we have learned about the impact the learning rate has on the training and validation datasets' accuracy. In the next section, we'll learn how the weight values' distribution varies across layers for different learning rate values.

Parameter distribution across layers for different learning rates

In the previous sections, we learned that with a high learning rate (0.1), the model was unable to be trained (the model underfitted). However, we could train the model so that it had a decent accuracy when the learning rate was either medium (0.001) or low (0.00001). Here, we saw that the medium learning rate was able to overfit quickly, while the low learning rate took a longer time to achieve an accuracy comparable to that of a medium learning rate model.

In this section, we will learn about how parameter distribution can be a good indicator of model overfit and underfit.

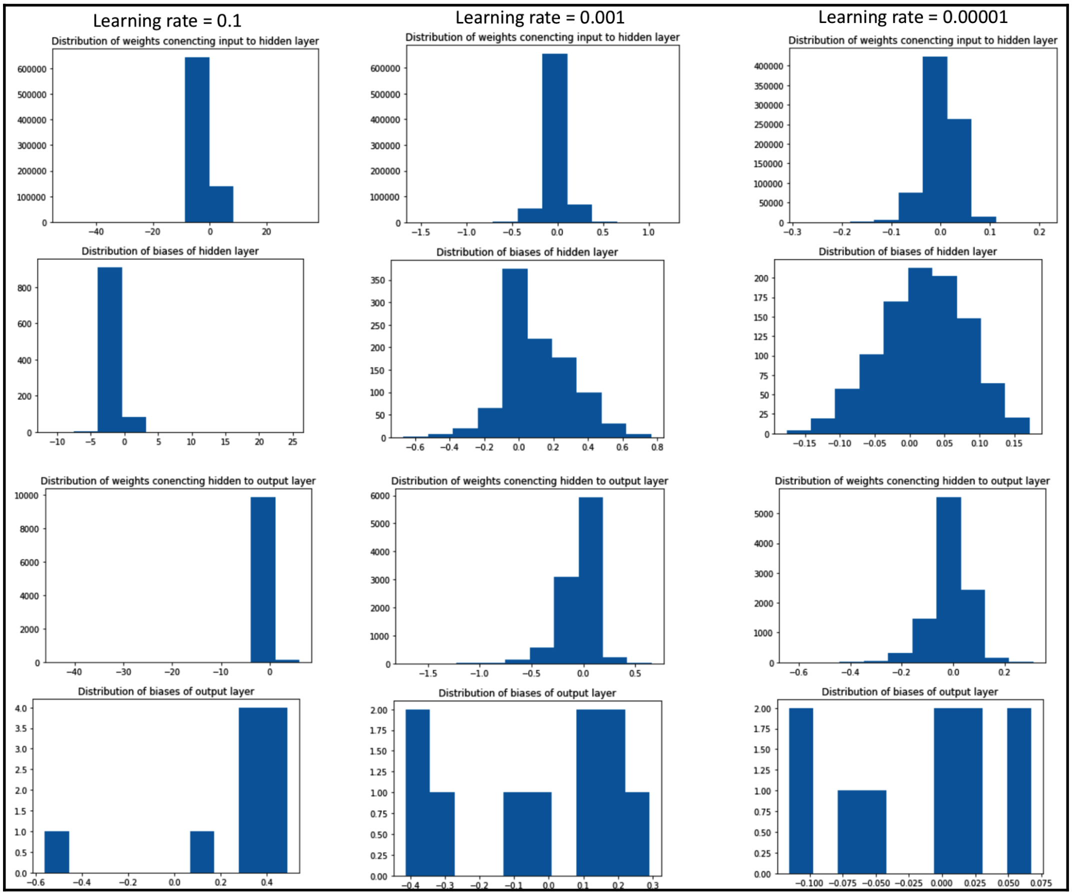
So far, there are four parameter groups in our model:

- Weights in the layer connecting the input layer to the hidden layer
- Bias in the hidden layer
- Weights in the layer connecting the hidden layer to the output layer
- Bias in the output layer

Let's take a look at the distribution of each of these parameters by using the following code (we'll execute the following code for each model):

```
for ix, par in enumerate(model.parameters()):
    if(ix==0):
        plt.hist(par.cpu().detach().numpy().flatten())
        plt.title('Distribution of weights conencting \
                    input to hidden layer')
        plt.show()
    elif(ix ==1):
        plt.hist(par.cpu().detach().numpy().flatten())
        plt.title('Distribution of biases of hidden layer')
        plt.show()
    elif(ix==2):
        plt.hist(par.cpu().detach().numpy().flatten())
        plt.title('Distribution of weights conencting \
                    hidden to output layer')
        plt.show()
    elif(ix ==3):
        plt.hist(par.cpu().detach().numpy().flatten())
        plt.title('Distribution of biases of output layer')
        plt.show()
```

Note that `model.parameters` will vary by the model we are plotting the distribution for. The output of the preceding code across the three learning rates is as follows:



Here, we can see the following:

- When the learning rate is high, parameters have a much larger distribution compared to medium and low learning rates.
- When parameters have a bigger distribution, overfitting occurs.

So far, we've studied the impact of varying the learning rate on a model that's been trained on a scaled dataset. In the next section, we'll learn about the impact of varying the learning rate on a model that's been trained on non-scaled data.

Note that even though we have already established that it is better to always scale input values, we will continue to establish the impact of training a model on a non-scaled dataset.

Impact of varying the learning rate on a non-scaled dataset

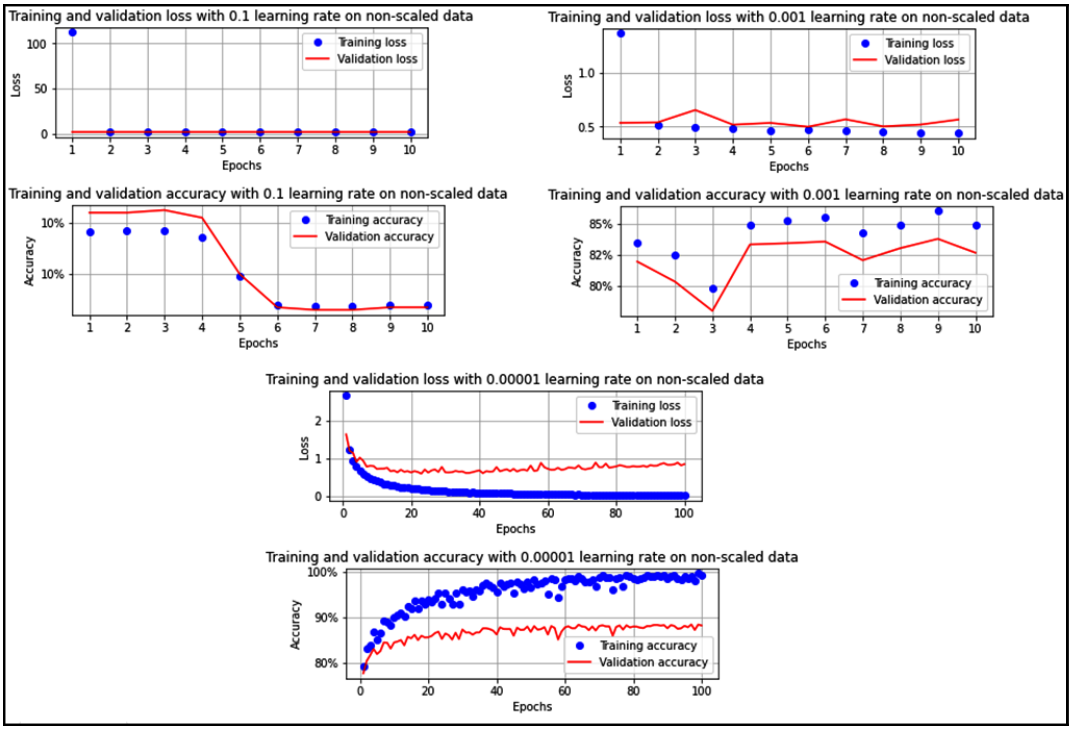
In this section, we will revert to working on a dataset by not performing division by 255 in the class where we define the dataset. This can be done like so:

The code for this section is available as `Varying_learning_rate_on_non_scaled_data.ipynb` in the `Chapter03` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` . Note that we are not providing all the steps for brevity and that only the steps where there is a change from the code we went through in the *Batch size of 32* section will be discussed in the following code. **We encourage you to refer to the notebooks in this book's GitHub repository while executing the code.**

```
class FMNISTDataset(Dataset):
    def __init__(self, x, y):
        x = x.float() # Note that the data is not scaled in this
        # scenario
        x = x.view(-1,28*28)
        self.x, self.y = x, y
    def __getitem__(self, ix):
        x, y = self.x[ix], self.y[ix]
        return x.to(device), y.to(device)
    def __len__(self):
        return len(self.x)
```

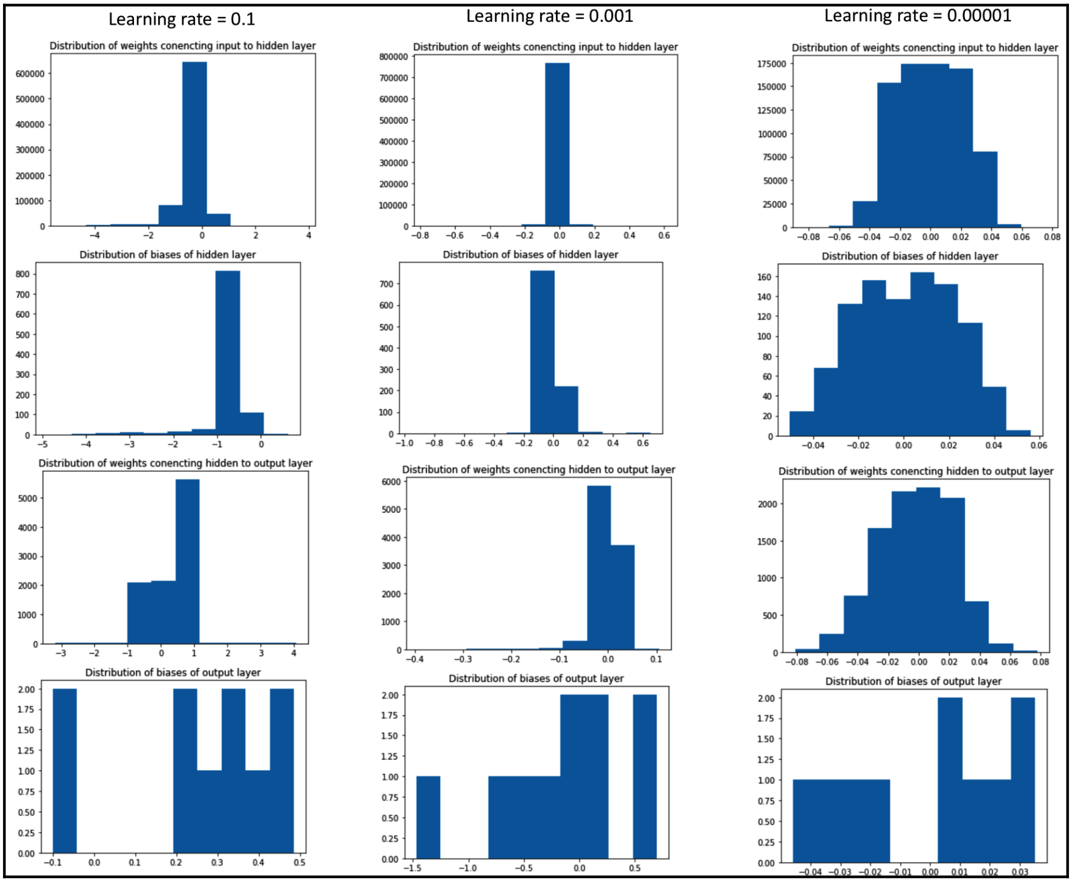
Note that, in the highlighted section in the preceding code (`x = x.float()`), we did not divide by 255, which we performed when we scaled the dataset.

The result of varying the learning rate by changing the accuracy and loss values across epochs is as follows:



As we can see, even when the dataset is non-scaled, we were not able to train an accurate model when the learning rate was 0.1. Furthermore, the accuracy was not as high as in the previous section when the learning rate was 0.001.

Finally, when the learning rate was very small (0.00001), the model was able to learn as well as it did in previous sections, but this time overfitted on the training data. Let's understand why this happened by going through the parameter distributions across layers, as follows:



Here, we can see that when the model accuracy was high (which is when the learning rate was 0.00001), the weights had a much smaller range (typically ranging between -0.05 to 0.05 in this case) compared to when the learning rate was high.

> *The weights can be tuned toward a small value since the learning rate is small. Note that the scenario where the learning rate is 0.00001 on a non-scaled dataset is equivalent to the scenario of the learning rate being 0.001 on a scaled dataset. This is because the weights can now move toward a very small value (because gradient * learning rate is a very small value, given that the learning rate is small).*

Now that we have established that having a high learning rate is not likely to yield the best possible results on both scaled and non-scaled datasets, in the next section, we will learn about how to reduce the learning rate automatically when the model starts overfitting.

Generally, a learning rate of 0.001 works. Having a very low learning rate means it will take a long time to train the model, while having a high learning rate results in the model becoming unstable.

Understanding the impact of learning rate annealing

So far, we have initialized a learning rate, and it has remained the same across all the epochs while training the model. However, initially, it would be intuitive for the weights to be updated quickly to a near-optimal scenario. From then on, they should be updated very slowly since the amount of loss that gets reduced initially is high and the amount of loss that gets reduced in the later epochs would be low.

This calls for having a high learning rate initially and gradually lowering it later on as the model achieves near-optimal accuracy. This requires us to understand when the learning rate must be reduced.

One potential way we can solve this problem is by continually monitoring the validation loss and if the validation loss does not decrease (let's say, over the previous x epochs), then we reduce the learning rate.

PyTorch provides us with tools we can use to perform learning rate reduction when the validation loss does not decrease in the previous "x" epochs. Here, we can use the `lr_scheduler` method:

```
from torch import optim
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer,
                                    factor=0.5,patience=0,
                                    threshold = 0.001,
                                    verbose=True,
                                    min_lr = 1e-5,
                                    threshold_mode = 'abs')
```

In the preceding code, we are specifying that we're reducing the learning rate parameter of `optimizer` by a `factor` of 0.5 if a certain value does not improve over the next *n* epochs (where *n* is 0 in this case) by a `threshold` (which in this case is 0.001). Finally, we are specifying that the learning rate, `min_lr` (given that it is reducing by a factor of 0.5), cannot be below 1e-5 and that `threshold_mode` should be absolute to ensure that a minimum threshold of 0.001 is crossed.

Now that we have learned about the scheduler, let's apply it while training our model.

Similar to the previous sections, all the code remains the same as in the *Batch size of 32* section, except for the bold code shown here, which has been added for calculating the validation loss:

> The code for this section is available as `Learning_rate_annealing.ipynb` in the `Chapter03` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` . Note that we are not providing all the steps for brevity and that only the steps where there is a change from the code we went through in the *Batch size of 32* section will be discussed in the following code. **We encourage you to refer to the notebooks in this book's GitHub repository while executing the code.**

```
from torch import optim
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer,
                                    factor=0.5, patience=0,
                                    threshold = 0.001,
                                    verbose=True,
                                    min_lr = 1e-5,
                                    threshold_mode = 'abs')
train_losses, train_accuracies = [], []
val_losses, val_accuracies = [], []
for epoch in range(30):
    #print(epoch)
    train_epoch_losses, train_epoch_accuracies = [], []
    for ix, batch in enumerate(iter(trn_dl)):
        x, y = batch
        batch_loss = train_batch(x, y, model, optimizer, \
                                 loss_fn)
        train_epoch_losses.append(batch_loss)
    train_epoch_loss = np.array(train_epoch_losses).mean()

    for ix, batch in enumerate(iter(trn_dl)):
        x, y = batch
        is_correct = accuracy(x, y, model)
        train_epoch_accuracies.extend(is_correct)
    train_epoch_accuracy = np.mean(train_epoch_accuracies)

    for ix, batch in enumerate(iter(val_dl)):
        x, y = batch
        val_is_correct = accuracy(x, y, model)
        validation_loss = val_loss(x, y, model)
        scheduler.step(validation_loss)
```

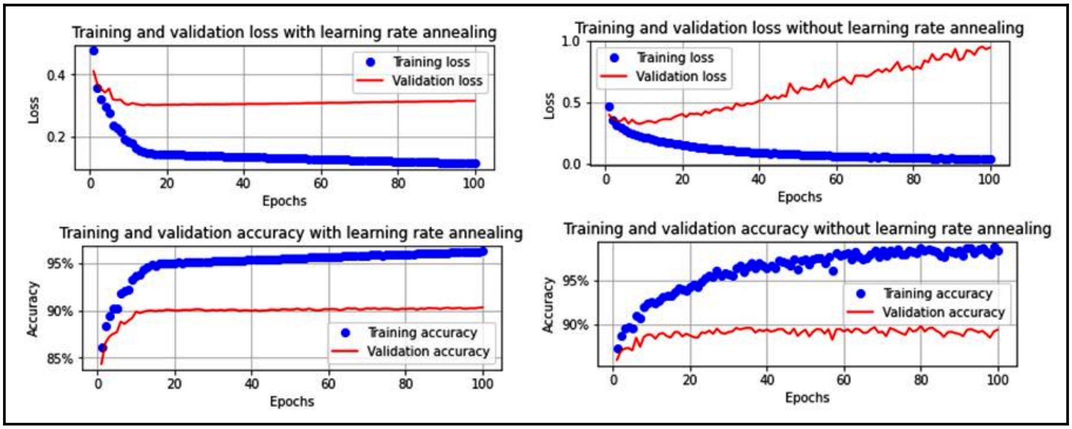
```
        val_epoch_accuracy = np.mean(val_is_correct)

        train_losses.append(train_epoch_loss)
        train_accuracies.append(train_epoch_accuracy)
        val_losses.append(validation_loss)
        val_accuracies.append(val_epoch_accuracy)
```

In the preceding code, we are specifying that the scheduler should be activated whenever the *validation loss* does not decrease over consecutive epochs. The learning rate is reduced by a factor of 0.5 x the current learning rate in those cases.

The output of performing this on our model is as follows:

```
Epoch     5: reducing learning rate of group 0 to 5.0000e-04.
Epoch     8: reducing learning rate of group 0 to 2.5000e-04.
Epoch    11: reducing learning rate of group 0 to 1.2500e-04.
Epoch    12: reducing learning rate of group 0 to 6.2500e-05.
Epoch    13: reducing learning rate of group 0 to 3.1250e-05.
Epoch    15: reducing learning rate of group 0 to 1.5625e-05.
Epoch    16: reducing learning rate of group 0 to 1.0000e-05.
```

Let's understand the variation in the training and validation datasets' accuracy and loss values over increasing epochs:



Note that the learning rate reduced by half whenever the validation loss increased by at least 0.001 over increasing epochs. This happened in epochs 5, 8, 11, 12, 13, 15, and 16.

Furthermore, we did not have any huge overfitting issues, even though we trained the model for 100 epochs. This is because the learning rate became so small that the weight update was very small, resulting in a smaller gap between the training and validation accuracies (when compared with the scenario where we had 100 epochs without learning rate annealing, where the training accuracy was close to 100% while the validation accuracy was close to ~89%).

So far, we have learned about the impact various hyperparameters have on the accuracy of a model. In the next section, we will learn about how the number of layers in a neural network impacts its accuracy.

Building a deeper neural network

So far, our neural network architecture only has one hidden layer. In this section, we will contrast the performance of models where there are two hidden layers and no hidden layer (with no hidden layer being a logistic regression).

A model with two layers within a network can be built as follows (note that we have kept the number of units in the second hidden layer set to 1,000). The modified `get_model` function (from the code in the *Batch size of 32* section), where there are two hidden layers, is as follows:

> The following code is available as `Impact_of_building_a_deeper_neural_network.ipynb` in the `Chapter03` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` . Note that we are not providing all the steps for brevity and that only the steps where there is a change from the code we went through in the *Batch size of 32* section will be discussed in the following code. **We encourage you to refer to the notebooks in this book's GitHub repository while executing the code.**

```
def get_model():
    model = nn.Sequential(
                nn.Linear(28 * 28, 1000),
                nn.ReLU(),
                nn.Linear(1000, 1000),
                nn.ReLU(),
                nn.Linear(1000, 10)
            ).to(device)

    loss_fn = nn.CrossEntropyLoss()
```

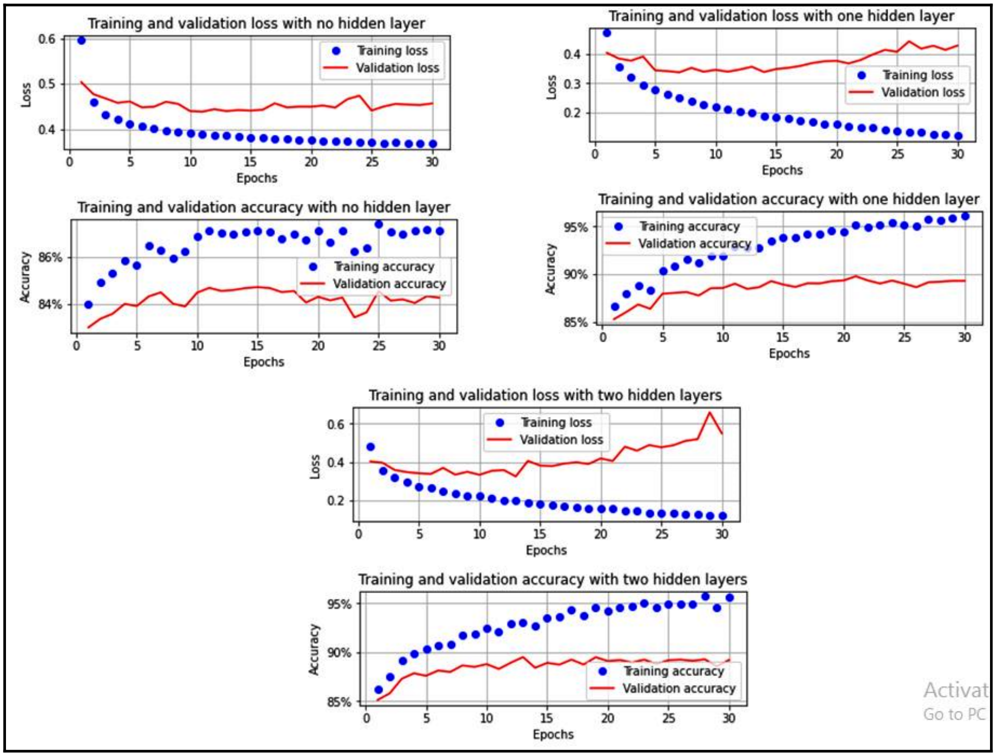
```
        optimizer = Adam(model.parameters(), lr=1e-3)
        return model, loss_fn, optimizer
```

Similarly, the `get_model` function, where there are *no hidden layers*, is as follows:

```
    def get_model():
        model = nn.Sequential(
                    nn.Linear(28 * 28, 10)
                ).to(device)

        loss_fn = nn.CrossEntropyLoss()
        optimizer = Adam(model.parameters(), lr=1e-3)
        return model, loss_fn, optimizer
```

Note that in the preceding function, we are connecting the input directly to the output layer.

Once we train the models as we did in the *Batch size of 32* section, the accuracy and loss on the train and validation datasets will be as follows:



Here, take note of the following:

- The model was unable to learn as well as when there were no hidden layers.
- The model overfit by a larger amount when there were two hidden layers compared to one hidden layer (the validation loss is higher in the model with two layers compared to the model with one layer).

So far, across different sections, we have seen that the model was unable to be trained well when the input data wasn't scaled (brought down to a small range). Non-scaled data (data with a higher range) can also occur in hidden layers (especially when we have deep neural networks with multiple hidden layers) because of the matrix multiplication that's involved in getting the values of nodes in hidden layers. In the next section, we will learn how to deal with such non-scaled data in intermediate layers.

Understanding the impact of batch normalization

Previously, we learned that when the input value is large, the variation of the Sigmoid output doesn't make much difference when the weight values change considerably.

Now, let's consider the opposite scenario, where the input values are very small:

Input	Weight	Sigmoid
0.01	0.00001	0.500
0.01	0.0001	0.500
0.01	0.001	0.500
0.01	0.01	0.500
0.01	0.1	0.500
0.01	0.2	0.500
0.01	0.3	0.501
0.01	0.4	0.501
0.01	0.5	0.501
0.01	0.6	0.501
0.01	0.7	0.502
0.01	0.8	0.502
0.01	0.9	0.502
0.01	1	0.502

When the input value is very small, the Sigmoid output changes slightly, making a big change to the weight value.

Additionally, in the *Scaling the input data* section, we saw that large input values have a negative effect on training accuracy. This suggests that we can neither have very small nor very big values for our input.

Along with very small or very big values in input, we may also encounter a scenario where the value of one of the nodes in the hidden layer could result in either a very small number or a very large number, resulting in the same issue we saw previously with the weights connecting the hidden layer to the next layer.

Batch normalization comes to the rescue in such a scenario since it normalizes the values at each node, just like when we scaled our input values.

Typically, all the input values in a batch are scaled as follows:

$$Batch\ mean\ \mu_B \ = \frac{1}{m}\sum_{i=1}^{m}\ x_i$$

$$Batch\ Variance\ \sigma_2^B \ = \frac{1}{m}\sum_{i=1}^{m}\ (x_i - \mu_B)^2$$

$$Normalized\ input\ \bar{x}_i \ = \frac{(x_i - \mu_B)}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$Batch\ normalized\ input\ = \ \gamma\ \bar{x}_i\ + \beta$$

By subtracting each data point from the batch mean and then dividing it by the batch variance, we have normalized all the data points of the batch at a node to a fixed range.

While this is known as hard normalization, by introducing the γ and β parameters, we are letting the network identify the best normalization parameters.

To understand how the batch normalization process helps, let's take a look at the loss and accuracy values on the training and validation datasets, as well as the distribution of hidden layer values, in the following scenarios:

- Very small input values without batch normalization
- Very small input values with batch normalization

Let's get started!

Very small input values without batch normalization

So far, when we had to scale input data, we scaled it to a value between 0 and 1. In this section, we will scale it further to a value between 0 and 0.0001 so that we can understand the impact of scaling data. As we saw at the beginning of this section, small input values could not change the Sigmoid value, even with a big variation in weight values.

To scale the input dataset so that it has a very low value, we'll change the scaling that typically we do in the `FMNISTDataset` class by reducing the range of input values from 0 to 0.0001, as follows:

> The following code is available as `Batch_normalization.ipynb` in the `Chapter03` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` . Note that we are not providing all the steps for brevity and that only the steps where there is a change from the code we went through in the *Batch size of 32* section will be discussed in the following code. **We encourage you to refer to the notebooks in this book's GitHub repository while executing the code.**

```
class FMNISTDataset(Dataset):
    def __init__(self, x, y):
        x = x.float()/(255*10000) # Done only for us to
        # understand the impact of Batch normalization
        x = x.view(-1,28*28)
        self.x, self.y = x, y
    def __getitem__(self, ix):
        x, y = self.x[ix], self.y[ix]
        return x.to(device), y.to(device)
    def __len__(self):
        return len(self.x)
```

Note that in the bold section of code (`x = x.float()/(255*10000)`), we have reduced the range of input pixel values by dividing them by 10,000.

Next, we must redefine the `get_model` function so that we can fetch the model's prediction, as well as the values for the hidden layer. We can do this by specifying a neural network class, as follows:

```
def get_model():
    class neuralnet(nn.Module):
        def __init__(self):
            super().__init__()
            self.input_to_hidden_layer = nn.Linear(784,1000)
            self.hidden_layer_activation = nn.ReLU()
            self.hidden_to_output_layer = nn.Linear(1000,10)
        def forward(self, x):
            x = self.input_to_hidden_layer(x)
            x1 = self.hidden_layer_activation(x)
            x2= self.hidden_to_output_layer(x1)
            return x2, x1
    model = neuralnet().to(device)
    loss_fn = nn.CrossEntropyLoss()
    optimizer = Adam(model.parameters(), lr=1e-3)
    return model, loss_fn, optimizer
```

In the preceding code, we defined the `neuralnet` class, which returns the output layer values (`x2`) and the hidden layer's activation values (`x1`). Note that the architecture of the network hasn't changed.

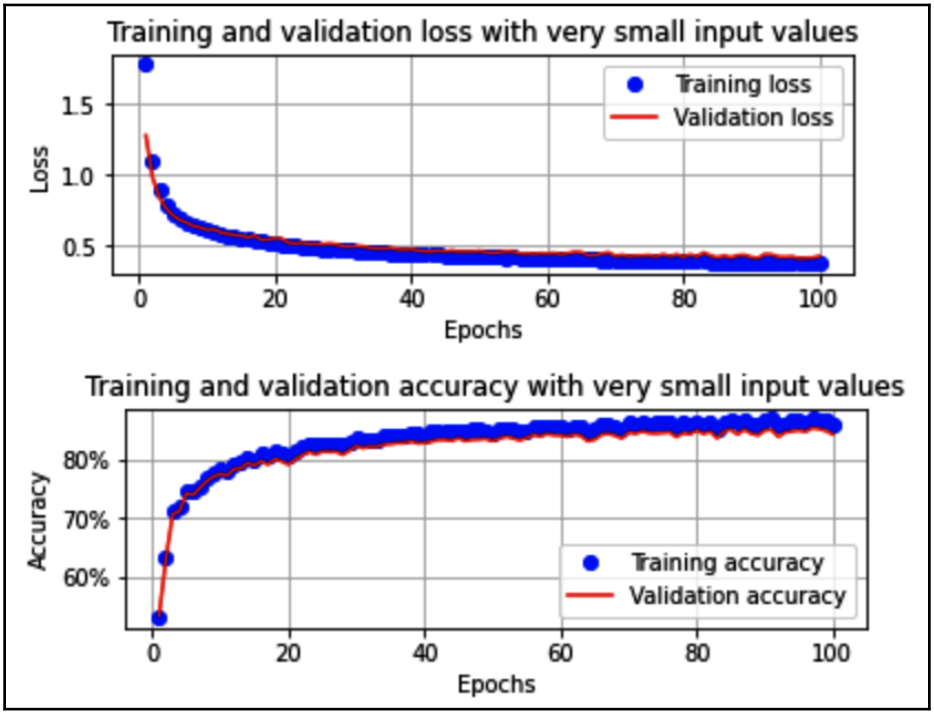
Given that the `get_model` function returns two outputs now, we need to modify the `train_batch` and `val_loss` functions, which make predictions, by passing input through the model. Here, we'll only fetch the output layer values, not the hidden layer values. Given that the output layer values are in the 0^{th} index of what is returned from the model, we'll modify the functions so that they only fetch the 0^{th} index of predictions, as follows:

```
def train_batch(x, y, model, opt, loss_fn):
    model.train()
    prediction = model(x)[0]
    batch_loss = loss_fn(prediction, y)
    batch_loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    return batch_loss.item()

def accuracy(x, y, model):
    model.eval()
```

```
    with torch.no_grad():
        prediction = model(x)[0]
    max_values, argmaxes = prediction.max(-1)
    is_correct = argmaxes == y
    return is_correct.cpu().numpy().tolist()
```

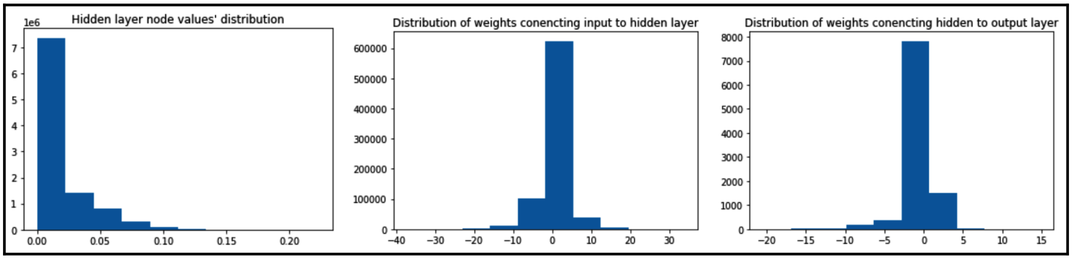
Note that the bold section in the preceding code is where we have ensured we only fetch the 0^{th} index of the model's output (since the 0^{th} index contains the output layer's values).

Now, when we run the rest of the code provided in the *Scaling the data* section, we'll see that the variation in the accuracy and loss values in the training and validation datasets over increasing epochs is as follows:



Note that in the preceding scenario, the model didn't train well, even after 100 epochs (the model was trained to an accuracy of ~90% on the validation dataset within 10 epochs in the previous sections, while the current model only has ~85% validation accuracy).

Let's understand the reason why the model doesn't train as well when the input values have a very small range by exploring the hidden values' distribution, as well as the parameter distribution:



Note that the first distribution indicates the distribution of values in the hidden layer (where we can see that the values have a very small range). Furthermore, given that both the input and hidden layer values have a very small range, the weights had to be varied by a large amount (for both the weights that are connecting the input to the hidden layer and the weights that are connecting the hidden layer to the output layer).

Now that we understand that the network doesn't train well when the input values have a very small range, let's understand how batch normalization helps increase the range of values within the hidden layer.

Very small input values with batch normalization

In this section, we'll only be making one change to the code from the previous subsection; that is, we'll be adding batch normalization while defining the model architecture.

The modified `get_model` function is as follows:

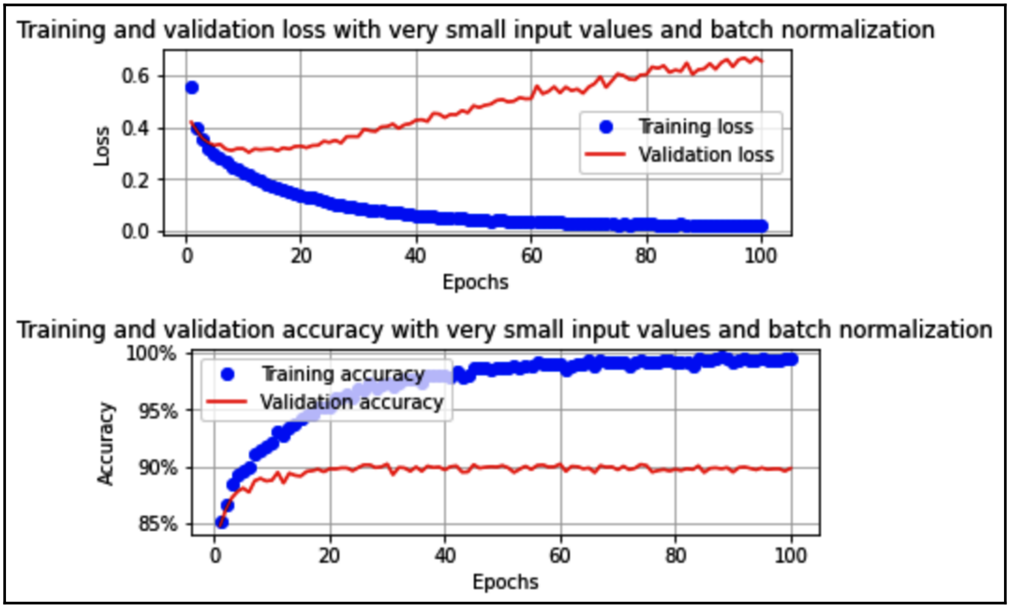
```
def get_model():
    class neuralnet(nn.Module):
        def __init__(self):
            super().__init__()
            self.input_to_hidden_layer = nn.Linear(784,1000)
            self.batch_norm = nn.BatchNorm1d(1000)
            self.hidden_layer_activation = nn.ReLU()
            self.hidden_to_output_layer = nn.Linear(1000,10)
        def forward(self, x):
```

```
            x = self.input_to_hidden_layer(x)
            x0 = self.batch_norm(x)
            x1 = self.hidden_layer_activation(x0)
            x2= self.hidden_to_output_layer(x1)
            return x2, x1
    model = neuralnet().to(device)
    loss_fn = nn.CrossEntropyLoss()
    optimizer = Adam(model.parameters(), lr=1e-3)
    return model, loss_fn, optimizer
```

Note that in the preceding code, we declared a variable (`batch_norm`) that performs batch normalization (`nn.BatchNorm1d`). The reason we are performing `nn.BatchNorm1d(1000)` is because the output dimension is 1,000 for each image (that is, a 1-dimensional output for the hidden layer).

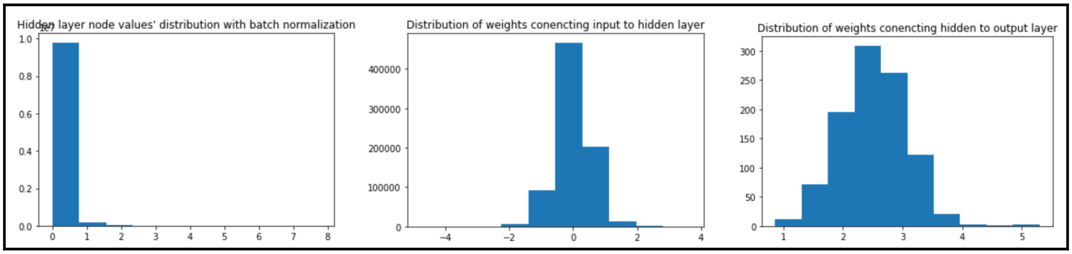
Furthermore, in the `forward` method, we are passing the output of the hidden layer values through batch normalization, prior to ReLU activation.

The variation in the training and validation datasets' accuracy and loss values over increasing epochs is as follows:



Here, we can see that the model was trained in a manner very similar to how it was trained when the input values did not have a very small range.

Let's understand the distribution of hidden layer values and the weight distribution, as seen in the previous section:



Here, we can see that the hidden layer values have a larger distribution when we have batch normalization and that the weights connecting the hidden layer to the output layer have a smaller distribution. The results in the model learning as effectively as it could in the previous sections.

> Batch normalization helps considerably when training deep neural networks. It helps us avoid gradients becoming so small that the weights are barely updated.

Note that in the preceding scenario, we attained high validation accuracy sooner than when there was no batch normalization at all. This could have been the result of normalizing the intermediate layers, resulting in fewer chances of saturation occurring in the weights. However, the issue of overfitting is yet to be fixed. We will look at this next.

The concept of overfitting

So far, we've seen that the accuracy of the training dataset is typically more than 95%, while the accuracy of the validation dataset is ~89%.

Essentially, this indicates that the model does not generalize as much on unseen datasets since it can learn from the training dataset. This also indicates that the model is learning all the possible edge cases for the training dataset; these can't be applied to the validation dataset.

Having high accuracy on the training dataset and considerably lower accuracy on the validation dataset refers to the scenario of overfitting.

Some of the typical strategies that are employed to reduce the effect of overfitting are as follows:

- Dropout
- Regularization

We will look at what impact they have in the following sections.

Impact of adding dropout

We have already learned that whenever `loss.backward()` is calculated, a weight update happens. Typically, we would have hundreds of thousands of parameters within a network and thousands of data points to train our model on. This gives us the possibility that while the majority of parameters help in training the model reasonably, certain parameters can be fine-tuned for the training images, resulting in their values being dictated by only a few images in the training dataset. This, in turn, results in the training data having a high accuracy but not that the validation dataset generalizes.

Dropout is a mechanism that randomly chooses a specified percentage of activations and drops them to 0. In the next iteration, another random set of hidden units are switched off. This way, the neural network does not optimize for edge cases as the network does not get that many opportunities to adjust the weight to memorize for edge cases (given that the weight is not updated in each iteration).

Note that, during prediction, dropout doesn't need to be applied since this mechanism can only be applied to a trained model. Furthermore, the weights will be downscaled automatically during prediction (evaluation) to adjust for the magnitude of the weights (since all the weights are present during prediction time).

Usually, there are cases where the layers behave differently during training and validation – as you saw in the case of dropout. For this reason, you must specify the mode for the model upfront using one of two methods – `model.train()` to let the model know it is in training mode and `model.eval()` to let it know that it is in evaluation mode. If we don't do this, we might get unexpected results. For example, in the following image, notice how the model (which contains dropout) gives us different predictions on the same input when in training mode. However, when the same model is in eval mode, it will suppress the dropout layer and return the same output:

```
[20]
model.train()
batch = next(iter(trn_dl))
for i in range(5):
    output = model(batch[0])
    print(output.mean().item())
```

```
-13.275323867797852
-12.834677696228027
-11.895054817199707
-12.713885307312012
-13.302783012390137
```

```
model.eval()
batch = next(iter(trn_dl))
for i in range(5):
    output = model(batch[0])
    print(output.mean().item())
```

```
-12.40117359161377
-12.40117359161377
-12.40117359161377
-12.40117359161377
-12.40117359161377
```

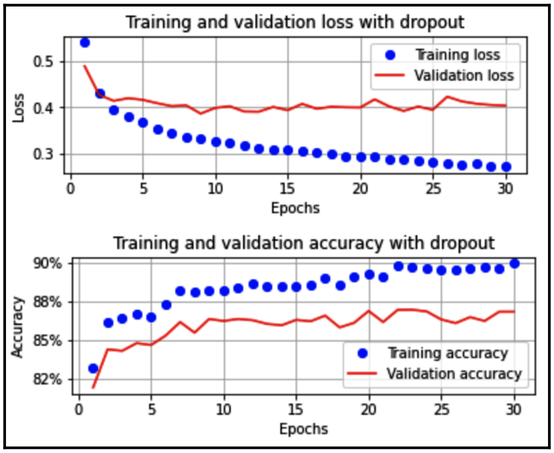
While defining the architecture, `Dropout` is specified in the `get_model` function as follows:

The following code is available as `Impact_of_dropout.ipynb` in the `Chapter03` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` . Note that we are not providing all the steps for brevity and that only the steps where there is a change from the code we went through in the *Batch size of 32* section will be discussed in the following code. **We encourage you to refer to the notebooks in this book's GitHub repository while executing the code.**

```
def get_model():
    model = nn.Sequential(
                nn.Dropout(0.25),
                nn.Linear(28 * 28, 1000),
                nn.ReLU(),
                nn.Dropout(0.25),
                nn.Linear(1000, 10)
            ).to(device)

    loss_fn = nn.CrossEntropyLoss()
    optimizer = Adam(model.parameters(), lr=1e-3)
    return model, loss_fn, optimizer
```

Note that in the preceding code, `Dropout` is specified before linear activation is. This specifies that a fixed percentage of the weights in the linear activation layer won't be updated.

Once the model training is completed, as in the *Batch size of 32* section, the loss and accuracy values of the training and validation datasets will be as follows:



Note that in the preceding scenario, the delta between the training and validation datasets' accuracy is not as large as we saw in the previous scenario, thus resulting in a scenario that has less overfitting.

Impact of regularization

Apart from the training accuracy being much higher than the validation accuracy, one other feature of overfitting is that certain weight values will be much higher than the other weight values. High weight values can be a symptom of the model learning very well on training data (essentially, a rot learning on what it has seen).

While dropout is a mechanism that's used so that the weight values aren't updated as frequently, regularization is another mechanism we can use for this purpose.

Regularization is a technique in which we penalize the model for having high weight values. Hence, it is a dual objective function – minimize the loss of training data, as well as the weight values. In this section, we will learn about two types of regularization:

- L1 regularization
- L2 regularization

The following code is available as `Impact_of_regularization.ipynb` in the `Chapter03` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` . Note that we are not providing all the steps for brevity and that only the steps where there is a change from the code we went through in the *Batch size of 32* section will be discussed in the following code. **We encourage you to refer to the notebooks in this book's GitHub repository while executing the code.**

Let's get started!

L1 regularization

L1 regularization is calculated as follows:

$$L1\ loss = -\frac{1}{n}\left(\sum_{i=1}^{n}(y_i * log(p_i) + (1-y_i) * log(1-p_i)) + \Lambda\sum_{j=1}^{m}|w_j|\right)$$

The first part of the preceding formula refers to the categorical cross-entropy loss that we have been using for optimization so far, while the second part refers to the absolute sum of the weight values of the model.

Note that L1 regularization ensures that it penalizes for the high absolute values of weights by incorporating them in the loss value calculation.

Λ refers to the weightage that we associate with the regularization (weight minimization) loss.

L1 regularization is implemented while training the model, as follows:

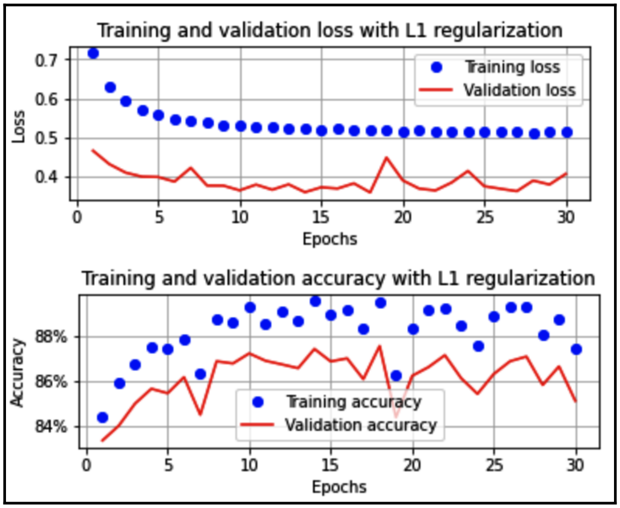
```
def train_batch(x, y, model, opt, loss_fn):
    model.train()
    prediction = model(x)
    l1_regularization = 0
    for param in model.parameters():
        l1_regularization += torch.norm(param,1)
    batch_loss = loss_fn(prediction, y)+0.0001*l1_regularization
    batch_loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    return batch_loss.item()
```

In the preceding code, we have enforced regularization on the weights and biases across all the layers by initializing `l1_regularization`.

`torch.norm(param,1)` provides the absolute value of the weight and bias values across layers.

Furthermore, we have a very small weightage (`0.0001`) associated with the sum of the absolute value of the parameters across layers.

Once we execute the remaining code, as in the *Batch size of 32* section, the training and validation datasets' loss and accuracy values over increasing epochs will be as follows:



Here, we can see that the difference between the training and validation datasets' accuracy is not as high as it was without L1 regularization.

L2 regularization

L2 regularization is calculated as follows:

$$L2\ loss = -\frac{1}{n}(\sum_{i=1}^{n}(y_i * log(p_i) + (1-y_i) * log(1-p_i)) + \Lambda\sum_{j=1}^{m} w_j^2)$$

The first part of the preceding formula refers to the categorical cross-entropy loss obtained, while the second part refers to the squared sum of the weight values of the model.

Similar to L1 regularization, we are penalizing for high weight values by having the sum of squared values of weights incorporated in the loss value calculation.

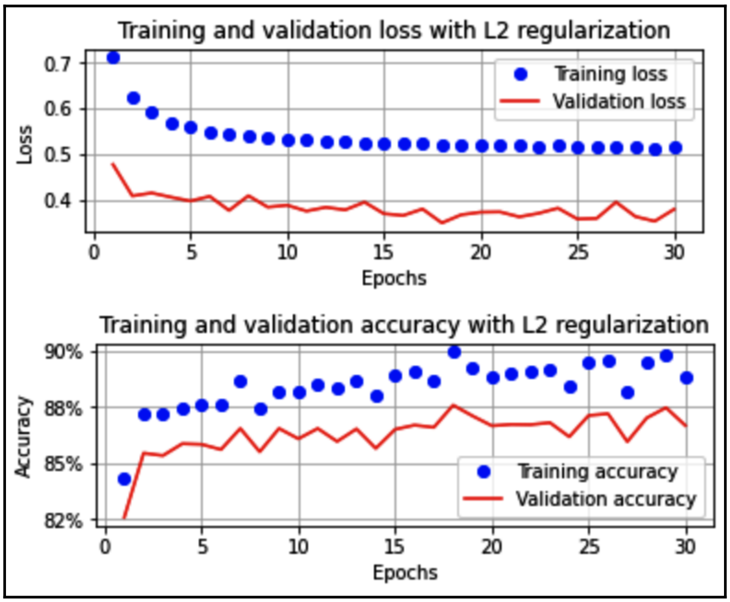
Λ refers to the weightage that we associate with the regularization (weight minimization) loss.

L2 regularization is implemented while training the model as follows:

```
def train_batch(x, y, model, opt, loss_fn):
    model.train()
    prediction = model(x)
    l2_regularization = 0
    for param in model.parameters():
        l2_regularization += torch.norm(param,2)
    batch_loss = loss_fn(prediction, y) + 0.01*l2_regularization
    batch_loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    return batch_loss.item()
```

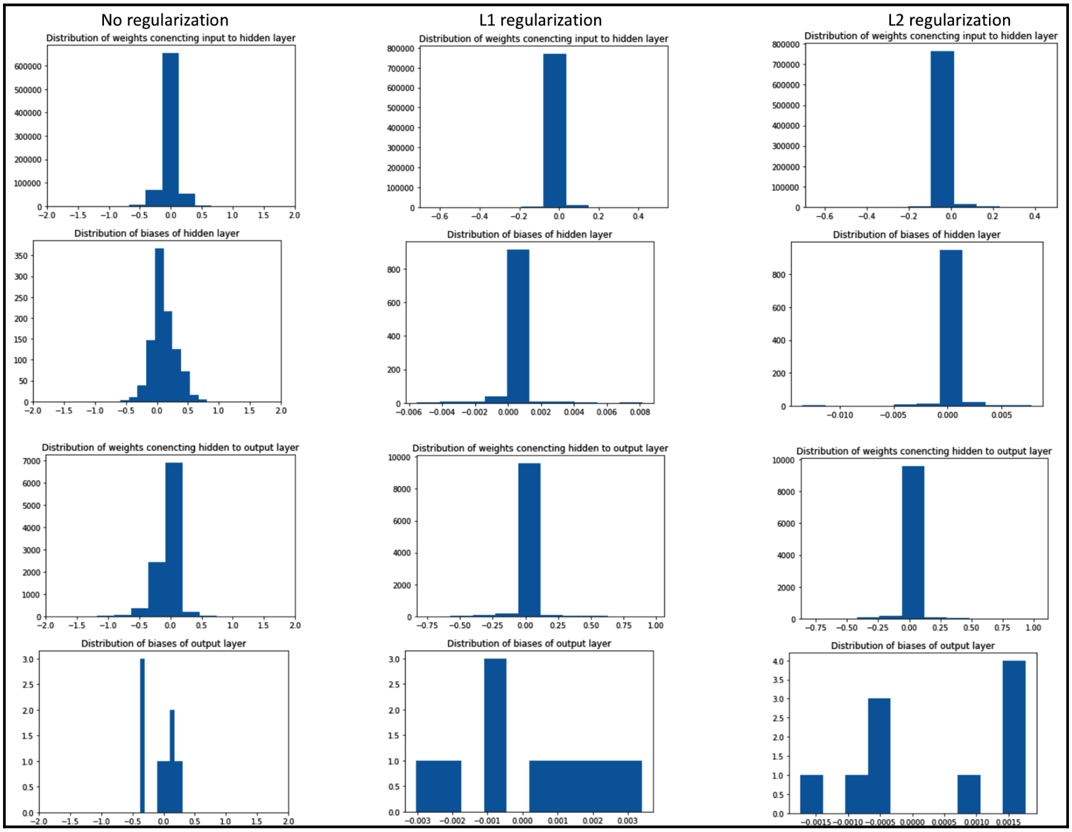
Note that in the preceding code, the regularization parameter, Λ (0.01), is slightly higher than in L1 regularization since the weights are generally between -1 to 1 and a square of them would result in even smaller values. Multiplying them by an even smaller number, as we did in L1 regularization, would result in us having very little weightage for regularization in the overall loss calculation.

Once we execute the remaining code, as in the *Batch size of 32* section, the training and validation datasets' loss and accuracy values over increasing epochs will be as follows:



Here, we can see that L2 regularization has also resulted in the validation and training datasets' accuracy and loss values being close to each other.

Finally, let's compare the weight values without regularization and with L1/ L2 regularization so that we can validate our understanding that certain weights vary considerably when it comes to memorizing the values for edge cases. We'll do this by going through the distribution of weights across layers, as shown in the following image:



Here, we can see that the distribution of parameters is very small when we perform L1/ L2 regularization compared to performing no regularization. This potentially reduces the chances that weights get updated for edge cases.

Summary

We started this chapter by learning about how an image is represented. Next, we learned about how scaling, the value of the learning rate, our choice of optimizer, and the batch size help improve the accuracy and speed of training. We then learned about how batch normalization helps in increasing the speed of training and addresses the issues of very small or large values in hidden layer. Next, we learned about scheduling the learning rate to increase accuracy further. We then proceeded to understand the concept of overfitting and learned about how dropout and L1 and L2 regularization help us avoid overfitting.

Now that we have learned about image classification using a deep neural network, as well as the various hyperparameters that help train a model, in the next chapter, we will learn about how what we've learned in this chapter can fail and how to address this using convolutional neural networks.

Questions

1. What happens if the input values are not scaled in the input dataset?
2. What could happen if the background has a white pixel color while the content has a black pixel color when you're training a neural network?
3. What impact does the batch size have on the model's training time, as well as its accuracy over a given number of epochs?
4. What impact does the input value range have on the weight distribution at the end of the training?
5. How does batch normalization help improve accuracy?
6. How do we know if a model has overfitted on training data?
7. How does regularization help in avoiding overfitting?
8. How do L1 and L2 regularization differ?
9. How does dropout help in reducing overfitting?

2
Section 2 - Object Classification and Detection

Armed with an understanding of **neural network** (**NN**) basics, in this section, we will discover more complex blocks of NNs that build on top of these basics to solve more complex vision-related issues, including object detection, image classification, and many more problems besides.

This section comprises the following chapters:

- `Chapter 4`, *Introducing Convolutional Neural Networks*
- `Chapter 5`, *Transfer Learning for Image Classification*
- `Chapter 6`, *Practical Aspects of Image Classification*
- `Chapter 7`, *Basics of Object Detection*
- `Chapter 8`, *Advanced Object Detection*
- `Chapter 9`, *Image Segmentation*
- `Chapter 10`, *Applications of Object Detection and Segmentation*

4
Introducing Convolutional Neural Networks

So far, we've learned how to build deep neural networks and the impact of tweaking their various hyperparameters. In this chapter, we will learn about where traditional deep neural networks do not work. We'll then learn about the inner workings of **convolutional neural networks** (**CNNs**) by using a toy example before understanding some of their major hyperparameters, including strides, pooling, and filters. Next, we will leverage CNNs, along with various data augmentation techniques, to solve the issue of traditional deep neural networks not having good accuracy. Following this, we will learn about what the outcome of a feature learning process in a CNN looks like. Finally, we'll put our learning together to solve a use case: we'll be classifying an image by stating whether the image contains a dog or a cat. By doing this, we'll be able to understand how the accuracy of prediction varies by the amount of data available for training.

The following topics will be covered in this chapter:

- The problem with traditional deep neural networks
- Building blocks of a CNN
- Implementing a CNN
- Classifying images using deep CNNs
- Implementing data augmentation
- Visualizing the outcome of feature learning
- Building a CNN for classifying real-world images

Let's get started!

The problem with traditional deep neural networks

Before we dive into CNNs, let's look at the major problem that's faced when using traditional deep neural networks.

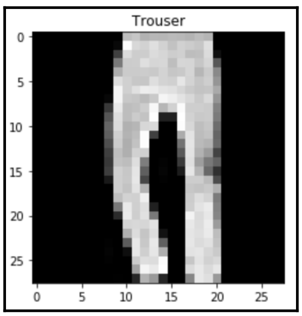
Let's reconsider the model we built on the Fashion-MNIST dataset in `Chapter 3`, *Building a Deep Neural Network with PyTorch*. We will fetch a random image and predict the class that corresponds to that image, as follows:

> The code for this section is available as `Issues_with_image_translation.ipynb` in the `Chapter04` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` . Note that the entire code is available in GitHub and that only the additional code corresponding to the issue of image translation will be discussed here for brevity. We strongly encourage you to refer to the notebooks in this book's GitHub repository while executing the code.

1. Fetch a random image from the available training images:

```
# Note that you should run the code in
# Batch size of 32 section in Chapter 3
# before running the following code
import matplotlib.pyplot as plt
%matplotlib inline
# ix = np.random.randint(len(tr_images))
ix = 24300
plt.imshow(tr_images[ix], cmap='gray')
plt.title(fmnist.classes[tr_targets[ix]])
```

The preceding code results in the following output:



2. Pass the image through the **trained model** (continue using the model we trained in the *Batch size of 32* section of `Chapter 3`, *Building a Deep Neural Network with PyTorch*).

    - Preprocess the image so it goes through the same pre-processing steps we performed while building the model:

    ```
    img = tr_images[ix]/255.
    img = img.view(28*28)
    img = img.to(device)
    ```

    - Extract the probabilities associated with the various classes:

    ```
    np_output = model(img).cpu().detach().numpy()
    np.exp(np_output)/np.sum(np.exp(np_output))
    ```

    The preceding code results in the following output:

```
array([1.7608897e-08, 1.0000000e+00, 2.6042574e-13, 1.1353759e-10,
       3.1050048e-12, 7.2957764e-16, 8.0109371e-11, 3.8039204e-22,
       1.2800090e-15, 2.8759430e-18], dtype=float32)
```

   From the preceding output, we can see that the highest probability is for the 1st index, which is of the `Trouser` class.

3. Translate (roll/slide) the image multiple times (one pixel at a time) from a translation of 5 pixels to the left to 5 pixels to the right and store the predictions in a list.

    - Create a list that stores predictions:

    ```
    preds = []
    ```

    - Create a loop that translates (rolls) an image from -5 pixels (5 pixels to the left) to +5 pixels (5 pixels to the right) of the original position (which is at the center of the image):

    ```
    for px in range(-5,6):
    ```

    In the preceding code, we specified 6 as the upper bound, even though we are interested in translating until +5 pixels, since the output of the range would be from -5 to +5 when (-5,6) is the specified range.

- Pre-process the image, as we did in *step 2*:

```
img = tr_images[ix]/255.
img = img.view(28, 28)
```

- Roll the image by a value equal to `px` within the `for` loop:

```
img2 = np.roll(img, px, axis=1)
```

In the preceding code, we specified `axis=1` since we want the image pixels to be moving horizontally and not vertically.

- Store the rolled image as a tensor object and register it to `device`:

```
img3 = torch.Tensor(img2).view(28*28).to(device)
```

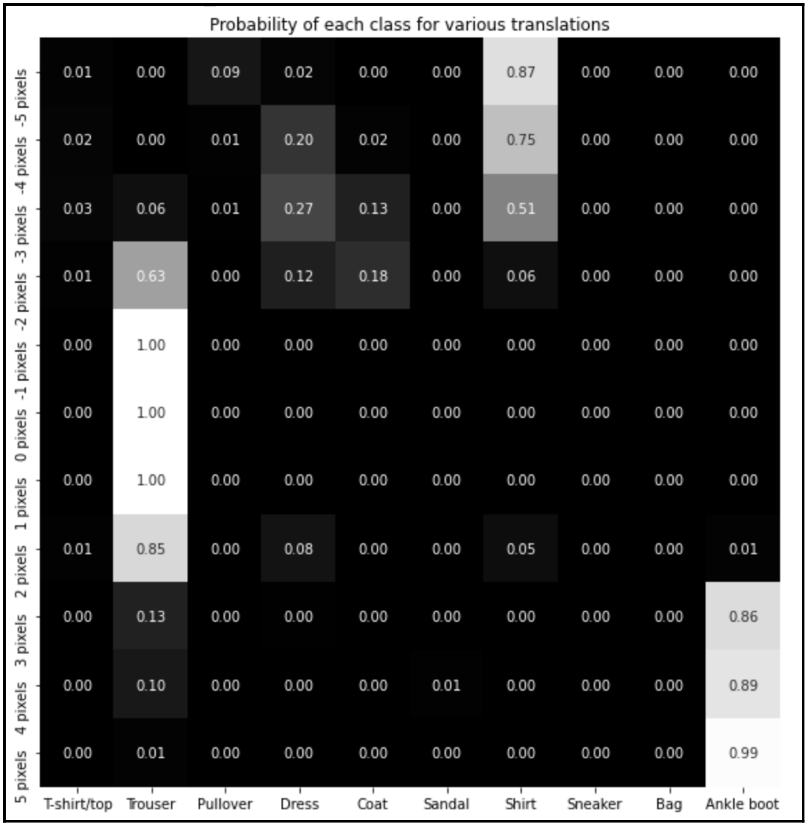
- Pass `img3` through the trained model to predict the class of the translated (rolled) image and append it to the list that is storing predictions for various translations:

```
np_output = model(img3).cpu().detach().numpy()
preds.append(np.exp(np_output)/np.sum(np.exp(np_output)))
```

4. Visualize the predictions of the model for all the translations (-5 pixels to +5 pixels):

```
import seaborn as sns
fig, ax = plt.subplots(1,1, figsize=(12,10))
plt.title('Probability of each class \
for various translations')
sns.heatmap(np.array(preds), annot=True, ax=ax, fmt='.2f', \
            xticklabels=fmnist.classes, \
            yticklabels=[str(i)+str(' pixels') \
                         for i in range(-5,6)], cmap='gray')
```

The preceding code results in the following output:



There was no change in the image's content since we only translated the images from 5 pixels to the left and 5 pixels to the right. However, the predicted class of the image changed when the translation was beyond 2 pixels. This is because while the model was being trained, the content in all the training and testing images was at the center. This differs from the preceding scenario where we tested with translated images that are off-center, resulting in an incorrectly predicted class.

Now that we have learned about a scenario where a traditional neural network fails, we will learn about how **CNNs** help address this problem. But before we do this, we will learn about the building blocks of a CNN.

Building blocks of a CNN

CNNs are the most prominent architectures that are used when working on images. CNNs address the major limitations of deep neural networks that we saw in the previous section. Besides image classification, they also help with object detection, image segmentation, GANs, and many more – essentially, wherever we use images. Furthermore, there are different ways of constructing a convolutional neural network, and there are multiple pre-trained models that leverage CNNs to perform various tasks. Starting with this chapter, we will be using CNNs extensively.

In the upcoming subsections, we will understand the fundamental building blocks of a CNN, which are as follows:

- Convolutions
- Filters
- Strides and padding
- Pooling

Let's get started!

Convolution

A convolution is basically multiplication between two matrices. As you saw in the previous chapter, matrix multiplication is a key ingredient of training a neural network. (We perform matrix multiplication when we calculate hidden layer values – which is a matrix multiplication of the input values and weight values connecting the input to the hidden layer. Similarly, we perform matrix multiplication to calculate output layer values.)

To ensure we have a solid understanding of the convolution process, let's go through the following example.

Let's assume we have two matrices we can use to perform convolution.

Here is Matrix A:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Here is Matrix B:

1	2
3	4

While performing the convolution operation, you are sliding Matrix B (the smaller matrix) over Matrix A (the bigger matrix). Furthermore, we are performing element to element multiplication between Matrix A and Matrix B, as follows:

1. Multiply {1,2,5,6} of the bigger matrix by {1,2,3,4} of the smaller matrix:

   *1*1 + 2*2 + 5*3 + 6*4 = 44*

2. Multiply {2,3,6,7} of the bigger matrix by {1,2,3,4} of the smaller matrix:

   *2*1 + 3*2 + 6*3 + 7*4 = 54*

3. Multiply {3,4,7,8} of the bigger matrix by {1,2,3,4} of the smaller matrix:

   *3*1 + 4*2 + 7*3 + 8*4 = 64*

4. Multiply {5,6,9,10} of the bigger matrix by {1,2,3,4} of the smaller matrix:

   *5*1 + 6*2 + 9*3 + 10*4 = 84*

5. Multiply {6,7,10,11} of the bigger matrix by {1,2,3,4} of the smaller matrix:

   *6*1 + 7*2 + 10*3 + 11*4 = 94*

6. Multiply {7,8,11,12} of the bigger matrix by {1,2,3,4} of the smaller matrix:

   *7*1 + 8*2 + 11*3 + 12*4 = 104*

7. Multiply {9,10,13,14} of the bigger matrix by {1,2,3,4} of the smaller matrix:

   *9*1 + 10*2 + 13*3 + 14*4 = 124*

8. Multiply {10,11,14,15} of the bigger matrix by {1,2,3,4} of the smaller matrix:

   *10*1 + 11*2 + 14*3 + 15*4 = 134*

9. Multiply {11,12,15,16} of the bigger matrix by {1,2,3,4} of the smaller matrix:

   *11*1 + 12*2 + 15*3 + 16*4 = 144*

The result of performing the preceding operations is as follows:

44	54	64
84	94	104
124	134	144

The smaller matrix is typically called a **filter** or a kernel, while the bigger matrix is the original image.

Filter

A filter is a matrix of weights that is initialized randomly at the start. The model learns the optimal weight values of a filter over increasing epochs.

The concept of filters brings us to two different aspects:

- What the filters learn about
- How filters are represented

In general, the more filters there are in a CNN, the more features of an image that the model can learn about. We will learn about what various filters learn in the *Visualizing the filters' learning* section of this chapter. For now, we'll ensure that we have an intermediate understanding that the filters learn about different features present in the image. For example, a certain filter might learn about the ears of a cat and provide high activation (a matrix multiplication value) when the part of the image it is convolving with contains the ear of a cat.

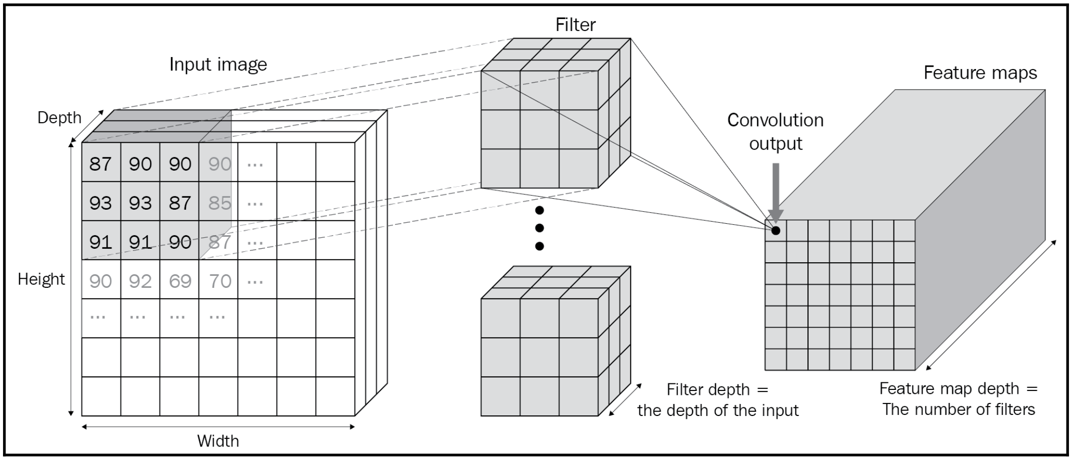
In the previous section, we learned that when we convolved one filter that has a size of 2 x 2 with a matrix that has a size of 4 x 4, we got an output that is 3 x 3 in dimension.

However, if 10 different filters multiply the bigger matrix (original image), the result is 10 sets of the 3 x 3 output matrices.

In the preceding case, a 4 x 4 image is convolved with 10 filters that are 2 x 2 in size, resulting in 3 x 3 x 10 output values. Essentially, when an image is convolved by multiple filters, the output has as many channels as there are filters that the image is convolved with.

Furthermore, in a scenario where we are dealing with color images where there are three channels, the filter that is convolving with the original image would also have three channels, resulting in a single scalar output per convolution. Also, if the filters are convolving with an intermediate output – let's say of 64 x 112 x 112 in shape – the filter would have 64 channels to fetch a scalar output. In addition, if there are 512 filters that are convolving with the output that was obtained in the intermediate layer, the output post convolution with 512 filters would be 512 x 111 x 111 in shape.

To solidify our understanding of the output of filters further, let's take a look at the following diagram:



In the preceding diagram, we can see that the input image is multiplied by the filters that have the same depth as that of the input (which the filters are convolving with) and that the number of channels in the output of a convolution is as many as there are filters.

Strides and padding

In the previous section, each filter strode across the image – one column and one row at a time (after exhausting all possible columns by the end of the image). This also resulted in the output size being 1 pixel less than the input image size – both in terms of height and width. This results in a partial loss of information and can affect the possibility of us adding the output of the convolution operation to the original image (this is known as residual addition and will be discussed in detail in the next chapter).

In this section, we will learn about how strides and padding influence the output shape of convolutions.

Strides

Let's understand the impact of stride by leveraging the same example that we saw in the *Filter* section. Furthermore, we'll stride Matrix B with a stride of 2 over Matrix A. The output of convolution with a stride of 2 is as follows:

1. {1,2,5,6} of the bigger matrix is multiplied by {1,2,3,4} of the smaller matrix:

   *1*1 + 2*2 + 5*3 + 6*4 = 44*

1. {3,4,7,8} of the bigger matrix is multiplied by {1,2,3,4} of the smaller matrix:

   *3*1 + 4*2 + 7*3 + 8*4 = 64*

7. {9,10,13,14} of the bigger matrix is multiplied by {1,2,3,4} of the smaller matrix:

   *9*1 + 10*2 + 13*3 + 14*4 = 124*

8. {11,12,15,16} of the bigger matrix is multiplied by {1,2,3,4} of the smaller matrix:

   *11*1 + 12*2 + 15*3 + 16*4 = 144*

The result of performing the preceding operations is as follows:

44	64
124	144

Note that the preceding output has a lower dimension compared to the scenario where the stride was 1 (where the output shape was 3 x 3) since we now have a stride of 2.

Padding

In the preceding case, we could not multiply the leftmost elements of the filter by the rightmost elements of the image. If we were to perform such matrix multiplication, we would pad the image with zeros. This would ensure that we can perform element to element multiplication of all the elements within an image with a filter.

Let's understand padding by using the same example we used in the *Convolution* section.

Once we add padding on top of Matrix A, the revised version of Matrix A will look as follows:

0	0	0	0	0	0
0	1	2	3	4	0
0	5	6	7	8	0
0	9	10	11	12	0
0	13	14	15	16	0
0	0	0	0	0	0

From the preceding matrix, we can see that we have padded Matrix A with zeros and that the convolution with Matrix B will not result in the output dimension being smaller than the input's dimension. This aspect comes in handy when we are working on residual network where we must add the output of the convolution to the original image.

Once we've done this, we can perform activation on top of the convolution operation's output. We could use any of the activation functions we saw in `Chapter 3`, *Building a Deep Neural Network with PyTorch*, for this.

Pooling

Pooling aggregates information in a small patch. Imagine a scenario where the output of convolution activation is as follows:

1	2
3	4

The max pooling for this patch is 4. Here, we have considered the elements in this pool of elements and have taken the maximum value across all the elements present.

Similarly, let's understand the max pooling for a bigger matrix:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

In the preceding case, if the pooling stride has a length of 2, the max pooling operation is calculated as follows, where we divide the input image by a stride of 2 (that is, we have divided the image into 2 x 2 divisions):

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

For the four sub-portions of the matrix, the maximum values in the pool of elements are as follows:

6	8
14	16

In practice, it is not necessary to always have a stride of 2; this has just been used for illustration purposes here.

Other variants of pooling are sum and average pooling. However, in practice, max pooling is used more often.

Note that by the end of performing the convolution and pooling operations, the size of the original matrix is reduced from 4 x 4 to 2 x 2. In a realistic scenario, if the original image is of shape 200 x 200 and the filter is of shape 3 x 3, the output of the convolution operation would be 198 x 198. After that, the output of the pooling operation with a stride of 2 is 99 X 99.

Putting them all together

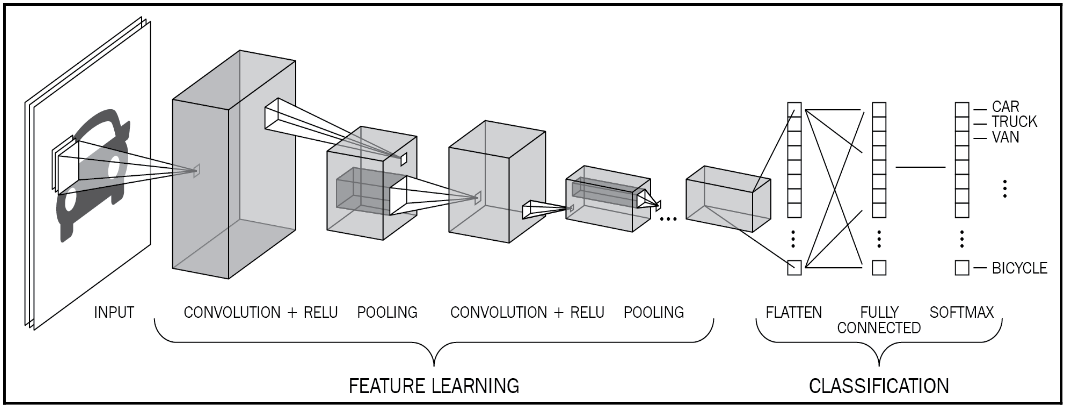
So far, we have learned about convolution, filters, and pooling, and their impact in reducing the dimension of an image. Now, we will learn about another critical component of a CNN – the flatten layer (fully connected layer) – before putting the three pieces we have learned about together.

To understand the flattening process, we'll take the output of the pooling layer in the previous section and flatten the output. The output of flattening the pooling layer is as follows:

{6, 8, 14, 16}

By doing this, we'll see that the flatten layer can be treated equivalent to the input layer (where we flattened the input image into a 784-dimensional input in `Chapter 3`, *Building a Deep Neural Network with PyTorch*). Once the flatten layer's (fully connected layer) values have been obtained, we can pass it through the hidden layer and then obtain the output for predicting the class of an image.

The overall flow of a CNN is as follows:



In the preceding image, we can see the overall flow of a CNN model, where we are passing an image through convolution via multiple filters and then pooling (and in the preceding case, repeating the convolution and pooling process twice), before flattening the output of the final pooling layer. This forms the **feature learning** part of the preceding image.

The operations of convolution and pooling constitute the feature learning section as filters help in extracting relevant features from images and pooling helps in aggregating information and thereby reducing the number of nodes at the flatten layer. (If we directly flatten the input image (which is 300 x 300 pixels in size, for example), we are dealing with 90K input values. If we have 90K input pixel values and 100K nodes in a hidden layer, we are looking at ~9 billion parameters, which is huge in terms of computation.)

Convolution and pooling help in fetching a flattened layer that has a much smaller representation than the original image.

Finally, the last part of the classification is similar to the way we classified images in `Chapter 3`, *Building a Deep Neural Network in PyTorch*, where we had a hidden layer and then obtained the output layer.

How convolution and pooling help in image translation

When we perform pooling, we can consider the output of the operation as an abstraction of a region (a small patch). This phenomenon comes in handy, especially when images are being translated.

Think of a scenario where an image is translated by 1 pixel to the left. Once we perform convolution, activation, and pooling on top of it, we'll have reduced the dimension of the image (due to pooling), which means that a fewer number of pixels store the majority of the information from the original image. Moreover, given that pooling stores information of a region (patch), the information within a pixel of the pooled image would not vary, even if the original image is translated by 1 unit. This is because the maximum value of that region is likely to get captured in the pooled image.

Convolution and pooling cam also help us with the **receptive field**. To understand the receptive field, let's imagine a scenario where we perform a convolution pooling operation twice on an image that is 100 x 100 in shape. The output at the end of the two convolution pooling operations is of the shape 25 x 25 (if the convolution operation was done with padding). Each cell in the 25 x 25 output now corresponds to a larger 4 x 4 portion of the original image. Thus, because of the convolution and pooling operations, each cell in the resulting image corresponds to a patch of the original image.

Now that we have learned about the core components of a CNN, let's apply them all to a toy example to understand how they work together.

Implementing a CNN

A CNN is one of the foundational blocks of computer vision techniques, and it is important for you to have a solid understanding of how they work. While we already know that a CNN constitutes convolution, pooling, flattening, and then the final classification layer, in this section, we will understand the various operations that occur during the forward pass of a CNN through code.

To gain a solid understanding of this, first, we will build a CNN architecture on a toy example using PyTorch and then match the output by building the feed-forward propagation from scratch in Python.

Building a CNN-based architecture using PyTorch

The CNN architecture will differ from the neural network architecture that we built in the previous chapter in that a CNN constitutes the following in addition to what a typical vanilla deep neural network would have:

- Convolution operation
- Pooling operation
- Flattening layer

In the following code, we will build a CNN model on a toy dataset, as follows:

The code for this section is available as `CNN_working_details.ipynb` in the `Chapter04` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. First, we need to import the relevant libraries:

```
import torch
from torch import nn
from torch.utils.data import TensorDataset, Dataset,
DataLoader
from torch.optim import SGD, Adam
device = 'cuda' if torch.cuda.is_available() else 'cpu'
from torchvision import datasets
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

2. Then, we need to create the dataset using the following steps:

```
X_train = torch.tensor([[[[1,2,3,4],[2,3,4,5], \
                          [5,6,7,8],[1,3,4,5]]], \
                [[[-1,2,3,-4],[2,-3,4,5], \
             [-5,6,-7,8],[-1,-3,-4,-5]]]]).to(device).float()
X_train /= 8
y_train = torch.tensor([0,1]).to(device).float()
```

Note that PyTorch expects inputs to be of the shape **N x C x H x W**, where *N* is the number of (batch size) of images, *C* is the number of channels, *H* is the height, and *W* is the width of the image.

Here, we are scaling the input dataset so that it has a range between -1 to +1 by dividing the input data by the maximum input value; that is, 8.

The shape of the input dataset is (2,1,4,4) since there are two data points, where each is 4 x 4 in shape and has 1 channel.

3. Define the model architecture:

```
def get_model():
    model = nn.Sequential(
                nn.Conv2d(1, 1, kernel_size=3),
                nn.MaxPool2d(2),
                nn.ReLU(),
                nn.Flatten(),
                nn.Linear(1, 1),
                nn.Sigmoid(),
            ).to(device)
    loss_fn = nn.BCELoss()
    optimizer = Adam(model.parameters(), lr=1e-3)
    return model, loss_fn, optimizer
```

Note that in the preceding model, we are specifying that there is 1 channel in the input and that we are extracting 1 channel from the output post convolution (that is, we have 1 filter with a size of 3 x 3) using the `nn.Conv2d` method. After this, we perform max pooling using `nn.MaxPool2d` and ReLU activation (using `nn.Relu()`) prior to flattening and connecting to the final layer, which has one output per data point.

Furthermore, note that the loss function is binary cross-entropy loss (`nn.BCELoss()`) since the output is from a binary class. We are also specifying that the optimization will be done using the Adam optimizer with a learning rate of 0.001.

4. Summarize the architecture of the model using the `summary` method that's available in the `torch_summary` package post fetching our `model`, loss function (`loss_fn`), and `optimizer` by calling the `get_model` function:

```
!pip install torch_summary
from torchsummary import summary
model, loss_fn, optimizer = get_model()
summary(model, X_train);
```

The preceding code results in the following output:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1             [-1, 1, 2, 2]              10
         MaxPool2d-2             [-1, 1, 1, 1]               0
              ReLU-3             [-1, 1, 1, 1]               0
           Flatten-4                   [-1, 1]               0
            Linear-5                   [-1, 1]               2
           Sigmoid-6                   [-1, 1]               0
================================================================
Total params: 12
Trainable params: 12
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.00
Estimated Total Size (MB): 0.00
----------------------------------------------------------------
```

Let's understand the reason why each layer contains so many parameters. The arguments of the `Conv2d` class are as follows:

```
help(nn.Conv2d)

 |  Args:
 |      in_channels (int): Number of channels in the input image
 |      out_channels (int): Number of channels produced by the convolution
 |      kernel_size (int or tuple): Size of the convolving kernel
 |      stride (int or tuple, optional): Stride of the convolution. Default: 1
 |      padding (int or tuple, optional): Zero-padding added to both sides of the input. Default: 0
 |      padding_mode (string, optional). Accepted values `zeros` and `circular` Default: `zeros`
 |      dilation (int or tuple, optional): Spacing between kernel elements. Default: 1
 |      groups (int, optional): Number of blocked connections from input channels to output channels. Default: 1
 |      bias (bool, optional): If ``True``, adds a learnable bias to the output. Default: ``True``
```

In the preceding case, we are specifying that the size of the convolving kernel (`kernel_size`) is 3 and that the number of `out_channels` is 1 (essentially, the number of filters is 1), where the number of initial (input) channels is 1. Thus, for each input image, we are convolving a filter of shape 3 x 3 on a shape of 1 x 4 x 4, which results in an output of the shape 1 x 2 x 2. There are 10 parameters since we are learning the nine weight parameters (3 x 3) and the one bias of the convolving kernel. For the MaxPool2d, ReLU, and Flatten layers, there are no parameters as these are operations that are performed on top of the output of the convolution layer; no weights or biases are involved.

- The linear layer has two parameters – one weight and one bias – which means there's a total of 12 parameters (10 from the convolution operation and two from the linear layer).

5. Train the model using the same model training code we used in `Chapter 3`, *Building a Deep Neural Network with PyTorch*, where we defined the function that will train on batches of data (`train_batch`). Then, fetch the DataLoader and train it on batches of data over 2,000 epochs (we're only using 2,000 because this is a small toy dataset), as follows:

    - Define the function that will train on batches of data (`train_batch`):

    ```
    def train_batch(x, y, model, opt, loss_fn):
        model.train()
        prediction = model(x)
        batch_loss = loss_fn(prediction.squeeze(0), y)
        batch_loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        return batch_loss.item()
    ```

    - Define the training DataLoader by specifying the dataset using the `TensorDataset` method and then loading it using `DataLoader`:

    ```
    trn_dl = DataLoader(TensorDataset(X_train, y_train))
    ```

    Note that, given we are not modifying the input data by a lot, we won't be building a class separately, instead leveraging the `TensorDataset` method directly, which provides an object that corresponds to the input data.

    - Train the model over 2,000 epochs:

    ```
    for epoch in range(2000):
        for ix, batch in enumerate(iter(trn_dl)):
            x, y = batch
            batch_loss = train_batch(x, y, model, optimizer, \
                                     loss_fn)
    ```

    With the preceding code, we have trained the CNN model on our toy dataset.

6. Perform a forward pass on top of the first data point:

    ```
    model(X_train[:1])
    ```

    The output of the preceding code is `0.1625`.

Note that you might have a different output value owing to a different random weight initialization when you execute the preceding code. However, you should be able to match the output against what you get in the next section.

In the next section, we will learn about how forward propagation in CNNs works so that we can obtain a value of 0.1625 on the first data point.

Forward propagating the output in Python

Before we proceed, note that this section is only here to help you clearly understand how CNNs work. We don't need to perform the following steps in a real-world scenario:

1. Extract the weights and biases of the convolution and linear layers of the architecture that's been defined, as follows:

   - Extract the various layers of the model:

   ```
   list(model.children())
   ```

   This results in the following output:

   ```
   [Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1)),
    MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False),
    ReLU(),
    Flatten(),
    Linear(in_features=1, out_features=1, bias=True),
    Sigmoid()]
   ```

   - Extract the layers among all the layers of the model that have the `weight` attribute associated with them:

   ```
   (cnn_w, cnn_b), (lin_w, lin_b) = [(layer.weight.data, \
                                layer.bias.data) for layer in \
                                list(model.children()) \
                                      if hasattr(layer,'weight')]
   ```

   In the preceding code, `hasattr(layer,'weight')` returns a boolean, regardless of whether the layer contains the `weight` attribute.

Note that the convolution (`Conv2d`) layer and the `Linear` layer at the end are the only layers that contain parameters, which is why we saved them as `cnn_w` and `cnn_b` for the `Conv2d` layer and `lin_w` and `lin_b` for the `Linear` layer, respectively.

The shape of `cnn_w` is 1 x 1 x 3 x 3 since we have initialized one filter, which has one channel and a dimension of 3 x 3. `cnn_b` has a shape of 1 as it corresponds to one filter.

2. To perform the `cnn_w` convolution operation over the input value, we must initialize a matrix of zeros for sumproduct (`sumprod`) where the height is *input height - filter height + 1* and the width is *width - filter width + 1*:

```
h_im, w_im = X_train.shape[2:]
h_conv, w_conv = cnn_w.shape[2:]
sumprod = torch.zeros((h_im - h_conv + 1, w_im - w_conv + 1))
```

3. Now, let's fill `sumprod` by convoluting the filter (`cnn_w`) across the first input and summing up the filter bias term (`cnn_b`) after reshaping the filter shape from a 1 x 1 x 3 x 3 shape to a 3 x 3 shape:

```
for i in range(h_im - h_conv + 1):
    for j in range(w_im - w_conv + 1):
        img_subset = X_train[0, 0, i:(i+3), j:(j+3)]
        model_filter = cnn_w.reshape(3,3)
        val = torch.sum(img_subset*model_filter) + cnn_b
        sumprod[i,j] = val
```

In the preceding code, `img_subset` stores the portion of the input that we would be convolving with the filter and hence we stride through it across the possible columns and then rows.

Furthermore, given that the input is 4 x 4 in shape and the filter is 3 x 3 in shape, the output is 2 x 2 in shape.

At this stage, the output of `sumprod` is as follows:

```
tensor([[-0.0143, -0.0636],
        [-0.0834, -0.0500]])
```

4. Perform the ReLU operation on top of the output and then fetch the maximum value of the pool (`MaxPooling`), as follows:

   - ReLU is performed on top of `sumprod` in Python as follows:

```
sumprod.clamp_min_(0)
```

   Note that we are clamping the output to a minimum of 0 in the preceding code (which is what ReLU activation does):

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

   - The output of the pooling layer can be calculated like so:

```
pooling_layer_output = torch.max(sumprod)
```

   The preceding code results in the following output:

```
tensor(0.)
```

5. Pass the preceding output through linear activation:

```
intermediate_output_value = pooling_layer_output*lin_w+lin_b
```

   The output of this operation is as follows:

```
tensor([[-1.6398]])
```

6. Pass the output through the `sigmoid` operation:

```
from torch.nn import functional as F # torch library
# for numpy like functions
print(F.sigmoid(intermediate_output_value))
```

   The preceding code gives us the following output:

```
tensor([[0.1625]])
```

Note that we perform `sigmoid` and not `softmax` since the loss function is binary cross-entropy and not categorical cross-entropy like it was in the Fashion-MNIST dataset.

The preceding code gives us the same output we obtained using PyTorch's feedforward method, thus strengthening our understanding of how CNNs work.

Now that we have learned about how CNNs work, in the next section, we'll apply this to the Fashion-MNIST dataset and see how it fares on translated images.

Classifying images using deep CNNs

So far, we have seen that the traditional neural network predicts incorrectly for translated images. This needs to be addressed because in real-world scenarios, various augmentations will need to be applied, such as translatation and rotation, that were not seen during the training phase. In this section, we will understand how CNNs address the problem of incorrect predictions when image translation happens on images in the Fashion-MNIST dataset.

The pre-processing portion of the Fashion-MNIST dataset remains the same as in the previous chapter, except that when we reshape (`.view`) the input data, instead of flattening the input to 28 x 28 = 784 dimensions, we reshape the input to a shape of (1,28,28) for each image (remember, channels are to be specified first, followed by their height and width, in PyTorch):

> The code for this section is available as `CNN_on_FashionMNIST.ipynb` in the `Chapter04` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` . Note that the entirety of the code is available in GitHub and that only the additional code corresponding to defining the model architecture is provided here for brevity. **We strongly encourage you to refer to the notebooks in this book's GitHub repository while executing the code.**

1. Import the necessary packages:

```
from torchvision import datasets
from torch.utils.data import Dataset, DataLoader
import torch
import torch.nn as nn
device = "cuda" if torch.cuda.is_available() else "cpu"
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

data_folder = '~/data/FMNIST' # This can be any directory you
# want to download FMNIST to
```

```
fmnist = datasets.FashionMNIST(data_folder, download=True, \
                                        train=True)
tr_images = fmnist.data
tr_targets = fmnist.targets
```

2. The Fashion-MNIST dataset class is defined as follows. Remember, the `Dataset` object will **always** need the `__init__`, `__getitem__`, and `__len__` methods we've defined:

```
class FMNISTDataset(Dataset):
    def __init__(self, x, y):
        x = x.float()/255
        x = x.view(-1,1,28,28)
        self.x, self.y = x, y
    def __getitem__(self, ix):
        x, y = self.x[ix], self.y[ix]
        return x.to(device), y.to(device)
    def __len__(self):
        return len(self.x)
```

The preceding line of code in bold is where we are reshaping each input image (differently to what we did in the previous chapter) since we are providing data to a CNN that expects each input to have a shape of batch size x channels x height x width.

3. The CNN model architecture is defined as follows:

```
from torch.optim import SGD, Adam
def get_model():
    model = nn.Sequential(
                nn.Conv2d(1, 64, kernel_size=3),
                nn.MaxPool2d(2),
                nn.ReLU(),
                nn.Conv2d(64, 128, kernel_size=3),
                nn.MaxPool2d(2),
                nn.ReLU(),
                nn.Flatten(),
                nn.Linear(3200, 256),
                nn.ReLU(),
                nn.Linear(256, 10)
            ).to(device)

    loss_fn = nn.CrossEntropyLoss()
    optimizer = Adam(model.parameters(), lr=1e-3)
    return model, loss_fn, optimizer
```

- A summary of the model can be created using the following code:

```
!pip install torch_summary
from torchsummary import summary
model, loss_fn, optimizer = get_model()
summary(model, torch.zeros(1,1,28,28));
```

This results in the following output:

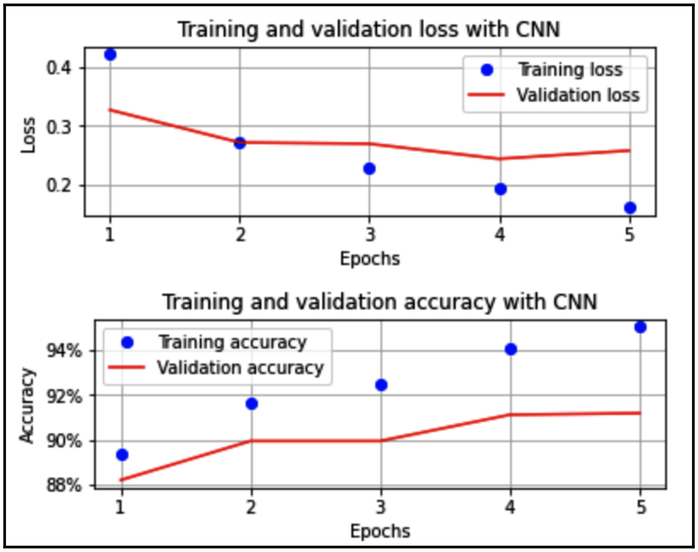
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 26, 26]             640
         MaxPool2d-2           [-1, 64, 13, 13]               0
              ReLU-3           [-1, 64, 13, 13]               0
            Conv2d-4          [-1, 128, 11, 11]          73,856
         MaxPool2d-5            [-1, 128, 5, 5]               0
              ReLU-6            [-1, 128, 5, 5]               0
           Flatten-7                 [-1, 3200]               0
            Linear-8                  [-1, 256]         819,456
              ReLU-9                  [-1, 256]               0
           Linear-10                   [-1, 10]           2,570
================================================================
Total params: 896,522
Trainable params: 896,522
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.69
Params size (MB): 3.42
Estimated Total Size (MB): 4.11
----------------------------------------------------------------
```

To solidify our understanding of CNNs, let's understand the reason why the number of parameters have been set the way they have in the preceding output:

- **Layer 1**: Given that there are 64 filters with a kernel size of 3, we have 64 x 3 x 3 weights and 64 x 1 biases, resulting in a total of 640 parameters.
- **Layer 4**: Given that there are 128 filters with a kernel size of 3, we have 128 x 64 x3 x 3 weights and 128 x 1 biases, resulting in a total of 73,856 parameters.
- **Layer 8**: Given that a layer with 3,200 nodes is getting connected to another layer with 256 nodes, we have a total of 3,200 x 256 weights + 256 biases, resulting in a total of 819,456 parameters.
- **Layer 10**: Given that a layer with 256 nodes is getting connected to a layer with 10 nodes, we have a total of 256 x 10 weights and 10 biases, resulting in a total of 2,570 parameters.

Now, we train the model, just like we trained it in the previous chapter. The full code is available in this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

Once the model has been trained, you'll notice that the variation of accuracy and loss over the training and test datasets is as follows:



Note that in the preceding scenario, the accuracy of the validation dataset is ~92% within the first five epochs, which is already better than the accuracy we saw across various techniques in the previous chapter, even without additional regularization.

Now, let's translate the image and predict the class of translated images:

1. Translate the image between -5 pixels to +5 pixels and predict its class:

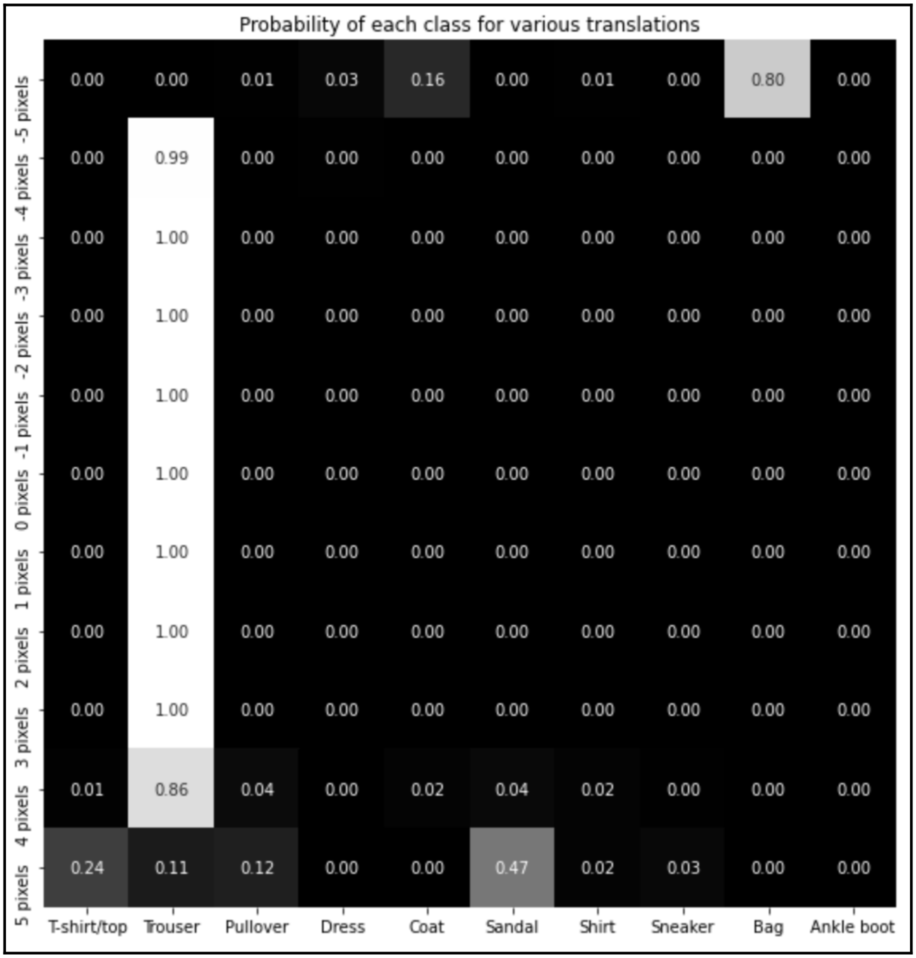
```
preds = []
ix = 24300
for px in range(-5,6):
    img = tr_images[ix]/255.
    img = img.view(28, 28)
    img2 = np.roll(img, px, axis=1)
    plt.imshow(img2)
    plt.show()
    img3 = torch.Tensor(img2).view(-1,1,28,28).to(device)
    np_output = model(img3).cpu().detach().numpy()
    preds.append(np.exp(np_output)/np.sum(np.exp(np_output)))
```

   In the preceding code, we reshaped the image (`img3`) so that it has a shape of (-1,1,28,28) so that we can pass the image to a CNN model.

2. Plot the probability of the classes across various translations:

```
import seaborn as sns
fig, ax = plt.subplots(1,1, figsize=(12,10))
plt.title('Probability of each class for \
various translations')
sns.heatmap(np.array(preds).reshape(11,10), annot=True, \
            ax=ax, fmt='.2f', xticklabels=fmnist.classes, \
            yticklabels=[str(i)+str(' pixels') \
                         for i in range(-5,6)], cmap='gray')
```

The preceding code results in the following output:



Note that in this scenario, even when the image was translated by 4 pixels, the prediction was correct, while in the scenario where we did not use a CNN, the prediction was incorrect when the image was translated by 4 pixels. Furthermore, when the image was translated by 5 pixels, the probability of "Trouser" dropped considerably.

As we can see, while CNNs help in addressing the challenge of image translation, they don't solve the problem at hand completely. We will learn how to address such a scenario by leveraging data augmentation alongside CNNs in the next section.

Implementing data augmentation

In the previous scenario, we learned about how CNNs help in predicting the class of an image when it is translated. While this worked well for translations of up to 5 pixels, anything beyond that is likely to have a very low probability for the right class. In this section, we'll learn how to ensure that we predict the right class, even if the image is translated by a considerable amount.

To address this challenge, we'll train the neural network by translating the input images by 10 pixels randomly (both toward the left and the right) and passing them to the network. This way, the same image will be processed as a different image in different passes since it will have had a different amount of translation in each pass.

Before we leverage augmentations to improve the accuracy of our model when images are translated, let's learn about the various augmentations that can be done on top of an image.

Image augmentations

So far, we have learned about the issues image translation can have on a model's prediction accuracy. However, in the real world, we might encounter various scenarios, such as the following:

- Images are rotated slightly
- Images are zoomed in/out (scaled)
- Some amount of noise is present in the image
- Images have low brightness
- Images have been flipped
- Images have been sheared (one side of the image is more twisted)

A neural network that does not take the preceding scenarios into consideration won't provide accurate results, just like in the previous section, where we had a neural network that had not been explicitly trained on images that had been heavily translated.

> Image augmentations come in handy in scenarios where we create more images from a given image. Each of the created images can vary in terms of rotation, translation, scale, noise, and brightness. Furthermore, the extent of the variation in each of these parameters can also vary (for example, translation of a certain image in a given iteration can be +10 pixels, while in a different iteration, it can be -5 pixels).

The `augmenters` class in the `imgaug` package has useful utilities for performing these augmentations. Let's take a look at the various utilities present in the `augmenters` class for generating augmented images from a given image. Some of the most prominent augmentation techniques are as follows:

- Affine transformations
- Change brightness
- Add noise

> Note that PyTorch has a handy image augmentation pipeline in the form of `torchvision.transforms`. However, we still opted to introduce a different library primarily because of the larger variety of options `imgaug` contains, as well as due to the ease of explaining augmentations to a new user. You are encouraged to research the torchvision transforms as an exercise and recreate all the functions that are presented to strengthen your understanding.

Affine transformations

Affine transformations involve translating, rotating, scaling, and shearing an image. They can be performed in code using the `Affine` method that's present in the `augmenters` class. Let's take a look at the parameters present in the `Affine` method by looking at the following screenshot. Here, we have defined all the parameters of the `Affine` method:

```
iaa.Affine(scale=1.0, translate_percent=None, translate_px=None, rotate=0.0, shear=0.0,
order=1, cval=0, mode='constant', fit_output=False, backend='auto', name=None,
deterministic=False, random_state=None)
```

Some of the important parameters in the `Affine` method are as follows:

- `scale` specifies the amount of zoom that is to be done for the image
- `translate_percent` specifies the amount of translation as a percentage of the image's height and width
- `translate_px` specifies the amount of translation as an absolute number of pixels
- `rotate` specifies the amount of rotation that is to be done on the image
- `shear` specifies the amount of rotation that is to be done on part of the image

Before we consider ay other parameters, let's understand where scaling, translation, and rotation come in handy.

The code for this section is available as `Image_augmentation.ipynb` in the `Chapter04` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

Fetch a random image from the training dataset for `fashionMNIST`:

1. Download images from the Fashion-MNIST dataset:

```
from torchvision import datasets
import torch
data_folder = '/content/' # This can be any directory
# you download FMNIST to
fmnist = datasets.FashionMNIST(data_folder, download=True, \
                               train=True)
```

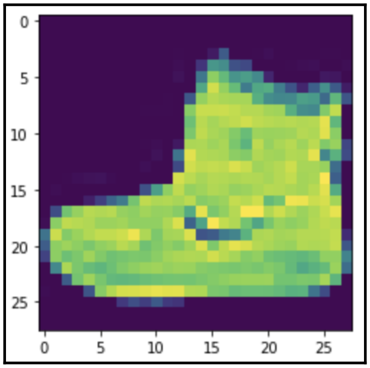
2. Fetch an image from the downloaded dataset:

```
tr_images = fmnist.data
tr_targets = fmnist.targets
```

3. Let's plot the first image:

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.imshow(tr_images[0])
```

The output of the preceding code is as follows:



Perform scaling on top of the image:

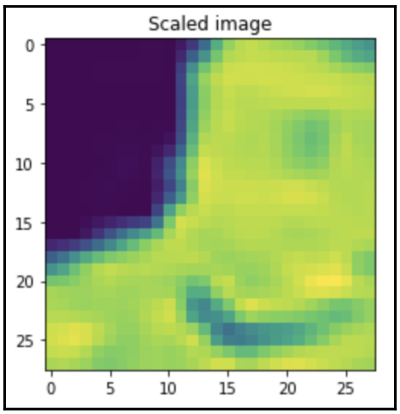
1. Define an object that performs scaling:

```
from imgaug import augmenters as iaa
aug = iaa.Affine(scale=2)
```

2. Specify that we want to augment the image using the `augment_image` method, which is available in the `aug` object, and plot it:

```
plt.imshow(aug.augment_image(tr_images[0]))
plt.title('Scaled image')
```

The output of the preceding code is as follows:

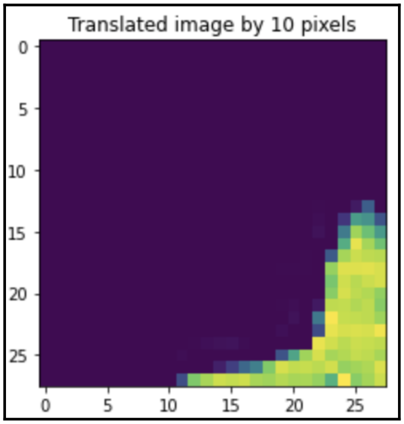


In the preceding output, the image has been zoomed into considerably. This has resulted in some pixels being cut from the original image since the output shape of the image hasn't changed.

Now, let's take a look at a scenario where an image has been translated by a certain number of pixels using the `translate_px` parameter:

```
aug = iaa.Affine(translate_px=10)
plt.imshow(aug.augment_image(tr_images[0]))
plt.title('Translated image by 10 pixels')
```

The output of the preceding code is as follows:



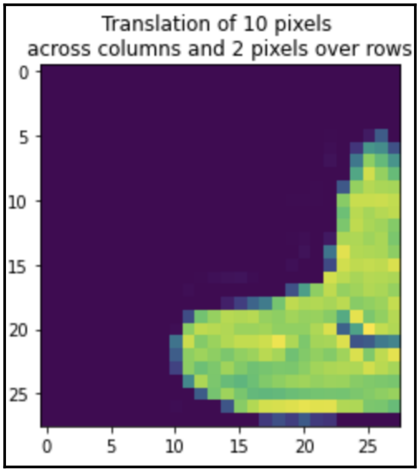
In the preceding output, the translation by 10 pixels has happened across both the x and y axes.

If we want to perform translation more in one axis and less in the other axis, we must specify the amount of translation we want in each axis:

```
aug = iaa.Affine(translate_px={'x':10,'y':2})
plt.imshow(aug.augment_image(tr_images[0]))
plt.title('Translation of 10 pixels \nacross columns \
and 2 pixels over rows')
```

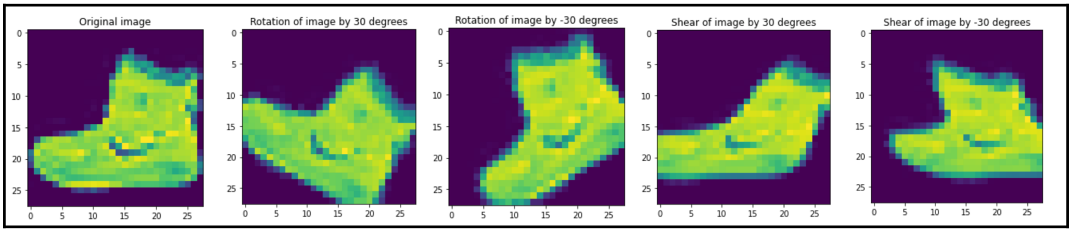
Here, we have provided a dictionary that states the amount of translation in the x and y axes in the `translate_px` parameter.

The output of the preceding code is as follows:



The preceding output shows that more translation happened across columns compared to rows. This has also resulted in a certain portion of the image being cropped.

Now, let's consider the impact rotation and shearing have on image augmentation:



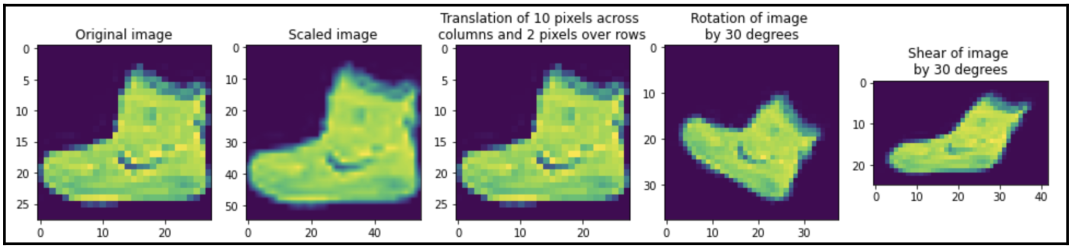
In the majority of the preceding outputs, we can see that certain pixels were cropped out of the image post-transformation. Now, let's take a look at how the rest of the parameters in the `Affine` method help us not lose information due to cropping post-augmentation.

`fit_output` is a parameter that can help with the preceding scenario. By default, it is set to `False`. However, let's see how the preceding outputs vary when we specify `fit_output` as `True` when we scale, translate, rotate, and shear the image:

```
plt.figure(figsize=(20,20))
plt.subplot(161)
plt.imshow(tr_images[0])
```

```
plt.title('Original image')
plt.subplot(162)
aug = iaa.Affine(scale=2, fit_output=True)
plt.imshow(aug.augment_image(tr_images[0]))
plt.title('Scaled image')
plt.subplot(163)
aug = iaa.Affine(translate_px={'x':10,'y':2}, fit_output=True)
plt.imshow(aug.augment_image(tr_images[0]))
plt.title('Translation of 10 pixels across \ncolumns and \
2 pixels over rows')
plt.subplot(164)
aug = iaa.Affine(rotate=30, fit_output=True)
plt.imshow(aug.augment_image(tr_images[0]))
plt.title('Rotation of image \nby 30 degrees')
plt.subplot(165)
aug = iaa.Affine(shear=30, fit_output=True)
plt.imshow(aug.augment_image(tr_images[0]))
plt.title('Shear of image \nby 30 degrees')
```

The output of the preceding code is as follows:



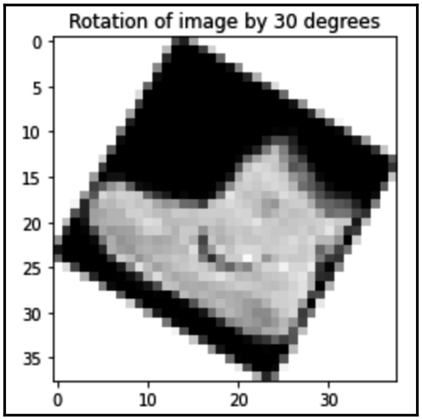
Here, we can see that the original image hasn't been cropped and that the size of the augmented image increased to account for the augmented image not being cropped (in the scaled image's output or when rotating the image by 30 degrees). Furthermore, we can also see that the activation of the `fit_output` parameter has negated the translation that we expected in the translation of a 10-pixel image (this is a known behavior, as explained in the documentation).

Note that when the size of the augmented image increases (for example, when the image is rotated), we need to figure out how the new pixels that are not part of the original image should be filled in.

The `cval` parameter solves this issue. It specifies the pixel value of the new pixels that are created when `fit_output` is `True`. In the preceding code, `cval` is filled with a default value of 0, which results in black pixels. Let's understand how changing the `cval` parameter to a value of 255 impacts the output when an image is rotated:

```
aug = iaa.Affine(rotate=30, fit_output=True, cval=255)
plt.imshow(aug.augment_image(tr_images[0]))
plt.title('Rotation of image by 30 degrees')
```

The output of the preceding code is as follows:



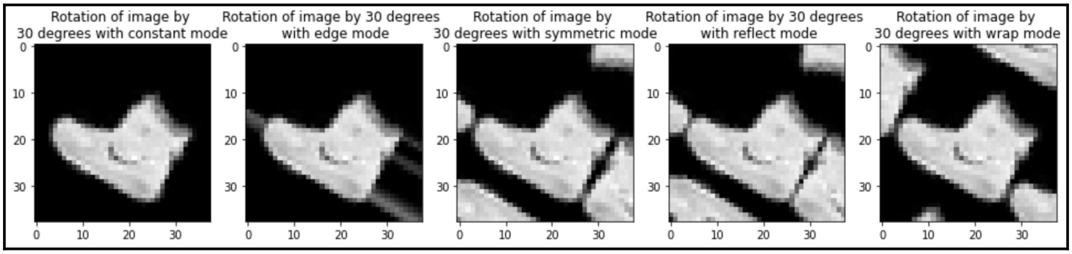
In the preceding image, the new pixels have been filled with a pixel value of 255, which corresponds to the color white.

Furthermore, there are different modes we can use to fill the values of newly created pixels. These values, which are for the `mode` parameter, are as follows:

- `constant`: Pads with a constant value.
- `edge`: Pads with the edge values of the array.
- `symmetric`: Pads with the reflection of the vector mirrored along the edge of the array.
- `reflect`: Pads with the reflection of the vector mirrored on the first and last values of the vector along each axis.
- `wrap`: Pads with the wrap of the vector along the axis.

The initial values are used to pad the end, while the end values are used to pad the beginning.

The outputs that we receive when `cval` is set to 0 and we vary the `mode` parameter are as follows:

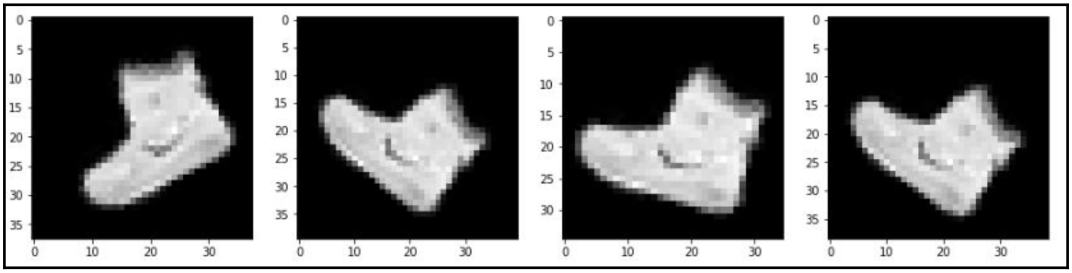


Here, we can see that for our current scenario based on the Fashion-MNIST dataset, it is more desirable to use the `constant` mode for data augmentation.

So far, we have specified that the translation needs to be a certain number of pixels. Similarly, we have specified that the rotation angle should be of a specific degree. However, in practice, it becomes difficult to specify the exact angle that an image needs to be rotated by. Thus, in the following code, we've provided a range that the image will be rotated by. This can be done like so:

```
plt.figure(figsize=(20,20))
plt.subplot(151)
aug = iaa.Affine(rotate=(-45,45), fit_output=True, cval=0, \
                 mode='constant')
plt.imshow(aug.augment_image(tr_images[0]), cmap='gray')
plt.subplot(152)
aug = iaa.Affine(rotate=(-45,45), fit_output=True, cval=0, \
                 mode='constant')
plt.imshow(aug.augment_image(tr_images[0]), cmap='gray')
plt.subplot(153)
aug = iaa.Affine(rotate=(-45,45), fit_output=True, cval=0, \
                 mode='constant')
plt.imshow(aug.augment_image(tr_images[0]), cmap='gray')
plt.subplot(154)
aug = iaa.Affine(rotate=(-45,45), fit_output=True, cval=0, \
                 mode='constant')
plt.imshow(aug.augment_image(tr_images[0]), cmap='gray')
```

The output of the preceding code is as follows:



In the preceding output, the same image was rotated differently in different iterations because we specified a range of possible rotation angles in terms of the upper and lower bounds of the rotation. Similarly, we can randomize augmentations when we are translating or sharing an image.

So far, we have looked at varying the image in different ways. However, the intensity/brightness of the image remains unchanged. Next, we'll learn how to augment the brightness of images.

Changing the brightness

Imagine a scenario where the difference between the background and the foreground is not as distinct as we have seen so far. This means the background does not have a pixel value of 0 and that the foreground does not have a pixel value of 255. Such a scenario can typically happen when the lighting conditions in the image are different.

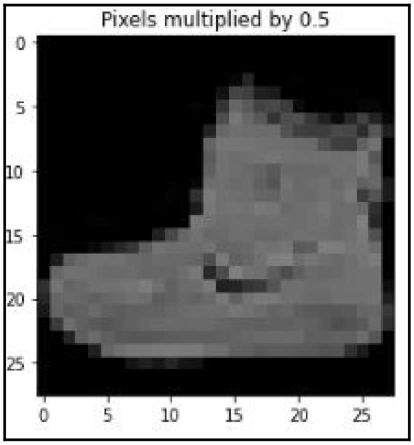
If the background has always had a pixel value of 0 and the foreground has always had a pixel value of 255 when the model has been trained but we are predicting an image that has a background pixel value of 20 and a foreground pixel value of 220, the prediction is likely to be incorrect.

`Multiply` and `Linearcontrast` are two different augmentation techniques that can be leveraged to resolve such scenarios.

The `Multiply` method multiplies each pixel value by the value that we specify. The output of multiplying each pixel value by 0.5 for the image we have been considering so far is as follows:

```
aug = iaa.Multiply(0.5)
plt.imshow(aug.augment_image(tr_images[0]), cmap='gray', \
           vmin = 0, vmax = 255)
plt.title('Pixels multiplied by 0.5')
```

The output of the preceding code is as follows:



`Linearcontrast` adjusts each pixel value based on the following formula:

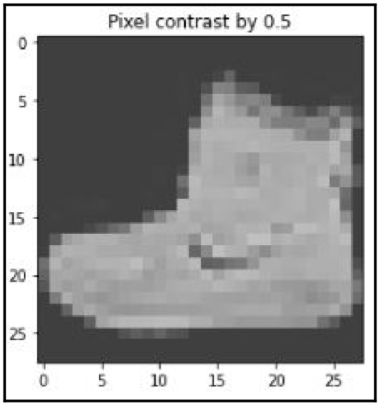
$$127 + \alpha \times (pixelvalue - 127)$$

In the preceding equation, when α is equal to 1, the pixel values remain unchanged. However, when α is less than 1, high pixel values are reduced and low pixel values are increased.

Let's take a look at the impact `Linearcontrast` has on the output of this image:

```
aug = iaa.LinearContrast(0.5)
plt.imshow(aug.augment_image(tr_images[0]), cmap='gray', \
           vmin = 0, vmax = 255)
plt.title('Pixel contrast by 0.5')
```

The output of the preceding code is as follows:

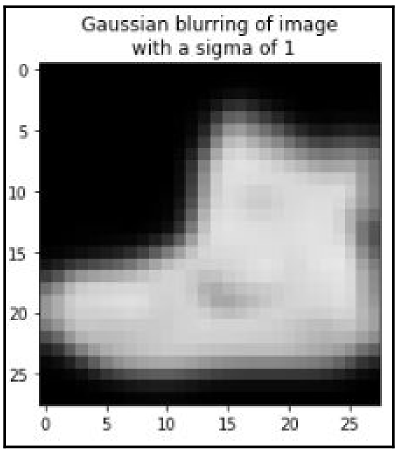


Here, we can see that the background became more bright, while the foreground pixels' intensity reduced.

Next, we'll blur the image to mimic a realistic scenario (where the image can be potentially blurred due to motion) using the `GaussianBlur` method:

```
aug = iaa.GaussianBlur(sigma=1)
plt.imshow(aug.augment_image(tr_images[0]), cmap='gray', \
           vmin = 0, vmax = 255)
plt.title('Gaussian blurring of image')
```

The output of the preceding code is as follows:



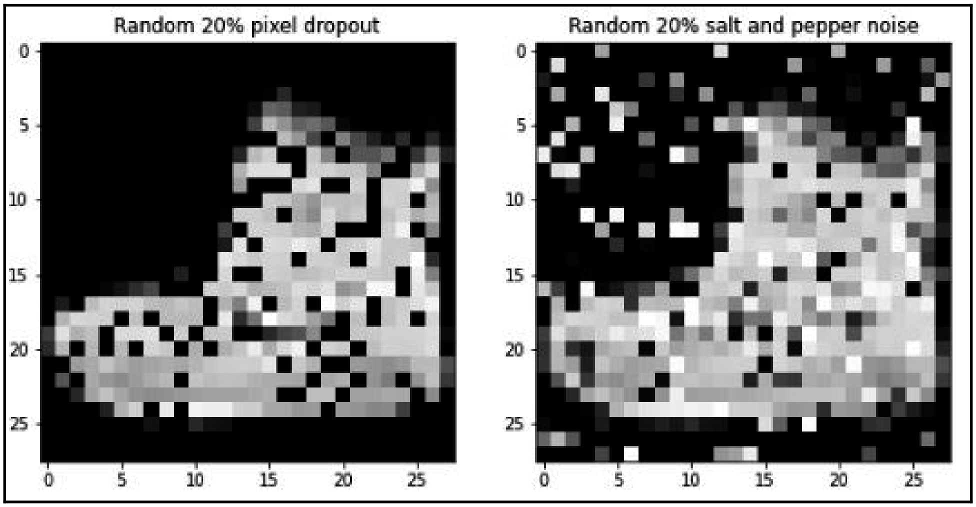
In the preceding image, we can see that the image was blurred considerably and that as the `sigma` value increases (where the default is 0 for no blurring), the image becomes even blurrier.

Adding noise

In a real-world scenario, we may encounter grainy images due to bad photography conditions. `Dropout` and `SaltAndPepper` are two prominent methods that can help in simulating grainy image conditions. Let's take a look at the output of augmenting an image with these two methods:

```
plt.figure(figsize=(10,10))
plt.subplot(121)
aug = iaa.Dropout(p=0.2)
plt.imshow(aug.augment_image(tr_images[0]), cmap='gray', \
           vmin = 0, vmax = 255)
plt.title('Random 20% pixel dropout')
plt.subplot(122)
aug = iaa.SaltAndPepper(0.2)
plt.imshow(aug.augment_image(tr_images[0]), cmap='gray', \
           vmin = 0, vmax = 255)
plt.title('Random 20% salt and pepper noise')
```

The output of the preceding code is as follows:



Here, we can see that while the `Dropout` method dropped a certain amount of pixels randomly (that is, it converted them so that they had a pixel value of 0), the `SaltAndPepper` method added some white-ish and black-ish pixels randomly to our image.

Performing a sequence of augmentations

So far, we have looked at various augmentations and have also performed. However, in a real-world scenario, we would have to account for as many augmentations as possible. In this section, we will learn about the sequential way of performing augmentations.

Using the `Sequential` method, we can construct the augmentation method using all the relevant augmentations that must be performed. For our example, we'll only consider `rotate` and `Dropout` for augmenting our image. The `Sequential` object looks as follows:

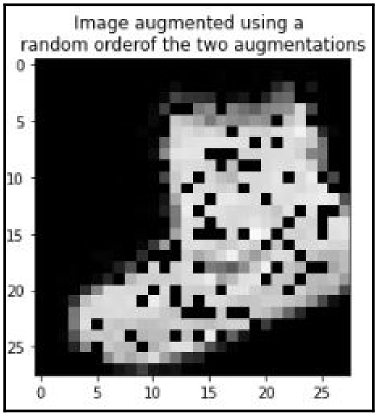
```
seq = iaa.Sequential([
      iaa.Dropout(p=0.2),
      iaa.Affine(rotate=(-30,30))], random_order= True)
```

In the preceding code, we are specifying that we are interested in the two augmentations and have also specified that we're going to be using the `random_order` parameter. The augmentation process is going to be performed randomly between the two.

Now, let's plot the image with these augmentations:

```
plt.imshow(seq.augment_image(tr_images[0]), cmap='gray', \
           vmin = 0, vmax = 255)
plt.title('Image augmented using a \nrandom order \
of the two augmentations')
```

The output of the preceding code is as follows:



From the preceding image, we can see that the two augmentations are performed on top of the original image (you can observe that the image has been rotated and that dropout has been applied).

Performing data augmentation on a batch of images and the need for collate_fn

We have already seen that it is preferable to perform different augmentations in different iterations on the same image.

If we have an augmentation pipeline defined in the `__init__` method, we would only need to perform augmentation once on the input set of images. This means we would not have different augmentations on different iterations.

Similarly, if the augmentation is in the `__getitem__` method – which is ideal since we want to perform a different set of augmentations on each image – the major bottleneck is that the augmentation is performed once for each image. It would be much faster if we were to perform augmentation on a batch of images instead of on one image at a time. Let's understand this in detail by looking at two scenarios where we will be working on 32 images:

- Augmenting 32 images, one at a time
- Augmenting 32 images as a batch in one go

To understand the time it takes to augment 32 images in both scenarios, let's leverage the first 32 images in the training images of the Fashion-MNIST dataset:

The following code is available as `Time_comparison_of_augmentation_scenario.ipynb` in the `Chapter04` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Fetch the first 32 images in the training dataset:

```
from torchvision import datasets
import torch
data_folder = '/content/'
fmnist = datasets.FashionMNIST(data_folder, download=True, \
                                                train=True)
tr_images = fmnist.data
tr_targets = fmnist.targets
```

2. Specify the augmentation to be performed on the images:

```
from imgaug import augmenters as iaa
aug = iaa.Sequential([
              iaa.Affine(translate_px={'x':(-10,10)},
                                          mode='constant'),
            ])
```

Next, we need to understand how to perform augmentation in the `Dataset` class. There are two possible ways of augmenting data:

- Augmenting a batch of images, one at a time
- Augmenting all the images in a batch in one go

Let's understand the time it takes to perform both the preceding scenarios:

- **Scenario 1:** Augmenting 32 images, one at a time:

  Calculate the time it takes to augment one image at a time using the `augment_image` method:

```
%%time
for i in range(32):
    aug.augment_image(tr_images[i])
```

  It takes ~180 milliseconds to augment for the 32 images.

- **Scenario 2:** Augmenting 32 images as a batch in one go:

  Calculate the time it takes to augment the batch of 32 images in one go using the `augment_images` method:

```
%%time
aug.augment_images(tr_images[:32])
```

  It takes ~8 milliseconds to perform augmentation on the batch of images.

> It is a best practice to augment on top of a batch of images than doing so one image at a time. In addition, the output of the `augment_images` method is a `numpy` array.

However, the traditional `Dataset` class that we have been working on provides the index of one image at a time in the `__getitem__` method. Hence, we need to learn how to use a new function – `collate_fn` – that enables us to perform manipulation on a batch of images.

3. Define the `Dataset` class, which takes the input images, their classes, and the augmentation object as initializers:

```
from torch.utils.data import Dataset, DataLoader
class FMNISTDataset(Dataset):
    def __init__(self, x, y, aug=None):
        self.x, self.y = x, y
        self.aug = aug
    def __getitem__(self, ix):
        x, y = self.x[ix], self.y[ix]
        return x, y
    def __len__(self): return len(self.x)
```

- Define `collate_fn`, which takes the batch of data as input:

```
    def collate_fn(self, batch):
```

- Separate the batch of images and their classes into two different variables:

```
        ims, classes = list(zip(*batch))
```

- Specify that augmentation must be done if the augmentation object is provided. This is useful is we need to perform augmentation on training data but not on validation data:

```
if self.aug: ims=self.aug.augment_images(images=ims)
```

In the preceding code, we leveraged the `augment_images` method so that we can work on a batch of images.

- Create tensors of images, along with scaling data, by dividing the image shape by 255:

```
ims = torch.tensor(ims)[:,None,:,:].to(device)/255.
classes = torch.tensor(classes).to(device)
return ims, classes
```

In general, we leverage the `collate_fn` method when we have to perform heavy computations. This is because performing such computations on a batch of images in one go is faster than doing it one image at a time.

4. From now on, to leverage the `collate_fn` method, we'll use a new argument while creating the DataLoader:
   - First, we create the `train` object:

```
train = FMNISTDataset(tr_images, tr_targets, aug=aug)
```

   - Next, we define the DataLoader, along with the object's `collate_fn` method, as follows:

```
trn_dl = DataLoader(train, batch_size=64, \
                    collate_fn=train.collate_fn,shuffle=True)
```

5. Finally, we train the model, as we have been training it so far. By leveraging the `collate_fn` method, we can train a model faster.

Now that we have a solid understanding of some of the prominent data augmentation techniques we can use, including pixel translation and `collate_fn`, which allows us to augment a batch of images, let's understand how they can be applied to a batch of data to address image translation issues.

Data augmentation for image translation

Now, we are in a position to train the model with augmented data. Let's create some augmented data and train the model:

The following code is available as `Data_augmentation_with_CNN.ipynb` in the `Chapter04` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Import the relevant packages and dataset:

```
from torchvision import datasets
import torch
from torch.utils.data import Dataset, DataLoader
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np

device = 'cuda' if torch.cuda.is_available() else 'cpu'
data_folder = '/content/' # This can be any directory
# you want to download FMNIST to
fmnist = datasets.FashionMNIST(data_folder, download=True, \
                                        train=True)
tr_images = fmnist.data
tr_targets = fmnist.targets
val_fmnist=datasets.FashionMNIST(data_folder, download=True, \
                                        train=False)
val_images = val_fmnist.data
val_targets = val_fmnist.targets
```

2. Create a class that can perform data augmentation on an image that's translated randomly anywhere between -10 to +10 pixels, either to the left or to the right:

   - Define the data augmentation pipeline:

```
from imgaug import augmenters as iaa
aug = iaa.Sequential([
              iaa.Affine(translate_px={'x':(-10,10)},
                                        mode='constant'),
            ])
```

- Define the `Dataset` class:

```
class FMNISTDataset(Dataset):
    def __init__(self, x, y, aug=None):
        self.x, self.y = x, y
        self.aug = aug
    def __getitem__(self, ix):
        x, y = self.x[ix], self.y[ix]
        return x, y
    def __len__(self): return len(self.x)
    def collate_fn(self, batch):
        'logic to modify a batch of images'
        ims, classes = list(zip(*batch))
        # transform a batch of images at once
        if self.aug: ims=self.aug.augment_images(images=ims)
        ims = torch.tensor(ims)[:,None,:,:].to(device)/255.
        classes = torch.tensor(classes).to(device)
        return ims, classes
```

In the preceding code, we've leveraged the `collate_fn` method to specify that we want to perform augmentations on a batch of images.

3. Define the model architecture, as we did in the previous section:

```
from torch.optim import SGD, Adam
def get_model():
    model = nn.Sequential(
                nn.Conv2d(1, 64, kernel_size=3),
                nn.MaxPool2d(2),
                nn.ReLU(),
                nn.Conv2d(64, 128, kernel_size=3),
                nn.MaxPool2d(2),
                nn.ReLU(),
                nn.Flatten(),
                nn.Linear(3200, 256),
                nn.ReLU(),
                nn.Linear(256, 10)
            ).to(device)

    loss_fn = nn.CrossEntropyLoss()
    optimizer = Adam(model.parameters(), lr=1e-3)
    return model, loss_fn, optimizer
```

4. Define the `train_batch` function in order to train on batches of data:

```
def train_batch(x, y, model, opt, loss_fn):
    model.train()
    prediction = model(x)
    batch_loss = loss_fn(prediction, y)
    batch_loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    return batch_loss.item()
```

5. Define the `get_data` function to fetch the training and validation DataLoaders:

```
def get_data():
    train = FMNISTDataset(tr_images, tr_targets, aug=aug)
    'notice the collate_fn argument'
    trn_dl = DataLoader(train, batch_size=64, \
                collate_fn=train.collate_fn, shuffle=True)
    val = FMNISTDataset(val_images, val_targets)
    val_dl = DataLoader(val, batch_size=len(val_images),
                collate_fn=val.collate_fn, shuffle=True)
    return trn_dl, val_dl
```

6. Specify the training and validation DataLoaders and fetch the model object, loss function, and optimizer:

```
trn_dl, val_dl = get_data()
model, loss_fn, optimizer = get_model()
```

7. Train the model over `5` epochs:

```
for epoch in range(5):
    for ix, batch in enumerate(iter(trn_dl)):
        x, y = batch
        batch_loss = train_batch(x, y, model, optimizer, \
                                    loss_fn)
```

8. Test the model on a translated image, as we did in the previous section:

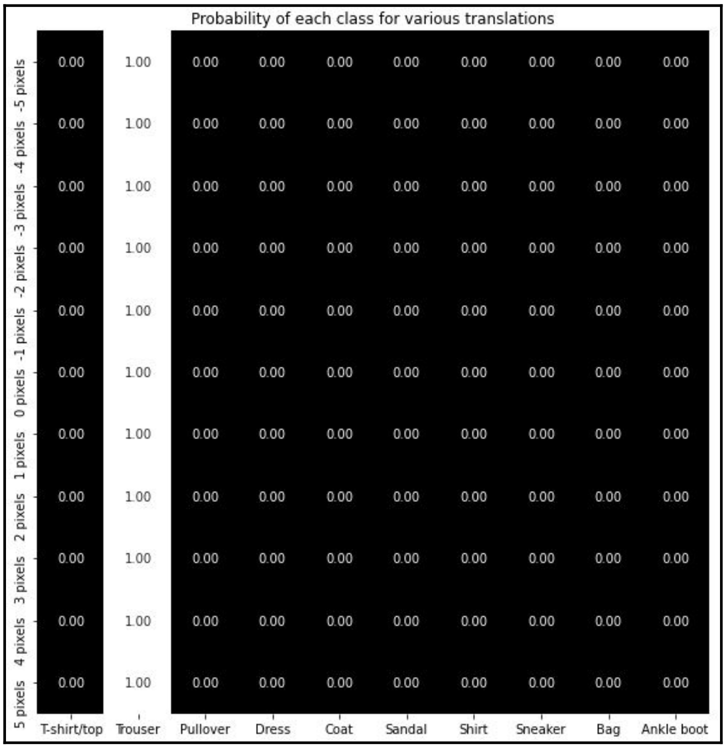
```
preds = []
ix = 24300
for px in range(-5,6):
    img = tr_images[ix]/255.
    img = img.view(28, 28)
    img2 = np.roll(img, px, axis=1)
    plt.imshow(img2)
```

```
    plt.show()
    img3 = torch.Tensor(img2).view(-1,1,28,28).to(device)
    np_output = model(img3).cpu().detach().numpy()
    preds.append(np.exp(np_output)/np.sum(np.exp(np_output)))
```

Now, let's plot the variation in the prediction class across different translations:

```
import seaborn as sns
fig, ax = plt.subplots(1,1, figsize=(12,10))
plt.title('Probability of each class \
for various translations')
sns.heatmap(np.array(preds).reshape(11,10), annot=True, \
            ax=ax, fmt='.2f', xticklabels=fmnist.classes, \
            yticklabels=[str(i)+str(' pixels') \
                         for i in range(-5,6)], cmap='gray')
```

The preceding code results in the following output:



Now, when we predict for various translations of an image, we'll see that the class prediction does not vary, thus ensuring that image translation is taken care of by training our model on augmented, translated images.

So far, we have seen how a CNN model trained with augmented images can predict well on translated images. In the next section, we'll understand what the filters learn, which makes predicting translated images possible.

Visualizing the outcome of feature learning

So far, we have learned about how CNNs help us classify images, even when the objects in the images have been translated. We have also learned that filters play a key role in learning the features of an image, which, in turn, help in classifying the image into the right class. However, we haven't mentioned what the filters learn that makes them powerful.

In this section, we will learn about what these filters learn that enables CNNs to classify an image correctly by classifying a dataset that contains images of X's and O's. We will also examine the fully connected layer (flatten layer) to understand what their activations look like. Let's take a look at what the filters learn:

> The code for this section is available as `Visualizing_the_features'_learning.ipynb` in the `Chapter04` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`.

1. Download the dataset:

```
!wget https://www.dropbox.com/s/5jh4hpuk2gcxaaq/all.zip
!unzip all.zip
```

Note that the images in the folder are named as follows:

```
all/o@InterconnectedDemo-Bold@IttL47.png
all/o@Refresh-Regular@LX2MG4.png
all/x@CallistaOllander@7EWgpq.png
all/x@ChristmasSeason@xZ7mjB.png
```

The class of an image can be obtained from the image's name, where the first character of the image's name specifies the class the image belongs to.

2. Import the required modules:

```
import torch
from torch import nn
from torch.utils.data import TensorDataset,Dataset,DataLoader
from torch.optim import SGD, Adam
device = 'cuda' if torch.cuda.is_available() else 'cpu'
from torchvision import datasets
import numpy as np, cv2
import matplotlib.pyplot as plt
%matplotlib inline
from glob import glob
from imgaug import augmenters as iaa
```

3. Define a class that fetches data. Also, ensure that the images have been resized to a shape of 28 x 28, batches have been shaped with three channels, and that the dependent variable is fetched as a numeric value. We'll do this in the following code, one step at a time:

   - Define the image augmented method, which resizes the image to a shape of 28 x 28:

```
tfm = iaa.Sequential(iaa.Resize(28))
```

   - Define a class that takes the folder path as input and loops through the files in that path in the `__init__` method:

```
class XO(Dataset):
    def __init__(self, folder):
        self.files = glob(folder)
```

   - Define the `__len__` method, which returns the lengths of the files that are to be considered:

```
    def __len__(self): return len(self.files)
```

   - Define the `__getitem__` method, which we use to fetch an index that returns the file present at that index, read the file, and then perform augmentation on the image. We have not used `collate_fn` here because this is a small dataset and it wouldn't affect the training time significantly:

```
    def __getitem__(self, ix):
        f = self.files[ix]
        im = tfm.augment_image(cv2.imread(f)[:,:,0])
```

- Given that each image is of the shape 28 x 28, we'll now create a dummy channel dimension at the beginning of the shape; that is, before the height and width of an image:

```
        im = im[None]
```

- Now, we can assign the class of each image based on the character post `'/'` and prior to `'@'` in the filename:

```
        cl = f.split('/')[-1].split('@')[0] == 'x'
```

- Finally, we return the image and the corresponding class:

```
        return torch.tensor(1 - im/255).to(device).float(), \
                      torch.tensor([cl]).float().to(device)
```

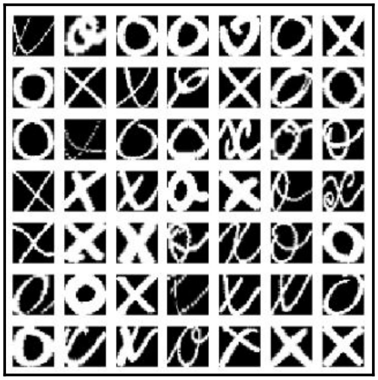
4. Inspect a sample of the images you've obtained. In the following code, we're extracting the images and their corresponding classes by fetching data from the class we defined previously:

```
data = XO('/content/all/*')
```

- Now, we can plot a sample of the images from the dataset we've obtained:

```
R, C = 7,7
fig, ax = plt.subplots(R, C, figsize=(5,5))
for label_class, plot_row in enumerate(ax):
    for plot_cell in plot_row:
        plot_cell.grid(False); plot_cell.axis('off')
        ix = np.random.choice(1000)
        im, label = data[ix]
        print()
        plot_cell.imshow(im[0].cpu(), cmap='gray')
plt.tight_layout()
```

The preceding code results in the following output:



5. Define the model architecture, loss function, and the optimizer:

```
from torch.optim import SGD, Adam
def get_model():
    model = nn.Sequential(
                nn.Conv2d(1, 64, kernel_size=3),
                nn.MaxPool2d(2),
                nn.ReLU(),
                nn.Conv2d(64, 128, kernel_size=3),
                nn.MaxPool2d(2),
                nn.ReLU(),
                nn.Flatten(),
                nn.Linear(3200, 256),
                nn.ReLU(),
                nn.Linear(256, 1),
                nn.Sigmoid()
            ).to(device)

    loss_fn = nn.BCELoss()
    optimizer = Adam(model.parameters(), lr=1e-3)
    return model, loss_fn, optimizer
```

Note that the loss function is binary cross-entropy loss (`nn.BCELoss()`) since the output provided is from a binary class. A summary of the preceding model can be obtained as follows:

```
!pip install torch_summary
from torchsummary import summary
model, loss_fn, optimizer = get_model()
summary(model, torch.zeros(1,1,28,28));
```

This results in the following output:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 26, 26]             640
         MaxPool2d-2           [-1, 64, 13, 13]               0
              ReLU-3           [-1, 64, 13, 13]               0
            Conv2d-4          [-1, 128, 11, 11]          73,856
         MaxPool2d-5            [-1, 128, 5, 5]               0
              ReLU-6            [-1, 128, 5, 5]               0
           Flatten-7                 [-1, 3200]               0
            Linear-8                  [-1, 256]         819,456
              ReLU-9                  [-1, 256]               0
           Linear-10                    [-1, 1]             257
          Sigmoid-11                    [-1, 1]               0
================================================================
Total params: 894,209
Trainable params: 894,209
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.69
Params size (MB): 3.41
Estimated Total Size (MB): 4.10
----------------------------------------------------------------
```

6. Define a function for training on batches that takes images and their classes as input and returns their loss values and accuracy after backpropagation has been performed on top of the given batch of data:

```
def train_batch(x, y, model, opt, loss_fn):
    model.train()
    prediction = model(x)
    is_correct = (prediction > 0.5) == y
    batch_loss = loss_fn(prediction, y)
    batch_loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    return batch_loss.item(), is_correct[0]
```

7. Define a `DataLoader` where the input is the `Dataset` class:

```
trn_dl = DataLoader(XO('/content/all/*'), batch_size=32, \
                    drop_last=True)
```

8. Initialize the model:

```
model, loss_fn, optimizer = get_model()
```

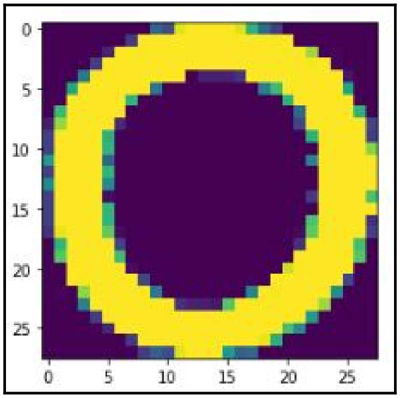
9. Train the model over `5` epochs:

```
for epoch in range(5):
    for ix, batch in enumerate(iter(trn_dl)):
        x, y = batch
        batch_loss = train_batch(x, y, model, optimizer, \
                                              loss_fn)
```

10. Fetch an image to check what the filters learn about the image:

```
im, c = trn_dl.dataset[2]
plt.imshow(im[0].cpu())
plt.show()
```

This results in the following output:



11. Pass the image through the trained model and fetch the output of the first layer. Then, store it in the `intermediate_output` variable:

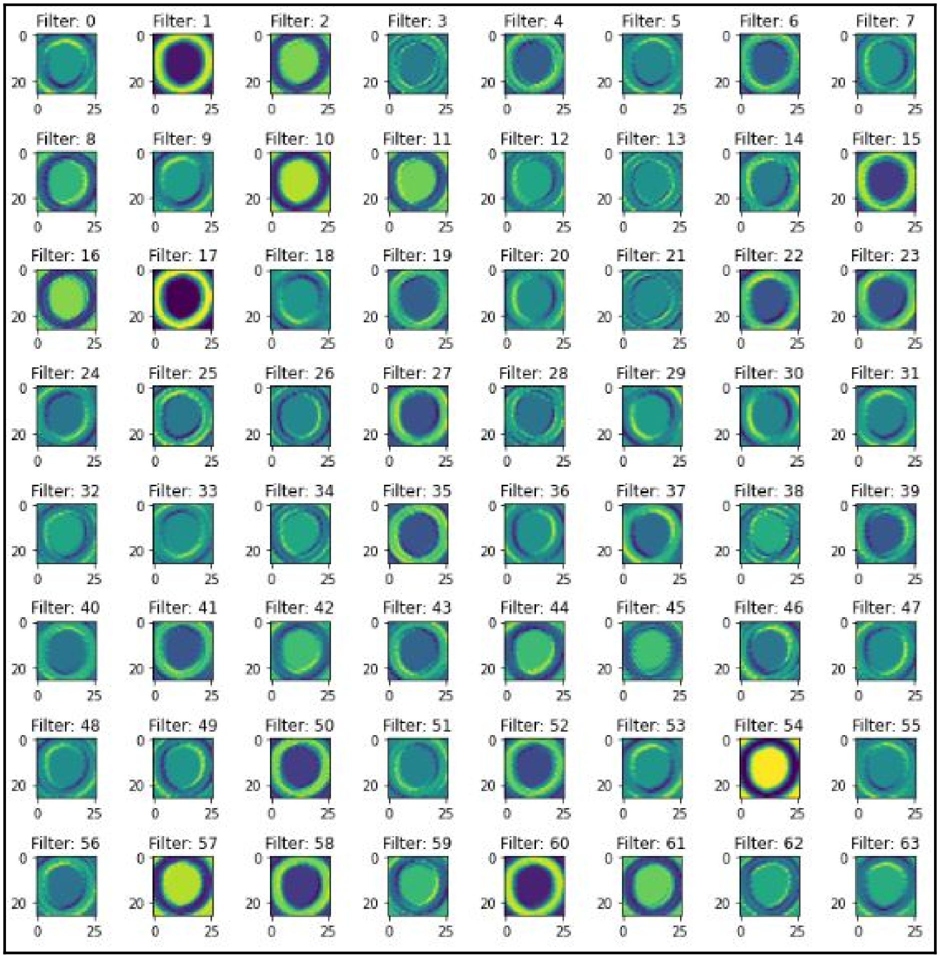
```
first_layer = nn.Sequential(*list(model.children())[:1])
intermediate_output = first_layer(im[None])[0].detach()
```

12. Plot the output of the 64 filters. Each channel in `intermediate_output` is the output of the convolution for each filter:

```
fig, ax = plt.subplots(8, 8, figsize=(10,10))
```

```
for ix, axis in enumerate(ax.flat):
    axis.set_title('Filter: '+str(ix))
    axis.imshow(intermediate_output[ix].cpu())
plt.tight_layout()
plt.show()
```

This results in the following output:



In the preceding output, notice that certain filters, such as filters 0, 4, 6, and 7, learn about the edges present in the network, while other filters, such as filter 54, learned to invert the image.

13. Pass multiple O images and inspect the output of the fourth filter across the images (we are only using the fourth filter for illustration purposes; you can choose a different filter if you wish):

    - Fetch multiple O images from the data:

    ```
    x, y = next(iter(trn_dl))
    x2 = x[y==0]
    ```

    - Reshape `x2` so that it has a proper input shape for a CNN model; that is, batch size x channels x height x width:

    ```
    x2 = x2.view(-1,1,28,28)
    ```

    - Define a variable that stores the model until the first layer:

    ```
    first_layer = nn.Sequential(*list(model.children())[:1])
    ```

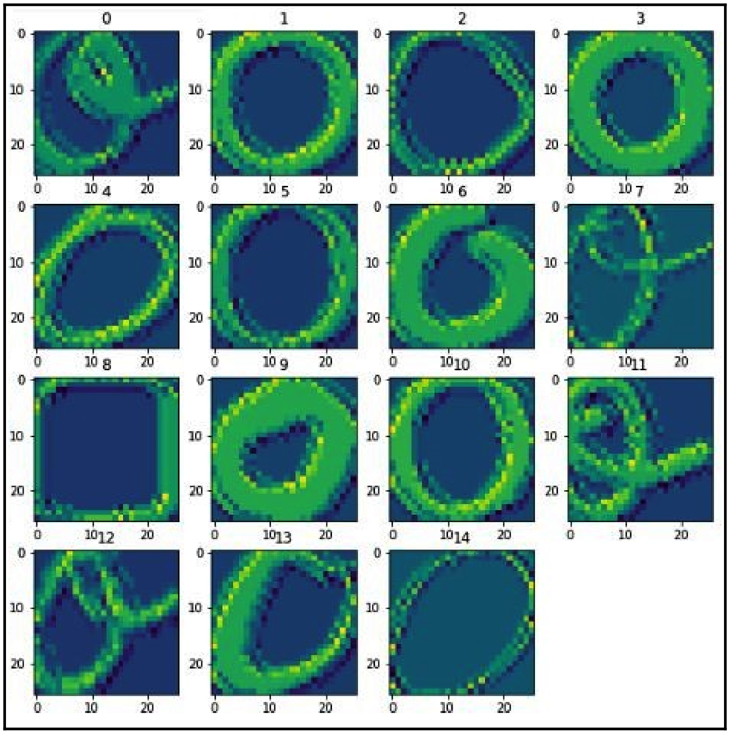
    - Extract the output of passing the O images (`x2`) through the model until the first layer (`first_layer`), as defined previously:

    ```
    first_layer_output = first_layer(x2).detach()
    ```

14. Plot the output of passing multiple images through the `first_layer` model:

    ```
    n = 4
    fig, ax = plt.subplots(n, n, figsize=(10,10))
    for ix, axis in enumerate(ax.flat):
        axis.imshow(first_layer_output[ix,4,:,:].cpu())
        axis.set_title(str(ix))
    plt.tight_layout()
    plt.show()
    ```

The preceding code results in the following output:



Note that the behavior of a given filter (in this case, the fourth filter of the first layer) has remained consistent across images.

15. Now, let's create another model that extracts layers until the second convolution layer (that is, until the four layers defined in the preceding model) and then extracts the output of passing the original O image. We will then plot the output of convolving the filters in the second layer with the input O image:

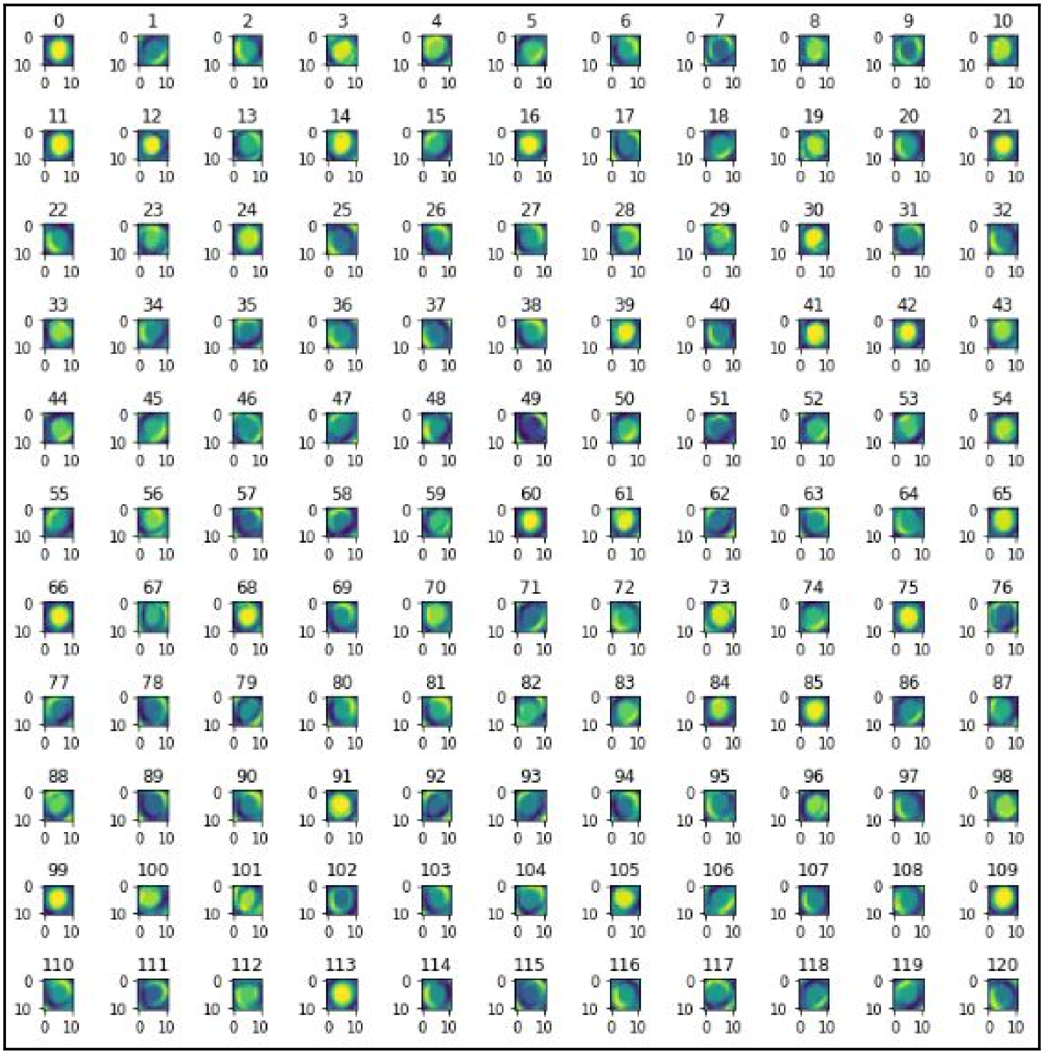
```
second_layer = nn.Sequential(*list(model.children())[:4])
second_intermediate_output=second_layer(im[None])[0].detach()
```

- Plot the output of convolving the filters with the respective image:

```
fig, ax = plt.subplots(11, 11, figsize=(10,10))
for ix, axis in enumerate(ax.flat):
```

```
        axis.imshow(second_intermediate_output[ix].cpu())
        axis.set_title(str(ix))
    plt.tight_layout()
    plt.show()
```

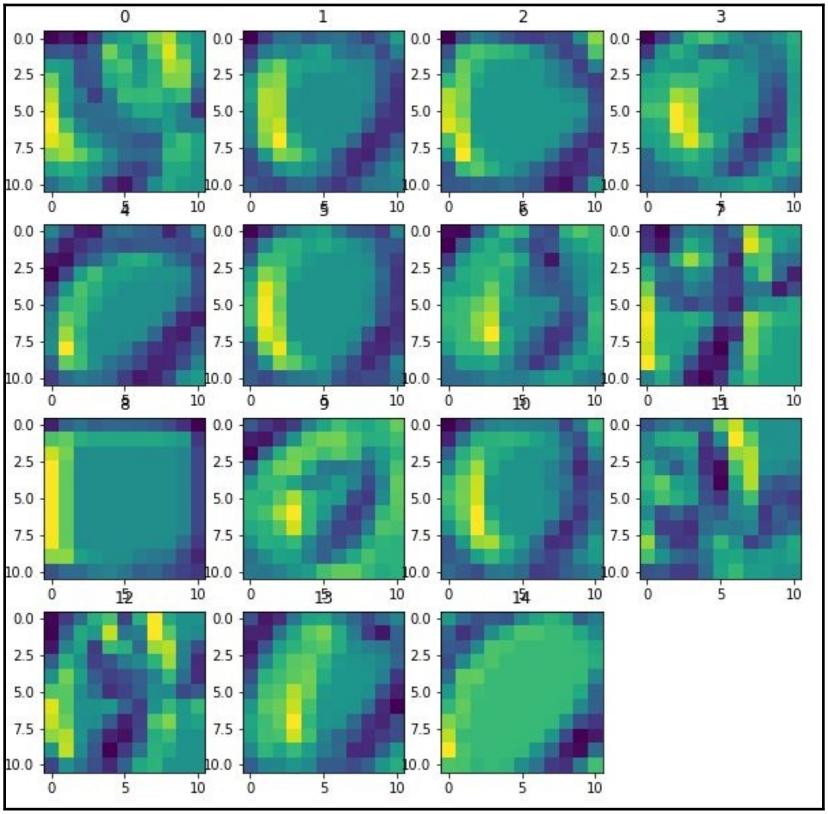
The preceding code results in the following output:



Now, let's use the 34th filter's output in the preceding image as an example. When we pass multiple O images through filter 34, we should see similar activations across images. Let's test this, as follows:

```
second_layer = nn.Sequential(*list(model.children())[:4])
second_intermediate_output = second_layer(x2).detach()
fig, ax = plt.subplots(4, 4, figsize=(10,10))
for ix, axis in enumerate(ax.flat):
    axis.imshow(second_intermediate_output[ix,34,:,:].cpu())
    axis.set_title(str(ix))
plt.tight_layout()
plt.show()
```

The preceding code results in the following output:



Note that, even here, the activations of the 34th filter on different images are similar in that the left half of O was activating the filter.

16. Plot the activations of a fully connected layer, as follows:

    - First, fetch a larger sample of images:

```
custom_dl= DataLoader(XO('/content/all/*'),batch_size=2498, \
                              drop_last=True)
```

    - Next, choose only the O images from the dataset and then reshape them so that they can be passed as input to our CNN model:

```
x, y = next(iter(custom_dl))
x2 = x[y==0]
x2 = x2.view(len(x2),1,28,28)
```

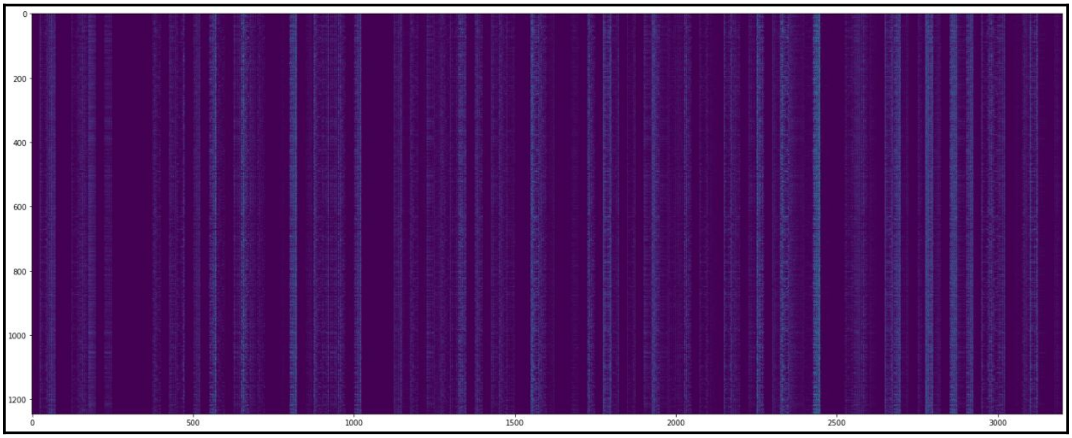
    - Fetch the flatten (fully connected) layer and pass thee preceding images through the model until they reach the flattened layer:

```
flatten_layer = nn.Sequential(*list(model.children())[:7])
flatten_layer_output = flatten_layer(x2).detach()
```

    - Plot the flattened layer:

```
plt.figure(figsize=(100,10))
plt.imshow(flatten_layer_output.cpu())
```

    The preceding code results in the following output:



Note that the shape of the output is 1245 x 3200 since there are 1,245 **O** images in our dataset and there are 3,200 dimensions for each image in the flattening layer.

It's also interesting to note that certain values in the fully connected layer are highlighted when the input is **O** (here, we can see white lines, where each dot represents an activation value greater than zero).

> Note that the model has learned to bring some structure to the fully connected layer, even though the input images – while all belonging to the same class – differ in style considerably.

Now that we have learned how CNNs work and how filters aid in this process, we will apply this so that we can classify images of cats and dogs.

Building a CNN for classifying real-world images

So far, we have learned how to perform image classification on the Fashion-MNIST dataset. In this section, we'll do the same for a more real-world scenario, where the task is to classify images containing cats or dogs. We will also learn about how the accuracy of the dataset varies when we change the number of images available for training.

We will be working on a dataset available in Kaggle: `https://www.kaggle.com/tongpython/cat-and-dog`.

> The code for this section is available as `Cats_Vs_Dogs.ipynb` in the `Chapter04` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` Be sure to copy the URL from the notebook in GitHub to avoid any issue while reproducing the results

1. Import the necessary packages:

```
import torchvision
import torch.nn as nn
import torch
import torch.nn.functional as F
from torchvision import transforms,models,datasets
from PIL import Image
from torch import optim
```

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
import cv2, glob, numpy as np, pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
from glob import glob
!pip install torch_summary
```

2. Download the dataset, as follows:

   - Here, we must download the dataset that's available in the `colab` environment. First, however, we must upload our Kaggle authentication file:

```
!pip install -q kaggle
from google.colab import files
files.upload()
```

You will have to upload your `kaggle.json` file for this step, which can be obtained from your Kaggle account. A detail of how to obtain the `kaggle.json` file is provided in the associated notebook on GitHub

   - Next, specify that we're moving to the Kaggle folder and copy the `kaggle.json` file to it:

```
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!ls ~/.kaggle
!chmod 600 /root/.kaggle/kaggle.json
```

   - Finally, download the cats and dogs dataset and unzip it:

```
!kaggle datasets download -d tongpython/cat-and-dog
!unzip cat-and-dog.zip
```

3. Provide the training and test dataset folders:

```
train_data_dir = '/content/training_set/training_set'
test_data_dir = '/content/test_set/test_set'
```

4. Build a class that fetches data from the preceding folders. Then, based on the directory the image corresponds to, provide a label of 1 for "dog" images and a label of 0 for "cat" images. Furthermore, ensure that the fetched image has been normalized to a scale between 0 and 1 and permute it so that channels are provided first (as PyTorch models expect to have channels specified first, before the height and width of the image).

    - Define the `__init__` method, which takes a folder as input and stores the file paths (image paths) corresponding to the images in the `cats` and `dogs` folders in separate objects, post concatenating the file paths into a single list:

```
from torch.utils.data import DataLoader, Dataset
class cats_dogs(Dataset):
    def __init__(self, folder):
        cats = glob(folder+'/cats/*.jpg')
        dogs = glob(folder+'/dogs/*.jpg')
        self.fpaths = cats + dogs
```

    - Next, randomize the file paths and create target variables based on the folder corresponding to these file paths:

```
        from random import shuffle, seed; seed(10);
        shuffle(self.fpaths)
        self.targets=[fpath.split('/')[-1].startswith('dog') \
                      for fpath in self.fpaths] # dog=1
```

    - Define the `__len__` method, which corresponds to the `self` class:

```
    def __len__(self): return len(self.fpaths)
```

    - Define the `__getitem__` method, which we use to specify a random file path from the list of file paths, read the image, and resize all the images so that they're 224 x 224 in size. Given that our CNN expects the inputs from the channel to be specified first for each image, we will `permute` the resized image so that channels are provided first before we return the scaled image and the corresponding `target` value:

```
    def __getitem__(self, ix):
        f = self.fpaths[ix]
        target = self.targets[ix]
        im = (cv2.imread(f)[:,:,::-1])
        im = cv2.resize(im, (224,224))
        return torch.tensor(im/255).permute(2,0,1)\
```

```
            .to(device).float(),\
        torch.tensor([target]) \
            .float().to(device)
```

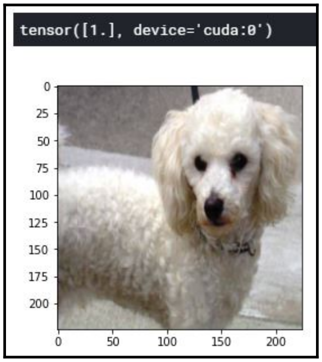
5. Inspect a random image:

```
data = cats_dogs(train_data_dir)
im, label = data[200]
```

We need to permute the image we've obtained to our channels last. This is because matplotlib expects an image to have the channels specified after the height and width of the image has been provided:

```
plt.imshow(im.permute(1,2,0).cpu())
print(label)
```

This results in the following output:



6. Define a model, loss function, and optimizer, as follows:

 - First, we must define the `conv_layer` function, where we perform convolution, ReLU activation, batch normalization, and max pooling in that order. This method will be reused in the final model, which we will define in the next step:

```
def conv_layer(ni,no,kernel_size,stride=1):
    return nn.Sequential(
        nn.Conv2d(ni, no, kernel_size, stride),
        nn.ReLU(),
        nn.BatchNorm2d(no),
        nn.MaxPool2d(2)
    )
```

In the preceding code, we are taking the number of input channels (`ni`), number of output channels (`no`), `kernel_size`, and the `stride` of filters as input for the `conv_layer` function.

- Define the `get_model` function, which performs multiple convolutions and pooling operations (by calling the `conv_layer` method), flattens the output, and connects a hidden layer to it prior to connecting to the output layer:

```
def get_model():
    model = nn.Sequential(
              conv_layer(3, 64, 3),
              conv_layer(64, 512, 3),
              conv_layer(512, 512, 3),
              conv_layer(512, 512, 3),
              conv_layer(512, 512, 3),
              conv_layer(512, 512, 3),
              nn.Flatten(),
              nn.Linear(512, 1),
              nn.Sigmoid(),
            ).to(device)
    loss_fn = nn.BCELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr= 1e-3)
    return model, loss_fn, optimizer
```

You can chain `nn.Sequential` inside `nn.Sequential` with as much depth as you want. In the preceding code, we used `conv_layer` as if it were any other `nn.Module` layer.

- Now, we must call the `get_model` function to fetch the model, loss function (`loss_fn`), and `optimizer` and then summarize the model using the `summary` method that we imported from the `torchsummary` package:

```
from torchsummary import summary
model, loss_fn, optimizer = get_model()
summary(model, torch.zeros(1,3, 224, 224));
```

The preceding code results in the following output:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 222, 222]           1,792
              ReLU-2         [-1, 64, 222, 222]               0
       BatchNorm2d-3         [-1, 64, 222, 222]             128
         MaxPool2d-4         [-1, 64, 111, 111]               0
            Conv2d-5        [-1, 512, 109, 109]         295,424
              ReLU-6        [-1, 512, 109, 109]               0
       BatchNorm2d-7        [-1, 512, 109, 109]           1,024
         MaxPool2d-8          [-1, 512, 54, 54]               0
            Conv2d-9          [-1, 512, 52, 52]       2,359,808
             ReLU-10          [-1, 512, 52, 52]               0
      BatchNorm2d-11          [-1, 512, 52, 52]           1,024
        MaxPool2d-12          [-1, 512, 26, 26]               0
           Conv2d-13          [-1, 512, 24, 24]       2,359,808
             ReLU-14          [-1, 512, 24, 24]               0
      BatchNorm2d-15          [-1, 512, 24, 24]           1,024
        MaxPool2d-16          [-1, 512, 12, 12]               0
           Conv2d-17          [-1, 512, 10, 10]       2,359,808
             ReLU-18          [-1, 512, 10, 10]               0
      BatchNorm2d-19          [-1, 512, 10, 10]           1,024
        MaxPool2d-20            [-1, 512, 5, 5]               0
           Conv2d-21            [-1, 512, 3, 3]       2,359,808
             ReLU-22            [-1, 512, 3, 3]               0
      BatchNorm2d-23            [-1, 512, 3, 3]           1,024
        MaxPool2d-24            [-1, 512, 1, 1]               0
          Flatten-25                  [-1, 512]               0
           Linear-26                    [-1, 1]             513
          Sigmoid-27                    [-1, 1]               0
================================================================
Total params: 9,742,209
Trainable params: 9,742,209
Non-trainable params: 0
```

7. Create the `get_data` function, which creates an object of the `cats_dogs` class and creates a `DataLoader` with a `batch_size` of 32 for both the training and validation folders:

```
def get_data():
    train = cats_dogs(train_data_dir)
    trn_dl = DataLoader(train, batch_size=32, shuffle=True, \
                        drop_last = True)
    val = cats_dogs(test_data_dir)
    val_dl = DataLoader(val, batch_size=32, shuffle=True, \
                        drop_last = True)
    return trn_dl, val_dl
```

In the preceding code, we are ignoring the last batch of data by specifying that `drop_last = True`. We're doing this because the last batch might not be the same size as the other batches.

8. Define the function that will train the model on a batch of data, as we've done in previous sections:

```
def train_batch(x, y, model, opt, loss_fn):
    model.train()
    prediction = model(x)
    batch_loss = loss_fn(prediction, y)
    batch_loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    return batch_loss.item()
```

9. Define the functions for calculating accuracy and validation loss, as we've done in previous sections:

   - Define the `accuracy` function:

```
@torch.no_grad()
def accuracy(x, y, model):
    prediction = model(x)
    is_correct = (prediction > 0.5) == y
    return is_correct.cpu().numpy().tolist()
```

   Note that the preceding code for accuracy calculation is different from the code in the Fashion-MNIST classification because the current model (cats versus dogs classification) is being built for binary classification, while the Fashion-MNIST model was built for multi-class classification.

   - Define the validation loss calculation function:

```
@torch.no_grad()
def val_loss(x, y, model):
    prediction = model(x)
    val_loss = loss_fn(prediction, y)
    return val_loss.item()
```

10. Train the model for `5` epochs and check the accuracy of the test data at the end of each epoch, as we've done in previous sections:

    - Define the model and fetch the required DataLoaders:

    ```
    trn_dl, val_dl = get_data()
    model, loss_fn, optimizer = get_model()
    ```

    - Train the model over increasing epochs:

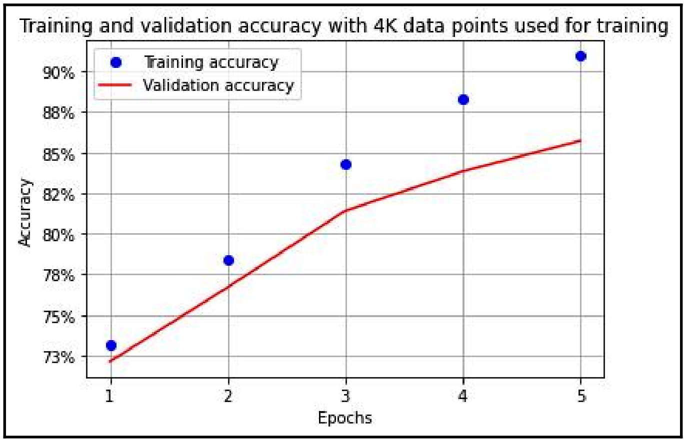
    ```
    train_losses, train_accuracies = [], []
    val_losses, val_accuracies = [], []
    for epoch in range(5):
        train_epoch_losses, train_epoch_accuracies = [], []
        val_epoch_accuracies = []
        for ix, batch in enumerate(iter(trn_dl)):
            x, y = batch
            batch_loss = train_batch(x, y, model, optimizer, \
                                                        loss_fn)
            train_epoch_losses.append(batch_loss)
        train_epoch_loss = np.array(train_epoch_losses).mean()

        for ix, batch in enumerate(iter(trn_dl)):
            x, y = batch
            is_correct = accuracy(x, y, model)
            train_epoch_accuracies.extend(is_correct)
        train_epoch_accuracy = np.mean(train_epoch_accuracies)

        for ix, batch in enumerate(iter(val_dl)):
            x, y = batch
            val_is_correct = accuracy(x, y, model)
            val_epoch_accuracies.extend(val_is_correct)
        val_epoch_accuracy = np.mean(val_epoch_accuracies)

        train_losses.append(train_epoch_loss)
        train_accuracies.append(train_epoch_accuracy)
        val_accuracies.append(val_epoch_accuracy)
    ```

11. Plot the variation of the training and validation accuracies over increasing epochs:

    ```
    epochs = np.arange(5)+1
    import matplotlib.ticker as mtick
    import matplotlib.pyplot as plt
    import matplotlib.ticker as mticker
    %matplotlib inline
    plt.plot(epochs, train_accuracies, 'bo',
             label='Training accuracy')
    ```

```
plt.plot(epochs, val_accuracies, 'r',
         label='Validation accuracy')
plt.gca().xaxis.set_major_locator(mticker.MultipleLocator(1))
plt.title('Training and validation accuracy \
with 4K data points used for training')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.gca().set_yticklabels(['{:.0f}%'.format(x*100) \
                           for x in plt.gca().get_yticks()])
plt.legend()
plt.grid('off')
plt.show()
```

The preceding code results in the following output:



Note that the classification accuracy at the end of `5` epochs is ~86%.

As we discussed in the previous chapter, batch normalization has a great impact on improving classification accuracy – check this out for yourself by training the model without batch normalization. Furthermore, the model can be trained without batch normalization if you use fewer parameters. You can do this by reducing the number of layers, increasing the stride, increasing the pooling, or resizing the image to a number that's lower than 224 x 224.

So far, the training we've done has been based on ~8K examples, where 4K examples have been from the `cat` class and the rest have been from the `dog` class. In the next section, we will learn about what impact having a reduced number of training examples has on each class when it comes to the classification accuracy of the test dataset.

Impact on the number of images used for training

We know that, generally, the more training examples we use, the better our classification accuracy is. In this section, we will learn what impact using different numbers of available images has on training accuracy by artificially reducing the number of images available for training and then testing the model's accuracy when classifying the test dataset.

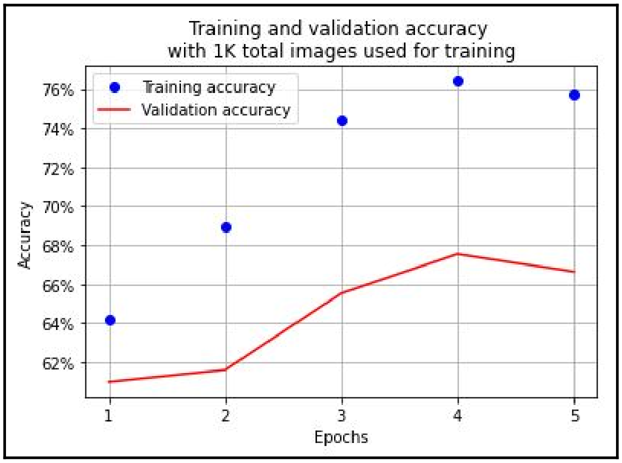
> The code for this section is available as `Cats_Vs_Dogs.ipynb` in the `Chapter04` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` . Given that the majority of the code that will be provided here is similar to what we have seen in the previous section, in text, we have only provided the modified code for brevity. The respective notebook in this book's GitHub repository will contain the full code.

Here, we only want to have 500 data points for each class in the training dataset. We can do this by limiting the number of files to only the first 500 image paths in each folder in the `__init__` method and ensuring the rest remain as they were in the previous section:

```
def __init__(self, folder):
    cats = glob(folder+'/cats/*.jpg')
    dogs = glob(folder+'/dogs/*.jpg')
    self.fpaths = cats[:500] + dogs[:500]
    from random import shuffle, seed; seed(10);
        shuffle(self.fpaths)
    self.targets = [fpath.split('/')[-1].startswith('dog') \
                    for fpath in self.fpaths]
```

In the preceding code, the only difference from the initialization we performed in the previous section is in `self.paths`, where we are now limiting the number of file paths to be considered in each folder to only the first 500.

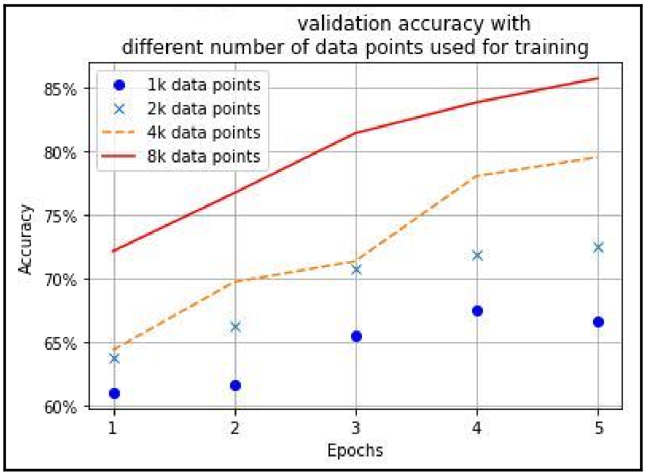
Now, once we execute the rest of the code, as we did in the previous section, the accuracy of the model that's been built on 1,000 images (500 of each class) in the test dataset will be as follows:



Here, we can see that because we had fewer examples of images in training, the accuracy of the test dataset reduced considerably; that is, down to ~66%.

Now, let's see how the number of training data points impacts the accuracy of the test dataset by varying the number of available training examples that will be used to train the model (where we build a model for each scenario).

We'll use the same code we used for the 500 data point training example but will vary the number of available images (to 2K, 4K, and 8K total data points, respectively). For brevity, we will only look at the output of running the model on a varying number of images available for training. This results in the following output:



As you can see, the more training data that's available, the higher the accuracy of the model on test data. However, we might not have a large enough amount of training data in every scenario that we encounter. The next chapter, which will cover transfer learning, will address this problem by walking you through various techniques you can use to attain high accuracy, even on a small amount of training data.

Summary

Traditional neural networks fail when new images that are very similar to previously seen images that have been translated are fed as input to the model. Convolutional neural networks play a key role in addressing this shortcoming. This is enabled through the various mechanisms that are present in CNNs, including filters, strides, and pooling. Initially, we built a toy example to learn about how CNNs work. Then, we learned about how data augmentation helps in increasing the accuracy of the model by creating translated augmentations on top of the original image. After that, we learned about what different filters learn in the feature learning process so that we could implement a CNN to classify images.

Finally, we saw the impact that differing amounts of training data have on the accuracy of test data. Here, we saw that the more training data that is available, the better the accuracy of the test data. In the next chapter, we will learn about how to leverage various transfer learning techniques to increase the accuracy of the test dataset, even when we have just a small amount of training data.

Questions

1. Why is the prediction on a translated image low when using traditional neural networks?
2. How is convolution done?
3. How are optimal weight values in a filter identified?
4. How does the combination of convolution and pooling help in addressing the issue of image translation?
5. What do the filters in layers closer to the input layer learn?
6. What functionality does pooling have that helps in building a model?

7. Why can't we take an input image, flatten it (as we did on the Fashion-MNIST dataset), and then train a model for real-world images?
8. How does data augmentation help in improving image translation?
9. In what scenario do we leverage `collate_fn` for DataLoaders?
10. What impact does varying the number of training data points have on the classification accuracy of the validation dataset?

5
Transfer Learning for Image Classification

In the previous chapter, we learned that, as the number of images available in the training dataset increased, the classification accuracy of the model kept on increasing, to the extent where a training dataset comprising 8,000 images had a higher accuracy on validation dataset than a training dataset comprising 1,000 images. However, we do not always have the option of hundreds or thousands of images, along with the ground truths of their corresponding classes, in order to train a model. This is where transfer learning comes to the rescue.

Transfer learning is a technique where we transfer the learning of the model on a generic dataset to the specific dataset of interest. Typically, the pre-trained models used to perform transfer learning are trained on millions of images (which are generic and not the dataset of interest to us) and those pre-trained models are now fine-tuned to our dataset of interest.

In this chapter, we will learn about two different families of transfer learning architectures – variants of VGG architecture, and variants of ResNet architecture.

Along with understanding the architectures, we will also understand their application in two different use cases, age and gender classification, where we will learn about optimizing over both cross-entropy and mean absolute error losses at the same time, and facial key point detection, where we will learn about leveraging neural networks to generate multiple (136, instead of 1 prediction) continuous outputs in a single prediction. Finally, we will learn about a new library that assists in reducing code complexity considerably across the remaining chapters.

In summary, the following topics are covered in the chapter:

- Introducing transfer learning
- Understanding VGG16 and ResNet architectures
- Implementing facial key point detection
- Multi-task learning: Implementing age estimation and gender classification
- Introducing the torch_snippets library

Introducing transfer learning

Transfer learning is a technique where knowledge gained from one task is leveraged to solve another similar task.

Imagine a model that is trained on millions of images that span thousands of classes of objects (not just cats and dogs). The various filters (kernels) of the model would activate for a wide variety of shapes, colors, and textures within the images. Those filters can now be reused to learn features on a new set of images. Post learning the features, they can be connected to a hidden layer prior to the final classification layer for customizing on the new data.

ImageNet (`http://www.image-net.org/`) is a competition hosted to classify approximately 14 million images into 1,000 different classes. It has a variety of classes in the dataset, including Indian elephant, lionfish, hard disk, hair spray, and jeep.

The deep neural network architectures that we will go through in this chapter have been trained on the ImageNet dataset. Furthermore, given the variety and the volume of objects that are to be classified in ImageNet, the models are very deep so as to capture as much information as possible.

Let's understand the importance of transfer learning through a hypothetical scenario:

Consider a situation where we are working with images of a road, trying to classify them in terms of the objects they contain. Building a model from scratch might result in sub-optimal results, as the number of images could be insufficient to learn the various variations within the dataset (as we have seen in the previous use case, where training on 8,000 images resulted in a higher accuracy on a validation dataset than training on 2,000 images). A pre-trained model, trained on ImageNet, comes in handy in such a scenario. It would have already learned a lot about the traffic-related classes, such as cars, roads, trees, and humans, during training on the large ImageNet dataset. Hence, leveraging the already trained model would result in faster and more accurate training as the model already knows the generic shapes and now has to fit them for the specific images. With the intuition in place, let's now understand the high-level flow of transfer learning as follows:

1. Normalize the input images, normalized by the **same mean and standard deviation** that was used during the training of the pre-trained model.
2. Fetch the pre-trained model's architecture. Fetch the weights for this architecture that arose as a result of being trained on a large dataset.
3. Discard the last few layers of the pre-trained model.
4. Connect the truncated pre-trained model to a freshly initialized layer (or layers) where weights are randomly initialized. Ensure that the output of the last layer has as many neurons as the classes/outputs we would want to predict
5. Ensure that the weights of the pre-trained model are not trainable (in other words, frozen/not updated during backpropagation), but that the weights of the newly initialized layer and the weights connecting it to the output layer are trainable:

    - We do not train the weights of the pre-trained model, as we assume those weights are already well learned for the task, and hence leverage the learning from a large model. In summary, we only learn the newly initialized layers for our small dataset.

6. Update the trainable parameters over increasing epochs to fit a model.

Now that we have an idea of how to implement transfer learning, let's understand the various architectures, how they are built, and the results when we apply transfer learning to the cats versus dogs use case in subsequent sections. First, we will cover in detail some of the various architectures that came out of **VGG**.

Understanding VGG16 architecture

VGG stands for **Visual Geometry Group**, which is based out of the University of Oxford, and 16 stands for the number of layers in the model. The VGG16 model is trained to classify objects in the ImageNet competition and stood as the runner-up architecture in 2014. The reason we are studying this architecture instead of the winning architecture (GoogleNet) is because of its simplicity and a larger acceptance in the vision community by using it in several other tasks. Let's understand the architecture of VGG16 along with how a VGG16 pre-trained model is accessible and represented in PyTorch.

> The code for this section is available as `VGG_architecture.ipynb` in the `Chapter05` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Install the required packages:

```
import torchvision
import torch.nn as nn
import torch
import torch.nn.functional as F
from torchvision import transforms,models,datasets
!pip install torch_summary
from torchsummary import summary
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

   The `models` module in the `torchvision` package hosts the various pre-trained models available in PyTorch.

2. Load the VGG16 model and register the model within the device:

```
model = models.vgg16(pretrained=True).to(device)
```

   In the preceding code, we have called the `vgg16` method within the `models` class. Furthermore, by mentioning `pretrained = True`, we are specifying that we load the weights that were used to classify images in the ImageNet competition, and then we are registering the model to the device.

3. Fetch the summary of the model:

```
summary(model, torch.zeros(1,3,224,224));
```

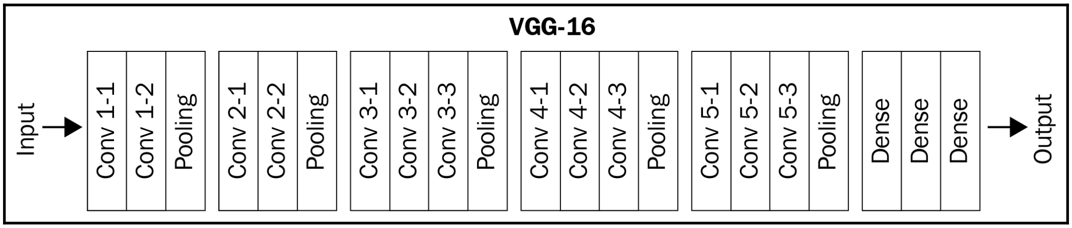
The output of the preceding code is as follows:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 224, 224]           1,792
              ReLU-2         [-1, 64, 224, 224]               0
            Conv2d-3         [-1, 64, 224, 224]          36,928
              ReLU-4         [-1, 64, 224, 224]               0
         MaxPool2d-5         [-1, 64, 112, 112]               0
            Conv2d-6        [-1, 128, 112, 112]          73,856
              ReLU-7        [-1, 128, 112, 112]               0
            Conv2d-8        [-1, 128, 112, 112]         147,584
              ReLU-9        [-1, 128, 112, 112]               0
        MaxPool2d-10          [-1, 128, 56, 56]               0
           Conv2d-11          [-1, 256, 56, 56]         295,168
             ReLU-12          [-1, 256, 56, 56]               0
           Conv2d-13          [-1, 256, 56, 56]         590,080
             ReLU-14          [-1, 256, 56, 56]               0
           Conv2d-15          [-1, 256, 56, 56]         590,080
             ReLU-16          [-1, 256, 56, 56]               0
        MaxPool2d-17          [-1, 256, 28, 28]               0
           Conv2d-18          [-1, 512, 28, 28]       1,180,160
             ReLU-19          [-1, 512, 28, 28]               0
           Conv2d-20          [-1, 512, 28, 28]       2,359,808
             ReLU-21          [-1, 512, 28, 28]               0
           Conv2d-22          [-1, 512, 28, 28]       2,359,808
             ReLU-23          [-1, 512, 28, 28]               0
        MaxPool2d-24          [-1, 512, 14, 14]               0
           Conv2d-25          [-1, 512, 14, 14]       2,359,808
             ReLU-26          [-1, 512, 14, 14]               0
           Conv2d-27          [-1, 512, 14, 14]       2,359,808
             ReLU-28          [-1, 512, 14, 14]               0
           Conv2d-29          [-1, 512, 14, 14]       2,359,808
             ReLU-30          [-1, 512, 14, 14]               0
        MaxPool2d-31            [-1, 512, 7, 7]               0
AdaptiveAvgPool2d-32            [-1, 512, 7, 7]               0
           Linear-33                 [-1, 4096]     102,764,544
             ReLU-34                 [-1, 4096]               0
          Dropout-35                 [-1, 4096]               0
           Linear-36                 [-1, 4096]      16,781,312
             ReLU-37                 [-1, 4096]               0
          Dropout-38                 [-1, 4096]               0
           Linear-39                 [-1, 1000]       4,097,000
================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
----------------------------------------------------------------
```

In the preceding summary, the 16 layers we mentioned are grouped as follows:

```
{1,2},{3,4,5},{6,7},{8,9,10},{11,12},{13,14},{15,16,17},{18
,19},{20,21},{22,23,24},{25,26},{27,28},{29,30,31,32},{33,3
4,35},{36,37,38],{39}
```

The same summary can also be visualized thus:



Note that there are ~138 million parameters (of which ~122 million are the linear layers at the end of the network – 102 + 16 + 4 million parameters) in this network, which comprises 13 layers of convolution and/or pooling, with increasing number of filters, and 3 linear layers.

Another way to understand the components of the VGG16 model is by simply printing it as follows:

```
model
```

This results in the following output:

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Note that there are three major sub-modules in the model—`features`, `avgpool`, and `classifier`. Typically, we would freeze the `features` and `avgpool` modules. Delete the `classifier` module (or only a few layers at the bottom) and create a new one in its place that will predict the required number of classes corresponding to our dataset (instead of the existing 1,000).

Let's now understand how the VGG16 model is used in practice, using the cats versus dogs dataset (considering only 500 images in each class for training) in the following code:

The following code is available as `Implementing_VGG16_for_image_classification.ipynb` in the `Chapter05` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` Be sure to copy the URL from the notebook in GitHub to avoid any issue while reproducing the results

1. Install the required packages:

```
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
from torchvision import transforms,models,datasets
import matplotlib.pyplot as plt
from PIL import Image
from torch import optim
device = 'cuda' if torch.cuda.is_available() else 'cpu'
import cv2, glob, numpy as np, pandas as pd
from glob import glob
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, Dataset
```

2. Download the dataset and specify the training and test directories:

   - Download the dataset. Assuming that we are working on Google Colab, we perform the following steps, where we provide the authentication key and place it in a location where Kaggle can use the key to authenticate us and download the dataset:

```
!pip install -q kaggle
from google.colab import files
files.upload()
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!ls ~/.kaggle
!chmod 600 /root/.kaggle/kaggle.json
```

   - Download the dataset and unzip it:

```
!kaggle datasets download -d tongpython/cat-and-dog
!unzip cat-and-dog.zip
```

- Specify the training and test image folders:

```
train_data_dir = 'training_set/training_set'
test_data_dir = 'test_set/test_set'
```

3. Provide the class that returns input-output pairs for the cats and dogs dataset, just like we did in `Chapter 4`, *Introducing Convolutional Neural Networks*. Note that, in this case, we are fetching the first 500 images only from each folder:

```
class CatsDogs(Dataset):
    def __init__(self, folder):
        cats = glob(folder+'/cats/*.jpg')
        dogs = glob(folder+'/dogs/*.jpg')
        self.fpaths = cats[:500] + dogs[:500]
        self.normalize = transforms.Normalize(mean=[0.485,
                     0.456, 0.406],std=[0.229, 0.224, 0.225])
        from random import shuffle, seed; seed(10);
        shuffle(self.fpaths)
        self.targets =[fpath.split('/')[-1].startswith('dog')
\
                        for fpath in self.fpaths]
    def __len__(self): return len(self.fpaths)
    def __getitem__(self, ix):
        f = self.fpaths[ix]
        target = self.targets[ix]
        im = (cv2.imread(f)[:,:,::-1])
        im = cv2.resize(im, (224,224))
        im = torch.tensor(im/255)
        im = im.permute(2,0,1)
        im = self.normalize(im)
        return im.float().to(device),
            torch.tensor([target]).float().to(device)
```

The main difference between the `cats_dogs` class in this section and in chapter 4 is the `normalize` function that we are applying using the `Normalize` function from the `transforms` module.

While leveraging pre-trained models, it is mandatory to resize, permute, and then normalize images (as appropriate for that pre-trained model), where the images are first scaled to a value between 0 and 1 across the 3 channels and then normalized to a mean of [0.485, 0.456, 0.406] and a standard deviation of [0.229, 0.224, 0.225] across the RGB channels.

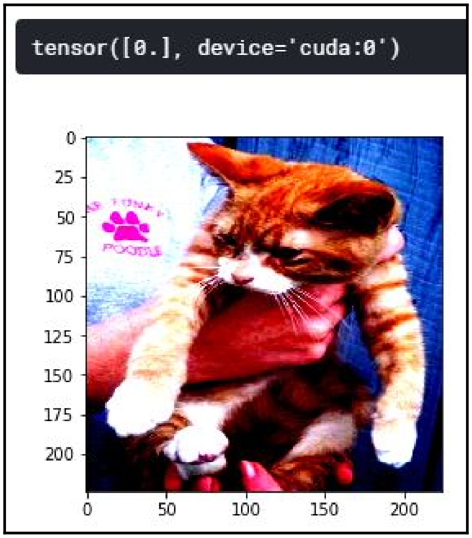
4. Fetch the images and their labels:

```
data = CatsDogs(train_data_dir)
```

Let's now inspect a sample image and its corresponding class:

```
im, label = data[200]
plt.imshow(im.permute(1,2,0).cpu())
print(label)
```

The preceding code results in the following output:



5. Define the model. Download the pre-trained VGG16 weights and then freeze the `features` module and train using the `avgpool` and `classifier` modules:

   - First, we download the pretrained VGG16 model from the `models` class:

```
def get_model():
    model = models.vgg16(pretrained=True)
```

   - Specify that we want to freeze all the parameters in the model downloaded previously:

```
    for param in model.parameters():
        param.requires_grad = False
```

In the preceding code, we are freezing parameter updates during backpropagation by specifying `param.requires_grad = False`.

- Replace the `avgpool` module to return a feature map of size 1 x 1 instead of 7 x 7, in other words, the output is now going to be `batch_size x 512 x 1 x 1`:

```
model.avgpool = nn.AdaptiveAvgPool2d(output_size=(1,1))
```

We have seen `nn.MaxPool2d`, where we are picking the maximum value from every section of a feature map. There is a counterpart to this layer called `nn.AvgPool2d`, which returns the average of a section instead of the maximum. In both these layers, we fix the kernel size. The layer above, `nn.AdaptiveAvgPool2d`, is yet another pooling layer with a twist. We specify the output feature map size instead. The layer automatically computes the kernel size so that the specified feature map size is returned. For example, if the input feature map size dimensions were `batch_size x 512 x k x k`, then the pooling kernel size is going to be `k x k`. The major advantage with this layer is that whatever the input size, the output from this layer is always fixed and, hence, the neural network can accept images of any height and width.

- Define the `classifier` module of the model, where we first flatten the output of the `avgpool` module, connect the 512 units to the 128 units, and perform an activation prior to connecting to the output layer:

```
model.classifier = nn.Sequential(nn.Flatten(),
                                 nn.Linear(512, 128),
                                 nn.ReLU(),
                                 nn.Dropout(0.2),
                                 nn.Linear(128, 1),
                                 nn.Sigmoid())
```

- Define the loss function (`loss_fn`), `optimizer`, and return them along with the defined model:

```
loss_fn = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(),lr= 1e-3)
return model.to(device), loss_fn, optimizer
```

Note that in the preceding code, we have first frozen all the parameters of the pre-trained model and have then overwritten the `avgpool` and `classifier` modules. Now, the rest of the code is going to look similar to what we have seen in the previous chapter.

A summary of the model is as follows:

```
!pip install torch_summary
from torchsummary import summary
model, criterion, optimizer = get_model()
summary(model, torch.zeros(1,3,224,224))
```

The preceding code results in the following output:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 224, 224]           1,792
              ReLU-2         [-1, 64, 224, 224]               0
            Conv2d-3         [-1, 64, 224, 224]          36,928
              ReLU-4         [-1, 64, 224, 224]               0
         MaxPool2d-5         [-1, 64, 112, 112]               0
            Conv2d-6        [-1, 128, 112, 112]          73,856
              ReLU-7        [-1, 128, 112, 112]               0
            Conv2d-8        [-1, 128, 112, 112]         147,584
              ReLU-9        [-1, 128, 112, 112]               0
        MaxPool2d-10          [-1, 128, 56, 56]               0
           Conv2d-11          [-1, 256, 56, 56]         295,168
             ReLU-12          [-1, 256, 56, 56]               0
           Conv2d-13          [-1, 256, 56, 56]         590,080
             ReLU-14          [-1, 256, 56, 56]               0
           Conv2d-15          [-1, 256, 56, 56]         590,080
             ReLU-16          [-1, 256, 56, 56]               0
        MaxPool2d-17          [-1, 256, 28, 28]               0
           Conv2d-18          [-1, 512, 28, 28]       1,180,160
             ReLU-19          [-1, 512, 28, 28]               0
           Conv2d-20          [-1, 512, 28, 28]       2,359,808
             ReLU-21          [-1, 512, 28, 28]               0
           Conv2d-22          [-1, 512, 28, 28]       2,359,808
             ReLU-23          [-1, 512, 28, 28]               0
        MaxPool2d-24          [-1, 512, 14, 14]               0
           Conv2d-25          [-1, 512, 14, 14]       2,359,808
             ReLU-26          [-1, 512, 14, 14]               0
           Conv2d-27          [-1, 512, 14, 14]       2,359,808
             ReLU-28          [-1, 512, 14, 14]               0
           Conv2d-29          [-1, 512, 14, 14]       2,359,808
             ReLU-30          [-1, 512, 14, 14]               0
        MaxPool2d-31            [-1, 512, 7, 7]               0
AdaptiveAvgPool2d-32            [-1, 512, 1, 1]               0
          Flatten-33                  [-1, 512]               0
           Linear-34                  [-1, 128]          65,664
             ReLU-35                  [-1, 128]               0
          Dropout-36                  [-1, 128]               0
           Linear-37                    [-1, 1]             129
          Sigmoid-38                    [-1, 1]               0
================================================================
Total params: 14,780,481
Trainable params: 65,793
Non-trainable params: 14,714,688
```

> **TIP:** Note that the number of trainable parameters is only 65,793 out of a total of 14.7 million, as we have frozen the `features` module and have overwritten the `avgpool` and `classifier` modules. Now, only the `classifier` module will have weights that will be learned.

6. Define a function to train on a batch, calculate accuracy, and to get data just like we did in `Chapter 4`, *Introducing Convolutional Neural Networks*:

   - Train on a batch of data:

```
def train_batch(x, y, model, opt, loss_fn):
    model.train()
    prediction = model(x)
    batch_loss = loss_fn(prediction, y)
    batch_loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    return batch_loss.item()
```

   - Define a function to calculate accuracy on a batch of data:

```
@torch.no_grad()
def accuracy(x, y, model):
    model.eval()
    prediction = model(x)
    is_correct = (prediction > 0.5) == y
    return is_correct.cpu().numpy().tolist()
```

   - Define a function to fetch the data loaders:

```
def get_data():
    train = CatsDogs(train_data_dir)
    trn_dl = DataLoader(train, batch_size=32, shuffle=True, \
                              drop_last = True)
    val = CatsDogs(test_data_dir)
    val_dl = DataLoader(val, batch_size=32, shuffle=True, \
                              drop_last = True)
    return trn_dl, val_dl
```

   - Initialize the `get_data` and `get_model` functions:

```
trn_dl, val_dl = get_data()
model, loss_fn, optimizer = get_model()
```

7. Train the model over increasing epochs, just like we did in `Chapter 4`, *Introducing Convolutional Neural Networks*:

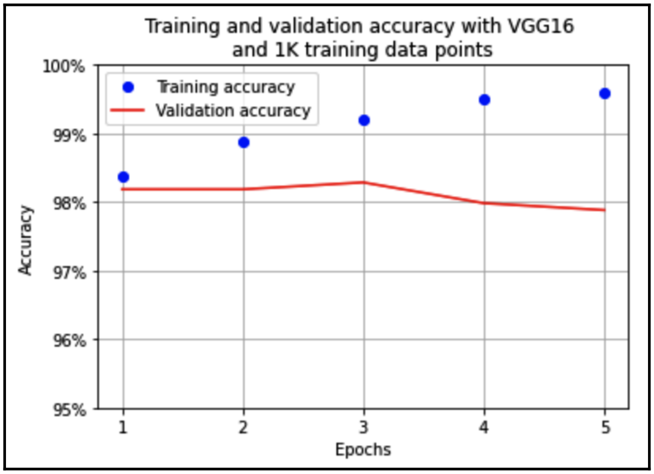
```
train_losses, train_accuracies = [], []
val_accuracies = []
for epoch in range(5):
    print(f" epoch {epoch + 1}/5")
    train_epoch_losses, train_epoch_accuracies = [], []
    val_epoch_accuracies = []

    for ix, batch in enumerate(iter(trn_dl)):
        x, y = batch
        batch_loss = train_batch(x, y, model, optimizer, \
                                 loss_fn)
        train_epoch_losses.append(batch_loss)
    train_epoch_loss = np.array(train_epoch_losses).mean()

    for ix, batch in enumerate(iter(trn_dl)):
        x, y = batch
        is_correct = accuracy(x, y, model)
        train_epoch_accuracies.extend(is_correct)
    train_epoch_accuracy = np.mean(train_epoch_accuracies)

    for ix, batch in enumerate(iter(val_dl)):
        x, y = batch
        val_is_correct = accuracy(x, y, model)
        val_epoch_accuracies.extend(val_is_correct)
    val_epoch_accuracy = np.mean(val_epoch_accuracies)

    train_losses.append(train_epoch_loss)
    train_accuracies.append(train_epoch_accuracy)
    val_accuracies.append(val_epoch_accuracy)
```

8. Plot the training and test accuracy values over increasing epochs:

```
epochs = np.arange(5)+1
import matplotlib.ticker as mtick
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
%matplotlib inline
plt.plot(epochs, train_accuracies, 'bo',
         label='Training accuracy')
plt.plot(epochs, val_accuracies, 'r',
         label='Validation accuracy')
plt.gca().xaxis.set_major_locator(mticker.MultipleLocator(1))
plt.title('Training and validation accuracy \
with VGG16 \nand 1K training data points')
plt.xlabel('Epochs')
```

```
plt.ylabel('Accuracy')
plt.ylim(0.95,1)
plt.gca().set_yticklabels(['{:.0f}%'.format(x*100) \
                           for x in plt.gca().get_yticks()])
plt.legend()
plt.grid('off')
plt.show()
```

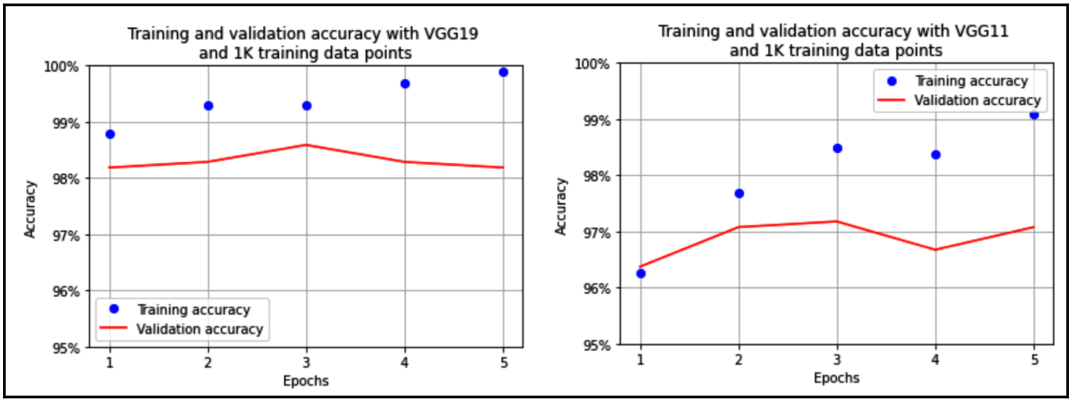
This results in the following output:



Note that we are able to get an accuracy of 98% within the first epoch, even on a small dataset of 1,000 images (500 images of each class).

In addition to VGG16, there are VGG11 and VGG19 pre-trained architectures that work just like VGG16, but with a different number of layers. VGG19 would have more parameters than that of VGG16 as it has a higher number of layers

The training and validation accuracy when we use VGG11 and VGG19 in place of the VGG16 pre-trained model is as follows:



Note that, while the VGG19-based model has slightly better accuracy than that of a VGG16-based model with an accuracy of 98% on validation data, the VGG11-based model has a slightly lower accuracy of 97%.

From VGG16 to VGG19, we have increased the number of layers, and generally, the deeper the neural network, the better its accuracy.

However, if merely increasing the number of layers is the trick, then we could keep on adding more layers (while taking care to avoid overfitting) to the model to get more accurate results on ImageNet and then fine-tune it for a dataset of interest. Unfortunately, that does not turn out to be true.

There are multiple reasons why it is not that easy. Any of the following are likely to happen as we go deeper in terms of architecture:

- We have to learn a larger number of features.
- Vanishing gradients arise.
- There is too much information modification at deeper layers.

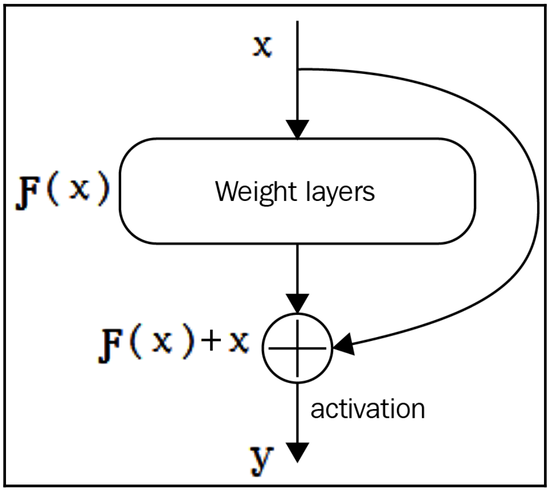
ResNet comes into the picture to address this specific scenario of identifying when not to learn, which we will learn about in the next section.

Understanding ResNet architecture

While building too deep a network, there are two problems. In forward propagation, the last few layers of the network have almost no information about what the original image was. In backpropagation, the first few layers near the input hardly get any gradient updates due to vanishing gradients (in other words, they are almost zero). To solve both problems, residual networks (ResNet) use a highway-like connection that transfers raw information from the previous few layers to the later layers. In theory, even the last layer will have the entire information of the original image due to this highway network. And because of the skipping layers, the backward gradients will flow freely to the initial layers with little modification.

The term **residual** in the residual network is the additional information that the model is expected to learn from the previous layer that needs to be passed on to the next layer.

A typical residual block appears as follows:



As you can see, while so far, we have been interested in extracting the F(x) value, where x is the value coming from the previous layer, in the case of a residual network, we are extracting not only the value after passing through the weight layers, which is F(x), but are also summing up F(x) with the original value, which is x.

So far, we have been using standard layers that performed either linear or convolution transformations `F(x)` along with some non-linear activation. Both of these operations in some sense destroy the input information. For the first time, we are seeing a layer that not only transforms the input, but also preserves it, by adding the input directly to the transformation – `F(x) + x`. This way, in certain scenarios, the layer has very little burden in remembering what the input is, and can focus on learning the correct transformation for the task.

Let's have a more detailed look at the residual layer through code by building a residual block:

> The code for this section is available as `Implementing_ResNet18_for_image_classification.ipynb` in the `Chapter05` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Define a class with the convolution operation (weight layer in the previous diagram) in the `__init__` method:

```
class ResLayer(nn.Module):
     def __init__(self,ni,no,kernel_size,stride=1):
        super(ResLayer, self).__init__()
        padding = kernel_size - 2
        self.conv = nn.Sequential(
                        nn.Conv2d(ni, no, kernel_size, stride,
                              padding=padding),
                        nn.ReLU()
                  )
```

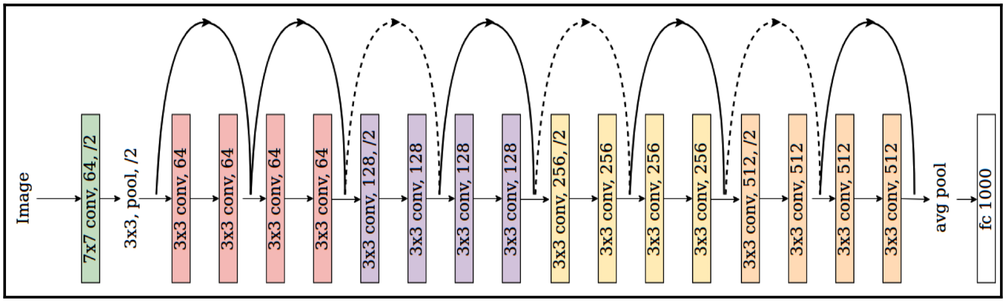
   Note that, in the preceding code, we defined `padding` as the dimension of the output when passed through convolution, and the dimension of the input should remain the same if we were to sum the two.

2. Define the `forward` method:

```
     def forward(self, x):
         x = self.conv(x) + x
         return x
```

   In the preceding code, we are getting an output that is a sum of the input passed through the convolution operations and the original input.

Now that we have learned about how residual blocks work, let's understand how the residual blocks are connected in a pre-trained, residual block-based network, ResNet18:



As you can see, there are 18 layers in the architecture, hence it is referred to as a ResNet18 architecture. Furthermore, notice how the skip connections are made across the network. It is not made at every convolution layer, but after every two layers instead.

Now that we understand the composition of a ResNet architecture, let's build a model based on ResNet18 architecture to classify between dogs and cats, just like we did in the previous section using VGG16.

To build a classifier, the code up to *step 3* of the VGG16 section remains the same as it deals with importing packages, fetching data, and inspecting them. So, we will start by understanding the composition of a pre-trained ResNet18 model:

> The code for this section is available as `Resnet_block_architecture.ipynb` in the `Chapter05` folder of the GitHub repository. Given that a majority of the code is similar to the code in the VGG section, we have only provided the additional code for brevity. For the full code, you are encouraged to check the notebook in GitHub.

1. Load the pre-trained ResNet18 model and inspect the modules within the loaded model:

```
model = models.resnet18(pretrained=True).to(device)
model
```

The structure of the ResNet18 model contains the following components:

- Convolution
- Batch normalization
- ReLU
- MaxPooling
- Four layers of ResNet blocks
- Average pooling (avgpool)
- A fully connected layer (fc)

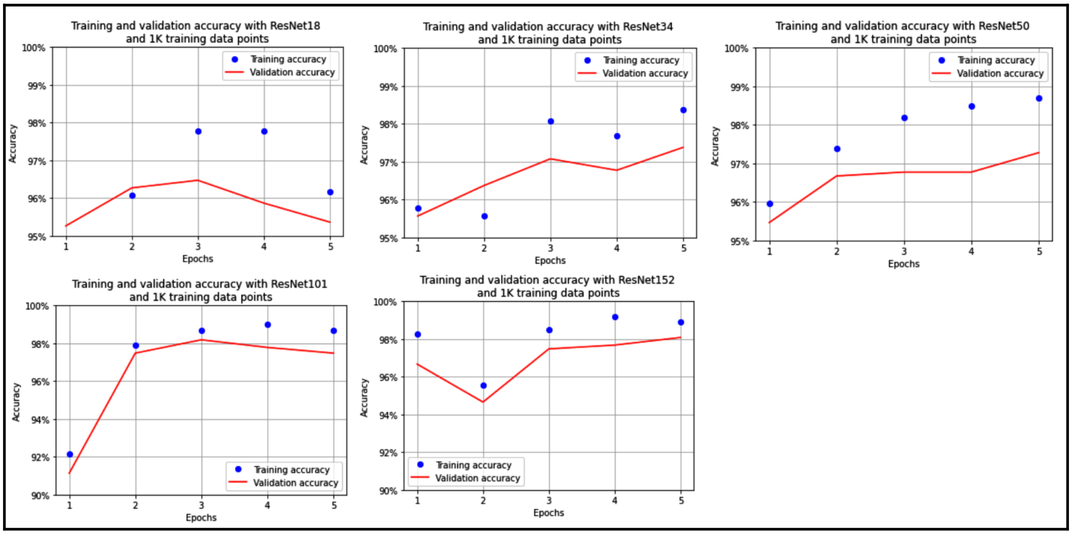
As we have done in VGG16, we will freeze all the different modules, but update the parameters in the `avgpool` and `fc` modules in the next step.

2. Define the model architecture, loss function, and optimizer:

```
def get_model():
    model = models.resnet18(pretrained=True)
    for param in model.parameters():
        param.requires_grad = False
    model.avgpool = nn.AdaptiveAvgPool2d(output_size=(1,1))
    model.fc = nn.Sequential(nn.Flatten(),
    nn.Linear(512, 128),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(128, 1),
    nn.Sigmoid())
    loss_fn = nn.BCELoss()
    optimizer = torch.optim.Adam(model.parameters(),lr= 1e-3)
    return model.to(device), loss_fn, optimizer
```

In the preceding model, the input shape of the `fc` module is 512, as the output of `avgpool` has the shape of batch size x 512 x 1 x 1.

Now that we have defined the model, let's execute *steps 5* and *6* as per the VGG section. The variation in training and validation accuracies after training the model (where the model is ResNet18, ResNet34, ResNet50, ResNet101, and ResNet152 for each of the following charts) over increasing epochs is as follows:



We see that the accuracy of the model, when trained on only 1,000 images, varies between 97% and 98%, where accuracy increases with an increase in the number of layers in ResNet.

Besides VGG and ResNet, some of the other prominent pre-trained models are Inception, MobileNet, DenseNet, and SqueezeNet.

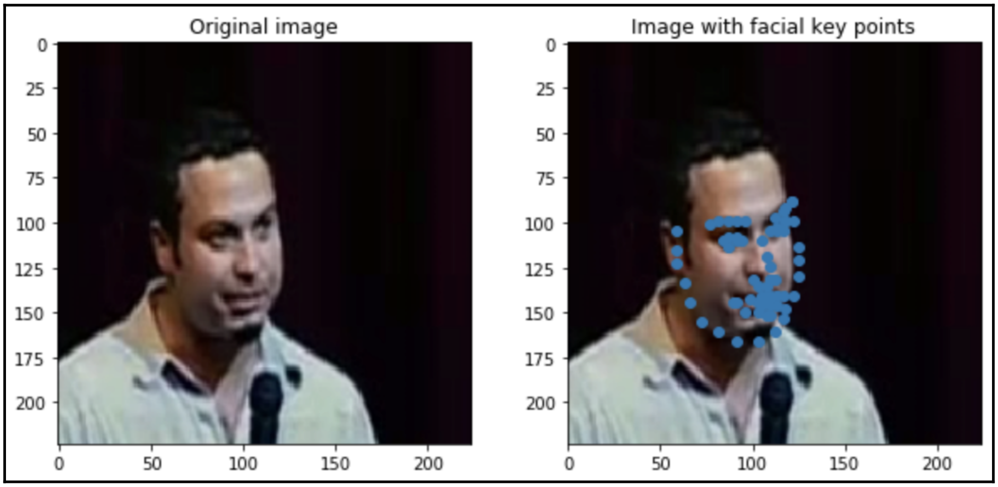
Now that we have learned about leveraging pre-trained models to predict for a class that is binary, in the next sections, we will learn about leveraging pre-trained models to solve real-world use cases that involve the following:

- **Multi-regression**: Prediction of multiple values given an image as input – facial key point detection
- **Multi-task learning**: Prediction of multiple items in a single shot – age estimation and gender classification

Implementing facial key point detection

So far, we have learned about predicting classes that are binary (cats versus dogs) or are multi-label (fashionMNIST). Let's now learn a regression problem and, in so doing, a task where we are predicting not one but several continuous outputs. Imagine a scenario where you are asked to predict the key points present on an image of a face, for example, the location of the eyes, nose, and chin. In this scenario, we need to employ a new strategy to build a model to detect the key points.

Before we dive further, let's understand what we are trying to achieve through the following image:



As you can observe in the preceding image, facial key points denote the markings of various key points on the image that contains a face.

To solve this problem, we would have to solve a few problems first:

- Images can be of different shapes:
  - This warrants an adjustment in the key point locations while adjusting images to bring them all to a standard image size.
- Facial key points are similar to points on a scatter plot, but scattered based on a certain pattern this time:
  - This means that the values are anywhere between 0 and 224 if the image is resized to a shape of 224 x 224 x 3.
- Normalize the dependent variable (the location of facial key points) as per the size of the image:

    - The key point values are always between 0 and 1 if we consider their location relative to image dimensions.
- Given that the dependent variable values are always between o and 1, we can use a sigmoid layer at the end to fetch values that will be between 0 and 1.

Let's formulate the pipeline of solving this use case:

1. Import the relevant packages.
2. Import data.
3. Define the class that prepares the dataset:
    - Ensure appropriate pre-processing is done on input images to perform transfer learning.
    - Ensure that the location of key points is processed in such a way that we fetch their relative position with respect to the processed image.
4. Define the model, loss function, and optimizer:
    - The loss function is the mean absolute error, as the output is a continuous value between 0 and 1.
5. Train the model over increasing epochs.

Let's now implement the preceding steps:

> The code for this section is available as `Facial_keypoints_detection.ipynb` in the `Chapter05` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` Be sure to copy the URL from the notebook in GitHub to avoid any issue while reproducing the results

1. Import the relevant packages and the dataset:

```
import torchvision
import torch.nn as nn
import torch
import torch.nn.functional as F
from torchvision import transforms, models, datasets
from torchsummary import summary
import numpy as np, pandas as pd, os, glob, cv2
from torch.utils.data import TensorDataset,DataLoader,Dataset
from copy import deepcopy
from mpl_toolkits.mplot3d import Axes3D
```

```
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn import cluster
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

2. Download and import the relevant data. You can download the relevant data that contains images and their corresponding facial key points:

```
!git clone https://github.com/udacity/P1_Facial_Keypoints.git
!cd P1_Facial_Keypoints
root_dir = 'P1_Facial_Keypoints/data/training/'
all_img_paths = glob.glob(os.path.join(root_dir, '*.jpg'))
data = pd.read_csv(\
    'P1_Facial_Keypoints/data/training_frames_keypoints.csv')
```

A sample of the imported dataset is as follows:

	Unnamed: 0	0	1	2	3	4	5	6
0	Luis_Fonsi_21.jpg	45.0	98.0	47.0	106.0	49.0	110.0	53.0
1	Lincoln_Chafee_52.jpg	41.0	83.0	43.0	91.0	45.0	100.0	47.0
2	Valerie_Harper_30.jpg	56.0	69.0	56.0	77.0	56.0	86.0	56.0
3	Angelo_Reyes_22.jpg	61.0	80.0	58.0	95.0	58.0	108.0	58.0
4	Kristen_Breitweiser_11.jpg	58.0	94.0	58.0	104.0	60.0	113.0	62.0

5 rows × 137 columns

In the preceding output, column 1 represents the name of the image, even columns represent the *x*-axis value corresponding to each of the 68 key points of the face, and the rest of the odd columns (except the first column) represent the *y*-axis value corresponding to each of the 68 key points.

3. Define the `FacesData` class that provides input and output data points for the data loader:

```
class FacesData(Dataset):
```

- Now let's define the `__init__` method, which takes the data frame of the file (`df`) as input:

```
    def __init__(self, df):
        super(FacesData).__init__()
        self.df = df
```

- Define the mean and standard deviation with which images are to be pre-processed so that they can be consumed by the pre-trained VGG16 model:

```
        self.normalize = transforms.Normalize(
                                mean=[0.485, 0.456, 0.406],
                                std=[0.229, 0.224, 0.225])
```

- Now, define the `__len__` method:

```
    def __len__(self): return len(self.df)
```

Next, we define the `__getitem__` method, where we fetch the image corresponding to a given index, scale it, fetch the key point values corresponding to the given index, normalize the key points so that we have the location of the key points as a proportion of the size of the image, and pre-process the image.

- Define the `__getitem__` method and fetch the path of the image corresponding to a given index (`ix`):

```
    def __getitem__(self, ix):
        img_path = 'P1_Facial_Keypoints/data/training/' + \
                                        self.df.iloc[ix,0]
```

- Scale the image:

```
        img = cv2.imread(img_path)/255.
```

- Normalize the expected output values (key points) as a proportion of the size of the original image:

```
        kp = deepcopy(self.df.iloc[ix,1:].tolist())
        kp_x = (np.array(kp[0::2])/img.shape[1]).tolist()
        kp_y = (np.array(kp[1::2])/img.shape[0]).tolist()
```

In the preceding code, we are ensuring that key points are provided as a proportion of the original image's size. This is done so that when we resize the original image, the location of the key points is not changed, as the key points are provided as a proportion of the original image. Furthermore, by getting key points as a proportion of the original image, we have expected output values that are between 0 and 1.

- Return the key points (`kp2`) and image (`img`) after pre-processing the image:

```
        kp2 = kp_x + kp_y
        kp2 = torch.tensor(kp2)
        img = self.preprocess_input(img)
        return img, kp2
```

- Define the function to pre-process an image (`preprocess_input`):

```
    def preprocess_input(self, img):
        img = cv2.resize(img, (224,224))
        img = torch.tensor(img).permute(2,0,1)
        img = self.normalize(img).float()
        return img.to(device)
```

- Define a function to load the image, which will be useful when we want to visualize a test image and the predicted key points of the test image:

```
    def load_img(self, ix):
        img_path = 'P1_Facial_Keypoints/data/training/' + \
                                        self.df.iloc[ix,0]
        img = cv2.imread(img_path)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)/255.
        img = cv2.resize(img, (224,224))
        return img
```

4. Let's now create a training and test data split and establish training and test datasets and data loaders:

```
from sklearn.model_selection import train_test_split

train, test = train_test_split(data, test_size=0.2, \
                                random_state=101)
train_dataset = FacesData(train.reset_index(drop=True))
test_dataset = FacesData(test.reset_index(drop=True))

train_loader = DataLoader(train_dataset, batch_size=32)
test_loader = DataLoader(test_dataset, batch_size=32)
```

In the preceding code, we have split the training and test datasets by person name in the input data frame and fetched their corresponding objects.

5. Let's now define the model that we will leverage to identify key points in an image:

    - Load the pre-trained VGG16 model:

```
def get_model():
    model = models.vgg16(pretrained=True)
```

    - Ensure that the parameters of the pre-trained model are frozen first:

```
    for param in model.parameters():
        param.requires_grad = False
```

    Overwrite and unfreeze the parameters of the last two layers of the model:

```
    model.avgpool = nn.Sequential( nn.Conv2d(512,512,3),
                                      nn.MaxPool2d(2),
                                      nn.Flatten())
    model.classifier = nn.Sequential(
                                      nn.Linear(2048, 512),
                                      nn.ReLU(),
                                      nn.Dropout(0.5),
                                      nn.Linear(512, 136),
                                      nn.Sigmoid()
                                    )
```

    Note that the last layer of the model in the `classifier` module is a sigmoid function that returns a value between o and 1 and that the expected output will always be between 0 and 1 as keypoint locations are a fraction of the original image's dimensions:

    - Define the loss function and optimizer and return them along with the model:

```
    criterion = nn.L1Loss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
    return model.to(device), criterion, optimizer
```

    Note that the loss function is `L1Loss`, in other words, we are performing mean absolute error reduction on the prediction of the location of facial key points (which will be predicted as a percentage of the image's width and height).

6. Get the model, loss function, and the corresponding optimizer:

```
model, criterion, optimizer = get_model()
```

7. Define functions to train on a batch of data points and also to validate on the test dataset:

   - Training a batch, as we have done earlier, involves fetching the output of passing input through the model, calculating the loss value, and performing backpropagation to update the weights:

```
def train_batch(img, kps, model, optimizer, criterion):
    model.train()
    optimizer.zero_grad()
    _kps = model(img.to(device))
    loss = criterion(_kps, kps.to(device))
    loss.backward()
    optimizer.step()
    return loss
```

   - Build a function that returns the loss on test data and the predicted key points:

```
def validate_batch(img, kps, model, criterion):
    model.eval()
    with torch.no_grad():
        _kps = model(img.to(device))
    loss = criterion(_kps, kps.to(device))
    return _kps, loss
```

8. Train the model based on training the data loader and test it on test data, as we have done hitherto in previous sections:

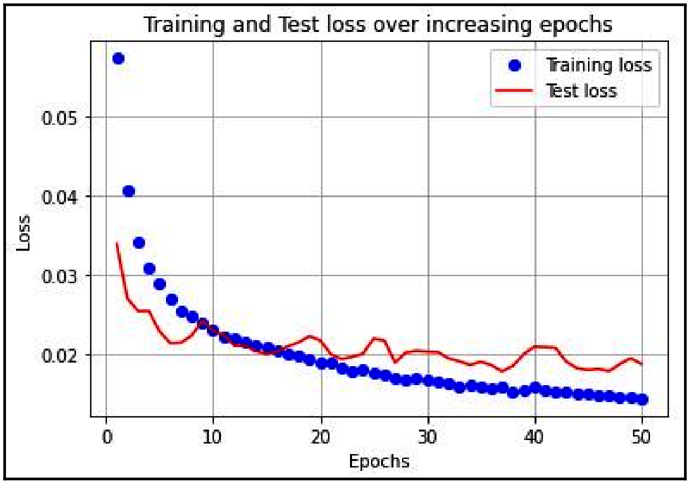
```
train_loss, test_loss = [], []
n_epochs = 50

for epoch in range(n_epochs):
    print(f" epoch {epoch+ 1} : 50")
    epoch_train_loss, epoch_test_loss = 0, 0
    for ix, (img,kps) in enumerate(train_loader):
        loss = train_batch(img, kps, model, optimizer, \
                                        criterion)
        epoch_train_loss += loss.item()
    epoch_train_loss /= (ix+1)

    for ix,(img,kps) in enumerate(test_loader):
        ps, loss = validate_batch(img, kps, model, criterion)
```

```
        epoch_test_loss += loss.item()
    epoch_test_loss /= (ix+1)

    train_loss.append(epoch_train_loss)
    test_loss.append(epoch_test_loss)
```

9. Plot the training and test loss over increasing epochs:

```
epochs = np.arange(50)+1
import matplotlib.ticker as mtick
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
%matplotlib inline
plt.plot(epochs, train_loss, 'bo', label='Training loss')
plt.plot(epochs, test_loss, 'r', label='Test loss')
plt.title('Training and Test loss over increasing epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid('off')
plt.show()
```

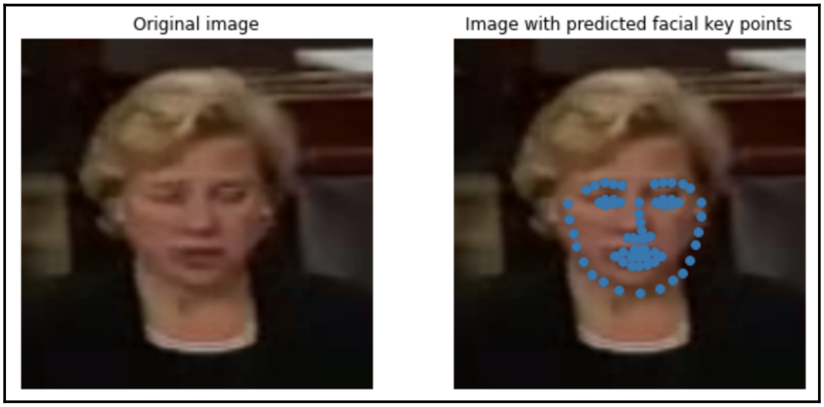
The preceding code results in the following output:



10. Test our model on a random test image's index, let's say 0. Note that in the following code, we are leveraging the `load_img` method in the `FacesData` class that was created earlier:

```
ix = 0
plt.figure(figsize=(10,10))
plt.subplot(221)
plt.title('Original image')
im = test_dataset.load_img(ix)
plt.imshow(im)
plt.grid(False)
plt.subplot(222)
plt.title('Image with facial keypoints')
x, _ = test_dataset[ix]
plt.imshow(im)
kp = model(x[None]).flatten().detach().cpu()
plt.scatter(kp[:68]*224, kp[68:]*224, c='r')
plt.grid(False)
plt.show()
```

The preceding code results in the following output:



From the preceding image, we see that the model is able to identify the facial key points fairly accurately, given the image as an input.

In this section, we have built the facial key point detector model from scratch. However, there are pre-trained models that are built both for 2D and 3D point detection. In the next section, we will learn about leveraging the face alignment library to fetch 2D and 3D key points of a face.

2D and 3D facial key point detection

In this section, we will leverage a pre-trained model that can detect the 2D and 3D key points present in a face in a few lines of code.

The following code is available as `2D_and_3D facial_keypoints.ipynb` in the `Chapter05` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` Be sure to copy the URL from the notebook in GitHub to avoid any issue while reproducing the results

To work on this, we will leverage the `face-alignment` library:

1. Install the required packages:

```
!pip install -qU face-alignment
import face_alignment, cv2
```

2. Import the image:

```
!wget https://www.dropbox.com/s/2s7xjto7rb6q7dc/Hema.JPG
```

3. Define the face alignment method, where we specify whether we want to fetch key point landmarks in 2D or 3D:

```
fa = face_alignment.FaceAlignment(\
                    face_alignment.LandmarksType._2D, \
                    flip_input=False, device='cpu')
```

4. Read the input image and provide it to the `get_landmarks` method:

```
input = cv2.imread('Hema.JPG')
preds = fa.get_landmarks(input)[0]
print(preds.shape)
# (68,2)
```

In the preceding lines of code, we are leveraging the `get_landmarks` method in the `fa` class to fetch the 68 *x* and *y* coordinates corresponding to the facial key points.

5. Plot the image with the detected key points:

```
import matplotlib.pyplot as plt
%matplotlib inline
fig,ax = plt.subplots(figsize=(5,5))
plt.imshow(cv2.cvtColor(cv2.imread('Hema.JPG'), \
                        cv2.COLOR_BGR2RGB))
ax.scatter(preds[:,0], preds[:,1], marker='+', c='r')
plt.show()
```

The preceding code results in the following output:

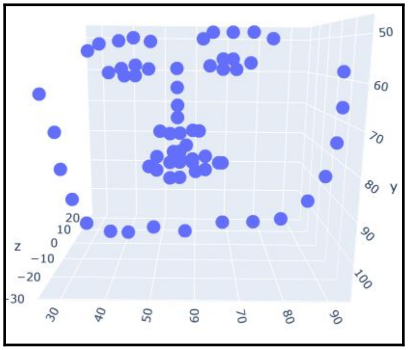
Notice the scatter plot of + symbols around the 60 possible facial key points.

In a similar manner, the 3D projections of facial key points are obtained as follows:

```
fa = face_alignment.FaceAlignment(
                        face_alignment.LandmarksType._3D,
                        flip_input=False, device='cpu')
input = cv2.imread('Hema.JPG')
preds = fa.get_landmarks(input)[0]
import pandas as pd
df = pd.DataFrame(preds)
df.columns = ['x','y','z']
import plotly.express as px
fig = px.scatter_3d(df, x = 'x', y = 'y', z = 'z')
fig.show()
```

Note that the only change from the code used in the 2D key points scenario is that we specified `LandmarksType` to be 3D in place of 2D

The preceding code results in the following output:



With the code leveraging the `face_alignment` library, we see that we are able to leverage the pre-trained facial key point detection models to have high accuracy in predicting on new images.

So far, across different use cases, we have learned the following:

- **Cats versus dogs**: Predicting for binary classification
- **FashionMNIST**: Predicting for a label among 10 possible classes
- **Facial key points**: Predicting multiple values between 0 and 1 for a given image

In the next section, we will learn about predicting a binary class and a regression value together in a single shot using a single network.

Multi-task learning – Implementing age estimation and gender classification

Multi-task learning is a branch of research where a single/few inputs are used to predict several different but ultimately connected outputs. For example, in a self-driving car, the model needs to identify obstacles, plan routes, give the right amount of throttle/brake and steering, to name but a few. It needs to do all of these in a split second by considering the same set of inputs (which would come from several sensors)

From the various use cases we have solved so far, we are in a position to train a neural network and estimate the age of a person given an image or predict the gender of the person given an image, separately, one task at a time. However, we have not looked at a scenario where we will be able to predict both age and gender in a single shot from an image. Predicting two different attributes in a single shot is important, as the same image is used for both predictions (this will be further appreciated as we perform object detection in `Chapter 7`, *Basics of Object Detection*).

In this section, we will learn about predicting both attributes, continuous and categorical predictions, in a single forward pass.

The strategy we adopt is as follows:

1. Import the relevant packages.
2. Fetch a dataset that contains images of persons, their gender, and age information.
3. Create training and test datasets by performing appropriate pre-processing.
4. Build a model where the following applies:
    - All the layers of the model remain similar to the models we have built so far, except for the last part.
    - In the last part, create two separate layers branching out from the preceding layer, where one layer corresponds to age estimation and the other to gender classification.
    - Ensure that you have different loss functions for each branch of output, as age is a continuous value (requiring an mse or mae loss calculation) and gender is a categorical value (requiring a cross-entropy loss calculation).
    - Take a weighted summation of age estimation loss and gender classification loss.
    - Minimize the overall loss by performing backpropagation that optimizes weight values.
5. Train model and predict on new images.

With the preceding strategy in place, let's code up the use case:

> The code for this section is available as `Age_and_gender_prediction.ipynb` in the `Chapter05` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from. We strongly recommend you to execute the notebook on GitHub.

1. Import the relevant packages:

```
import torch
import numpy as np, cv2, pandas as pd, glob, time
import matplotlib.pyplot as plt
%matplotlib inline
import torch.nn as nn
from torch import optim
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
import torchvision
from torchvision import transforms, models, datasets
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

2. Fetch the dataset:

```
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials

auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials=GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)

def getFile_from_drive( file_id, name ):
    downloaded = drive.CreateFile({'id': file_id})
    downloaded.GetContentFile(name)

getFile_from_drive('1Z1RqRo0_JiavaZw2yzZG6WETdZQ8qX86',
                   'fairface-img-margin025-trainval.zip')
getFile_from_drive('1k5vvyREmHDW5TSM9QgB04Bvc8C8_7dl-',
                   'fairface-label-train.csv')
getFile_from_drive('1_rtz1M1zhvS0d5vVoXUamnohB6cJ02iJ',
                   'fairface-label-val.csv')

!unzip -qq fairface-img-margin025-trainval.zip
```

3. The dataset we downloaded can be loaded and is structured in the following way:

```
trn_df = pd.read_csv('fairface-label-train.csv')
val_df = pd.read_csv('fairface-label-val.csv')
trn_df.head()
```

The preceding code results in the following output:

	file	age	gender	race	service_test
0	train/1.jpg	59	Male	East Asian	True
1	train/2.jpg	39	Female	Indian	False
2	train/3.jpg	11	Female	Black	False
3	train/4.jpg	26	Female	Indian	True
4	train/5.jpg	26	Female	Indian	True

4. Build the `GenderAgeClass` class that takes a filename as input and returns the corresponding image, gender, and scaled age. We scale age as it is a continuous number and, as we have seen in `Chapter 3`, *Building a Deep Neural Network with PyTorch*, it is better to scale data to avoid vanishing gradients and then rescale it during post-processing:

   - Provide file paths (`fpaths`) of images in the `__init__` method:

```
IMAGE_SIZE = 224
class GenderAgeClass(Dataset):
    def __init__(self, df, tfms=None):
        self.df = df
        self.normalize = transforms.Normalize(
                                mean=[0.485, 0.456, 0.406],
                                std=[0.229, 0.224, 0.225])
```

   - Define the `__len__` method as the one that returns the number of images in the input:

```
    def __len__(self): return len(self.df)
```

   - Define the `__getitem__` method that fetches information of an image at a given position, `ix`:

```
    def __getitem__(self, ix):
        f = self.df.iloc[ix].squeeze()
        file = f.file
        gen = f.gender == 'Female'
        age = f.age
        im = cv2.imread(file)
        im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
        return im, age, gen
```

- Write a function that pre-processes an image, which involves resizing the image, permuting the channels, and performing normalization on a scaled image:

```
def preprocess_image(self, im):
    im = cv2.resize(im, (IMAGE_SIZE, IMAGE_SIZE))
    im = torch.tensor(im).permute(2,0,1)
    im = self.normalize(im/255.)
    return im[None]
```

- Create the `collate_fn` method, which fetches a batch of data where the data points are pre-processed as follows:
  - Process each image using the `process_image` method.
  - Scale the age by 80 (the maximum age value present in the dataset), so that all values are between 0 and 1.
  - Convert gender to a float value.
  - Image, age, and gender are each converted into torch objects and returned:

```
def collate_fn(self, batch):
    'preprocess images, ages and genders'
    ims, ages, genders = [], [], []
    for im, age, gender in batch:
        im = self.preprocess_image(im)
        ims.append(im)

        ages.append(float(int(age)/80))
        genders.append(float(gender))

    ages, genders = [torch.tensor(x).to(device).float() \
                        for x in [ages, genders]]
    ims = torch.cat(ims).to(device)

    return ims, ages, genders
```

5. We now define the training and validation datasets and data loaders:

   - Create the datasets:

```
trn = GenderAgeClass(trn_df)
val = GenderAgeClass(val_df)
```

- Specify the data loaders:

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
train_loader = DataLoader(trn, batch_size=32, shuffle=True, \
                    drop_last=True,collate_fn=trn.collate_fn)
test_loader = DataLoader(val, batch_size=32,
                         collate_fn=val.collate_fn)
a,b,c, = next(iter(train_loader))
print(a.shape, b.shape, c.shape)
```

6. Define the model, loss function, and optimizer:

- First, in the function, we load the pre-trained VGG16 model:

```
def get_model():
    model = models.vgg16(pretrained = True)
```

- Next, freeze the loaded model (by specifying `param.requires_grad = False`):

```
    for param in model.parameters():
        param.requires_grad = False
```

- Overwrite the `avgpool` layer with our own layer:

```
    model.avgpool = nn.Sequential(
                        nn.Conv2d(512,512, kernel_size=3),
                        nn.MaxPool2d(2),
                        nn.ReLU(),
                        nn.Flatten()
                    )
```

Now comes the key part. We deviate from what we have learned so far by creating two branches of outputs. This is performed as follows:

- Build a neural network `class` named `ageGenderClassifier` with the following in the `__init__` method:

```
    class ageGenderClassifier(nn.Module):
        def __init__(self):
            super(ageGenderClassifier, self).__init__()
```

- Define the `intermediate` layer calculations:

```
            self.intermediate = nn.Sequential(
                                    nn.Linear(2048,512),
                                    nn.ReLU(),
                                    nn.Dropout(0.4),
```

```
                                    nn.Linear(512,128),
                                    nn.ReLU(),
                                    nn.Dropout(0.4),
                                    nn.Linear(128,64),
                                    nn.ReLU(),
                                )
```

- Define`age_classifier` and `gender_classifier`:

```
        self.age_classifier = nn.Sequential(
                                        nn.Linear(64, 1),
                                        nn.Sigmoid()
                                    )
        self.gender_classifier = nn.Sequential(
                                        nn.Linear(64, 1),
                                        nn.Sigmoid()
                                    )
```

Note that, in the preceding code, the last layers have a sigmoid activation since the age output will be a value between 0 and 1 (as it is scaled by 80) and gender has a sigmoid as the output is either a *0* or a *1*.

- Define the `forward` pass method that stacks layers as `intermediate` first, followed by `age_classifier` and then `gender_classifier`:

```
    def forward(self, x):
        x = self.intermediate(x)
        age = self.age_classifier(x)
        gender = self.gender_classifier(x)
        return gender, age
```

- Overwrite the `classifier` module with the class we defined previously:

```
model.classifier = ageGenderClassifier()
```

- Define the loss functions of both the gender (binary cross-entropy loss) and age (L1 loss) predictions. Define the optimizer and return the model, loss functions, and optimizer, as follows:

```
gender_criterion = nn.BCELoss()
age_criterion = nn.L1Loss()
loss_functions = gender_criterion, age_criterion
optimizer = torch.optim.Adam(model.parameters(),lr= 1e-4)
return model.to(device), loss_functions, optimizer
```

- Call the `get_model` function to initialize values in the variables:

```
model, criterion, optimizer = get_model()
```

7. Define the function to train on a batch of data and validate on a batch of the dataset.

The `train_batch` method takes an image, actual values of gender, age, model, optimizer, and loss function, as input to calculate total loss, as follows:

- Define the `train_batch` method with the input arguments in place:

```
def train_batch(data, model, optimizer, criteria):
```

- Specify that we are training the model, reset the optimizer to `zero_grad`, and calculate the predicted value of age and gender:

```
    model.train()
    ims, age, gender = data
    optimizer.zero_grad()
    pred_gender, pred_age = model(ims)
```

- Fetch the loss functions for both age and gender before calculating the loss corresponding to age estimation and gender classification:

```
    gender_criterion, age_criterion = criteria
    gender_loss = gender_criterion(pred_gender.squeeze(), \
                                    gender)
    age_loss = age_criterion(pred_age.squeeze(), age)
```

- Calculate the overall loss by summing up `gender_loss` and `age_loss` and perform backpropagation to reduce the overall loss by optimizing the trainable weights of the model and return the overall loss:

```
    total_loss = gender_loss + age_loss
    total_loss.backward()
    optimizer.step()
    return total_loss
```

The `validate_batch` method takes the image, model, and loss functions, as well as the actual values of age and gender, as input to calculate the predicted values of age and gender along with the loss values, as follows:

- Define the `vaidate_batch` function with proper input parameters:

```
def validate_batch(data, model, criteria):
```

- Specify that we want to evaluate the model, and so no gradient calculations are required before predicting the age and gender values by passing the image through the model:

```
    model.eval()
    with torch.no_grad():
        pred_gender, pred_age = model(img)
```

- Calculate the loss values corresponding to age and gender predictions (`gender_loss` and `age_loss`). We squeeze the predictions (which have a shape of (batch size, 1) so that it is reshaped to the same shape of the original values (which has a shape of batch size):

```
    gender_criterion, age_criterion = criteria
    gender_loss = gender_criterion(pred_gender.squeeze(), \
                                        gender)
    age_loss = age_criterion(pred_age.squeeze(), age)
```

- Calculate the overall loss, final predicted gender class (`pred_gender`), and return the predicted gender, age, and total loss:

```
    total_loss = gender_loss + age_loss
    pred_gender = (pred_gender > 0.5).squeeze()
    gender_acc = (pred_gender == gender).float().sum()
    age_mae = torch.abs(age - pred_age).float().sum()
    return total_loss, gender_acc, age_mae
```

8. Train the model over five epochs:

- Define placeholders to store the train and test loss values and also to specify the number of epochs:

```
import time
model, criteria, optimizer = get_model()
val_gender_accuracies = []
val_age_maes = []
train_losses = []
```

```
val_losses = []

n_epochs = 5
best_test_loss = 1000
start = time.time()
```

- Loop through different epochs and reinitialize the train and test loss values at the start of each epoch:

```
for epoch in range(n_epochs):
    epoch_train_loss, epoch_test_loss = 0, 0
    val_age_mae, val_gender_acc, ctr = 0, 0, 0
    _n = len(train_loader)
```

- Loop through the training data loader (`train_loader`) and train the model:

```
    for ix, data in enumerate(train_loader):
        loss = train_batch(data, model, optimizer, criteria)
        epoch_train_loss += loss.item()
```

- Loop through the test data loader and calculate gender accuracy as well as the mae of age:

```
    for ix, data in enumerate(test_loader):
        loss, gender_acc, age_mae = validate_batch(data, \
                                            model, criteria)
        epoch_test_loss += loss.item()
        val_age_mae += age_mae
        val_gender_acc += gender_acc
        ctr += len(data[0])
```

- Calculate the overall accuracy of age prediction and gender classification:

```
    val_age_mae /= ctr
    val_gender_acc /= ctr
    epoch_train_loss /= len(train_loader)
    epoch_test_loss /= len(test_loader)
```

- Log the metrics for each epoch:

```
    elapsed = time.time()-start
    best_test_loss = min(best_test_loss, epoch_test_loss)
    print('{}/{} ({:.2f}s - {:.2f}s remaining)'.format(\
                    epoch+1, n_epochs, time.time()-start, \
                    (n_epochs-epoch)*(elapsed/(epoch+1))))
    info = f'''Epoch: {epoch+1:03d}
```

```
                    \tTrain Loss: {epoch_train_loss:.3f}
                    \tTest:\{epoch_test_loss:.3f}
                    \tBest Test Loss: {best_test_loss:.4f}'''
info += f'\nGender Accuracy:
                    {val_gender_acc*100:.2f}%\tAge MAE: \
                                        {val_age_mae:.2f}\n'
print(info)
```

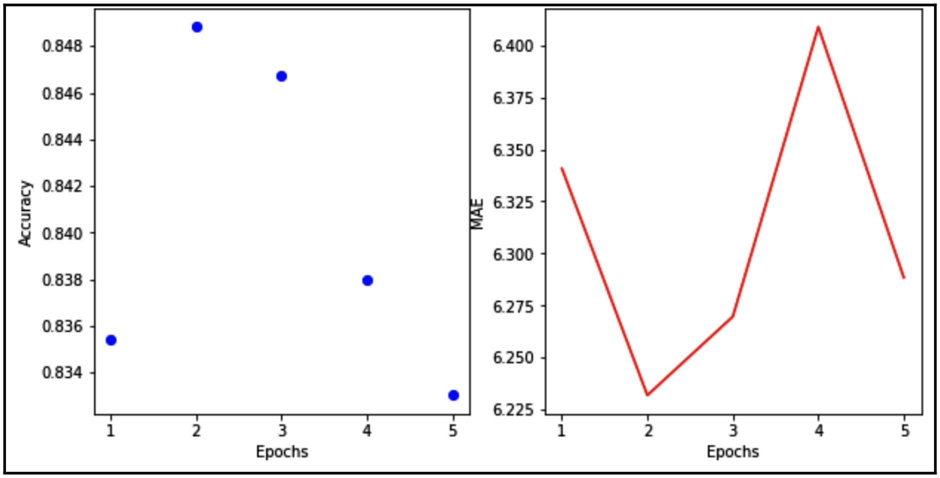
- Store the age and gender accuracy of the test dataset in each epoch:

```
val_gender_accuracies.append(val_gender_acc)
val_age_maes.append(val_age_mae)
```

9. Plot the accuracy of age estimation and gender prediction over increasing epochs:

```
epochs = np.arange(1,(n_epochs+1))
fig,ax = plt.subplots(1,2,figsize=(10,5))
ax = ax.flat
ax[0].plot(epochs, val_gender_accuracies, 'bo')
ax[1].plot(epochs, val_age_maes, 'r')
ax[0].set_xlabel('Epochs')  ; ax[1].set_xlabel('Epochs')
ax[0].set_ylabel('Accuracy'); ax[1].set_ylabel('MAE')
ax[0].set_title('Validation Gender Accuracy')
ax[0].set_title('Validation Age Mean-Absolute-Error')
plt.show()
```

The preceding code results in the following output:



We are off by 6 years in terms of age prediction and are approximately 84% accurate in predicting the gender.

10. Make a prediction of age and gender on a random test image:

    - Fetch an image:

```
!wget https://www.dropbox.com/s/6kzr8l68e9kpjkf/5_9.JPG
```

    - Load the image and pass it through the `preprocess_image` method in the `trn` object that we created earlier:

```
im = cv2.imread('/content/5_9.JPG')
im = trn.preprocess_image(im).to(device)
```

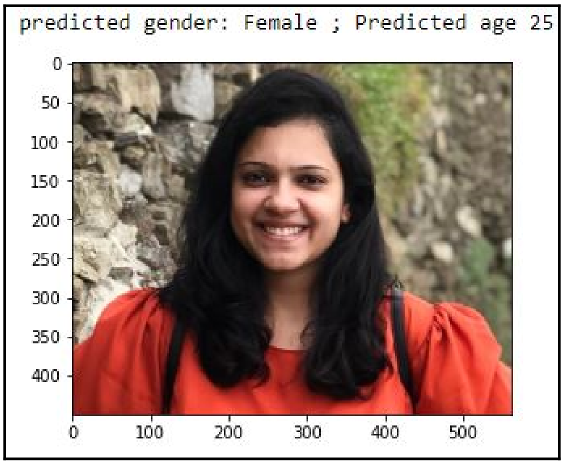
    - Pass the image through the trained model:

```
gender, age = model(im)
pred_gender = gender.to('cpu').detach().numpy()
pred_age = age.to('cpu').detach().numpy()
```

    - Plot the image along with printing the original and predicted values:

```
im = cv2.imread('/content/5_9.JPG')
im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
plt.imshow(im)
print('predicted gender:',np.where(pred_gender[0][0]<0.5, \
                                   'Male','Female'),
      '; Predicted age', int(pred_age[0][0]*80))
```

The preceding code results in the following output:



With the above, we can see that we are able to make predictions for both age and gender in a single shot. However, we need to note that this is highly unstable and that the age value varies considerably with different orientations of the image and also lighting conditions. Data augmentation comes in handy in such a scenario.

So far, we have learned about transfer learning, pre-trained architectures, and how to leverage them in two different use cases. You would have also noticed that the code is slightly on the lengthier side where we import extensive packages manually, create empty lists to log metrics, and constantly read/show images for debugging purposes. In the next section, we will learn about a library that the authors have built to avoid such verbose code.

Introducing the torch_snippets library

As you may have noticed, we are using the same functions in almost all the sections. It is a waste of our time to write the same lines of functions again and again. For convenience, the authors of this book have written a Python library by the name of `torch_snippets` so that our code looks short and clean.

Utilities such as reading an image, showing an image, and the entire training loop are quite repetitive. We want to circumvent writing the same functions over and over by wrapping them in code that is preferably a single function call. For example, to read a color image, we need not write `cv2.imread(...)` followed by `cv2.cvtColor(...)` every time. Instead, we can simply call `read(...)`. Similarly, for `plt.imshow(...)`, there are numerous hassles, including the fact that the size of the image should be optimal, and that the channel dimension should be last (remember PyTorch has them first). These will always be taken care of by the single function, `show`. Similar to `read` and `show`, there are over 20 convenience functions and classes that we will be using throughout the book. We will use `torch_snippets` from now on so as to focus more on actual deep learning without distractions. Let's dive a little and understand the salient functions by training `age-and-gender` with this library instead so that we can learn to use these functions and derive the maximum benefit.

> The code for this section is available as `age_gender_torch_snipets.ipynb` in the `Chapter05` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` . To maintain brevity, we have only provided the additional code in this section. For the full code, we encourage you to refer to the notebook in GitHub.

1. Install and load the library:

```
!pip install torch_snippets
from torch_snippets import *
```

   Right out of the gate, the library allows us to load all the important torch modules and utilities such as NumPy, pandas, Matplotlib, Glob, Os, and more.

2. Download the data and create a dataset as in the previous section. Create a dataset class, `GenderAgeClass`, with a few changes, which are shown in bold in the following code:

```
class GenderAgeClass(Dataset):
    ...
    def __getitem__(self, ix):
        ...
        age = f.age
        im = read(file, 1)
        return im, age, gen

    def preprocess_image(self, im):
        im = resize(im, IMAGE_SIZE)
        im = torch.tensor(im).permute(2,0,1)
        ...
```

   In the preceding code block, the line `im = read(file, 1)` is wrapping `cv2.imread` and `cv2.COLOR_BGR2RGB` into a single function call. The "1" stands for "read as color image", and if not given, will load a black and white image by default. There is also a `resize` function that wraps `cv2.resize`. Next, let's look at the `show` function.

3. Specify the training and validation datasets and view the sample images:

```
trn = GenderAgeClass(trn_df)
val = GenderAgeClass(val_df)
train_loader = DataLoader(trn, batch_size=32, shuffle=True, \
                  drop_last=True, collate_fn=trn.collate_fn)
test_loader = DataLoader(val, batch_size=32, \
                         collate_fn=val.collate_fn)

im, gen, age = trn[0]
show(im, title=f'Gender: {gen}\nAge: {age}', sz=5)
```

As we are dealing with images throughout the book, it makes sense to wrap `import matplotlib.pyplot as plt` and `plt.imshow` into a function. Calling `show(<2D/3D-Tensor>)` will do exactly that. Unlike Matplotlib, it can plot torch arrays present on the GPU, irrespective of whether the image contains a channel as the first dimension or the last dimension. The keyword `title` will plot a title with the image, and the keyword `sz` (short for size) will plot a larger/smaller image based on the integer value passed (if not passed, `sz` will pick a sensible default based on image resolution). During object detection chapters, we will use the same function to show bounding boxes as well. Check out `help(show)` for more arguments. Let's create some datasets here and inspect the first batch of images along with their targets.

4. Create data loaders and inspect the tensors. Inspecting tensors for their data type, min, mean, max, and shape is such a common activity that it is wrapped as a function. It can accept any number of tensor inputs:

```
train_loader = DataLoader(trn, batch_size=32, shuffle=True, \
                drop_last=True, collate_fn=trn.collate_fn)
test_loader = DataLoader(val, batch_size=32, \
                         collate_fn=val.collate_fn)

ims, gens, ages = next(iter(train_loader))
inspect(ims, gens, ages)
```

The `inspect` output will look like this:

```
============================================================
Tensor Shape: torch.Size([32, 3, 224, 224]) Min: -2.118 Max:
2.640 Mean: 0.133 dtype: torch.float32
============================================================
Tensor Shape: torch.Size([32]) Min: 0.000 Max: 1.000 Mean:
0.594 dtype: torch.float32
============================================================
Tensor Shape: torch.Size([32]) Min: 0.087 Max: 0.925 Mean:
0.400 dtype: torch.float32
============================================================
```

5. Create `model`, `optimizer`, `loss_functions`, `train_batch`, and `validate_batch` as usual. As each deep learning experiment is unique, there aren't any wrapper functions for this step.

> In this section, we will leverage the `get_model`, `train_batch`, and `validate_batch` functions that we defined in the previous section. For brevity, we are not providing the code in this section. However, all the relevant code is available in the corresponding notebook in GitHub.

6. Finally, we need to load all the components and start training. Log the metrics over increasing epochs.

This is a highly repetitive loop with minimal changes required. We will always loop over a fixed number of epochs, first over the train data loader, and then over the validation data loader. Each batch is called using either `train_batch` or `validate_batch`, every time that you have to create empty lists of metrics and keep track of them after training/validation. At the end of an epoch, you have to print the averages of all of these metrics and repeat the task. It is also helpful that you know how long (in seconds) each epoch /batch is going to train for. Finally, at the end of the training, it is common to plot the same metrics using `matplotlib`. All of these are wrapped into a single utility called `Report`. It is a Python class that has different methods to understand. The bold parts in the following code highlight the functionality of `Report`:

```
model, criterion, optimizer = get_model()
n_epochs = 5
log = Report(n_epochs)
for epoch in range(n_epochs):
    N = len(train_loader)
    for ix, data in enumerate(train_loader):
        total_loss,gender_loss,age_loss = train_batch(data, \
                                    model, optimizer, criterion)
        log.record(epoch+(ix+1)/N, trn_loss=total_loss, \
                                                   end='\r')

    N = len(test_loader)
    for ix, data in enumerate(test_loader):
        total_loss,gender_acc,age_mae = validate_batch(data, \
                                                model, criterion)
        gender_acc /= len(data[0])
        age_mae /= len(data[0])
        log.record(epoch+(ix+1)/N, val_loss=total_loss, \
```

```
                        val_gender_acc=gender_acc, \
                        val_age_mae=age_mae, end='\r')
        log.report_avgs(epoch+1)
    log.plot_epochs()
```

The `Report` class is instantiated with the only argument, the number of epochs to be trained on, and is instantiated just before the start of training.

At each train/validation step, we can call the `Report.record` method with exactly one positional argument, which is the position (in terms of batch number) of training/ validation we are at (typically, this is `( epoch_number + (1+batch number)/(total_N_batches) )`. Following the positional argument, we pass a bunch of keyword arguments that we are free to choose. If it's training loss that needs to be captured, the keyword argument could be `trn_loss`. In the preceding, we are logging four metrics, `trn_loss`, `val_loss`, `val_gender_acc`, and `val_age_mae`, without creating a single empty list.

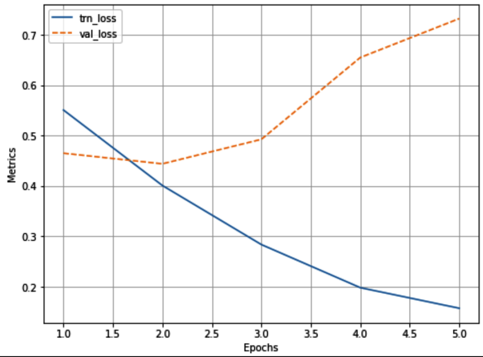
Not only does it record, but it will also print the same losses in the output. The use of `'\r'` as an end argument is a special way of saying replace this line the next time a new set of losses are to be recorded. Furthermore, `Report` will compute the time remaining for training and validation automatically and print that too.

`Report` will remember when the metric was logged and print all the average metrics at that epoch when the `Report.report_avgs` function is called. This will be a permanent print.

Finally, the same average metrics are plotted as a line chart in the function call `Report.plot_epochs`, without the need for formatting (you can also use `Report.plot` for plotting every batch metric of the entire training, but this might look messy). The same function can selectively plot metrics if asked for. By way of an example, in the preceding case, if you are interested in plotting only the `trn_loss` and `val_loss` metrics, this can be done by calling `log.plot_epochs(['trn_loss, 'val_loss'])` or even simply `log.plot_epochs('_loss')`. It will search for a string match with all the metrics and figure out what metrics we are asking for.

Once training is complete, the output for the preceding code snippet should look like this:

```
EPOCH: 1.000    trn_loss: 0.551 val_loss: 0.465 val_gender_acc: 0.834   val_age_mae: 6.238      (779.37s - 3117.47s remaining)
EPOCH: 2.000    trn_loss: 0.401 val_loss: 0.444 val_gender_acc: 0.847   val_age_mae: 6.229      (1555.70s - 2333.56s remaining)
EPOCH: 3.000    trn_loss: 0.284 val_loss: 0.493 val_gender_acc: 0.846   val_age_mae: 6.340      (2335.09s - 1556.73s remaining)
EPOCH: 4.000    trn_loss: 0.198 val_loss: 0.655 val_gender_acc: 0.842   val_age_mae: 6.339      (3110.83s - 777.71s remaining)
EPOCH: 4.994    val_loss: 0.375 val_gender_acc: 0.969   val_age_mae: 6.541      (3887.91s - 4.54s remaining)
  0%|          | 0/6 [00:00<?, ?it/s]/usr/local/lib/python3.6/dist-packages/numpy/core/fromnumeric.py:3335: RuntimeWarning: Mear
  out=out, **kwargs)
/usr/local/lib/python3.6/dist-packages/numpy/core/_methods.py:161: RuntimeWarning: invalid value encountered in double_scalars
  ret = ret.dtype.type(ret / rcount)
100%|██████████| 6/6 [00:00<00:00, 291.40it/s]EPOCH: 5.000      trn_loss: 0.157 val_loss: 0.733 val_gender_acc: 0.847   val_age_
```



Note that the output has corresponding training and validation dataset loss and accuracy values for age and gender values, even though we did not initialize any empty lists to log those metrics in the training and validation dataset (which we did in the previous sections)

7. Load a sample image and effect a prediction:

```
!wget -q https://www.dropbox.com/s/6kzr8l68e9kpjkf/5_9.JPG
IM = read('/content/5_9.JPG', 1)
im = trn.preprocess_image(IM).to(device)

gender, age = model(im)
pred_gender = gender.to('cpu').detach().numpy()
pred_age = age.to('cpu').detach().numpy()

info = f'predicted gender: {np.where(pred_gender[0][0]<0.5, \
"Male","Female")}\n Predicted age {int(pred_age[0][0]*80)}'
show(IM, title=info, sz=10)
```

To summarize, here are the important functions (and the functions they are wrapped around) that we will use in the rest of the book wherever needed:

- `from torch_snippets import *`
- Glob (`glob.glob`)
- Choose (`np.random.choice`)
- Read (`cv2.imread`)
- Show (`plt.imshow`)
- Subplots (`plt.subplots` – show a list of images)
- Inspect (`tensor.min`, `tensor.mean`, `tensor.max`, `tensor.shape`, and `tensor.dtype` – statistics of several tensors)
- Report (keeping track of all metrics while training and plotting them after training)

You can view the complete list of functions by running `torch_snippets; print(dir(torch_snippets))`. For each function, you can print its help using `help(function)` or even simply `??function` in a Jupyter notebook. With the understanding of leveraging `torch_snippets`, you should be able to simplify code considerably. You will notice this in action starting with the next chapter.

Summary

In this chapter, we have learned about how transfer learning helps to achieve high accuracy, even with a smaller number of data points. We have also learned about the popular pre-trained models, VGG and ResNet. Furthermore, we understood how to build models when we are trying to predict different scenarios, such as the location of key points on a face and combining loss values while training a model to predict for both age and gender together, where age is of a certain data type and gender is of a different data type.

With this foundation of image classification through transfer learning, in the next chapter, we will learn about some of the practical aspects of training an image classification model. We will learn about how to explain a model and also learn the tricks of how to train a model to achieve high accuracy and finally, learn the pitfalls that a practitioner needs to avoid while implementing a trained model.

Questions

1. What are VGG and ResNet pre-trained architectures trained on?
2. Why does VGG11 have an inferior accuracy to VGG16?
3. What does the number 11 in VGG11 represent?
4. What is *residual* in the residual network?
5. What is the advantage of a residual network?
6. What are the various popular pre-trained models?
7. During transfer learning, why should images be normalized with the same mean and standard deviation as those that were used during the training of a pre-trained model?
8. Why do we freeze certain parameters in a model?
9. How do we know the various modules that are present in a pre-trained model?
10. How do we train a model that predicts categorical and numerical values together?
11. Why might age and gender prediction code not always work for an image of your own interest if we execute the same code as we wrote in the age and gender estimation section?
12. How can we further improve the accuracy of the facial keypoint recognition model that we wrote about in the facial key points prediction section?

6
Practical Aspects of Image Classification

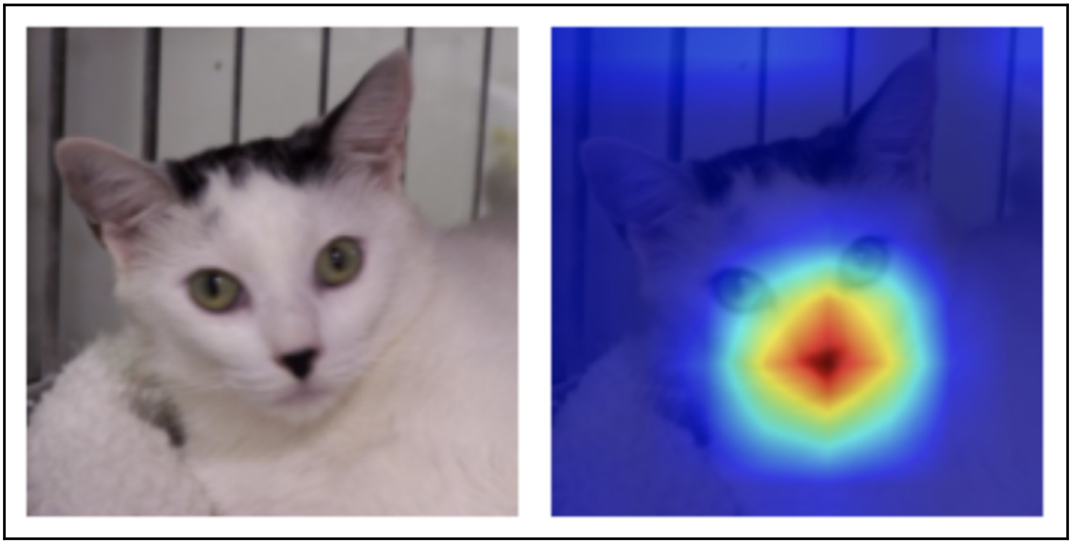
In previous chapters, we learned about leveraging **convolutional neural networks** (**CNNs**) along with pre-trained models to perform image classification. This chapter will further solidify our understanding of CNNs and the various practical aspects to be considered when leveraging them in real-world applications. We will start by understanding the reasons why CNNs predict the classes that they do by using **class activation maps** (**CAMs**). Following this, we will understand the various data augmentations that can be done to improve the accuracy of a model. Finally, we will learn about the various instances where models could go wrong in the real world and highlight the aspects that should be taken care of in such scenarios to avoid pitfalls.

The following topics will be covered in this chapter:

- Generating CAMs
- Understanding the impact of batch normalization and data augmentation
- Practical aspects to take care of during model implementation

Generating CAMs

Imagine a scenario where you have built a model that is able to make good predictions. However, the stakeholder that you are presenting the model to wants to understand the reason why the model predictions are as they are. CAMs come in handy in this scenario. An example CAM is as follows, where we have the input image on the left and the pixels that were used to come up with the class prediction highlighted on the right:

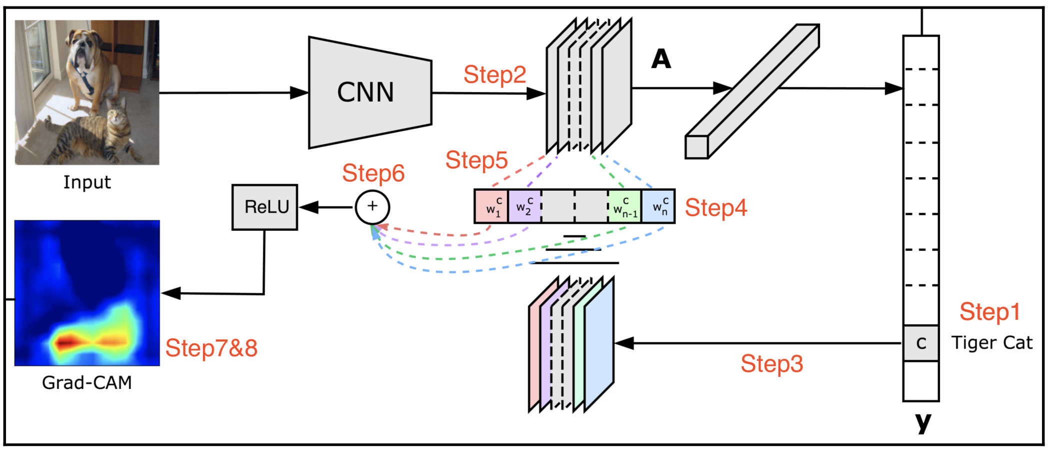


Let's understand how CAMs can be generated once a model is trained. Feature maps are intermediate activations that come after a convolution operation. Typically, the shape of these activation maps is `n-channels x height x width`. If we take the mean of all these activations, they show the hotspots of all the classes in the image. But if we are interested in locations that are only important for a particular class (say, `cat`), we need to figure out only those feature maps among `n-channels` that are responsible for that class. For the convolution layer that generated these feature maps, we can compute its gradients with respect to the `cat` class. Note that only those channels that are responsible for predicting `cat` will have a high gradient. This means that we can use the gradient information to give weightage to each of `n-channels` and obtain an activation map exclusively for `cat`.

Now that we understand the high-level strategy of how to generate CAMs, let's put it into practice step by step:

1. Decide for which class you want to calculate the CAM and for which convolutional layer in the neural network you want to compute the CAM.
2. Calculate the activations arising from any convolutional layer – let's say the feature shape at a random convolution layer is 512 x 7 x 7.
3. Fetch the gradient values arising from this layer with respect to the class of interest. The output gradient shape is 256 x 512 x 3 x 3 (which is the shape of the convolutional tensor – that is, `in-channels x out-channels x kernel-size x kernel-size`).
4. Compute the mean of the gradients within each output channel. The output shape is 512.
5. Calculate the weighted activation map – which is the multiplication of the 512 gradient means by the 512 activation channels. The output shape is 512 x 7 x 7.
6. Compute the mean (across 512 channels) of the weighted activation map to fetch an output of the shape 7 x 7.
7. Resize (upscale) the weighted activation map outputs to fetch an image of a size that is of the same size as the input. This is done so that we have an activation map that resembles the original image.
8. Overlay the weighted activation map onto the input image.

The following diagram from the paper *Grad-CAM: Gradient-weighted Class Activation Mapping* (`https://arxiv.org/abs/1610.02391`) pictorially describes the preceding steps:



The key to the entire process lies in *step 5*. We consider two aspects of the step:

- If a certain pixel is important, then the CNN will have a large activation at those pixels.
- If a certain convolutional channel is important with respect to the required class, the gradients at that channel will be very large.

On multiplying these two, we indeed end up with a map of importance across all the pixels.

The preceding strategy is implemented in code to understand the reason why the CNN model predicts that an image indicates the likelihood of an incident of malaria, as follows:

> The following code is available as `Class_activation_maps.ipynb` in the `Chapter06` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from and is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components in text.

1. Download the dataset and import the relevant packages:

```
import os
if not os.path.exists('cell_images'):
    !pip install -U -q torch_snippets
    !wget -q ftp://lhcftp.nlm.nih.gov/Open-Access-Datasets/
     Malaria/cell_images.zip
    !unzip -qq cell_images.zip
    !rm cell_images.zip
from torch_snippets import *
```

2. Specify the indices corresponding to the output classes:

```
id2int = {'Parasitized': 0, 'Uninfected': 1}
```

3. Perform the transformations to be done on top of the images:

```
from torchvision import transforms as T

trn_tfms = T.Compose([
                T.ToPILImage(),
                T.Resize(128),
                T.CenterCrop(128),
```

```
                    T.ColorJitter(brightness=(0.95,1.05),
                                  contrast=(0.95,1.05),
                                  saturation=(0.95,1.05),
                                  hue=0.05),
                    T.RandomAffine(5, translate=(0.01,0.1)),
                    T.ToTensor(),
                    T.Normalize(mean=[0.5, 0.5, 0.5],
                                std=[0.5, 0.5, 0.5]),
                ])
```

In the preceding code, we have a pipeline of transformations on top of the input image – which is a pipeline of resizing the image (which ensures that the minimum size of one of the dimensions is `128`, in this case) and then cropping it from the center. Furthermore, we are performing random color jittering and affine transformation. Next, we are scaling an image using the `.ToTensor` method to have a value between `0` and `1`, and finally, we are normalizing the image. As discussed in `Chapter 4`, *Introducing Convolutional Neural Networks*, we can also use the `imgaug` library.

- Specify the transformations to be done on the validation images:

```
val_tfms = T.Compose([
                T.ToPILImage(),
                T.Resize(128),
                T.CenterCrop(128),
                T.ToTensor(),
                T.Normalize(mean=[0.5, 0.5, 0.5],
                            std=[0.5, 0.5, 0.5]),
            ])
```

4. Define the dataset class – `MalariaImages`:

```
class MalariaImages(Dataset):

    def __init__(self, files, transform=None):
        self.files = files
        self.transform = transform
        logger.info(len(self))

    def __len__(self):
        return len(self.files)

    def __getitem__(self, ix):
        fpath = self.files[ix]
        clss = fname(parent(fpath))
        img = read(fpath, 1)
        return img, clss
```

```
    def choose(self):
        return self[randint(len(self))]

    def collate_fn(self, batch):
        _imgs, classes = list(zip(*batch))
        if self.transform:
            imgs = [self.transform(img)[None] \
                        for img in _imgs]
        classes = [torch.tensor([id2int[clss]]) \
                       for class in classes]
        imgs, classes = [torch.cat(i).to(device) \
                             for i in [imgs, classes]]
        return imgs, classes, _imgs
```

5. Fetch the training and validation datasets and dataloaders:

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
all_files = Glob('cell_images/*/*.png')
np.random.seed(10)
np.random.shuffle(all_files)

from sklearn.model_selection import train_test_split
trn_files, val_files = train_test_split(all_files, \
                                        random_state=1)

trn_ds = MalariaImages(trn_files, transform=trn_tfms)
val_ds = MalariaImages(val_files, transform=val_tfms)
trn_dl = DataLoader(trn_ds, 32, shuffle=True,
                    collate_fn=trn_ds.collate_fn)
val_dl = DataLoader(val_ds, 32, shuffle=False,
                    collate_fn=val_ds.collate_fn)
```

6. Define the model – `MalariaClassifier`:

```
def convBlock(ni, no):
    return nn.Sequential(
        nn.Dropout(0.2),
        nn.Conv2d(ni, no, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.BatchNorm2d(no),
        nn.MaxPool2d(2),
    )
class MalariaClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            convBlock(3, 64),
            convBlock(64, 64),
```

```
                convBlock(64, 128),
                convBlock(128, 256),
                convBlock(256, 512),
                convBlock(512, 64),
                nn.Flatten(),
                nn.Linear(256, 256),
                nn.Dropout(0.2),
                nn.ReLU(inplace=True),
                nn.Linear(256, len(id2int))
            )
            self.loss_fn = nn.CrossEntropyLoss()

        def forward(self, x):
            return self.model(x)

        def compute_metrics(self, preds, targets):
            loss = self.loss_fn(preds, targets)
            acc =(torch.max(preds, 1)[1]==targets).float().mean()
            return loss, acc
```

7. Define the functions to train and validate on a batch of data:

```
def train_batch(model, data, optimizer, criterion):
    model.train()
    ims, labels, _ = data
    _preds = model(ims)
    optimizer.zero_grad()
    loss, acc = criterion(_preds, labels)
    loss.backward()
    optimizer.step()
    return loss.item(), acc.item()

@torch.no_grad()
def validate_batch(model, data, criterion):
    model.eval()
    ims, labels, _ = data
    _preds = model(ims)
    loss, acc = criterion(_preds, labels)
    return loss.item(), acc.item()
```

8. Train the model over increasing epochs:

```
model = MalariaClassifier().to(device)
criterion = model.compute_metrics
optimizer = optim.Adam(model.parameters(), lr=1e-3)
n_epochs = 2

log = Report(n_epochs)
```

```
for ex in range(n_epochs):
    N = len(trn_dl)
    for bx, data in enumerate(trn_dl):
        loss, acc = train_batch(model, data, optimizer, \
                                    criterion)
        log.record(ex+(bx+1)/N,trn_loss=loss,trn_acc=acc, \
                                    end='\r')

    N = len(val_dl)
    for bx, data in enumerate(val_dl):
        loss, acc = validate_batch(model, data, criterion)
        log.record(ex+(bx+1)/N,val_loss=loss,val_acc=acc, \
                                    end='\r')
    log.report_avgs(ex+1)
```

9. Fetch the convolution layer in the fifth `convBlock` in the model:

```
im2fmap = nn.Sequential(*(list(model.model[:5].children())+ \
                        list(model.model[5][:2].children())))
```

In the preceding line of code, we are fetching the fourth layer of the model and also the first two layers within `convBlock` – which happens to be the `Conv2D` layer.

10. Define the `im2gradCAM` function that takes an input image and fetches the heatmap corresponding to activations of the image:

```
def im2gradCAM(x):
    model.eval()
    logits = model(x)
    heatmaps = []
    activations = im2fmap(x)
    print(activations.shape)
    pred = logits.max(-1)[-1]
    # get the model's prediction
    model.zero_grad()
    # compute gradients with respect to
    # model's most confident logit
    logits[0,pred].backward(retain_graph=True)
    # get the gradients at the required featuremap location
    # and take the avg gradient for every featuremap
    pooled_grads = model.model[-7][1]\
                        .weight.grad.data.mean((0,2,3))
```

```
        # multiply each activation map with
        # corresponding gradient average
        for i in range(activations.shape[1]):
            activations[:,i,:,:] *= pooled_grads[i]
        # take the mean of all weighted activation maps
        # (that has been weighted by avg. grad at each fmap)
        heatmap =torch.mean(activations, dim=1)[0].cpu().detach()
        return heatmap, 'Uninfected' if pred.item() \
    else 'Parasitized'
```

11. Define the `upsampleHeatmap` function to up-sample the heatmap to a shape that corresponds to the shape of the image:

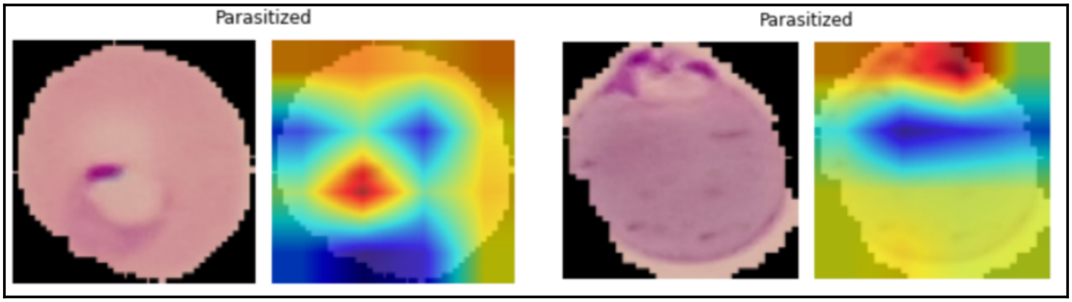
```
SZ = 128
def upsampleHeatmap(map, img):
    m,M = map.min(), map.max()
    map = 255 * ((map-m) / (M-m))
    map = np.uint8(map)
    map = cv2.resize(map, (SZ,SZ))
    map = cv2.applyColorMap(255-map, cv2.COLORMAP_JET)
    map = np.uint8(map)
    map = np.uint8(map*0.7 + img*0.3)
    return map
```

In the preceding lines of code, we are de-normalizing the image and also overlaying the heatmap on top of the image.

12. Run the preceding functions on a set of images:

```
N = 20
_val_dl = DataLoader(val_ds, batch_size=N, shuffle=True, \
                     collate_fn=val_ds.collate_fn)
x,y,z = next(iter(_val_dl))

for i in range(N):
    image = resize(z[i], SZ)
    heatmap, pred = im2gradCAM(x[i:i+1])
    if(pred=='Uninfected'):
        continue
    heatmap = upsampleHeatmap(heatmap, image)
    subplots([image, heatmap], nc=2, figsize=(5,3), \
                suptitle=pred)
```

The output of the preceding code is as follows:



From this, we can see that the prediction is as it is because of the content that is highlighted in red (which has the highest CAM value).

Now that we have learned about generating class activation heatmaps for images using a trained model, we are in a position to explain what makes a certain classification so.

In the next section, let's learn about additional tricks around data augmentation that can help when building models.

Understanding the impact of data augmentation and batch normalization

One clever way of improving the accuracy of models is by leveraging data augmentation. We have already seen this in `Chapter 4`, *Introducing Convolutional Neural Networks*, where we used data augmentation to improve the accuracy of classification on a translated image. In the real world, you would encounter images that have different properties – for example, some images might be much brighter, some might contain objects of interest near the edges, and some images might be more jittery than others. In this section, we will learn about how the usage of data augmentation can help in improving the accuracy of a model. Furthermore, we will learn about how data augmentation can practically be a pseudo-regularizer for our models.

To understand the impact of data augmentation and batch normalization, we will go through a dataset of recognizing traffic signs. We will evaluate three scenarios:

- No batch normalization/data augmentation
- Only batch normalization, but no data augmentation
- Both batch normalization and data augmentation

Note that given that the dataset and processing remain the same across the three scenarios, and only the data augmentation and model (the addition of the batch normalization layer) differ, we will only provide the following code for the first scenario, but the other two scenarios are available in the notebook on GitHub.

Coding up road sign detection

Let's code up for road sign detection without data augmentation and batch normalization, as follows:

> Note that we are not explaining the code here, as it is very much inline with the code that we have gone through in previous chapters – only the lines with bold font are different across the three scenarios. The following code is available as `road_sign_detection.ipynb` in the `Chapter06` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Download the dataset and import the relevant packages:

```
import os
if not os.path.exists('GTSRB'):
    !pip install -U -q torch_snippets
    !wget -qq https://sid.erda.dk/public/archives/
        daaeac0d7ce1152aea9b61d9f1e19370/
        GTSRB_Final_Training_Images.zip
    !wget -qq https://sid.erda.dk/public/archives/
        daaeac0d7ce1152aea9b61d9f1e19370/
        GTSRB_Final_Test_Images.zip
    !unzip -qq GTSRB_Final_Training_Images.zip
    !unzip -qq GTSRB_Final_Test_Images.zip
    !wget https://raw.githubusercontent.com/georgesung/
     traffic_sign_classification_german/master/signnames.csv
    !rm GTSRB_Final_Training_Images.zip
       GTSRB_Final_Test_Images.zip

from torch_snippets import *
```

2. Assign the class IDs to possible output classes:

```
classIds = pd.read_csv('signnames.csv')
classIds.set_index('ClassId', inplace=True)
classIds = classIds.to_dict()['SignName']
classIds = {f'{k:05d}':v for k,v in classIds.items()}
id2int = {v:ix for ix,(k,v) in enumerate(classIds.items())}
```

3. Define the transformation pipeline on top of the images without any augmentation:

```
from torchvision import transforms as T
trn_tfms = T.Compose([
                T.ToPILImage(),
                T.Resize(32),
                T.CenterCrop(32),
                # T.ColorJitter(brightness=(0.8,1.2),
                # contrast=(0.8,1.2),
                # saturation=(0.8,1.2),
                # hue=0.25),
                # T.RandomAffine(5, translate=(0.01,0.1)),
                T.ToTensor(),
                T.Normalize(mean=[0.485, 0.456, 0.406],
                            std=[0.229, 0.224, 0.225]),
            ])
```

```
val_tfms = T.Compose([
                T.ToPILImage(),
                T.Resize(32),
                T.CenterCrop(32),
                T.ToTensor(),
                T.Normalize(mean=[0.485, 0.456, 0.406],
                            std=[0.229, 0.224, 0.225]),
            ])
```

In the preceding code, we are specifying that we convert each image into a PIL image and resize and crop the image from the center. Furthermore, we are scaling the image to have pixel values that are between `0` and `1` using the `.ToTensor` method. Finally, we are normalizing the input image so that a pre-trained model can be leveraged.

> The commented part of the preceding code is what you should uncomment and re-run to understand the scenario of performing data augmentation. Furthermore, we are not performing augmentations on `val_tfms` as those images are not used during the training of the model. However, the `val_tfms` images should go through the same transformation pipeline as `trn_tfms`.

4. Define the dataset class – `GTSRB`:

```
class GTSRB(Dataset):

    def __init__(self, files, transform=None):
        self.files = files
        self.transform = transform
        logger.info(len(self))

    def __len__(self):
        return len(self.files)

    def __getitem__(self, ix):
        fpath = self.files[ix]
        clss = fname(parent(fpath))
        img = read(fpath, 1)
        return img, classIds[clss]

    def choose(self):
        return self[randint(len(self))]
```

```
    def collate_fn(self, batch):
        imgs, classes = list(zip(*batch))
        if self.transform:
            imgs =[self.transform(img)[None] \
                   for img in imgs]
        classes = [torch.tensor([id2int[clss]]) \
                   for clss in classes]
        imgs, classes = [torch.cat(i).to(device) \
                         for i in [imgs, classes]]
        return imgs, classes
```

5. Create the training and validation datasets and dataloaders:

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
all_files = Glob('GTSRB/Final_Training/Images/*/*.ppm')
np.random.seed(10)
np.random.shuffle(all_files)

from sklearn.model_selection import train_test_split
trn_files, val_files = train_test_split(all_files, \
                                        random_state=1)

trn_ds = GTSRB(trn_files, transform=trn_tfms)
val_ds = GTSRB(val_files, transform=val_tfms)
trn_dl = DataLoader(trn_ds, 32, shuffle=True, \
                    collate_fn=trn_ds.collate_fn)
val_dl = DataLoader(val_ds, 32, shuffle=False, \
                    collate_fn=val_ds.collate_fn)
```

6. Define the model – `SignClassifier`:

```
import torchvision.models as models

def convBlock(ni, no):
    return nn.Sequential(
                nn.Dropout(0.2),
                nn.Conv2d(ni, no, kernel_size=3, padding=1),
                nn.ReLU(inplace=True),
                #nn.BatchNorm2d(no),
                nn.MaxPool2d(2),
            )
class SignClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
                        convBlock(3, 64),
                        convBlock(64, 64),
                        convBlock(64, 128),
```

```
                        convBlock(128, 64),
                        nn.Flatten(),
                        nn.Linear(256, 256),
                        nn.Dropout(0.2),
                        nn.ReLU(inplace=True),
                        nn.Linear(256, len(id2int))
                    )
        self.loss_fn = nn.CrossEntropyLoss()

    def forward(self, x):
        return self.model(x)

    def compute_metrics(self, preds, targets):
        ce_loss = self.loss_fn(preds, targets)
        acc =(torch.max(preds, 1)[1]==targets).float().mean()
        return ce_loss, acc
```

Make sure to uncomment the line in bold in the preceding code when you are testing the model with the `BatchNormalization` scenario.

7. Define the functions to train and validate on a batch of data, respectively:

```
def train_batch(model, data, optimizer, criterion):
    model.train()
    ims, labels = data
    _preds = model(ims)
    optimizer.zero_grad()
    loss, acc = criterion(_preds, labels)
    loss.backward()
    optimizer.step()
    return loss.item(), acc.item()

@torch.no_grad()
def validate_batch(model, data, criterion):
    model.eval()
    ims, labels = data
    _preds = model(ims)
    loss, acc = criterion(_preds, labels)
    return loss.item(), acc.item()
```

8. Define the model and train it over increasing epochs:

```
model = SignClassifier().to(device)
criterion = model.compute_metrics
optimizer = optim.Adam(model.parameters(), lr=1e-3)
n_epochs = 50
```

```
log = Report(n_epochs)
for ex in range(n_epochs):
    N = len(trn_dl)
    for bx, data in enumerate(trn_dl):
        loss, acc = train_batch(model, data, optimizer, \
                                    criterion)
        log.record(ex+(bx+1)/N,trn_loss=loss, trn_acc=acc, \
                                     end='\r')

    N = len(val_dl)
    for bx, data in enumerate(val_dl):
        loss, acc = validate_batch(model, data, criterion)
        log.record(ex+(bx+1)/N, val_loss=loss, val_acc=acc, \
                                    end='\r')
    log.report_avgs(ex+1)
    if ex == 10: optimizer = optim.Adam(model.parameters(), \
                                    lr=1e-4)
```

The lines of code that are bold are the ones that you would change in the three scenarios. The results of the three scenarios in terms of training and validation accuracy are as follows:

Augment	Batch-Norm	Train Accuracy	Validation Accuracy
No	No	95.9	94.5
No	Yes	99.3	97.7
Yes	Yes	97.7	97.6

Note that in the preceding three scenarios, we see the following:

- The model did not have as high accuracy when there was no batch normalization.
- The accuracy of the model increased considerably but also the model overfitted on training data when we had batch normalization only but no data augmentation.
- The model with both batch normalization and data augmentation had high accuracy and minimal overfitting (as the training and validation loss values are very similar).

With the importance of batch normalization and data augmentation in place, in the next section, we will learn about some key aspects to take care of when training/implementing our image classification models.

Practical aspects to take care of during model implementation

So far, we have seen the various ways of building an image classification model. In this section, we will learn about some of the practical considerations that need to be taken care of when building models. The ones we will discuss in this chapter are as follows:

- Dealing with imbalanced data
- The size of an object within an image when performing classification
- The difference between training and validation images
- The number of convolutional and pooling layers in a network
- Image sizes to train on GPUs
- Leveraging OpenCV utilities

Dealing with imbalanced data

Imagine a scenario where you are trying to predict an object that occurs very rarely within our dataset – let's say in 1% of the total images. For example, this can be the task of predicting whether an X-ray image suggests a rare lung infection.

How do we measure the accuracy of the model that is trained to predict the rare lung infection? If we simply predict a class of no infection for all images, the accuracy of classification is 99%, while still being useless. A confusion matrix that depicts the number of times the rare object class has occurred and the number of times the model predicted the rare object class correctly comes in handy in this scenario. Thus, the right set of metrics to look at in this scenario is the metrics related to the confusion matrix.

A typical confusion matrix looks as follows:

	Predicted	
Actual	0	1
0	TN	FP
1	FN	TP

In the preceding confusion matrix, `0` stands for no infection and `1` stands for infection. Typically, we would fill up the matrix to understand how accurate our model is.

Next comes the question of ensuring that the model gets trained. Typically, the loss function (binary or categorical cross-entropy) takes care of ensuring that the loss values are high when the amount of misclassification is high. However, in addition to the loss function, we can also assign a higher weight to the rarely occurring class, thereby ensuring that we explicitly mention to the model that we want to correctly classify the rare class images.

In addition to assigning class weights, we have already seen that image augmentation and/or transfer learning help considerably in improving the accuracy of the model. Furthermore, when augmenting an image, we can over-sample the rare class images to increase their mix in the overall population.

The size of the object within an image

Imagine a scenario where the presence of a small patch within a large image dictates the class of the image – for example, lung infection identification where the presence of certain tiny nodules indicates an incident of the disease. In such a scenario, image classification is likely to result in inaccurate results, as the object occupies a smaller portion of the entire image. Object detection comes in handy in this scenario (which we will study in the next chapter).

A high-level intuition to solve these problems would be to first divide the input images into smaller grid cells (let's say a 10 x 10 grid) and then identify whether a grid cell contains the object of interest.

Dealing with the difference between training and validation data

Imagine a scenario where you have built a model to predict whether the image of an eye indicates that the person is likely to be suffering from diabetic retinopathy. To build the model, you have collected data, curated it, cropped it, normalized it, and then finally built a model that has very high accuracy on validation images. However, hypothetically, when the model is used in a real setting (let's say by a doctor/nurse), the model is not able to predict well. Let's understand a few possible reasons why:

- Are the images taken at the doctor's office similar to the images used to train the model?
  - Images used when training and real-world images could be very different if you built a model on a curated set of data that has all the preprocessing done, while the images taken at the doctor's end are non-curated.
  - Images could be different if the device used to capture images at the doctor's office has a different resolution of capturing images when compared to the device used to collect images that are used for training.
  - Images can be different if there are different lighting conditions at which the images are getting captured in both places.
- Are the subjects (images) representative enough of the overall population?
  - Images are representative if they are trained on images of the male population but are tested on the female population, or if, in general, the training and real-world images correspond to different demographics.
- Is the training and validation split done methodically?
  - Imagine a scenario where there are 10,000 images and the first 5,000 images belong to one class and the last 5,000 images belong to another class. When building a model, if we do not randomize but split the dataset into training and validation with consecutive indices (without random indices), we are likely to see a higher representation of one class while training and of the other class during validation.

In general, we need to ensure that the training, validation, and real-world images all have similar data distribution before an end user leverages the system.

The number of nodes in the flatten layer

Consider a scenario where you are working on images that are 300 x 300 in dimensions. Technically, we can perform more than five convolutional pooling operations to get the final layer that has as many features as possible. Furthermore, we can have as many channels as we want in this scenario within a CNN. Practically, though, in general, we would design a network so that it has 500–5,000 nodes in the flatten layer.

As we saw in `Chapter 4`, *Introducing Convolutional Neural Networks*, if we have a greater number of nodes in the flatten layer, we would have a very high number of parameters when the flatten layer is connected to the subsequent dense layer before connecting to the final classification layer.

In general, it is good practice to have a pre-trained model that obtains the flatten layer so that relevant filters are activated as appropriate. Furthermore, when leveraging pre-trained models, make sure to freeze the parameters of the pre-trained model.

Generally, the number of trainable parameters in a CNN can be anywhere between 1 million to 10 million in a less complex classification exercise.

Image size

Let's say we are working on images that are of very high dimensions – for example, 2,000 x 1,000 in shape. When working on such large images, we need to consider the following possibilities:

- Can the images be resized to lower dimensions? Images of objects might not lose information if resized; however, images of text documents might lose considerable information if resized to a smaller size.
- Can we have a lower batch size so that the batch fits into GPU memory? Typically, if we are working with large images, there is a good chance that for the given batch size, the GPU memory is not sufficient to perform computations on the batch of images.
- Do certain portions of the image contain the majority of the information, and hence can the rest of the image be cropped?

Leveraging OpenCV utilities

OpenCV is an open source package that has extensive modules that help in fetching information from images (more on OpenCV utilities in `Chapter 18`, *Using OpenCV Utilities for Image Analysis*). It was one of the most prominent libraries used prior to the deep learning revolution in computer vision. Traditionally, it has been built on top of multiple hand-engineered features and at the time of writing this book, OpenCV has a few packages that integrate deep learning models' outputs.

Imagine a scenario where you have to move a model to production; less complexity is generally preferable in such a scenario – sometimes even at the cost of accuracy. If any OpenCV module solves the problem that you are already trying to solve, in general, it should be preferred over building a model (unless building a model from scratch gives a considerable boost in accuracy than leveraging off-the-shelf modules).

Summary

In this chapter, we learned about multiple practical aspects that we need to take into consideration when building CNN models – batch normalization, data augmentation, explaining the outcomes using CAMs, and some scenarios that you need to be aware of when moving a model to production.

In the next chapter, we will switch gears and learn about the fundamentals of object detection – where we will not only identify the classes corresponding to objects in an image but also draw a bounding box around the location of the object.

Questions

1. How are CAMs obtained?
2. How do batch normalization and data augmentation help when training a model?
3. What are the common reasons why a CNN model overfits?
4. What are the various scenarios where the CNN model works with training and validation data at the data scientists' end but not in the real world?
5. What are the various scenarios where we leverage OpenCV packages?

7
Basics of Object Detection

So far, in the previous chapters, we learned about performing image classification. Imagine a scenario where we are leveraging computer vision for a self-driving car. It is not only necessary to detect whether the image of a road contains the images of vehicles, a sidewalk, and pedestrians, but it is also important to identify *where* those objects are located. Various techniques of object detection that we will study in this chapter and the next will come in handy in such a scenario.

In this chapter and the next, we will learn about some of the techniques for performing object detection. We will start by learning about the fundamentals—labeling the ground truth of bounding box objects using a tool named `ybat`, extracting region proposals using the `selectivesearch` method, and defining the accuracy of bounding box predictions by using the **Intersection over Union** (**IoU**) metric and the mean average precision metric. After this, we will learn about two region proposal-based networks – R-CNN and Fast R-CNN, by first learning about their working details and then implementing them on a dataset that contains images belonging to trucks and buses.

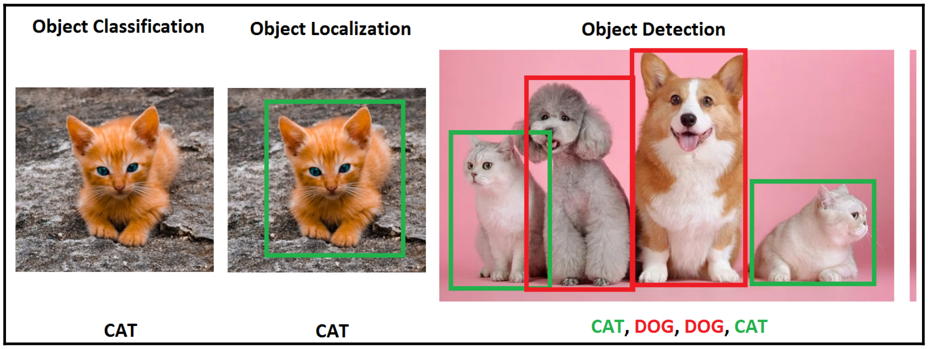
The following topics will be covered in this chapter:

- Introducing object detection
- Creating a bounding box ground truth for training
- Understanding region proposals
- Understanding IoU, non-max suppression, and mean average precision
- Training R-CNN-based custom object detectors
- Training Fast R-CNN-based custom object detectors

Introducing object detection

With the rise of autonomous cars, facial detection, smart video surveillance, and people-counting solutions, fast and accurate object detection systems are in great demand. These systems include not only object classification from an image, but also location of each one of the objects by drawing appropriate bounding boxes around them. This (drawing bounding boxes and classification) makes object detection a harder task than its traditional computer vision predecessor, image classification.

To understand what the output of object detection looks like, let's go through the following diagram:



In the preceding diagram, we can see that, while a typical object classification merely mentions the class of object present in the image, object localization draws a bounding box around the objects present in the image. Object detection, on the other hand, would involve drawing the bounding boxes around individual objects in the image, along with identifying the class of object within a bounding box across the multiple objects present in the image.

Before we understand the broad use cases of object detection, let's understand how it adds to the object classification task that we have covered in the previous chapter.

Imagine a scenario where you have multiple objects in an image. I ask you to predict the class of objects present in the image. For example, let's say that the image contains both cats and dogs. How would you classify such images? Object detection comes in handy in such a scenario, where it not only predicts the location of objects (bounding box) present in it, but also predicts the class of object present within the individual bounding boxes.

Some of the various use cases leveraging object detection include the following:

- **Security:** This can be useful for recognizing intruders.
- **Autonomous cars**: This can be helpful in recognizing the various objects present on the image of a road.
- **Image searching:** This can help identify the images containing an object (or a person) of interest.
- **Automotives**: This can help in identifying a number plate within the image of a car.

In all the preceding cases, object detection is leveraged to draw bounding boxes around a variety of objects present within the image.

In this chapter, we will learn about predicting the class of the object and also having a tight bounding box around the object in the image, which is the localization task. We will also learn about detecting the class corresponding to multiple objects in the picture, along with a bounding box around each object, which is the object detection task.

Training a typical object detection model involves the following steps:

1. Creating ground truth data that contains labels of the bounding box and class corresponding to various objects present in the image.
2. Coming up with mechanisms that scan through the image to identify regions (region proposals) that are likely to contain objects. In this chapter, we will learn about leveraging region proposals generated by a method named *selective search*. In the next chapter, we will learn about leveraging anchor boxes to identify regions containing objects. In the chapter on combining computer vision and NLP techniques (Chapter 15), we will learn about leveraging positional embeddings in transformers to aid in identifying the regions containing an object.
3. Creating the target class variable by using the IoU metric.
4. Creating the target bounding box offset variable to make corrections to the location of region proposal coming in the second step.
5. Building a model that can predict the class of object along with the target bounding box offset corresponding to the region proposal.
6. Measuring the accuracy of object detection using **mean Average Precision** (**mAP**).

Now that we have a high-level overview of what is to be done to train an object detection model, we will learn about creating the dataset for a bounding box (which is the first step in building an object detection model) in the next section.

Creating a bounding box ground truth for training

We have learned that object detection gives us the output where a bounding box surrounds the object of interest in an image. For us to build an algorithm that detects the bounding box surrounding the object in an image, we would have to create the input-output combinations, where the input is the image and the output is the bounding boxes surrounding the objects in the given image, and the classes corresponding to the objects.

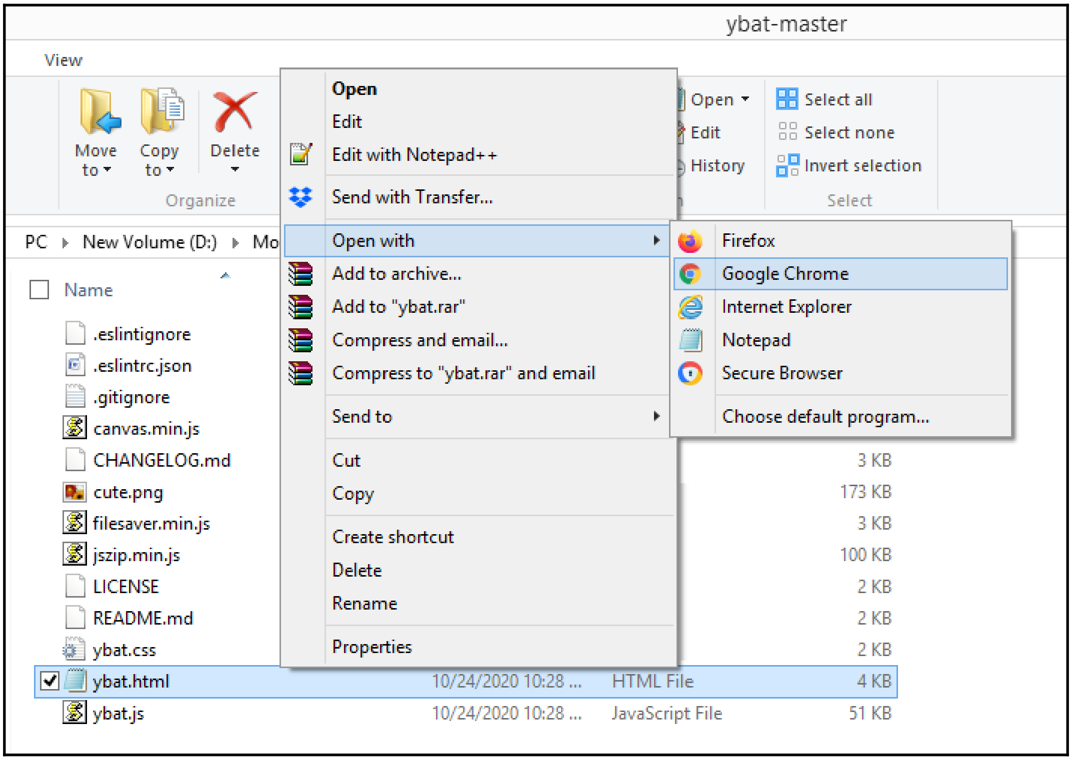
Note that when we detect the bounding box, we are detecting the pixel locations of the four corners of the bounding box surrounding the image.

To train a model that provides the bounding box, we need the image, and also the corresponding bounding box coordinates of all the objects in an image. In this section, we will learn about one way to create the training dataset, where the image is the input and the corresponding bounding boxes and classes of objects are stored in an XML file as output. We will use the `ybat` tool to annotate the bounding boxes and the corresponding classes.

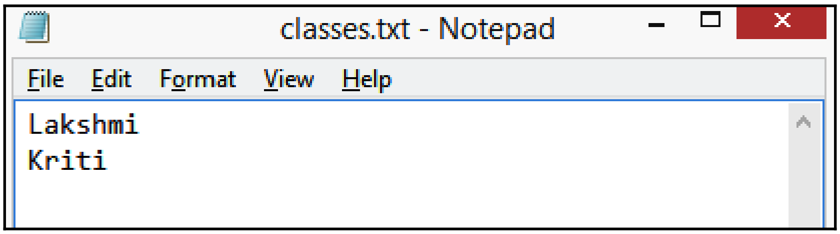
Let's understand about installing and using `ybat` to create (annotate) bounding boxes around objects in the image. Furthermore, we will also be inspecting the XML files that contain the annotated class and bounding box information in the following section.

Installing the image annotation tool

Let's start by downloading `ybat-master.zip` from the following GitHub link, `https://github.com/drainingsun/ybat`, and unzip it. Post unzipping, store it in a folder of your choice. Open `ybat.html` using a browser of your choice and you will see an empty page. The following screenshot shows a sample of what the folder looks like and how to open the `ybat.html` file:



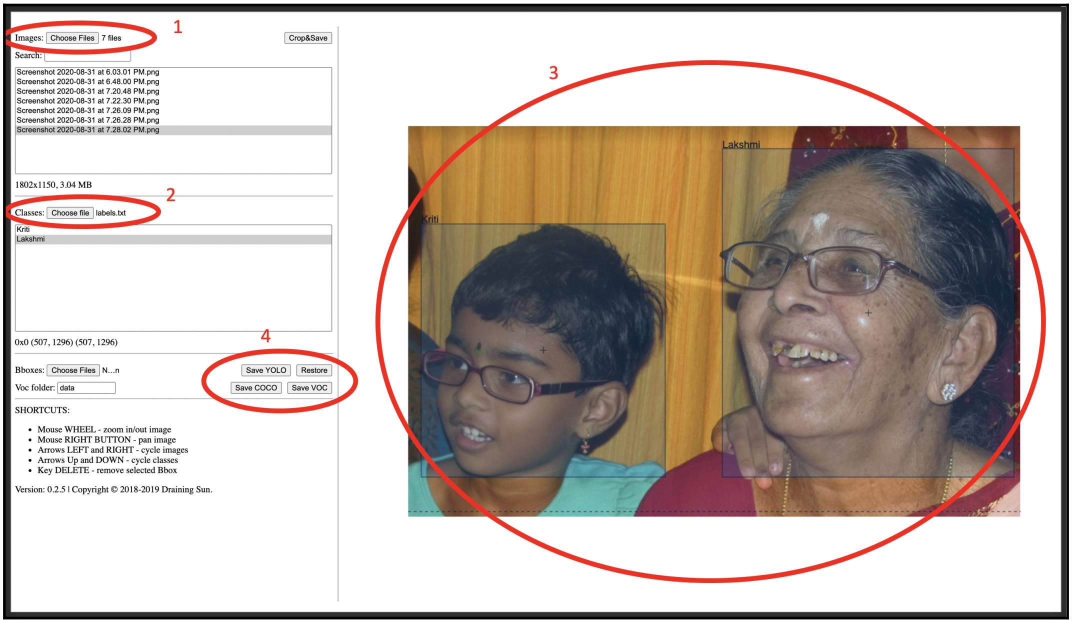
Before we start creating the ground truth corresponding to an image, let's specify all the possible classes that we want to label across images and store in the `classes.txt` file as follows:



Now, let's prepare the ground truth corresponding to an image. This involves drawing a bounding box around objects (persons in the following diagram) and assigning labels/classes to the objects present in the image in the following steps:

1. Upload all the images you want to annotate (step number 1 in the following image).
2. Upload the `classes.txt` file (step number 2 in the following image).
3. Label each image by first selecting the filename and then drawing a crosshair around each object you want to label (step number 3 in the following image). Before drawing a crosshair, ensure you select the correct class in the classes region (the classes pane below the second oval in the following image).
4. Save the data dump in the desired format (step number 4 in the following image). Each format was independently developed by a different research team, and all are equally valid. Based on their popularity and convenience, every implementation prefers a different format.

All these steps can be better represented using the following diagram:



For example, when we download the PascalVOC format, it downloads a zip of XML files. A snapshot of the XML file after drawing the rectangular bounding box is as follows:

```
<?xml version="1.0"?>
<annotation>
<folder>data</folder>
<filename>Screenshot 2020-08-31 at 7.28.02 PM.png</filename>
<path/>
<source>
<database>Unknown</database>
</source>
<size>
<width>1802</width>
<height>1150</height>
<depth>3</depth>
</size>
<segmented>0</segmented>
<object>
<name>Kriti</name>
<pose>Unspecified</pose>
<truncated>0</truncated>
<occluded>0</occluded>
<difficult>0</difficult>
<bndbox>
<xmin>38</xmin>
<ymin>287</ymin>
<xmax>757</xmax>
<ymax>1033</ymax>
</bndbox>
</object>
<object>
<name>Lakshmi</name>
<pose>Unspecified</pose>
<truncated>0</truncated>
<occluded>0</occluded>
<difficult>0</difficult>
<bndbox>
<xmin>925</xmin>
<ymin>66</ymin>
<xmax>1784</xmax>
<ymax>1033</ymax>
</bndbox>
</object>
</annotation>
```

From the preceding screenshot, note that the `bndbox` field contains the coordinates of the minimum and maximum values of the x and y coordinates corresponding to the objects of interest in the image. We should also be able to extract the classes corresponding to the objects in the image using the `name` field.

Now that we understand how to create a ground truth of objects (class label and bounding box) present in an image, in the following sections, we will dive into the building blocks of recognizing objects in an image. First, we will talk about region proposals that help in highlighting the portions of the image that are most likely to contain an object.

Understanding region proposals

Imagine a hypothetical scenario where the image of interest contains a person and sky in the background. Furthermore, for this scenario, let's assume that there is little change in pixel intensity of the background (sky) and that there is a considerable change in pixel intensity of the foreground (the person).

Just from the preceding description itself, we can conclude that there are two primary regions here – one is of the person and the other is of the sky. Furthermore, within the region of the image of a person, the pixels corresponding to hair will have a different intensity to the pixels corresponding to the face, establishing that there can be multiple sub-regions within a region.

Region proposal is a technique that helps in identifying islands of regions where the pixels are similar to one another.

Generating a region proposal comes in handy for object detection where we have to identify the locations of objects present in the image. Furthermore, given a region proposal generates a proposal for the region, it aids in object localization where the task is to identify a bounding box that fits exactly around the object in the image. We will learn how region proposals assist in object localization and detection in a later section on *Training R-CNN-based custom object detectors*, but let's first understand how to generate region proposals from an image.

Leveraging SelectiveSearch to generate region proposals

SelectiveSearch is a region proposal algorithm used for object localization where it generates proposals of regions that are likely to be grouped together based on their pixel intensities. SelectiveSearch groups pixels based on the hierarchical grouping of similar pixels, which, in turn, leverages the color, texture, size, and shape compatibility of content within an image.

Initially, SelectiveSearch over-segments an image by grouping pixels based on the preceding attributes. Next, it iterates through these over-segmented groups and groups them based on similarity. At each iteration, it combines smaller regions to form a larger region.

Let's understand the `selectivesearch` process through the following example:

> The following code is available as `Understanding_selectivesearch.ipynb` in the `Chapter07` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` Be sure to copy the URL from the notebook in GitHub to avoid any issue while reproducing the results

1. Install the required packages:

```
!pip install selectivesearch
!pip install torch_snippets
from torch_snippets import *
import selectivesearch
from skimage.segmentation import felzenszwalb
```

2. Fetch and load the required image:

```
!wget https://www.dropbox.com/s/l98leemr7r5stnm/Hemanvi.jpeg
img = read('Hemanvi.jpeg', 1)
```

3. Extract the `felzenszwalb` segments (which are obtained based on the color, texture, size, and shape compatibility of content within an image) from the image:

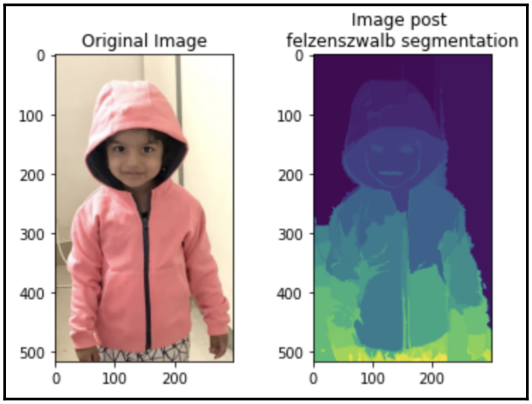
```
segments_fz = felzenszwalb(img, scale=200)
```

Note that in the `felzenszwalb` method, `scale` represents the number of clusters that can be formed within the segments of the image. The higher the value of `scale`, the greater the detail of the original image that is preserved.

4. Plot the original image and the image with segmentation:

```
subplots([img, segments_fz], \
         titles=['Original Image',\
                 'Image post\nfelzenszwalb segmentation'],\
         sz=10, nc=2)
```

The preceding code results in the following output:



From the preceding output, note that pixels that belong to the same group have similar pixel values.

Pixels that have similar values form a region proposal. This now helps in object detection, as we now pass each region proposal to a network and ask it to predict whether the region proposal is a background or an object. Furthermore, if it is an object, it would help us to identify the offset to fetch the tight bounding box corresponding to the object and also the class corresponding to the content within the region proposal.

Now that we understand what SelectiveSearch does, let's implement the `selectivesearch` function to fetch region proposals for the given image.

Implementing SelectiveSearch to generate region proposals

In this section, we will define the `extract_candidates` function using `selectivesearch` so that it can be leveraged in the subsequent sections on training R-CNN- and Fast R-CNN-based custom object detectors:

1. Define the `extract_candidates` function that fetches the region proposals from an image:

    - Define the function that takes an image as the input parameter:

```
def extract_candidates(img):
```

    - Fetch the candidate regions within the image using the `selective_search` method available in the `selectivesearch` package:

```
    img_lbl, regions = selectivesearch.selective_search(img, \
                                   scale=200, min_size=100)
```

    - Calculate the image area and initialize a list (candidates) that we will use to store the candidates that pass a defined threshold:

```
    img_area = np.prod(img.shape[:2])
    candidates = []
```

    - Fetch only those candidates (regions) that are over 5% of the total image area and less than or equal to 100% of the image area and return them:

```
    for r in regions:
        if r['rect'] in candidates: continue
        if r['size'] < (0.05*img_area): continue
        if r['size'] > (1*img_area): continue
        x, y, w, h = r['rect']
        candidates.append(list(r['rect']))
    return candidates
```

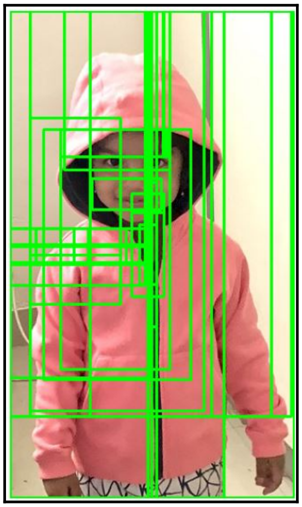
2. Import the relevant packages and fetch an image:

```
!pip install selectivesearch
!pip install torch_snippets
from torch_snippets import *
import selectivesearch
!wget https://www.dropbox.com/s/l98leemr7r5stnm/Hemanvi.jpeg
img = read('Hemanvi.jpeg', 1)
```

3. Extract candidates and plot them on top of an image:

```
candidates = extract_candidates(img)
show(img, bbs=candidates)
```

The preceding code generates the following output:



The grids in the preceding diagram represent the candidate regions (region proposals) coming from the `selective_search` method.

Now that we understand region proposal generation, one question remains unanswered. How do we leverage region proposals for object detection and localization?

A region proposal that has a high intersection with the location (ground truth) of an object in the image of interest is labeled as the one that contains the object, and a region proposal with a low intersection is labeled as background.

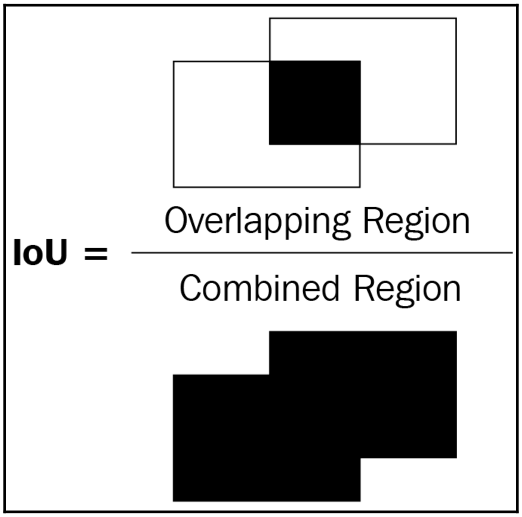
In the next section, we will learn about how to calculate the intersection of a region proposal candidate with a ground truth bounding box in our journey to understanding the various techniques that form the backbone of building an object detection model.

Understanding IoU

Imagine a scenario where we came up with a prediction of a bounding box for an object. How do we measure the accuracy of our prediction? The concept of **Intersection over Union** (**IoU**) comes in handy in such a scenario.

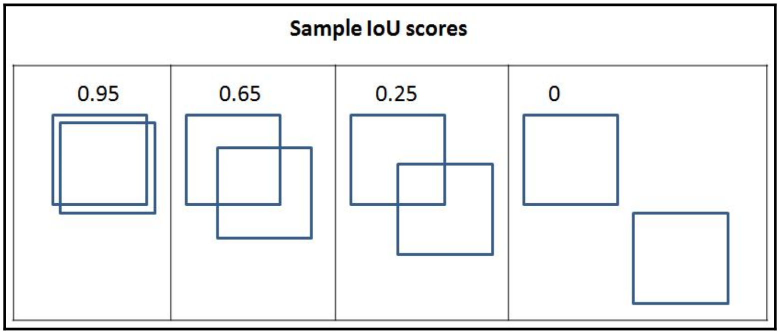
Intersection within the term *Intersection over Union* measures how overlapping the predicted and actual bounding boxes are, while *Union* measures the overall space possible for overlap. IoU is the ratio of the overlapping region between the two bounding boxes over the combined region of both the bounding boxes.

This can be represented in a diagram as follows:



In the preceding diagram of two bounding boxes (rectangles), let's consider the left bounding box as the ground truth and the right bounding box as the predicted location of the object. IoU as a metric is the ratio of the overlapping region over the combined region between the two bounding boxes.

In the following diagram, you can observe the variation in the IoU metric as the overlap between bounding boxes varies:



From the preceding diagram, we can see that as the overlap decreases, IoU decreases and, in the final one, where there is no overlap, the IoU metric is 0.

Now that we have an intuition of measuring IoU, let's implement it in code and create a function to calculate IoU as we will leverage it in the sections of training R-CNN and training Fast R-CNN.

> The following code is available as `Calculating_Intersection_Over_Union.ipynb` in the `Chapter07` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

Let's define a function that takes two bounding boxes as input and returns IoU as the output:

1. Specify the `get_iou` function that takes `boxA` and `boxB` as inputs where `boxA` and `boxB` are two different bounding boxes (you can consider `boxA` as the ground truth bounding box and `boxB` as the region proposal):

```
def get_iou(boxA, boxB, epsilon=1e-5):
```

   We define the `epsilon` parameter to address the rare scenario when the union between the two boxes is 0, resulting in a division by zero error. Note that in each of the bounding boxes, there will be four values corresponding to the four corners of the bounding box.

2. Calculate the coordinates of the intersection box:

```
    x1 = max(boxA[0], boxB[0])
    y1 = max(boxA[1], boxB[1])
```

```
x2 = min(boxA[2], boxB[2])
y2 = min(boxA[3], boxB[3])
```

Note that `x1` is storing the maximum value of the left-most *x*-value between the two bounding boxes. Similarly, `y1` is storing the topmost *y*-value and `x2` and `y2` are storing the right-most *x*-value and bottom-most *y*-value, respectively, corresponding to the intersection part.

3. Calculate`width` and `height` corresponding to the intersection area (overlapping region):

```
width = (x2 - x1)
height = (y2 - y1)
```

4. Calculate the area of overlap (`area_overlap`):

```
if (width<0) or (height <0):
    return 0.0
area_overlap = width * height
```

Note that, in the preceding code, we specify that if the width or height corresponding to the overlapping region is less than 0, the area of intersection is 0. Otherwise, we calculate the area of overlap (intersection) similar to the way a rectangle's area is calculated – width multiplied by the height.

5. Calculate the combined area corresponding to the two bounding boxes:

```
area_a = (boxA[2] - boxA[0]) * (boxA[3] - boxA[1])
area_b = (boxB[2] - boxB[0]) * (boxB[3] - boxB[1])
area_combined = area_a + area_b - area_overlap
```

In the preceding code, we have calculated the combined area of the two bounding boxes – `area_a` and `area_b`, and then subtracted the overlapping area while calculating `area_combined` as `area_overlap` is counted twice, once when calculating `area_a` and then when calculating `area_b`.

6. Calculate the IoU and return it:

```
iou = area_overlap / (area_combined+epsilon)
return iou
```

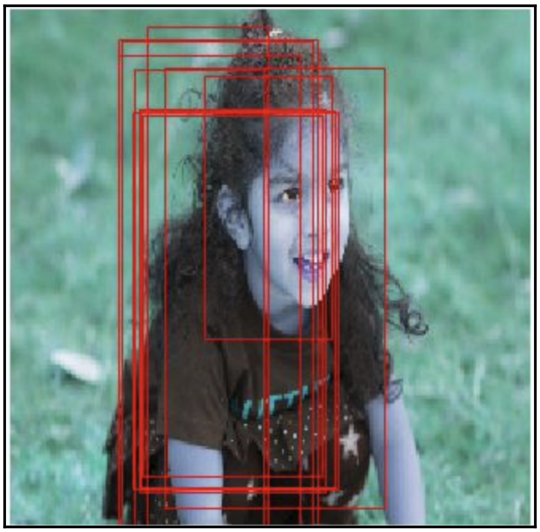
In the preceding code, we calculated `iou` as the ratio of the area of overlap (`area_overlap`) over the area of the combined region (`area_combined`) and returning it.

So far, we have learned about creating ground truth and calculating IoU, which helps in preparing training data. Next, the object detection models will come in handy in detecting objects in the image. Finally, we will calculate model performance and infer on a new image.

We will hold off on building a model until the forthcoming sections as training a model is more involved and we would also have to learn a few more components before we train it. In the next section, we will learn about non-max suppression, which helps in shortlisting from the different possible predicted bounding boxes around an object when inferring using the trained model on a new image.

Non-max suppression

Imagine a scenario where multiple region proposals are generated and significantly overlap one another. Essentially, all the predicted bounding box coordinates (offsets to region proposals) significantly overlap one another. For example, let's consider the following image, where multiple region proposals are generated for the person in the image:



In the preceding image, I ask you to identify the box among the many region proposals that we will consider as the one containing an object and the boxes that we will discard. Non-max suppression comes in handy in such a scenario. Let's unpack the term "Non-max suppression."

Non-max refers to the boxes that do not contain the highest probability of containing an object, and **suppression** refers to us discarding those boxes that do not contain the highest probabilities of containing an object. In non-max suppression, we identify the bounding box that has the highest probability and discard all the other bounding boxes that have an IoU greater than a certain threshold with the box containing the highest probability of containing an object.

In PyTorch, non-max suppression is performed using the `nms` function in the `torchvision.ops` module. The `nms` function takes the bounding box coordinates, the confidence of the object in the bounding box, and the threshold of IoU across bounding boxes, to identify the bounding boxes to be retained. You will be leveraging the `nms` function when predicting object classes and bounding boxes of objects in a new image in both the *Training R-CNN-based custom object detectors* and *Training Fast R-CNN-based custom object detectors* sections in *steps 19* and *16*, respectively.

Mean average precision

So far, we have looked at getting an output that comprises a bounding box around each object within the image and the class corresponding to the object within the bounding box. Now comes the next question: How do we quantify the accuracy of the predictions coming from our model?

mAP comes to the rescue in such a scenario. Before we try to understand mAP, let's first understand precision, then average precision, and finally, mAP:

- **Precision**: Typically, we calculate precision as:

$$Precision = \frac{True\ positives}{(True\ positives + False\ positives)}$$

  A true positive refers to the bounding boxes that predicted the correct class of objects and that have an IoU with the ground truth that is greater than a certain threshold. A false positive refers to the bounding boxes that predicted the class incorrectly or have an overlap that is less than the defined threshold with the ground truth. Furthermore, if there are multiple bounding boxes that are identified for the same ground truth bounding box, only one box can get into a true positive, and everything else gets into a false positive.

- **Average Precision:** Average precision is the average of precision values calculated at various IoU thresholds.
- **mAP:** mAP is the average of precision values calculated at various IoU threshold values across all the classes of objects present within the dataset.

So far, we have learned about preparing a training dataset for our model, performing non-max suppression on the model's predictions, and calculating its accuracies. In the following sections, we will learn about training a model (R-CNN-based and Fast R-CNN-based) to detect objects in new images.

Training R-CNN-based custom object detectors

R-CNN stands for **Region-based Convolutional Neural Network**. **Region-based** within R-CNN stands for the region proposals. Region proposals are used to identify objects within an image. Note that R-CNN assists in identifying both the objects present in the image and the location of objects within the image.

In the following sections, we will learn about the working details of R-CNN before training it on our custom dataset.

Working details of R-CNN

Let's get an idea of R-CNN-based object detection at a high level using the following diagram:

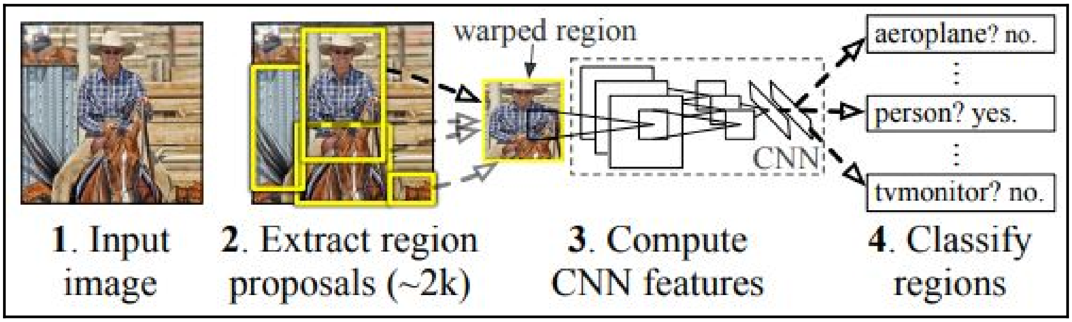
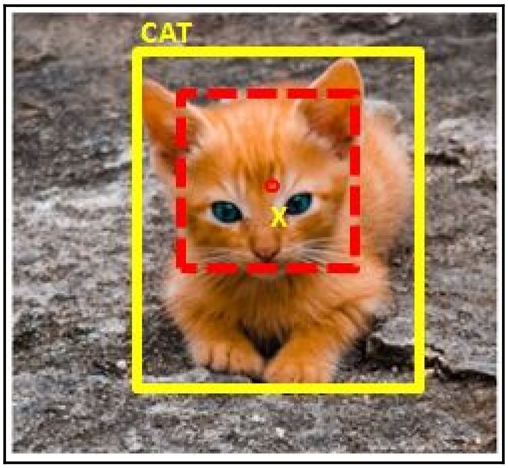


Image source: https://arxiv.org/pdf/1311.2524.pdf

We perform the following steps when leveraging the R-CNN technique for object detection:

1. Extract region proposals from an image:
    - Ensure that we extract a high number of proposals to not miss out on any potential object within the image.
2. Resize (warp) all the extracted regions to get images of the same size.
3. Pass the resized region proposals through a network:
    - Typically, we pass the resized region proposals through a pretrained model such as VGG16 or ResNet50 and extract the features in a fully connected layer.
4. Create data for model training, where the input is features extracted by passing the region proposals through a pretrained model, and the outputs are the class corresponding to each region proposal and the offset of the region proposal from the ground truth corresponding to the image:
    - If a region proposal has an IoU greater than a certain threshold with the object, we prepare training data in such a way that the region is responsible for predicting the class of object it is overlapping with and also the offset of region proposal with the ground truth bounding box that contains the object of interest.

    A sample as a result of creating a bounding box offset and a ground truth class for a region proposal is as follows:



In the preceding image, o (in red) represents the center of the region proposal (dotted bounding box) and x represents the center of the ground truth bounding box (solid bounding box) corresponding to the cat class. We calculate the offset between the region proposal bounding box and the ground truth bounding box as the difference between center co-ordinates of the two bounding boxes (dx, dy) and the difference between the height and width of the bounding boxes (dw, dh).

5. Connect two output heads, one corresponding to the class of image and the other corresponding to the offset of region proposal with the ground truth bounding box to extract the fine bounding box on the object:
    - This exercise would be similar to the use case where we predicted gender (categorical variable, analogous to the class of object in this case study) and age (continuous variable, analogous to the offsets to be done on top of region proposals) based on the image of the face of a person in the previous chapter.
6. Train the model post, writing a custom loss function that minimizes both object classification error and the bounding box offset error.

Note that the loss function that we will minimize differs from the loss function that is optimized in the original paper. We are doing this to reduce the complexity associated with building R-CNN and Fast R-CNN from scratch. Once the reader is familiar with how the model works and can build a model using the following code, we highly encourage them to implement the original paper from scratch.

In the next section, we will learn about fetching datasets and creating data for training. In the section after that, we will learn about designing the model and training it before predicting the class of objects present and their bounding boxes in a new image.

Implementing R-CNN for object detection on a custom dataset

So far, we have a theoretical understanding of how R-CNN works. In this section, we will learn about creating data for training. This process involves the following steps:

1. Downloading the dataset
2. Preparing the dataset
3. Defining the region proposals extraction and IoU calculation functions

4. Creating the training data
    - Creating input data for the model
        - Resizing the region proposals
        - Passing them through a pretrained model to fetch the fully connected layer values
    - Creating output data for the model
        - Labeling each region proposal with a class or background label
        - Defining the offset of the region proposal from the ground truth if the region proposal corresponds to an object and not background
5. Defining and training the model
6. Predicting on new images

Let's get started with coding in the following sections.

Downloading the dataset

For the scenario of object detection, we will download the data from the Google Open Images v6 dataset (available at `https://storage.googleapis.com/openimages/v5/test-annotations-bbox.csv`). However, in code, we will work on only those images that are of a bus or a truck to ensure that we can train images (as you will shortly notice the memory issues associated with using `selectivesearch`). We will expand the number of classes (more classes in addition to bus and truck) we will train on in `Chapter 10`, *Applications of Object Detection and Segmentation*.

> The following code is available as `Training_RCNN.ipynb` in the `Chapter07` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from and is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of the various code components from text.

1. Import the relevant packages to download files that contain images and their ground truths:

```
!pip install -q --upgrade selectivesearch torch_snippets
from torch_snippets import *
import selectivesearch
from google.colab import files
```

```
files.upload() # upload kaggle.json file
!mkdir -p ~/.kaggle
!mv kaggle.json ~/.kaggle/
!ls ~/.kaggle
!chmod 600 /root/.kaggle/kaggle.json
!kaggle datasets download -d sixhky/open-images-bus-trucks/
!unzip -qq open-images-bus-trucks.zip
from torchvision import transforms, models, datasets
from torch_snippets import Report
from torchvision.ops import nms
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

Once we execute the preceding code, we would have the images and their corresponding ground truths stored in a CSV file available.

Preparing the dataset

Now that we have downloaded the dataset, we will prepare the dataset. This involves the following steps:

1. Fetching each image and its corresponding class and bounding box values
2. Fetching the region proposals within each image, their corresponding IoU, and the delta by which the region proposal is to be corrected with respect to the ground truth
3. Assigning numeric labels for each class (where we have an additional background class (besides the bus and truck classes) where IoU with the ground truth bounding box is below a threshold)
4. Resizing each region proposal to a common size in order to pass them to a network

By the end of this exercise, we will have resized crops of region proposals, along with assigning the ground truth class to each region proposal, and calculated the offset of the region proposal in relation to the ground truth bounding box. We will continue coding from where we left off in the preceding section:

1. Specify the location of images and read the ground truths present in the CSV file that we downloaded:

```
IMAGE_ROOT = 'images/images'
DF_RAW = pd.read_csv('df.csv')
print(DF_RAW.head())
```

A sample of the preceding data frame is as follows:

	ImageID	Source	LabelName	Confidence	XMin	XMax	YMin	YMax	IsOccluded	IsTruncated	IsGroupOf	IsDepiction	IsInside
20	002f8241bd829022	xclick	Bus	1	0.257812	0.515625	0.485417	0.891667	1	0	0	0	0
21	002f8241bd829022	xclick	Bus	1	0.535937	0.907813	0.347917	0.997917	1	1	0	0	0
191	013b99371484d3d5	xclick	Bus	1	0.154688	0.920312	0.102083	0.872917	0	0	0	0	0
322	01f8886b50a031a1	xclick	Truck	1	0.012821	0.987179	0.000000	0.969512	0	0	0	0	0
405	02717d30304f4849	xclick	Bus	1	0.106250	0.926562	0.266667	0.635417	0	0	0	0	0

Note that `XMin`, `XMax`, `YMin`, and `YMax` correspond to the ground truth of the bounding box of the image. Furthermore, `LabelName` provides the class of image.

2. Define a class that returns the image and its corresponding class and ground truth along with the file path of the image:

    - Pass the data frame (`df`) and the path to the folder containing images (`image_folder`) as input to the `__init__` method and fetch the unique `ImageID` values present in the data frame (`self.unique_images`). We do so, as an image can contain a multiple number of objects and so multiple rows can correspond to the same `ImageID` value:

```
class OpenImages(Dataset):
    def __init__(self, df, image_folder=IMAGE_ROOT):
        self.root = image_folder
        self.df = df
        self.unique_images = df['ImageID'].unique()
    def __len__(self): return len(self.unique_images)
```

    - Define the `__getitem__` method, where we fetch the image (`image_id`) corresponding to an index (`ix`), fetch its bounding box co-ordinates (`boxes`), `classes`, and return the image, bounding box, class, and image path:

```
    def __getitem__(self, ix):
        image_id = self.unique_images[ix]
        image_path = f'{self.root}/{image_id}.jpg'
        # Convert BGR to RGB
        image = cv2.imread(image_path, 1)[...,::-1]
        h, w, _ = image.shape
        df = self.df.copy()
        df = df[df['ImageID'] == image_id]
        boxes = df['XMin,YMin,XMax,YMax'.split(',')].values
        boxes = (boxes*np.array([w,h,w,h])).astype(np.uint16)\
```

```
                                                          .tolist()
        classes = df['LabelName'].values.tolist()
        return image, boxes, classes, image_path
```

3. Inspect a sample image and its corresponding class and bounding box ground truth:

```
ds = OpenImages(df=DF_RAW)
im, bbs, clss, _ = ds[9]
show(im, bbs=bbs, texts=clss, sz=10)
```

The preceding code results in the following:

4. Define the `extract_iou` and `extract_candidates` functions:

```
def extract_candidates(img):
    img_lbl,regions = selectivesearch.selective_search(img, \
                                       scale=200, min_size=100)
    img_area = np.prod(img.shape[:2])
    candidates = []
    for r in regions:
        if r['rect'] in candidates: continue
        if r['size'] < (0.05*img_area): continue
        if r['size'] > (1*img_area): continue
        x, y, w, h = r['rect']
        candidates.append(list(r['rect']))
    return candidates
def extract_iou(boxA, boxB, epsilon=1e-5):
    x1 = max(boxA[0], boxB[0])
    y1 = max(boxA[1], boxB[1])
    x2 = min(boxA[2], boxB[2])
    y2 = min(boxA[3], boxB[3])
    width = (x2 - x1)
    height = (y2 - y1)
```

```
    if (width<0) or (height <0):
        return 0.0
    area_overlap = width * height
    area_a = (boxA[2] - boxA[0]) * (boxA[3] - boxA[1])
    area_b = (boxB[2] - boxB[0]) * (boxB[3] - boxB[1])
    area_combined = area_a + area_b - area_overlap
    iou = area_overlap / (area_combined+epsilon)
    return iou
```

By now, we have defined all the functions necessary to prepare data and initialize data loaders. In the next section, we will fetch region proposals (input regions to the model) and the ground truth of the bounding box offset along with the class of object (expected output).

Fetching region proposals and the ground truth of offset

In this section, we will learn about creating the input and output values corresponding to our model. The input constitutes the candidates that are extracted using the `selectivesearch` method and the output constitutes the class corresponding to candidates and the offset of the candidate with respect to the bounding box it overlaps the most with if the candidate contains an object. We will continue coding from where we ended in the preceding section:

1. Initialize empty lists to store file paths (`FPATHS`), ground truth bounding boxes (`GTBBS`), classes (`CLSS`) of objects, the delta offset of a bounding box with region proposals (`DELTAS`), region proposal locations (`ROIS`), and the IoU of region proposals with ground truths (`IOUS`):

```
FPATHS, GTBBS, CLSS, DELTAS, ROIS, IOUS = [],[],[],[],[],[]
```

2. Loop through the dataset and populate the lists initialized above:

   - For this exercise, we can use all the data points for training or illustrate with just the first 500 data points. You can choose between either of the two, which dictates the training time and training accuracy (the greater the data points, the greater the training time and accuracy):

```
N = 500
for ix, (im, bbs, labels, fpath) in enumerate(ds):
    if(ix==N):
        break
```

In the preceding code, we are specifying that we will work on 500 images.

- Extract candidates from each image (`im`) in absolute pixel values (note that `XMin`, `Xmax`, `YMin`, and `YMax` are available as a proportion of the shape of images in the downloaded data frame) using the `extract_candidates` function and convert the extracted regions coordinates from an (x,y,w,h) system to an (x,y,x+w,y+h) system:

```
H, W, _ = im.shape
candidates = extract_candidates(im)
candidates = np.array([(x,y,x+w,y+h) \
                       for x,y,w,h in candidates])
```

- Initialize `ious`, `rois`, `deltas`, and `clss` as lists that store `iou` for each candidate, region proposal location, bounding box offset, and class corresponding to every candidate for each image. We will go through all the proposals from SelectiveSearch and store those with a high IOU as bus/truck proposals (whichever is the class in labels) and the rest as background proposals:

```
ious, rois, clss, deltas = [], [], [], []
```

- Store the IoU of all candidates with respect to all ground truths for an image where `bbs` is the ground truth bounding box of different objects present in the image and `candidates` are the region proposal candidates obtained in the previous step:

```
ious = np.array([[extract_iou(candidate, _bb_) for \
            candidate in candidates] for _bb_ in bbs]).T
```

- Loop through each candidate and store the XMin (`cx`), YMin (`cy`), XMax (`cX`), and YMax (`cY`) values of a candidate:

```
for jx, candidate in enumerate(candidates):
    cx,cy,cX,cY = candidate
```

- Extract the IoU corresponding to the candidate with respect to all the ground truth bounding boxes that were already calculated when fetching the list of lists of `ious`:

```
    candidate_ious = ious[jx]
```

- Find the index of a candidate (`best_iou_at`) that has the highest IoU and the corresponding ground truth (`best_bb`):

```
best_iou_at = np.argmax(candidate_ious)
best_iou = candidate_ious[best_iou_at]
best_bb = _x,_y,_X,_Y = bbs[best_iou_at]
```

- If IoU (`best_iou`) is greater than a threshold (0.3), we assign the label of class corresponding to the candidate, and the background otherwise:

```
if best_iou > 0.3: clss.append(labels[best_iou_at])
else : clss.append('background')
```

- Fetch the offsets needed (`delta`) to transform the current proposal into the candidate that is the best region proposal (which is the ground truth bounding box) – `best_bb`, in other words, how much should the left, right, top, and bottom margins of the current proposal be adjusted so that it aligns exactly with `best_bb` from the ground truth:

```
delta = np.array([_x-cx, _y-cy, _X-cX, _Y-cY]) /\
            np.array([W,H,W,H])
deltas.append(delta)
rois.append(candidate / np.array([W,H,W,H]))
```

- Append the file paths, IoU, roi, class delta, and ground truth bounding boxes:

```
FPATHS.append(fpath)
IOUS.append(ious)
ROIS.append(rois)
CLSS.append(clss)
DELTAS.append(deltas)
GTBBS.append(bbs)
```

- Fetch the image path names and store all the information obtained, `FPATHS`, `IOUS`, `ROIS`, `CLSS`, `DELTAS`, and `GTBBS`, in a list of lists:

```
FPATHS = [f'{IMAGE_ROOT}/{stem(f)}.jpg' for f in FPATHS]
FPATHS, GTBBS, CLSS, DELTAS, ROIS = [item for item in \
                                     [FPATHS, GTBBS, \
                                      CLSS, DELTAS, ROIS]]
```

Note that, so far, classes are available as the name of the class. Now, we will convert them into their corresponding indices so that a background class has a class index of 0, a bus class has a class index of 1, and a truck class has a class index of 2.

3. Assign indices to each class:

```
targets = pd.DataFrame(flatten(CLSS), columns=['label'])
label2target = {l:t for t,l in \
                enumerate(targets['label'].unique())}
target2label = {t:l for l,t in label2target.items()}
background_class = label2target['background']
```

So far, we have assigned a class to each region proposal and also created the other ground truth of the bounding box offset. In the next section, we will fetch the dataset and the data loaders corresponding to the information obtained (`FPATHS`, `IOUS`, `ROIS`, `CLSS`, `DELTAS`, and `GTBBS`).

Creating the training data

So far, we have fetched data, region proposals across all images, prepared the ground truths of the class of object present within each region proposal, and the offset corresponding to each region proposal that has a high overlap (IoU) with the object in the corresponding image.

In this section, we will prepare a dataset class based on the ground truth of region proposals that are obtained by the end of *step 8* and create data loaders from it. Next, we will normalize each region proposal by resizing them to the same shape and scaling them. We will continue coding from where we left off in the preceding section:

1. Define the function to normalize an image:

```
normalize= transforms.Normalize(mean=[0.485, 0.456, 0.406], \
                                std=[0.229, 0.224, 0.225])
```

2. Define a function (`preprocess_image`) to preprocess the image (`img`), where we switch channels, normalize the image, and register it with the device:

```
def preprocess_image(img):
    img = torch.tensor(img).permute(2,0,1)
    img = normalize(img)
    return img.to(device).float()
```

- Define the function to the class `decode` prediction:

```
def decode(_y):
    _, preds = _y.max(-1)
    return preds
```

3. Define the dataset (`RCNNDataset`) using the preprocessed region proposals along with the ground truths obtained in the previous step (*step 7*):

```
class RCNNDataset(Dataset):
    def __init__(self, fpaths, rois, labels, deltas, gtbbs):
        self.fpaths = fpaths
        self.gtbbs = gtbbs
        self.rois = rois
        self.labels = labels
        self.deltas = deltas
    def __len__(self): return len(self.fpaths)
```

- Fetch the crops as per the region proposals, along with the other ground truths related to class and the bounding box offset:

```
    def __getitem__(self, ix):
        fpath = str(self.fpaths[ix])
        image = cv2.imread(fpath, 1)[...,::-1]
        H, W, _ = image.shape
        sh = np.array([W,H,W,H])
        gtbbs = self.gtbbs[ix]
        rois = self.rois[ix]
        bbs = (np.array(rois)*sh).astype(np.uint16)
        labels = self.labels[ix]
        deltas = self.deltas[ix]
        crops = [image[y:Y,x:X] for (x,y,X,Y) in bbs]
        return image,crops,bbs,labels,deltas,gtbbs,fpath
```

- Define `collate_fn`, which performs the resizing and normalizing (`preprocess_image`) of an image of a crop:

```
    def collate_fn(self, batch):
        input, rois, rixs, labels, deltas =[],[],[],[],[]
        for ix in range(len(batch)):
            image, crops, image_bbs, image_labels, \
                image_deltas, image_gt_bbs, \
                image_fpath = batch[ix]
            crops = [cv2.resize(crop, (224,224)) \
                     for crop in crops]
            crops = [preprocess_image(crop/255.)[None] \
                     for crop in crops]
            input.extend(crops)
```

```
                labels.extend([label2target[c] \
                               for c in image_labels])
                deltas.extend(image_deltas)
            input = torch.cat(input).to(device)
            labels = torch.Tensor(labels).long().to(device)
            deltas = torch.Tensor(deltas).float().to(device)
            return input, labels, deltas
```

4. Create the training and validation datasets and data loaders:

```
n_train = 9*len(FPATHS)//10
train_ds = RCNNDataset(FPATHS[:n_train], ROIS[:n_train], \
                       CLSS[:n_train], DELTAS[:n_train], \
                       GTBBS[:n_train])
test_ds = RCNNDataset(FPATHS[n_train:], ROIS[n_train:], \
                      CLSS[n_train:], DELTAS[n_train:], \
                      GTBBS[n_train:])

from torch.utils.data import TensorDataset, DataLoader
train_loader = DataLoader(train_ds, batch_size=2, \
                          collate_fn=train_ds.collate_fn, \
                          drop_last=True)
test_loader = DataLoader(test_ds, batch_size=2, \
                         collate_fn=test_ds.collate_fn, \
                         drop_last=True)
```

So far, we have learned about preparing data. Next, we will learn about defining and training the model that predicts the class and offset to be made to the region proposal to fit a tight bounding box around objects in the image.

R-CNN network architecture

Now that we have prepared the data, in this section, we will learn about building a model that can predict both the class of region proposal and the offset corresponding to it in order to draw a tight bounding box around the object in the image. The strategy we adopt is as follows:

1. Define a VGG backbone.
2. Fetch the features post passing the normalized crop through a pretrained model.
3. Attach a linear layer with sigmoid activation to the VGG backbone to predict the class corresponding to the region proposal.
4. Attach an additional linear layer to predict the four bounding box offsets.

5. Define the loss calculations for each of the two outputs (one to predict class and the other to predict the four bounding box offsets).
6. Train the model that predicts both the class of region proposal and the four bounding box offsets.

Execute the following code. We will continue coding from where we ended in the preceding section:

1. Define a VGG backbone:

```
vgg_backbone = models.vgg16(pretrained=True)
vgg_backbone.classifier = nn.Sequential()
for param in vgg_backbone.parameters():
    param.requires_grad = False
vgg_backbone.eval().to(device)
```

2. Define the `RCNN` network module:

   - Define the class:

```
class RCNN(nn.Module):
    def __init__(self):
        super().__init__()
```

   - Define the backbone (`self.backbone`) and how we calculate the class score (`self.cls_score`) and the bounding box offset values (`self.bbox`):

```
        feature_dim = 25088
        self.backbone = vgg_backbone
        self.cls_score = nn.Linear(feature_dim, \
                                    len(label2target))
        self.bbox = nn.Sequential(
                      nn.Linear(feature_dim, 512),
                      nn.ReLU(),
                      nn.Linear(512, 4),
                      nn.Tanh(),
                    )
```

   - Define the loss functions corresponding to class prediction (`self.cel`) and bounding box offset regression (`self.sl1`):

```
        self.cel = nn.CrossEntropyLoss()
        self.sl1 = nn.L1Loss()
```

- Define the feed-forward method where we pass the image through a VGG backbone (`self.backbone`) to fetch features (`feat`), which are further passed through the methods corresponding to classification and bounding box regression to fetch the probabilities across classes (`cls_score`) and the bounding box offsets (`bbox`):

```
def forward(self, input):
    feat = self.backbone(input)
    cls_score = self.cls_score(feat)
    bbox = self.bbox(feat)
    return cls_score, bbox
```

- Define the function to calculate loss (`calc_loss`). Note that we do not calculate regression loss corresponding to offsets if the actual class is of the background:

```
def calc_loss(self, probs, _deltas, labels, deltas):
    detection_loss = self.cel(probs, labels)
    ixs, = torch.where(labels != 0)
    _deltas = _deltas[ixs]
    deltas = deltas[ixs]
    self.lmb = 10.0
    if len(ixs) > 0:
        regression_loss = self.sl1(_deltas, deltas)
        return detection_loss + self.lmb *\
            regression_loss, detection_loss.detach(), \
            regression_loss.detach()
    else:
        regression_loss = 0
        return detection_loss + self.lmb *\
            regression_loss, detection_loss.detach(), \
            regression_loss
```

With the model class in place, we now define the functions to train on a batch of data and predict on validation data.

3. Define the `train_batch` function:

```
def train_batch(inputs, model, optimizer, criterion):
    input, clss, deltas = inputs
    model.train()
    optimizer.zero_grad()
    _clss, _deltas = model(input)
    loss, loc_loss, regr_loss = criterion(_clss, _deltas, \
                                          clss, deltas)
    accs = clss == decode(_clss)
    loss.backward()
```

```
        optimizer.step()
        return loss.detach(), loc_loss, regr_loss, \
            accs.cpu().numpy()
```

4. Define the `validate_batch` function:

```
    @torch.no_grad()
    def validate_batch(inputs, model, criterion):
        input, clss, deltas = inputs
        with torch.no_grad():
            model.eval()
            _clss,_deltas = model(input)
            loss,loc_loss,regr_loss = criterion(_clss, _deltas, \
                                                      clss, deltas)
            _, _clss = _clss.max(-1)
            accs = clss == _clss
        return _clss,_deltas,loss.detach(),loc_loss, regr_loss, \
             accs.cpu().numpy()
```

5. Now, let's create an object of the model, fetch the loss criterion, and then define the optimizer and the number of epochs:

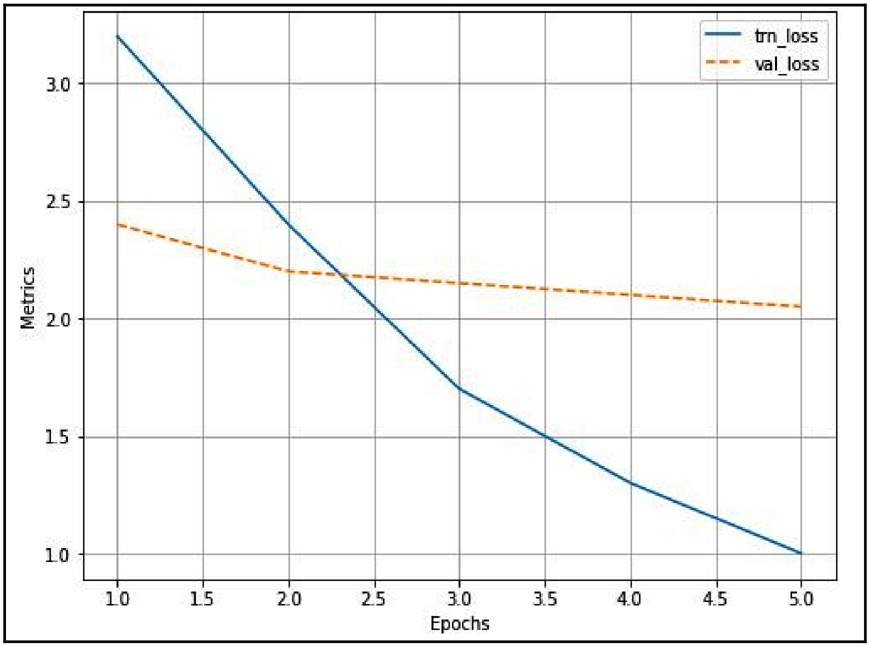
```
    rcnn = RCNN().to(device)
    criterion = rcnn.calc_loss
    optimizer = optim.SGD(rcnn.parameters(), lr=1e-3)
    n_epochs = 5
    log = Report(n_epochs)
```

6. We now train the model over increasing epochs:

```
    for epoch in range(n_epochs):

        _n = len(train_loader)
        for ix, inputs in enumerate(train_loader):
            loss, loc_loss,regr_loss,accs = train_batch(inputs, \
                                      rcnn, optimizer, criterion)
            pos = (epoch + (ix+1)/_n)
            log.record(pos, trn_loss=loss.item(), \
                       trn_loc_loss=loc_loss, \
                       trn_regr_loss=regr_loss, \
                       trn_acc=accs.mean(), end='\r')
        _n = len(test_loader)
        for ix,inputs in enumerate(test_loader):
            _clss, _deltas, loss, \
            loc_loss, regr_loss, \
            accs = validate_batch(inputs, rcnn, criterion)
            pos = (epoch + (ix+1)/_n)
            log.record(pos, val_loss=loss.item(), \
```

```
                                val_loc_loss=loc_loss, \
                                val_regr_loss=regr_loss, \
                                val_acc=accs.mean(), end='\r')

# Plotting training and validation metrics
log.plot_epochs('trn_loss,val_loss'.split(','))
```

The plot of overall loss across training and validation data is as follows:



Now that we have trained a model, we will use it to predict on a new image in the next section.

Predict on a new image

In this section, we will leverage the model trained so far to predict and draw bounding boxes around objects and the corresponding class of object within the predicted bounding box on new images. The strategy we adopt is as follows:

1. Extract region proposals from the new image.
2. Resize and normalize each crop.

3. Feed-forward the processed crops to make predictions of class and the offsets.
4. Perform non-max suppression to fetch only those boxes that have the highest confidence of containing an object.

We execute the preceding strategy through a function that takes an image as input and a ground truth bounding box (this is used only so that we compare the ground truth and the predicted bounding box). We will continue coding from where we left off in the preceding section:

1. Define the `test_predictions` function to predict on a new image:

    - The function takes `filename` as input:

    ```
    def test_predictions(filename, show_output=True):
    ```

    - Read the image and extract candidates:

    ```
        img = np.array(cv2.imread(filename, 1)[...,::-1])
        candidates = extract_candidates(img)
        candidates = [(x,y,x+w,y+h) for x,y,w,h in candidates]
    ```

    - Loop through the candidates to resize and preprocess the image:

    ```
        input = []
        for candidate in candidates:
            x,y,X,Y = candidate
            crop = cv2.resize(img[y:Y,x:X], (224,224))
            input.append(preprocess_image(crop/255.)[None])
        input = torch.cat(input).to(device)
    ```

    - Predict the class and offset:

    ```
        with torch.no_grad():
            rcnn.eval()
            probs, deltas = rcnn(input)
            probs = torch.nn.functional.softmax(probs, -1)
            confs, clss = torch.max(probs, -1)
    ```

    - Extract the candidates that do not belong to the background class and sum up the candidates with the predicted bounding box offset values:

    ```
        candidates = np.array(candidates)
        confs,clss,probs,deltas =[tensor.detach().cpu().numpy() \
                                      for tensor in [confs, \
                                             clss, probs, deltas]]
    ```

```
    ixs = clss!=background_class
    confs, clss,probs,deltas,candidates = [tensor[ixs] for \
           tensor in [confs,clss, probs, deltas,candidates]]
    bbs = (candidates + deltas).astype(np.uint16)
```

- Use non-max suppression `nms` to eliminate near-duplicate bounding boxes (pairs of boxes that have an IoU greater than 0.05 are considered duplicates in this case). Among the duplicated boxes, we pick that box with the highest confidence and discard the rest:

```
    ixs = nms(torch.tensor(bbs.astype(np.float32)), \
                torch.tensor(confs), 0.05)
    confs,clss,probs,deltas,candidates,bbs = [tensor[ixs] \
                                            for tensor in \
                            [confs, clss, probs, deltas, \
                            candidates, bbs]]
    if len(ixs) == 1:
        confs, clss, probs, deltas, candidates, bbs = \
                [tensor[None] for tensor in [confs, clss,
                            probs, deltas, candidates, bbs]]
```

- Fetch the bounding box with the highest confidence:

```
    if len(confs) == 0 and not show_output:
        return (0,0,224,224), 'background', 0
    if len(confs) > 0:
        best_pred = np.argmax(confs)
        best_conf = np.max(confs)
        best_bb = bbs[best_pred]
        x,y,X,Y = best_bb
```

- Plot the image along with the predicted bounding box:

```
    _, ax = plt.subplots(1, 2, figsize=(20,10))
    show(img, ax=ax[0])
    ax[0].grid(False)
    ax[0].set_title('Original image')
    if len(confs) == 0:
        ax[1].imshow(img)
        ax[1].set_title('No objects')
        plt.show()
        return
    ax[1].set_title(target2label[clss[best_pred]])
    show(img, bbs=bbs.tolist(),
        texts=[target2label[c] for c in clss.tolist()],
        ax=ax[1], title='predicted bounding box and class')
    plt.show()
    return (x,y,X,Y),target2label[clss[best_pred]],best_conf
```

2. Execute the preceding function on a new image:

```
image, crops, bbs, labels, deltas, gtbbs, fpath = test_ds[7]
test_predictions(fpath)
```

The preceding code generates the following images:

From the preceding diagram, we can see that the prediction of the class of an image is accurate and the bounding box prediction is decent, too. Note that it took ~1.5 seconds to generate a prediction for the preceding image.

All of this time is consumed in generating region proposals, resizing each region proposal, passing them through a VGG backbone, and generating predictions using the defined model. Most of the time, however, is spent in passing each proposal through a VGG backbone. In the next section, we will learn about getting around this "passing each proposal to VGG" problem by using the Fast R-CNN architecture-based model.

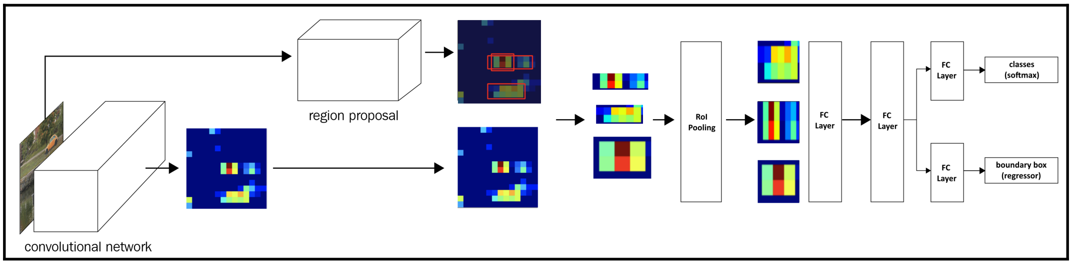
Training Fast R-CNN-based custom object detectors

One of the major drawbacks of R-CNN is that it takes considerable time to generate predictions, as generating region proposals for each image, resizing the crops of regions, and extracting features corresponding to **each crop** (region proposal), constitute the bottleneck.

Fast R-CNN gets around this problem by passing the **entire image** through the pretrained model to extract features and then fetching the region of features that correspond to the region proposals (which are obtained from `selectivesearch`) of the original image. In the following sections, we will learn about the working details of Fast R-CNN before training it on our custom dataset.

Working details of Fast R-CNN

Let's understand Fast R-CNN through the following diagram:



Let's understand the preceding diagram through the following steps:

1. Pass the image through a pretrained model to extract features prior to the flattening layer; let's call the output as feature maps.
2. Extract region proposals corresponding to the image.
3. Extract the feature map area corresponding to the region proposals (note that when an image is passed through a VGG16 architecture, the image is downscaled by 32 at the output as there are 5 pooling operations performed. Thus, if a region exists with a bounding box of (40,32,200,240) in the original image, the feature map corresponding to the bounding box of (5,4,25,30) would correspond to the exact same region).
4. Pass the feature maps corresponding to region proposals through the RoI (Region of Interest) pooling layer one at a time so that all feature maps of region proposals have a similar shape. This is a replacement for the warping that was executed in the R-CNN technique.
5. Pass the RoI pooling layer output value through a fully connected layer.
6. Train the model to predict the class and offsets corresponding to each region proposal.

Note that the big difference between R-CNN and Fast R-CNN is that, in R-CNN, we are passing the crops (resized region proposals) through the pretrained model one at a time, while in Fast R-CNN, we are cropping the feature map (which is obtained by passing the whole image through a pretrained model) corresponding to each region proposal and thereby avoiding the need to pass each resized region proposal through the pretrained model.

Now armed with an understanding of how Fast R-CNN works, in the next section, we will build the model using the same dataset that we leveraged in the R-CNN section.

Implementing Fast R-CNN for object detection on a custom dataset

In this section, we will work toward training our custom object detector using Fast R-CNN. Furthermore, so as to remain succinct, we provide only the additional or the changed code in this section (you should run all the code until *step 2* in the *Creating the training data* sub-section in the previous section of R-CNN):

To maintain brevity, we have only provided the additional code to train Fast R-CNN. The full code is available as *`Training_Fast_R_CNN.ipynb`* in the `Chapter07` folder of this book's GitHub repository.

1. Create an `FRCNNDataset` class that returns images, labels, ground truths, region proposals, and the delta corresponding to each region proposal:

```
class FRCNNDataset(Dataset):
    def __init__(self, fpaths, rois, labels, deltas, gtbbs):
        self.fpaths = fpaths
        self.gtbbs = gtbbs
        self.rois = rois
        self.labels = labels
        self.deltas = deltas
    def __len__(self): return len(self.fpaths)
    def __getitem__(self, ix):
        fpath = str(self.fpaths[ix])
        image = cv2.imread(fpath, 1)[...,::-1]
        gtbbs = self.gtbbs[ix]
        rois = self.rois[ix]
        labels = self.labels[ix]
        deltas = self.deltas[ix]
```

```
        assert len(rois) == len(labels) == len(deltas), \
            f'{len(rois)}, {len(labels)}, {len(deltas)}'
        return image, rois, labels, deltas, gtbbs, fpath

    def collate_fn(self, batch):
        input, rois, rixs, labels, deltas = [],[],[],[],[]
        for ix in range(len(batch)):
            image, image_rois, image_labels, image_deltas, \
                image_gt_bbs, image_fpath = batch[ix]
            image = cv2.resize(image, (224,224))
            input.append(preprocess_image(image/255.)[None])
            rois.extend(image_rois)
            rixs.extend([ix]*len(image_rois))
            labels.extend([label2target[c] for c in \
                                    image_labels])
            deltas.extend(image_deltas)
        input = torch.cat(input).to(device)
        rois = torch.Tensor(rois).float().to(device)
        rixs = torch.Tensor(rixs).float().to(device)
        labels = torch.Tensor(labels).long().to(device)
        deltas = torch.Tensor(deltas).float().to(device)
        return input, rois, rixs, labels, deltas
```

Note that the preceding code is very similar to what we have learned in the *R-CNN* section, with the only change being that we are returning more information (`rois` and `rixs`).

The `rois` matrix holds information regarding which RoI belongs to which image in the batch. Note that `input` contains multiple images, whereas `rois` is a single list of boxes. We wouldn't know how many rois belong to the first image and how many belong to the second image, and so on. This is where `ridx` comes into the picture. It is a list of indexes. Each integer in the list associates the corresponding bounding box with the appropriate image; for example, if `ridx` is `[0,0,0,1,1,2,3,3,3]`, then we know the first three bounding boxes belong to the first image in the batch, and the next two belong to the second image in the batch.

2. Create training and test datasets:

```
n_train = 9*len(FPATHS)//10
train_ds = FRCNNDataset(FPATHS[:n_train], ROIS[:n_train], \
                        CLSS[:n_train], DELTAS[:n_train], \
                        GTBBS[:n_train])
test_ds = FRCNNDataset(FPATHS[n_train:], ROIS[n_train:], \
                       CLSS[n_train:], DELTAS[n_train:], \
                       GTBBS[n_train:])
```

```
from torch.utils.data import TensorDataset, DataLoader
train_loader = DataLoader(train_ds, batch_size=2, \
                          collate_fn=train_ds.collate_fn, \
                          drop_last=True)
test_loader = DataLoader(test_ds, batch_size=2, \
                         collate_fn=test_ds.collate_fn, \
                         drop_last=True)
```

3. Define a model to train on the dataset:

- First, import the `RoIPool` method present in the `torchvision.ops` class:

```
from torchvision.ops import RoIPool
```

- Define the `FRCNN` network module:

```
class FRCNN(nn.Module):
    def __init__(self):
        super().__init__()
```

- Load the pretrained model and freeze the parameters:

```
        rawnet = torchvision.models.vgg16_bn(pretrained=True)
        for param in rawnet.features.parameters():
            param.requires_grad = True
```

- Extract features until the last layer:

```
        self.seq = nn.Sequential(*list(\
                        rawnet.features.children())[:-1])
```

- Specify that `RoIPool` is to extract a 7 x 7 output.
  Here, `spatial_scale` is the factor by which proposals (which come from the original image) need to be shrunk so that every output has the same shape prior to passing through the flatten layer. Images are 224 x 224 in size, while the feature map is 14 x 14 in size:

```
        self.roipool = RoIPool(7, spatial_scale=14/224)
```

- Define the output heads – `cls_score` and `bbox`:

```
    feature_dim = 512*7*7
    self.cls_score = nn.Linear(feature_dim, \
                                len(label2target))
    self.bbox = nn.Sequential(
                      nn.Linear(feature_dim, 512),
                      nn.ReLU(),
                      nn.Linear(512, 4),
                      nn.Tanh(),
                    )
```

- Define the loss functions:

```
    self.cel = nn.CrossEntropyLoss()
    self.sl1 = nn.L1Loss()
```

- Define the `forward` method, which takes the image, region proposals, and the index of region proposals as input for the network defined earlier:

```
def forward(self, input, rois, ridx):
```

- Pass the `input` image through the pretrained model:

```
    res = input
    res = self.seq(res)
```

- Create a matrix of `rois` as input for `self.roipool`, first by concatenating `ridx` as the first column and the next four columns being the absolute values of the region proposal bounding boxes:

```
    rois = torch.cat([ridx.unsqueeze(-1), rois*224], \
                       dim=-1)
    res = self.roipool(res, rois)
    feat = res.view(len(res), -1)
    cls_score = self.cls_score(feat)
    bbox=self.bbox(feat)#.view(-1,len(label2target),4)
    return cls_score, bbox
```

- Define the loss value calculation (`calc_loss`), just like we did in the *R-CNN* section:

```
def calc_loss(self, probs, _deltas, labels, deltas):
    detection_loss = self.cel(probs, labels)
    ixs, = torch.where(labels != background_class)
    _deltas = _deltas[ixs]
```

```
            deltas = deltas[ixs]
            self.lmb = 10.0
            if len(ixs) > 0:
                regression_loss = self.sl1(_deltas, deltas)
                return detection_loss +\
                    self.lmb * regression_loss, \
                    detection_loss.detach(), \
                    regression_loss.detach()
            else:
                regression_loss = 0
                return detection_loss + \
                    self.lmb * regression_loss, \
                    detection_loss.detach(), \
                    regression_loss
```

4. Define the functions to train and validate on a batch just like we did in the *R-CNN* section:

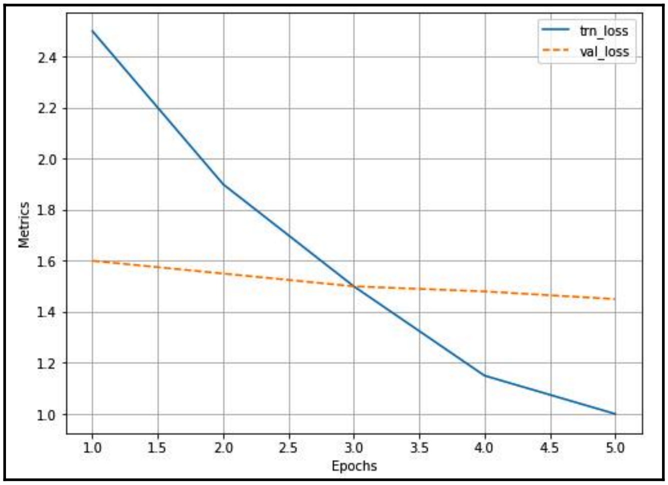
```
def train_batch(inputs, model, optimizer, criterion):
    input, rois, rixs, clss, deltas = inputs
    model.train()
    optimizer.zero_grad()
    _clss, _deltas = model(input, rois, rixs)
    loss, loc_loss, regr_loss = criterion(_clss, _deltas, \
                                            clss, deltas)
    accs = clss == decode(_clss)
    loss.backward()
    optimizer.step()
    return loss.detach(), loc_loss, regr_loss, \
        accs.cpu().numpy()
def validate_batch(inputs, model, criterion):
    input, rois, rixs, clss, deltas = inputs
    with torch.no_grad():
        model.eval()
        _clss,_deltas = model(input, rois, rixs)
        loss, loc_loss,regr_loss = criterion(_clss, _deltas, \
                                            clss, deltas)
        _clss = decode(_clss)
        accs = clss == _clss
    return _clss, _deltas,loss.detach(), loc_loss,regr_loss, \
        accs.cpu().numpy()
```

5. Define and train the model over increasing epochs:

```
frcnn = FRCNN().to(device)
criterion = frcnn.calc_loss
optimizer = optim.SGD(frcnn.parameters(), lr=1e-3)
```

```
n_epochs = 5
log = Report(n_epochs)
for epoch in range(n_epochs):

    _n = len(train_loader)
    for ix, inputs in enumerate(train_loader):
        loss, loc_loss,regr_loss, accs = train_batch(inputs, \
                                         frcnn, optimizer, criterion)
        pos = (epoch + (ix+1)/_n)
        log.record(pos, trn_loss=loss.item(), \
                   trn_loc_loss=loc_loss, \
                   trn_regr_loss=regr_loss, \
                   trn_acc=accs.mean(), end='\r')
    _n = len(test_loader)
    for ix,inputs in enumerate(test_loader):
        _clss, _deltas, loss, \
        loc_loss, regr_loss, accs = validate_batch(inputs, \
                                                frcnn, criterion)
        pos = (epoch + (ix+1)/_n)
        log.record(pos, val_loss=loss.item(), \
                val_loc_loss=loc_loss, \
                val_regr_loss=regr_loss, \
                val_acc=accs.mean(), end='\r')

# Plotting training and validation metrics
log.plot_epochs('trn_loss,val_loss'.split(','))
```

The variation in overall loss is as follows:



6. Define a function to predict on test images:

    - Define the function that takes a filename as input and then reads the file and resizes it to 224 x 224:

```
import matplotlib.pyplot as plt
%matplotlib inline
import matplotlib.patches as mpatches
from torchvision.ops import nms
from PIL import Image
def test_predictions(filename):
    img = cv2.resize(np.array(Image.open(filename)), \
                                (224,224))
```

    - Obtain region proposals and convert them to (x1,y1,x2,y2) format (top-left pixel and bottom-right pixel coordinates), and then convert these values to the ratio of width and height they are present in, in proportion to the image:

```
    candidates = extract_candidates(img)
    candidates = [(x,y,x+w,y+h) for x,y,w,h in candidates]
```

    - Preprocess the image and scale the region of interests (`rois`):

```
    input = preprocess_image(img/255.)[None]
    rois = [[x/224,y/224,X/224,Y/224] for x,y,X,Y in \
                candidates]
```

    - As all proposals belong to the same image, `rixs` will be a list of zeros (as many as the number of proposals):

```
    rixs = np.array([0]*len(rois))
```

    - Forward propagate the input and `rois` through the trained model and get confidences and class scores for each proposal:

```
    rois,rixs = [torch.Tensor(item).to(device) for item in \
                    [rois, rixs]]
    with torch.no_grad():
        frcnn.eval()
        probs, deltas = frcnn(input, rois, rixs)
        confs, clss = torch.max(probs, -1)
```

    - Filter out the background class:

```
    candidates = np.array(candidates)
    confs,clss,probs,deltas=[tensor.detach().cpu().numpy() \
```

```
                                    for tensor in [confs, \
                                         clss, probs, deltas]]
        ixs = clss!=background_class
        confs, clss, probs, deltas,candidates = [tensor[ixs] for \
               tensor in [confs, clss, probs, deltas,candidates]]
        bbs = candidates + deltas
```

- Remove near-duplicate bounding boxes with `nms` and get indices of those proposals in which the models that are highly confident are objects:

```
        ixs = nms(torch.tensor(bbs.astype(np.float32)), \
                    torch.tensor(confs), 0.05)
        confs, clss, probs,deltas,candidates,bbs = [tensor[ixs] \
                                for tensor in [confs,clss,probs, \
                                deltas, candidates, bbs]]
        if len(ixs) == 1:
            confs, clss, probs, deltas, candidates, bbs = \
                        [tensor[None] for tensor in [confs,clss, \
                         probs, deltas, candidates, bbs]]
        bbs = bbs.astype(np.uint16)
```

- Plot the bounding boxes obtained:

```
        _, ax = plt.subplots(1, 2, figsize=(20,10))
        show(img, ax=ax[0])
        ax[0].grid(False)
        ax[0].set_title(filename.split('/')[-1])
        if len(confs) == 0:
            ax[1].imshow(img)
            ax[1].set_title('No objects')
            plt.show()
            return
        else:
            show(img,bbs=bbs.tolist(),texts=[target2label[c] for \
                                    c in clss.tolist()],ax=ax[1])
            plt.show()
```

7. Predict on a test image:

```
    test_predictions(test_ds[29][-1])
```

The preceding code results in the following:

The preceding code executes in 0.5 seconds, which is significantly better than that of R-CNN. However, it is still very slow to be used in real time. This is primarily because we are still using two different models, one to generate region proposals and another to make predictions of class and corrections. In the next chapter, we will learn about having a single model to make predictions, so that inference is quick in a real-time scenario.

Summary

In this chapter, we started with learning about creating a training dataset for the process of object localization and detection. Next, we learned about SelectiveSearch, a region proposal technique that recommends regions based on the similarity of pixels in proximity. We next learned about calculating the IoU metric to understand the goodness of the predicted bounding box around the objects present in the image. We next learned about performing non-max suppression to fetch one bounding box per object within an image before learning about building R-CNN and Fast R-CNN models from scratch. In addition, we learned about the reason why R-CNN is slow and how Fast R-CNN leverages RoI pooling and fetches region proposals from feature maps to make inference faster. Finally, we understood that having region proposals coming from a separate model is resulting in the higher time taken to predict on new images.

In the next chapter, we will learn about some of the modern object detection techniques that are used to make inference on a more real-time basis.

Questions

1. How does a region proposal technique generate proposals?
2. How is IoU calculated if there are multiple objects in an image?
3. Why does R-CNN take a long time to generate predictions?
4. Why is Fast R-CNN faster when compared with R-CNN?
5. How does RoI pooling work?
6. What is the impact of not having multiple layers post the feature map obtained when predicting the bounding box corrections?
7. Why do we have to assign a higher weight to regression loss when calculating overall loss?
8. How does non-max suppression work?

8
Advanced Object Detection

In the previous chapter, we learned about R-CNN and Fast R-CNN techniques, which leveraged region proposals to generate predictions of the locations of objects in an image along with the classes corresponding to objects in the image. Furthermore, we learned about the bottleneck of the speed of inference, which happens because of having two different models – one for region proposal generation and another for object detection. In this chapter, we will learn about different modern techniques, such as Faster R-CNN, YOLO, and **Single-Shot Detector** (**SSD**), that overcome slow inference time by employing a single model to make predictions for both the class of object and the bounding box in a single shot. We will start by learning about anchor boxes and then proceed to learn about how each of the techniques works and how to implement them to detect objects in an image.

We will cover the following topics in this chapter:

- Components of modern object detection algorithms
- Training Faster R-CNN on a custom dataset
- Working details of YOLO
- Training YOLO on a custom dataset
- Working details of SSD
- Training SSD on a custom dataset

Components of modern object detection algorithms

The drawback of the R-CNN and Fast R-CNN techniques is that they have two disjointed networks – one to identify the regions that likely contain an object and the other to make corrections to the bounding box where an object is identified. Furthermore, both the models require as many forward propagations as there are region proposals. Modern object detection algorithms focus heavily on training a single neural network and have the capability to detect all objects in one forward pass. In the subsequent sections, we will learn about the various components of a typical modern object detection algorithm:

- Anchor boxes
- **Region proposal network** (**RPN**)
- Region of interest pooling

Anchor boxes

So far, we have had region proposals coming from the `selectivesearch` method. Anchor boxes come in as a handy replacement for selective search – we will learn how they replace `selectivesearch`-based region proposals in this section.

Typically, a majority of objects have a similar shape – for example, in a majority of cases, a bounding box corresponding to an image of a person will have a greater height than width, and a bounding box corresponding to the image of a truck will have a greater width than height. Thus, we will have a decent idea of the height and width of the objects present in an image even before training the model (by inspecting the ground truths of bounding boxes corresponding to objects of various classes).

Furthermore, in some images, the objects of interest might be scaled – resulting in a much smaller or much greater height and width than average – while still maintaining the aspect ratio (that is, $\frac{height}{width}$).

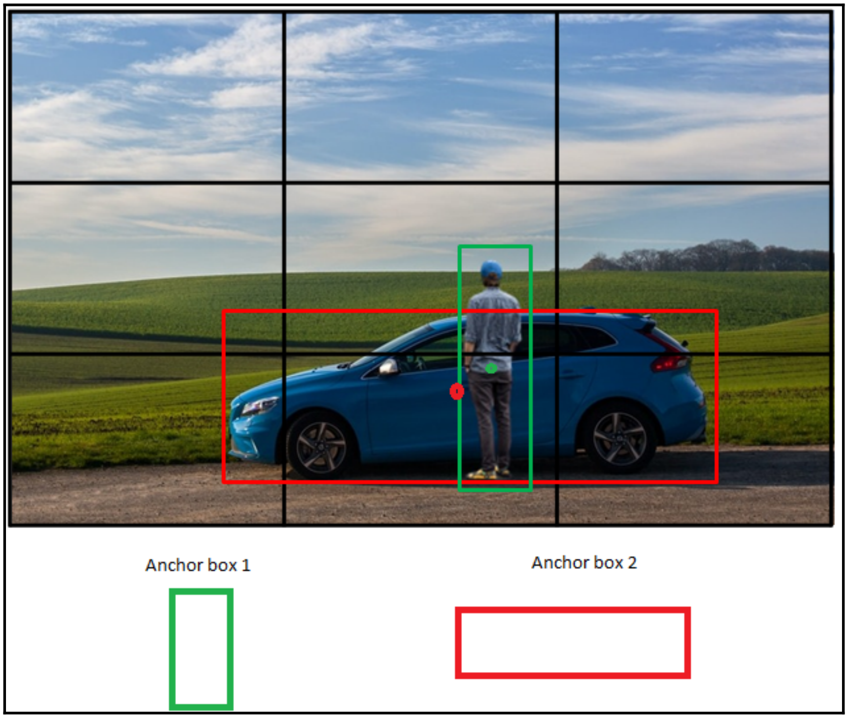
Once we have a decent idea of the aspect ratio and the height and width of objects (which can be obtained from ground truth values in the dataset) present in our images, we define the anchor boxes with heights and widths representing the majority of objects' bounding boxes within our dataset.

Typically, this is obtained by employing K-means clustering on top of the ground truth bounding boxes of objects present in images.

Now that we understand how anchor boxes' heights and widths are obtained, we will learn about how to leverage them in the process:

1. Slide each anchor box over an image from top left to bottom right.
2. The anchor box that has a high **intersection over union** (**IoU**) with the object will have a label that mentions that it contains an object, and the others will be labeled 0:
    - We can modify the threshold of the IoU by mentioning that if the IoU is greater than a certain threshold, the object class is 1; if it is less than another threshold, the object class is 0, and it is unknown otherwise.

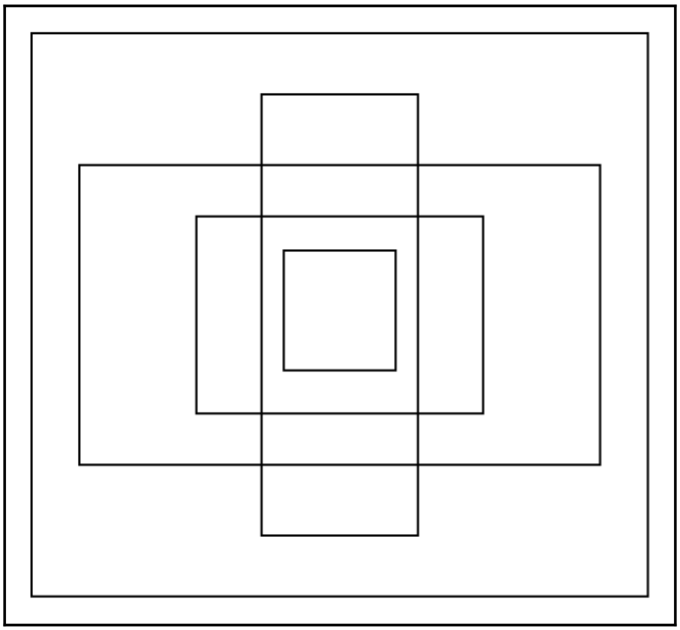
Once we obtain the ground truths as defined here, we can build a model that can predict the location of an object and also the offset corresponding to the anchor box to match it with ground truth. Let's now understand how anchor boxes are represented in the following image:



In the preceding image, we have two anchor boxes, one that has a greater height than width and the other with a greater width than height, to correspond to the objects (classes) in the image – a person and a car.

We slide the two anchor boxes over the image and note the locations where the IoU of the anchor box with the ground truth is the highest and denote that this particular location contains an object while the rest of the locations do not contain an object.

In addition to the preceding two anchor boxes, we would also create anchor boxes with varying scales so that we accommodate the differing scales at which an object can be presented within an image. An example of how the different scales of anchor boxes look follows:



Note that all the anchor boxes have the same center but different aspect ratios or scales.

Now that we understand anchor boxes, in the next section, we will learn about the RPN, which leverages anchor boxes to come up with predictions of regions that are likely to contain an object.

Region Proposal Network

Imagine a scenario where we have a 224 x 224 x 3 image. Furthermore, let's say that the anchor box is of shape 8 x 8 for this example. If we have a stride of 8 pixels, we are fetching 224/8 = 28 crops of a picture for every row – essentially 28*28 = 576 crops from a picture. We then take each of these crops and pass through a Region Proposal Network model (RPN) that indicates whether the crop contains an image. Essentially, an **RPN** suggests the likelihood of a crop containing an object.

Let's compare the output of `selectivesearch` and the output of an RPN.

`selectivesearch` gives a region candidates based on a set of computations on top of pixel values. However, an RPN generates region candidates based on the anchor boxes and the strides with which anchor boxes are slid over the image. Once we obtain the region candidates using either of these two methods, we identify the candidates that are most likely to contain an object.

While region proposal generation based on `selectivesearch` is done outside of the neural network, we can build an RPN that is a part of the object detection network. Using an RPN, we are now in a position where we don't have to perform unnecessary computations to calculate region proposals outside of the network. This way, we have a single model to identify regions, identify classes of objects in image, and identify their corresponding bounding box locations.

Next, we will learn how an RPN identifies whether a region candidate (a crop obtained after sliding an anchor box) contains an object or not. In our training data, we would have the ground truth correspond to objects. We now take each region candidate and compare with the ground truth bounding boxes of objects in an image to identify whether the IoU between a region candidate and a ground truth bounding box is greater than a certain threshold. If the IoU is greater than a certain threshold (say, 0.5), the region candidate contains an object, and if the IoU is less than a threshold (say 0.1), the region candidate does not contain an object and all the candidates that have an IoU between the two thresholds (0.1 - 0.5) are ignored while training.

Once we train a model to predict if the region candidate contains an object, we then perform non-max suppression, as multiple overlapping regions can contain an object.

In summary, an RPN trains a model to enable it to identify region proposals with a high likelihood of containing an object by performing the following steps:

1. Slide anchor boxes of different aspect ratios and sizes across the image to fetch crops of an image.
2. Calculate the IoU between the ground truth bounding boxes of objects in the image and the crops obtained in the previous step.
3. Prepare the training dataset in such a way that crops with an IoU greater than a threshold contain an object and crops with an IoU less than a threshold do not contain an object.
4. Train the model to identify regions that contain an object.
5. Perform non-max suppression to identify the region candidate that has the highest probability of containing an object and eliminate other region candidates that have a high overlap with it.

Classification and regression

So far, we have learned about the following steps in order to identify objects and perform offsets to bounding boxes:

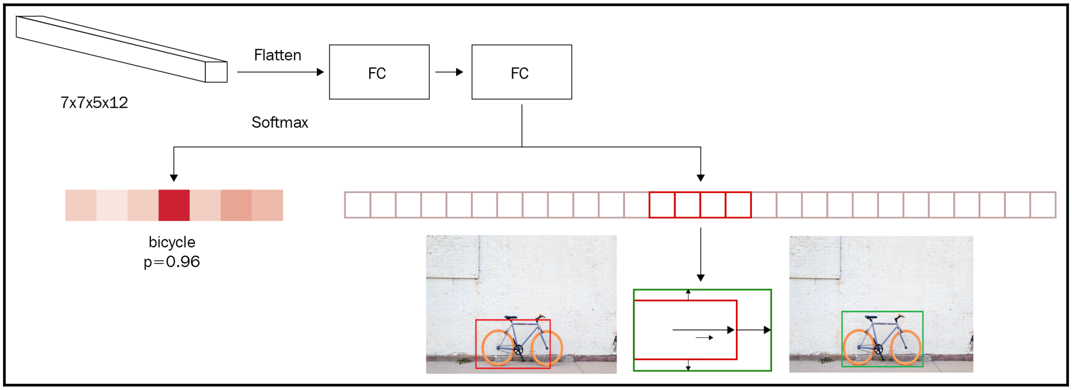
1. Identify the regions that contain objects.
2. Ensure that all the feature maps of regions, irrespective of the regions' shape, are exactly the same using **region of interest** (**RoI**) pooling (which we learned about in the previous chapter).

Two issues with these steps are as follows:

1. The region proposals do not correspond tightly over the object (IoU>0.5 is the threshold we had in the RPN).
2. We identified whether the region contains an object or not, but not the class of the object located in the region.

We address these two issues in this section, where we take the uniformly shaped feature map obtained previously and pass it through a network. We expect the network to predict the class of the object contained within the region and also the offsets corresponding to the region to ensure that the bounding box is as tight as possible around the object in the image.

Let's understand this through the following diagram:

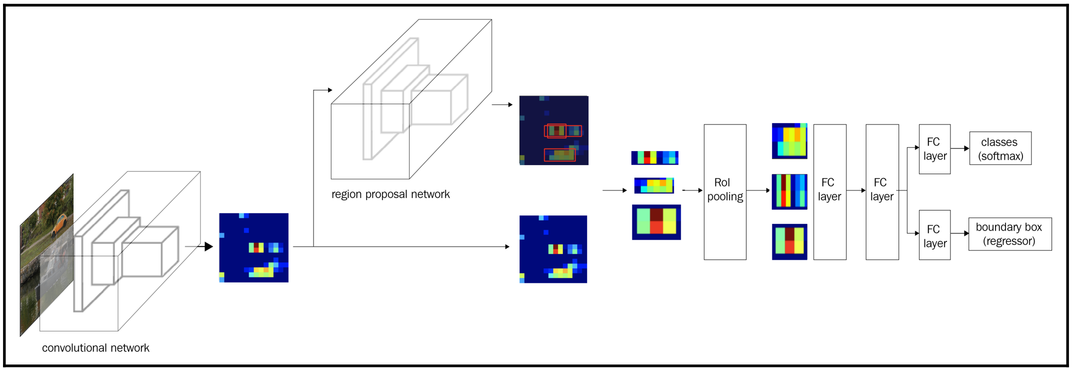


In the preceding diagram, we are taking the output of RoI pooling as input (the 7 x 7 x 512 shape), flattening it, and connecting to a dense layer before predicting two different aspects:

1. Class of object in the region
2. Amount of offset to be done on the predicted bounding boxes of the region to maximize the IoU with the ground truth

Hence, if there are 20 classes in the data, the output of the neural network contains a total of 25 outputs – 21 classes (including the background class) and the 4 offsets to be applied to the height, width, and two center coordinates of the bounding box.

Now that we have learned the different components of an object detection pipeline, let's summarize it with the following diagram:



With the working details of each of the components of Faster R-CNN in place, in the next section, we will code up object detection using the Faster R-CNN algorithm.

Training Faster R-CNN on a custom dataset

In the following code, we will train the Faster R-CNN algorithm to detect the bounding boxes around objects present in images. For this, we will work on the same truck versus bus detection exercise that we worked on in the previous chapter:

The following code is available as `Training_Faster_RCNN.ipynb` in the `Chapter08` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`.

1. Download the dataset:

```
import os
if not os.path.exists('images'):
    !pip install -qU torch_snippets
    from google.colab import files
    files.upload() # upload kaggle.json
    !mkdir -p ~/.kaggle
    !mv kaggle.json ~/.kaggle/
    !ls ~/.kaggle
    !chmod 600 /root/.kaggle/kaggle.json
    !kaggle datasets download \
        -d sixhky/open-images-bus-trucks/
    !unzip -qq open-images-bus-trucks.zip
    !rm open-images-bus-trucks.zip
```

2. Read the DataFrame containing metadata of information about images and their bounding box, and classes:

```
from torch_snippets import *
from PIL import Image
IMAGE_ROOT = 'images/images'
DF_RAW = df = pd.read_csv('df.csv')
```

3. Define the indices corresponding to labels and targets:

```
label2target = {l:t+1 for t,l in \
                enumerate(DF_RAW['LabelName'].unique())}
label2target['background'] = 0
```

```
target2label = {t:l for l,t in label2target.items()}
background_class = label2target['background']
num_classes = len(label2target)
```

4. Define the function to pre-process an image – `preprocess_image`:

```
def preprocess_image(img):
    img = torch.tensor(img).permute(2,0,1)
    return img.to(device).float()
```

5. Define the dataset class – `OpenDataset`:

- Define an `__init__` method that takes the folder containing images and the DataFrame containing the metadata of the images as inputs:

```
class OpenDataset(torch.utils.data.Dataset):
    w, h = 224, 224
    def __init__(self, df, image_dir=IMAGE_ROOT):
        self.image_dir = image_dir
        self.files = glob.glob(self.image_dir+'/*')
        self.df = df
        self.image_infos = df.ImageID.unique()
```

- Define the `__getitem__` method, where we return the pre-processed image and the target values:

```
    def __getitem__(self, ix):
        # load images and masks
        image_id = self.image_infos[ix]
        img_path = find(image_id, self.files)
        img = Image.open(img_path).convert("RGB")
        img = np.array(img.resize((self.w, self.h), \
                              resample=Image.BILINEAR))/255.
        data = df[df['ImageID'] == image_id]
        labels = data['LabelName'].values.tolist()
        data = data[['XMin','YMin','XMax','YMax']].values
        # Convert to absolute coordinates
        data[:,[0,2]] *= self.w
        data[:,[1,3]] *= self.h
        boxes = data.astype(np.uint32).tolist()
        # torch FRCNN expects ground truths as
        # a dictionary of tensors
        target = {}
        target["boxes"] = torch.Tensor(boxes).float()
        target["labels"] = torch.Tensor([label2target[i] \
                                for i in labels]).long()
        img = preprocess_image(img)
        return img, target
```

Note that for the first time, we are returning the output as a dictionary of tensors and not as a list of tensors. This is because the official PyTorch implementation of the `FRCNN` class expects the target to contain the absolute coordinates of bounding boxes and the label information.

- Define the `collate_fn` method (by default, `collate_fn` works only with tensors as inputs, but here, we are dealing with a list of dictionaries) and the `__len__` method:

```
    def collate_fn(self, batch):
        return tuple(zip(*batch))

    def __len__(self):
        return len(self.image_infos)
```

6. Create the training and validation dataloaders and datasets:

```
from sklearn.model_selection import train_test_split
trn_ids, val_ids = train_test_split(df.ImageID.unique(), \
                    test_size=0.1, random_state=99)
trn_df, val_df = df[df['ImageID'].isin(trn_ids)], \
                    df[df['ImageID'].isin(val_ids)]

train_ds = OpenDataset(trn_df)
test_ds = OpenDataset(val_df)

train_loader = DataLoader(train_ds, batch_size=4, \
            collate_fn=train_ds.collate_fn, drop_last=True)
test_loader = DataLoader(test_ds, batch_size=4, \
            collate_fn=test_ds.collate_fn, drop_last=True)
```

7. Define the model:

```
import torchvision
from torchvision.models.detection.faster_rcnn import
FastRCNNPredictor

device = 'cuda' if torch.cuda.is_available() else 'cpu'

def get_model():
    model = torchvision.models.detection\
                .fasterrcnn_resnet50_fpn(pretrained=True)
    in_features = model.roi_heads.box_predictor\
                        .cls_score.in_features
    model.roi_heads.box_predictor = FastRCNNPredictor(\
```

```
                                    in_features, num_classes)
        return model
```

The model contains the following key submodules:

```
=================================================================
Layer (type:depth-idx)                   Param #
=================================================================
├─GeneralizedRCNNTransform: 1-1          --
├─BackboneWithFPN: 1-2                   (26,799,296)
├─RegionProposalNetwork: 1-3             593,935
├─RoIHeads: 1-4                          13,905,930
=================================================================
Total params: 41,299,161
Trainable params: 14,499,865
Non-trainable params: 26,799,296
=================================================================
```

We notice the following:

- `GeneralizedRCNNTransform` is a simple resize followed by a normalize transformation:

```
(transform): GeneralizedRCNNTransform(
    Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    Resize(min_size=(800,), max_size=1333, mode='bilinear')
)
```

- `BackboneWithFPN` is a neural network that transforms input into a feature map.
- `RegionProposalNetwork` generates the anchor boxes for the preceding feature map and predicts individual feature maps for classification and regression tasks:

```
(rpn): RegionProposalNetwork(
  (anchor_generator): AnchorGenerator()
  (head): RPNHead(
    (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (cls_logits): Conv2d(256, 3, kernel_size=(1, 1), stride=(1, 1))
    (bbox_pred): Conv2d(256, 12, kernel_size=(1, 1), stride=(1, 1))
  )
)
```

- `RoIHeads` takes the preceding maps, aligns them using RoI pooling, processes them, and returns classification probabilities for each proposal and the corresponding offsets:

```
(roi_heads): RoIHeads(
  (box_roi_pool): MultiScaleRoIAlign()
  (box_head): TwoMLPHead(
    (fc6): Linear(in_features=12544, out_features=1024, bias=True)
    (fc7): Linear(in_features=1024, out_features=1024, bias=True)
  )
  (box_predictor): FastRCNNPredictor(
    (cls_score): Linear(in_features=1024, out_features=2, bias=True)
    (bbox_pred): Linear(in_features=1024, out_features=8, bias=True)
  )
)
```

8. Define functions to train on batches of data and calculate loss values on the validation data:

```
# Defining training and validation functions
def train_batch(inputs, model, optimizer):
    model.train()
    input, targets = inputs
    input = list(image.to(device) for image in input)
    targets = [{k: v.to(device) for k, v \
                in t.items()} for t in targets]
    optimizer.zero_grad()
    losses = model(input, targets)
    loss = sum(loss for loss in losses.values())
    loss.backward()
    optimizer.step()
    return loss, losses

@torch.no_grad()
def validate_batch(inputs, model):
    model.train()
#to obtain losses, model needs to be in train mode only
#Note that here we arn't defining the model's forward method
#hence need to work per the way the model class is defined
    input, targets = inputs
    input = list(image.to(device) for image in input)
    targets = [{k: v.to(device) for k, v \
                in t.items()} for t in targets]

    optimizer.zero_grad()
    losses = model(input, targets)
    loss = sum(loss for loss in losses.values())
    return loss, losses
```

9. Train the model over increasing epochs:

    - Define the model:

```
model = get_model().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=0.005, \
                            momentum=0.9,weight_decay=0.0005)
n_epochs = 5
log = Report(n_epochs)
```

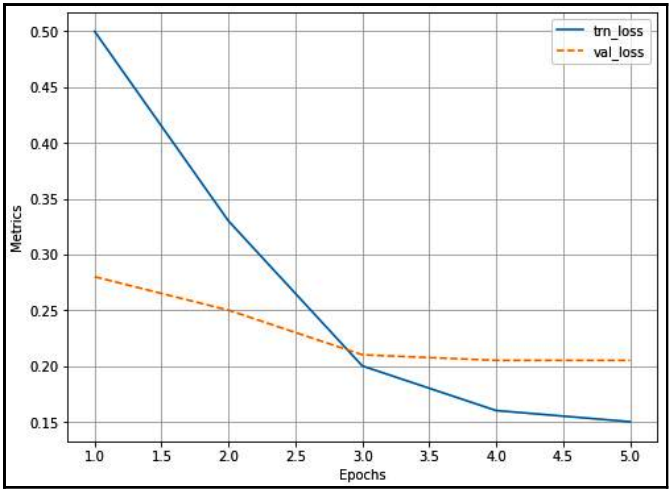
    - Train the model and calculate the loss values on the training and test datasets:

```
for epoch in range(n_epochs):
    _n = len(train_loader)
    for ix, inputs in enumerate(train_loader):
        loss, losses = train_batch(inputs, model, optimizer)
        loc_loss, regr_loss, loss_objectness, \
            loss_rpn_box_reg = \
                [losses[k] for k in ['loss_classifier', \
                'loss_box_reg', 'loss_objectness', \
                'loss_rpn_box_reg']]
        pos = (epoch + (ix+1)/_n)
        log.record(pos, trn_loss=loss.item(), \
                 trn_loc_loss=loc_loss.item(), \
                 trn_regr_loss=regr_loss.item(), \
                 trn_objectness_loss=loss_objectness.item(), \
               trn_rpn_box_reg_loss=loss_rpn_box_reg.item(), \
                 end='\r')

    _n = len(test_loader)
    for ix,inputs in enumerate(test_loader):
        loss, losses = validate_batch(inputs, model)
        loc_loss, regr_loss, loss_objectness, \
            loss_rpn_box_reg = \
                [losses[k] for k in ['loss_classifier', \
                'loss_box_reg', 'loss_objectness', \
                'loss_rpn_box_reg']]
        pos = (epoch + (ix+1)/_n)
        log.record(pos, val_loss=loss.item(), \
                 val_loc_loss=loc_loss.item(), \
                 val_regr_loss=regr_loss.item(), \
                val_objectness_loss=loss_objectness.item(), \
               val_rpn_box_reg_loss=loss_rpn_box_reg.item(), \
                 end='\r')
    if (epoch+1)%(n_epochs//5)==0: log.report_avgs(epoch+1)
```

10. Plot the variation of the various loss values over increasing epochs:

```
log.plot_epochs(['trn_loss','val_loss'])
```

This results in the following output:



11. Predict on a new image:

- The output of the trained model contains boxes, labels, and scores corresponding to classes. In the following code, we define a `decode_output` function that takes the model's output and provides the list of boxes, scores, and classes after non-max suppression:

```
from torchvision.ops import nms
def decode_output(output):
    'convert tensors to numpy arrays'
    bbs = \
    output['boxes'].cpu().detach().numpy().astype(np.uint16)
    labels = np.array([target2label[i] for i in \
                output['labels'].cpu().detach().numpy()])
    confs = output['scores'].cpu().detach().numpy()
    ixs = nms(torch.tensor(bbs.astype(np.float32)),
                            torch.tensor(confs), 0.05)
    bbs, confs, labels = [tensor[ixs] for tensor in [bbs, \
                                                    confs, labels]]

    if len(ixs) == 1:
        bbs,confs,labels = [np.array([tensor]) for tensor \
```

```
                                        in [bbs, confs, labels]]
    return bbs.tolist(), confs.tolist(), labels.tolist()
```

- Fetch the predictions of the boxes and classes on test images:

```
model.eval()
for ix, (images, targets) in enumerate(test_loader):
    if ix==3: break
    images = [im for im in images]
    outputs = model(images)
    for ix, output in enumerate(outputs):
        bbs, confs, labels = decode_output(output)
        info = [f'{l}@{c:.2f}' for l,c in zip(labels, confs)]
        show(images[ix].cpu().permute(1,2,0), bbs=bbs, \
                texts=labels, sz=5)
```

The preceding code provides the following output:

In this section, we have trained a Faster R-CNN model using the `fasterrcnn_resnet50_fpn` model class provided in the PyTorch `models` package. In the next section, we will learn about YOLO, a modern object detection algorithm that performs both object class detection and region correction in a single shot without the need to have a separate RPN.

Working details of YOLO

You Only Look Once (**YOLO**) and its variants are one of the prominent object detection algorithms. In this section, we will understand at a high level how YOLO works and the potential limitations of R-CNN-based object detection frameworks that YOLO overcomes.

First, let's learn about the possible limitations of R-CNN-based detection algorithms. In Faster R-CNN, we slide over the image using anchor boxes and identify the regions that are likely to contain an object, and then we make the bounding box corrections. However, in the fully connected layer, where only the detected region's RoI pooling output is passed as input, in the case of regions that do not fully encompass the object (where the object is beyond the boundaries of the bounding box of region proposal), the network has to guess the real boundaries of object, as it has not seen the full image (but has seen only the region proposal).

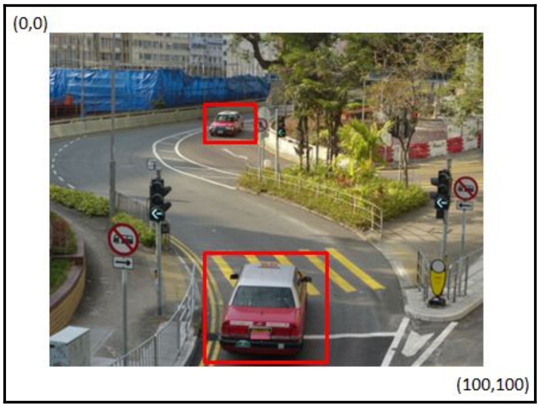
YOLO comes in handy in such scenarios, as it looks at the whole image while predicting the bounding box corresponding to an image.

Furthermore, Faster R-CNN is still slow, as we have two networks: the RPN and the final network that predicts classes and bounding boxes around objects.

Here, we will understand how YOLO overcomes the limitations of Faster R-CNN, both by looking at the whole image at once as well as by having a single network to make predictions. We will look at how data is prepared for YOLO through the following example:

1. Create a ground truth to train a model for a given image:

    - Let's consider an image with the given ground truth of bounding boxes in red:



- Divide the image into *N* x *N* grid cells – for now, let's say *N*=3:

- Identify those grid cells that contain the center of at least one ground truth bounding box. In our case, they are cells **b1** and **b3** of our 3 x 3 grid image.
- The cell(s) where the middle point of ground truth bounding box falls is/are responsible for predicting the bounding box of the object. Let's create the ground truth corresponding to each cell.
- The output ground truth corresponding to each cell is as follows:

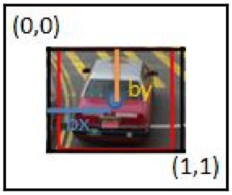
$$y= \begin{bmatrix} pc \\ bx \\ by \\ bw \\ bh \\ c1 \\ c2 \\ c3 \end{bmatrix}$$

Here, **pc** (the objectness score) is the probability of the cell containing an object.

Let's understand how to calculate **bx**, **by**, **bw**, and **bh**.

First, we consider the grid cell (let's consider the **b1** grid cell) as our universe, and normalize it to a scale between 0 and 1, as follows:

bx and **by** are the locations of the mid-point of the ground truth bounding box with respect to the image (of the grid cell), as defined previously. In our case, **bx** = 0.5, as the mid-point of the ground truth is at a distance of 0.5 units from the origin. Similarly, **by**= 0.5:



So far, we have calculated offsets from the grid cell center to the ground truth center corresponding to the object in the image. Now, let's understand how **bw** and **bh** are calculated.

bw is the ratio of the width of the bounding box with respect to the the width of the grid cell.

bh is the ratio of the height of the bounding box with respect to the height of the grid cell.

Next, we will predict the class corresponding to the grid cell. If we have three classes (c1 – truck, c2 – car, c3 – bus), we will predict the probability of the cell containing an object among any of the three classes. Note that we do not need a background class here, as **pc** corresponds to whether the grid cell contains an object.

Now that we understand how to represent the output layer of each cell, let's understand how we construct the output of our 3 x 3 grid cells.

- Let's consider the output of the grid cell **a3**:

$$y = \begin{bmatrix} 0 \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \end{bmatrix}$$

The output of cell **a3** is as shown in the preceding screenshot. As the grid cell does not contain an object, the first output (**pc** – objectness score) is 0 and the remaining values do not matter as the cell does not contain the center of any ground truth bounding boxes of an object.

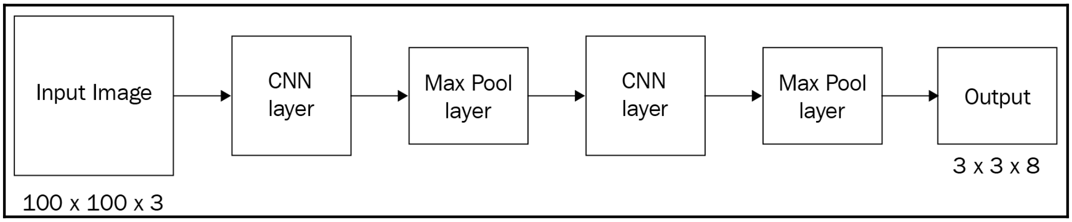
- Let's consider the output corresponding to grid cell **b1**:

$$y = \begin{bmatrix} 1 \\ 0.5 \\ 0.5 \\ 0.95 \\ 0.8 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

The preceding output is the way it is because the grid cell contains an object with the **bx**, **by**, **bw**, and **bh** values that were obtained in the same way as we went through earlier (in the bullet point before last), and finally the class being `car` resulting in c2 being 1 while c1 and c3 are 0.

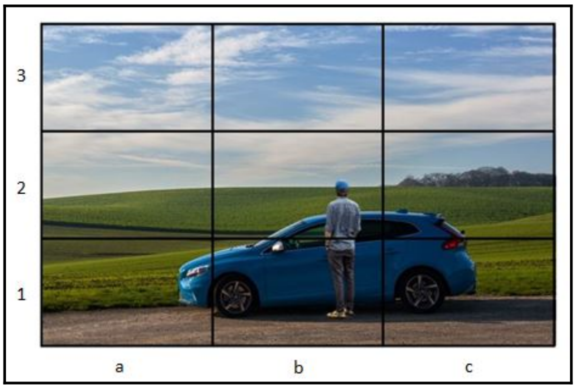
Note that for each cell, we are able to fetch 8 outputs. Hence, for the 3 x 3 grid of cells, we fetch 3 x 3 x 8 outputs.

2. Define a model where the input is an image and the output is 3 x 3 x 8 with the ground truth being as defined in the previous step:



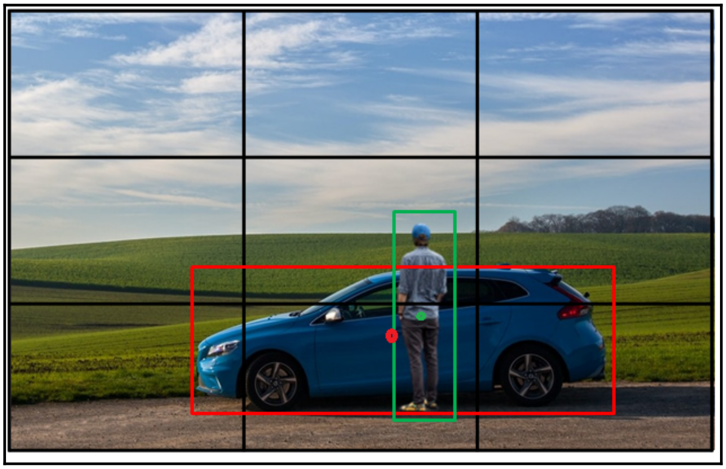
3. Define the ground truth by considering the anchor boxes.

   So far, we have been building for a scenario where the expectation is that there is only one object within a grid cell. However, in reality, there can be scenarios where there are multiple objects within the same grid cell. This would result in creating ground truths that are incorrect. Let's understand this phenomenon through the following example image:



   In the preceding example, the mid-point of the ground truth bounding boxes for both the car and the person fall in the same cell – cell **b1**.

   One way to avoid such a scenario is by having a grid that has more rows and columns – for example, a 19 x 19 grid. However, there can still be a scenario where an increase in the number of grid cells does not help. Anchor boxes come in handy in such a scenario. Let's say we have two anchor boxes – one that has a greater height than width (corresponding to the person) and another that has a greater width than height (corresponding to the car):



Typically, the anchor boxes would have the grid cell center as their centers. The output for each cell in a scenario where we have two anchor boxes is represented as a concatenation of the output expected of the two anchor boxes:

	pc
	bx
	by
	bh
	bw
	c1
	c2
	c3
y =	pc
	bx
	by
	bh
	bw
	c1
	c2
	c3

Here, **bx**, **by**, **bw**, and **bh** represent the offset from the anchor box (which is the universe in this scenario as seen in the image instead of the grid cell).

From the preceding screenshot, we see we have an output that is 3 x 3 x 16, as we have two anchors. The expected output is of the shape *N* x *N* x (`num_classes` + *1*) x (`num_anchor_boxes`), where *N* x *N* is the number of cells in the grid, `num_classes` is the number of classes in the dataset, and `num_anchor_boxes` is the number of anchor boxes.

4. Now we define the loss function to train the model.

When calculating the loss associated with the model, we need to ensure that we do not calculate the regression loss and classification loss when the objectness score is less than a certain threshold (this corresponds to the cells that do not contain an object).

Next, if the cell contains an object, we need to ensure that the classification across different classes is as accurate as possible.

Finally, if the cell contains an object, the bounding box offsets should be as close to expected as possible. However, since the offsets of width and height can be much higher when compared to the offset of the center (as offsets of the center range between 0 and 1, while the offsets of width and height need not), we give a lower weightage to offsets of width and height by fetching a square root value.

Calculate the loss of localization and classification as follows:

$$L_{loc} = \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 + (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2]$$

$$L_{cls} = \sum_{i=0}^{S^2} \sum_{j=0}^{B} (1_{ij}^{obj} + \lambda_{noobj}(1 - 1_{ij}^{obj}))(C_{ij} - \hat{C}_{ij})^2 + \sum_{i=0}^{S^2} \sum_{c \epsilon C} 1_{i}^{obj} (p_i(c) - \hat{p}_i(c))^2$$

$$L = L_{loc} + L_{cls}$$

Here, we observe the following:

- λ_{coord} is the weightage associated with regression loss.
- 1_{ij}^{obj} represents whether the cell contains an object.
- $\hat{p}_i(c)$ corresponds to the predicted class probability, and C_{ij} represents the objectness score.

The overall loss is a sum of classification and regression loss values.

With this in place, we are now in a position to train a model to predict the bounding boxes around objects. However, for a stronger understanding of YOLO and the variants of it, we encourage you to go through the original papers. Now that we understand how YOLO predicts bounding boxes and class of objects in a single shot, we will code it up in the next section.

Training YOLO on a custom dataset

Building on top of others' work is very important to becoming a successful practitioner in deep learning. For this implementation, we will use the official YOLO-v4 implementation to identify the location of buses and trucks in images. We will clone the repository of the authors' own implementation of YOLO and customize it to our needs in the following code.

> The following code is available as `Training_YOLO.ipynb` in the `Chapter08` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`.

Installing Darknet

First, pull the `darknet` repository from GitHub and compile it in the environment. The model is written in a separate language called Darknet, which is different from PyTorch. We will do so using the following code:

1. Pull the Git repo:

```
!git clone https://github.com/AlexeyAB/darknet
%cd darknet
```

2. Reconfigure the `Makefile` file:

```
!sed -i 's/OPENCV=0/OPENCV=1/' Makefile
# In case you dont have a GPU, make sure to comment out the
# below 3 lines
!sed -i 's/GPU=0/GPU=1/' Makefile
!sed -i 's/CUDNN=0/CUDNN=1/' Makefile
!sed -i 's/CUDNN_HALF=0/CUDNN_HALF=1/' Makefile
```

`Makefile` is a configuration file needed for installing `darknet` in the environment (think of this process as similar to the selections you make when installing software on Windows). We are forcing `darknet` to be installed with the following flags: `OPENCV`, `GPU`, `CUDNN`, and `CUDNN_HALF`. These are all important optimizations to make the training faster. Furthermore, in the preceding code, there is a curious function called `sed`, which stands for **stream editor**. It is a powerful Linux command that can modify information in text files directly from Command Prompt. Specifically, here we are using its search-and-replace function to replace `OPENCV=0` with `OPENCV=1`, and so on. The syntax to understand here is `sed 's/<search-string>/<replace-with>/' path/to/text/file`.

3. Compile the `darknet` source code:

```
!make
```

4. Install the `torch_snippets` package:

```
!pip install -q torch_snippets
```

5. Download and extract the dataset, and remove the ZIP file to save space:

```
!wget --quiet \
https://www.dropbox.com/s/agmzwk95v96ihic/open-images-bus-truc
ks.tar.xz
!tar -xf open-images-bus-trucks.tar.xz
!rm open-images-bus-trucks.tar.xz
```

6. Fetch the pre-trained weights to make a sample prediction:

```
!wget --quiet\
https://github.com/AlexeyAB/darknet/releases/download/darknet_
yolo_v3_optimal/yolov4.weights
```

7. Test whether the installation is successful by running the following command:

```
!./darknet detector test cfg/coco.data cfg/yolov4.cfg\
yolov4.weights
 data/person.jpg
```

This would make a prediction on `data/person.jpg` using the network built from `cfg/yolov4.cfg` and pre-trained weights – `yolov4.weights`. Furthermore, it fetches the classes from `cfg/coco.data`, which is what the pre-trained weights were trained on.

The preceding code results in predictions on the sample image (`data/person.jpg`) as follows:

```
data/person.jpg: Predicted in 54.532000 milli-seconds.
dog: 99%
person: 100%
horse: 98%
```

Now that we have learned about installing `darknet`, in the next section, we will learn about creating ground truths for our custom dataset to leverage `darknet`.

Setting up the dataset format

YOLO uses a fixed format for training. Once we store the images and labels in the required format, we can train on the dataset with a single command. So, let's learn about the files and folder structure needed for YOLO to train.

There are three important steps:

1. Create a text file at `data/obj.names` containing the names of classes, one class per line, by running the following line (`%%writefile` is a magic command that creates a text file at `data/obj.names` with whatever content is present in the notebook cell):

```
%%writefile data/obj.names
bus
truck
```

2. Create a text file at `data/obj.data` describing the parameters in the dataset and the locations of text files containing train and test image paths and the location of the file containing object names and the folder where you want to save trained models:

```
%%writefile data/obj.data
classes = 2
train = data/train.txt
valid = data/val.txt
names = data/obj.names
backup = backup/
```

> The extensions for the preceding text files are not `.txt`.
> Yolo uses hardcoded names and folders to identify where data is.
> Also, the magic `%%writefile` Jupyter function creates a file with the content mentioned in a cell, as shown previously. Treat each `%%writefile ...` as a separate cell in Jupyter.

3. Move all images and ground truth text files to the `data/obj` folder. We will copy images from the `bus-trucks` dataset to this folder along with the labels:

```
!mkdir -p data/obj
!cp -r open-images-bus-trucks/images/* data/obj/
!cp -r open-images-bus-trucks/yolo_labels/all/\
{train,val}.txt data/
!cp -r open-images-bus-trucks/yolo_labels/all/\
labels/*.txt data/obj/
```

Note that all the training and validation images are in the same `data/obj` folder. We also move a bunch of text files to the same folder. Each file that contains the ground truth for an image shares the same name as the image. For example, the folder might contain `1001.jpg` and `1001.txt`, implying that the text file contains labels and bounding boxes for that image. If `data/train.txt` contains `1001.jpg` as one of its lines, then it is a training image. If it's present in `val.txt`, then it is a validation image.

The text file itself should contain information like so: `cls, xc, yc, w, h,`, where `cls` is the class index of the object in the bounding box present at `(xc, yc)` which represents the centroid of the rectangle of width `w` and height `h`. Each of `xc`, `yc`, `w`, and `h` is a fraction of the image width and height. Store each object on a separate line.

For example, if an image of width 800 and height 600 contains one truck and one bus at centers (500,300) and (100,400) respectively and has widths and heights respectively of (200,100) and (300,50), then the text file would look as follows:

```
1 0.62 0.50 0.25 0.12
0 0.12 0.67 0.38 0.08
```

Now that we have created the data, let's configure the network architecture in the next section.

Configuring the architecture

YOLO comes with a long list of architectures. Some are large and some are small, to train on large or small datasets. Configurations can have different backbones. There are pre-trained configurations for standard datasets. Each configuration is a `.cfg` file present in the `cfgs` folder of the same GitHub repo that we cloned. Each of them contains the architecture of the network as a text file (as opposed to how we were building it with the `nn.Module` class) along with a few hyperparameters, such as batch size and learning rate. We will take the smallest available architecture and configure it for our dataset:

```
# create a copy of existing configuration and modify it in place
!cp cfg/yolov4-tiny-custom.cfg cfg/\
yolov4-tiny-bus-trucks.cfg
# max_batches to 4000 (since the dataset is small enough)
!sed -i 's/max_batches = 500200/max_batches=4000/' \
cfg/yolov4-tiny-bus-trucks.cfg
# number of sub-batches per batch
!sed -i 's/subdivisions=1/subdivisions=16/' \
cfg/yolov4-tiny-bus-trucks.cfg
# number of batches after which learning rate is decayed
!sed -i 's/steps=400000,450000/steps=3200,3600/' \
cfg/yolov4-tiny-bus-trucks.cfg
# number of classes is 2 as opposed to 80
# (which is the number of COCO classes)
!sed -i 's/classes=80/classes=2/g' \
cfg/yolov4-tiny-bus-trucks.cfg
# in the classification and regression heads,
# change number of output convolution filters
# from 255 -> 21 and 57 -> 33, since we have fewer classes
# we don't need as many filters
!sed -i 's/filters=255/filters=21/g' \
cfg/yolov4-tiny-bus-trucks.cfg
!sed -i 's/filters=57/filters=33/g' \
cfg/yolov4-tiny-bus-trucks.cfg
```

This way, we have repurposed `yolov4-tiny` to be trainable on our dataset. The only remaining step is to load the pre-trained weights and train the model, which we will do in the next section.

Training and testing the model

We will get the weights from the following GitHub location and store them in `build/darknet/x64`:

```
!wget --quiet \
https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v4_
pre/yolov4-tiny.conv.29
!cp yolov4-tiny.conv.29 build/darknet/x64/
```

Finally, we will train the model using the following line:

```
!./darknet detector train data/obj.data \
cfg/yolov4-tiny-bus-trucks.cfg yolov4-tiny.conv.29 \
-dont_show -mapLastAt
```

The `-dont_show` flag skips showing intermediate prediction images and `-mapLastAt` will periodically print the mean average precision on the validation data. The whole of the training might take 1 or 2 hours. The weights are periodically stored in a backup folder and can be used after training for predictions such as the following code, which makes predictions on a new image:

```
!pip install torch_snippets
from torch_snippets import Glob, stem, show, read
# upload your own images to a folder
image_paths = Glob('images-of-trucks-and-busses')
for f in image_paths:
    !./darknet detector test \
    data/obj.data cfg/yolov4-tiny-bus-trucks.cfg\
    backup/yolov4-tiny-bus-trucks_4000.weights {f}
    !mv predictions.jpg {stem(f)}_pred.jpg
for i in Glob('*_pred.jpg'):
    show(read(i, 1), sz=20)
```

The preceding code results in this:

Now that we have learned about leveraging YOLO to perform object detection on our custom dataset, in the next section, we will learn about leveraging SSD to perform object detection.

Working details of SSD

So far, we have seen a scenario where we made predictions after gradually convolving and pooling the output from the previous layer. However, we know that different layers have different receptive fields to the original image. For example, the initial layers have a smaller receptive field when compared to the final layers, which have a larger receptive field. Here, we will learn how SSD leverages this phenomenon to come up with a prediction of bounding boxes for images.

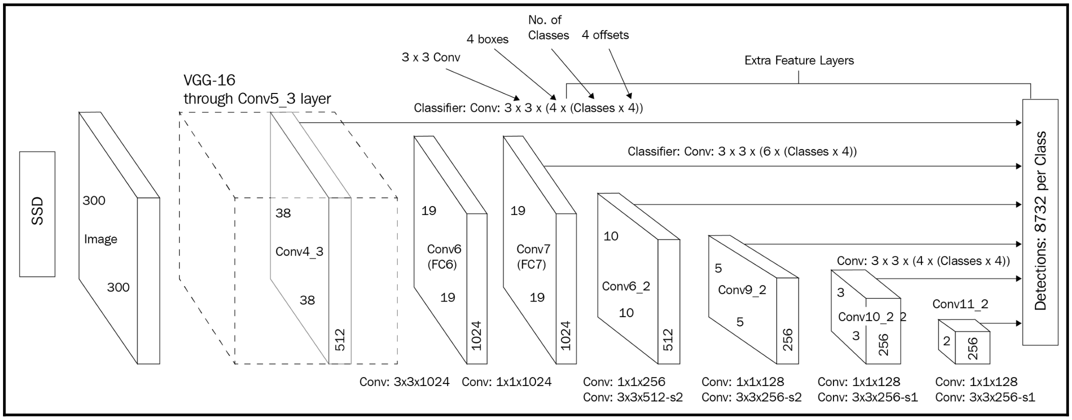
The workings behind how SSD helps overcome the issue of detecting objects with different scales is as follows:

- We leverage the pre-trained VGG network and extend it with a few additional layers until we obtain a 1 x 1 block.
- Instead of leveraging only the final layer for bounding box and class predictions, we will leverage all of the last few layers to make class and bounding box predictions.
- In place of anchor boxes, we will come up with default boxes that have a specific set of scale and aspect ratios.
- Each of the default boxes should predict the object and bounding box offset just like how anchor boxes are expected to predict classes and offsets in YOLO.

Now that we understand the main ways in which SSD differs from YOLO (which is that default boxes in SSD replace anchor boxes in YOLO and multiple layers are connected to the final layer in SSD, instead of gradual convolution pooling in YOLO), let's learn about the following:

- The network architecture of SSD
- How to leverage different layers for bounding box and class predictions
- How to assign scale and aspect ratios for default boxes in different layers

The network architecture of SSD is as follows:



As you can see in the preceding diagram, we are taking an image of size 300 x 300 x 3 and passing it through a pre-trained VGG-16 network to obtain the `conv5_3` layer's output. Furthermore, we are extending the network by adding a few more convolutions to the `conv5_3` output.

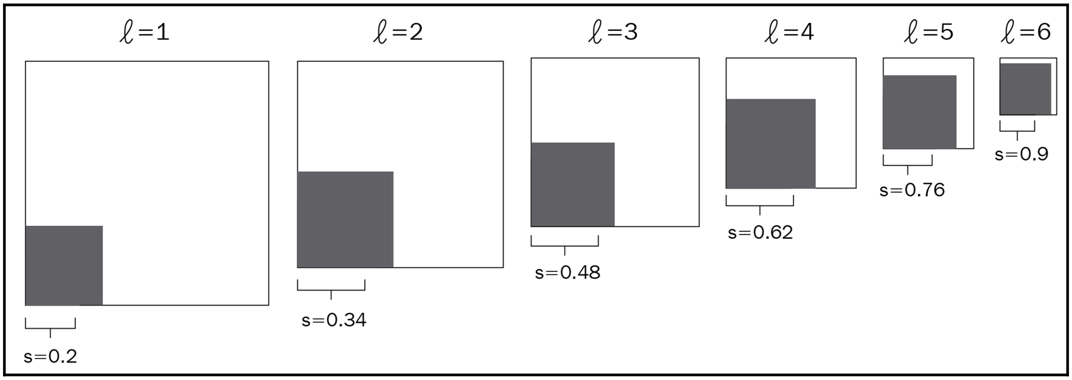
Next, we obtain a bounding box offset and class prediction for each cell and each default box (more on default boxes in the next section; for now, let's imagine that this is similar to an anchor box). The total number of predictions coming from the `conv5_3` output is 38 x 38 x 4, where 38 x 38 is the output shape of the `conv5_3` layer and 4 is the number of default boxes operating on the `conv5_3` layer. Similarly, the total number of parameters across the network is as follows:

Layer	**Number of parameters**
`conv5_3`	38 X 38 X 4 = 5,776
`FC6`	19 X 19 X 6 = 2,166
`conv8_2`	10 X 10 X 6 = 600
`conv9_2`	5 X 5 X 6 = 150
`conv10_2`	3 X 3 X 4 = 36
`conv11_2`	1 X 1 X 4 = 4
Total parameters	**8732**

Note that certain layers have a larger number of boxes (6 and not 4) when compared to other layers in the architecture described in the original paper.

Now, let's learn about the different scales and aspect ratios of default boxes. We will start with scales and then proceed to aspect ratios.

Let's imagine a scenario where the minimum scale of an object is 20% of the height and 20% of the width of an image, and the maximum scale of the object is 90% of the height and 90% of the width. In such a scenario, we gradually increase scale across layers (as we proceed toward later layers, the image size shrinks considerably), as follows:



The formula that enables the gradual scaling of the image is as follows:

$$level\ index : l = 1, \ldots, L$$
$$scale\ of\ boxes : s_l = s_{min} + \frac{s_{max} - s_{min}}{L-1}(l-1)$$

Now that we understand how to calculate scale across layers, we will now learn about coming up with boxes of different aspect ratios.

The possible aspect ratios are as follows:

$$aspect\ ratio :\ r \epsilon \{1, 2, 3, 1/2, 1/3\}$$

The center of the box for different layers are as follows:

$$center\ location : (x_l^i, y_l^i) = (\frac{i+0.5}{m}, \frac{j+0.5}{n})$$

Here *i* and *j* together represent a cell in layer *l*.

The width and height corresponding to different aspect ratios are calculated as follows:

$$width : w_l^r = s_l \sqrt{r}$$
$$height : h_l^r = s_l / \sqrt{r}$$

Note that we were considering four boxes in certain layers and six boxes in another layer. Now, if we want to have four boxes, we remove the {3,1/3} aspect ratios, else we consider all of the six possible boxes (five boxes with the same scale and one box with a different scale). So, let's learn how we obtain the sixth box:

$$additional\ scale : s_l^{'} = \sqrt{s_l s_{l+1}}\ when\ r = 1$$

Now that we have all the possible boxes, let's understand how we prepare the training dataset.

The default boxes that have an IoU greater than a threshold (say, 0.5) are considered positive matches, and the rest are negative matches.

In the output of SSD, we predict the probability of the box belonging to a class (where the 0^{th} class represents the background) and also the offset of the ground truth with respect to the default box.

Finally, we train the model by optimizing the following loss values:

- **Classification loss**: This is represented using the following equation:

$$L_{cls} = -\sum_{i\epsilon pos} 1_{ij}^{k} log(\hat{c}_i^k) - \sum_{i\epsilon neg} log(\hat{c}_i^0), where\ \hat{c}_i^k = softmax(c_i^k)$$

  In the preceding equation, `pos` represents the few default boxes that have a high overlap with the ground truth, while `neg` represents the misclassified boxes that were predicting a class but in fact did not contain an object. Finally, we ensure that the `pos`:`neg` ratio is at most 1:3, as if we do not perform this sampling, we would have a dominance of background class boxes.

- **Localization loss:** For localization, we consider the loss values only when the objectness score is greater than a certain threshold. The localization loss is calculated as follows:

$$L_{loc} = \sum_{i,j} \sum_{m \epsilon \{x,y,w,h\}} 1_{ij}^{match} L_1^{smooth}(d_m^i - t_m^j)^2$$

$$L_1^{smooth}(x) = \begin{cases} 0.5x^2 & if\ |x| < 1 \\ |x| - 0.5 & otherwise \end{cases}$$

$$t_x^j = (g_x^j - p_x^i)/p_w^i$$

$$t_y^j = (g_y^j - p_y^i)/p_h^i$$

$$t_w^j = log(g_w^j/p_w^i)$$

$$t_h^j = log(g_h^j/p_h^i)$$

Here *t* is the predicted offset and *d* is the actual offset.

Now that we understand how to train SSD, let's use it for our bus versus truck object detection exercise in the next section.

The core utility functions for this section are present in the GitHub repo: `https://github.com/sizhky/ssd-utils/`. Let's learn about them one by one before starting the training process.

Components in SSD code

There are three files in the GitHub repo. Let's dig into them a little and understand them before training. **Note that this section is not part of the training process, but is instead for understanding the imports used during training.**

We are importing the `SSD300` and `MultiBoxLoss` classes from the `model.py` file in the GitHub repository. Let's learn about both of them.

SSD300

When you look at the `SSD300` function definition, it is evident that the model comprises three sub-modules:

```
class SSD300(nn.Module):
    ...
    def __init__(self, n_classes, device):
```

```
...
self.base = VGGBase()
self.aux_convs = AuxiliaryConvolutions()
self.pred_convs = PredictionConvolutions(n_classes)
...
```

We send the input to `VGGBase` first, which returns two feature vectors of dimensions `(N, 512, 38, 38)` and `(N, 1024, 19, 19)`. The second output is going to be the input for `AuxiliaryConvolutions`, which returns more feature maps of dimensions `(N, 512, 10, 10)`, `(N, 256, 5, 5)`, `(N, 256, 3, 3)`, and `(N, 256, 1, 1)`. Finally, the first output from `VGGBase` and these four feature maps are sent to `PredictionConvolutions`, which returns 8,732 anchor boxes as we discussed previously.

The other key aspect of the `SSD300` class is the `create_prior_boxes` method. For every feature map, there are three items associated with it: the size of the grid, the scale to shrink the grid cell by (this is the base anchor box for this feature map), and the aspect ratios for all anchors in a cell. Using these three configurations, the code uses a triple `for` loop and creates a list of `(cx, cy, w, h)` for all 8,732 anchor boxes.

Finally, the `detect_objects` method takes tensors of classification and regression values (of the predicted anchor boxes) and converts them to actual bounding box coordinates.

MultiBoxLoss

As humans, we are only worried about a handful of bounding boxes. But for the way SSD works, we need to compare 8,732 bounding boxes from several feature maps and predict whether an anchor box contains valuable information or not. We assign this loss computation task to `MultiBoxLoss`.

The input for the forward method is the anchor box predictions from the model and the ground truth bounding boxes.

First, we convert the ground truth boxes into a list of 8,732 anchor boxes by comparing each anchor from the model with the bounding box. If the IoU is high enough, that particular anchor box will have non-zero regression coordinates and associates an object as the ground truth for classification. Naturally, most of the computed anchor boxes will have their associated class as `background` because their IoU with the actual bounding box will be tiny or, in quite a few cases, zero.

Once the ground truths are converted to these 8,732 anchor box regression and classification tensors, it is easy to compare them with the model's predictions since the shapes are now the same.

We perform `MSE-Loss` on the regression tensor and `CrossEntropy-Loss` on the localization tensor and add them up to be returned as the final loss.

Training SSD on a custom dataset

In the following code, we will train the SSD algorithm to detect the bounding boxes around objects present in images. We will use the truck versus bus object detection task we have been working on:

> The following code is available as `Training_SSD.ipynb` in the `Chapter08` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from and is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components from text.

1. Download the image dataset and clone the Git repository hosting the code for the model and the other utilities for processing the data:

```
import os
if not os.path.exists('open-images-bus-trucks'):
    !pip install -q torch_snippets
    !wget --quiet https://www.dropbox.com/s/agmzwk95v96ihic/\
    open-images-bus-trucks.tar.xz
    !tar -xf open-images-bus-trucks.tar.xz
    !rm open-images-bus-trucks.tar.xz
    !git clone https://github.com/sizhky/ssd-utils/
%cd ssd-utils
```

2. Pre-process the data, just like we did in the *Training Faster R-CNN on a custom dataset* section:

```
from torch_snippets import *
DATA_ROOT = '../open-images-bus-trucks/'
IMAGE_ROOT = f'{DATA_ROOT}/images'
DF_RAW = pd.read_csv(f'{DATA_ROOT}/df.csv')
df = DF_RAW.copy()

df = df[df['ImageID'].isin(df['ImageID'].unique().tolist())]
```

```
label2target = {l:t+1 for t,l in
enumerate(DF_RAW['LabelName'].unique())}
label2target['background'] = 0
target2label = {t:l for l,t in label2target.items()}
background_class = label2target['background']
num_classes = len(label2target)

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

3. Prepare a dataset class, just like we did in the *Training Faster R-CNN on a custom dataset* section:

```
import collections, os, torch
from PIL import Image
from torchvision import transforms
normalize = transforms.Normalize(
                mean=[0.485, 0.456, 0.406],
                std=[0.229, 0.224, 0.225]
            )
denormalize = transforms.Normalize(
                mean=[-0.485/0.229,-0.456/0.224,-0.406/0.255],
                std=[1/0.229, 1/0.224, 1/0.255]
            )


def preprocess_image(img):
    img = torch.tensor(img).permute(2,0,1)
    img = normalize(img)
    return img.to(device).float()
class OpenDataset(torch.utils.data.Dataset):
    w, h = 300, 300
    def __init__(self, df, image_dir=IMAGE_ROOT):
        self.image_dir = image_dir
        self.files = glob.glob(self.image_dir+'/*')
        self.df = df
        self.image_infos = df.ImageID.unique()
        logger.info(f'{len(self)} items loaded')
    def __getitem__(self, ix):
        # load images and masks
        image_id = self.image_infos[ix]
        img_path = find(image_id, self.files)
        img = Image.open(img_path).convert("RGB")
        img = np.array(img.resize((self.w, self.h), \
                       resample=Image.BILINEAR))/255.
        data = df[df['ImageID'] == image_id]
        labels = data['LabelName'].values.tolist()
        data = data[['XMin','YMin','XMax','YMax']].values
        data[:,[0,2]] *= self.w
```

```
            data[:,[1,3]] *= self.h
            boxes = data.astype(np.uint32).tolist() # convert to
            # absolute coordinates
            return img, boxes, labels

        def collate_fn(self, batch):
            images, boxes, labels = [], [], []
            for item in batch:
                img, image_boxes, image_labels = item
                img = preprocess_image(img)[None]
                images.append(img)
                boxes.append(torch.tensor( \
                            image_boxes).float().to(device)/300.)
                labels.append(torch.tensor([label2target[c] \
                        for c in image_labels]).long().to(device))
            images = torch.cat(images).to(device)
            return images, boxes, labels
        def __len__(self):
            return len(self.image_infos)
```

4. Prepare the training and test datasets and the dataloaders:

```
from sklearn.model_selection import train_test_split
trn_ids, val_ids = train_test_split(df.ImageID.unique(), \
                             test_size=0.1, random_state=99)
trn_df, val_df = df[df['ImageID'].isin(trn_ids)], \
                df[df['ImageID'].isin(val_ids)]

train_ds = OpenDataset(trn_df)
test_ds = OpenDataset(val_df)

train_loader = DataLoader(train_ds, batch_size=4, \
                          collate_fn=train_ds.collate_fn, \
                          drop_last=True)
test_loader = DataLoader(test_ds, batch_size=4, \
                         collate_fn=test_ds.collate_fn, \
                         drop_last=True)
```

5. Define functions to train on a batch of data and calculate the accuracy and loss values on the validation data:

```
def train_batch(inputs, model, criterion, optimizer):
    model.train()
    N = len(train_loader)
    images, boxes, labels = inputs
    _regr, _clss = model(images)
    loss = criterion(_regr, _clss, boxes, labels)
    optimizer.zero_grad()
```

```
        loss.backward()
        optimizer.step()
        return loss
    @torch.no_grad()
    def validate_batch(inputs, model, criterion):
        model.eval()
        images, boxes, labels = inputs
        _regr, _clss = model(images)
        loss = criterion(_regr, _clss, boxes, labels)
        return loss
```

6. Import the model:

```
from model import SSD300, MultiBoxLoss
from detect import *
```

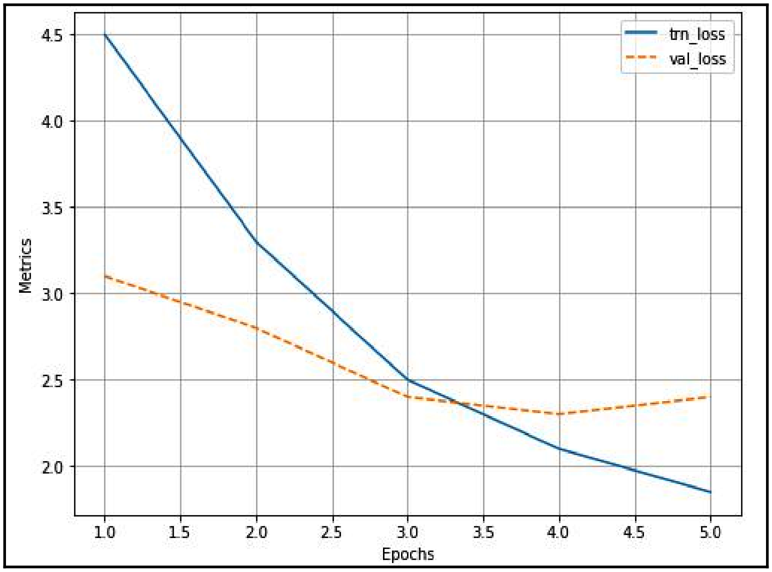
7. Initialize the model, optimizer, and loss function:

```
n_epochs = 5

model = SSD300(num_classes, device)
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-4, \
                              weight_decay=1e-5)
criterion = MultiBoxLoss(priors_cxcy=model.priors_cxcy, \
                         device=device)

log = Report(n_epochs=n_epochs)
logs_to_print = 5
```

8. Train the model over increasing epochs:

```
for epoch in range(n_epochs):
    _n = len(train_loader)
    for ix, inputs in enumerate(train_loader):
        loss = train_batch(inputs, model, criterion, \
                           optimizer)
        pos = (epoch + (ix+1)/_n)
        log.record(pos, trn_loss=loss.item(), end='\r')

    _n = len(test_loader)
    for ix,inputs in enumerate(test_loader):
        loss = validate_batch(inputs, model, criterion)
        pos = (epoch + (ix+1)/_n)
        log.record(pos, val_loss=loss.item(), end='\r')
```

The variation of training and test loss values over epochs is as follows:



9. Fetch a prediction on a new image:

- Fetch a random image:

```
image_paths = Glob(f'{DATA_ROOT}/images/*')
image_id = choose(test_ds.image_infos)
img_path = find(image_id, test_ds.files)
original_image = Image.open(img_path, mode='r')
original_image = original_image.convert('RGB')
```

- Fetch the bounding box, label, and score corresponding to the objects present in the image:

```
bbs, labels, scores = detect(original_image, model, \
                             min_score=0.9, max_overlap=0.5,\
                             top_k=200, device=device)
```

- Overlay the obtained output on the image:

```
labels = [target2label[c.item()] for c in labels]
label_with_conf = [f'{l} @ {s:.2f}' \
                   for l,s in zip(labels,scores)]
print(bbs, label_with_conf)
show(original_image, bbs=bbs, \
     texts=label_with_conf, text_sz=10)
```

The preceding code fetches a sample of outputs as follows (one image for each iteration of execution):

From this, we can see that we can detect objects in the image reasonably accurately.

Summary

In this chapter, we have learned about the working details of modern object detection algorithms: Faster R-CNN, YOLO, and SSD. We learned how they overcome the limitation of having two separate models – one for fetching region proposals and the other for fetching class and bounding box offsets on region proposals. Furthermore, we implemented Faster R-CNN using PyTorch, YOLO using `darknet`, and SSD from scratch.

In the next chapter, we will learn about image segmentation, which goes one step beyond object localization by identifying the pixels that correspond to an object.

Furthermore, in `Chapter 15`, *Combining Computer Vision and NLP Techniques*, we will learn about DETR, a transformer-based object detection algorithm, and in `Chapter 10`, *Applications of Object Detection, and Segmentation*, we will learn about the Detectron2 framework, which helps in not only detecting objects but also segmenting them in a single shot.

Test your understanding

1. Why is Faster R-CNN faster when compared to Fast R-CNN?
2. How are YOLO and SSD faster when compared to Faster R-CNN?
3. What makes YOLO and SSD single-shot algorithms?
4. What is the difference between the objectness score and the class score?
5. What is the difference between an anchor box and a default box?

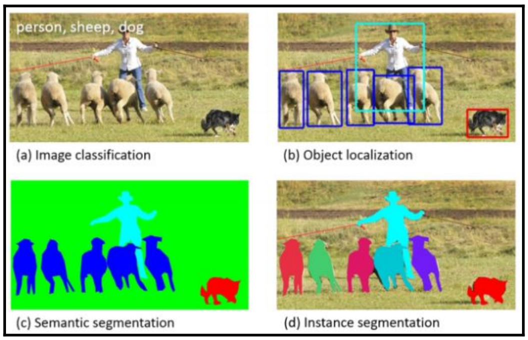
9
Image Segmentation

In the previous chapter, we learned about detecting objects present in images, along with the classes that correspond to the detected objects. In this chapter, we will go one step further by not only drawing a bounding box around the object but also by identifying the exact pixels that contain an object. In addition to that, by the end of this chapter, we will be able to single out instances/objects that belong to the same class.

In this chapter, we will learn about semantic segmentation and instance segmentation by taking a look at the U-Net and Mask R-CNN architectures. Specifically, we will cover the following topics:

- Exploring the U-Net architecture
- Implementing semantic segmentation using U-Net
- Exploring the Mask R-CNN architecture
- Implementing instance segmentation using Mask R-CNN

A succinct image of what we are trying to achieve through image segmentation (`https://arxiv.org/pdf/1405.0312.pdf`) is as follows:



Let's get started!

Exploring the U-Net architecture

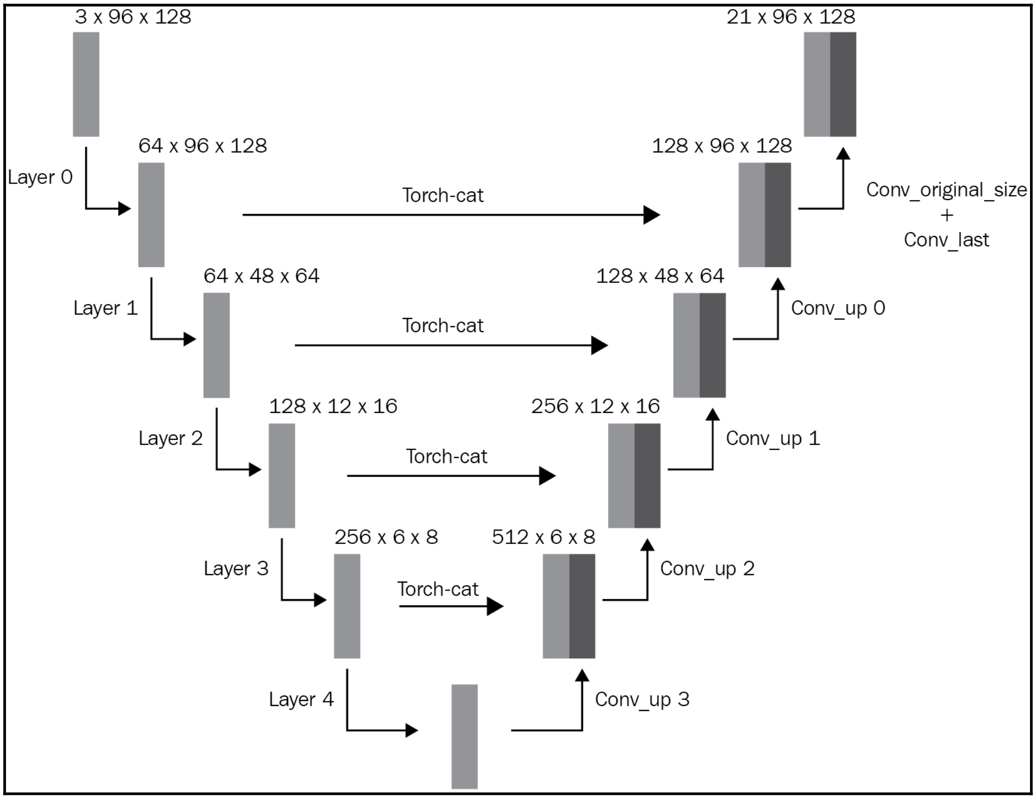
Imagine a scenario where you've been given an image and been asked to predict which pixel corresponds to what object. So far, when we have been predicting the class of an object and the bounding box corresponding to the object, we passed the image through a network, which then passes the image through a backbone architecture (such as VGG or ResNet), flattens the output at a certain layer, and connects additional dense layers before making predictions for the class and bounding box offsets. However, in the case of image segmentation, where the output shape is the same as that of the input image's shape, flattening the convolutions' outputs and then reconstructing the image might result in a loss of information. Furthermore, the contours and shapes present in the original image will not vary in the output image in the case of image segmentation, so the networks we have dealt with so far (which flatten the last layer and connect additional dense layers) are not optimal when we are performing segmentation.

In this section, we will learn about how to perform image segmentation.

The two aspects that we need to keep in mind while performing segmentation are as follows:

- The shape and structure of the objects in the original image remain the same in the segmented output.
- Leveraging a fully convolutional architecture (and not a structure where we flatten a certain layer) can help here since we are using one image as input and another as output.

The U-Net architecture helps us achieve this. A typical representation of U-Net is as follows (the input image is of the shape 3 x 96 x 128, while the number of classes present in the image is 21; this means that the output contains 21 channels):



The preceding architecture is called a **U-Net architecture** because of its "**U**"-like shape.

In the left half of the preceding diagram, we can see that the image passes through convolution layers, as we have seen in previous chapters, and that the image size keeps reducing while the number of channels keeps increasing. However, in the right half, we can see that we are upscaling the downscaled image, back to the original height and width but with as many channels as there are classes.

In addition, while upscaling, we are also leveraging information from the corresponding layers in the left half using **skip connections** so that we can preserve the structure/objects in the original image.

This way, the U-Net architecture learns to preserve the structure (and shapes of objects) of the original image while leveraging the convolution's features to predict the classes that correspond to each pixel.

In general, we have as many channels in the output as the number of classes we want to predict.

Performing upscaling

In the U-Net architecture, upscaling is performed using the `nn.ConvTranspose2d` method, which takes the number of input channels, the number of output channels, the kernel size, and stride as input parameters. An example calculation for `ConvTranspose2d` is as follows:

Input array

1	1	1
1	1	1
1	1	1

Input array adjusted for stride

1	0	1	0	1
0	0	0	0	0
1	0	1	0	1
0	0	0	0	0
1	0	1	0	1

Input array adjusted for stride and padding

0	0	0	0	0	0	0
0	1	0	1	0	1	0
0	0	0	0	0	0	0
0	1	0	1	0	1	0
0	0	0	0	0	0	0
0	1	0	1	0	1	0
0	0	0	0	0	0	0

Filter/Kernel

1	1
1	1

Output array

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

In the preceding example, we took an input array of shape 3 x 3 (**Input array**), applied a stride of 2 where we distributed the input values to accommodate the stride (**Input array adjusted for stride**), padded the array with zeros (**Input array adjusted for stride and padding**), and convolved the padded input with a filter (**Filter/Kernel**) to fetch the output array.

> By leveraging a combination of padding and stride, we have upscaled an input that is 3 x 3 in shape to an array of 6 x 6 in shape. While the preceding example is only for illustration purposes, the optimal filter values learn (because the filter weights and bias are optimized during the model training process) to reconstruct the original image as much as possible.

The hyperparameters in `nn.ConvTranspose2d` are as follows:

```
help(nn.ConvTranspose2d)

 |  Args:
 |      in_channels (int): Number of channels in the input image
 |      out_channels (int): Number of channels produced by the convolution
 |      kernel_size (int or tuple): Size of the convolving kernel
 |      stride (int or tuple, optional): Stride of the convolution. Default: 1
 |      padding (int or tuple, optional): ``dilation * (kernel_size - 1) - padding`` zero-padding
 |          will be added to both sides of each dimension in the input. Default: 0
 |      output_padding (int or tuple, optional): Additional size added to one side
 |          of each dimension in the output shape. Default: 0
 |      groups (int, optional): Number of blocked connections from input channels to output channels. Default: 1
 |      bias (bool, optional): If ``True``, adds a learnable bias to the output. Default: ``True``
 |      dilation (int or tuple, optional): Spacing between kernel elements. Default: 1
```

In order to understand how `nn.ConvTranspose2d` helps upscale an array, let's go through the following code:

1. Import the relevant packages:

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

2. Initialize a network, `m`, with the `nn.ConvTranspose2d` method:

```
m = nn.ConvTranspose2d(1, 1, kernel_size=(2,2),
                       stride=2, padding = 0)
```

In the preceding code, we are specifying that the input channel's value is `1`, the output channel's value is `1`, the size of the kernel is `(2,2)`, the stride is `2`, and that the padding is `0`.

Internally, padding is calculated as dilation * (kernel_size - 1) - padding.

Hence 1*(2-1)-0 = 1, where we add zero padding of 1 to both dimensions of the input array.

3. Initialize an input array and pass it through the model:

```
input = torch.ones(1, 1, 3, 3)
output = m(input)
output.shape
```

The preceding code results in a shape of `1x1x6x6`, as shown in the example image provided earlier.

Now that we understand how the U-Net architecture works and how `nn.ConvTranspose2d` helps upscale an image, let's implement it so that we can predict the different objects present in an image of a road scene.

Implementing semantic segmentation using U-Net

In this section, we'll leverage the U-Net architecture to predict the class that corresponds to all the pixels in the image. A sample of such an input-output combination is as follows:

Note that, in the preceding picture, the objects that belong to the same class (in the left image – the input image) have the same pixel value (in the right image – the output image), which is why we are **segmenting** the pixels that are **semantically** similar to each other. This is also known as semantic segmentation.

Now, let's learn how to code semantic segmentation:

> The following code is available as `Semantic_Segmentation_with_U_Net.ipynb` in the `Chapter09` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from and is moderately lengthy.

1. Let's begin by downloading the necessary datasets, installing the necessary packages, and then importing them. Once we've done that, we can define the device:

```
import os
if not os.path.exists('dataset1'):
    !wget -q \
     https://www.dropbox.com/s/0pigmmmynbf9xwq/dataset1.zip
    !unzip -q dataset1.zip
    !rm dataset1.zip
    !pip install -q torch_snippets pytorch_model_summary

from torch_snippets import *
from torchvision import transforms
from sklearn.model_selection import train_test_split
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

2. Define the function that will be used to transform images (`tfms`):

```
tfms = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406],
                                 [0.229, 0.224, 0.225])
        ])
```

3. Define the dataset class (`SegData`):

   - Specify the folder that contains images in the `__init__` method:

```
class SegData(Dataset):
    def __init__(self, split):
        self.items=stems(f'dataset1/images_prepped_{split}')
        self.split = split
```

   - Define the `__len__` method:

```
    def __len__(self):
        return len(self.items)
```

- Define the `__getitem__` method:

```
def __getitem__(self, ix):
        image = read(f'dataset1/images_prepped_{self.split}/\
{self.items[ix]}.png', 1)
        image = cv2.resize(image, (224,224))
        mask=read(f'dataset1/annotations_prepped_{self.split}\
/{self.items[ix]}.png')
        mask = cv2.resize(mask, (224,224))
        return image, mask
```

In the `__getitem__` method, we are resizing both the input (`image`) and output (`mask`) images so that they're the same shape. Note that the mask images contain integers that range between `[0,11]`. This indicates that there are 12 different classes.

- Define a function (`choose`) for selecting a random image index (mainly for debugging purposes):

```
def choose(self): return self[randint(len(self))]
```

- Define the `collate_fn` method for performing preprocessing on a batch of images:

```
def collate_fn(self, batch):
    ims, masks = list(zip(*batch))
    ims = torch.cat([tfms(im.copy()/255.)[None] \
                     for im in ims]).float().to(device)
    ce_masks = torch.cat([torch.Tensor(mask[None]) for \
                        mask in masks]).long().to(device)
    return ims, ce_masks
```

In the preceding code, we are preprocessing all the input images so that they have a channel (so that each image can be passed through a CNN later) once we've transformed the scaled images. Notice that `ce_masks` is a tensor of long integers, similar to the cross-entropy targets.

4. Define the training and validation datasets, as well as the dataloaders:

```
trn_ds = SegData('train')
val_ds = SegData('test')
trn_dl = DataLoader(trn_ds, batch_size=4, shuffle=True, \
                    collate_fn=trn_ds.collate_fn)
val_dl = DataLoader(val_ds, batch_size=1, shuffle=True, \
                    collate_fn=val_ds.collate_fn)
```

5. Define the neural network model:

   - Define the convolution block (`conv`):

```
def conv(in_channels, out_channels):
    return nn.Sequential(
        nn.Conv2d(in_channels,out_channels,kernel_size=3, \
                  stride=1, padding=1),
        nn.BatchNorm2d(out_channels),
        nn.ReLU(inplace=True)
    )
```

   In the preceding definition of `conv`, we are sequentially performing the `Conv2d` operation, the `BatchNorm2d` operation, and the `ReLU` operation.

   - Define the `up_conv` block:

```
def up_conv(in_channels, out_channels):
    return nn.Sequential(
        nn.ConvTranspose2d(in_channels, out_channels, \
                           kernel_size=2, stride=2),
        nn.ReLU(inplace=True)
    )
```

   `ConvTranspose2d` ensures that we upscale the images. This differs from the `Conv2d` operation, where we reduce the dimensions of the image. It takes an image that has `in_channels` number of channels as input channels and produces an image that has `out_channels` number of output channels.

   - Define the network class (`UNet`):

```
from torchvision.models import vgg16_bn
class UNet(nn.Module):
    def __init__(self, pretrained=True, out_channels=12):
        super().__init__()

        self.encoder = \
                vgg16_bn(pretrained=pretrained).features
        self.block1 = nn.Sequential(*self.encoder[:6])
        self.block2 = nn.Sequential(*self.encoder[6:13])
        self.block3 = nn.Sequential(*self.encoder[13:20])
        self.block4 = nn.Sequential(*self.encoder[20:27])
        self.block5 = nn.Sequential(*self.encoder[27:34])

        self.bottleneck = nn.Sequential(*self.encoder[34:])
```

```
        self.conv_bottleneck = conv(512, 1024)

        self.up_conv6 = up_conv(1024, 512)
        self.conv6 = conv(512 + 512, 512)
        self.up_conv7 = up_conv(512, 256)
        self.conv7 = conv(256 + 512, 256)
        self.up_conv8 = up_conv(256, 128)
        self.conv8 = conv(128 + 256, 128)
        self.up_conv9 = up_conv(128, 64)
        self.conv9 = conv(64 + 128, 64)
        self.up_conv10 = up_conv(64, 32)
        self.conv10 = conv(32 + 64, 32)
        self.conv11 = nn.Conv2d(32, out_channels, \
                                kernel_size=1)
```

In the preceding `__init__` method, we are defining all the layers that we would use in the `forward` method.

- Define the `forward` method:

```
    def forward(self, x):
        block1 = self.block1(x)
        block2 = self.block2(block1)
        block3 = self.block3(block2)
        block4 = self.block4(block3)
        block5 = self.block5(block4)

        bottleneck = self.bottleneck(block5)
        x = self.conv_bottleneck(bottleneck)

        x = self.up_conv6(x)
        x = torch.cat([x, block5], dim=1)
        x = self.conv6(x)

        x = self.up_conv7(x)
        x = torch.cat([x, block4], dim=1)
        x = self.conv7(x)

        x = self.up_conv8(x)
        x = torch.cat([x, block3], dim=1)
        x = self.conv8(x)

        x = self.up_conv9(x)
        x = torch.cat([x, block2], dim=1)
        x = self.conv9(x)

        x = self.up_conv10(x)
        x = torch.cat([x, block1], dim=1)
```

```
        x = self.conv10(x)

        x = self.conv11(x)

        return x
```

In the preceding code, we are making the U-style connection between the downscaling and upscaling convolution features by using `torch.cat` on the appropriate pairs of tensors.

- Define a function (`UNetLoss`) that will calculate our loss and accuracy values:

```
ce = nn.CrossEntropyLoss()
def UnetLoss(preds, targets):
    ce_loss = ce(preds, targets)
    acc = (torch.max(preds, 1)[1] == targets).float().mean()
    return ce_loss, acc
```

- Define a function that will train on batch (`train_batch`) and calculate metrics on the validation dataset (`validate_batch`):

```
def train_batch(model, data, optimizer, criterion):
    model.train()
    ims, ce_masks = data
    _masks = model(ims)
    optimizer.zero_grad()
    loss, acc = criterion(_masks, ce_masks)
    loss.backward()
    optimizer.step()
    return loss.item(), acc.item()

@torch.no_grad()
def validate_batch(model, data, criterion):
    model.eval()
    ims, masks = data
    _masks = model(ims)
    loss, acc = criterion(_masks, masks)
    return loss.item(), acc.item()
```

- Define the model, optimizer, loss function, and the number of epochs:

```
model = UNet().to(device)
criterion = UnetLoss
optimizer = optim.Adam(model.parameters(), lr=1e-3)
n_epochs = 20
```

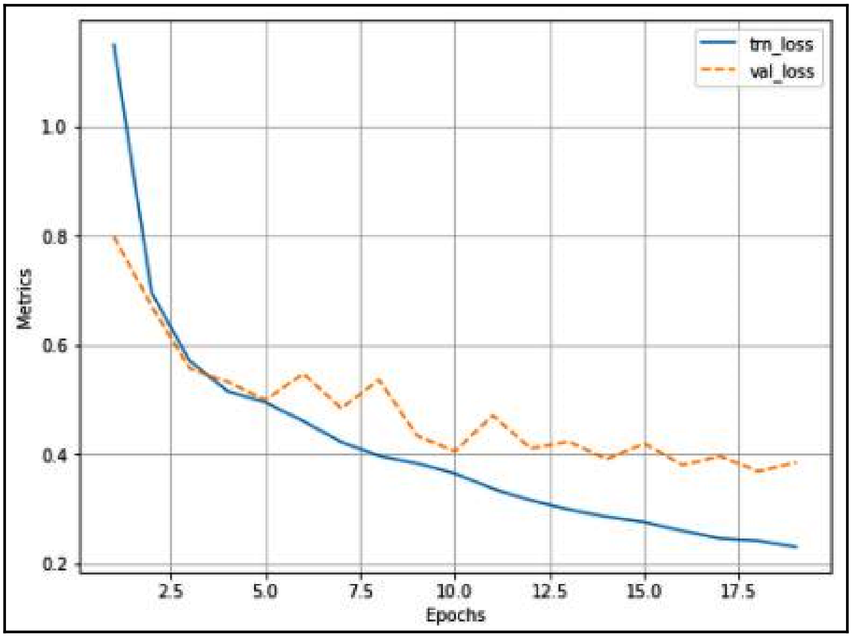
6. Train the model over increasing epochs:

```
log = Report(n_epochs)
for ex in range(n_epochs):
    N = len(trn_dl)
    for bx, data in enumerate(trn_dl):
        loss, acc = train_batch(model, data, optimizer, \
                                criterion)
        log.record(ex+(bx+1)/N,trn_loss=loss,trn_acc=acc, \
                                 end='\r')

    N = len(val_dl)
    for bx, data in enumerate(val_dl):
        loss, acc = validate_batch(model, data, criterion)
        log.record(ex+(bx+1)/N,val_loss=loss,val_acc=acc, \
                                 end='\r')
    log.report_avgs(ex+1)
```

7. Plot the training, validation loss, and accuracy values over increasing epochs:

```
log.plot_epochs(['trn_loss','val_loss'])
```

The preceding code generates the following output:



8. Calculate the predicted output on a new image:

   - Fetch model predictions on a new image:

```
im, mask = next(iter(val_dl))
_mask = model(im)
```

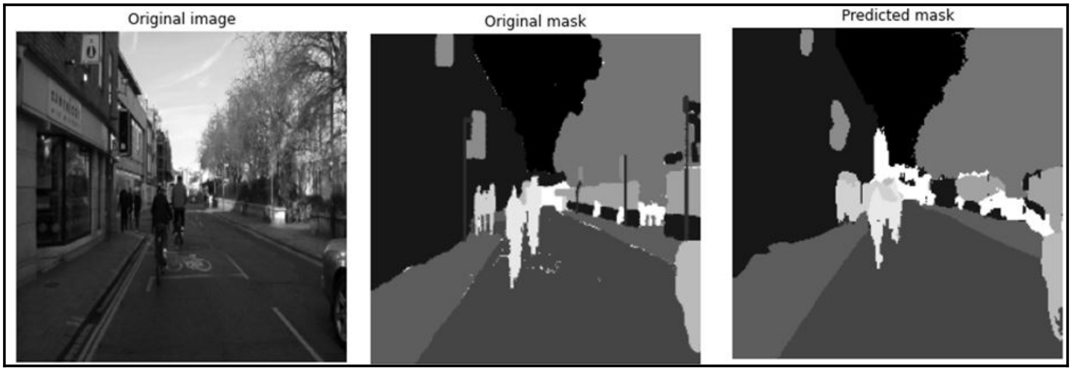
   - Fetch the channel that has the highest probability:

```
_, _mask = torch.max(_mask, dim=1)
```

   - Show the original and predicted images:

```
subplots([im[0].permute(1,2,0).detach().cpu()[:,:,0], \
          mask.permute(1,2,0).detach().cpu()[:,:,0], \
          _mask.permute(1,2,0).detach().cpu()[:,:,0]],nc=3, \
          titles=['Original image','Original mask', \
          'Predicted mask'])
```

   The preceding code generates the following output:



From the preceding picture, we can see that we can successfully generate a segmentation mask using the U-Net architecture. However, all instances of the same class will have the same predicted pixel value. What if we want to separate the instances of the `Person` class in the image? In the next section, we will learn about the Mask R-CNN architecture, which helps with generating instance-level masks so that we can differentiate between instances (even instances of the same class).

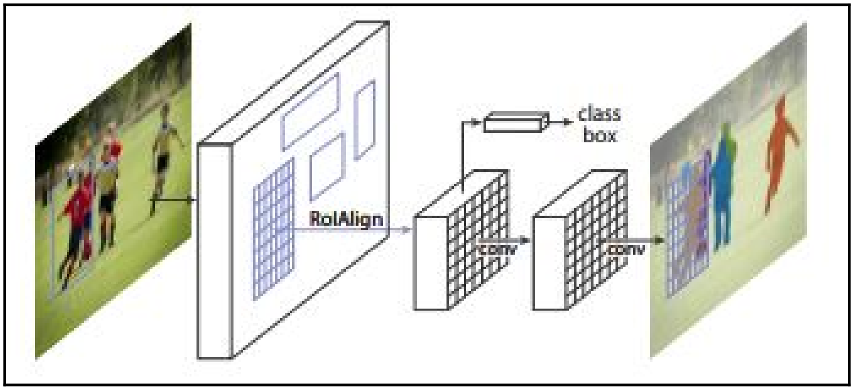
Exploring the Mask R-CNN architecture

The Mask R-CNN architecture helps in identifying/highlighting the instances of objects of a given class within an image. This comes in especially handy when there are multiple objects of the same type present within the image. Furthermore, the term **Mask** represents the segmentation that's done at the pixel level by Mask R-CNN.

The Mask R-CNN architecture is an extension of the Faster R-CNN network, which we learned about in the previous chapter. However, a few modifications have been made to the Mask R-CNN architecture, as follows:

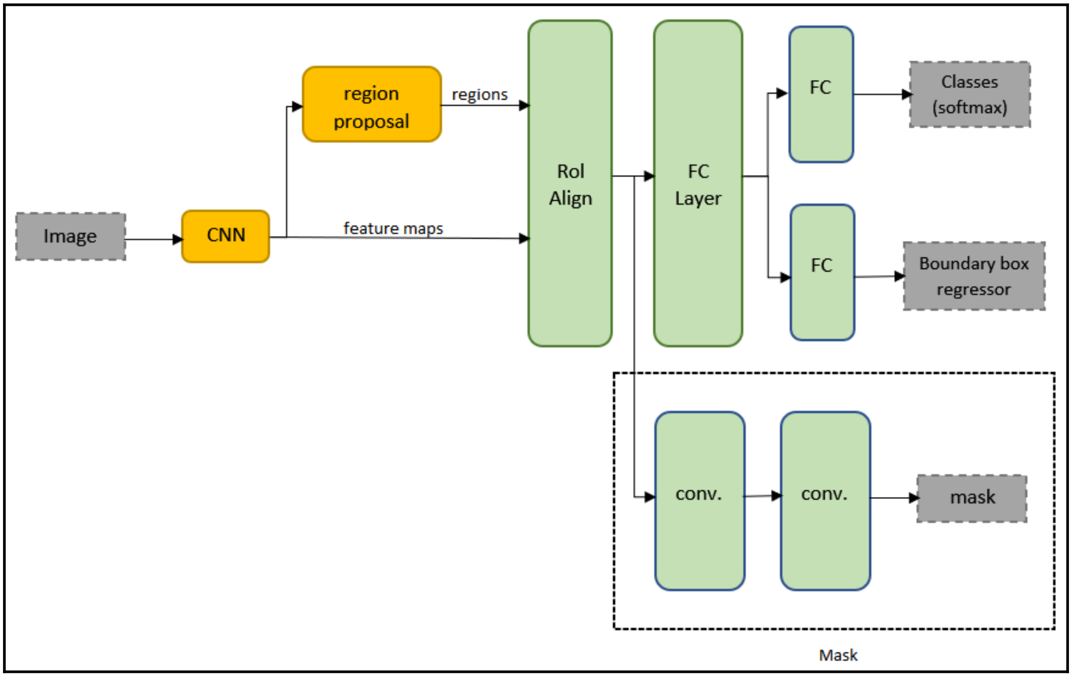
- The RoI Pooling layer has been replaced with the RoI Align layer.
- A mask head has been included to predict a mask of objects in addition to the head, which already predicts the classes of objects and bounding box correction in the final layer.
- A **fully convolutional network** (**FCN**) is leveraged for mask prediction.

Let's have a quick look at the events that occur within Mask R-CNN before we understand how each of the components works (image source: `https://arxiv.org/pdf/1703.06870.pdf`):



In the preceding diagram, note that we are fetching the class and bounding box information from one layer and the mask information from another layer.

The working details of the Mask R-CNN architecture are as follows:

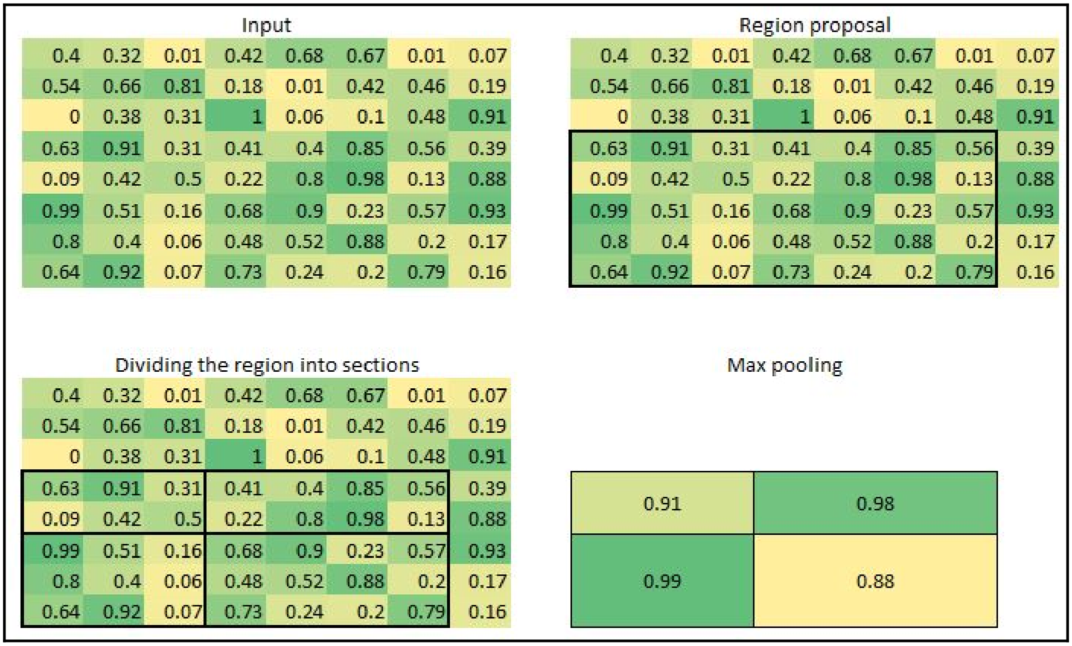


Before we implement the Mask R-CNN architecture, we need to understand its components. We'll start with RoI Align.

RoI Align

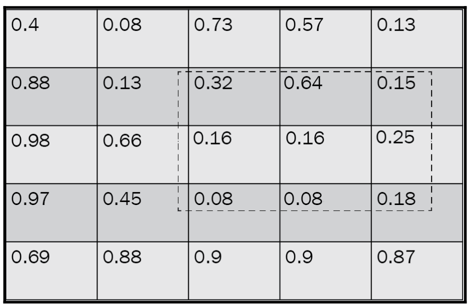
With Faster R-CNN, we learned about RoI Pooling. One of the drawbacks of RoI Pooling is that we are likely to lose certain information when we are performing the RoI pooling operation. This is because we are likely to have an even representation of content across all the areas of an image before pooling.

Let's go through the example we provided in the previous chapter:



In the preceding image, the region proposal is 5 x 7 in shape and we have to convert it into a 2 x 2 shape. While converting it into a 2 x 2 shape (a phenomenon called quantization), one part of the region has less representation compared to other parts of the region. This results in information loss since certain parts of the region have more weight than others. RoI Align comes to the rescue to address such a scenario.

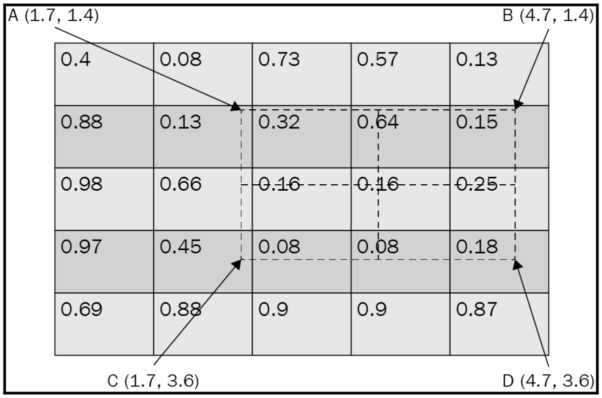
To understand how RoI Align works, let's go through a simple example. Here, we are trying to convert the following region (which is represented in dashed lines) into a 2 x 2 shape:



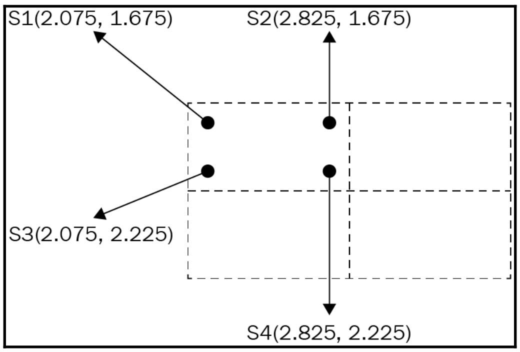
Note that the region (in dashed lines) is not equally spread across all the cells in the feature map.

We must perform the following steps to get a reasonable representation of the region in a 2 x 2 shape:

1. First, divide the region into an equal 2 x 2 shape:

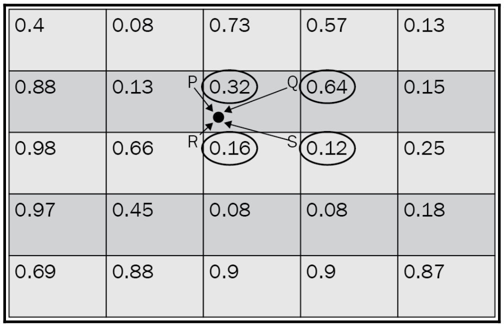


2. Define four points that are equally spaced within each of the 2 x 2 cells:

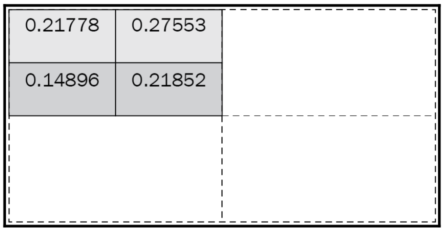


Note that, in the preceding diagram, the distance between two consecutive points is 0.75.

3. Calculate the weighted average value of each point based on its distance to the nearest known value:



4. Repeat the preceding interpolation step for all four points in a cell:



5. Perform average pooling across all four points within a cell:

0.2152	0.2335
0.3763	0.3562

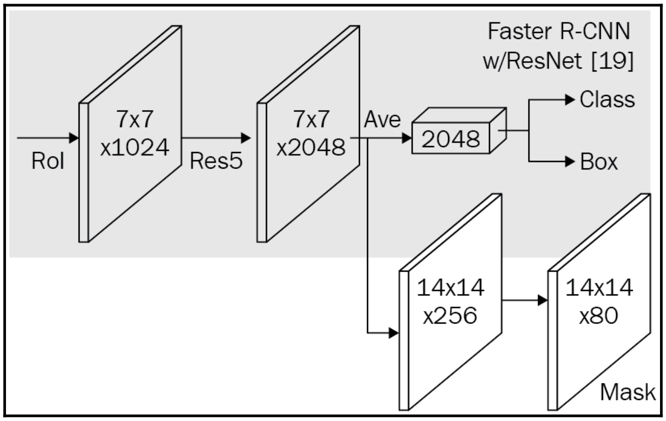
By implementing the preceding steps, we don't lose out on information when performing RoI Align; that is, when we place all the regions inside the same shape.

Mask head

Using RoI Align, we can get a more accurate representation of the region proposal that is obtained from the Region Proposal Network. Now, we want to obtain the segmentation (mask) output, given a standard shaped RoI Align output, for every region proposal.

Typically, in the case of object detection, we would pass the RoI Align through a flattened layer in order to predict the object's class and bounding box offset. However, in the case of image segmentation, we predict the pixels within a bounding box that contains the object. Hence, we now have a third output (apart from class and bounding box offset), which is the predicted mask within the region of interest.

Here, we are predicting the mask, which is an image overlay on top of the original image. Given that we are predicting an image, instead of flattening the RoI Align's output, we'll connect it to another convolution layer to get another image-like structure (width x height in dimension). Let's understand this phenomenon by taking a look at the following diagram:



In the preceding diagram, we have obtained an output of shape 7 x 7 x 2048 using the **feature pyramid network** (**FPN**), which now has 2 branches:

- The first branch returns the class of the object and the bounding box, post flattening the FPN output.
- The second branch performs convolution on top of the FPN's output to get a mask.

The ground truth corresponding to the 14 x 14 output is the resized image of the region proposals. The ground truth of the region proposal is of the shape 80 x 14 x 14 if there are 80 unique classes in the dataset. Each of the 80 x 14 x 14 pixels is a 1 or a 0, which indicates whether the pixel contains an object or not. Thus, we are performing binary cross-entropy loss minimization while predicting the class of a pixel.

Post model training, we are able to detect regions, get classes, get bounding box offsets, and get the mask corresponding to each region. When making an inference, we first detect the objects present in the image and make bounding box corrections. Then, we pass the offsetted region to the mask head to predict the mask that corresponds to different pixels in the region.

Now that we understand how the Mask R-CNN architecture works, let's code it up so that we can detect instances of people in an image.

Implementing instance segmentation using Mask R-CNN

To help us understand how to code Mask R-CNN for instance segmentation, we will leverage a dataset that masks people who are present within an image. The dataset we'll be using has been created from a subset of the ADE20K dataset, which is available at `https://groups.csail.mit.edu/vision/datasets/ADE20K/`. We will only use those images where we have masks for people.

The strategy that we'll adopt is as follows:

1. Fetch the dataset and then create datasets and dataloaders from it.
2. Create a ground truth in a format needed for PyTorch's official implementation of Mask R-CNN.
3. Download the pre-trained Faster R-CNN model and attach a Mask R-CNN head to it.
4. Train the model with a PyTorch code snippet that has been standardized for training Mask R-CNN.
5. Infer on an image by performing non-max suppression first and then identifying the bounding box and the mask corresponding to the people in the image.

Let's code up the preceding strategy:

The following code is available as `Instance_Segmentation.ipynb` in the `Chapter09` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from and is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components from text.

1. Import the relevant dataset and training utilities from GitHub:

```
!wget --quiet \
http://sceneparsing.csail.mit.edu/data/ChallengeData2017/image
s.tar
!wget --quiet \
http://sceneparsing.csail.mit.edu/data/ChallengeData2017/annot
ations_instance.tar
!tar -xf images.tar
!tar -xf annotations_instance.tar
!rm images.tar annotations_instance.tar
!pip install -qU torch_snippets
!wget --quiet \
https://raw.githubusercontent.com/pytorch/vision/master/refere
nces/detection/engine.py
!wget --quiet \
https://raw.githubusercontent.com/pytorch/vision/master/refere
nces/detection/utils.py
!wget --quiet \
https://raw.githubusercontent.com/pytorch/vision/master/refere
nces/detection/transforms.py
!wget --quiet \
https://raw.githubusercontent.com/pytorch/vision/master/refere
nces/detection/coco_eval.py
!wget --quiet \
https://raw.githubusercontent.com/pytorch/vision/master/refere
nces/detection/coco_utils.py
!pip install -q -U \
'git+https://github.com/cocodataset/cocoapi.git#subdirectory=P
ythonAPI'
```

2. Import all the necessary packages and define `device`:

```
from torch_snippets import *

import torchvision
from torchvision.models.detection.faster_rcnn import
```

```
FastRCNNPredictor
from torchvision.models.detection.mask_rcnn import
MaskRCNNPredictor

from engine import train_one_epoch, evaluate
import utils
import transforms as T
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

3. Fetch images that contain masks of people, as follows:

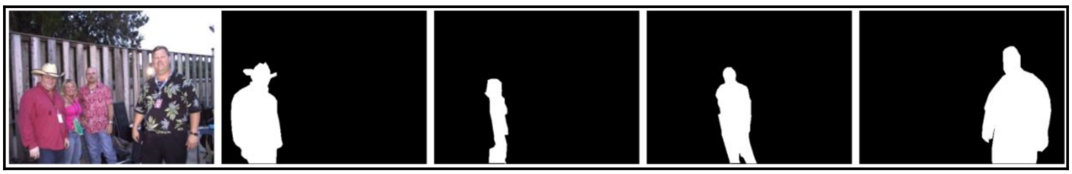
   - Loop through the `images` and `annotations_instance` folders to fetch filenames:

```
all_images = Glob('images/training')
all_annots = Glob('annotations_instance/training')
```

   - Inspect the original image and the representation of masks of instances of people:

```
f = 'ADE_train_00014301'

im = read(find(f, all_images), 1)
an = read(find(f, all_annots), 1).transpose(2,0,1)
r,g,b = an
nzs = np.nonzero(r==4) # 4 stands for person
instances = np.unique(g[nzs])
masks = np.zeros((len(instances), *r.shape))
for ix,_id in enumerate(instances):
    masks[ix] = g==_id

subplots([im, *masks], sz=20)
```

   The preceding code generates the following output:



   From the preceding image, we can see that a separate mask has been generated for each person. Here, there are four instances of the `Person` class.

> In this particular dataset, the ground truth instance annotations are provided in such a way that the Red channel in RGB corresponds to the class of object, while the Green channel corresponds to the instance number (in case there are multiple objects of the same class in the image – as in our example here). Furthermore, the `Person` class is encoded with a value of 4.

- Loop through the annotations and store the files that contain at least one person:

```
annots = []
for ann in Tqdm(all_annots):
    _ann = read(ann, 1).transpose(2,0,1)
    r,g,b = _ann
    if 4 not in np.unique(r): continue
    annots.append(ann)
```

- Split the files into training and validation files:

```
from sklearn.model_selection import train_test_split
_annots = stems(annots)
trn_items,val_items=train_test_split(_annots,random_state=2)
```

4. Define the transformation method:

```
def get_transform(train):
    transforms = []
    transforms.append(T.ToTensor())
    if train:
        transforms.append(T.RandomHorizontalFlip(0.5))
    return T.Compose(transforms)
```

5. Create the dataset class (`MasksDataset`), as follows:

- Define the `__init__` method, which takes the image names (`items`), transformation method (`transforms`), and the number of files to consider (`N`) as input:

```
class MasksDataset(Dataset):
    def __init__(self, items, transforms, N):
        self.items = items
        self.transforms = transforms
        self.N = N
```

- Define a method (`get_mask`) that will fetch a number of masks that's equivalent to the instances present in the image:

```
def get_mask(self, path):
    an = read(path, 1).transpose(2,0,1)
    r,g,b = an
    nzs = np.nonzero(r==4)
    instances = np.unique(g[nzs])
    masks = np.zeros((len(instances), *r.shape))
    for ix,_id in enumerate(instances):
        masks[ix] = g==_id
    return masks
```

- Fetch the image and the corresponding target values to be returned. Each person (instance) is treated as a different object class; that is, each instance is a different class. Note that, similar to training the Faster R-CNN model, the targets are returned as a dictionary of tensors. Let's define the `__getitem__` method:

```
def __getitem__(self, ix):
    _id = self.items[ix]
    img_path = f'images/training/{_id}.jpg'
    mask_path=f'annotations_instance/training/{_id}.png'
    masks = self.get_mask(mask_path)
    obj_ids = np.arange(1, len(masks)+1)
    img = Image.open(img_path).convert("RGB")
    num_objs = len(obj_ids)
```

- Apart from the masks themselves, Mask R-CNN also needs the bounding box information. However, this is easy to prepare, as shown in the following code:

```
    boxes = []
    for i in range(num_objs):
        obj_pixels = np.where(masks[i])
        xmin = np.min(obj_pixels[1])
        xmax = np.max(obj_pixels[1])
        ymin = np.min(obj_pixels[0])
        ymax = np.max(obj_pixels[0])
        if (((xmax-xmin)<=10) | (ymax-ymin)<=10):
            xmax = xmin+10
            ymax = ymin+10
        boxes.append([xmin, ymin, xmax, ymax])
```

In the preceding code, we are adjusting for scenarios where there are dubious ground truths (the height or width of the `Person` class is less than 10 pixels) by adding 10 pixels to the minimums of the x and y coordinates of the bounding box.

- Convert all the target values into tensor objects:

```
        boxes = torch.as_tensor(boxes, dtype=torch.float32)
        labels = torch.ones((num_objs,), dtype=torch.int64)
        masks = torch.as_tensor(masks, dtype=torch.uint8)
        area = (boxes[:, 3] - boxes[:, 1]) *\
                    (boxes[:, 2] - boxes[:, 0])
        iscrowd = torch.zeros((num_objs,), dtype=torch.int64)
        image_id = torch.tensor([ix])
```

- Store the target values in a dictionary:

```
        target = {}
        target["boxes"] = boxes
        target["labels"] = labels
        target["masks"] = masks
        target["image_id"] = image_id
        target["area"] = area
        target["iscrowd"] = iscrowd
```

- Specify the transformation method and return image; that is, `target`:

```
        if self.transforms is not None:
            img, target = self.transforms(img, target)
        return img, target
```

- Specify the `__len__` method:

```
    def __len__(self):
        return self.N
```

- Define the function that will choose a random image:

```
    def choose(self):
        return self[randint(len(self))]
```

- Inspect an input-output combination:

```
x = MasksDataset(trn_items, get_transform(train=True), N=100)
im,targ = x[0]
inspect(im,targ)
subplots([im, *targ['masks']], sz=10)
```

The following is some example output that the preceding code produces when it's run:

```
==============================================================
Tensor  Shape: torch.Size([3, 512, 683])      Min: 0.000     Max: 1.000     Mean: 0.555    dtype: torch.float32
==============================================================
Dict Of 6 items
==============================================================
        BOXES:
        Tensor  Shape: torch.Size([2, 4])      Min: 233.000   Max: 480.000   Mean: 362.750   dtype: torch.float32
==============================================================
==============================================================
        LABELS:
        Tensor  Shape: torch.Size([2])  Min: 1.000     Max: 1.000     Mean: 1.000    dtype: torch.int64
==============================================================
==============================================================
        MASKS:
        Tensor  Shape: torch.Size([2, 512, 683])      Min: 0.000     Max: 1.000     Mean: 0.002    dtype: torch.uint8
==============================================================
==============================================================
        IMAGE_ID:
        Tensor  Shape: torch.Size([1])  Min: 0.000     Max: 0.000     Mean: 0.000    dtype: torch.int64
==============================================================
==============================================================
        AREA:
        Tensor  Shape: torch.Size([2])  Min: 864.000   Max: 935.000   Mean: 899.500   dtype: torch.float32
==============================================================
... ... 1 more items
```

From the preceding output, we can see that the masks' shape is 2 x 512 x 683, indicating there are two people in the image.

Note that, in the `__getitem__` method, we have as many masks and bounding boxes in an image as there are objects (instances) present within the image. Furthermore, because we only have two classes (the `Background` class and the `Person` class), we are specifying the `Person` class as 1.

By the end of this step, we have quite a lot of information in the output dictionary; that is, the object classes, bounding boxes, masks, the area of the masks, and if a mask corresponds to a crowd. All of this information is available in the `target` dictionary. For the training function that we are going to use, it is important for the data to be standardized in the format that the `torchvision.models.detection.maskrcnn_resnet50_fpn` class requires it to be in.

6. Next, we need to define the instance segmentation model (`get_model_instance_segmentation`). We are going to use a pre-trained model with only the heads reinitialized to predict two classes (background and person). First, we need to initialize a pre-trained model and replace the `box_predictor` and `mask_predictor` heads so that they can be learned from scratch:

```
def get_model_instance_segmentation(num_classes):
    # load an instance segmentation model pre-trained on
    # COCO
    model = torchvision.models.detection\
                       .maskrcnn_resnet50_fpn(pretrained=True)

    # get number of input features for the classifier
    in_features = model.roi_heads\
                       .box_predictor.cls_score.in_features
    # replace the pre-trained head with a new one
    model.roi_heads.box_predictor = FastRCNNPredictor(\
                                    in_features,num_classes)
    in_features_mask = model.roi_heads\
                       .mask_predictor.conv5_mask.in_channels
    hidden_layer = 256
    # and replace the mask predictor with a new one
    model.roi_heads.mask_predictor = MaskRCNNPredictor(\
                                       in_features_mask,\
                                    hidden_layer, num_classes)
    return model
```

`FastRCNNPredictor` expects two inputs – `in_features` (the number of input channels) and `num_classes` (the number of classes). Based on the number of classes to predict, the number of bounding box predictions is calculated – which is four times the number of classes.

`MaskRCNNPredictor` expects three inputs – `in_features_mask` (the number of input channels), `hidden_layer` (the number of channels in the output), and `num_classes` (the number of classes to predict).

details of the defined model can be obtained by specifying the following:

```
model = get_model_instance_segmentation(2).to(device)
model
```

The bottom half of the model (that is, without the backbone) would look like this:

```
(roi_heads): RoIHeads(
  (box_roi_pool): MultiScaleRoIAlign()
  (box_head): TwoMLPHead(
    (fc6): Linear(in_features=12544, out_features=1024, bias=True)
    (fc7): Linear(in_features=1024, out_features=1024, bias=True)
  )
  (box_predictor): FastRCNNPredictor(
    (cls_score): Linear(in_features=1024, out_features=2, bias=True)
    (bbox_pred): Linear(in_features=1024, out_features=8, bias=True)
  )
  (mask_roi_pool): MultiScaleRoIAlign()
  (mask_head): MaskRCNNHeads(
    (mask_fcn1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu1): ReLU(inplace=True)
    (mask_fcn2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu2): ReLU(inplace=True)
    (mask_fcn3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu3): ReLU(inplace=True)
    (mask_fcn4): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu4): ReLU(inplace=True)
  )
  (mask_predictor): MaskRCNNPredictor(
    (conv5_mask): ConvTranspose2d(256, 256, kernel_size=(2, 2), stride=(2, 2))
    (relu): ReLU(inplace=True)
    (mask_fcn_logits): Conv2d(256, 2, kernel_size=(1, 1), stride=(1, 1))
```

Note that the major difference between the Faster R-CNN network (which we trained in the previous chapter) and the Mask R-CNN model is in the `roi_heads` module, which itself contains multiple sub-modules. Let's take a look at what tasks they perform:

- `roi_heads`: Aligns the inputs taken from the FPN network and creates two tensors.
- `box_predictor`: Uses the outputs we obtained to predict classes and bounding box offsets for each RoI.
- `mask_roi_pool`: RoI then aligns the outputs coming from the FPN network.

- `mask_head`: Converts the aligned outputs obtained previously into feature maps that can be used to predict masks.
- `mask_predictor`: Takes the outputs from `mask_head` and predicts the final masks.

7. Fetch the dataset and dataloaders that correspond to the train and validation images:

```
dataset = MasksDataset(trn_items, get_transform(train=True), \
                                                      N=3000)
dataset_test = MasksDataset(val_items, \
                            get_transform(train=False), N=800)

# define training and validation data loaders
data_loader=torch.utils.data.DataLoader(dataset,batch_size=2,
\
                                 shuffle=True, num_workers=0, \
                                  collate_fn=utils.collate_fn)

data_loader_test = torch.utils.data.DataLoader(dataset_test, \
                                 batch_size=1, shuffle=False, \
                   num_workers=0,collate_fn=utils.collate_fn)
```

8. Define the model, parameters, and optimization criterion:

```
num_classes = 2
model = get_model_instance_segmentation(\
                       num_classes).to(device)
params = [p for p in model.parameters() if p.requires_grad]
optimizer = torch.optim.SGD(params, lr=0.005, \
                            momentum=0.9,weight_decay=0.0005)
# and a learning rate scheduler
lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer, \
                                               step_size=3, \
                                               gamma=0.1)
```

The defined pre-trained model architecture takes the image and the `targets` dictionary as input to reduce loss. A sample of the output that will be received from the model can be seen by running the following command:

```
# The following code is for illustration purpose only
model.eval()
pred = model(dataset[0][0][None].to(device))
inspect(pred[0])
```

The preceding code results in the following output:

```
=================================================================
Dict Of 4 items
=================================================================
        BOXES:
        Tensor  Shape: torch.Size([100, 4])     Min: 0.000      Max: 1024.000   Mean: 385.767   dtype: torch.float32
=================================================================
=================================================================
        LABELS:
        Tensor  Shape: torch.Size([100])        Min: 1.000      Max: 1.000      Mean: 1.000     dtype: torch.int64
=================================================================
=================================================================
        SCORES:
        Tensor  Shape: torch.Size([100])        Min: 0.491      Max: 0.648      Mean: 0.531     dtype: torch.float32
=================================================================
=================================================================
        MASKS:
        Tensor  Shape: torch.Size([100, 1, 692, 1024])  Min: 0.000      Max: 1.000      Mean: 0.012     dtype: torch.float32
=================================================================
```

Here, we can see a dictionary with bounding boxes (`BOXES`), classes corresponding to bounding boxes (`LABELS`), confidence scores corresponding to class predictions (`SCORES`), and the location of our mask instances (`MASKS`). As you can see, the model is hardcoded to return 100 predictions, which is reasonable since we shouldn't expect more than 100 objects in a typical image.

To fetch the number of instances that have been detected, we would use the following code:

```
# The following code is for illustration purpose only
pred[0]['masks'].shape
# torch.Size([100, 1, 536, 559])
```

The preceding code fetches a maximum of 100 mask instances (where the instances correspond to a non-background class) for an image (along with the dimensions corresponding to the image). For these 100 instances, it would also return the corresponding class label, bounding box, and the 100 corresponding confidence values of the class.

9. Train the model over increasing epochs:

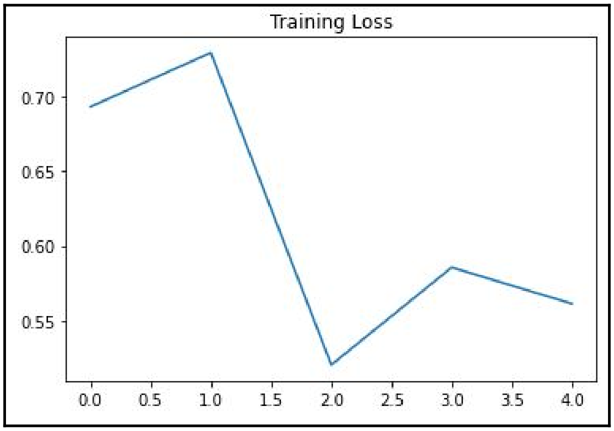
```
num_epochs = 5

trn_history = []
for epoch in range(num_epochs):
    # train for one epoch, printing every 10 iterations
    res = train_one_epoch(model, optimizer, data_loader, \
                          device, epoch, print_freq=10)
    trn_history.append(res)
```

```
        # update the learning rate
        lr_scheduler.step()
        # evaluate on the test dataset
        res = evaluate(model, data_loader_test, device=device)
```

By doing this, we can now overlay our masks over people in an image. We can log our training loss variation over increasing epochs as follows:

```
import matplotlib.pyplot as plt
plt.title('Training Loss')
losses =[np.mean(list(trn_history[i].meters['loss'].deque)) \
            for i in range(len(trn_history))]
plt.plot(losses)
```

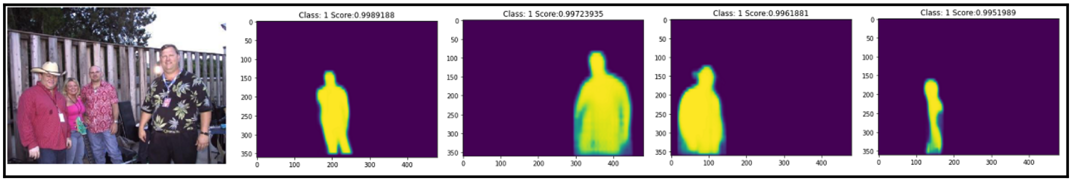
The preceding code results in the following output:



10. Predict on a test image:

```
model.eval()
im = dataset_test[0][0]
show(im)
with torch.no_grad():
    prediction = model([im.to(device)])
    for i in range(len(prediction[0]['masks'])):
        plt.imshow(Image.fromarray(prediction[0]['masks']\
                      [i, 0].mul(255).byte().cpu().numpy()))
        plt.title('Class: '+str(prediction[0]['labels']\
                   [i].cpu().numpy())+' Score:'+str(\
                  prediction[0]['scores'][i].cpu().numpy()))
        plt.show()
```

The preceding code results in the following output:



From the preceding image, we can see that we can successfully identify the four people in the image. Furthermore, the model predicts multiple other segments in the image (which we have not shown in the preceding output), though this is with low confidence.

Now that the model can detect instances well, let's run predictions on a custom image that is not present within the provided dataset.

11. Run predictions on a new image of your own:

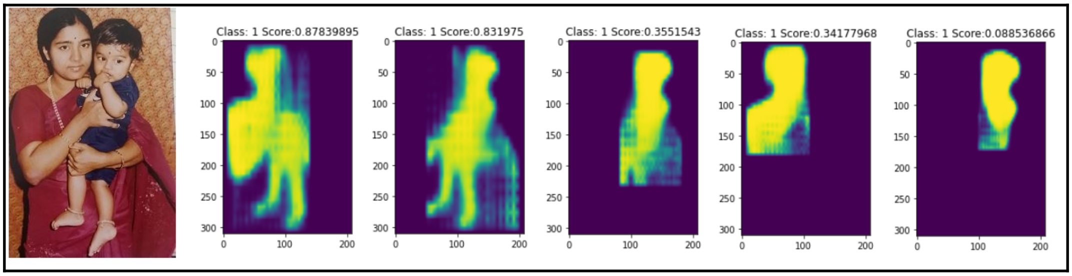
```
!wget https://www.dropbox.com/s/e92sui3a4ktvb4j/Hema18.JPG
img = Image.open('Hema18.JPG').convert("RGB")
from torchvision import transforms
pil_to_tensor = transforms.ToTensor()(img).unsqueeze_(0)
Image.fromarray(pil_to_tensor[0].mul(255)\
                            .permute(1, 2, 0).byte().numpy())
```

The input image is as follows:

- Fetch predictions on the input image:

```
model.eval()
with torch.no_grad():
    prediction = model([pil_to_tensor[0].to(device)])
    for i in range(len(prediction[0]['masks'])):
        plt.imshow(Image.fromarray(prediction[0]['masks']\
                        [i, 0].mul(255).byte().cpu().numpy()))
        plt.title('Class: '+str(prediction[0]\
                              ['labels'][i].cpu().numpy())+'\
        Score:'+str(prediction[0]['scores'][i].cpu().numpy()))
        plt.show()
```

The preceding code results in the following output:



Note that, in the preceding image, the trained model did not work as well as it did on the test images. This could be due to the following reasons:

- The people might not have been in such close proximity during training.
- The model might not have been trained on as many images where the classes of interest occupy the majority of the image.
- The images in the dataset that we have trained our model on have a different data distribution from the image being predicted on.

However, even though duplicate masks have been detected, having lower class scores in those regions (starting with the third mask) is a good indicator that there might be duplicates in predictions.

So far, we have learned about segmenting multiple instances of the `Person` class. In the next section, we will learn about what we need to tweak in the code we built in this section to segment multiple instances of multiple classes of objects in an image.

Predicting multiple instances of multiple classes

In the previous section, we learned about segmenting the `Person` class. In this section, we will learn about segmenting for person and table instances in one go by using the same model we built in the previous section. Let's get started:

> Given that the majority of the code remains the same as it was in the previous section, we will only explain the additional code within this section. While executing code, we encourage you to go through the `predicting_multiple_instances_of_multiple_classes.ipynb` notebook, which can be found in the `Chapter09` folder of this book's GitHub repository

1. Fetch images that contain the classes of interest – `Person` (class ID 4) and `Table` (class ID 6):

```
classes_list = [4,6]
annots = []
for ann in Tqdm(all_annots):
    _ann = read(ann, 1).transpose(2,0,1)
    r,g,b = _ann
    if np.array([num in np.unique(r) for num in \
                classes_list]).sum()==0: continue
    annots.append(ann)
from sklearn.model_selection import train_test_split
_annots = stems(annots)
trn_items, val_items = train_test_split(_annots, \
                                        random_state=2)
```

   In the preceding code, we are fetching the images that contain at least one of the classes of interest (`classes_list`).

2. Modify the `get_mask` method so that it returns both masks, as well as the classes that correspond to each mask in the `MasksDataset` class:

```
    def get_mask(self,path):
        an = read(path, 1).transpose(2,0,1)
        r,g,b = an
        cls = list(set(np.unique(r)).intersection({4,6}))
        masks = []
        labels = []
        for _cls in cls:
            nzs = np.nonzero(r==_cls)
```

```
            instances = np.unique(g[nzs])
            for ix,_id in enumerate(instances):
                masks.append(g==_id)
                labels.append(classes_list.index(_cls)+1)
        return np.array(masks), np.array(labels)
```

In the preceding code, we are fetching the classes of interest that exist within the image and are storing them in `cls`. Next, we are looping through each identified class (`cls`) and storing the locations where the Red channel values correspond to class (`cls`) in `nzs`. Next, we are fetching the instance IDs (`instances`) in those locations. Furthermore, we are appending `instances` to `masks` and the classes corresponding to instances in `labels` before returning the NumPy arrays for `masks` and `labels`.

3. Modify the `labels` object in the `__getitem__` method so that it contains labels that have been obtained from the `get_mask` method instead of filling it with `torch.ones`. The bold part of the following code is where this change was implemented on the `__getitem__` method in the previous section:

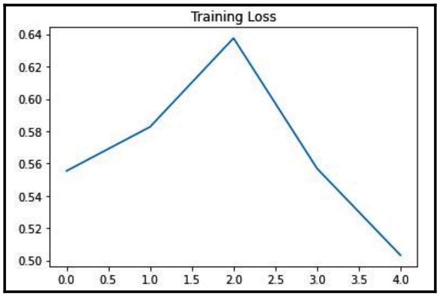
```
    def __getitem__(self, ix):
        _id = self.items[ix]
        img_path = f'images/training/{_id}.jpg'
        mask_path = f'annotations_instance/training/{_id}.png'
        masks, labels = self.get_mask(mask_path)
        #print(labels)
        obj_ids = np.arange(1, len(masks)+1)
        img = Image.open(img_path).convert("RGB")
        num_objs = len(obj_ids)
        boxes = []
        for i in range(num_objs):
            obj_pixels = np.where(masks[i])
            xmin = np.min(obj_pixels[1])
            xmax = np.max(obj_pixels[1])
            ymin = np.min(obj_pixels[0])
            ymax = np.max(obj_pixels[0])
            if (((xmax-xmin)<=10) | (ymax-ymin)<=10):
                xmax = xmin+10
                ymax = ymin+10
            boxes.append([xmin, ymin, xmax, ymax])
        boxes = torch.as_tensor(boxes, dtype=torch.float32)
        labels = torch.as_tensor(labels, dtype=torch.int64)
        masks = torch.as_tensor(masks, dtype=torch.uint8)
        area = (boxes[:, 3] - boxes[:, 1]) *
                    (boxes[:, 2] - boxes[:, 0])
        iscrowd = torch.zeros((num_objs,), dtype=torch.int64)
```

```
            image_id = torch.tensor([ix])
            target = {}
            target["boxes"] = boxes
            target["labels"] = labels
            target["masks"] = masks
            target["image_id"] = image_id
            target["area"] = area
            target["iscrowd"] = iscrowd
            if self.transforms is not None:
                img, target = self.transforms(img, target)
            return img, target
        def __len__(self):
            return self.N
        def choose(self):
            return self[randint(len(self))]
```

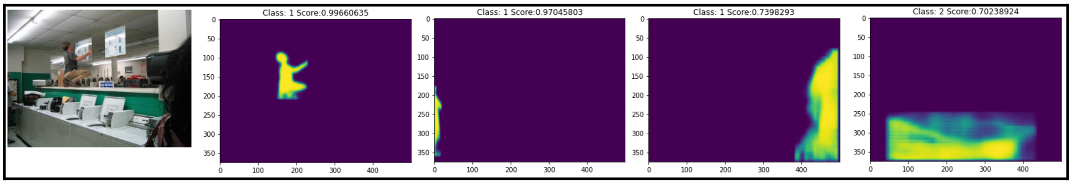
4. Specify that you have three classes instead of two while defining `model`:

```
num_classes = 3
model=get_model_instance_segmentation(num_classes).to(device)
```

Upon training the model, as we did in the previous section, we'll see that the variation of training loss over increasing epochs is as follows:



Furthermore, the predicted segments for a sample image that contains a person and a table are as follows:



From the preceding image, we can see that we are able to predict both classes using the same model. As an exercise, we encourage you to increase the number of classes and the number of epochs and see what results you get.

Summary

In this chapter, we learned how to leverage U-Net and Mask R-CNN to perform segmentation on top of images. We understood how the U-Net architecture can perform downscaling and upscaling on images using convolutions to retain the structure of the image, while still being able to predict masks around objects within an image. We then cemented our understanding of this using the road scene detection exercise, where we segmented the image into multiple classes. Next, we learned about RoI Align, which helps ensure that the issues with RoI pooling surrounding image quantization are addressed. After that, we learned about how Mask R-CNN works so that we could train models to predict instances of people in images, as well as instances of people and tables in an image.

Now that we have a good understanding of various object detection techniques and image segmentation techniques, in the next chapter, we will learn about applications that leverage the techniques we have learned about so far so that we can expand the number of classes that we will predict. In addition, we will also learn about the Detectron2 framework, which reduces code complexity while we're building Faster R-CNN and Mask R-CNN models.

Questions

1. How does upscaling help in the U-Net architecture?
2. Why do we need to have a fully convolutional network in U-Net?
3. How does RoI Align improve upon RoI pooling in Mask-RCNN?
4. What is the major difference between U-Net and Mask-RCNN for segmentation?
5. What is instance segmentation?

10
Applications of Object Detection and Segmentation

In previous chapters, we learned about various object detection techniques, such as the R-CNN family of algorithms, YOLO, SSD, and the U-Net and Mask R-CNN image segmentation algorithms. In this chapter, we will take our learning a step further – we will work on more realistic scenarios and learn about frameworks/architectures that are more optimized to solve detection and segmentation problems. We will start by leveraging the Detectron2 framework to train and detect custom objects present in an image. We will also predict the pose of humans present in an image using a pre-trained model. Furthermore, we will learn how to count the number of people in a crowd in an image and then learn about leveraging segmentation techniques to perform image colorization. Finally, we will learn about a modified version of YOLO to predict 3D bounding boxes around objects by using point clouds obtained from a LIDAR sensor.

By the end of this chapter, you will have learned about the following:

- Multi-object instance segmentation
- Human pose detection
- Crowd counting
- Image colorization
- 3D object detection with point clouds

Multi-object instance segmentation

In previous chapters, we learned about various object detection algorithms. In this section, we will learn about the Detectron2 platform (`https://ai.facebook.com/blog/-detectron2-a-pytorch-based-modular-object-detection-library-/`) before we implement it using the Google Open Images dataset. Detectron2 is a platform built by the Facebook team. Detectron2 includes high-quality implementations of state-of-the-art object detection algorithms, including DensePose of the Mask R-CNN model family. The original Detectron framework was written in Caffe2, while the Detectron2 framework is written using PyTorch.

Detectron2 supports a range of tasks related to object detection. Like the original Detectron, it supports object detection with boxes and instance segmentation masks, as well as human pose prediction. Beyond that, Detectron2 adds support for semantic segmentation and panoptic segmentation (a task that combines both semantic and instance segmentation). By leveraging Detectron2, we are able to build object detection, segmentation, and pose estimation in a few lines of code.

In this section, we will learn about the following:

1. Fetching data from the `open-images` repository
2. Converting the data into COCO format that Detectron2 accepts
3. Training the model for instance segmentation
4. Making inferences on new images

Let's go through each of these in the following sections.

Fetching and preparing data

We will be working on the images that are available in the Open Images dataset (which contains millions of images along with their annotations) provided by Google at `https://storage.googleapis.com/openimages/web/index.html`.

In this part of the code, we will learn about fetching only the required images and not the entire dataset. Note that this step is required, as the dataset size prohibits a typical user who might not have extensive resources from building a model:

The following code is available as `Multi_object_segmentation.ipynb` in the `Chapter10` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from and is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components from text.

1. Install the required packages:

```
!pip install -qU openimages torch_snippets
```

2. Download the required annotations files:

```
from torch_snippets import *
!wget -O train-annotations-object-segmentation.csv -q
https://storage.googleapis.com/openimages/v5/train-annotations
-object-segmentation.csv
!wget -O classes.csv -q \
https://raw.githubusercontent.com/openimages/dataset/master/di
ct.csv
```

3. Specify the classes that we want our model to predict (you can visit the Open Images website to see the list of all classes):

```
required_classes = 'person,dog,bird,car,elephant,football,\
jug,laptop,Mushroom,Pizza,Rocket,Shirt,Traffic sign,\
Watermelon,Zebra'
required_classes = [c.lower() for c in \
                        required_classes.lower().split(',')]

classes = pd.read_csv('classes.csv', header=None)
classes.columns = ['class','class_name']
classes = classes[classes['class_name'].map(lambda x: x \
                                 in required_classes)]
```

4. Fetch the image IDs and masks corresponding to `required_classes`:

```
from torch_snippets import *
df = pd.read_csv('train-annotations-object-segmentation.csv')

data = pd.merge(df, classes, left_on='LabelName',
                right_on='class')

subset_data = data.groupby('class_name').agg( \
                        {'ImageID': lambda x: list(x)[:500]})
```

```
subset_data = flatten(subset_data.ImageID.tolist())
subset_data = data[data['ImageID'].map(lambda x: x \
                                       in subset_data)]
subset_masks = subset_data['MaskPath'].tolist()
```

Given the vast amount of data, we are only fetching 500 images per class in `subset_data`. It is up to you whether you fetch a smaller or larger set of files per class and the list of unique classes (`required_classes`).

So far, we only have the `ImageId` and `MaskPath` values corresponding to an image. In the next steps, we will go ahead and download the actual images and masks from `open-images`.

5. Now that we have the subset of masks data to download, let's start the download. Open Images has 16 ZIP files for training masks. Each ZIP file will have only a few masks from `subset_masks`, so we will delete the rest after moving the required masks into a separate folder. This *download -> move -> delete* action will keep the memory footprint relatively small. We will have to run this step once for each of the 16 files:

```
!mkdir -p masks
for c in Tqdm('0123456789abcdef'):
    !wget -q \
https://storage.googleapis.com/openimages/v5/train-masks/train
-masks-{c}.zip
    !unzip -q train-masks-{c}.zip -d tmp_masks
    !rm train-masks-{c}.zip
    tmp_masks = Glob('tmp_masks', silent=True)
    items = [(m,fname(m)) for m in tmp_masks]
    items = [(i,j) for (i,j) in items if j in subset_masks]
    for i,j in items:
        os.rename(i, f'masks/{j}')
    !rm -rf tmp_masks
```

6. Download the images corresponding to `ImageId`:

```
masks = Glob('masks')
masks = [fname(mask) for mask in masks]

subset_data = subset_data[subset_data['MaskPath'].map(lambda \
                                              x: x in masks)]
subset_imageIds = subset_data['ImageID'].tolist()

from openimages.download import _download_images_by_id
!mkdir images
_download_images_by_id(subset_imageIds, 'train', './images/')
```

7. Zip all the images, masks, and ground truths and save them – just in case your session crashes, it is helpful to save and retrieve the file for later training. Once the ZIP file is created, ensure you save the file in your drive or download it. The file size ends up being around 2.5 GB:

```
import zipfile
files = Glob('images') + Glob('masks') + \
['train-annotations-object-segmentation.csv', 'classes.csv']
with zipfile.ZipFile('data.zip','w') as zipme:
    for file in Tqdm(files):
        zipme.write(file, compress_type=zipfile.ZIP_DEFLATED)
```

Finally, move the data into a single directory:

```
!mkdir -p train/
!mv images train/myData2020
!mv masks train/annotations
```

Given that there are so many moving components in object detection code, as a way of standardization, Detectron accepts a rigid data format for training. While it is possible to write a dataset definition and feed it to Detectron, it is easier (and more profitable) to save the entire training data in COCO format. This way, you can leverage other training algorithms, such as **detectron transformers** (**DETR**), with no change to the data whatsoever. First, we will start by defining the categories of classes.

8. Define the required categories in COCO format:

```
!pip install \
 git+git://github.com/waspinator/pycococreator.git@0.2.0
import datetime

INFO = {
    "description": "MyData2020",
    "url": "None",
    "version": "1.0",
    "year": 2020,
    "contributor": "sizhky",
    "date_created": datetime.datetime.utcnow().isoformat(' ')
}

LICENSES = [
    {
        "id": 1,
        "name": "MIT"
    }
]
```

```
CATEGORIES = [{'id': id+1, 'name': name.replace('/',''), \
               'supercategory': 'none'} \
              for id,(_,(name, clss_name)) in \
              enumerate(classes.iterrows())]
```

In the preceding code, in the definition of `CATEGORIES`, we are creating a new key called `supercategory`. To understand `supercategory`, let's go through an example: the `Man` and `Woman` classes are categories belonging to the `Person` supercategory. In our case, given that we are not interested in supercategories, we will specify it as `none`.

- Import the relevant packages and create an empty dictionary with the keys needed to save the COCO JSON file:

```
!pip install pycocotools
from pycococreatortools import pycococreatortools
from os import listdir
from os.path import isfile, join
from PIL import Image

coco_output = {
    "info": INFO,
    "licenses": LICENSES,
    "categories": CATEGORIES,
    "images": [],
    "annotations": []
}
```

- Set a few variables in place that contain the information on the image locations and annotation file locations:

```
ROOT_DIR = "train"
IMAGE_DIR, ANNOTATION_DIR = 'train/myData2020/', \
                            'train/annotations/'
image_files = [f for f in listdir(IMAGE_DIR) if \
               isfile(join(IMAGE_DIR, f))]
annotation_files = [f for f in listdir(ANNOTATION_DIR) if \
                    isfile(join(ANNOTATION_DIR, f))]
```

- Loop through each image filename and populate the `images` key in the `coco_output` dictionary:

```
image_id = 1
# go through each image
for image_filename in Tqdm(image_files):
    image = Image.open(IMAGE_DIR + '/' + image_filename)
    image_info = pycococreatortools\
```

```
                        .create_image_info(image_id, \
                    os.path.basename(image_filename), image.size)
        coco_output["images"].append(image_info)
        image_id = image_id + 1
```

9. Loop through each segmentation annotation and populate the `annotations` key in the `coco_output` dictionary:

```
    segmentation_id = 1
    for annotation_filename in Tqdm(annotation_files):
        image_id = [f for f in coco_output['images'] if \
                    stem(f['file_name']) == \
                    annotation_filename.split('_')[0]][0]['id']
        class_id = [x['id'] for x in CATEGORIES \
                    if x['name'] in annotation_filename][0]
        category_info = {'id': class_id, \
                        'is_crowd': 'crowd' in image_filename}
        binary_mask = np.asarray(Image.open(f'{ANNOTATION_DIR}/\
    {annotation_filename}').convert('1')).astype(np.uint8)

        annotation_info = pycococreatortools\
                        .create_annotation_info( \
                        segmentation_id, image_id, category_info,
                        binary_mask, image.size, tolerance=2)

        if annotation_info is not None:
            coco_output["annotations"].append(annotation_info)
            segmentation_id = segmentation_id + 1
```

10. Save `coco_output` in a JSON file:

```
    coco_output['categories'] = [{'id': id+1, 'name':clss_name, \
                                  'supercategory': 'none'} for \
                                 id,(_,(name, clss_name)) in \
                                 enumerate(classes.iterrows())]

    import json
    with open('images.json', 'w') as output_json_file:
        json.dump(coco_output, output_json_file)
```

With this, we have our files in COCO format, which can be easily used to train our model using the Detectron2 framework.

Training the model for instance segmentation

Training with Detectron2 can be done in a few steps:

1. Install the required Detectron2 packages. You should check your CUDA and PyTorch version before installing the right package. Colab contains PyTorch 1.7 and CUDA 10.1, as of the time of writing this book, so we will use the corresponding file:

```
!pip install detectron2 -f
https://dl.fbaipublicfiles.com/detectron2/wheels/cu101/torch1.
7/index.html
!pip install pyyaml==5.1 pycocotools>=2.0.1
```

   Restart Colab before proceeding to the next step.

2. Import the relevant `detectron2` packages:

```
from detectron2 import model_zoo
from detectron2.engine import DefaultPredictor
from detectron2.config import get_cfg
from detectron2.utils.visualizer import Visualizer
from detectron2.data import MetadataCatalog, DatasetCatalog
from detectron2.engine import DefaultTrainer
```

   - Given that we have restarted Colab, let's re-fetch the required classes:

```
from torch_snippets import *
required_classes= 'person,dog,bird,car,elephant,football,jug,\
laptop,Mushroom,Pizza,Rocket,Shirt,Traffic sign,\
Watermelon,Zebra'
required_classes = [c.lower() for c in \
                    required_classes.lower().split(',')]

classes = pd.read_csv('classes.csv', header=None)
classes.columns = ['class','class_name']
classes = classes[classes['class_name'].map(lambda \
                                x: x in required_classes)]
```

3. Register the created datasets using `register_coco_instances`:

```
from detectron2.data.datasets import register_coco_instances
register_coco_instances("dataset_train", {}, \
                        "images.json", "train/myData2020")
```

4. Define all the parameters in the `cfg` configuration file.

   Configuration (`cfg`) is a special Detectron object that holds all the relevant information for training a model:

```
cfg = get_cfg()
cfg.merge_from_file(model_zoo.get_config_file("COCO-\
InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml"))
cfg.DATASETS.TRAIN = ("dataset_train",)
cfg.DATASETS.TEST = ()
cfg.DATALOADER.NUM_WORKERS = 2
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-\
InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml") # pretrained
# weights
cfg.SOLVER.IMS_PER_BATCH = 2
cfg.SOLVER.BASE_LR = 0.00025 # pick a good LR
cfg.SOLVER.MAX_ITER = 5000 # instead of epochs, we train on
# 5000 batches
cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 512
cfg.MODEL.ROI_HEADS.NUM_CLASSES = len(classes)
```

   As you can see in the preceding code, you can to set up all the major hyperparameters needed for training the model. `merge_from_file` is importing all the core parameters from a pre-existing configuration file that was used for pre-training `mask_rccnn` with `FPN` as the backbone. This will also contain additional information on the pre-training experiment, such as the optimizer and loss functions. The hyperparameters that have been set, for our purpose, in `cfg` are self-explanatory.

5. Train the model:

```
os.makedirs(cfg.OUTPUT_DIR, exist_ok=True)
trainer = DefaultTrainer(cfg)
trainer.resume_or_load(resume=False)
trainer.train()
```

   With the preceding lines of code, we can train a model to predict classes, bounding boxes, and also the segmentation of objects belonging to the defined classes within our custom dataset.

   - Save the model in a folder:

```
!cp output/model_final.pth output/trained_model.pth
```

By this point, we have trained our model. In the next section, we will make inferences on a new image.

Making inferences on a new image

To perform inference on a new image, we load the path, set the probability threshold, and pass it through the `DefaultPredictor` method, as follows:

1. Load the weights with the trained model. Use the same `cfg` and load the model weights as shown in the following code:

```
cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_DIR, \
                                 "trained_model.pth")
```

2. Set the threshold for the probability of the object belonging to a certain class:

```
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.25
```

3. Define the `predictor` method:

```
predictor = DefaultPredictor(cfg)
```

4. Perform segmentation on the image of interest and visualize it:

In the following code, we are randomly plotting 30 training images (note that we haven't created validation data; we have left this as an exercise for you), but you can also load your own image path in place of `choose(files)`:

```
from detectron2.utils.visualizer import ColorMode
files = Glob('train/myData2020')
for _ in range(30):
    im = cv2.imread(choose(files))
    outputs = predictor(im)
    v = Visualizer(im[:, :, ::-1], scale=0.5, \
                   metadata=MetadataCatalog.get(\
                             "dataset_train"), \
                   instance_mode=ColorMode.IMAGE_BW
# remove the colors of unsegmented pixels.
# This option is only available for segmentation models
    )

    out = v.draw_instance_predictions(\
                        outputs["instances"].to("cpu"))
    show(out.get_image())
```

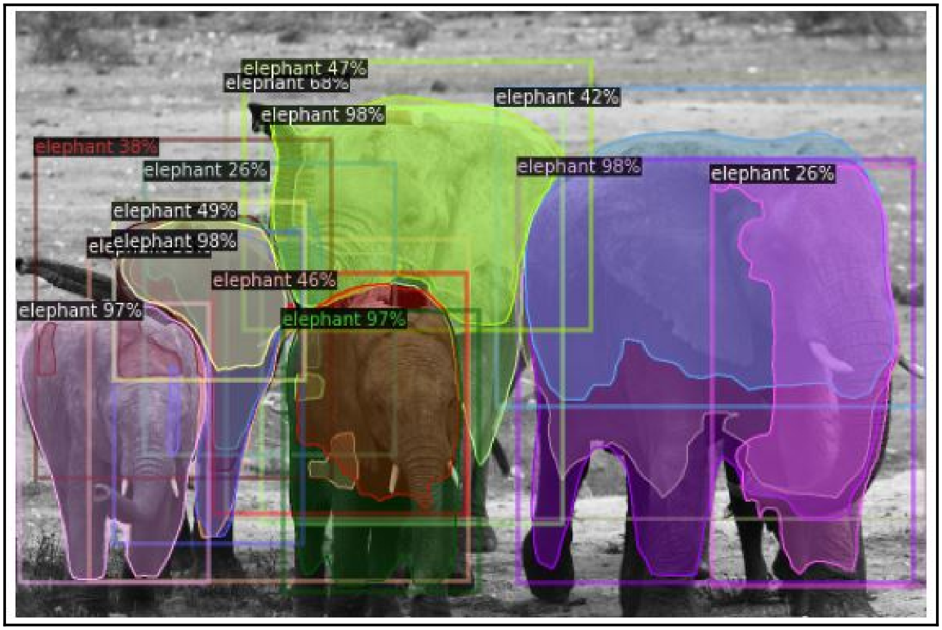
`Visualizer` is Detectron2's way of plotting object instances. Given that the predictions (present in the `outputs` variable) are a mere dictionary of tensors, `Visualizer` converts them into pixel information and draws them on an image.

Let's see what each input means:

- `im`: The image we want to visualize.
- `scale`: The size of the image when plotted. Here, we are asking it to shrink the image down to 50%.
- `metadata`: We need class-level information of the dataset, mainly the index-to-class mapping so that when we send the raw tensors as input to be plotted, the class will decode them into actual human-readable classes.
- `instance_mode`: We are asking the model to only highlight the segmented pixels.

Finally, once the class is created (in our example, it is `v`), we can ask it to draw instance predictions coming from the model and show the image.

The preceding code gives the following output:



From the preceding output, you can see that we are able to identify the pixels corresponding to the elephants fairly accurately.

Now that we have learned about leveraging Detectron2 to identify the pixels corresponding to classes within an image, in the next section, we will learn about leveraging Detectron2 to perform pose detection of humans present in an image.

Human pose detection

In the previous section, we learned about detecting multiple objects and segmenting them. In this section, we will learn about detecting multiple people in an image, as well as detecting the keypoints of various body parts of the people present in the image using Detectron2. Detecting keypoints comes in handy in multiple use cases. such as in sports analytics and security.

For this exercise, we will be leveraging the pre-trained keypoint model that is available in the configuration file:

> The following code is available as `Human_pose_detection.ipynb` in the `Chapter10` folder of the book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components from text.

1. Install all the requirements as shown in the previous section:

```
!pip install detectron2 -f \
https://dl.fbaipublicfiles.com/detectron2/wheels/cu101/torch1.
7/index.html
!pip install torch_snippets
!pip install pyyaml==5.1 pycocotools>=2.0.1

from torch_snippets import *
import detectron2
from detectron2.utils.logger import setup_logger
setup_logger()

from detectron2 import model_zoo
from detectron2.engine import DefaultPredictor
from detectron2.config import get_cfg
from detectron2.utils.visualizer import Visualizer
from detectron2.data import MetadataCatalog, DatasetCatalog
```

2. Fetch the configuration file and load the pre-trained keypoint detection model present in Detectron2:

```
cfg = get_cfg() # get a fresh new config
cfg.merge_from_file(model_zoo.get_config_file("COCO-\
Keypoints/keypoint_rcnn_R_50_FPN_3x.yaml"))
```

3. Specify the configuration parameters:

```
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.5 # set threshold
# for this model
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-\
Keypoints/keypoint_rcnn_R_50_FPN_3x.yaml")
predictor = DefaultPredictor(cfg)
```

4. Load the image that we want to predict:

```
from torch_snippets import read, resize
!wget -q https://i.imgur.com/ldzGSHk.jpg -O image.png
im = read('image.png',1)
im = resize(im, 0.5) # resize image to half its dimensions
```

5. Predict on the image and plot the keypoints:

```
outputs = predictor(im)
v = Visualizer(im[:,:,::-1], \
               MetadataCatalog.get(cfg.DATASETS.TRAIN[0]), \
               scale=1.2)
out = v.draw_instance_predictions(\
                outputs["instances"].to("cpu"))
import matplotlib.pyplot as plt
%matplotlib inline
plt.imshow(out.get_image())
```

The preceding code gives an output as follows:

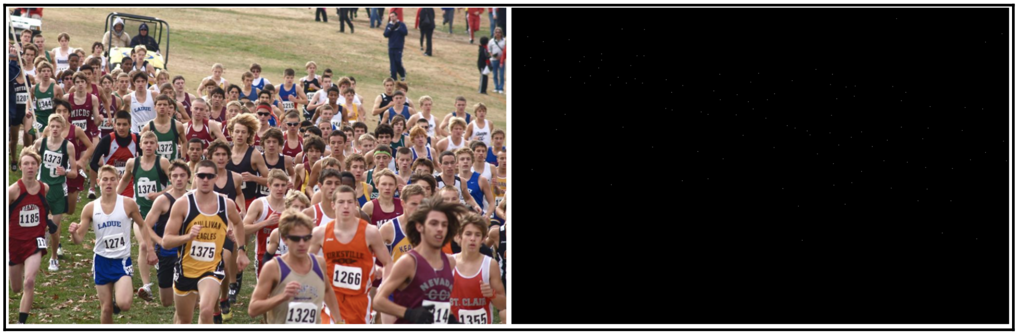
From the preceding output, we can see that the model is able to identify the various keypoints corresponding to the people in the image accurately.

In this section, we have learned how to perform keypoint detection using the Detectron2 platform. In the next section, we will learn about implementing a modified VGG architecture from scratch to estimate the number of people present in an image.

Crowd counting

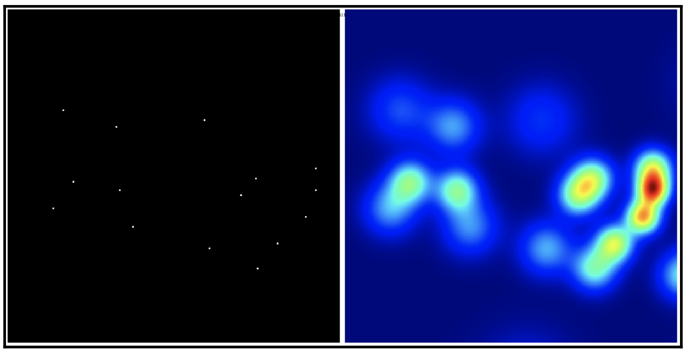
Imagine a scenario where you are given a picture of a crowd and are asked to estimate the number of people present in the image. A crowd counting model comes in handy in such a scenario. Before we go ahead and build a model to perform crowd counting, let's understand the data available and the model architecture first.

In order to train a model that predicts the number of people in an image, we will have to load the images first. The images should constitute the location of the center of the heads of all the people present in the image. A sample of the input image and the location of the center of the heads of the respective people in the image is as follows (source: ShanghaiTech dataset (`https://github.com/desenzhou/ShanghaiTechDataset`)):

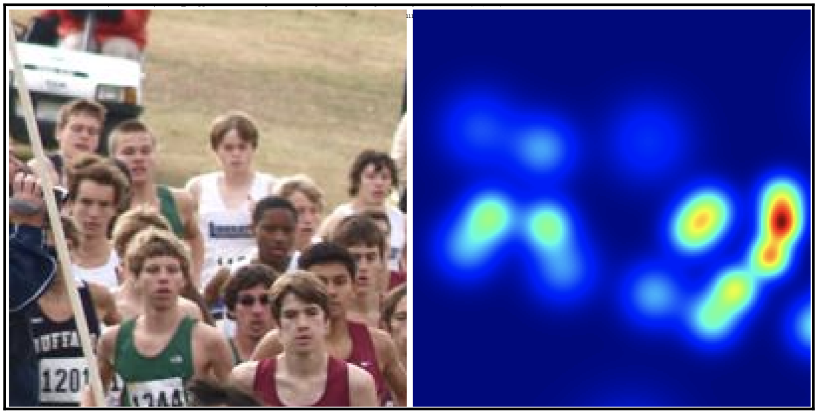


In the preceding example, the image representing ground truth (the image on the right – the center of the heads of the people present in the image) is extremely sparse. There are exactly *N* white pixels, where *N* is the number of people in the image. Let's zoom in to the top-left corner of the image and see the same map again:

In the next step, we transform the ground truth sparse image into a density map that represents the number of people in that region of the image:



The final input-output pair of the same crop would look like so:



The same for the entire image would look like so:

Note that, in the preceding image, when two people are close to each other, the pixel intensity is high. However, when a person is far away from the rest, the pixel density corresponding to the person is more evenly spread out, resulting in a lower pixel intensity corresponding to the person who is far away from the rest. Essentially, the heatmap is generated in such a way that the sum of the pixel values is equal to the number of people present in the image.

Now that we are in a position to accept an input image and the location of the center of the heads of the people in the image (which is processed to fetch the ground truth output heatmap), we will leverage the architecture detailed in the paper titled *CSRNet: Dilated Convolutional Neural Networks for Understanding the Highly Congested Scenes* to predict the number of people present in an image.

The model architecture (`https://arxiv.org/pdf/1802.10062.pdf`) is as follows:

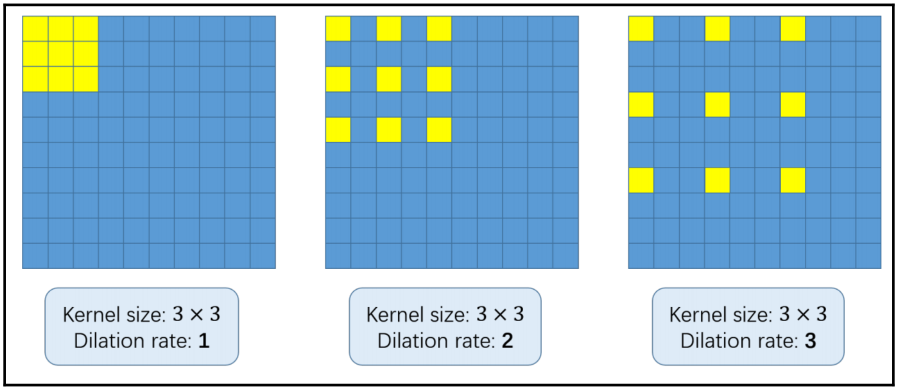
Configurations of CSRNet			
A	B	C	D
input(unfixed-resolution color image)			
front-end (fine-tuned from VGG-16)			
conv3-64-1 conv3-64-1			
max-pooling			
conv3-128-1 conv3-128-1			
max-pooling			
conv3-256-1 conv3-256-1 conv3-256-1			
max-pooling			
conv3-512-1 conv3-512-1 conv3-512-1			
back-end (four different configurations)			
conv3-512-1 conv3-512-1 conv3-512-1 conv3-256-1 conv3-128-1 conv3-64-1	conv3-512-2 conv3-512-2 conv3-512-2 conv3-256-2 conv3-128-2 conv3-64-2	conv3-512-2 conv3-512-2 conv3-512-2 conv3-256-4 conv3-128-4 conv3-64-4	conv3-512-4 conv3-512-4 conv3-512-4 conv3-256-4 conv3-128-4 conv3-64-4
conv1-1-1			

In the preceding structure of the model architecture, we are passing the image through four additional layers of convolutions after first passing it through the standard VGG-16 backbone. This output is passed through one of the four configurations and finally through a 1 x 1 x 1 convolution layer. We will be using the **A** configuration as it is the smallest.

Next, we perform **Mean Squared Error** (**MSE**) loss minimization on the output image to arrive at the optimal weight values while keeping track of the actual crowd count using MAE.

One additional detail of the architecture is that the authors used **dilated convolution** instead of normal convolution.

A typical dilated convolution looks as follows (image source: `https://arxiv.org/pdf/1802.10062.pdf`):



In the preceding, the diagram on the left represents a typical kernel that we have been working on so far. The second and third diagrams represent the dilated kernels, which have a gap between individual pixels. This way, the kernel has a larger receptive field. A large receptive field can come in handy as we need to understand the number of people near to a given person in order to estimate the pixel density corresponding to the person. We are using a dilated kernel (of nine parameters) instead of a normal kernel (which will have 49 parameters to be equivalent to a dilation rate of three kernels) to capture more information with fewer parameters.

With an understanding of how the model is to be architected in place, let's go ahead and code up the model to perform crowd counting in the following section. (For those of you looking to understand the working details, we suggest you go through the paper here: `https://arxiv.org/pdf/1802.10062.pdf`. The model we will be training in the following section is inspired by this paper.)

Coding up crowd counting

The strategy that we'll adopt to perform crowd counting is as follows:

1. Import the relevant packages and dataset.
2. The dataset that we will be working on – the ShanghaiTech dataset – already has the center of faces converted into a distribution based on Gaussian filter density, so we need not perform it again. Map the input image and the output Gaussian density map using a network.

3. Define a function to perform dilated convolution.
4. Define the network model and train on batches of data to minimize the MSE.

Let's go ahead and code up our strategy as follows:

> The following code is available as `crowd_counting.ipynb` in the `Chapter 10` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from and is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components from text.

1. Import the packages and download the dataset:

```
%%time
import os
if not os.path.exists('CSRNet-pytorch/'):
    !pip install -U scipy torch_snippets torch_summary
    !git clone https://github.com/sizhky/CSRNet-pytorch.git
    from google.colab import files
    files.upload() # upload kaggle.json
    !mkdir -p ~/.kaggle
    !mv kaggle.json ~/.kaggle/
    !ls ~/.kaggle
    !chmod 600 /root/.kaggle/kaggle.json
    print('downloading data...')
    !kaggle datasets download -d \
        tthien/shanghaitech-with-people-density-map/
    print('unzipping data...')
    !unzip -qq shanghaitech-with-people-density-map.zip

%cd CSRNet-pytorch
!ln -s ../shanghaitech_with_people_density_map
from torch_snippets import *
import h5py
from scipy import io
```

- Provide the location of the images (`image_folder`), the ground truth (`gt_folder`), and the heatmap folders (`heatmap_folder`):

```
part_A = Glob('shanghaitech_with_people_density_map/\
ShanghaiTech/part_A/train_data/');

image_folder = 'shanghaitech_with_people_density_map/\
```

```
ShanghaiTech/part_A/train_data/images/'
heatmap_folder = 'shanghaitech_with_people_density_map/\
ShanghaiTech/part_A/train_data/ground-truth-h5/'
gt_folder = 'shanghaitech_with_people_density_map/\
ShanghaiTech/part_A/train_data/ground-truth/'
```

2. Define the training and validation datasets and dataloaders:

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
tfm = T.Compose([
    T.ToTensor()
])

class Crowds(Dataset):
    def __init__(self, stems):
        self.stems = stems

    def __len__(self):
        return len(self.stems)

    def __getitem__(self, ix):
        _stem = self.stems[ix]
        image_path = f'{image_folder}/{_stem}.jpg'
        heatmap_path = f'{heatmap_folder}/{_stem}.h5'
        gt_path = f'{gt_folder}/GT_{_stem}.mat'

        pts = io.loadmat(gt_path)
        pts = len(pts['image_info'][0,0][0,0][0])

        image = read(image_path, 1)
        with h5py.File(heatmap_path, 'r') as hf:
            gt = hf['density'][:]
        gt = resize(gt, 1/8)*64
        return image.copy(), gt.copy(), pts

    def collate_fn(self, batch):
        ims, gts, pts = list(zip(*batch))
        ims = torch.cat([tfm(im)[None] for im in \
                        ims]).to(device)
        gts = torch.cat([tfm(gt)[None] for gt in \
                        gts]).to(device)
        return ims, gts, torch.tensor(pts).to(device)

    def choose(self):
        return self[randint(len(self))]

from sklearn.model_selection import train_test_split
trn_stems, val_stems = train_test_split(\
```

```
            stems(Glob(image_folder)), random_state=10)

trn_ds = Crowds(trn_stems)
val_ds = Crowds(val_stems)

trn_dl = DataLoader(trn_ds, batch_size=1, shuffle=True, \
                    collate_fn=trn_ds.collate_fn)
val_dl = DataLoader(val_ds, batch_size=1, shuffle=True, \
                    collate_fn=val_ds.collate_fn)
```

Note that the only addition to the typical dataset class that we have written so far is the lines of code in bold in the preceding code. We are resizing the ground truth as the output of our network would be shrunk to 1/8th of the original size, and hence we are multiplying the map by 64 so that the sum of the image pixels will be scaled back to the original crowd count.

3. Define the network architecture:

   - Define the function that enables dilated convolutions (`make_layers`):

```
import torch.nn as nn
import torch
from torchvision import models
from utils import save_net,load_net

def make_layers(cfg, in_channels = 3, batch_norm=False,
                dilation = False):
    if dilation:
        d_rate = 2
    else:
        d_rate = 1
    layers = []
    for v in cfg:
        if v == 'M':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        else:
            conv2d = nn.Conv2d(in_channels,v,kernel_size=3,\
                               padding=d_rate,
dilation=d_rate)
            if batch_norm:
                layers += [conv2d, nn.BatchNorm2d(v), \
                           nn.ReLU(inplace=True)]
            else:
                layers += [conv2d, nn.ReLU(inplace=True)]
            in_channels = v
    return nn.Sequential(*layers)
```

- Define the network architecture – `CSRNet`:

```
class CSRNet(nn.Module):
    def __init__(self, load_weights=False):
        super(CSRNet, self).__init__()
        self.seen = 0
        self.frontend_feat = [64, 64, 'M', 128, 128, 'M',256,
                              256, 256, 'M', 512, 512, 512]
        self.backend_feat = [512, 512, 512, 256, 128, 64]
        self.frontend = make_layers(self.frontend_feat)
        self.backend = make_layers(self.backend_feat,
                          in_channels = 512,dilation = True)
        self.output_layer = nn.Conv2d(64, 1, kernel_size=1)
        if not load_weights:
            mod = models.vgg16(pretrained = True)
            self._initialize_weights()
            items = list(self.frontend.state_dict().items())
            _items = list(mod.state_dict().items())
            for i in range(len(self.frontend.state_dict()\
                                   .items())):
                items[i][1].data[:] = _items[i][1].data[:]
    def forward(self,x):
        x = self.frontend(x)
        x = self.backend(x)
        x = self.output_layer(x)
        return x
    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.normal_(m.weight, std=0.01)
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
```

4. Define the functions to train and validate on a batch of data:

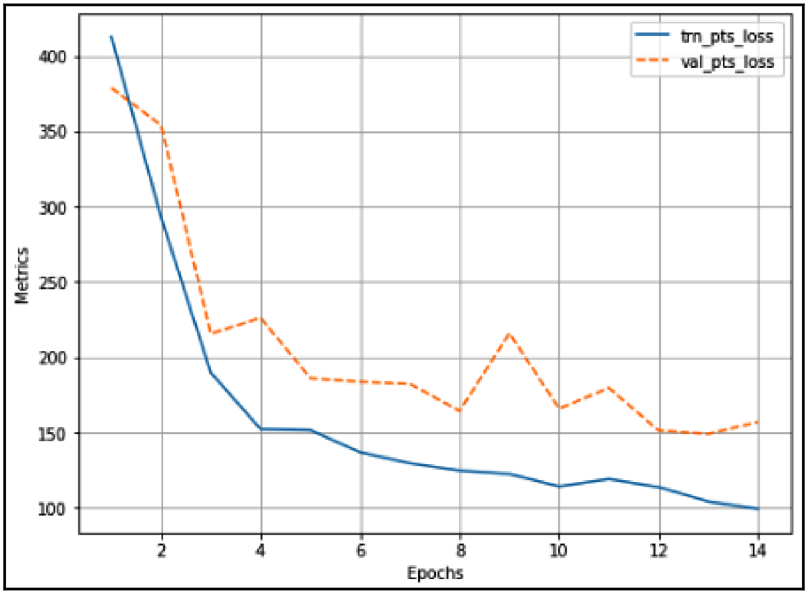
```
def train_batch(model, data, optimizer, criterion):
    model.train()
    optimizer.zero_grad()
    ims, gts, pts = data
    _gts = model(ims)
    loss = criterion(_gts, gts)
    loss.backward()
    optimizer.step()
    pts_loss = nn.L1Loss()(_gts.sum(), gts.sum())
    return loss.item(), pts_loss.item()
```

```
@torch.no_grad()
def validate_batch(model, data, criterion):
    model.eval()
    ims, gts, pts = data
    _gts = model(ims)
    loss = criterion(_gts, gts)
    pts_loss = nn.L1Loss()(_gts.sum(), gts.sum())
    return loss.item(), pts_loss.item()
```

5. Train the model over increasing epochs:

```
model = CSRNet().to(device)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=1e-6)
n_epochs = 20

log = Report(n_epochs)
for ex in range(n_epochs):
    N = len(trn_dl)
    for bx, data in enumerate(trn_dl):
        loss,pts_loss=train_batch(model, data, optimizer, \
                                        criterion)
        log.record(ex+(bx+1)/N, trn_loss=loss,
                        trn_pts_loss=pts_loss, end='\r')

    N = len(val_dl)
    for bx, data in enumerate(val_dl):
        loss, pts_loss = validate_batch(model, data, \
                                        criterion)
        log.record(ex+(bx+1)/N, val_loss=loss,
                    val_pts_loss=pts_loss, end='\r')

    log.report_avgs(ex+1)
    if ex == 10: optimizer = optim.Adam(model.parameters(), \
                                        lr=1e-7)
```

The preceding code results in a variation in the training and validation loss (here, the loss is the MAE of the crowd count), as follows:



From the preceding plot, we can see that we are off in our predictions by around 150 people. We can improve the model in the following two ways:

- By using data augmentation and training on crops of the original image
- By using a larger network (we used the **A** configuration, while **B**, **C**, and **D** are larger).

6. Make inferences on a new image:

    - Fetch a test image and normalize it:

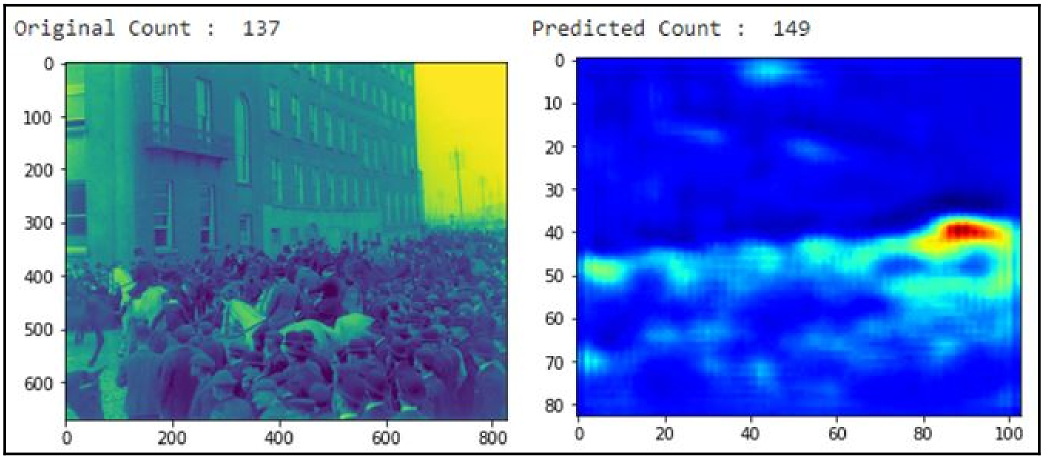
```
from matplotlib import cm as c
from torchvision import datasets, transforms
from PIL import Image
transform=transforms.Compose([
                    transforms.ToTensor(),transforms.Normalize(\
                              mean=[0.485, 0.456, 0.406],\
                              std=[0.229, 0.224, 0.225]),\
                     ])

test_folder = 'shanghaitech_with_people_density_map/\
```

```
ShanghaiTech/part_A/test_data/'
imgs = Glob(f'{test_folder}/images')
f = choose(imgs)
print(f)
img = transform(Image.open(f).convert('RGB')).to(device)
```

- Pass the image through the trained model:

```
output = model(img[None])
print("Predicted Count : ", int(output.detach().cpu()\
                                .sum().numpy()))
temp = np.asarray(output.detach().cpu()\
                    .reshape(output.detach().cpu()\
                    .shape[2],output.detach()\
                    .cpu().shape[3]))
plt.imshow(temp,cmap = c.jet)
plt.show()
```

The preceding code results in a heatmap (right image) of the input image (left image):



From the preceding output, we can see that the model predicted the heatmap reasonably accurately and the prediction count of people is close to the actual value.

In the next section, we will leverage a U-Net architecture to colorize an image.

Image colorization

Imagine a scenario where you are given a bunch of black-and-white images and are asked to turn them into color images. How would you solve this problem? One way to solve this is by using a pseudo-supervised pipeline where we take a raw image, convert it into black and white, and treat them as input-output pairs. We will demonstrate this by leveraging the CIFAR-10 dataset to perform colorization on images.

The strategy that we will adopt as we code up the image colorization network is as follows:

1. Take the original color image in the training dataset and convert it into grayscale to fetch the input (grayscale) and output (original colored image) combination.
2. Normalize the input and output.
3. Build a U-Net architecture.
4. Train the model over increasing epochs.

With the preceding strategy in place, let's go ahead and code up the model as follows:

> The following code is available as `Image colorization.ipynb` in the `Chapter 10` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Install the required packages and import them:

```
!pip install torch_snippets
from torch_snippets import *
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

2. Download the dataset and define the training and validation datasets and dataloaders:

    - Download the dataset:

```
from torchvision import datasets
import torch
data_folder = '~/cifar10/cifar/'
datasets.CIFAR10(data_folder, download=True)
```

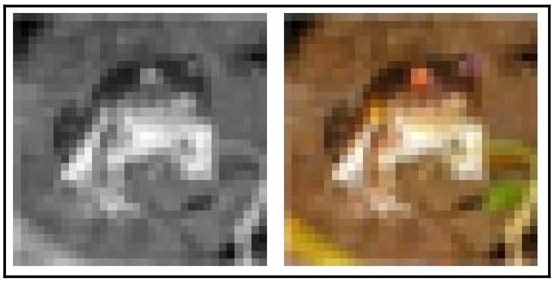
- Define the training and validation datasets and dataloaders:

```
class Colorize(torchvision.datasets.CIFAR10):
    def __init__(self, root, train):
        super().__init__(root, train)
    def __getitem__(self, ix):
        im, _ = super().__getitem__(ix)
        bw = im.convert('L').convert('RGB')
        bw, im = np.array(bw)/255., np.array(im)/255.
        bw, im = [torch.tensor(i).permute(2,0,1)\
                    .to(device).float() for i in [bw,im]]
        return bw, im

trn_ds = Colorize('~/cifar10/cifar/', train=True)
val_ds = Colorize('~/cifar10/cifar/', train=False)

trn_dl = DataLoader(trn_ds, batch_size=256, shuffle=True)
val_dl = DataLoader(val_ds, batch_size=256, shuffle=False)
```

A sample of the input and output images is as follows:

```
a,b = trn_ds[0]
subplots([a,b], nc=2)
```

The preceding code results in the following output:



Note that CIFAR-10 has images that are 32 x 32 in shape.

3. Define the network architecture:

```
class Identity(nn.Module):
    def __init__(self):
        super().__init__()
    def forward(self, x):
        return x

class DownConv(nn.Module):
```

```
    def __init__(self, ni, no, maxpool=True):
        super().__init__()
        self.model = nn.Sequential(
            nn.MaxPool2d(2) if maxpool else Identity(),
            nn.Conv2d(ni, no, 3, padding=1),
            nn.BatchNorm2d(no),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(no, no, 3, padding=1),
            nn.BatchNorm2d(no),
            nn.LeakyReLU(0.2, inplace=True),
        )
    def forward(self, x):
        return self.model(x)

class UpConv(nn.Module):
    def __init__(self, ni, no, maxpool=True):
        super().__init__()
        self.convtranspose = nn.ConvTranspose2d(ni, no, \
                                                2, stride=2)
        self.convlayers = nn.Sequential(
            nn.Conv2d(no+no, no, 3, padding=1),
            nn.BatchNorm2d(no),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(no, no, 3, padding=1),
            nn.BatchNorm2d(no),
            nn.LeakyReLU(0.2, inplace=True),
        )
    def forward(self, x, y):
        x = self.convtranspose(x)
        x = torch.cat([x,y], axis=1)
        x = self.convlayers(x)
        return x

class UNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.d1 = DownConv( 3, 64, maxpool=False)
        self.d2 = DownConv( 64, 128)
        self.d3 = DownConv( 128, 256)
        self.d4 = DownConv( 256, 512)
        self.d5 = DownConv( 512, 1024)
        self.u5 = UpConv (1024, 512)
        self.u4 = UpConv ( 512, 256)
        self.u3 = UpConv ( 256, 128)
        self.u2 = UpConv ( 128, 64)
        self.u1 = nn.Conv2d(64, 3, kernel_size=1, stride=1)

    def forward(self, x):
```

```
        x0 = self.d1( x) # 32
        x1 = self.d2(x0) # 16
        x2 = self.d3(x1) # 8
        x3 = self.d4(x2) # 4
        x4 = self.d5(x3) # 2
        X4 = self.u5(x4, x3)# 4
        X3 = self.u4(X4, x2)# 8
        X2 = self.u3(X3, x1)# 16
        X1 = self.u2(X2, x0)# 32
        X0 = self.u1(X1) # 3
        return X0
```

4. Define the model, optimizer, and loss function:

```
def get_model():
    model = UNet().to(device)
    optimizer = optim.Adam(model.parameters(), lr=1e-3)
    loss_fn = nn.MSELoss()
    return model, optimizer, loss_fn
```

5. Define the functions to train and validate on a batch of data:

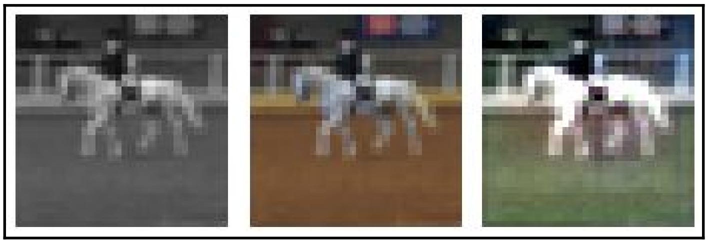
```
def train_batch(model, data, optimizer, criterion):
    model.train()
    x, y = data
    _y = model(x)
    optimizer.zero_grad()
    loss = criterion(_y, y)
    loss.backward()
    optimizer.step()
    return loss.item()

@torch.no_grad()
def validate_batch(model, data, criterion):
    model.eval()
    x, y = data
    _y = model(x)
    loss = criterion(_y, y)
    return loss.item()
```

6. Train the model over increasing epochs:

```
model, optimizer, criterion = get_model()
exp_lr_scheduler = optim.lr_scheduler.StepLR(optimizer, \
                                    step_size=10, gamma=0.1)

_val_dl = DataLoader(val_ds, batch_size=1, shuffle=True)
```

```
n_epochs = 100
log = Report(n_epochs)
for ex in range(n_epochs):
    N = len(trn_dl)
    for bx, data in enumerate(trn_dl):
        loss = train_batch(model, data, optimizer, criterion)
        log.record(ex+(bx+1)/N, trn_loss=loss, end='\r')
        if (bx+1)%50 == 0:
            for _ in range(5):
                a,b = next(iter(_val_dl))
                _b = model(a)
                subplots([a[0], b[0], _b[0]], nc=3, \
                          figsize=(5,5))

    N = len(val_dl)
    for bx, data in enumerate(val_dl):
        loss = validate_batch(model, data, criterion)
        log.record(ex+(bx+1)/N, val_loss=loss, end='\r')
    exp_lr_scheduler.step()
    if (ex+1) % 5 == 0: log.report_avgs(ex+1)

    for _ in range(5):
        a,b = next(iter(_val_dl))
        _b = model(a)
        subplots([a[0], b[0], _b[0]], nc=3, figsize=(5,5))

log.plot_epochs()
```

The preceding code generates an output as follows:



From the preceding output, we can see that the model is able to color the grayscale image reasonably well.

So far, we have learned about leveraging Detectron2 for segmentation and keypoint detection, dilated convolutions in crowd counting, and U-Net in image colorization. In the next section, we will learn about leveraging YOLO for 3D object detection.

3D object detection with point clouds

So far, we have learned how to predict a bounding rectangle on 2D images using algorithms that have the core underlying concept of anchor boxes. We will now learn how the same concept can be extended to predict 3D bounding boxes around objects.

In a self-driving car, tasks such as pedestrian/obstacle detection and route planning cannot happen without knowing the environment. Predicting 3D object locations along with their orientations becomes an important task. Not only is the 2D bounding box around obstacles important, but also knowing the distance from the object, height, width, and orientation of the obstacle are critical to navigating safely in the 3D world.

In this section, we will learn how YOLO is used to predict the 3D orientation and position of cars and pedestrians on a real-world dataset.

> The instructions for downloading the data, training, and testing sets are all given in this GitHub repo: `https://github.com/sizhky/Complex-YOLOv4-Pytorch/blob/master/README.md#training-instructions`. Given that there are very few openly available 3D datasets, we have chosen the most-used dataset for this exercise, which you still need to register for download. We have provided the instructions for registration in the preceding link as well.

Theory

One of the well-known sensors for collecting real-time 3D data is **LIDAR** (**Light Detection and Ranging**). It is a laser mounted on a rotating apparatus that fires beams of lasers hundreds of times every second. Another sensor receives the reflection of the laser from surrounding objects and calculates how far the laser has traveled before encountering an obstruction. Doing this in all directions of the car will result in a 3D point cloud of distances that is reflective of the environment itself. In the dataset that we will learn about, we have obtained the 3D point clouds from specific hardware known as `velodyne`. Let's understand how input and output are encoded for 3D object detection.

Input encoding

Our raw inputs are going to be 3D point clouds presented to us in the form of `.bin` files. Each can be loaded as a NumPy array using `np.fromfile(<filepath>)` and here's how the data looks for a sample file (these files are found in the `dataset/.../training/velodyne` directory after downloading and moving the raw files as per the GitHub repo instructions):

```
files = Glob('training/velodyne')
F = choose(files)
pts = np.fromfile(F, dtype=np.float32).reshape(-1, 4)
pts
```

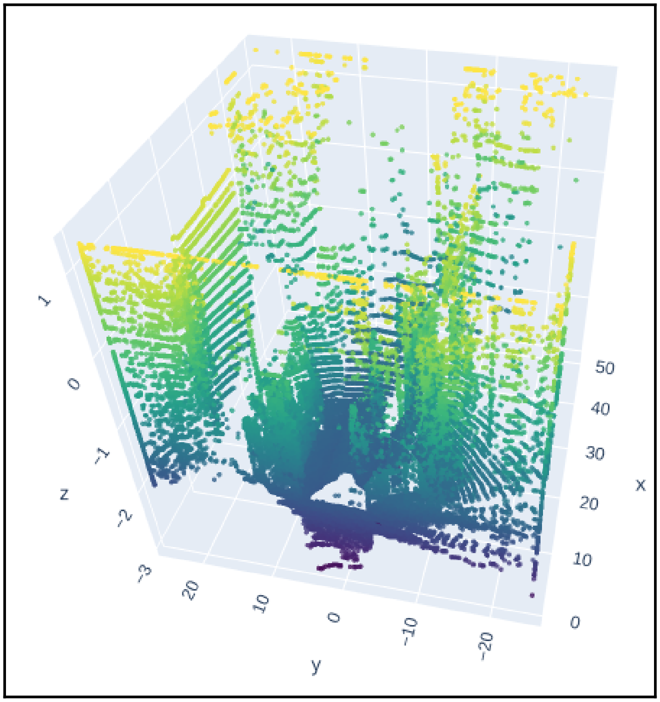
The preceding code gives the following output:

```
array([[62.502,  8.628,  2.343,  0.   ],
       [62.468,  8.824,  2.342,  0.   ],
       [66.793, 10.832,  2.497,  0.   ],
       ...,
       [ 3.75 , -1.418, -1.753,  0.25 ],
       [ 3.759, -1.409, -1.756,  0.32 ],
       [ 3.767, -1.398, -1.758,  0.   ]], dtype=float32)
```

This can be visualized as follows:

```
# take the points and remove faraway points
x,y,z = np.clip(pts[:,0], 0, 50),
        np.clip(pts[:,1], -25, 25),
        np.clip(pts[:,2],-3, 1.27)

fig = go.Figure(data=[go.Scatter3d(\
        x=x, y=y, z=z, mode='markers',
        marker=dict(
            size=2,
            color=z, # set color to a list of desired values
            colorscale='Viridis', # choose a colorscale
            opacity=0.8
        )
    )])

fig.update_layout(margin=dict(l=0, r=0, b=0, t=0))
fig.show()
```

The preceding code results in the following output:

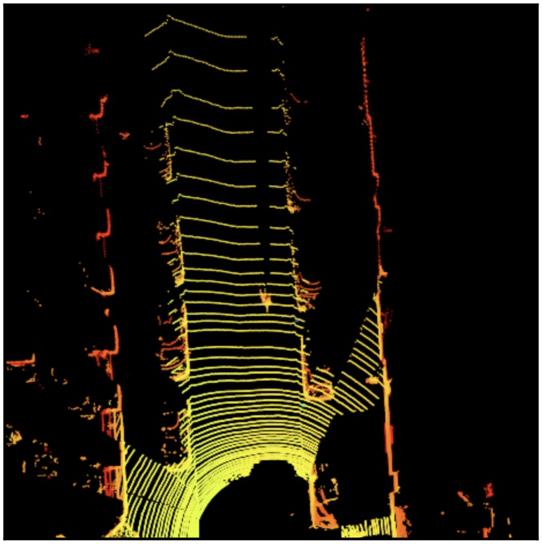


We can convert this information into an image of a bird's-eye view by performing the following steps.

1. Project the 3D point cloud onto the XY plane (ground) and split it into a grid with a resolution of 8 cm^2 per grid cell.
2. For each cell, compute the following and associate them with the specified channel:

   - Red channel: The height of the highest point in the grid
   - Green channel: The intensity of the highest point in the grid
   - Blue channel: The number of points in the grid divided by 64 (which is a normalizing factor)

For example, the reconstructed top view of the cloud may look like this:



You can clearly see the "shadows" in the image, indicating that there is an obstacle.

This is how we create an image from the LIDAR point cloud data.

We have taken 3D point clouds as the raw input and obtained the bird's-eye image as the output. This is the preprocessing step necessary to create the image that is going to be the input for the YOLO model.

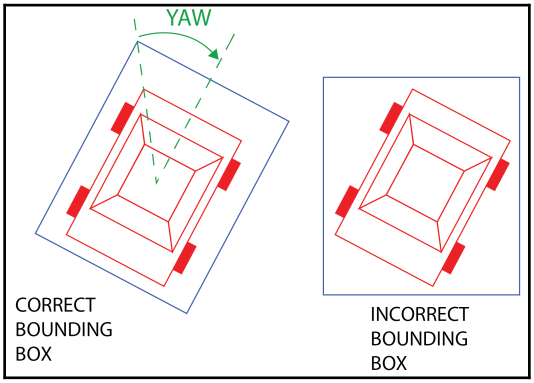
Output encoding

Now that we have the bird's-eye image (of the 3D point cloud) as input to the model, the model needs to predict the following real-world features:

- What the object (**class**) present in the image is
- How far the object is (in meters) from the car on the east-west axis (**x**)
- How far the object is (in meters) from the car on the north-south axis (**y**)
- What the orientation of the object (**yaw**) is
- How big the object is (the **length** and **width** of the object in meters)

It is possible to predict the bounding box in the pixel coordinate system (of the bird's-eye image). But it does not have any real-world significance as the predictions would still be in pixel space (in a bird's-eye view). In this case, we need to convert these pixel coordinate (of the bird's-eye view) bounding box predictions into real-world coordinates in meters. To avoid additional steps during postprocessing, we are directly predicting the real-world values.

Furthermore, in a realistic scenario, the object can be oriented in any direction. If we only calculate the length and width, it will not be sufficient to describe the tight bounding box. An example of such a scenario is as follows:



To get a tight bounding box for the object, we also need the information on which direction the obstacle is facing, and hence we also need the additional yaw parameter. Formally, it is the orientation made by the object with the north-south axis.

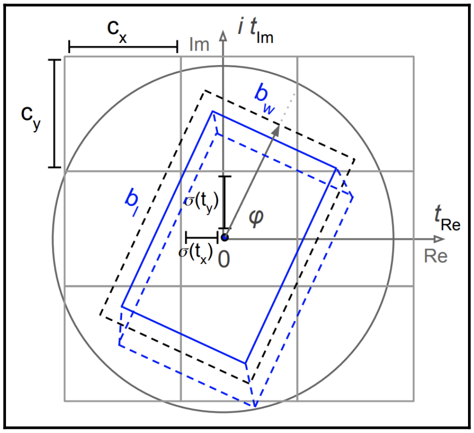
First, the YOLO model uses an anchor grid of 32 x 64 cells (more width than height) taking into consideration that the car's dashcam (and hence LIDAR) views are more wide than tall. The model uses two losses for the task. The first one is the normal YOLO loss (which is responsible for predicting the *x*, *y*, *l*, and *w* classes) we learned about in `Chapter 8`, *Advanced Object Detection*, and another loss called the EULER loss, which exclusively predicts the yaw. Formally, the set of equations to predict the final bounding boxes from the model's outputs are as follows:

$$b_x = \sigma(\mathbf{t_x}) + c_x$$
$$b_y = \sigma(\mathbf{t_y}) + c_y$$
$$b_w = p_w e^{\mathbf{t_w}}$$
$$b_l = p_l e^{\mathbf{t_l}}$$
$$b_\varphi = \arctan2(\mathbf{t_{Im}}, \mathbf{t_{Re}})$$

Here, b_x, b_y, b_w, b_l, and b_φ are the x and z coordinate values, the width, the length, and the yaw of the obstacle, respectively.
t_x, t_y, t_w, t_l, t_{Im}, and t_{Re} are the six regression values that being getting predicted from YOLO.
c_x and c_y are the positions of the center of the grid cell within the 32 x 64 matrix and p_w and p_l are pre-defined priors chosen by taking the average widths and lengths of cars and pedestrians. Furthermore, there are five priors (anchor boxes) in the implementation.

The height of each object of the same class is assumed as a fixed number.

Refer to the illustration given here, which shows this pictorally (image source: `https://arxiv.org/pdf/1803.06199.pdf`):



The total loss is calculated as follows:

$$Loss = Loss_{YOLO} + Loss_{EULER}$$

You already know $Loss_{YOLO}$ from the previous chapter (using t_x, t_y, t_w, and t_l as the targets). Also, note the following:

$$LOSS_{EULER} = \sum_{all\ objects} \sum^{all\ grid\ cells} f(object, cell)$$

$$f(object, cell) = \begin{cases} (t_{Im} - \hat{t}_{Im})^2 + (t_{Re} - \hat{t}_{re})^2 & if\ object\ is\ in\ cell \\ 0 & otherwise \end{cases}$$

Now that we have understood how the fundamentals of 3D object detection remain the same as that of 2D object detection (but with more number of parameters to predict) and the input-output pairs of this task, let's leverage an existing GitHub repo to train our model.

For more details on 3D object detection, refer to the paper *Complex-YOLO* at `https://arxiv.org/pdf/1803.06199.pdf`.

Training the YOLO model for 3D object detection

The coding effort is largely taken away from the user due to the standardized code. Much like Detectron2, we can train and test the algorithm by ensuring that the data is in the right format in the right location. Once that is ensured, we can train and test the code with a minimal number of lines.

We need to clone the `Complex-YOLOv4-Pytorch` repository first:

```
$ git clone https://github.com/sizhky/Complex-YOLOv4-Pytorch
```

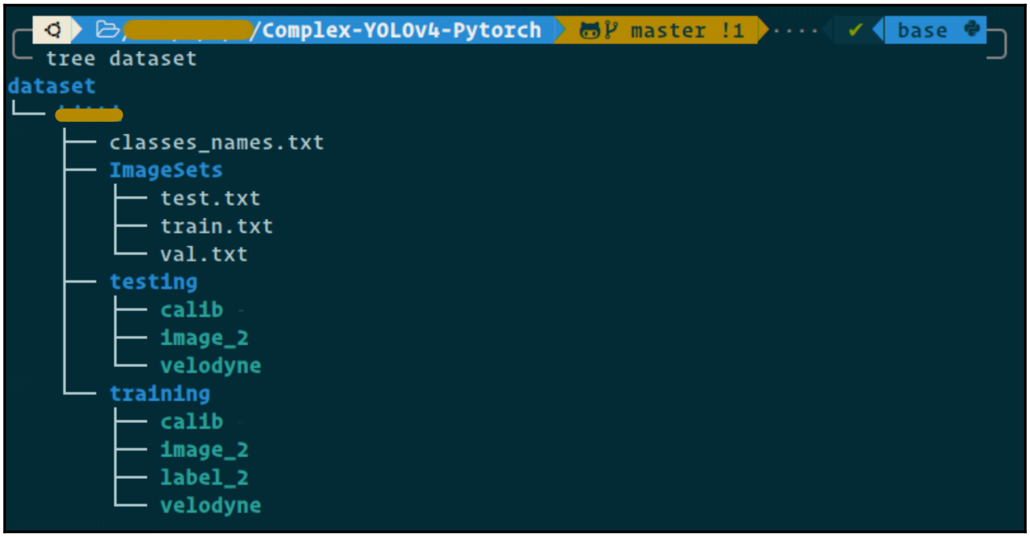
Follow the instructions in the `README.md` file to download and move the datasets to the right locations.

The instructions for downloading the data, training, and testing sets are all given in this GitHub repo: `https://github.com/sizhky/Complex-YOLOv4-Pytorch/blob/master/README.md#training-instructions`.

Given that there are very few openly available 3D datasets, we have chosen the most-used dataset for this exercise, which you still need to register for download. We also give the instructions for registration in the preceding link.

Data format

We can use any 3D point cloud data with ground truths for this exercise. Refer to the README file on the GitHub repo for more instructions on how to download and move the data. The data needs to be stored in the following format in the root directory:



The three folders that are new to us are `velodyne`, `calib`, and `label_2`:

- `velodyne` contains a list of `.bin` files that encode 3D point cloud information for corresponding images present in the `image_2` folder.
- `calib` contains calibration files corresponding to each point cloud. The 3D coordinates from the LIDAR point cloud coordinate system can be projected onto the camera coordinate system – that is, the image – by using the 3 x 4 projection matrix present in each file in the `calib` folder. Essentially, the LIDAR sensor captures the points that are slightly offset from what the camera is capturing. This offset is due to the fact that both sensors are mounted a few inches apart from each other. Knowing the right offsets will help us to rightfully project bounding boxes and 3D points on to the image from the camera.
- `label_2` contains the ground truths (one ground truth per line) for each image in the form of 15 values that are explained in the following table:

COLUMN	Example	Description	Range
type	Pedestrian	Class	Car/Pedestrian
truncation	0	Is object leaving image boundaries	0/1
occlusion	0	Is object occluded (0=fully visible, 1=partial, 2=mostly, 3=unknown)	0,1,2,3
alpha	-0.2	Observation angle	-pi to pi
x1	712.4	bbox	Image-coordinates
y1	143	bbox	Image-coordinates
x2	810.73	bbox	Image-coordinates
y2	307.92	bbox	Image-coordinates
h	1.89	height	meters
w	0.48	width	meters
l	1.2	length	meters
x	1.84	object location from camera	meters
y	1.47	object location from camera	meters
z	8.41	object location from camera	meters
ry	0.01	rotation of object around its own y-axis	-pi to pi

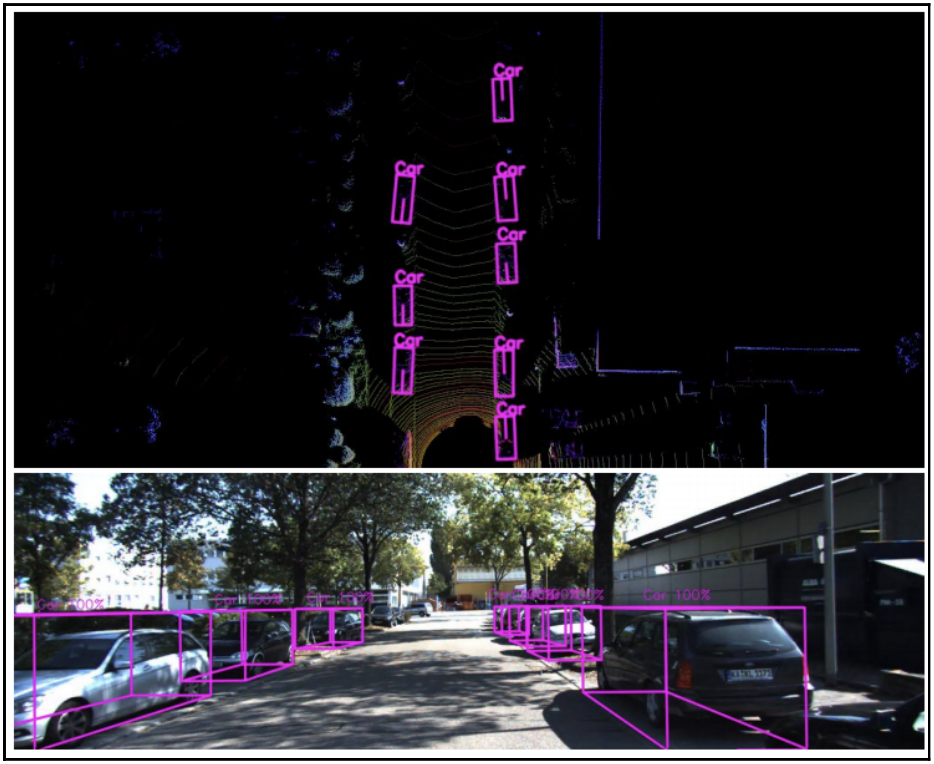
Note that our target columns are type (class), *w*, *l*, *x*, *z*, and *ry* (yaw) among the ones seen here. We will ignore the rest of the values for this task.

Data inspection

We can verify that the data is downloaded properly by running the following:

```
$ cd Complex-YOLOv4-Pytorch/src/data_process
$ python kitti_dataloader.py --output-width 600
```

The preceding code shows multiple images, one image at a time. The following is one such example (image source: `https://arxiv.org/pdf/1803.06199.pdf`):



Now that we are able to download and view a few images, in the next section, we will learn about training the model to predict 3D bounding boxes.

Training

The training code is wrapped in a single Python file and can be called as follows:

```
$ cd Complex-YOLOv4-Pytorch/src
$ python train.py --gpu_idx 0 --batch_size 2 --num_workers 4 \
                  --num_epochs 5
```

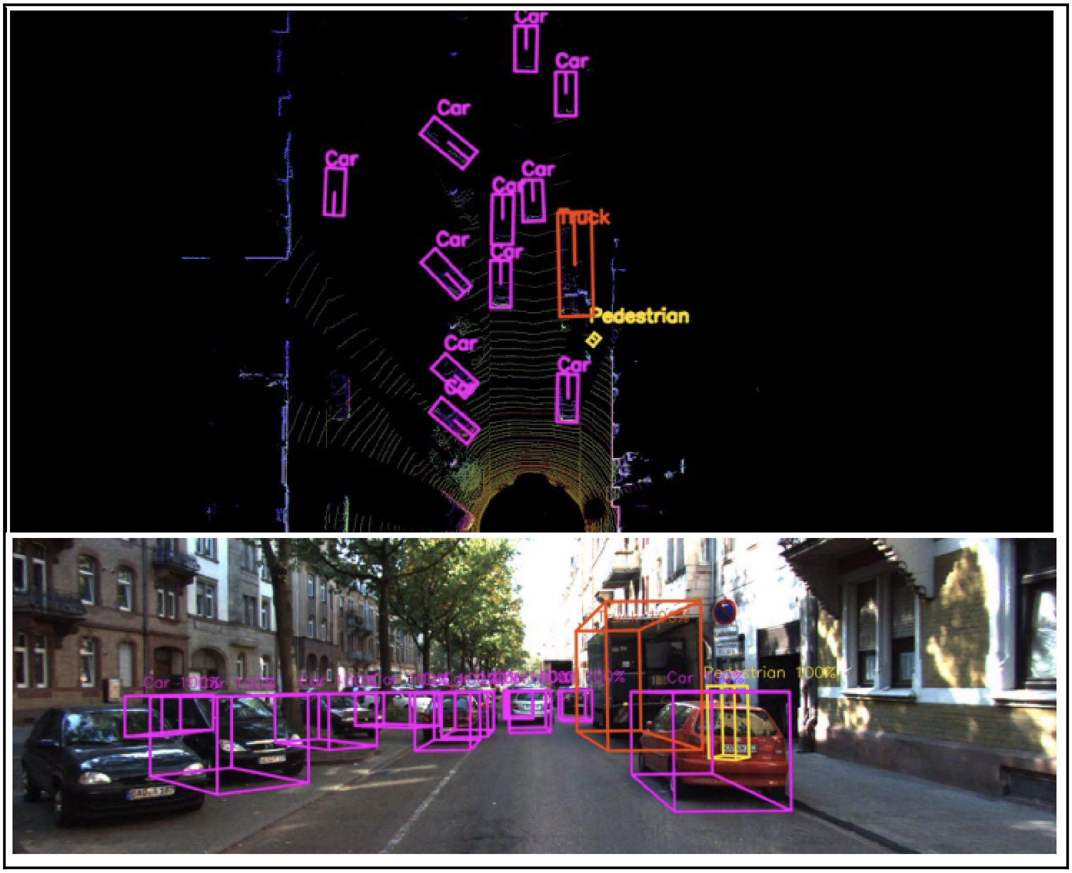
The default number of epochs is 300, but the results are fairly reasonable starting the fifth epoch itself. Each epoch takes 30 to 45 minutes on a GTX 1070 GPU. You can use `--resume_path` to resume training if training cannot be done in a single stretch. The code saves a new checkpoint every five epochs.

Testing

Just like in the *Data inspection* section, the trained model can be tested with the following code:

```
$ cd Complex-YOLOv4-Pytorch/src
$ python test.py --gpu_idx 0 --pretrained_path
../checkpoints/complexer_yolo/Model_complexer_yolo_epoch_5.pth --
cfgfile ./config/cfg/complex_yolov4.cfg --show_image
```

The main inputs to the code are the checkpoint path and the model configuration path. After giving them and running the code, the following output pops up (image source: `https://arxiv.org/pdf/1803.06199.pdf`):



Because of the simplicity of the model, we can use it in real-time scenarios with a normal GPU, getting about 15–20 predictions per second.

Summary

In this chapter, we learned about the various practical aspects of dealing with object localization and segmentation. Specifically, we learned about how the Detectron2 platform is leveraged to perform image segmentation and detection, and keypoint detection. In addition, we also learned about some of the intricacies involved in working with large datasets when we were working on fetching images from the Open Images dataset. Next, we worked on leveraging the VGG and U-Net architectures for crowd counting and image colorization, respectively. Finally, we understood the theory and implementation steps behind 3D object detection using point cloud images. As you can see from all these examples, the underlying basics are the same as those described in the previous chapters, with modifications only in the input/output of the networks to accommodate the task at hand.

In the next chapter, we will switch gears and learn about image encoding, which helps in identifying similar images as well as generating new images.

3
Section 3 - Image Manipulation

In this section, we will explore various techniques to manipulate images, including autoencoders and various types of GANs. We will leverage these techniques to improve image quality, to manipulate the style, and also to generate new images from existing ones.

This section comprises the following chapters:

11
Autoencoders and Image Manipulation

In the previous chapters, we have learned about classifying images, detecting objects in an image, and segmenting the pixels corresponding to objects in images. In this chapter, we will learn about representing an image in a lower dimension using autoencoders and leveraging the lower-dimensional representation of an image to generate new images by using variational autoencoders. Learning to represent images in a lower number of dimensions helps us manipulate (modify) the images to a considerable degree. We will learn about leveraging lower-dimensional representations to generate new images as well as novel images that are based on the content and style of two different images. Next, we will also learn about modifying images in such a way that the image is visually unaltered, however, the class corresponding to the image is changed from one to another. Finally, we will learn about generating deep fakes: given a source image of person A, we generate a target image of person B with a similar facial expression as that of person A.

Overall, we will go through the following topics in this chapter:

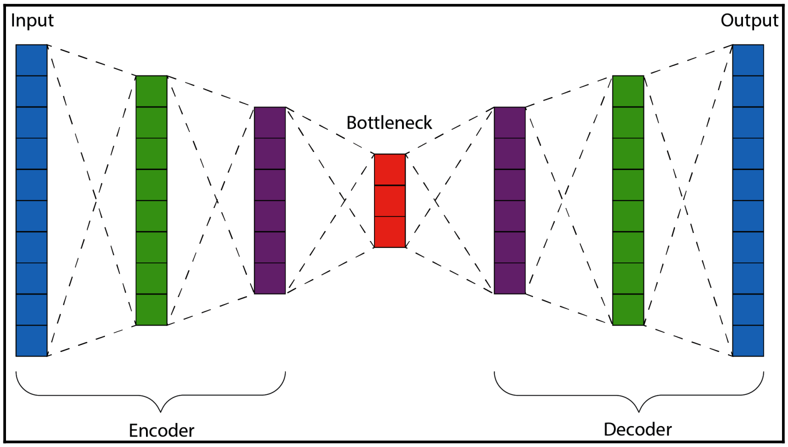
- Understanding and implementing autoencoders
- Understanding convolutional autoencoders
- Understanding variational autoencoders
- Performing an adversarial attack on images
- Performing neural style transfer
- Generating deep fakes

Understanding autoencoders

So far, in the previous chapters, we have learned about classifying images by training a model based on the input image and its corresponding label. Now let's imagine a scenario where we need to cluster images based on their similarity and with the constraint of not having their corresponding labels. Autoencoders come in handy to identify and group similar images.

An autoencoder takes an image as input, stores it in a lower dimension, and tries to reproduce the same image as output, hence the term **auto** (which stands for being able to reproduce the input). However, if we just reproduce the input in the output, we would not need a network, but a simple multiplication of the input by 1 would do. The differentiating aspect of an autoencoder is that it encodes the information present in an image in a lower dimension and then reproduces the image, hence the term **encoder** (which stands for representing the information of an image in a lower dimension). This way, images that are similar will have similar encoding. Further, the **decoder** works towards reconstructing the original image from the encoded vector.

In order to further understand autoencoders, let's take a look at the following diagram:



Let's say the input image is a flattened version of the MNIST handwritten digits and the output image is the same as what is provided as input. The middlemost layer is the layer of encoding called the **bottleneck** layer. The operations happening between the input and the bottleneck layer represent the **encoder** and the operations between the bottleneck layer and output represent the **decoder**.

Through the bottleneck layer, we can represent an image in a much lower dimension. Furthermore, with the bottleneck layer, we can reconstruct the original image. We leverage the bottleneck layer to solve the problems of identifying similar images as well as generating new images, which we will learn how to do in subsequent sections.

The bottleneck layer helps in the following ways:

- Images that have similar bottleneck layer values (encoded representations) are likely to be similar to each other.
- By changing the node values of the bottleneck layer, we can change the output image.

With the preceding understanding, let's do the following:

- Implement autoencoders from scratch
- Visualize the similarity of images based on bottleneck layer values

In the next section, we will learn about how autoencoders are built and also will learn about the impact of different units in the bottleneck layer on the decoder's output.

Implementing vanilla autoencoders

To understand how to build an autoencoder, let's implement one on the MNIST dataset, which contains images of handwritten digits:

The following code is available as `simple_auto_encoder_with_different_latent_size.ipynb` in the `chapter11` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components in text.

1. Import the relevant packages and define the device:

```
!pip install -q torch_snippets
from torch_snippets import *
from torchvision.datasets import MNIST
from torchvision import transforms
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

2. Specify the transformation that we want our images to pass through:

```
img_transform = transforms.Compose([
                    transforms.ToTensor(),
                    transforms.Normalize([0.5], [0.5]),
                    transforms.Lambda(lambda x: x.to(device))
                ])
```

In the preceding code, we see that we are converting an image into a tensor, normalizing it, and then passing it to the device.

3. Create the train and validation datasets:

```
trn_ds = MNIST('/content/', transform=img_transform, \
               train=True, download=True)
val_ds = MNIST('/content/', transform=img_transform, \
               train=False, download=True)
```

4. Define the dataloaders:

```
batch_size = 256
trn_dl = DataLoader(trn_ds, batch_size=batch_size, \
                    shuffle=True)
val_dl = DataLoader(val_ds, batch_size=batch_size, \
                    shuffle=False)
```

5. Define the network architecture. We define the `AutoEncoder` class constituting the encoder and decoder in the `__init__` method, along with the dimension of the bottleneck layer, `latent_dim`, and the `forward` method, and visualize the model summary:

- Define the `AutoEncoder` class and the `__init__` method containing the encoder, decoder, and the dimension of the bottleneck layer:

```
class AutoEncoder(nn.Module):
    def __init__(self, latent_dim):
        super().__init__()
        self.latend_dim = latent_dim
        self.encoder = nn.Sequential(
                            nn.Linear(28 * 28, 128),
                            nn.ReLU(True),
                            nn.Linear(128, 64),
                            nn.ReLU(True),
                            nn.Linear(64, latent_dim))
        self.decoder = nn.Sequential(
                            nn.Linear(latent_dim, 64),
                            nn.ReLU(True),
```

```
                                    nn.Linear(64, 128),
                                    nn.ReLU(True),
                                    nn.Linear(128, 28 * 28),
                                    nn.Tanh())
```

- Define the `forward` method:

```
    def forward(self, x):
        x = x.view(len(x), -1)
        x = self.encoder(x)
        x = self.decoder(x)
        x = x.view(len(x), 1, 28, 28)
        return x
```

- Visualize the preceding model:

```
!pip install torch_summary
from torchsummary import summary
model = AutoEncoder(3).to(device)
summary(model, torch.zeros(2,1,28,28))
```

This results in the following output:

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 3]                   --
|    └─Linear: 2-1                       [-1, 128]                 100,480
|    └─ReLU: 2-2                         [-1, 128]                 --
|    └─Linear: 2-3                       [-1, 64]                  8,256
|    └─ReLU: 2-4                         [-1, 64]                  --
|    └─Linear: 2-5                       [-1, 3]                   195
├─Sequential: 1-2                        [-1, 784]                 --
|    └─Linear: 2-6                       [-1, 64]                  256
|    └─ReLU: 2-7                         [-1, 64]                  --
|    └─Linear: 2-8                       [-1, 128]                 8,320
|    └─ReLU: 2-9                         [-1, 128]                 --
|    └─Linear: 2-10                      [-1, 784]                 101,136
|    └─Tanh: 2-11                        [-1, 784]                 --
==========================================================================================
Total params: 218,643
Trainable params: 218,643
Non-trainable params: 0
Total mult-adds (M): 0.43
```

From the preceding output, we can see that the `Linear: 2-5 layer` is the bottleneck layer, where each image is represented as a 3-dimensional vector. Furthermore, the decoder layer reconstructs the original image using the three values in the bottleneck layer.

6. Define a function to train on a batch of data (`train_batch`), just like we did in the previous chapters:

```
def train_batch(input, model, criterion, optimizer):
    model.train()
    optimizer.zero_grad()
    output = model(input)
    loss = criterion(output, input)
    loss.backward()
    optimizer.step()
    return loss
```

7. Define the function to validate on the batch of data (`validate_batch`):

```
@torch.no_grad()
def validate_batch(input, model, criterion):
    model.eval()
    output = model(input)
    loss = criterion(output, input)
    return loss
```

8. Define the model, loss criterion, and optimizer:

```
model = AutoEncoder(3).to(device)
criterion = nn.MSELoss()
optimizer = torch.optim.AdamW(model.parameters(), \
                              lr=0.001, weight_decay=1e-5)
```

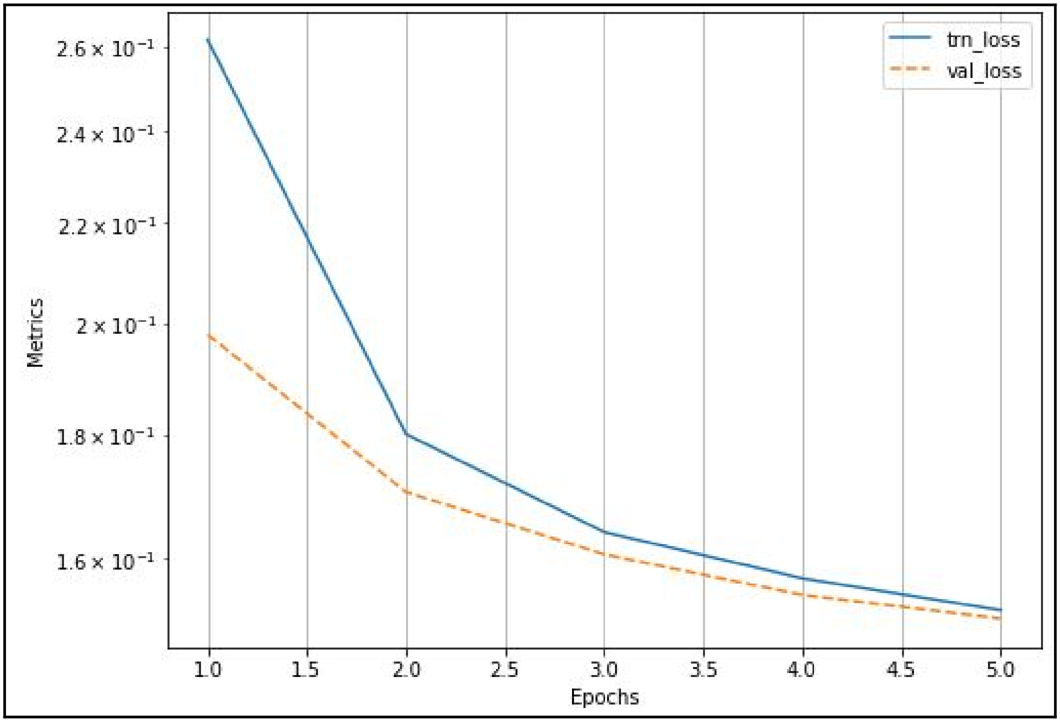
9. Train the model over increasing epochs:

```
num_epochs = 5
log = Report(num_epochs)

for epoch in range(num_epochs):
    N = len(trn_dl)
    for ix, (data, _) in enumerate(trn_dl):
        loss = train_batch(data, model, criterion, optimizer)
        log.record(pos=(epoch + (ix+1)/N), \
                   trn_loss=loss, end='\r')
    N = len(val_dl)
    for ix, (data, _) in enumerate(val_dl):
        loss = validate_batch(data, model, criterion)
        log.record(pos=(epoch + (ix+1)/N), \
                   val_loss=loss, end='\r')
    log.report_avgs(epoch+1)
```

10. Visualize the training and validation loss over increasing epochs:

```
log.plot_epochs(log=True)
```

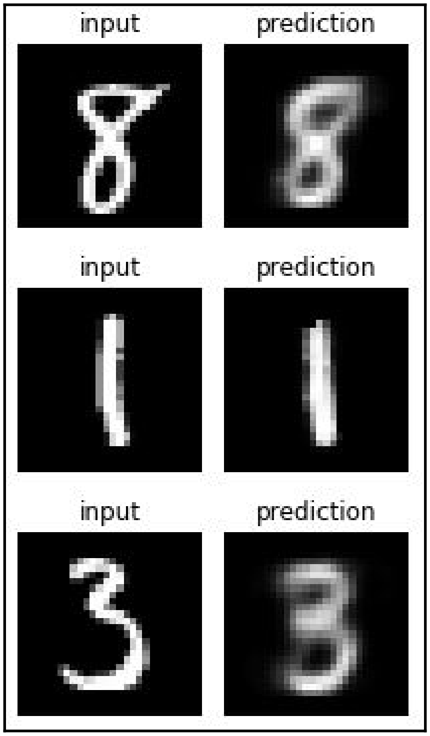
The preceding snippet returns the following output:



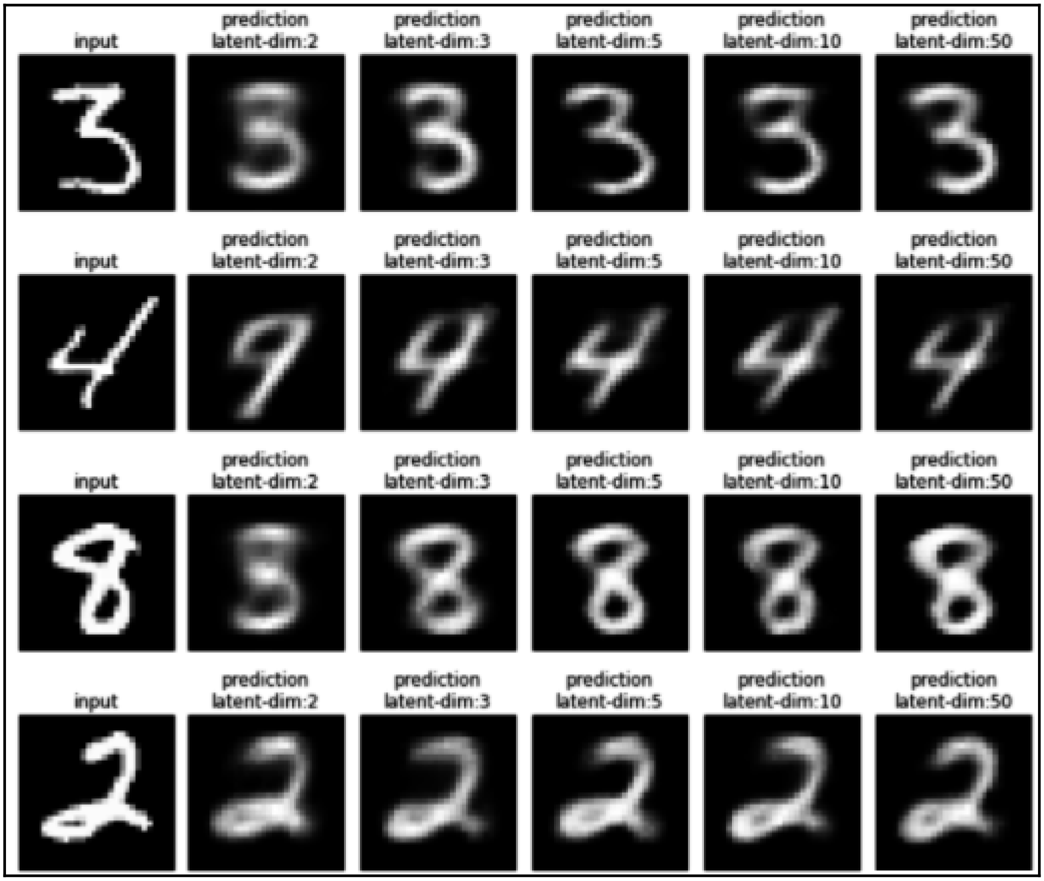
11. Validate the model on the `val_ds` dataset, which was not provided during training:

```
for _ in range(3):
    ix = np.random.randint(len(val_ds))
    im, _ = val_ds[ix]
    _im = model(im[None])[0]
    fig, ax = plt.subplots(1, 2, figsize=(3,3))
    show(im[0], ax=ax[0], title='input')
    show(_im[0], ax=ax[1], title='prediction')
    plt.tight_layout()
    plt.show()
```

The output of the preceding code is as follows:



We can see that the network can reproduce input with a very high level of accuracy even though the bottleneck layer is only three dimensions in size. However, the images are not as clear as we expect them to be. This is primarily because of the small number of nodes in the bottleneck layer. In the following image, we will visualize the reconstructed images after training networks with different bottleneck layer sizes - 2, 3, 5, 10, and 50:



It is clear that as the number of vectors in the bottleneck layer increased, the clarity of the reconstructed image improved.

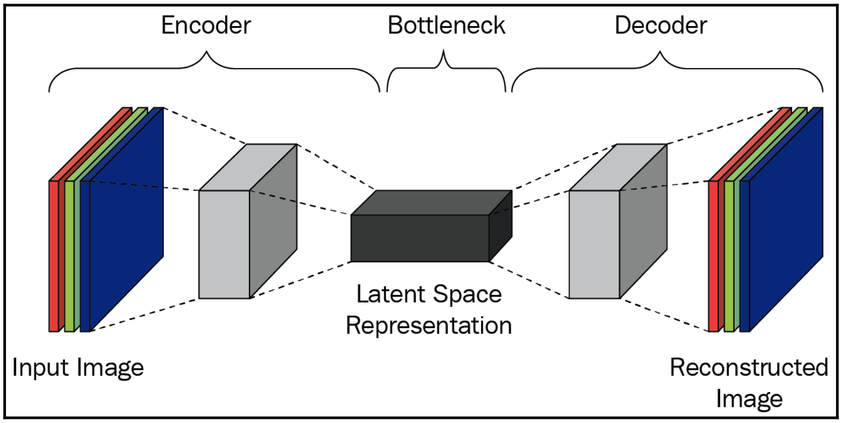
In the next section, we will learn about generating more clear images using a **convolutional neural network** (**CNN**) and we will learn about grouping similar images.

Understanding convolutional autoencoders

In the previous section, we learned about autoencoders and implemented them in PyTorch. While we have implemented them, one convenience that we had through the dataset was that each image has only 1 channel (each image was represented as a black and white image) and the images are relatively small (28 x 28). Hence the network flattened the input and was able to train on 784 (28*28) input values to predict 784 output values. However, in reality, we will encounter images that have 3 channels and are much bigger than a 28 x 28 image.

In this section, we will learn about implementing a convolutional autoencoder that is able to work on multi-dimensional input images. However, for the purpose of comparison with vanilla autoencoders, we will work on the same MNIST dataset that we worked on in the previous section, but modify the network in such a way that we now build a convolutional autoencoder and not a vanilla autoencoder.

A convolutional autoencoder is represented as follows:



From the preceding image, we can see that the input image is represented as a block in the bottleneck layer that is used to reconstruct the image. The image goes through multiple convolutions to fetch the bottleneck representation (which is the **Bottleneck layer** that is obtained by passing through **Encoder**) and the bottleneck representation is up-scaled to fetch the original image (the original image is reconstructed by passing through the **decoder**).

Now that we know how a convolutional autoencoder is represented, let's implement it in the following code:

> Given that the majority of the code is similar to the code in the previous section, we have only provided the additional code for brevity. The following code is available as `conv_auto_encoder.ipynb` in `Chapter11` folder of this book's GitHub repository. We encourage you to go through the notebook in GitHub if you want to see the complete code.

1. Steps 1 to 4, which are exactly the same as in the vanilla autoencoder section, are as follows:

```
!pip install -q torch_snippets
from torch_snippets import *
from torchvision.datasets import MNIST
from torchvision import transforms
device = 'cuda' if torch.cuda.is_available() else 'cpu'
img_transform = transforms.Compose([
                    transforms.ToTensor(),
                    transforms.Normalize([0.5], [0.5]),
                    transforms.Lambda(lambda x: x.to(device))
                                    ])

trn_ds = MNIST('/content/', transform=img_transform, \
               train=True, download=True)
val_ds = MNIST('/content/', transform=img_transform, \
               train=False, download=True)

batch_size = 128
trn_dl = DataLoader(trn_ds, batch_size=batch_size, \
                    shuffle=True)
val_dl = DataLoader(val_ds, batch_size=batch_size, \
                    shuffle=False)
```

2. Define the class of neural network, `ConvAutoEncoder`:

- Define the class and the `__init__` method:

```
class ConvAutoEncoder(nn.Module):
    def __init__(self):
        super().__init__()
```

- Define the `encoder` architecture:

```
        self.encoder = nn.Sequential(
                            nn.Conv2d(1, 32, 3, stride=3, \
```

```
                          padding=1),
                    nn.ReLU(True),
                    nn.MaxPool2d(2, stride=2),
                    nn.Conv2d(32, 64, 3, stride=2, \
                          padding=1),
                    nn.ReLU(True),
                    nn.MaxPool2d(2, stride=1)
                )
```

Note that in the preceding code, we started with the initial number of channels, which is `1`, and increased it to `32`, and then further increased it to `64` while reducing the size of the output values by performing `nn.MaxPool2d` and `nn.Conv2d` operations.

- Define the `decoder` architecture:

```
        self.decoder = nn.Sequential(
                    nn.ConvTranspose2d(64, 32, 3, \
                                 stride=2),
                    nn.ReLU(True),
                    nn.ConvTranspose2d(32, 16, 5, \
                                stride=3,padding=1),
                    nn.ReLU(True),
                    nn.ConvTranspose2d(16, 1, 2, \
                                stride=2,padding=1),
                    nn.Tanh()
                )
```

- Define the `forward` method:

```
    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

3. Get the summary of the model using the `summary` method:

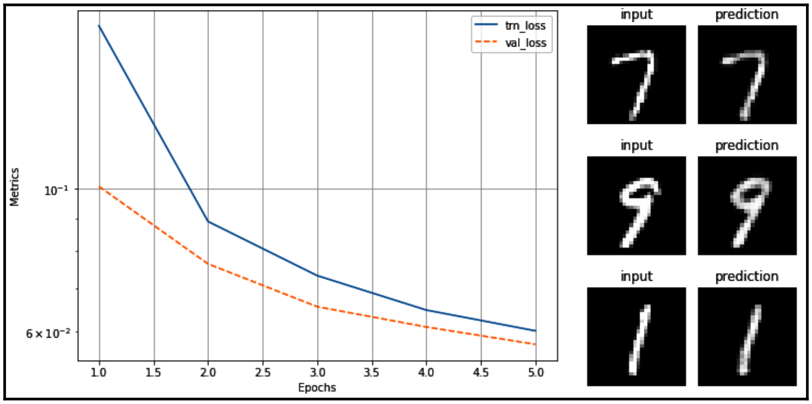
```
model = ConvAutoEncoder().to(device)
!pip install torch_summary
from torchsummary import summary
summary(model, torch.zeros(2,1,28,28));
```

The preceding code results in the following output:

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 64, 2, 2]            --
|    └─Conv2d: 2-1                       [-1, 32, 10, 10]          320
|    └─ReLU: 2-2                         [-1, 32, 10, 10]          --
|    └─MaxPool2d: 2-3                    [-1, 32, 5, 5]            --
|    └─Conv2d: 2-4                       [-1, 64, 3, 3]            18,496
|    └─ReLU: 2-5                         [-1, 64, 3, 3]            --
|    └─MaxPool2d: 2-6                    [-1, 64, 2, 2]            --
├─Sequential: 1-2                        [-1, 1, 28, 28]           --
|    └─ConvTranspose2d: 2-7              [-1, 32, 5, 5]            18,464
|    └─ReLU: 2-8                         [-1, 32, 5, 5]            --
|    └─ConvTranspose2d: 2-9              [-1, 16, 15, 15]          12,816
|    └─ReLU: 2-10                        [-1, 16, 15, 15]          --
|    └─ConvTranspose2d: 2-11             [-1, 1, 28, 28]           65
|    └─Tanh: 2-12                        [-1, 1, 28, 28]           --
==========================================================================================
Total params: 50,161
Trainable params: 50,161
Non-trainable params: 0
Total mult-adds (M): 3.64
```

From the preceding summary, we can see that the `MaxPool2d-6` layer with a shape of batch size x 64 x 2 x 2 acts as the bottleneck layer.

Once we train the model, just like we did in the previous section (in steps 6, 7, 8, and 9), the variation of training and validation loss over increasing epochs and the predictions on input images is as follows:



From the preceding image, we can see that a convolutional autoencoder is able to make much clearer predictions of the image than the vanilla autoencoder. As an exercise, we suggest you vary the number of channels in the encoder and decoder and then analyze the variation in results.

In the next section, we will address the question of grouping similar images based on bottleneck layer values when the labels of images are not present.

Grouping similar images using t-SNE

In the previous sections, we represented each image in a much lower dimension with the assumption that similar images will have similar embeddings, and images that are not similar will have dissimilar embeddings. However, we have not yet looked at the image similarity measure or examined embedding representations in detail.

In this section, we will plot embedding (bottleneck) vectors in a 2-dimensional space. We can reduce the 64-dimensional vector of convolutional autoencoder to a 2-dimensional space by using a technique called **t-SNE**. (More about t-SNE is available here: `http://www.jmlr.org/papers/v9/vandermaaten08a.html`.)

This way, our understanding that similar images will have similar embeddings can be proved, as similar images should be clustered together in the two-dimensional plane. In the following code, we will represent embeddings of all the test images in a two-dimensional plane:

> The following code is a continuation of the code built in the previous section, *Understanding convolutional autoencoders*, and is available as `conv_auto_encoder.ipynb` in the `Chapter 11` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Initialize lists so that we store the latent vectors (`latent_vectors`) and the corresponding `classes` of images (note that we store the class of each image only to verify if images of the same class, which are expected to have a very high similarity with each other, are indeed close to each other in terms of representation):

```
latent_vectors = []
classes = []
```

2. Loop through the images in the validation dataloader (`val_dl`) and store the output of the encoder layer `(model.encoder(im).view(len(im),-1)` and the class (`clss`) corresponding to each image (`im`):

```
for im,clss in val_dl:
    latent_vectors.append(model.encoder(im).view(len(im),-1))
    classes.extend(clss)
```

3. Concatenate the NumPy array of `latent_vectors`:

```
latent_vectors = torch.cat(latent_vectors).cpu()\
                          .detach().numpy()
```

4. Import t-SNE (`TSNE`) and specify that each vector is to be converted into a 2-dimensional vector (`TSNE(2)`) so that we can plot it:

```
from sklearn.manifold import TSNE
tsne = TSNE(2)
```

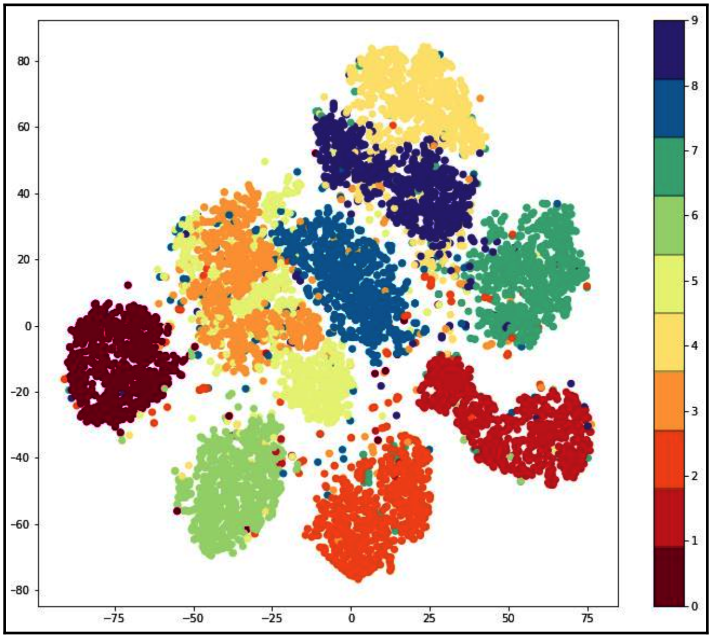
5. Fit t-SNE by running the `fit_transform` method on image embeddings (`latent_vectors`):

```
clustered = tsne.fit_transform(latent_vectors)
```

6. Plot the data points after fitting t-SNE:

```
fig = plt.figure(figsize=(12,10))
cmap = plt.get_cmap('Spectral', 10)
plt.scatter(*zip(*clustered), c=classes, cmap=cmap)
plt.colorbar(drawedges=True)
```

The preceding code provides the following output:



We can see that images of the same class are clustered together, which reinforces our understanding that the bottleneck layer has values in such a way that images that look similar will have similar values.

So far, we have learned about using autoencoders to group similar images together. In the next section, we will learn about using autoencoders to generate new images.

Understanding variational autoencoders

So far, we have seen a scenario where we can group similar images into clusters. Furthermore, we have learned that when we take embeddings of images that fall in a given cluster, we can re-construct (decode) them. However, what if an embedding (a latent vector) falls in between two clusters? There is no guarantee that we would generate realistic images. Variational autoencoders come in handy in such a scenario.

Before we dive into building a variational autoencoder, let's explore the limitations of generating images from embeddings that do not fall into a cluster (or in the middle of different clusters). First, we generate images by sampling vectors:

> The following code is a continuation of the code built in the previous section, *Understanding convolutional autoencoders*, and is available as `conv_auto_encoder.ipynb` in the `chapter11` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Calculate the latent vectors (embeddings) of the validation images in the previous section:

```
latent_vectors = []
classes = []
for im,clss in val_dl:
    latent_vectors.append(model.encoder(im))
    classes.extend(clss)
latent_vectors = torch.cat(latent_vectors).cpu()\
                      .detach().numpy().reshape(10000, -1)
```

2. Generate random vectors with a column-level mean (`mu`) and a standard deviation (`sigma`) and add slight noise to the standard deviation (`torch.randn(1,100)`) before creating a vector from the mean and standard deviation. Finally, save them in a list (`rand_vectors`):

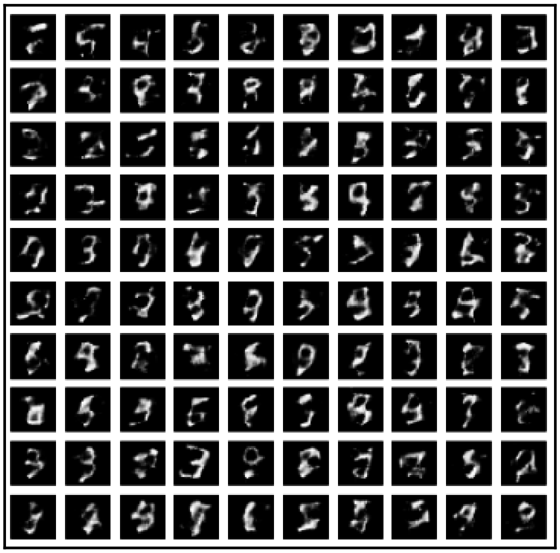
```
rand_vectors = []
for col in latent_vectors.transpose(1,0):
```

```
        mu, sigma = col.mean(), col.std()
        rand_vectors.append(sigma*torch.randn(1,100) + mu)
```

3. Plot the images reconstructed from the vectors obtained in step 2 and the model trained in the previous section:

```
rand_vectors=torch.cat(rand_vectors).transpose(1,0).to(device)
fig,ax = plt.subplots(10,10,figsize=(7,7)); ax = iter(ax.flat)
for p in rand_vectors:
    img = model.decoder(p.reshape(1,64,2,2)).view(28,28)
    show(img, ax=next(ax))
```

The preceding code results in the following output:



We can see from the preceding output that when we plot images that were generated from the mean and the noise-added standard deviation of columns of known vectors, we got images that are less clear than before. This is a realistic scenario, as we would not know beforehand about the range of embedding vectors that would generate realistic pictures.

Variational Autoencoders (**VAE**) help us resolve this problem by generating vectors that have a mean of 0 and a standard deviation of 1, thereby ensuring that we generate images that have a mean of 0 and a standard deviation of 1.

In essence, in VAE, we are specifying that the bottleneck layer should follow a certain distribution. In the next sections, we will learn about the strategy we adopt with VAE, and we will also learn about KL divergence loss, which helps us fetch bottleneck features that follow a certain distribution.

Working of VAE

In a VAE, we are building the network in such a way that a random vector that is generated from a pre-defined distribution can generate a realistic image. This was not possible with a simple autoencoder, as we did not specify the distribution of data that generates an image in the network. We enable that with a VAE by adopting the following strategy:

1. The output of the encoder is two vectors for each image:
    - One vector represents the mean.
    - The other represents the standard deviation.
2. From these two vectors, we fetch a modified vector that is the sum of the mean and standard deviation (which is multiplied by a random small number). The modified vector will be of the same number of dimensions as each vector.
3. The modified vector obtained in the previous step is passed as input to the decoder to fetch the image.
4. The loss value that we optimize for is a combination of the mean squared error and the KL divergence loss:
    - KL divergence loss measures the deviation of the distribution of the mean vector and the standard deviation vector from 0 and 1, respectively.
    - Mean squared loss is the optimization we use to re-construct (decode) an image.

By specifying that the mean vector should have a distribution centered around 0 and the standard deviation vector should be centered around 1, we are training the network in such a way that when we generate random noise with a mean of 0 and standard deviation of 1, the decoder will be able to generate a realistic image.

Further, note that, had we only minimized KL divergence, the encoder would have predicted a value of 0 for the mean vector and a standard deviation of 1 for every input. Thus, it is important to minimize KL divergence loss and mean squared loss together.

In the next section, let's learn about KL divergence so that we can incorporate it in the model's loss value calculation.

KL divergence

KL divergence helps explain the difference between two distributions of data. In our specific case, we want our bottleneck feature values to be following a normal distribution with a mean of 0 and a standard deviation of 1.

Thus, we use KL divergence loss to understand how different our bottleneck feature values are with respect to the expected distribution of values having a mean of 0 and a standard deviation of 1.

Let's take a look at how KL divergence loss helps by going through how it is calculated:

$$\sum_{i=1}^{n} \sigma_i^2 \ + \ \mu_i^2 \ - \ log(\sigma_i) \ - \ 1$$

In the preceding equation, σ and μ stand for the mean and standard deviation values of each input image.

Let's understand the intuition behind the preceding equation:

- Ensure that the mean vector is distributed around 0:
    - Minimizing mean squared error (μ_i^2) in the preceding equation ensures that $\mu$is as close to 0 as possible.
- Ensure that the standard deviation vector is distributed around 1:
    - The terms in the rest of the equation (except μ_i^2) ensure that sigma (the standard deviation vector) is distributed around 1.

> *The preceding loss function is minimized when the mean (μ) is 0 and the standard deviation is 1. Further, by specifying that we are considering the logarithm of standard deviation, we are ensuring that sigma values cannot be negative.*

Now that we understand the high-level strategy of building a VAE and the loss function to minimize in order to obtain a pre-defined distribution of encoder output, let's implement a VAE in the next section.

Building a VAE

In this section, we will code up a VAE to generate new images of handwritten digits.

The following code is available as `VAE.ipynb` in the `Chapter11` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

Since we have the same data, all the steps in the *Implementing vanilla autoencoders* section remain the same except steps 5 and 6, where we define the network architecture and train model respectively, which we define in the following code:

1. Step 1 to step 4, which are exactly the same as in the vanilla autoencoder section, are as follows:

```
!pip install -q torch_snippets
from torch_snippets import *
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torchvision.utils import make_grid
device = 'cuda' if torch.cuda.is_available() else 'cpu'
train_dataset = datasets.MNIST(root='MNIST/', train=True, \
                        transform=transforms.ToTensor(), \
                               download=True)
test_dataset = datasets.MNIST(root='MNIST/', train=False, \
                        transform=transforms.ToTensor(), \
                              download=True)

train_loader = torch.utils.data.DataLoader(dataset = \
                train_dataset, batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset= \
                test_dataset, batch_size=64, shuffle=False)
```

2. Define the neural network class, `VAE`:

- Define the layers in the `__init__` method that will be used in the other methods:

```
class VAE(nn.Module):
    def __init__(self, x_dim, h_dim1, h_dim2, z_dim):
        super(VAE, self).__init__()
        self.d1 = nn.Linear(x_dim, h_dim1)
```

```
        self.d2 = nn.Linear(h_dim1, h_dim2)
        self.d31 = nn.Linear(h_dim2, z_dim)
        self.d32 = nn.Linear(h_dim2, z_dim)
        self.d4 = nn.Linear(z_dim, h_dim2)
        self.d5 = nn.Linear(h_dim2, h_dim1)
        self.d6 = nn.Linear(h_dim1, x_dim)
```

Note that the `d1` and `d2` layers will correspond to the encoder section, and `d5` and `d6` will correspond to the decoder section. The `d31` and `d32` layers are the layers that correspond to mean and standard deviation vectors respectively. However, for convenience, one assumption we will make is that we will use the `d32` layer as a representation of the log of the variance vectors.

- Define the `encoder` method:

```
    def encoder(self, x):
        h = F.relu(self.d1(x))
        h = F.relu(self.d2(h))
        return self.d31(h), self.d32(h)
```

Note that the encoder returns two vectors: one vector for the mean `(self.d31(h))` and the other for the log of variance values `(self.d32(h))`.

- Define the method to sample (`sampling`) from the encoder's outputs:

```
    def sampling(self, mean, log_var):
        std = torch.exp(0.5*log_var)
        eps = torch.randn_like(std)
        return eps.mul(std).add_(mean)
```

Note that exponential of *0.5*log_var* (`torch.exp(0.5*log_var)`) represents the standard deviation (`std`). Also, we are returning the addition of the mean and the standard deviation multiplied by noise generated by a random normal distribution. By multiplying by `eps`, we ensure that even with a slight change in the encoder vector, we can generate an image.

- Define the `decoder` method:

```
    def decoder(self, z):
        h = F.relu(self.d4(z))
        h = F.relu(self.d5(h))
        return F.sigmoid(self.d6(h))
```

- Define the `forward` method:

```
def forward(self, x):
    mean, log_var = self.encoder(x.view(-1, 784))
    z = self.sampling(mean, log_var)
    return self.decoder(z), mean, log_var
```

In the preceding method, we are ensuring that the encoder returns the mean and log of the variance values. Next, we are sampling with the addition of mean with epsilon multiplied by the log of the variance and returning the values after passing through the decoder.

3. Define functions to train on a batch and validate on a batch:

```
def train_batch(data, model, optimizer, loss_function):
    model.train()
    data = data.to(device)
    optimizer.zero_grad()
    recon_batch, mean, log_var = model(data)
    loss, mse, kld = loss_function(recon_batch, data, \
                                   mean, log_var)
    loss.backward()
    optimizer.step()
    return loss, mse, kld, log_var.mean(), mean.mean()

@torch.no_grad()
def validate_batch(data, model, loss_function):
    model.eval()
    data = data.to(device)
    recon, mean, log_var = model(data)
    loss, mse, kld = loss_function(recon, data, mean, \
                                   log_var)
    return loss, mse, kld, log_var.mean(), mean.mean()
```

4. Define the loss function:

```
def loss_function(recon_x, x, mean, log_var):
    RECON = F.mse_loss(recon_x, x.view(-1, 784), \
                       reduction='sum')
    KLD = -0.5 * torch.sum(1 + log_var - mean.pow(2) - \
                           log_var.exp())
    return RECON + KLD, RECON, KLD
```

In the preceding code, we are fetching the MSE loss (`RECON`) between the original image (`x`) and the reconstructed image (`recon_x`). Next, we are calculating the KL divergence loss (`KLD`) based on the formula we defined in the previous section. Note that the exponential of the log of the variance is the variance value.

5. Define the model object (`vae`) and the `optimizer` function:

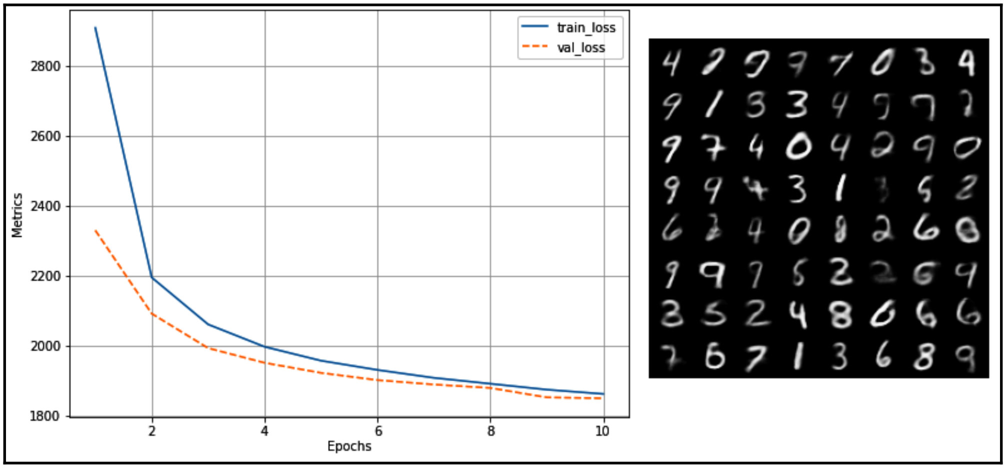
```
vae = VAE(x_dim=784, h_dim1=512, h_dim2=256, \
          z_dim=50).to(device)
optimizer = optim.AdamW(vae.parameters(), lr=1e-3)
```

6. Train the model over increasing epochs:

```
n_epochs = 10
log = Report(n_epochs)

for epoch in range(n_epochs):
    N = len(train_loader)
    for batch_idx, (data, _) in enumerate(train_loader):
        loss, recon, kld, log_var, mean = train_batch(data, \
                                                    vae, optimizer, \
                                                       loss_function)
        pos = epoch + (1+batch_idx)/N
        log.record(pos, train_loss=loss, train_kld=kld, \
                   train_recon=recon,train_log_var=log_var, \
                   train_mean=mean, end='\r')
    N = len(test_loader)
    for batch_idx, (data, _) in enumerate(test_loader):
        loss, recon, kld,log_var,mean = validate_batch(data, \
                                                   vae, loss_function)
        pos = epoch + (1+batch_idx)/N
        log.record(pos, val_loss=loss, val_kld=kld, \
                   val_recon=recon, val_log_var=log_var, \
                   val_mean=mean, end='\r')
    log.report_avgs(epoch+1)
    with torch.no_grad():
        z = torch.randn(64, 50).to(device)
        sample = vae.decoder(z).to(device)
        images = make_grid(sample.view(64, 1, 28, 28))\
                                         .permute(1,2,0)
        show(images)
log.plot_epochs(['train_loss','val_loss'])
```

While the majority of the preceding code is familiar, let's understand the grid image generation process. We are first generating a random vector (`z`) and passing it through the decoder (`vae.decoder`) to fetch a sample of images. The `make_grid` function plots images (and denormalizes them automatically, if required, before plotting).

The output of loss value variations and a sample of images generated is as follows:



We can see that we are able to generate realistic new images that were not present in the original image.

So far, we have learned about generating new images using VAEs. However, what if we want to modify images in such a way that a model cannot identify the right class? We will learn about the technique leveraged to address this in the next section.

Performing an adversarial attack on images

In the previous section, we learned about generating an image from random noise using a VAE. However, it was an unsupervised exercise. What if we want to modify an image in such a way that the change in image is so minimal that it is indistinguishable from the original image for a human, but still the neural network model perceives the object as belonging to a different class? Adversarial attacks on images come in handy in such a scenario.

Adversarial attacks refer to the changes that we make to input image values (pixels) so that we meet a certain objective.

In this section, we will learn about modifying an image slightly in such a way that the pre-trained models now predict them as a different class (specified by the user) and not the original class. The strategy we will adopt is as follows:

1. Provide an image of an elephant.
2. Specify the target class corresponding to the image.
3. Import a pre-trained model where the parameters of the model are set so that they are not updated (`gradients = False`).
4. Specify that we calculate gradients on input image pixel values and not on the weight values of the network. This is because while training to fool a network, we do not have control over the model, but have control only over the image we send to the model.
5. Calculate the loss corresponding to the model predictions and the target class.
6. Perform backpropagation on the model. This step helps us understand the gradient associated with each input pixel value.
7. Update the input image pixel values based on the direction of the gradient corresponding to each input pixel value.
8. Repeat *steps 5*, *6*, and *7* over multiple epochs.

Let's do this with code:

The following code is available as `adversarial_attack.ipynb` in the `Chapter11` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components in text.

1. Import the relevant packages, the image that we work on for this use case, and the pre-trained ResNet50 model. Also, specify that we want to freeze parameters:

```
!pip install torch_snippets
from torch_snippets import inspect, show, np, torch, nn
from torchvision.models import resnet50
model = resnet50(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
```

```
model = model.eval()
import requests
from PIL import Image
url =
'https://lionsvalley.co.za/wp-content/uploads/2015/11/african-
elephant-square.jpg'
original_image = Image.open(requests.get(url, stream=True)\
                            .raw).convert('RGB')
original_image = np.array(original_image)
original_image = torch.Tensor(original_image)
```

2. Import Imagenet classes and assign IDs to each class:

```
image_net_classes =
'https://gist.githubusercontent.com/yrevar/942d3a0ac09ec9e5eb3
a/raw/238f720ff059c1f82f368259d1ca4ffa5dd8f9f5/imagenet1000_cl
sidx_to_labels.txt'
image_net_classes = requests.get(image_net_classes).text
image_net_ids = eval(image_net_classes)
image_net_classes = {i:j for j,i in image_net_ids.items()}
```

3. Specify a function to normalize (`image2tensor`) and denormalize (`tensor2image`) the image:

```
from torchvision import transforms as T
from torch.nn import functional as F
normalize = T.Normalize([0.485, 0.456, 0.406],
                        [0.229, 0.224, 0.225])
denormalize=T.Normalize( \
                [-0.485/0.229,-0.456/0.224,-0.406/0.225],
                [1/0.229, 1/0.224, 1/0.225])
def image2tensor(input):
    x = normalize(input.clone().permute(2,0,1)/255.)[None]
    return x
def tensor2image(input):
    x = (denormalize(input[0].clone()).permute(1,2,0)*255.)\
                                        .type(torch.uint8)
    return x
```

4. Define a function to predict on a given image (`predict_on_image`):

```
def predict_on_image(input):
    model.eval()
    show(input)
    input = image2tensor(input)
    pred = model(input)
    pred = F.softmax(pred, dim=-1)[0]
    prob, clss = torch.max(pred, 0)
```

```
clss = image_net_ids[clss.item()]
print(f'PREDICTION: `{clss}` @ {prob.item()}')
```

In the preceding code, we are converting an input image into a tensor (which is a function to normalize using the `image2tensor` method defined earlier) and passing through a `model` to fetch the class (`clss`) of the object in the image and probability (`prob`) of prediction.

5. Define the `attack` function:

- The `attack` function takes `image`, `model`, and `target` as input:

```
from tqdm import trange
losses = []
def attack(image, model, target, epsilon=1e-6):
```

- Convert the image into a tensor and specify that the input requires gradients to be calculated:

```
    input = image2tensor(image)
    input.requires_grad = True
```

- Calculate the prediction by passing the normalized input (`input`) through the model, and then calculate the loss value corresponding to the specified target class:

```
    pred = model(input)
    loss = nn.CrossEntropyLoss()(pred, target)
```

- Perform backpropagation to reduce the loss:

```
    loss.backward()
    losses.append(loss.mean().item())
```

- Update the image very slightly based on the gradient direction:

```
    output = input - epsilon * input.grad.sign()
```

In the preceding code, we are updating input values by a very small amount (multiplying by `epsilon`). Also, we are not updating the image by the magnitude of the gradient, but the direction of gradient only (`input.grad.sign()`) after multiplying it by a very small value (`epsilon`).

- Return the output after converting the tensor to an image (`tensor2image`), which denormalizes the image:

```
        output = tensor2image(output)
        del input
        return output.detach()
```

6. Modify the image to belong to a different class:

- Specify the targets (`desired_targets`) that we want to convert the image to:

```
modified_images = []
desired_targets = ['lemon', 'comic book', 'sax, saxophone']
```

- Loop through the targets and specify the target class in each iteration:

```
for target in desired_targets:
    target = torch.tensor([image_net_classes[target]])
```

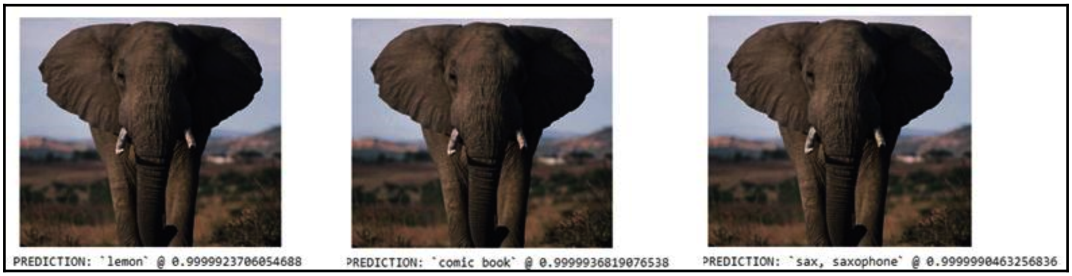
- Modify the image to attack over increasing epochs and collect them in a list:

```
    image_to_attack = original_image.clone()
    for _ in trange(10):
        image_to_attack = attack(image_to_attack,model,target)
    modified_images.append(image_to_attack)
```

- The following code results in modified images and the corresponding classes:

```
for image in [original_image, *modified_images]:
    predict_on_image(image)
    inspect(image)
```

The preceding code generates the following:



We can see that as we modify the image very slightly, the prediction class is completely different but with very high confidence.

Now that we understand how to modify images to so that they are classed as we wish, in the next section, we will learn about modifying an image (a content image) in the style of our choice. We must provide a content image and a style image.

Performing neural style transfer

In neural style transfer, we have a content image and a style image, and we combine these two images in such a way that the combined image preserves the content of the content image while maintaining the style of the style image.

An example style image and content image are as follows:

In the preceding picture, we want to retain the content in the picture on right (the content image), but overlay it with the color and texture in the picture on the left (the style image).

The process of performing neural style transfer is as follows. We try to modify the original image in a way that the loss value is split into **content loss** and **style loss**. Content loss refers to how **different** the generated image is from the content image. Style loss refers to how **correlated** the style image is to the generated image.

While we mentioned that the loss is calculated based on the difference in images, in practice, we modify it slightly by ensuring that the loss is calculated using the feature layer activations of images and not the original images. For example, the content loss at layer 2 will be the squared difference between the *activations of the content image and the generated image* when passed through the second layer.

Loss is calculated on feature layers and not the original image, as the feature layers capture certain attributes of the original image (for example, the outline of the foreground corresponding to the original image in the higher layers and the details of fine-grained objects in the lower layers).

While calculating the content loss seems straightforward, let's try to understand how to calculate the similarity between the generated image and the style image. A technique called **gram matrix** comes in handy. Gram matrix calculates the similarity between a generated image and a style image, and is calculated as follows:

$$\boxed{L_{GM}(S,G,l) = \frac{1}{4N_l^2 M_l^2} \sum_{ij} (GM[l](S)_{ij} - GM[l](G)_{ij})^2}$$

GM[l] is the gram matrix value at layer *l* for the style image, *S*, and the generated image, *G*.
A gram matrix results from multiplying a matrix by the transpose of itself. Let's understand the use of this operation.

Imagine that you are working on a layer that has a feature output of 32 x 32 x 256. The gram matrix is calculated as the correlation of each of the 32 x 32 values in a channel with respect to the values across all channels. Thus, the gram matrix calculation results in a matrix that is 256 x 256 in shape. We now compare the 256 x 256 values of the style image and the generated image to calculate the style loss.

Let's understand why GramMatrix is important for style transfer.

In a successful scenario, say we transferred Picasso's style to the Mona Lisa. Let's call the Picasso style *St* (for style), the original Mona Lisa *So* (for source), and the final image *Ta* (for target). Note that in an ideal scenario, the local features in image *Ta* are the same as the local features in *St*. Even though the content might not be the same, getting similar colors, shapes, and textures as the style image into the target image is what is important in style transfer.

By extension, if we were to send *So* and extract its features from an intermediate layer of VGG19, they will vary from the features obtained by sending *Ta*. However, within each feature set, the corresponding vectors will vary relatively with each other in a similar fashion. Say, for example, the ratio of the mean of the first channel to the mean of the second channel in both the feature sets will be similar. This is why we are trying to compute using Gram Loss.

Content loss is calculated by comparing the difference in feature activations of the content image with respect to the generated image. Style loss is calculated by first calculating the gram matrix in the pre-defined layers and then comparing the gram matrices of the generated image and the style image.

Now that we are able to calculate the style loss and the content loss, the final modified input image is the image that minimizes the overall loss, that is, a weighted average of the style and content loss.
The high-level strategy we adopt to implement neural style transfer is as follows:

1. Pass the input image through a pre-trained model.
2. Extract the layer values at pre-defined layers.
3. Pass the generated image through the model and extract its values at the same pre-defined layers.
4. Calculate the content loss at each layer corresponding to the content image and generated image.
5. Pass the style image through multiple layers of the model and calculate the gram matrix values of the style image.
6. Pass the generated image through the same layers that the style image is passed through and calculate its corresponding gram matrix values.
7. Extract the squared difference of the gram matrix values of the two images. This will be the style loss.
8. The overall loss will be the weighted average of the style loss and content loss.
9. The input image that minimizes the overall loss will be the final image of interest.

Let's now code up the preceding strategy:

> The following code is available as `neural_style_transfer.ipynb` in the `chapter11` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from and is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components in text.

1. Import the relevant packages:

```
!pip install torch_snippets
from torch_snippets import *
from torchvision import transforms as T
from torch.nn import functional as F
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

2. Define the functions to preprocess and postprocess the data:

```
from torchvision.models import vgg19
preprocess = T.Compose([
                T.ToTensor(),
                T.Normalize(mean=[0.485, 0.456, 0.406],
                            std=[0.229, 0.224, 0.225]),
                T.Lambda(lambda x: x.mul_(255))
            ])
postprocess = T.Compose([
                T.Lambda(lambda x: x.mul_(1./255)),
                T.Normalize(\
                mean=[-0.485/0.229,-0.456/0.224,-0.406/0.225],
                            std=[1/0.229, 1/0.224, 1/0.225]),
            ])
```

3. Define the `GramMatrix` module:

```
class GramMatrix(nn.Module):
    def forward(self, input):
        b,c,h,w = input.size()
        feat = input.view(b, c, h*w)
        G = feat@feat.transpose(1,2)
        G.div_(h*w)
        return G
```

In the preceding code, we are computing all the possible inner products of the features with themselves, which is basically asking how all the vectors relate to each other.

4. Define the gram matrix's corresponding MSE loss, `GramMSELoss`:

```
class GramMSELoss(nn.Module):
    def forward(self, input, target):
        out = F.mse_loss(GramMatrix()(input), target)
        return(out)
```

Once we have the gram vectors for both feature sets, it is important that they match as closely as possible, and hence the `mse_loss`.

5. Define the model class, `vgg19_modified`:

- Initialize the class:

```
class vgg19_modified(nn.Module):
    def __init__(self):
        super().__init__()
```

- Extract the features:

```
        features = list(vgg19(pretrained = True).features)
        self.features = nn.ModuleList(features).eval()
```

- Define the `forward` method, which takes the list of layers and returns the features corresponding to each layer:

```
    def forward(self, x, layers=[]):
        order = np.argsort(layers)
        _results, results = [], []
        for ix,model in enumerate(self.features):
            x = model(x)
            if ix in layers: _results.append(x)
        for o in order: results.append(_results[o])
        return results if layers is not [] else x
```

- Define the model object:

```
vgg = vgg19_modified().to(device)
```

6. Import the content and style images:

```
!wget https://www.dropbox.com/s/z1y0fy2r6z6m6py/60.jpg
!wget
https://www.dropbox.com/s/1svdliljyo0a98v/style_image.png
```

- Make sure that the images are resized to be of the same shape, 512 x 512 x 3:

```
imgs = [Image.open(path).resize((512,512)).convert('RGB') \
        for path in ['style_image.png', '60.jpg']]
style_image,content_image=[preprocess(img).to(device)[None] \
                              for img in imgs]
```

7. Specify that the content image is to modified with `requires_grad = True`:

```
opt_img = content_image.data.clone()
opt_img.requires_grad = True
```

8. Specify the layers that define content loss and style loss, that is, which intermediate VGG layers we are using, to compare gram matrices for style and raw feature vectors for content:

```
style_layers = [0, 5, 10, 19, 28]
content_layers = [21]
loss_layers = style_layers + content_layers
```

9. Define the loss function for content and style loss values:

```
loss_fns = [GramMSELoss()] * len(style_layers) + \
            [nn.MSELoss()] * len(content_layers)
loss_fns = [loss_fn.to(device) for loss_fn in loss_fns]
```

10. Define the weightage associated with content and style loss:

```
style_weights = [1000/n**2 for n in [64,128,256,512,512]]
content_weights = [1]
weights = style_weights + content_weights
```

11. We need to manipulate our image such that the style of the target image resembles `style_image` as much as possible. Hence we compute the `style_targets` values of `style_image` by computing `GramMatrix` of features obtained from a few chosen layers of VGG. Since the overall content should be preserved, we choose the `content_layer` variable at which we compute the raw features from VGG:

```
style_targets = [GramMatrix()(A).detach() for A in \
                 vgg(style_image, style_layers)]
content_targets = [A.detach() for A in \
                   vgg(content_image, content_layers)]
targets = style_targets + content_targets
```

12. Define the `optimizer` and the number of iterations (`max_iters`). Even though we could have used Adam or any other optimizer, LBFGS is an optimizer that has been observed to work best in deterministic scenarios. Additionally, since we are dealing with exactly one image, there is nothing random. Many experiments have revealed that LBFGS converges faster and to lower losses in neural transfer settings, so we will use this optimizer:

```
max_iters = 500
optimizer = optim.LBFGS([opt_img])
log = Report(max_iters)
```

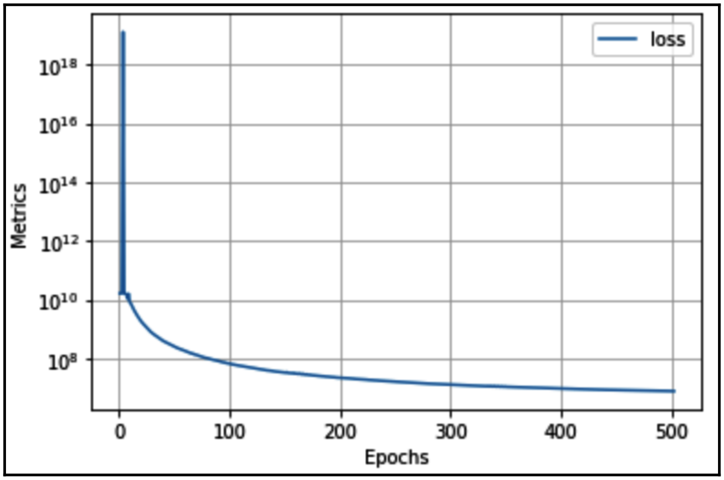
13. Perform the optimization. In deterministic scenarios where we are iterating on the same tensor again and again, we can wrap the optimizer step as a function with zero arguments and repeatedly call it, as shown here:

```
iters = 0
while iters < max_iters:
    def closure():
        global iters
        iters += 1
        optimizer.zero_grad()
        out = vgg(opt_img, loss_layers)
        layer_losses = [weights[a]*loss_fns[a](A,targets[a]) \
                        for a,A in enumerate(out)]
        loss = sum(layer_losses)
        loss.backward()
        log.record(pos=iters, loss=loss, end='\r')
        return loss
    optimizer.step(closure)
```

14. Plot the variation in the loss:

```
log.plot(log=True)
```

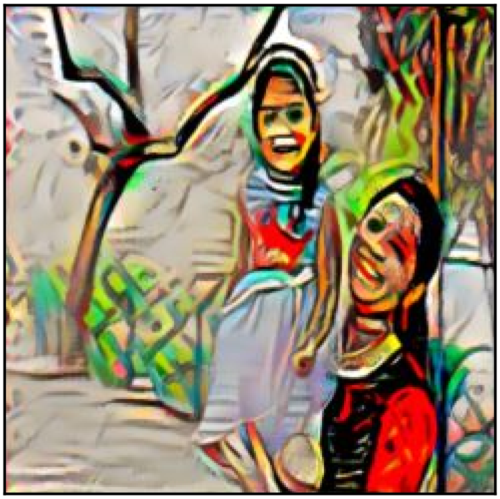
This results in the following output:



15. Plot the image with the combination of content and style images:

```
out_img = postprocess(opt_img[0]).permute(1,2,0)
show(out_img)
```

The output is as follows:



From the preceding picture, we can see that the image is a combination of content and style images.

With this, we have seen two ways of manipulating an image: an adversarial attack to modify the class of an image, and style transfer to combine the style of one image with the content of another image. In the next section, we will learn about generating deep fakes, which transfer an expression from one face to another.

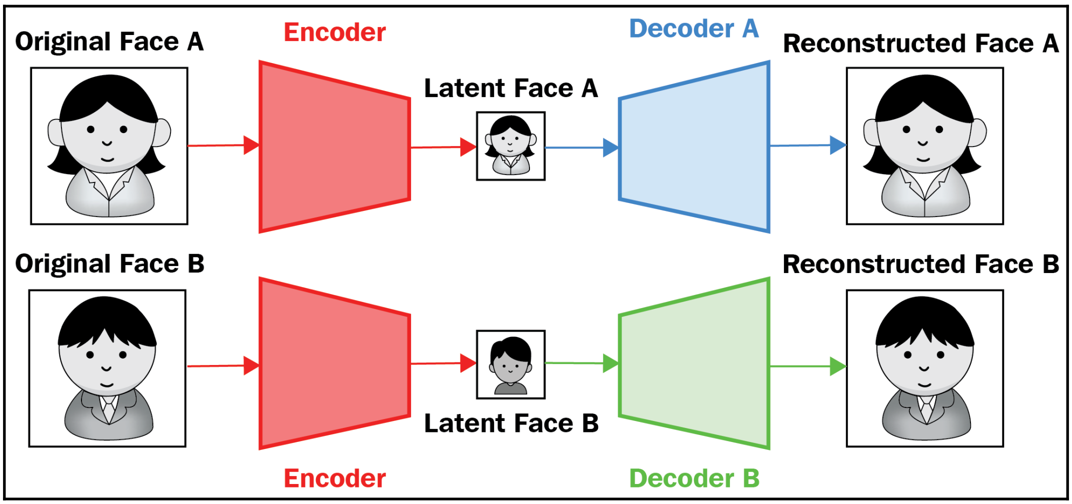
Generating deep fakes

We have learned about two different image-to-image tasks so far: semantic segmentation with UNet and image reconstruction with autoencoders. Deep fakery is an image-to-image task that has a very similar underlying theory.

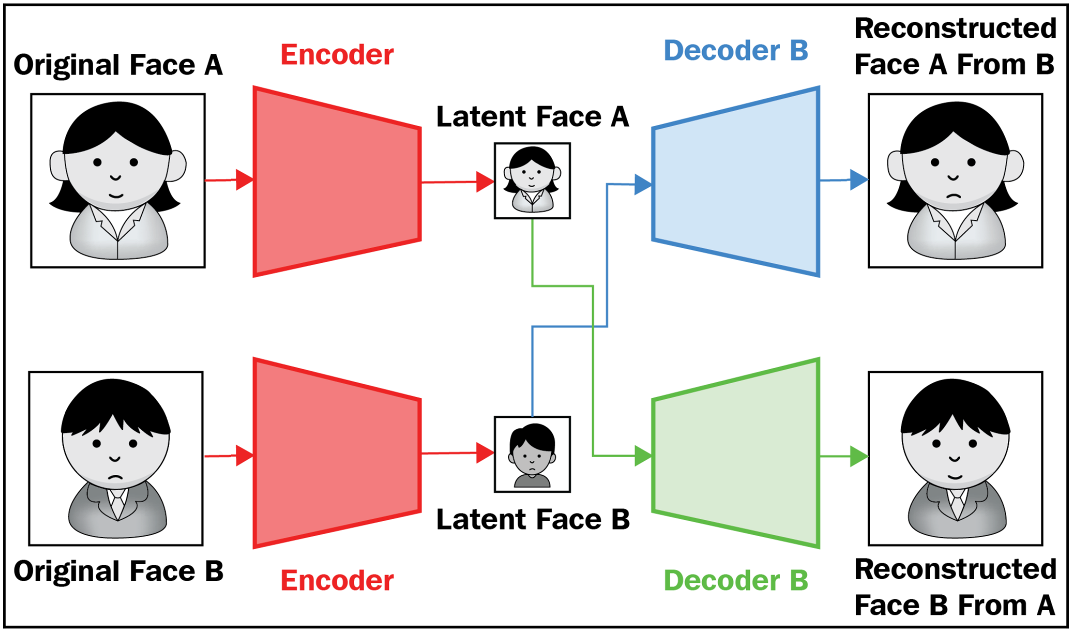
Imagine a scenario where you want to create an application that takes a given image of a face and changes the facial expression in a way that you want. Deep fakes come in handy in this scenario. While we will not discuss the very latest in deep fakes in this book, techniques such as few-shot adversarial learning are developed to generate realistic images with the facial expression of interest. Knowledge of how deep fakes work and GANs (which you will learn about in the next chapters) will help you identify videos that are fake videos.

In the task of deep fakery, we would have a few hundred pictures of person A and a few hundred pictures of person B. The objective is to reconstruct person B's face with the facial expression of person A and vice versa.

The following diagram explains how the deep fake image generation process works:



In the preceding picture, we are passing images of person A and person B through an encoder (**Encoder**). Once we get the latent vectors corresponding to person A (**Latent Face A**) and person B (**Latent Face B**), we pass the latent vectors through their corresponding decoders (**Decoder A** and **Decoder B**) to fetch the corresponding original images (**Reconstructed Face A** and **Reconstructed Face B**). So far, the concept of encoder and decoder is very similar to what we learned in the *Autoencoders* section. However, in this scenario, *we have only one encoder, but two decoders* (each decoder corresponding to a different person). The expectation is that the latent vectors obtained from the encoder represent the information about the facial expression present within the image, while the decoder fetches the image corresponding to the person. Once the encoder and the two decoders are trained, while performing deep fake image generation, we switch the connection within our architecture as follows:



When the latent vector of person A is passed through decoder B, the reconstructed face of person B will have the characteristics of person A (a smiling face) and vice versa for person B when passed through decoder A (a sad face).

TIP

One additional trick that helps in generating a realistic image is warping face images and feeding them to the network in such a way that the warped face is the input and the original image is expected as the output.

Now that we understand how it works, let's implement the generation of fake images of one person with the expression of another person using autoencoders with the following code:

> The following code is available as `Generating_Deep_Fakes.ipynb` in the `Chapter11` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from and is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components in text.

1. Let's download the data and the source code as follows:

```
import os
if not os.path.exists('Faceswap-Deepfake-Pytorch'):
    !wget -q
https://www.dropbox.com/s/5ji7jl7httso9ny/person_images.zip
    !wget -q
https://raw.githubusercontent.com/sizhky/deep-fake-util/main/r
andom_warp.py
    !unzip -q person_images.zip
!pip install -q torch_snippets torch_summary
from torch_snippets import *
from random_warp import get_training_data
```

2. Fetch face crops from the images:

- Define the face cascade, which draws a bounding box around the face in an image. There's more on cascades in *Chapter 18, OpenCV Utilities for Image Analysis*. However, for now, it suffices to say that the face cascade draws a tight bounding box around the face present in the image:

```
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + \
                        'haarcascade_frontalface_default.xml')
```

- Define a function (`crop_face`) for cropping faces from an image:

```
def crop_face(img):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray, 1.3, 5)
    if(len(faces)>0):
        for (x,y,w,h) in faces:
            img2 = img[y:(y+h),x:(x+w),:]
        img2 = cv2.resize(img2,(256,256))
        return img2, True
```

```
        else:
            return img, False
```

In the preceding function, we are passing the grayscaled image (`gray`) through face cascade and cropping the rectangle that contains the face. Next, we are returning a re-sized image (`img2`). Further, to account for a scenario where there is no face detected in the image, we are passing a flag to show whether a face is detected.

- Crop the images of `personA` and `personB` and place them in separate folders:

```
!mkdir cropped_faces_personA
!mkdir cropped_faces_personB

def crop_images(folder):
    images = Glob(folder+'/*.jpg')
    for i in range(len(images)):
        img = read(images[i],1)
        img2, face_detected = crop_face(img)
        if(face_detected==False):
            continue
        else:
            cv2.imwrite('cropped_faces_'+folder+'/'+str(i)+ \
                '.jpg',cv2.cvtColor(img2, cv2.COLOR_RGB2BGR))
crop_images('personA')
crop_images('personB')
```

3. Create a dataloader and inspect the data:

```
class ImageDataset(Dataset):
    def __init__(self, items_A, items_B):
        self.items_A = np.concatenate([read(f,1)[None] \
                                       for f in items_A])/255.
        self.items_B = np.concatenate([read(f,1)[None] \
                                       for f in items_B])/255.
        self.items_A += self.items_B.mean(axis=(0, 1, 2)) \
                        - self.items_A.mean(axis=(0, 1, 2))

    def __len__(self):
        return min(len(self.items_A), len(self.items_B))
    def __getitem__(self, ix):
        a, b = choose(self.items_A), choose(self.items_B)
        return a, b

    def collate_fn(self, batch):
        imsA, imsB = list(zip(*batch))
```

```
            imsA, targetA = get_training_data(imsA, len(imsA))
            imsB, targetB = get_training_data(imsB, len(imsB))
            imsA, imsB, targetA, targetB = [torch.Tensor(i)\
                                            .permute(0,3,1,2)\
                                            .to(device) \
                                            for i in [imsA, imsB,\
                                            targetA, targetB]]
            return imsA, imsB, targetA, targetB

    a = ImageDataset(Glob('cropped_faces_personA'), \
                     Glob('cropped_faces_personB'))
    x = DataLoader(a, batch_size=32, collate_fn=a.collate_fn)
```

The dataloader is returning four tensors, `imsA`, `imsB`, `targetA`, and `targetB`. The first tensor (`imsA`) is a distorted (warped) version of the third tensor (`targetA`) and the second (`imsB`) is a distorted (warped) version of the fourth tensor (`targetB`).

Also, as you can see in the line `a =ImageDataset(Glob('cropped_faces_personA'), Glob('cropped_faces_personB'))`, we have two folders of images, one for each person. There is no relation between any of the faces, and in the `__iteritems__` dataset, we are randomly fetching two faces every time.

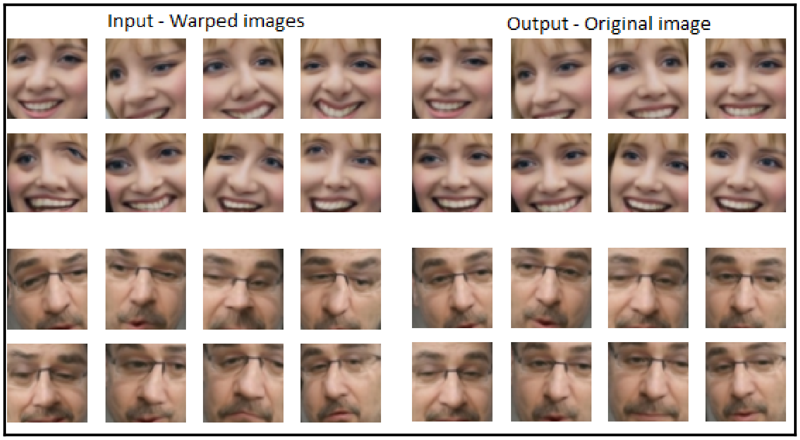
The key function in this step is `get_training_data`, present in `collate_fn`. This is an augmentation function for warping (distorting) faces. We are giving distorted faces as input to the autoencoder and trying to predict regular faces.

> TIP: The advantage of warping is that not only does it increase our training data size but also acts as a regularizer to the network, which is forced to understand key facial features despite being given a distorted face.

- Let's inspect a few images:

```
inspect(*next(iter(x)))

for i in next(iter(x)):
    subplots(i[:8], nc=4, sz=(4,2))
```

The preceding code results in the following output:



Note that the input images are warped, while the output images are not, and the input to output images now have a one-to-one correspondence.

4. Build the model and inspect it:

- Define the convolution (`_ConvLayer`) and upscaling (`_UpScale`) functions as well as the `Reshape` class that will be leveraged while building model:

```
def _ConvLayer(input_features, output_features):
    return nn.Sequential(
        nn.Conv2d(input_features, output_features,
                  kernel_size=5, stride=2, padding=2),
        nn.LeakyReLU(0.1, inplace=True)
    )

def _UpScale(input_features, output_features):
    return nn.Sequential(
        nn.ConvTranspose2d(input_features, output_features,
                           kernel_size=2, stride=2, padding=0),
        nn.LeakyReLU(0.1, inplace=True)
    )

class Reshape(nn.Module):
    def forward(self, input):
        output = input.view(-1, 1024, 4, 4) # channel * 4 * 4
        return output
```

- Define the `Autoencoder` model class, which has a single `encoder` and two decoders (`decoder_A` and `decoder_B`):

```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()

        self.encoder = nn.Sequential(
                        _ConvLayer(3, 128),
                        _ConvLayer(128, 256),
                        _ConvLayer(256, 512),
                        _ConvLayer(512, 1024),
                        nn.Flatten(),
                        nn.Linear(1024 * 4 * 4, 1024),
                        nn.Linear(1024, 1024 * 4 * 4),
                        Reshape(),
                        _UpScale(1024, 512),
                    )

        self.decoder_A = nn.Sequential(
                        _UpScale(512, 256),
                        _UpScale(256, 128),
                        _UpScale(128, 64),
                        nn.Conv2d(64, 3, kernel_size=3, \
                                  padding=1),
                        nn.Sigmoid(),
                    )

        self.decoder_B = nn.Sequential(
                        _UpScale(512, 256),
                        _UpScale(256, 128),
                        _UpScale(128, 64),
                        nn.Conv2d(64, 3, kernel_size=3, \
                                  padding=1),
                        nn.Sigmoid(),
                    )

    def forward(self, x, select='A'):
        if select == 'A':
            out = self.encoder(x)
            out = self.decoder_A(out)
        else:
            out = self.encoder(x)
            out = self.decoder_B(out)
        return out
```

- Generate a summary of the model:

```
from torchsummary import summary
model = Autoencoder()
summary(model, torch.zeros(32,3,64,64), 'A');
```

The preceding code generates the following output:

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 512, 8, 8]           --
|    └─Sequential: 2-1                   [-1, 128, 32, 32]         --
|    |    └─Conv2d: 3-1                  [-1, 128, 32, 32]         9,728
|    |    └─LeakyReLU: 3-2               [-1, 128, 32, 32]         --
|    └─Sequential: 2-2                   [-1, 256, 16, 16]         --
|    |    └─Conv2d: 3-3                  [-1, 256, 16, 16]         819,456
|    |    └─LeakyReLU: 3-4               [-1, 256, 16, 16]         --
|    └─Sequential: 2-3                   [-1, 512, 8, 8]           --
|    |    └─Conv2d: 3-5                  [-1, 512, 8, 8]           3,277,312
|    |    └─LeakyReLU: 3-6               [-1, 512, 8, 8]           --
|    └─Sequential: 2-4                   [-1, 1024, 4, 4]          --
|    |    └─Conv2d: 3-7                  [-1, 1024, 4, 4]          13,108,224
|    |    └─LeakyReLU: 3-8               [-1, 1024, 4, 4]          --
|    └─Flatten: 2-5                      [-1, 16384]               --
|    └─Linear: 2-6                       [-1, 1024]                16,778,240
|    └─Linear: 2-7                       [-1, 16384]               16,793,600
|    └─Reshape: 2-8                      [-1, 1024, 4, 4]          --
|    └─Sequential: 2-9                   [-1, 512, 8, 8]           --
|    |    └─ConvTranspose2d: 3-9         [-1, 512, 8, 8]           2,097,664
|    |    └─LeakyReLU: 3-10              [-1, 512, 8, 8]           --
├─Sequential: 1-2                        [-1, 3, 64, 64]           --
|    └─Sequential: 2-10                  [-1, 256, 16, 16]         --
|    |    └─ConvTranspose2d: 3-11        [-1, 256, 16, 16]         524,544
|    |    └─LeakyReLU: 3-12              [-1, 256, 16, 16]         --
|    └─Sequential: 2-11                  [-1, 128, 32, 32]         --
|    |    └─ConvTranspose2d: 3-13        [-1, 128, 32, 32]         131,200
|    |    └─LeakyReLU: 3-14              [-1, 128, 32, 32]         --
|    └─Sequential: 2-12                  [-1, 64, 64, 64]          --
|    |    └─ConvTranspose2d: 3-15        [-1, 64, 64, 64]          32,832
|    |    └─LeakyReLU: 3-16              [-1, 64, 64, 64]          --
|    └─Conv2d: 2-13                      [-1, 3, 64, 64]           1,731
|    └─Sigmoid: 2-14                     [-1, 3, 64, 64]           --
==========================================================================================
Total params: 53,574,531
Trainable params: 53,574,531
Non-trainable params: 0
Total mult-adds (G): 1.29
```

5. Define the `train_batch` logic:

```
def train_batch(model, data, criterion, optimizes):
    optA, optB = optimizers
    optA.zero_grad()
    optB.zero_grad()
    imgA, imgB, targetA, targetB = data
    _imgA, _imgB = model(imgA, 'A'), model(imgB, 'B')
```

```
        lossA = criterion(_imgA, targetA)
        lossB = criterion(_imgB, targetB)
        lossA.backward()
        lossB.backward()

        optA.step()
        optB.step()

        return lossA.item(), lossB.item()
```

What we are interested in is running `model(imgA, 'B')` (which would return an image of class B using an input image from class A), but we do not have a ground truth to compare it against. So instead, what we are doing is predicting `_imgA` from `imgA` (where `imgA` is a distorted version of `targetA`) and comparing `_imgA` with `targetA` using `nn.L1Loss`.

We do not need `validate_batch` as there is no validation dataset. We will predict new images during training and qualitatively see the progress.

6. Create all the required components to train the model:

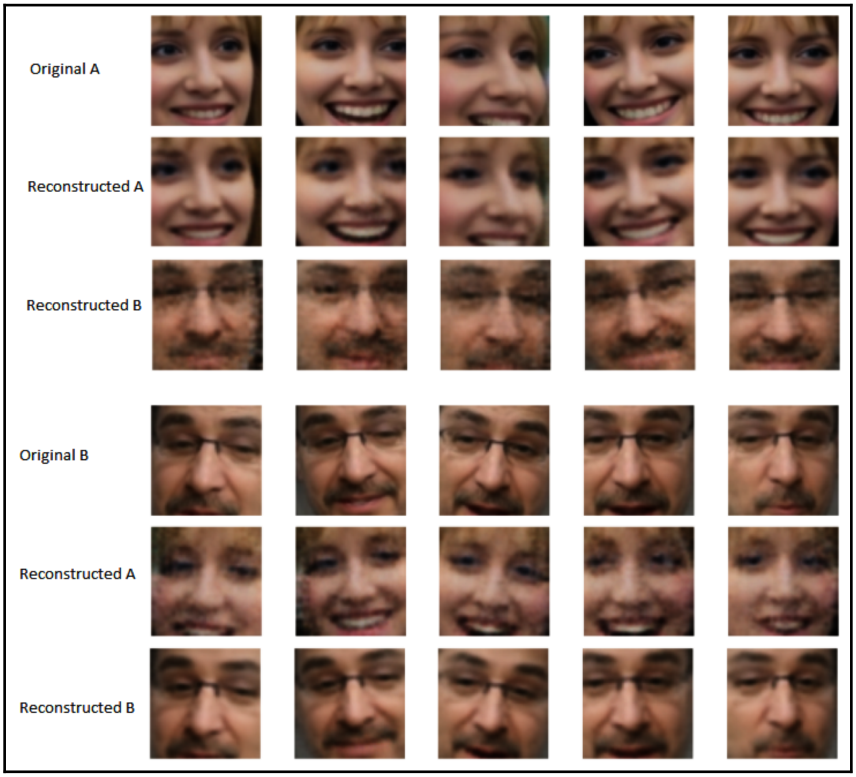
```
model = Autoencoder().to(device)

dataset = ImageDataset(Glob('cropped_faces_personA'), \
                       Glob('cropped_faces_personB'))
dataloader = DataLoader(dataset, 32, \
                        collate_fn=dataset.collate_fn)

optimizers=optim.Adam( \
                [{'params': model.encoder.parameters()}, \
                 {'params': model.decoder_A.parameters()}], \
                 lr=5e-5, betas=(0.5, 0.999)), \
        optim.Adam([{'params': model.encoder.parameters()}, \
                 {'params': model.decoder_B.parameters()}], \
                        lr=5e-5, betas=(0.5, 0.999))
criterion = nn.L1Loss()
```

7. Train the model:

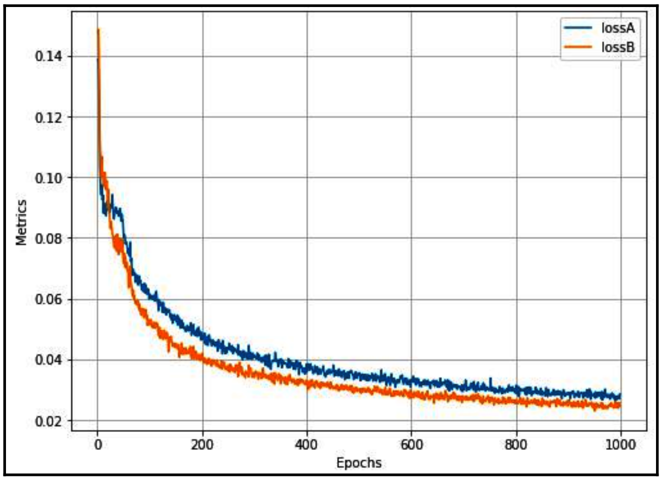
```
n_epochs = 1000
log = Report(n_epochs)
!mkdir checkpoint
for ex in range(n_epochs):
    N = len(dataloader)
    for bx,data in enumerate(dataloader):
        lossA, lossB = train_batch(model, data,
                                   criterion, optimizers)
```

```
                log.record(ex+(1+bx)/N, lossA=lossA,
                           lossB=lossB, end='\r')

            log.report_avgs(ex+1)
            if (ex+1)%100 == 0:
                state = {
                        'state': model.state_dict(),
                        'epoch': ex
                    }
                torch.save(state, './checkpoint/autoencoder.pth')

            if (ex+1)%100 == 0:
                bs = 5
                a,b,A,B = data
                line('A to B')
                _a = model(a[:bs], 'A')
                _b = model(a[:bs], 'B')
                x = torch.cat([A[:bs],_a,_b])
                subplots(x, nc=bs, figsize=(bs*2, 5))

                line('B to A')
                _a = model(b[:bs], 'A')
                _b = model(b[:bs], 'B')
                x = torch.cat([B[:bs],_a,_b])
                subplots(x, nc=bs, figsize=(bs*2, 5))

        log.plot_epochs()
```

The preceding code results in reconstructed images, as follows:



The variation in loss values is as follows:



As you can see, we can swap expressions from one face to another by tweaking an autoencoder to have two decoders instead of one. Furthermore, with a higher number of epochs, the reconstructed image gets more realistic.

Summary

In this chapter, we have learned about the different variants of autoencoders: vanilla, convolutional, and variational. We also learned about how the number of units in the bottleneck layer influences the reconstructed image. Next, we learned about identifying images that are similar to a given image using the t-SNE technique. We learned that when we sample vectors, we cannot get realistic images, and by using variational autoencoders, we learned about generating new images by using a combination of reconstruction loss and KL divergence loss. Next, we learned how to perform an adversarial attack on images to modify the class of an image while not changing the perceptive content of the image. Finally, we learned about leveraging the combination of content loss and gram matrix-based style loss to optimize for content and style loss of images to come up with an image that is a combination of two input images. Finally, we learned about tweaking an autoencoder to swap two faces without any supervision.

Now that we have learned about generating novel images from a given set of images, in the next chapter, we will build upon this topic to generate completely new images using variants of a network called the Generative Adversarial Network.

Questions

1. What is an encoder in an autoencoder?
2. What loss function does an autoencoder optimize for?
3. How do autoencoders help in grouping similar images?
4. When is a convolutional autoencoder useful?
5. Why do we get non-intuitive images if we randomly sample from vector space of embeddings obtained from vanilla/convolutional autoencoders?
6. What are the loss functions that VAEs optimize for?
7. How do VAEs overcome the limitation of vanilla/convolutional autoencoders to generate new images?
8. During an adversarial attack, why do we modify the input image pixels and not the weight values?

9. In a neural style transfer, what are the losses that we optimize for?
10. Why do we consider the activation of different layers and not the original image when calculating style and content loss?
11. Why do we consider gram matrix loss and not the difference between images when calculating style loss?
12. Why do we warp images while building a model to generate deep fakes?

12 Image Generation Using GANs

In the previous chapter, we learned about manipulating an image using neural style transfer and super-imposed the expression in one image on another. However, what if we give the network a bunch of images and ask it to come up with an entirely new image, all on its own?

Generative Adversarial Network (**GAN**) is a step toward achieving the feat of generating an image given a collection of images. In this chapter, we will start by learning about the idea behind what makes GANs work, before building one from scratch. GANs are a vast field that is expanding as we write this book. This chapter will lay the foundation of GANs through three variants of GANs, while we will learn about more advanced GANs and their applications in the next chapter.

In this chapter, we will explore the following topics:

- Introducing GANs
- Using GANs to generate handwritten digits
- Using DCGANs to generate face images
- Implementing conditional GANs

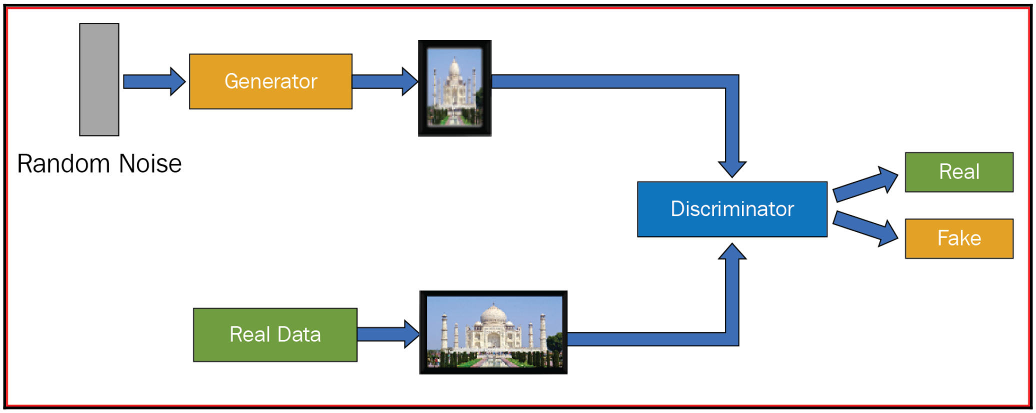
Introducing GANs

To understand GANs, we need to understand two terms: **generator** and **discriminator**. First, we should have a reasonable sample of images of an object. A generative network (generator) learns representation from a sample of images and then generates images similar to the sample of images. A discriminator network (discriminator) is one that looks at the image generated (by the generator network) and the original sample of images and classifies images as original ones or generated (fake) ones.

The generator network generates images in such a way that the discriminator classifies the images as real ones. The discriminator network classifies the generated images as fake and the images in the original sample as real.

Essentially, the adversarial term in GAN represents the opposite nature of the two networks—a generator network, which generates images to fool the discriminator network, and a discriminator network that classifies each image by saying whether the image is generated or is an original.

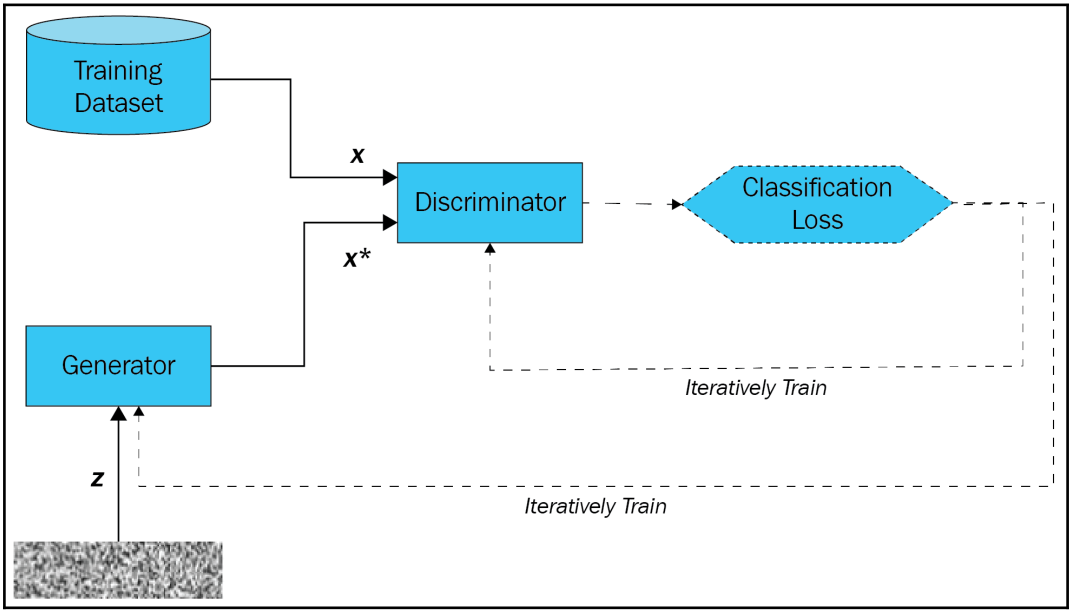
Let's understand the process employed by GANs through the following diagram:



In the preceding diagram, the generator network is generating images from random noise as input. A discriminator network is looking at the images generated by the generator and comparing them with real data (a sample of images that are provided) to specify whether the generated image is real or fake. The generator tries to generate as many realistic images as possible, while the discriminator tries to detect which of the images that are generated by the generator are fake. This way, the generator learns to generate as many realistic images as possible by learning from what the discriminator looks at to identify whether an image is fake.

Typically, the generator and discriminator are trained alternately. This way, it becomes a police-and-thief game, where the generator is the thief trying to generate fake data, while the discriminator is the police trying to identify the available data as real or fake.

Let's now understand how we compute the loss values for both the generator and discriminator to train both the networks together using the following diagram and steps:



The steps involved in training GANs are as follows:

1. Train the generator (and not the discriminator) to generate images such that the discriminator classifies the images as real.
2. Train the discriminator (and not the generator) to classify the images that the generator generates as fake.
3. Repeat the process until an equilibrium is achieved.

In the preceding scenario, when the discriminator can detect generated images really well, the loss corresponding to the generator is much higher when compared to the loss corresponding to the discriminator.

Thus, the gradients adjust in such a way that the generator would have a loss. However, it would tip the discriminator loss to a higher side. In the next iteration, the gradients adjust so that the discriminator loss is lower. This way, the generator and discriminator keep getting trained until a point where the generator generates realistic images and the discriminator cannot distinguish between a real or a generated image.

With this understanding, let's generate images relating to the MNIST dataset in the next section.

Using GANs to generate handwritten digits

To generate images of handwritten digits, we will leverage the same network as we learned about in the previous section. The strategy we will adopt is as follows:

1. Import MNIST data.
2. Initialize random noise.
3. Define the generator model.
4. Define the discriminator model.
5. Train the two models alternately.
6. Let the model train until the generator and discriminator losses are largely the same.

Let's execute each of the preceding steps in the following code:

> The following code is available as `Handwritten_digit_generation_using_GAN.ipynb` in the `Chapter12` folder in this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components from text.

1. Import the relevant packages and define the device:

```
!pip install -q torch_snippets
from torch_snippets import *
device = "cuda" if torch.cuda.is_available() else "cpu"
from torchvision.utils import make_grid
```

2. Import the `MNIST` data and define the dataloader with built-in data transformation so that the input data is scaled to a mean of 0.5 and a standard deviation of 0.5:

```
from torchvision.datasets import MNIST
from torchvision import transforms

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

data_loader = torch.utils.data.DataLoader(MNIST('~/data', \
            train=True, download=True, transform=transform), \
            batch_size=128, shuffle=True, drop_last=True)
```

3. Define the `Discriminator` model class:

```
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
                                nn.Linear(784, 1024),
                                nn.LeakyReLU(0.2),
                                nn.Dropout(0.3),
                                nn.Linear(1024, 512),
                                nn.LeakyReLU(0.2),
                                nn.Dropout(0.3),
                                nn.Linear(512, 256),
                                nn.LeakyReLU(0.2),
                                nn.Dropout(0.3),
                                nn.Linear(256, 1),
                                nn.Sigmoid()
                            )
    def forward(self, x): return self.model(x)
```

Note that, in the preceding code, in place of `ReLU`, we have used `LeakyReLU` as the activation function. A summary of the discriminator network is as follows:

```
!pip install torch_summary
from torchsummary import summary
discriminator = Discriminator().to(device)
summary(discriminator,torch.zeros(1,784))
```

The preceding code generates the following output:

```
================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
================================================================================
├─Sequential: 1-1                        [-1, 1]                   --
|    └─Linear: 2-1                       [-1, 1024]                803,840
|    └─LeakyReLU: 2-2                    [-1, 1024]                --
|    └─Dropout: 2-3                      [-1, 1024]                --
|    └─Linear: 2-4                       [-1, 512]                 524,800
|    └─LeakyReLU: 2-5                    [-1, 512]                 --
|    └─Dropout: 2-6                      [-1, 512]                 --
|    └─Linear: 2-7                       [-1, 256]                 131,328
|    └─LeakyReLU: 2-8                    [-1, 256]                 --
|    └─Dropout: 2-9                      [-1, 256]                 --
|    └─Linear: 2-10                      [-1, 1]                   257
|    └─Sigmoid: 2-11                     [-1, 1]                   --
================================================================================
Total params: 1,460,225
Trainable params: 1,460,225
Non-trainable params: 0
Total mult-adds (M): 2.92
```

4. Define the `Generator` model class:

```
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
                                nn.Linear(100, 256),
                                nn.LeakyReLU(0.2),
                                nn.Linear(256, 512),
                                nn.LeakyReLU(0.2),
                                nn.Linear(512, 1024),
                                nn.LeakyReLU(0.2),
                                nn.Linear(1024, 784),
                                nn.Tanh()
                            )

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

Note that the generator takes a 100-dimensional input (which is of random noise) and generates an image from the input. A summary of the generator model is as follows:

```
generator = Generator().to(device)
summary(generator,torch.zeros(1,100))
```

The preceding code generates the following output:

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 784]                 --
|    └─Linear: 2-1                       [-1, 256]                 25,856
|    └─LeakyReLU: 2-2                    [-1, 256]                 --
|    └─Linear: 2-3                       [-1, 512]                 131,584
|    └─LeakyReLU: 2-4                    [-1, 512]                 --
|    └─Linear: 2-5                       [-1, 1024]                525,312
|    └─LeakyReLU: 2-6                    [-1, 1024]                --
|    └─Linear: 2-7                       [-1, 784]                 803,600
|    └─Tanh: 2-8                         [-1, 784]                 --
==========================================================================================
Total params: 1,486,352
Trainable params: 1,486,352
Non-trainable params: 0
Total mult-adds (M): 2.97
```

5. Define a function to generate random noise and register it to the device:

```
def noise(size):
    n = torch.randn(size, 100)
    return n.to(device)
```

6. Define a function to train the discriminator:

- The discriminator training function (`discriminator_train_step`) takes real data (`real_data`) and fake data (`fake_data`) as input:

```
def discriminator_train_step(real_data, fake_data):
```

- Reset the gradients:

```
    d_optimizer.zero_grad()
```

- Predict on the real data (`real_data`) and calculate the loss (`error_real`) before performing backpropagation on the loss value:

```
    prediction_real = discriminator(real_data)
    error_real = loss(prediction_real, \
                      torch.ones(len(real_data),1).to(device))
    error_real.backward()
```

When we calculate the discriminator loss on real data, we expect the discriminator to predict an output of 1. Hence, the discriminator loss on real data is calculated by expecting the discriminator to predict output as 1 using `torch.ones` during discriminator training.

- Predict on the fake data (`fake_data`) and calculate the loss (`error_fake`) before performing backpropagation on the loss value:

```
prediction_fake = discriminator(fake_data)
error_fake = loss(prediction_fake, \
            torch.zeros(len(fake_data),1).to(device))
error_fake.backward()
```

When we calculate the discriminator loss on fake data, we expect the discriminator to predict an output of 0. Hence, the discriminator loss on fake data is calculated by expecting the discriminator to predict output as 0 using `torch.zeros` during discriminator training.

- Update the weights and return the overall loss (summing up the loss values of `error_real` on `real_data` and `error_fake` on `fake_data`):

```
d_optimizer.step()
return error_real + error_fake
```

7. Train the generator model:

- Define the generator training function (`generator_train_step`) that takes fake data (`fake_data`):

```
def generator_train_step(fake_data):
```

- Reset the gradients of the generator optimizer:

```
g_optimizer.zero_grad()
```

- Predict the output of the discriminator on fake data (`fake_data`):

```
prediction = discriminator(fake_data)
```

- Calculate the generator loss value by passing `prediction` and the expected value as `torch.ones` since we want to fool the discriminator to output a value of `1` when training the generator:

```
error = loss(prediction, \
             torch.ones(len(real_data),1).to(device))
```

- Perform backpropagation, update the weights, and return the error:

```
error.backward()
g_optimizer.step()
return error
```

8. Define the model objects, the optimizer for each generator and discriminator, and the loss function to optimize:

```
discriminator = Discriminator().to(device)
generator = Generator().to(device)
d_optimizer= optim.Adam(discriminator.parameters(),lr=0.0002)
g_optimizer = optim.Adam(generator.parameters(), lr=0.0002)
loss = nn.BCELoss()
num_epochs = 200
log = Report(num_epochs)
```

9. Run the models over increasing epochs:

- Loop through 200 epochs (`num_epochs`) over the `data_loader` function obtained in *step 2*:

```
for epoch in range(num_epochs):
    N = len(data_loader)
    for i, (images, _) in enumerate(data_loader):
```

- Load real data (`real_data`) and fake data, where fake data (`fake_data`) is obtained by passing `noise` (with a batch size of the number of data points in `real_data` – `len(real_data)`) through the `generator` network. Note that it is important to run `fake_data.detach()`, or else training will not work. On detaching, we are creating a fresh copy of the tensor so that when `error.backward()` is called in `discriminator_train_step`, the tensors associated with the generator (which create `fake_data`) are not affected:

```
        real_data = images.view(len(images), -1).to(device)
        fake_data=generator(noise(len(real_data))).to(device)
        fake_data = fake_data.detach()
```

- Train the discriminator using the `discriminator_train_step` function defined in *step 6*:

```
        d_loss=discriminator_train_step(real_data, fake_data)
```

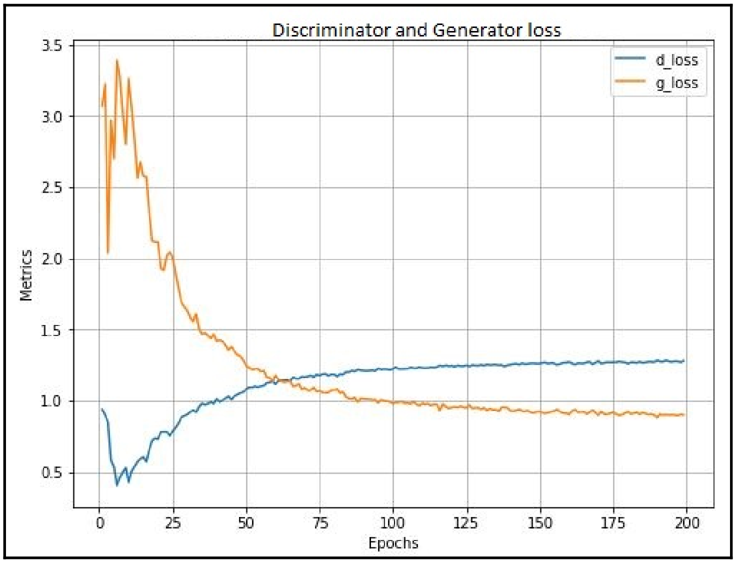
- Now that we have trained the discriminator, let's train the generator in this step. Generate a new set of fake images (`fake_data`) from noisy data and train the generator using `generator_train_step` defined in *step 6*:

```
        fake_data=generator(noise(len(real_data))).to(device)
        g_loss = generator_train_step(fake_data)
```

- Record the losses:

```
        log.record(epoch+(1+i)/N, d_loss=d_loss.item(), \
                   g_loss=g_loss.item(), end='\r')
    log.report_avgs(epoch+1)
log.plot_epochs(['d_loss', 'g_loss'])
```

The discriminator and generator losses over increasing epochs are as follows:



10. Visualize the fake data post-training:

```
z = torch.randn(64, 100).to(device)
sample_images = generator(z).data.cpu().view(64, 1, 28, 28)
grid = make_grid(sample_images, nrow=8, normalize=True)
show(grid.cpu().detach().permute(1,2,0), sz=5)
```

The preceding code generates the following output:

From this, we can see that we can leverage GANs to generate images that are realistic, but still with some scope for improvement. In the next section, we will learn about using deep convolutional GANs to generate more realistic images.

Using DCGANs to generate face images

In the previous section, we learned about generating images using GANs. However, we have already seen that **Convolutional Neural Networks** (**CNNs**) perform better in the context of images when compared to vanilla neural networks. In this section, we will learn about generating images using **Deep Convolutional Generative Adversarial Networks** (**DCGANs**), which use convolution and pooling operations in the model.

First, let's understand the technique we will leverage to generate an image using a set of 100 random numbers. We will first convert noise into a shape of batch size x 100 x 1 x 1. The reason for appending additional channel information in DCGANs and not doing it in the GAN section is that we will leverage CNNs in this section, which requires inputs in the form of batch size x channels x height x width.

Next, we convert the generated noise into an image by leveraging `ConvTranspose2d`.

As we learned in `Chapter 9`, *Image Segmentation*, `ConvTranspose2d` does the opposite of a convolution operation, which is to take input with a smaller feature map size (height x width) and upsample it to that of a larger size using a predefined kernel size, stride, and padding. This way, we would gradually convert a vector from a shape of batch size x 100 x 1 x 1 into a shape of batch size x 3 x 64 x 64. With this, we have taken a random noise vector of size 100 and converted it into an image of a face.

With this understanding, let's now build a model to generate images of faces:

> The following code is available as `Face_generation_using_DCGAN.ipynb` in the `Chapter12` folder in this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from and is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components from text.

1. Download and extract the face images:

```
!wget
https://www.dropbox.com/s/rbajpdlh7efkdo1/male_female_face_ima
ges.zip
!unzip male_female_face_images.zip
```

A sample of the images is shown here:

2. Import the relevant packages:

```
!pip install -q --upgrade torch_snippets
from torch_snippets import *
import torchvision
from torchvision import transforms
import torchvision.utils as vutils
import cv2, numpy as np, pandas as pd
device = "cuda" if torch.cuda.is_available() else "cpu"
```

3. Define the dataset and dataloader:

    - Ensure that we crop the images so that we retain only the faces and discard additional details in the image. First, we will download the cascade filter (more on cascade filters in OpenCV in `Chapter 18`, *Using OpenCV Utilities for Image Analysis*), which will help in identifying faces within an image:

```
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + \
                        'haarcascade_frontalface_default.xml')
```

    - Create a new folder and dump all the cropped face images into the new folder:

```
!mkdir cropped_faces
images = Glob('/content/females/*.jpg') + \
            Glob('/content/males/*.jpg')
for i in range(len(images)):
    img = read(images[i],1)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray, 1.3, 5)
    for (x,y,w,h) in faces:
        img2 = img[y:(y+h),x:(x+w),:]
    cv2.imwrite('cropped_faces/'+str(i)+'.jpg', \
                cv2.cvtColor(img2, cv2.COLOR_RGB2BGR))
```

A sample of the cropped faces is as follows:

Note that, by cropping and keeping faces only, we are retaining only the information that we want to generate.

- Specify the transformation to perform on each image:

```
transform=transforms.Compose([
                               transforms.Resize(64),
                               transforms.CenterCrop(64),
                               transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

- Define the `Faces` dataset class:

```
class Faces(Dataset):
    def __init__(self, folder):
        super().__init__()
        self.folder = folder
        self.images = sorted(Glob(folder))
    def __len__(self):
        return len(self.images)
    def __getitem__(self, ix):
        image_path = self.images[ix]
        image = Image.open(image_path)
        image = transform(image)
        return image
```

- Create the dataset object – `ds`:

```
ds = Faces(folder='cropped_faces/')
```

- Define the `dataloader` class as follows:

```
dataloader = DataLoader(ds, batch_size=64, shuffle=True, \
                        num_workers=8)
```

4. Define weight initialization so that the weights have a smaller spread:

```
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
```

5. Define the `Discriminator` model class, which takes an image of a shape of batch size x 3 x 64 x 64 and predicts whether it is real or fake:

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
                        nn.Conv2d(3,64,4,2,1,bias=False),
                        nn.LeakyReLU(0.2,inplace=True),
                        nn.Conv2d(64,64*2,4,2,1,bias=False),
                        nn.BatchNorm2d(64*2),
                        nn.LeakyReLU(0.2,inplace=True),
                        nn.Conv2d(64*2,64*4,4,2,1,bias=False),
                        nn.BatchNorm2d(64*4),
                        nn.LeakyReLU(0.2,inplace=True),
                        nn.Conv2d(64*4,64*8,4,2,1,bias=False),
                        nn.BatchNorm2d(64*8),
                        nn.LeakyReLU(0.2,inplace=True),
                        nn.Conv2d(64*8,1,4,1,0,bias=False),
                        nn.Sigmoid()
                    )
        self.apply(weights_init)
def forward(self, input):
    return self.model(input)
```

- Obtain a summary of the defined model:

```
!pip install torch_summary
from torchsummary import summary
discriminator = Discriminator().to(device)
summary(discriminator,torch.zeros(1,3,64,64));
```

The preceding code generates the following output:

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 1, 1, 1]             --
|    └─Conv2d: 2-1                       [-1, 64, 32, 32]          3,072
|    └─LeakyReLU: 2-2                    [-1, 64, 32, 32]          --
|    └─Conv2d: 2-3                       [-1, 128, 16, 16]         131,072
|    └─BatchNorm2d: 2-4                  [-1, 128, 16, 16]         256
|    └─LeakyReLU: 2-5                    [-1, 128, 16, 16]         --
|    └─Conv2d: 2-6                       [-1, 256, 8, 8]           524,288
|    └─BatchNorm2d: 2-7                  [-1, 256, 8, 8]           512
|    └─LeakyReLU: 2-8                    [-1, 256, 8, 8]           --
|    └─Conv2d: 2-9                       [-1, 512, 4, 4]           2,097,152
|    └─BatchNorm2d: 2-10                 [-1, 512, 4, 4]           1,024
|    └─LeakyReLU: 2-11                   [-1, 512, 4, 4]           --
|    └─Conv2d: 2-12                      [-1, 1, 1, 1]             8,192
|    └─Sigmoid: 2-13                     [-1, 1, 1, 1]             --
==========================================================================================
Total params: 2,765,568
Trainable params: 2,765,568
Non-trainable params: 0
Total mult-adds (M): 106.58
```

6. Define the `Generator` model class that generates fake images from an input of shape batch size x 100 x 1 x 1:

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator,self).__init__()
        self.model = nn.Sequential(
            nn.ConvTranspose2d(100,64*8,4,1,0,bias=False,),
            nn.BatchNorm2d(64*8),
            nn.ReLU(True),
            nn.ConvTranspose2d(64*8,64*4,4,2,1,bias=False),
            nn.BatchNorm2d(64*4),
            nn.ReLU(True),
            nn.ConvTranspose2d( 64*4,64*2,4,2,1,bias=False),
            nn.BatchNorm2d(64*2),
```

```
                nn.ReLU(True),
                nn.ConvTranspose2d( 64*2,64,4,2,1,bias=False),
                nn.BatchNorm2d(64),
                nn.ReLU(True),
                nn.ConvTranspose2d( 64,3,4,2,1,bias=False),
                nn.Tanh()
            )
            self.apply(weights_init)
        def forward(self,input): return self.model(input)
```

- Obtain a summary of the defined model:

```
generator = Generator().to(device)
summary(generator,torch.zeros(1,100,1,1))
```

The preceding code generates the following output:

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 3, 64, 64]           --
|    └─ConvTranspose2d: 2-1              [-1, 512, 4, 4]           819,200
|    └─BatchNorm2d: 2-2                  [-1, 512, 4, 4]           1,024
|    └─ReLU: 2-3                         [-1, 512, 4, 4]           --
|    └─ConvTranspose2d: 2-4              [-1, 256, 8, 8]           2,097,152
|    └─BatchNorm2d: 2-5                  [-1, 256, 8, 8]           512
|    └─ReLU: 2-6                         [-1, 256, 8, 8]           --
|    └─ConvTranspose2d: 2-7              [-1, 128, 16, 16]         524,288
|    └─BatchNorm2d: 2-8                  [-1, 128, 16, 16]         256
|    └─ReLU: 2-9                         [-1, 128, 16, 16]         --
|    └─ConvTranspose2d: 2-10             [-1, 64, 32, 32]          131,072
|    └─BatchNorm2d: 2-11                 [-1, 64, 32, 32]          128
|    └─ReLU: 2-12                        [-1, 64, 32, 32]          --
|    └─ConvTranspose2d: 2-13             [-1, 3, 64, 64]           3,072
|    └─Tanh: 2-14                        [-1, 3, 64, 64]           --
==========================================================================================
Total params: 3,576,704
Trainable params: 3,576,704
Non-trainable params: 0
Total mult-adds (M): 431.92
```

Note that we have leveraged `ConvTranspose2d` to gradually upsample an array so that it closely resembles an image.

7. Define the functions to train the generator (`generator_train_step`) and the discriminator (`discriminator_train_step`):

```
def discriminator_train_step(real_data, fake_data):
    d_optimizer.zero_grad()
    prediction_real = discriminator(real_data)
    error_real = loss(prediction_real.squeeze(), \
                      torch.ones(len(real_data)).to(device))
    error_real.backward()
    prediction_fake = discriminator(fake_data)
    error_fake = loss(prediction_fake.squeeze(), \
                      torch.zeros(len(fake_data)).to(device))
    error_fake.backward()
    d_optimizer.step()
    return error_real + error_fake

def generator_train_step(fake_data):
    g_optimizer.zero_grad()
    prediction = discriminator(fake_data)
    error = loss(prediction.squeeze(), \
                 torch.ones(len(real_data)).to(device))
    error.backward()
    g_optimizer.step()
    return error
```

In the preceding code, we are performing a `.squeeze` operation on top of the prediction as the output of the model has a shape of batch size x 1 x 1 x 1 and it needs to be compared to a tensor that has a shape of batch size x 1.

8. Create the generator and discriminator model objects, the optimizers, and the loss function of the discriminator to be optimized:

```
discriminator = Discriminator().to(device)
generator = Generator().to(device)
loss = nn.BCELoss()
d_optimizer = optim.Adam(discriminator.parameters(), \
                         lr=0.0002, betas=(0.5, 0.999))
g_optimizer = optim.Adam(generator.parameters(), \
                         lr=0.0002, betas=(0.5, 0.999))
```

9. Run the models over increasing epochs:

   - Loop through 25 epochs over the `dataloader` function defined in *step 3*:

```
log = Report(25)
for epoch in range(25):
```

```
    N = len(dataloader)
    for i, images in enumerate(dataloader):
```

- Load real data (`real_data`) and generate fake data (`fake_data`) by passing through the generator network:

```
        real_data = images.to(device)
        fake_data = generator(torch.randn(len(real_data), \
                            100, 1, 1).to(device)).to(device)
        fake_data = fake_data.detach()
```

Note that the major difference between vanilla GANs and DCGANs when generating `real_data` is that we did not have to flatten `real_data` in the case of DCGANs as we are leveraging CNNs.

- Train the discriminator using the `discriminator_train_step` function defined in *step 7*:

```
        d_loss=discriminator_train_step(real_data, fake_data)
```

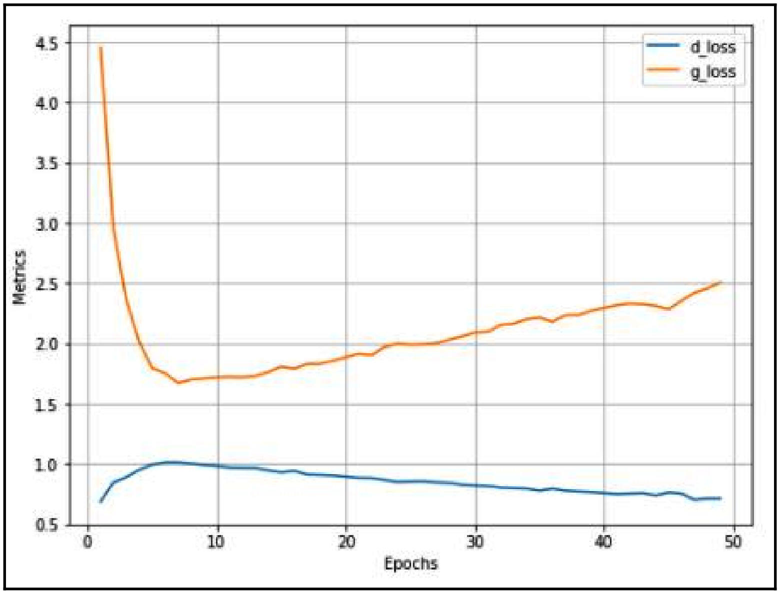
- Generate a new set of images (`fake_data`) from the noisy data (`torch.randn(len(real_data))`) and train the generator using the `generator_train_step` function defined in *step 7*:

```
        fake_data = generator(torch.randn(len(real_data), \
                            100, 1, 1).to(device)).to(device)
        g_loss = generator_train_step(fake_data)
```

- Record the losses:

```
        log.record(epoch+(1+i)/N, d_loss=d_loss.item(), \
                   g_loss=g_loss.item(), end='\r')
    log.report_avgs(epoch+1)
log.plot_epochs(['d_loss','g_loss'])
```

The preceding code generates the following output:



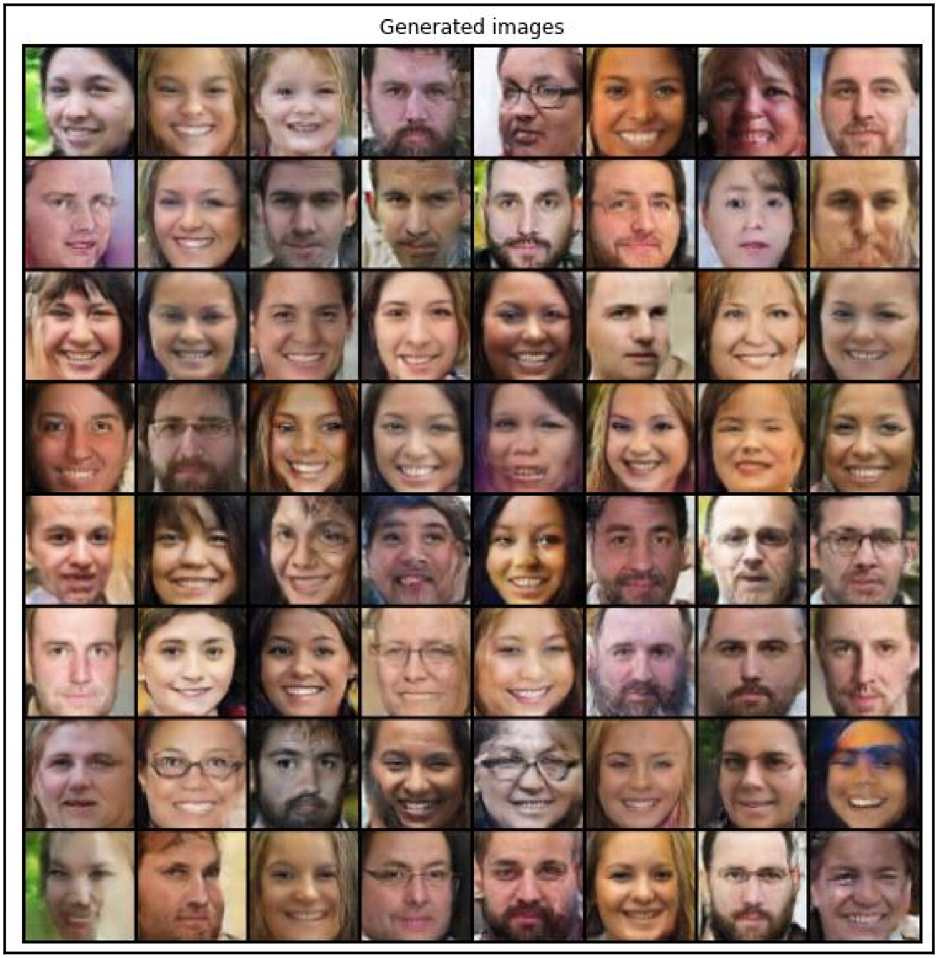
Note that in this setting, the variation in generator and discriminator losses does not follow the pattern that we have seen in the case of handwritten digit generation on account of the following:

1. We are dealing with bigger images (images that are 64 x 64 x 3 in shape when compared to images of 28 x 28 x 1 shape, which we have seen in the previous section).
2. Digits have fewer variations when compared to the features that are present in the image of a face.
3. Information in handwritten digits is available in only a minority of pixels when compared to the information in images of a face.

Once the training process is complete, generate a sample of images using the following code:

```
generator.eval()
noise = torch.randn(64, 100, 1, 1, device=device)
sample_images = generator(noise).detach().cpu()
grid = vutils.make_grid(sample_images,nrow=8,normalize=True)
show(grid.cpu().detach().permute(1,2,0), sz=10, \
     title='Generated images')
```

The preceding code generates the following set of images:



Note that while the generator generated images of a face from random noise, the images are decent but still not sufficiently realistic. One potential reason is that not all input images have the same face alignment. As an exercise, we suggest you train the DCGAN only on those images where there is no tilted face and the person is looking straight into the camera in the original image. In addition, we suggest you try and contrast the generated images with high discriminator scores to the ones with low discriminator scores.

In this section, we have learned about generating images of a face. However, we cannot specify the generation of an image that is of interest to us. In the next section, we will work toward generating images of a specific class.

Implementing conditional GANs

Imagine a scenario where we want to generate an image of a class of our interest; for example, an image of a cat or an image of a dog, or an image of a man with spectacles. How do we specify that we want to generate an image of interest to us? Conditional GANs come to the rescue in this scenario.

For now, let's assume that we have the images of male and female faces only along with their corresponding labels. In this section, we will learn about generating images of a specified class of interest from random noise.

The strategy we adopt is as follows:

1. Specify the label of the image we want to generate as a one-hot-encoded version.
2. Pass the label through an embedding layer to generate a multi-dimensional representation of each class.
3. Generate random noise and concatenate with the embedding layer generated in the previous step.
4. Train the model just like we did in the previous sections, but this time with the noise vector concatenated with the embedding of the class of image we wish to generate.

In the following code, we will code up the preceding strategy:

> The following code is available as `Face_generation_using_Conditional_GAN.ipynb` in the `Chapter12` folder in this book's GitHub repository - `https://tinyurl.com/mcvp-packt`. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components from text.

1. Import the images and the relevant packages:

```
!wget
https://www.dropbox.com/s/rbajpdlh7efkdo1/male_female_face_ima
ges.zip
```

```
!unzip male_female_face_images.zip
!pip install -q --upgrade torch_snippets
from torch_snippets import *
device = "cuda" if torch.cuda.is_available() else "cpu"
from torchvision.utils import make_grid
from torch_snippets import *
from PIL import Image
import torchvision
from torchvision import transforms
import torchvision.utils as vutils
```

2. Create the dataset and dataloader:

   - Store the male and female image paths:

```
female_images = Glob('/content/females/*.jpg')
male_images = Glob('/content/males/*.jpg')
```

   - Ensure that we crop the images so that we retain only faces and discard additional details in an image. First, we will download the cascade filter (more on cascade filters in OpenCV in `Chapter 18`, *Using OpenCV Utilities for Image Analysis*), which will help in identifying faces within an image:

```
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + \
                        'haarcascade_frontalface_default.xml')
```

   - Create two new folders (one corresponding to male and another for female images) and dump all the cropped face images into the respective folders:

```
!mkdir cropped_faces_females
!mkdir cropped_faces_males

def crop_images(folder):
    images = Glob(folder+'/*.jpg')
    for i in range(len(images)):
        img = read(female_images[i],1)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(gray, 1.3, 5)
        for (x,y,w,h) in faces:
            img2 = img[y:(y+h),x:(x+w),:]
            cv2.imwrite('cropped_faces_'+folder+'/'+ \
                    str(i)+'.jpg',cv2.cvtColor(img2, \
                                    cv2.COLOR_RGB2BGR))
crop_images('females')
crop_images('males')
```

- Specify the transformation to perform on each image:

```
transform=transforms.Compose([
                               transforms.Resize(64),
                               transforms.CenterCrop(64),
                               transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
                           ])
```

- Create the `Faces` dataset class that returns the image and the corresponding gender of the person in it:

```
class Faces(Dataset):
    def __init__(self, folders):
        super().__init__()
        self.folderfemale = folders[0]
        self.foldermale = folders[1]
        self.images = sorted(Glob(self.folderfemale)) + \
                        sorted(Glob(self.foldermale))
    def __len__(self):
        return len(self.images)
    def __getitem__(self, ix):
        image_path = self.images[ix]
        image = Image.open(image_path)
        image = transform(image)
        gender = np.where('female' in image_path,1,0)
        return image, torch.tensor(gender).long()
```

- Define the `ds` dataset and `dataloader`:

```
ds = Faces(folders=['cropped_faces_females', \
                    'cropped_faces_males'])
dataloader = DataLoader(ds, batch_size=64, \
                        shuffle=True, num_workers=8)
```

3. Define the weight initialization method (just like we did in the DCGAN section) so that we do not have a widespread variation across randomly initialized weight values:

```
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
```

4. Define the `Discriminator` model class:

    - Define the model architecture:

```
class Discriminator(nn.Module):
    def __init__(self, emb_size=32):
        super(Discriminator, self).__init__()
        self.emb_size = 32
        self.label_embeddings = nn.Embedding(2, self.emb_size)
        self.model = nn.Sequential(
            nn.Conv2d(3,64,4,2,1,bias=False),
            nn.LeakyReLU(0.2,inplace=True),
            nn.Conv2d(64,64*2,4,2,1,bias=False),
            nn.BatchNorm2d(64*2),
            nn.LeakyReLU(0.2,inplace=True),
            nn.Conv2d(64*2,64*4,4,2,1,bias=False),
            nn.BatchNorm2d(64*4),
            nn.LeakyReLU(0.2,inplace=True),
            nn.Conv2d(64*4,64*8,4,2,1,bias=False),
            nn.BatchNorm2d(64*8),
            nn.LeakyReLU(0.2,inplace=True),
            nn.Conv2d(64*8,64,4,2,1,bias=False),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2,inplace=True),
            nn.Flatten()
        )
        self.model2 = nn.Sequential(
            nn.Linear(288,100),
            nn.LeakyReLU(0.2,inplace=True),
            nn.Linear(100,1),
            nn.Sigmoid()
        )
        self.apply(weights_init)
```

    Note that in the model class, we have an additional parameter, `emb_size`, present in conditional GANs and not in DCGANs. `emb_size` represents the number of embeddings into which we convert the input class label (the class of image we want to generate), which is stored as `label_embeddings`. The reason we convert the input class label from a one-hot-encoded version to embeddings of a higher dimension is that a model has a higher degree of freedom to learn and adjust to deal with different classes.

While the model class, to a large extent, remains the same as what we have seen in DCGANs, we are initializing another model (`model2`) that does the classification exercise. There will be more about how the second model helps after we discuss the `forward` method next. You will also understand the reason why `self.model2` has 288 values as input after you go through the following `forward` method and the summary of the model:

- Define the `forward` method that takes the image and the label of the image as input:

```
def forward(self, input, labels):
    x = self.model(input)
    y = self.label_embeddings(labels)
    input = torch.cat([x, y], 1)
    final_output = self.model2(input)
    return final_output
```

In the `forward` method defined, we are fetching the output of the first model (`self.model(input)`) and the output of passing `labels` through `label_embeddings` and then concatenating the outputs. Next, we are passing the concatenated outputs through the second model (`self.model2`) we have defined earlier that fetches us the discriminator output.

- Obtain the summary of the defined model:

```
!pip install torch_summary
from torchsummary import summary
discriminator = Discriminator().to(device)
summary(discriminator,torch.zeros(32,3,64,64).to(device), \
        torch.zeros(32).long().to(device));
```

The preceding code generates the following output:

```
==========================================================================================
Layer (type:depth-idx)                        Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                             [-1, 256]                 --
|    └─Conv2d: 2-1                            [-1, 64, 32, 32]          3,072
|    └─LeakyReLU: 2-2                         [-1, 64, 32, 32]          --
|    └─Conv2d: 2-3                            [-1, 128, 16, 16]         131,072
|    └─BatchNorm2d: 2-4                       [-1, 128, 16, 16]         256
|    └─LeakyReLU: 2-5                         [-1, 128, 16, 16]         --
|    └─Conv2d: 2-6                            [-1, 256, 8, 8]           524,288
|    └─BatchNorm2d: 2-7                       [-1, 256, 8, 8]           512
|    └─LeakyReLU: 2-8                         [-1, 256, 8, 8]           --
|    └─Conv2d: 2-9                            [-1, 512, 4, 4]           2,097,152
|    └─BatchNorm2d: 2-10                      [-1, 512, 4, 4]           1,024
|    └─LeakyReLU: 2-11                        [-1, 512, 4, 4]           --
|    └─Conv2d: 2-12                           [-1, 64, 2, 2]            524,288
|    └─BatchNorm2d: 2-13                      [-1, 64, 2, 2]            128
|    └─LeakyReLU: 2-14                        [-1, 64, 2, 2]            --
|    └─Flatten: 2-15                          [-1, 256]                 --
├─Embedding: 1-2                              [-1, 32]                  64
├─Sequential: 1-3                             [-1, 1]                   --
|    └─Linear: 2-16                           [-1, 100]                 28,900
|    └─LeakyReLU: 2-17                        [-1, 100]                 --
|    └─Linear: 2-18                           [-1, 1]                   101
|    └─Sigmoid: 2-19                          [-1, 1]                   --
==========================================================================================
Total params: 3,310,857
Trainable params: 3,310,857
Non-trainable params: 0
Total mult-adds (M): 109.25
```

Note that `self.model2` takes an input of 288 values as the output of `self.model` has 256 values per data point, which is then concatenated with the 32 embedding values of the input class label, resulting in 256 + 32 = 288 input values to `self.model2`.

5. Define the `Generator` network class:

   - Define the `__init__` method:

```
class Generator(nn.Module):
    def __init__(self, emb_size=32):
        super(Generator,self).__init__()
        self.emb_size = emb_size
        self.label_embeddings = nn.Embedding(2, self.emb_size)
```

Note that in the preceding code, we are using `nn.Embedding` to convert the 2D input (which is of classes) to a 32-dimensional vector (`self.emb_size`):

```
self.model = nn.Sequential(
    nn.ConvTranspose2d(100+self.emb_size,\
                       64*8,4,1,0,bias=False),
    nn.BatchNorm2d(64*8),
    nn.ReLU(True),
    nn.ConvTranspose2d(64*8,64*4,4,2,1,bias=False),
    nn.BatchNorm2d(64*4),
    nn.ReLU(True),
    nn.ConvTranspose2d(64*4,64*2,4,2,1,bias=False),
    nn.BatchNorm2d(64*2),
    nn.ReLU(True),
    nn.ConvTranspose2d(64*2,64,4,2,1,bias=False),
    nn.BatchNorm2d(64),
    nn.ReLU(True),
    nn.ConvTranspose2d(64,3,4,2,1,bias=False),
    nn.Tanh()
)
```

Note that in the preceding code, we have leveraged `nn.ConvTranspose2d` to upscale toward fetching an image as output.

- Apply weight initialization:

```
self.apply(weights_init)
```

- Define the `forward` method that takes the noise values (`input_noise`) and input label (`labels`) as input and generates the output of the image:

```
def forward(self,input_noise,labels):
    label_embeddings = self.label_embeddings(labels) \
                        .view(len(labels), \
                              self.emb_size,1, 1)
    input = torch.cat([input_noise, label_embeddings], 1)
    return self.model(input)
```

- Obtain a summary of the defined `generator` function:

```
generator = Generator().to(device)
summary(generator,torch.zeros(32,100,1,1).to(device), \
        torch.zeros(32).long().to(device));
```

The preceding code generates the following output:

```
================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
================================================================================
├─Embedding: 1-1                         [-1, 32]                  64
├─Sequential: 1-2                        [-1, 3, 64, 64]           --
|    └─ConvTranspose2d: 2-1              [-1, 512, 4, 4]           1,081,344
|    └─BatchNorm2d: 2-2                  [-1, 512, 4, 4]           1,024
|    └─ReLU: 2-3                         [-1, 512, 4, 4]           --
|    └─ConvTranspose2d: 2-4              [-1, 256, 8, 8]           2,097,152
|    └─BatchNorm2d: 2-5                  [-1, 256, 8, 8]           512
|    └─ReLU: 2-6                         [-1, 256, 8, 8]           --
|    └─ConvTranspose2d: 2-7              [-1, 128, 16, 16]         524,288
|    └─BatchNorm2d: 2-8                  [-1, 128, 16, 16]         256
|    └─ReLU: 2-9                         [-1, 128, 16, 16]         --
|    └─ConvTranspose2d: 2-10             [-1, 64, 32, 32]          131,072
|    └─BatchNorm2d: 2-11                 [-1, 64, 32, 32]          128
|    └─ReLU: 2-12                        [-1, 64, 32, 32]          --
|    └─ConvTranspose2d: 2-13             [-1, 3, 64, 64]           3,072
|    └─Tanh: 2-14                        [-1, 3, 64, 64]           --
================================================================================
Total params: 3,838,912
Trainable params: 3,838,912
Non-trainable params: 0
Total mult-adds (M): 436.38
```

6. Define a function (`noise`) to generate random noise with 100 values and register it to the device:

```
def noise(size):
    n = torch.randn(size, 100, 1, 1, device=device)
    return n.to(device)
```

7. Define the function to train the discriminator – `discriminator_train_step`:

   - The discriminator takes four inputs—real images (`real_data`), real labels (`real_labels`), fake images (`fake_data`), and fake labels (`fake_labels`):

```
def discriminator_train_step(real_data, real_labels, \
                             fake_data, fake_labels):
    d_optimizer.zero_grad()
```

Here, we are resetting the gradient corresponding to the discriminator:

- Calculate the loss value corresponding to predictions on the real data (`prediction_real`). The loss value output when `real_data` and `real_labels` are passed through the `discriminator` network is compared with the expected value of `(torch.ones(len(real_data),1).to(device))` to obtain `error_real` before performing backpropagation:

```
prediction_real = discriminator(real_data, real_labels)
error_real = loss(prediction_real, \
                  torch.ones(len(real_data),1).to(device))
error_real.backward()
```

- Calculate the loss value corresponding to predictions on the fake data (`prediction_fake`). The loss value output when `fake_data` and `fake_labels` are passed through the `discriminator` network is compared with the expected value of `(torch.zeros(len(fake_data),1).to(device))` to obtain `error_fake` before performing backpropagation:

```
prediction_fake = discriminator(fake_data, fake_labels)
error_fake = loss(prediction_fake, \
                torch.zeros(len(fake_data),1).to(device))
error_fake.backward()
```

- Update weights and return the loss values:

```
d_optimizer.step()
return error_real + error_fake
```

8. Define the training steps for the generator where we pass the fake images (`fake_data`) along with the fake labels (`fake_labels`) as input:

```
def generator_train_step(fake_data, fake_labels):
    g_optimizer.zero_grad()
    prediction = discriminator(fake_data, fake_labels)
    error = loss(prediction, \
                 torch.ones(len(fake_data), 1).to(device))
    error.backward()
    g_optimizer.step()
    return error
```

Note that the `generator_train_step` function is similar to `discriminator_train_step`, with the exception that this has an expectation of `torch.ones(len(fake_data),1).to(device))` as output in place of zeros given that we are training the generator.

9. Define the `generator` and `discriminator` model objects, the loss optimizers, and the `loss` function:

```
discriminator = Discriminator().to(device)
generator = Generator().to(device)
loss = nn.BCELoss()
d_optimizer = optim.Adam(discriminator.parameters(), \
                         lr=0.0002, betas=(0.5, 0.999))
g_optimizer = optim.Adam(generator.parameters(), \
                         lr=0.0002, betas=(0.5, 0.999))
fixed_noise = torch.randn(64, 100, 1, 1, device=device)
fixed_fake_labels = torch.LongTensor([0]* \
                                  (len(fixed_noise)//2) \
                    + [1]*(len(fixed_noise)//2)).to(device)
loss = nn.BCELoss()
n_epochs = 25
img_list = []
```

In the preceding code, while defining `fixed_fake_labels`, we are specifying that half of the images correspond to one class (class 0) and the rest to another class (class 1). Additionally, we are defining `fixed_noise`, which will be used to generate images from random noise.

10. Train the model over increasing epochs (`n_epochs`):

 - Specify the length of `dataloader`:

```
log = Report(n_epochs)
for epoch in range(n_epochs):
    N = len(dataloader)
```

 - Loop through the batch of images along with their labels:

```
    for bx, (images, labels) in enumerate(dataloader):
```

 - Specify `real_data` and `real_labels`:

```
        real_data, real_labels = images.to(device), \
                                 labels.to(device)
```

- Initialize `fake_data` and `fake_labels`:

```
fake_labels = torch.LongTensor(np.random.randint(0, \
                        2,len(real_data))).to(device)
fake_data=generator(noise(len(real_data)),fake_labels)
fake_data = fake_data.detach()
```

- Train the discriminator using the `discriminator_train_step` function defined in *step 7* to calculate discriminator loss (`d_loss`):

```
d_loss = discriminator_train_step(real_data, \
                real_labels, fake_data, fake_labels)
```

- Regenerate fake images (`fake_data`) and fake labels (`fake_labels`) and train the generator using the `generator_train_step` function defined in *step 8* to calculate the generator loss (`g_loss`):

```
fake_labels = torch.LongTensor(np.random.randint(0, \
                        2,len(real_data))).to(device)
fake_data = generator(noise(len(real_data)), \
                      fake_labels).to(device)
g_loss = generator_train_step(fake_data, fake_labels)
```

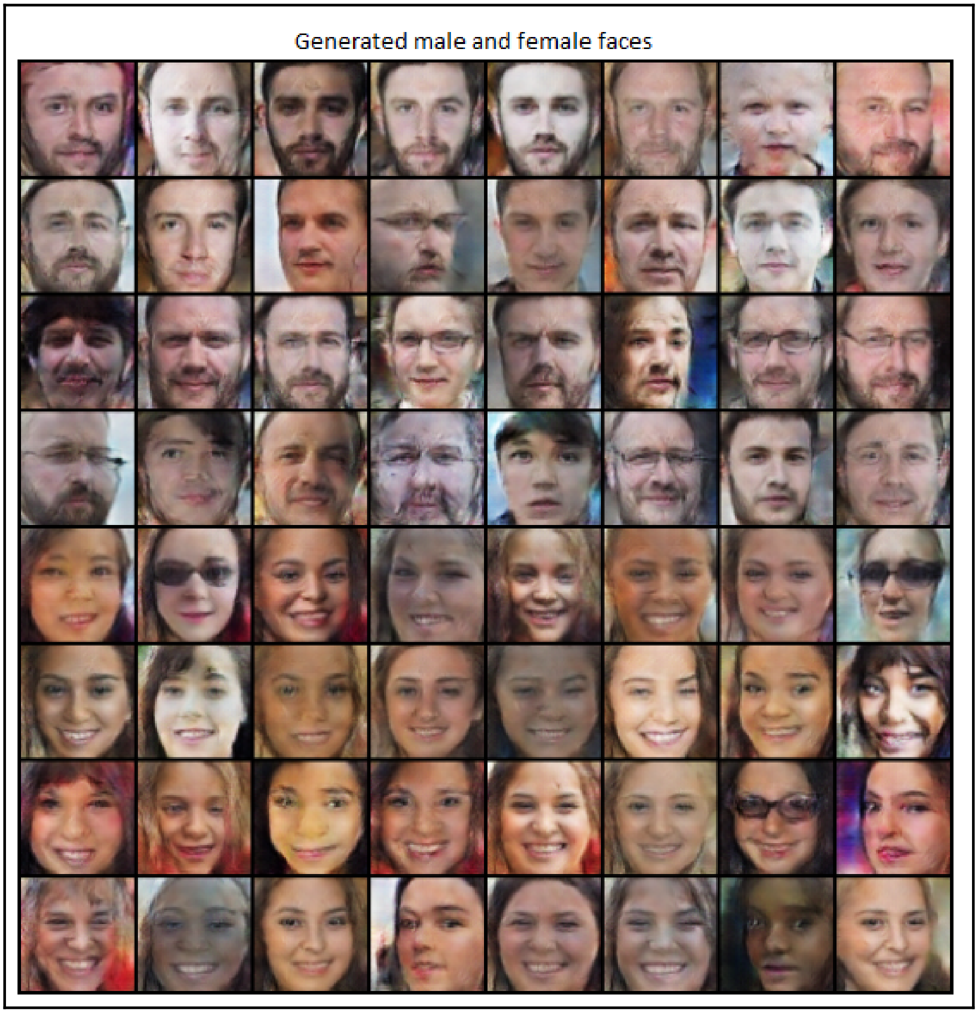
- Log the metrics as follows:

```
        pos = epoch + (1+bx)/N
        log.record(pos, d_loss=d_loss.detach(), \
                   g_loss=g_loss.detach(), end='\r')
    log.report_avgs(epoch+1)
```

Once we train the model, generate the male and female images:

```
with torch.no_grad():
    fake = generator(fixed_noise, \
                     fixed_fake_labels).detach().cpu()
    imgs = vutils.make_grid(fake, padding=2, \
                        normalize=True).permute(1,2,0)
    img_list.append(imgs)
    show(imgs, sz=10)
```

In the preceding code, we are passing the noise (`fixed_noise`) and labels (`fixed_fake_labels`) to the generator to fetch the `fake` images, which are as follows at the end of 25 epochs of training the models:



From the preceding image, we can see that the first 32 images correspond to male images, while the next 32 correspond to female images, which substantiates the fact that the conditional GANs performed as expected.

Summary

In this chapter, we have learned about leveraging two different neural networks to generate new images of handwritten digits using GANs. Next, we generated realistic faces using DCGANs. Finally, we learned about conditional GANs, which help us in generating images of a certain class. Having generated images using different techniques, we could still see that the generated images were not sufficiently realistic. Furthermore, while we generated images by specifying the class of images we want to generate in conditional GANs, we are still not in a position to perform image translation, where we ask to replace one object in the image with another one, with everything else left as is. In addition, we are yet to have an image generation mechanism where the number of classes (styles) to generate is more unsupervised.

In the next chapter, we will learn about generating images that are more realistic using some of the latest variants of GANs. In addition, we will learn about generating images of different styles in a more unsupervised manner.

Questions

1. What happens if the learning rate of generator and discriminator models is high?
2. In a scenario where the generator and discriminator are very well trained, what is the probability of a given image being real?
3. Why do we use `convtranspose2d` in generating images?
4. Why do we have embeddings with a high embedding size compared with the number of classes in conditional GANs?
5. How can we generate images of men who have a beard?
6. Why do we have Tanh activation in the last layer in the generator and not ReLU or Sigmoid?
7. Why did we get realistic images even though we did not denormalize the generated data?

8. What happens if we do not crop faces corresponding to images prior to training the GAN?
9. Why do the weights of the discriminator not get updated when training the generator (as the `generator_train_step` function involves the discriminator network)?
10. Why do we fetch losses on both real and fake images while training the discriminator, but only the loss on fake images while training the generator?

13 Advanced GANs to Manipulate Images

In the previous chapter, we learned about leveraging **Generative Adversarial Networks** (**GANs**) to generate realistic images. In this chapter, we will learn about leveraging GANs to manipulate images. We will learn about two variations of generating images using GANs – supervised and unsupervised methods. In the supervised method, we will provide the input and output pair combinations to generate images based on an input image, which we will learn about in the Pix2Pix GAN. In the unsupervised method, we will specify the input and output, however, we will not provide one-to-one correspondence between the input and output, but expect the GAN to learn the structure of the two classes, and convert an image from one class to another, which we will learn about in CycleGAN.

Another class of unsupervised image manipulation involves generating images from a latent space of random vectors and seeing how images change as the latent vector values change, which we will learn about in the *Leveraging StyleGAN on custom images* section. Finally, we will learn about leveraging a pre-trained GAN – SRGAN, which helps in turning a low-resolution image into an image with high resolution.

Specifically, we will learn about the following topics:

- Leveraging the Pix2Pix GAN
- Leveraging CycleGAN
- Leveraging StyleGAN on custom images
- Super-resolution GAN

Leveraging the Pix2Pix GAN

Imagine a scenario where we have pairs of images that are related to each other (for example, an image of edges of an object as input and an actual image of the object as output). The challenge given is that we want to generate an image given the input image of the edges of an object. In a traditional setting, this would have been a simple mapping of input to output and hence a supervised learning problem. However, imagine that you are working with a creative team that is trying to come up with a fresh look for products. In such a scenario, supervised learning does not help as much – as it learns only from history. A GAN comes in handy here because it will ensure that the generated image looks realistic enough and leaves room for experimentation (as we are interested in checking whether the generated image seems like one of the classes of interest or not).

In this section, we will learn about the architecture to generate the image of a shoe from a hand-drawn doodle (contours) of a shoe. The strategy that we will adopt to generate a realistic image from the doodle is as follows:

1. Fetch a lot of actual images and create corresponding contours using standard cv2 edge detection techniques.
2. Sample colors from patches of the original image so that the generator knows the colors to generate.
3. Build a UNet architecture that takes the contours with sample patch colors as input and predicts the corresponding image – this is our generator.
4. Build a discriminator architecture that takes an image and predicts whether it is real or fake.
5. Train the generator and discriminator together to a point where the generator can generate images that fool the discriminator.

Let's code the strategy:

> The following code is available as `Pix2Pix_GAN.ipynb` in the `Chapter13` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from and is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components from text.

1. Import the dataset and install the relevant packages:

```
try:
    !wget https://bit.ly/3kiuN93
    !mv 3kiuN93 ShoeV2.zip
    !unzip ShoeV2.zip
    !unzip ShoeV2_F/ShoeV2_photo.zip
except:
    !wget
https://www.dropbox.com/s/g6b6gtvmdu0h77x/ShoeV2_photo.zip
!pip install torch_snippets
from torch_snippets import *
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

The preceding code downloads images of shoes. A sample of downloaded images is as follows:

For our problem, we want to draw the shoe given a contour (edge) and sample patch colors of the shoe. In the next step, we will fetch the edge given an image of a shoe. This way, we can train a model that reconstructs an image of a shoe given a contour and sample patch colors of the shoe.

2. Define a function to fetch edges from the downloaded images:

```
def detect_edges(img):
    img_gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    img_gray = cv2.bilateralFilter(img_gray, 5, 50, 50)
```

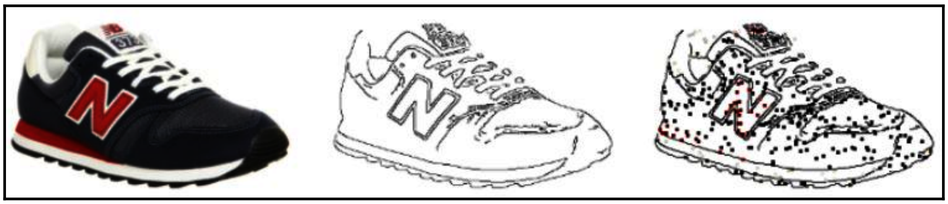
```
        img_gray_edges = cv2.Canny(img_gray, 45, 100)
        # invert black/white
        img_gray_edges = cv2.bitwise_not(img_gray_edges)
        img_edges=cv2.cvtColor(img_gray_edges,cv2.COLOR_GRAY2RGB)
        return img_edges
```

In the preceding code, we are leveraging the various methods available in the OpenCV package to fetch edges within an image (there are more details on how the OpenCV methods work in `Chapter 18`, *Using OpenCV Utilities for Image Analysis*).

3. Define the image transformation pipeline (`preprocess` and `normalize`):

```
IMAGE_SIZE = 256
preprocess = T.Compose([
                    T.Lambda(lambda x: torch.Tensor(x.copy())\
                             .permute(2, 0, 1).to(device))
                ])
normalize = lambda x: (x - 127.5)/127.5
```

4. Define the dataset class (`ShoesData`). This dataset class returns the original image and the image with edges. One additional detail we will pass to the network is the patches of color that are present in randomly chosen regions. This way, we are enabling the user to take a hand-drawn contour image, sprinkle the required colors at different parts of the image, and generate a new image. A sample input (the third image) and output (the first image) are shown here (best viewed in color):



However, the input image we have in step 1 is just of the shoe (the first image) – which we will use to extract the edges of the shoe (the second image). Further, we will sprinkle color in the next step to fetch the preceding image's input (the third image) – output (the first image) combination.

In the following code, we will build the class that takes the contour images, sprinkles colors, and returns the pair of color-sprinkled images and the original shoe image (the image that generated contours):

- Define the `ShoesData` class, the `__init__` method, and the `__len__` method:

```
class ShoesData(Dataset):
    def __init__(self, items):
        self.items = items
    def __len__(self): return len(self.items)
```

- Define the `__getitem__` method. In this method, we will process the input image to fetch an image with edges and then sprinkle the image with the colors present in the original image. Here, we are fetching the edges of a given image:

```
    def __getitem__(self, ix):
        f = self.items[ix]
        try: im = read(f, 1)
        except:
            blank = preprocess(Blank(IMAGE_SIZE, \
                                     IMAGE_SIZE, 3))
            return blank, blank
        edges = detect_edges(im)
```

- Once we fetch the edges in the image, resize and normalize the image:

```
        im, edges = resize(im, IMAGE_SIZE), \
                    resize(edges, IMAGE_SIZE)
        im, edges = normalize(im), normalize(edges)
```

- Sprinkle color on the `edges` image and `preprocess` the original and `edges` images:

```
        self._draw_color_circles_on_src_img(edges, im)
        im, edges = preprocess(im), preprocess(edges)
        return edges, im
```

- Define the functions to sprinkle color:

```
    def _draw_color_circles_on_src_img(self, img_src, \
                                       img_target):
        non_white_coords = self._get_non_white_coordinates\
                                    (img_target)
        for center_y, center_x in non_white_coords:
            self._draw_color_circle_on_src_img(img_src, \
```

```
                        img_target, center_y, center_x)

    def _get_non_white_coordinates(self, img):
        non_white_mask = np.sum(img, axis=-1) < 2.75
        non_white_y, non_white_x = np.nonzero(non_white_mask)
        # randomly sample non-white coordinates
        n_non_white = len(non_white_y)
        n_color_points = min(n_non_white, 300)
        idxs = np.random.choice(n_non_white, n_color_points, \
                                replace=False)
        non_white_coords = list(zip(non_white_y[idxs], \
                                    non_white_x[idxs]))
        return non_white_coords

    def _draw_color_circle_on_src_img(self, img_src, \
                            img_target, center_y, center_x):
        assert img_src.shape == img_target.shape
        y0, y1, x0, x1 = self._get_color_point_bbox_coords(\
                                        center_y, center_x)
        color = np.mean(img_target[y0:y1, x0:x1],axis=(0, 1))
        img_src[y0:y1, x0:x1] = color

    def _get_color_point_bbox_coords(self, center_y,center_x):
        radius = 2
        y0 = max(0, center_y-radius+1)
        y1 = min(IMAGE_SIZE, center_y+radius)
        x0 = max(0, center_x-radius+1)
        x1 = min(IMAGE_SIZE, center_x+radius)
        return y0, y1, x0, x1

    def choose(self): return self[randint(len(self))]
```

5. Define the training and validation data's corresponding datasets and dataloaders:

```
from sklearn.model_selection import train_test_split
train_items, val_items = train_test_split(\
                        Glob('ShoeV2_photo/*.png'), \
                        test_size=0.2, random_state=2)
trn_ds, val_ds = ShoesData(train_items), ShoesData(val_items)

trn_dl = DataLoader(trn_ds, batch_size=32, shuffle=True)
val_dl = DataLoader(val_ds, batch_size=32, shuffle=True)
```

6. Define generator and discriminator architectures, which leverage the weight initialization (`weights_init_normal`), `UNetDown`, and `UNetUp` architectures just as we did in `Chapter 9`, *Image Segmentation* and `Chapter 10`, *Applications of Object Detection and Segmentation*, to define the `GeneratorUNet` and `Discriminator` architectures.

- Initialize weights so that they follow a normal distribution:

```
def weights_init_normal(m):
    classname = m.__class__.__name__
    if classname.find("Conv") != -1:
        torch.nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find("BatchNorm2d") != -1:
        torch.nn.init.normal_(m.weight.data, 1.0, 0.02)
        torch.nn.init.constant_(m.bias.data, 0.0)
```

- Define the `UNetwDown` and `UNetUp` classes:

```
class UNetDown(nn.Module):
    def __init__(self, in_size, out_size, normalize=True, \
                 dropout=0.0):
        super(UNetDown, self).__init__()
        layers = [nn.Conv2d(in_size, out_size, 4, 2, 1, \
                            bias=False)]
        if normalize:
            layers.append(nn.InstanceNorm2d(out_size))
        layers.append(nn.LeakyReLU(0.2))
        if dropout:
            layers.append(nn.Dropout(dropout))
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        return self.model(x)

class UNetUp(nn.Module):
    def __init__(self, in_size, out_size, dropout=0.0):
        super(UNetUp, self).__init__()
        layers = [
            nn.ConvTranspose2d(in_size, out_size, 4, 2, 1, \
                               bias=False),
            nn.InstanceNorm2d(out_size),
            nn.ReLU(inplace=True),
        ]
        if dropout:
            layers.append(nn.Dropout(dropout))

        self.model = nn.Sequential(*layers)
```

```
    def forward(self, x, skip_input):
        x = self.model(x)
        x = torch.cat((x, skip_input), 1)

        return x
```

- Define the `GeneratorUNet` class:

```
class GeneratorUNet(nn.Module):
    def __init__(self, in_channels=3, out_channels=3):
        super(GeneratorUNet, self).__init__()

        self.down1 = UNetDown(in_channels,64,normalize=False)
        self.down2 = UNetDown(64, 128)
        self.down3 = UNetDown(128, 256)
        self.down4 = UNetDown(256, 512, dropout=0.5)
        self.down5 = UNetDown(512, 512, dropout=0.5)
        self.down6 = UNetDown(512, 512, dropout=0.5)
        self.down7 = UNetDown(512, 512, dropout=0.5)
        self.down8 = UNetDown(512, 512, normalize=False, \
                              dropout=0.5)

        self.up1 = UNetUp(512, 512, dropout=0.5)
        self.up2 = UNetUp(1024, 512, dropout=0.5)
        self.up3 = UNetUp(1024, 512, dropout=0.5)
        self.up4 = UNetUp(1024, 512, dropout=0.5)
        self.up5 = UNetUp(1024, 256)
        self.up6 = UNetUp(512, 128)
        self.up7 = UNetUp(256, 64)

        self.final = nn.Sequential(
            nn.Upsample(scale_factor=2),
            nn.ZeroPad2d((1, 0, 1, 0)),
            nn.Conv2d(128, out_channels, 4, padding=1),
            nn.Tanh(),
        )

    def forward(self, x):
        d1 = self.down1(x)
        d2 = self.down2(d1)
        d3 = self.down3(d2)
        d4 = self.down4(d3)
        d5 = self.down5(d4)
        d6 = self.down6(d5)
        d7 = self.down7(d6)
        d8 = self.down8(d7)
        u1 = self.up1(d8, d7)
        u2 = self.up2(u1, d6)
```

```
        u3 = self.up3(u2, d5)
        u4 = self.up4(u3, d4)
        u5 = self.up5(u4, d3)
        u6 = self.up6(u5, d2)
        u7 = self.up7(u6, d1)
        return self.final(u7)
```

- Define the `Discriminator` class:

```
class Discriminator(nn.Module):
    def __init__(self, in_channels=3):
        super(Discriminator, self).__init__()

        def discriminator_block(in_filters, out_filters, \
                                normalization=True):
            """Returns downsampling layers of each
            discriminator block"""
            layers = [nn.Conv2d(in_filters, out_filters, \
                                4, stride=2, padding=1)]
            if normalization:
                layers.append(nn.InstanceNorm2d(out_filters))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers

        self.model = nn.Sequential(
            *discriminator_block(in_channels * 2, 64, \
                                 normalization=False),
            *discriminator_block(64, 128),
            *discriminator_block(128, 256),
            *discriminator_block(256, 512),
            nn.ZeroPad2d((1, 0, 1, 0)),
            nn.Conv2d(512, 1, 4, padding=1, bias=False)
        )

    def forward(self, img_A, img_B):
        img_input = torch.cat((img_A, img_B), 1)
        return self.model(img_input)
```

7. Define the `generator` and `discriminator` model objects and fetch summaries:

```
generator = GeneratorUNet().to(device)
discriminator = Discriminator().to(device)
!pip install torch_summary
from torchsummary import summary
print(summary(generator, torch.zeros(3, 3, IMAGE_SIZE, \
                                IMAGE_SIZE).to(device)))
print(summary(discriminator, torch.zeros(3, 3, IMAGE_SIZE, \
```

```
IMAGE_SIZE).to(device), torch.zeros(3, 3, \
IMAGE_SIZE, IMAGE_SIZE).to(device)))
```

The generator architecture summary is as follows:

Layer (type)	Output Shape	Param #	Tr. Param #
UNetDown-1	[3, 64, 128, 128]	3,072	3,072
UNetDown-2	[3, 128, 64, 64]	131,072	131,072
UNetDown-3	[3, 256, 32, 32]	524,288	524,288
UNetDown-4	[3, 512, 16, 16]	2,097,152	2,097,152
UNetDown-5	[3, 512, 8, 8]	4,194,304	4,194,304
UNetDown-6	[3, 512, 4, 4]	4,194,304	4,194,304
UNetDown-7	[3, 512, 2, 2]	4,194,304	4,194,304
UNetDown-8	[3, 512, 1, 1]	4,194,304	4,194,304
UNetUp-9	[3, 1024, 2, 2]	4,194,304	4,194,304
UNetUp-10	[3, 1024, 4, 4]	8,388,608	8,388,608
UNetUp-11	[3, 1024, 8, 8]	8,388,608	8,388,608
UNetUp-12	[3, 1024, 16, 16]	8,388,608	8,388,608
UNetUp-13	[3, 512, 32, 32]	4,194,304	4,194,304
UNetUp-14	[3, 256, 64, 64]	1,048,576	1,048,576
UNetUp-15	[3, 128, 128, 128]	262,144	262,144
Upsample-16	[3, 128, 256, 256]	0	0
ZeroPad2d-17	[3, 128, 257, 257]	0	0
Conv2d-18	[3, 3, 256, 256]	6,147	6,147
Tanh-19	[3, 3, 256, 256]	0	0

Total params: 54,404,099
Trainable params: 54,404,099
Non-trainable params: 0

The discriminator architecture summary is as follows:

Layer (type)	Output Shape	Param #	Tr. Param #
Conv2d-1	[3, 64, 128, 128]	6,208	6,208
LeakyReLU-2	[3, 64, 128, 128]	0	0
Conv2d-3	[3, 128, 64, 64]	131,200	131,200
InstanceNorm2d-4	[3, 128, 64, 64]	0	0
LeakyReLU-5	[3, 128, 64, 64]	0	0
Conv2d-6	[3, 256, 32, 32]	524,544	524,544
InstanceNorm2d-7	[3, 256, 32, 32]	0	0
LeakyReLU-8	[3, 256, 32, 32]	0	0
Conv2d-9	[3, 512, 16, 16]	2,097,664	2,097,664
InstanceNorm2d-10	[3, 512, 16, 16]	0	0
LeakyReLU-11	[3, 512, 16, 16]	0	0
ZeroPad2d-12	[3, 512, 17, 17]	0	0
Conv2d-13	[3, 1, 16, 16]	8,192	8,192

Total params: 2,767,808
Trainable params: 2,767,808
Non-trainable params: 0

8. Define the function to train the discriminator (`discriminator_train_step`):

- The discriminator function takes the source image (`real_src`), real target (`real_trg`), and fake target (`fake_trg`) as input:

```
def discriminator_train_step(real_src, real_trg, fake_trg):
    d_optimizer.zero_grad()
```

- Calculate the loss (`error_real`) by comparing the real target (`real_trg`) and predicted values (`real_src`) of the target, where the expectation is that the discriminator will predict the images as real (indicated by `torch.ones`), and then perform back-propagation:

```
    prediction_real = discriminator(real_trg, real_src)
    error_real = criterion_GAN(prediction_real, \
                     torch.ones(len(real_src), 1, 16, 16)\
                                  .to(device))
    error_real.backward()
```

- Calculate the discriminator loss (`error_fake`) corresponding to fake images (`fake_trg`) where the expectation is that the discriminator classifies the fake targets as fake images (indicated by `torch.zeros`) and then perform back-propagation:

```
    prediction_fake = discriminator( real_src, \
                                     fake_trg.detach())
    error_fake = criterion_GAN(prediction_fake, \
                                  torch.zeros(len(real_src), 1, \
                                              16, 16).to(device))
    error_fake.backward()
```

- Perform the optimizer step and return the overall error and loss values on predicted real and fake targets:

```
    d_optimizer.step()
    return error_real + error_fake
```

9. Define the function to train the generator (`generator_train_step`) where it takes a fake target (`fake_trg`) and trains it towards a scenario where it has a low chance of getting identified as fake when passed through the discriminator:

```
def generator_train_step(real_src, fake_trg):
    g_optimizer.zero_grad()
    prediction = discriminator(fake_trg, real_src)
```

```
        loss_GAN = criterion_GAN(prediction, torch.ones(\
                                len(real_src), 1, 16, 16)\
                                 .to(device))
        loss_pixel = criterion_pixelwise(fake_trg, real_trg)
        loss_G = loss_GAN + lambda_pixel * loss_pixel

        loss_G.backward()
        g_optimizer.step()
        return loss_G
```

Note that, in the preceding code, in addition to generator loss, we are also fetching the pixel loss (`loss_pixel`) corresponding to the difference between the generated and the real image of a given contour:

- Define a function to fetch a sample of predictions:

```
denorm = T.Normalize((-1, -1, -1), (2, 2, 2))
def sample_prediction():
    """Saves a generated sample from the validation set"""
    data = next(iter(val_dl))
    real_src, real_trg = data
    fake_trg = generator(real_src)
    img_sample = torch.cat([denorm(real_src[0]), \
                            denorm(fake_trg[0]), \
                            denorm(real_trg[0])], -1)
    img_sample = img_sample.detach().cpu()\
                           .permute(1,2,0).numpy()
    show(img_sample, title='Source::Generated::GroundTruth', \
         sz=12)
```

10. Apply weight initialization (`weights_init_normal`) to the generator and discriminator model objects:

```
generator.apply(weights_init_normal)
discriminator.apply(weights_init_normal)
```

11. Specify the loss criterion and optimization methods (`criterion_GAN` and `criterion_pixelwise`):

```
criterion_GAN = torch.nn.MSELoss()
criterion_pixelwise = torch.nn.L1Loss()

lambda_pixel = 100
g_optimizer = torch.optim.Adam(generator.parameters(), \
                               lr=0.0002, betas=(0.5, 0.999))
d_optimizer = torch.optim.Adam(discriminator.parameters(), \
                               lr=0.0002, betas=(0.5, 0.999))
```

12. Train the model over 100 epochs:

```
epochs = 100
log = Report(epochs)
for epoch in range(epochs):
    N = len(trn_dl)
    for bx, batch in enumerate(trn_dl):
        real_src, real_trg = batch
        fake_trg = generator(real_src)
        errD = discriminator_train_step(real_src, real_trg, \
                                        fake_trg)
        errG = generator_train_step(real_src, fake_trg)
        log.record(pos=epoch+(1+bx)/N, errD=errD.item(), \
                   errG=errG.item(), end='\r')
    [sample_prediction() for _ in range(2)]
```

13. Generate on a sample hand-drawn contour:

```
[sample_prediction() for _ in range(2)]
```

The preceding code generates the following output:

Note that in the preceding output, we have generated images that have similar colors as those of the original image.

In this section, we have learned about leveraging the contours of an image to generate an image. However, this required us to provide the input and output as pairs, which can be a tedious process sometimes. In the next section, we will learn about unpaired image translation where the network would figure the translation without us specifying the input and output mappings of images.

Leveraging CycleGAN

Imagine a scenario where we ask you to perform image translation from one class to another, but not give the input and the corresponding output images to train the model. However, we give you the images of both classes in two distinct folders. CycleGAN comes in handy in such a scenario.

In this section, we will learn how to train CycleGAN to convert the image of an apple into the image of an orange and vice versa. The **Cycle** in CycleGAN refers to the fact that we are translating (converting) an image from one class to another and back to the original class.

At a high level, we will have three separate loss values in this architecture (more detail is provided here):

- **Discriminator loss**: This ensures that the object class is modified while training the model (as seen in the previous section).
- **Cycle loss**: The loss of recycling an image from the generated image to the original to ensure that the surrounding pixels are not changed.
- **Identity loss**: The loss when an image of one class is passed through a generator that is expected to convert an image of another class into the class of the input image.

Here, we will understand at a high level the steps of building CycleGAN:

1. Import and preprocess the dataset
2. Build the generator and discriminator network UNet architectures
3. Define two generators:
    - **G_AB**: Generator that converts an image of class A to an image of class B
    - **G_BA**: Generator that converts an image of class B to an image of class A

4. Define **Identity loss**:

    - If you were to send an orange image to an orange-generator, ideally if the generator has understood everything about oranges, it should not change the image and should "generate" the exact same image. We thus create an identity using this knowledge.
    - Identity loss should be minimal when an image of class A (real_A) is passed through G_BA and compared with real_A.
    - Identity loss should be minimal when an image of class B (real_B) is passed through G_AB and compared with real_B.

5. Define **GAN loss:**

    - Discriminator and generator loss for real_A and fake_A (fake_A is obtained when real_B image is passed through G_BA)
    - Discriminator and generator loss for real_B and fake_B (fake_B is obtained when the real_A image is passed through G_AB)

6. Define re-**cycle loss:**

    - Consider a scenario where an image of an apple is to be transformed by an orange-generator to generate a fake orange, and the fake orange is to be transformed back to an apple by the apple-generator.
    - fake_B, which is the output when real_A is passed through G_AB, should regenerate real_A when fake_B is passed through G_BA.
    - fake_A, which is the output when real_B is passed through G_BA, should regenerate real_B when fake_A is passed through G_AB.

7. Optimize for the weighted loss of the three losses.

Now that we understand the steps, let's code them to convert apples to oranges and vice versa, as follows:

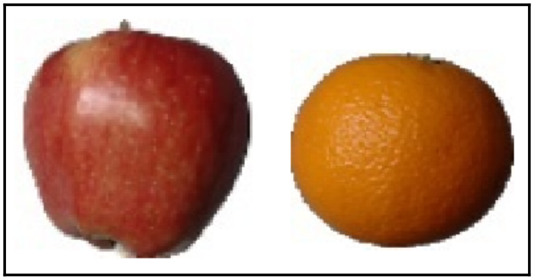
> The following code is available as `CycleGAN.ipynb` in the `Chapter13` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from and is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components in text.

1. Import the relevant datasets and packages:

- Download and extract the datasets:

```
!wget
https://www.dropbox.com/s/2xltmolfbfharri/apples_oranges.zip
!unzip apples_oranges.zip
```

A sample of the images we will be working on:



Note that there is no one-to-one correspondence between the apple and orange images (unlike the contour to shoe generation use case that we learned about in the *Leveraging the Pix2Pix GAN* section).

- Import the required packages:

```
!pip install torch_snippets torch_summary
import itertools
from PIL import Image
from torch_snippets import *
from torchvision import transforms
from torchvision.utils import make_grid
from torchsummary import summary
```

2. Define the image transformation pipeline (`transform`):

```
IMAGE_SIZE = 256
device = 'cuda' if torch.cuda.is_available() else 'cpu'
transform = transforms.Compose([
    transforms.Resize(int(IMAGE_SIZE*1.33)),
    transforms.RandomCrop((IMAGE_SIZE,IMAGE_SIZE)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])
```

3. Define the dataset class (`CycleGANDataset`), which takes the `apple` and `orange` folders (which are obtained after unzipping the downloaded dataset) as input and provides a batch of apple and orange images:

```
class CycleGANDataset(Dataset):
    def __init__(self, apples, oranges):
        self.apples = Glob(apples)
        self.oranges = Glob(oranges)

    def __getitem__(self, ix):
        apple = self.apples[ix % len(self.apples)]
        orange = choose(self.oranges)
        apple = Image.open(apple).convert('RGB')
        orange = Image.open(orange).convert('RGB')
        return apple, orange

    def __len__(self): return max(len(self.apples), \
                                  len(self.oranges))
    def choose(self): return self[randint(len(self))]

    def collate_fn(self, batch):
        srcs, trgs = list(zip(*batch))
        srcs=torch.cat([transform(img)[None] for img in srcs]\
                        , 0).to(device).float()
        trgs=torch.cat([transform(img)[None] for img in trgs]\
                        , 0).to(device).float()
        return srcs.to(device), trgs.to(device)
```

4. Define the training and validation datasets and data loaders:

```
trn_ds = CycleGANDataset('apples_train', 'oranges_train')
val_ds = CycleGANDataset('apples_test', 'oranges_test')

trn_dl = DataLoader(trn_ds, batch_size=1, shuffle=True, \
                    collate_fn=trn_ds.collate_fn)
val_dl = DataLoader(val_ds, batch_size=5, shuffle=True, \
                    collate_fn=val_ds.collate_fn)
```

5. Define the weight initialization method of the network (`weights_init_normal`) as defined in previous sections:

```
def weights_init_normal(m):
    classname = m.__class__.__name__
    if classname.find("Conv") != -1:
        torch.nn.init.normal_(m.weight.data, 0.0, 0.02)
        if hasattr(m, "bias") and m.bias is not None:
            torch.nn.init.constant_(m.bias.data, 0.0)
    elif classname.find("BatchNorm2d") != -1:
        torch.nn.init.normal_(m.weight.data, 1.0, 0.02)
        torch.nn.init.constant_(m.bias.data, 0.0)
```

6. Define the residual block network (`ResidualBlock`), as in this instance, we will leverage the ResNet:

```
class ResidualBlock(nn.Module):
    def __init__(self, in_features):
        super(ResidualBlock, self).__init__()

        self.block = nn.Sequential(
            nn.ReflectionPad2d(1),
            nn.Conv2d(in_features, in_features, 3),
            nn.InstanceNorm2d(in_features),
            nn.ReLU(inplace=True),
            nn.ReflectionPad2d(1),
            nn.Conv2d(in_features, in_features, 3),
            nn.InstanceNorm2d(in_features),
        )

    def forward(self, x):
        return x + self.block(x)
```

7. Define the generator network (`GeneratorResNet`):

```
class GeneratorResNet(nn.Module):
    def __init__(self, num_residual_blocks=9):
        super(GeneratorResNet, self).__init__()
```

```
        out_features = 64
        channels = 3
        model = [
            nn.ReflectionPad2d(3),
            nn.Conv2d(channels, out_features, 7),
            nn.InstanceNorm2d(out_features),
            nn.ReLU(inplace=True),
        ]
        in_features = out_features
        # Downsampling
        for _ in range(2):
            out_features *= 2
            model += [
                nn.Conv2d(in_features, out_features, 3, \
                          stride=2, padding=1),
                nn.InstanceNorm2d(out_features),
                nn.ReLU(inplace=True),
            ]
            in_features = out_features

        # Residual blocks
        for _ in range(num_residual_blocks):
            model += [ResidualBlock(out_features)]

        # Upsampling
        for _ in range(2):
            out_features //= 2
            model += [
                nn.Upsample(scale_factor=2),
                nn.Conv2d(in_features, out_features, 3, \
                          stride=1, padding=1),
                nn.InstanceNorm2d(out_features),
                nn.ReLU(inplace=True),
            ]
            in_features = out_features

        # Output layer
        model += [nn.ReflectionPad2d(channels), \
                  nn.Conv2d(out_features, channels, 7), \
                  nn.Tanh()]
        self.model = nn.Sequential(*model)
        self.apply(weights_init_normal)
    def forward(self, x):
        return self.model(x)
```

8. Define the discriminator network (`Discriminator`):

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        channels, height, width = 3, IMAGE_SIZE, IMAGE_SIZE

        def discriminator_block(in_filters, out_filters, \
                                normalize=True):
            """Returns downsampling layers of each
            discriminator block"""
            layers = [nn.Conv2d(in_filters, out_filters, \
                                4, stride=2, padding=1)]
            if normalize:
                layers.append(nn.InstanceNorm2d(out_filters))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers

        self.model = nn.Sequential(
            *discriminator_block(channels,64,normalize=False),
            *discriminator_block(64, 128),
            *discriminator_block(128, 256),
            *discriminator_block(256, 512),
            nn.ZeroPad2d((1, 0, 1, 0)),
            nn.Conv2d(512, 1, 4, padding=1)
        )
        self.apply(weights_init_normal)

    def forward(self, img):
        return self.model(img)
```

- Define the function to generate a sample of images – `generate_sample`:

```
@torch.no_grad()
def generate_sample():
    data = next(iter(val_dl))
    G_AB.eval()
    G_BA.eval()
    real_A, real_B = data
    fake_B = G_AB(real_A)
    fake_A = G_BA(real_B)
    # Arange images along x-axis
    real_A = make_grid(real_A, nrow=5, normalize=True)
    real_B = make_grid(real_B, nrow=5, normalize=True)
    fake_A = make_grid(fake_A, nrow=5, normalize=True)
    fake_B = make_grid(fake_B, nrow=5, normalize=True)
    # Arange images along y-axis
```

```
    image_grid = torch.cat((real_A,fake_B,real_B,fake_A), 1)
    show(image_grid.detach().cpu().permute(1,2,0).numpy(), \
         sz=12)
```

9. Define the function to train the generator (`generator_train_step`):

- The function takes the two generator models (G_AB and G_BA as `Gs`), `optimizer`, and real images of the two classes – `real_A` and `real_B` – as input:

```
def generator_train_step(Gs, optimizer, real_A, real_B):
```

- Specify the generators:

```
    G_AB, G_BA = Gs
```

- Set gradients to zero for the optimizer:

```
    optimizer.zero_grad()
```

- If you were to send an orange image to an orange-generator, ideally, if the generator has understood everything about oranges, it should not make any changes to the image and should "generate" the exact image. We thus create an identity using this knowledge. The loss function corresponding to `criterion_identity` will be given just prior to training the model. Calculate the identity loss (`loss_identity`) for images of type A (apples) and type B (oranges):

```
    loss_id_A = criterion_identity(G_BA(real_A), real_A)
    loss_id_B = criterion_identity(G_AB(real_B), real_B)

    loss_identity = (loss_id_A + loss_id_B) / 2
```

- Calculate the GAN loss when the image is passed through the generator and the generated image is expected to be as close to the other class as possible (we have `np.ones` in this case when training the generator, as we are passing the fake images of a class to the discriminator of the same class):

```
    fake_B = G_AB(real_A)
    loss_GAN_AB = criterion_GAN(D_B(fake_B), \
                torch.Tensor(np.ones((len(real_A), 1, \
                                      16, 16))).to(device))
    fake_A = G_BA(real_B)
    loss_GAN_BA = criterion_GAN(D_A(fake_A), \
                torch.Tensor(np.ones((len(real_A), 1, \
```

```
                                        16, 16))).to(device))

        loss_GAN = (loss_GAN_AB + loss_GAN_BA) / 2
```

- Calculate the cycle loss. Consider a scenario where an image of an apple is to be transformed by an orange-generator to generate a fake orange, and such a fake orange is to be transformed back to an apple by the apple-generator. If the generators were perfect, this process should give back the original image, which means the following cycle losses should be zero:

```
        recov_A = G_BA(fake_B)
        loss_cycle_A = criterion_cycle(recov_A, real_A)
        recov_B = G_AB(fake_A)
        loss_cycle_B = criterion_cycle(recov_B, real_B)

        loss_cycle = (loss_cycle_A + loss_cycle_B) / 2
```

- Calculate the overall loss and perform backpropagation before returning the calculated values:

```
        loss_G = loss_GAN + lambda_cyc * loss_cycle + \
                lambda_id * loss_identity
        loss_G.backward()
        optimizer.step()
        return loss_G, loss_identity, loss_GAN, loss_cycle, \
                loss_G, fake_A, fake_B
```

10. Define the function to train the discriminator (`discriminator_train_step`):

```
    def discriminator_train_step(D, real_data, fake_data, \
                                 optimizer):
        optimizer.zero_grad()
        loss_real = criterion_GAN(D(real_data), \
                 torch.Tensor(np.ones((len(real_data), 1, \
                                      16, 16))).to(device))
        loss_fake = criterion_GAN(D(fake_data.detach()), \
                 torch.Tensor(np.zeros((len(real_data), 1, \
                                      16, 16))).to(device))
        loss_D = (loss_real + loss_fake) / 2
        loss_D.backward()
        optimizer.step()
        return loss_D
```

11. Define the generator, discriminator objects, optimizers, and loss functions:

```
G_AB = GeneratorResNet().to(device)
G_BA = GeneratorResNet().to(device)
D_A = Discriminator().to(device)
D_B = Discriminator().to(device)

criterion_GAN = torch.nn.MSELoss()
criterion_cycle = torch.nn.L1Loss()
criterion_identity = torch.nn.L1Loss()

optimizer_G = torch.optim.Adam(
    itertools.chain(G_AB.parameters(), G_BA.parameters()), \
    lr=0.0002, betas=(0.5, 0.999))
optimizer_D_A = torch.optim.Adam(D_A.parameters(), \
                        lr=0.0002, betas=(0.5, 0.999))
optimizer_D_B = torch.optim.Adam(D_B.parameters(), \
                        lr=0.0002, betas=(0.5, 0.999))

lambda_cyc, lambda_id = 10.0, 5.0
```

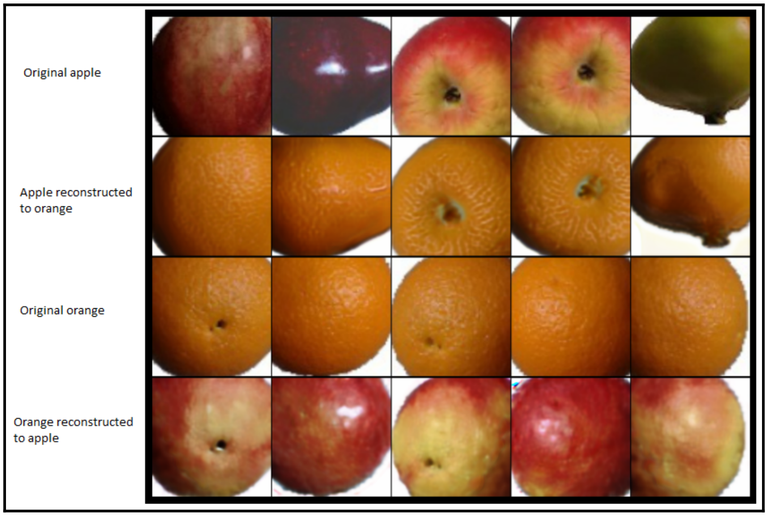
12. Train the networks over increasing epochs:

```
n_epochs = 10
log = Report(n_epochs)
for epoch in range(n_epochs):
    N = len(trn_dl)
    for bx, batch in enumerate(trn_dl):
        real_A, real_B = batch

        loss_G, loss_identity, loss_GAN, loss_cycle, \
        loss_G, fake_A, fake_B = generator_train_step(\
                                  (G_AB,G_BA), optimizer_G, \
                                  real_A, real_B)
        loss_D_A = discriminator_train_step(D_A, real_A, \
                                    fake_A, optimizer_D_A)
        loss_D_B = discriminator_train_step(D_B, real_B, \
                                    fake_B, optimizer_D_B)
        loss_D = (loss_D_A + loss_D_B) / 2
        log.record(epoch+(1+bx)/N, loss_D=loss_D.item(), \
            loss_G=loss_G.item(), loss_GAN=loss_GAN.item(), \
            loss_cycle=loss_cycle.item(), \
           loss_identity=loss_identity.item(), end='\r')
        if bx%100==0: generate_sample()

    log.report_avgs(epoch+1)
```

13. Generate images once we train the models:

    ```
    generate_sample()
    ```

    The preceding code generates the following output:



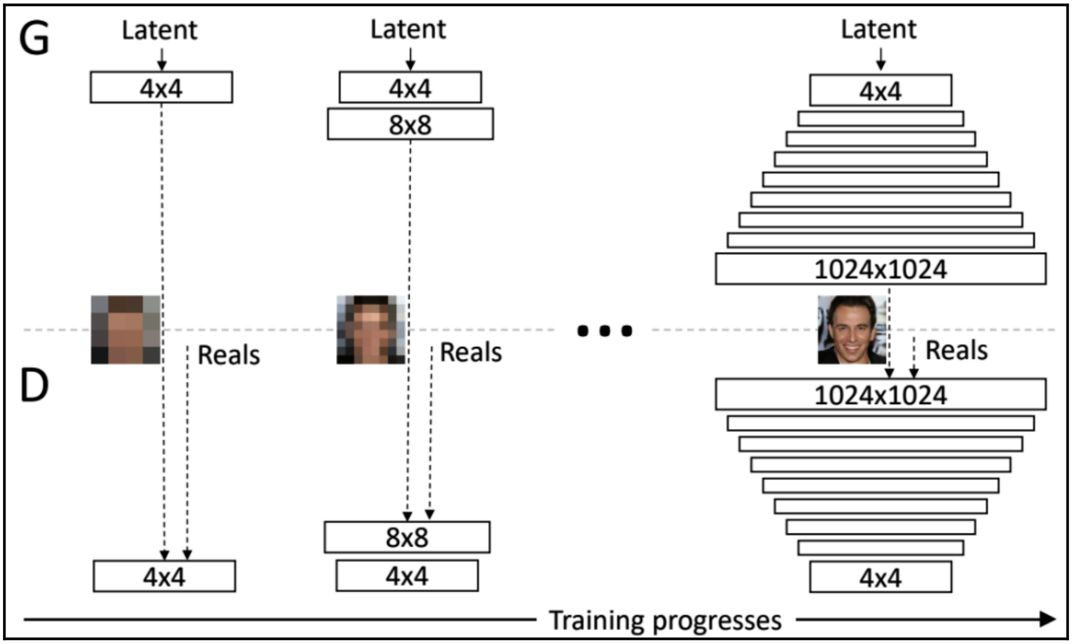
From the preceding, we can see that we are successfully able to convert apples into oranges (the first two rows) and oranges into apples (the last two rows).

So far, we have learned about paired image-to-image translation through the Pix2Pix GAN and unpaired image-to-image translation through CycleGAN. In the next section, we will learn about leveraging StyleGAN to convert an image of one style into an image of another style.

Leveraging StyleGAN on custom images

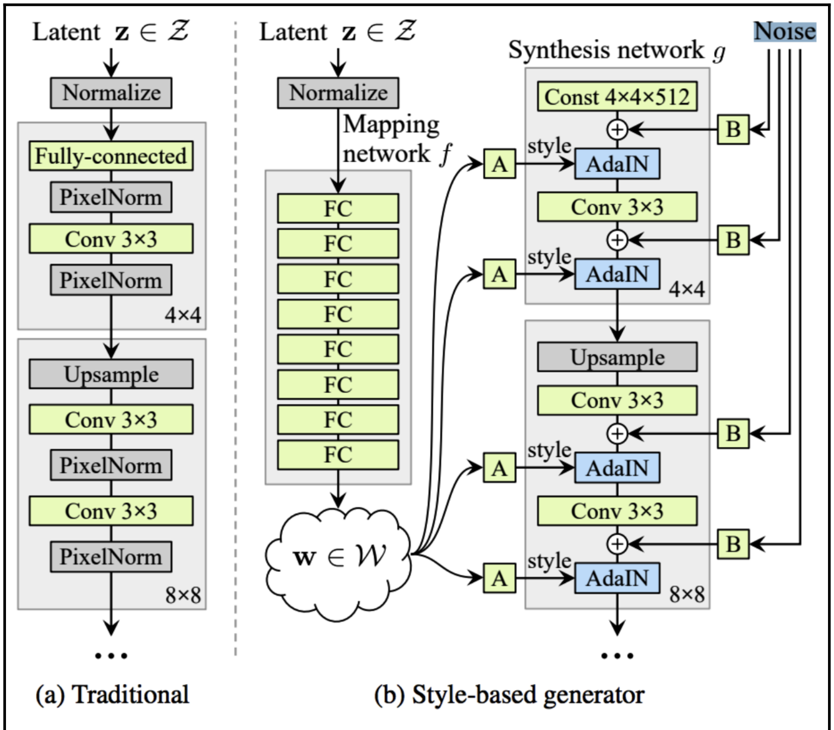
Let's first understand a few historical developments prior to the invention of StyleGAN. As we know, generating fake faces from the previous chapter involved the usage of GANs. The biggest problem that research faced was that the images that could be generated were small (typically 64 x 64). And any effort to generate images of a larger size caused the generators or discriminators to fall into local minima that would stop training and generate gibberish. One of the major leaps in generating high-quality images involved a research paper called ProGAN (short for Progressive GAN), which involved a clever trick.

The size of both the generator and discriminator is progressively increased. In the first step, you create a generator and discriminator to generate 4 x 4 images from a latent vector. After this, additional convolution (and upscaling) layers are added to the trained generator and discriminator, which will be responsible for accepting the 4 x 4 images (which are generated from latent vectors in step 1) and generating/discriminating 8 x 8 images. Once this step is also done, new layers are created in the generator and discriminator once again, to be trained to generate larger images. Step by step (progressively), the image size is increased in this way. The logic being that it is easier to add a new layer to an already well functioning network than trying to learn all the layers from scratch. In this manner, images are upscaled to a resolution of 1024 x 1024 pixels (image source: `https://arxiv.org/pdf/1710.10196v3.pdf`):



As much as it succeeded, it was fairly difficult to control individual aspects of the generated image (such as gender and age), primarily because the network is getting exactly one input (in the preceding image: Latent at the top of the network). StyleGAN addresses this scenario.

StyleGAN uses a similar training scheme where images are progressively generated, but with an added set of latent inputs every time the network grows. This means the network now accepts multiple latent vectors at regular intervals of image size generated. Every latent given at a stage of generation dictates the features (style) that are going to be generated at that stage of that network. Let's discuss the working details of StyleGAN in more detail here:



https://arxiv.org/pdf/1812.04948.pdf

In the preceding diagram, we can contrast the traditional way of generating images and the style-based generator. In a traditional generator, there is only one input. However, there is a mechanism in place within a style-based generator. Let's understand the details here:

1. Create a random noise vector *z* of size 1 x 512.
2. Feed this to an auxiliary network called the style network (or mapping network), which creates a tensor *w* of size 18 x 512.

3. The generator (synthesis) network contains 18 convolution layers. Each layer will accept the following as inputs:
    - The corresponding row of w ('A')
    - A random noise vector ('B')
    - The output from the previous layer

Note that noise ('B') is given only for regularization purposes.

The preceding three combined will create a pipeline that takes in a 1 x 512 vector and creates a 1024 x 1024 image.

Let's now understand how each of the 18 1 x 512 vectors within the 18 x 512 vector that is generated from the mapping network contributes towards the generation of an image. The 1 x 512 vector that is added at the first few layers of the synthesis network contributes towards the overall pose and large-scale features present in the image, such as pose, face shape, and so on, (as they are responsible for generating the 4 x 4, 8 x 8 images, and so on – which are the first few images that will be further enhanced in the later layers). The vectors added in the middle layers correspond to small-scale features such as hairstyle, eyes open/closed (as they are responsible for generating the 16 x 16, 32 x 32, and 64 x 64 images). The vectors added in the last few layers correspond to the color scheme and other microstructures of the image. By the time we reach the last few layers, the image structure is preserved, and the facial features are preserved but only image-level details such as lighting conditions are changed.

In this section, we will leverage a pre-trained StyleGAN2 model to customize our image of interest to have different styles.

For our objective, we will perform style transfer using the StyleGAN2 model. At a high level, here's how style-transfer on faces works (the following will be clearer as you go through the results of the code):

- Say the w_1 style vector is used to generate face-1 and the w_2 style vector is used to generate face-2. Both of them are 18 x 512.
- The first few of the 18 vectors in w_2 (which are responsible for generating images from 4 x 4 to 8 x 8 resolutions) are replaced with the corresponding vectors from w_1. Then, we transfer very coarse features such as the pose from face-1 to face-2.

- If the later style vectors (say the third to the fifteenth of the 18 x 512 – which are responsible for generating 64 x 64 to 256 x 256 dimensional batch of images) are replaced in w_2 with those from w_1, then we transfer features such as eyes, nose, and other facial mid-level features.
- If the last few style vectors (which are responsible for generating 512 x 512 to 1024 x 1024 dimensional batch of images) are replaced, fine-level features such as complexion and background (which don't affect the overall face in a significant manner) are transferred.

With an understanding of how style transfer is done, let's now understand how to perform style transfer using StyleGAN2 on custom images:

1. Take a custom image.
2. Align the custom image so that only the face region of the image is stored.
3. Fetch the latent vector that is likely to generate the custom aligned image.
4. Generate an image by passing a random latent vector (1 x 512) to the mapping network.

By this step, we have two images – our custom aligned image and the image generated by the StyleGAN2 network. We now want to transfer some of the features of the custom image to the generated image and vice versa.

Let's code up the preceding strategy.

Note that we are leveraging a pre-trained network fetched from a GitHub repository, as training such a network takes days if not weeks:

> You need a CUDA-enabled environment to run the following code. The following code is available as `Customizing_StyleGAN2.ipynb` in the `Chapter13` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from and is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components from text.

1. Clone the repository, install the requirements, and fetch the pre-trained weights:

```
import os
if not os.path.exists('pytorch_stylegan_encoder'):
    !git clone
https://github.com/jacobhallberg/pytorch_stylegan_encoder.git
```

```
    %cd pytorch_stylegan_encoder
    !git submodule update --init --recursive
    !wget -q
https://github.com/jacobhallberg/pytorch_stylegan_encoder/rele
ases/download/v1.0/trained_models.zip
    !unzip -q trained_models.zip
    !rm trained_models.zip
    !pip install -qU torch_snippets
    !mv trained_models/stylegan_ffhq.pth
InterFaceGAN/models/pretrain
else:
    %cd pytorch_stylegan_encoder
from torch_snippets import *
```

2. Load the pre-trained generator and the synthesis network, mapping the network's weights:

```
from InterFaceGAN.models.stylegan_generator import
StyleGANGenerator
from models.latent_optimizer import PostSynthesisProcessing

synthesizer=StyleGANGenerator("stylegan_ffhq").model.synthesis
mapper = StyleGANGenerator("stylegan_ffhq").model.mapping
trunc = StyleGANGenerator("stylegan_ffhq").model.truncation
```

3. Define the function to generate an image from a random vector:

```
post_processing = PostSynthesisProcessing()
post_process = lambda image: post_processing(image)\
                .detach().cpu().numpy().astype(np.uint8)[0]

def latent2image(latent):
    img = post_process(synthesizer(latent))
    img = img.transpose(1,2,0)
    return img
```

4. Generate a random vector:

```
rand_latents = torch.randn(1,512).cuda()
```

In the preceding code, we are passing the random 1 x 512-dimensional vector through mapping and truncation networks to generate a vector that is 1 x 18 x 512. These 18 x 512 vectors are the ones that dictate the style of the generated image.

5. Generate an image from the random vector:

```
show(latent2image(trunc(mapper(rand_latents))), sz=5)
```

The preceding code generates the following output:

So far, we have generated an image. In the next few lines of code, you will learn about performing style transfer between the preceding generated image and an image of your choice.

6. Fetch a custom image (`MyImage.jpg`) and align it. Alignment is important to generate proper latent vectors as all generated images in StyleGAN have the face centered and features prominently visible:

```
!wget https://www.dropbox.com/s/lpw10qawsc5ipbn/MyImage.JPG\
 -O MyImage.jpg
!git clone https://github.com/Puzer/stylegan-encoder.git
!mkdir -p stylegan-encoder/raw_images
!mkdir -p stylegan-encoder/aligned_images
!mv MyImage.jpg stylegan-encoder/raw_images
```

7. Align the custom image:

```
!python stylegan-encoder/align_images.py \
stylegan-encoder/raw_images/ \
stylegan-encoder/aligned_images/
!mv stylegan-encoder/aligned_images/* ./MyImage.jpg
```

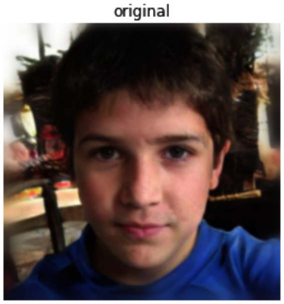
8. Use the aligned image to generate latents that can reproduce the aligned image perfectly. This is a process of identifying the latent vector combination that minimizes the difference between the aligned image and the image generated from the latent vector:

```
from PIL import Image
img = Image.open('MyImage.jpg')
show(np.array(img), sz=4, title='original')

!python encode_image.py ./MyImage.jpg\
```

```
 pred_dlatents_myImage.npy\
 --use_latent_finder true\
 --image_to_latent_path ./trained_models/image_to_latent.pt

pred_dlatents = np.load('pred_dlatents_myImage.npy')
pred_dlatent = torch.from_numpy(pred_dlatents).float().cuda()
pred_image = latent2image(pred_dlatent)
show(pred_image, sz=4, title='synthesized')
```

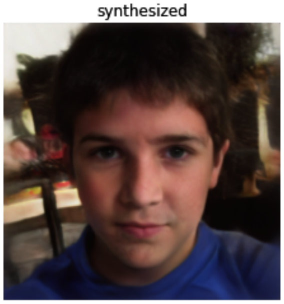
The preceding code generates the following output:

original



```
Optimizing Latents.
[2020-10-13 04:54:26,699][INFO] Loading pytorch model from `InterFaceGAN/models/pretrain/stylegan_ffhq.pth`.
[2020-10-13 04:54:26,982][INFO] Successfully loaded!
[2020-10-13 04:54:26,983][INFO]   `lod` of the loaded model is 0.0.
Step: 999, Loss: 2.4222376346588135: 100% 1000/1000 [02:06<00:00,  7.89it/s]
```

synthesized



The Python script `encode_image.py`, at a high level, does the following:

1. Creates a random vector in w space.
2. Synthesizes an image with this vector.
3. Compares the synthesized image with the original input image using VGG's perceptual loss (the same loss that was used in neural style transfer).

4. Perform backpropagation on the w random vector to reduce this loss for a fixed number of iterations.
5. The optimized w vector will now synthesize an image for which VGG gives near-identical features as the input image, and hence the synthesized image will look similar to the input image.

Now that we have the latent vectors that correspond to the image of interest, let's perform style transfer between images in the next step.

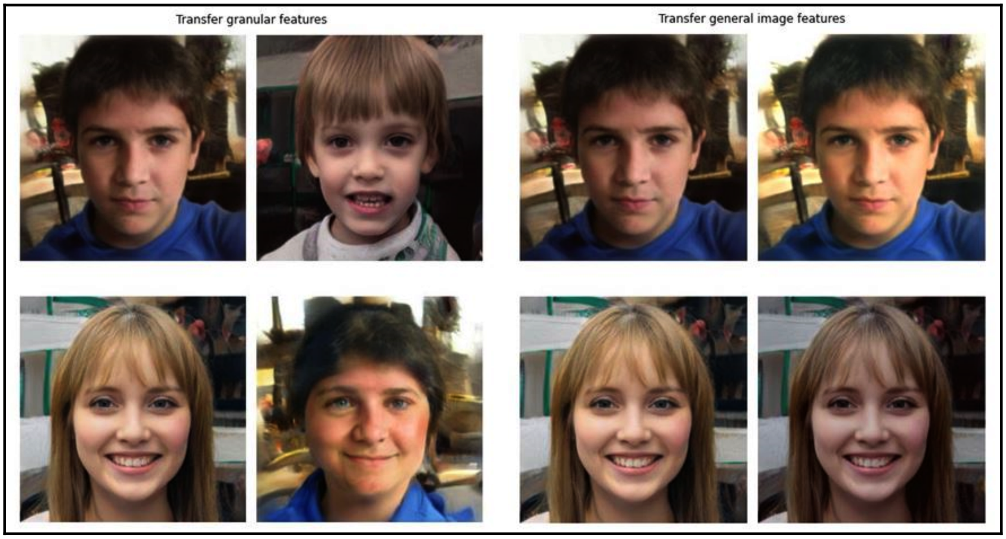
9. Perform style transfer:

As discussed, the core logic behind style transfer is actually the transfer of parts of style tensors, that is, a subset of 18 of the 18 x 512 style tensors. Here, we will be transferring the first two rows (of the 18 x 512) in one case, 3-15 rows in one case, and 15-18 rows in one case. Since each set of vectors is responsible for generating different aspects of the image, each set of swapped vectors swap different features in the image:

```
idxs_to_swap = slice(0,3)
my_latents=torch.Tensor(np.load('pred_dlatents_myImage.npy', \
                                   allow_pickle=True))

A, B = latent2image(my_latents.cuda()),
latent2image(trunc(mapper(rand_latents)))
generated_image_latents = trunc(mapper(rand_latents))

x = my_latents.clone()
x[:,idxs_to_swap] = generated_image_latents[:,idxs_to_swap]
a = latent2image(x.float().cuda())

x = generated_image_latents.clone()
x[:,idxs_to_swap] = my_latents[:,idxs_to_swap]
b = latent2image(x.float().cuda())

subplots([A,a,B,b], figsize=(7,8), nc=2, \
         suptitle='Transfer high level features')
```

The preceding code generates this:

Here's the output with `idxs_to_swap` as `slice(4,15)` and `slice(15,18)` respectively.



10. Next, we extrapolate a style vector such that the new vectors will only change the smileyness of our custom image. For this, you need to compute the right direction to move the z latent vector in. We can achieve this by first creating a lot of fake images. An SVM classifier is then used to train and find out if the persons within images are smiling or not. This SVM hence creates a hyperplane that separates smiling from non-smiling faces. The required direction to move z is going to be normal to this hyperplane, which is presented as `stylegan_ffhq_smile_w_boundary.npy`. Implementation details can be found in the `InterfaceGAN/edit.py` code itself:

```
!python InterFaceGAN/edit.py\
 -m stylegan_ffhq\
 -o results_new_smile\
 -b
InterFaceGAN/boundaries/stylegan_ffhq_smile_w_boundary.npy\
 -i pred_dlatents_myImage.npy\
 -s WP\
 --steps 20

generated_faces = glob.glob('results_new_smile/*.jpg')

subplots([read(im,1) for im in sorted(generated_faces)], \
         figsize=(10,10))
```

Here's how the generated images look:

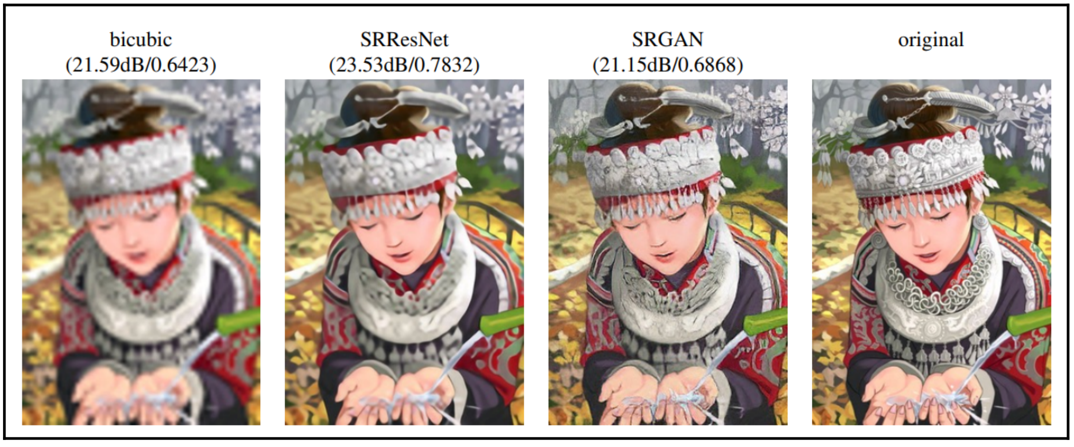
In summary, we have learned how the research has progressed in generating very high-resolution images of faces using GANs. The trick is to increase the complexity of both the generator and discriminator in steps of increasing resolution so that at each step, both the models are decent at their tasks. We learned how you are able to manipulate the style of a generated image by ensuring features at every resolution are dictated by an independent input called a style vector. We also learned how to manipulate styles of different images by swapping styles from one image to another.

Now that we have learned about leveraging the pre-trained StyleGAN2 model to perform style transfer, in the next section, we will leverage the pre-trained Super-resolution GAN model to generate images in high resolution.

Super-resolution GAN

In the previous section, we saw a scenario where we leveraged the pre-trained StyleGAN to generate images in a given style. In this section, we will take it a step further and learn about leveraging pre-trained models to perform image super-resolution. We will gain an understanding of the architecture of the Super-resolution GAN model before implementing it on images.

First, we will understand the reason why a GAN is a good solution for the task of super-resolution. Imagine a scenario where you are given an image and asked to increase its resolution. Intuitively, you would consider various interpolation techniques to perform super-resolution. Here's a sample low-resolution image along with the outputs of various techniques (image source: `https://arxiv.org/pdf/1609.04802.pdf`):



From the preceding image, we can see that traditional interpolation techniques such as bicubic interpolation do not help as much when reconstructing the image from a low resolution (a 4X down-scaled image of the original image).

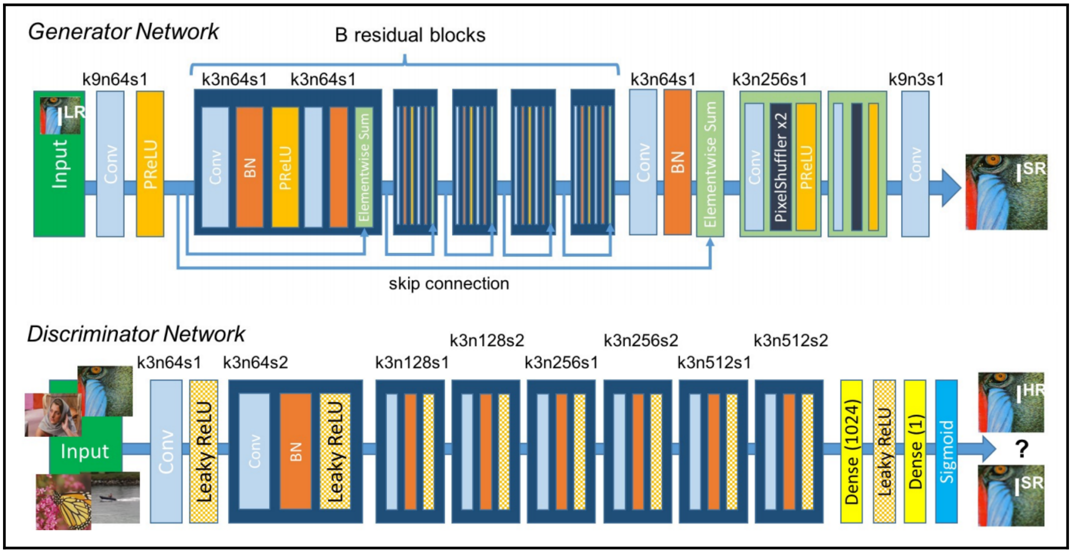
While a super-resolution ResNet-based UNet can come in handy in such a scenario, GANs can be more useful, as they simulate human perception. The discriminator, given that it knows how a typical super-resolution image looks, can detect a scenario where the generated image has properties that do not necessarily look like an image with high resolution.

With the need for GANs for super-resolution established, let's learn about and leverage the pre-trained model.

Architecture

While it is possible to code and train a super-resolution GAN from scratch, we will leverage pre-trained models where we can. Hence, for this section, we will leverage the model developed by Christian Ledig and team and published in the paper titled *Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network*.

The architecture of SRGAN is as follows (image source: `https://arxiv.org/pdf/1609.04802.pdf`):



From the preceding image, we see that the discriminator takes high-resolution images as input to train a model that predicts whether an image is a high-resolution or a low-resolution image. The generator network takes the low-resolution image as input and comes up with a high-resolution image. While training the model, both content loss and adversarial loss are minimized. For a detailed understanding of the details of model training and a comparison of results across the various techniques used to come up with a high-resolution image, we recommend you go through the paper.

With a high-level understanding of how the model is built, we will code the way to leverage a pre-trained SRGAN model to convert a low-resolution image into a high-resolution image.

Coding SRGAN

Here are the steps for loading the pre-trained SRGAN and making our predictions:

> The following code is available as `Image super resolution using SRGAN.ipynb` in the `Chapter 13` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components from text.

1. Import the relevant packages and the pre-trained model:

```
import os
if not os.path.exists('srgan.pth.tar'):
    !pip install -q torch_snippets
    !wget -q
https://raw.githubusercontent.com/sizhky/a-PyTorch-Tutorial-to
-Super-Resolution/master/models.py -O models.py
    from pydrive.auth import GoogleAuth
    from pydrive.drive import GoogleDrive
    from google.colab import auth
    from oauth2client.client import GoogleCredentials

    auth.authenticate_user()
    gauth = GoogleAuth()
    gauth.credentials = \
            GoogleCredentials.get_application_default()
    drive = GoogleDrive(gauth)

    downloaded = drive.CreateFile({'id': \
```

```
                  '1_PJ1Uimbr0xrPjE8U3Q_bG7XycGgsbVo'})
    downloaded.GetContentFile('srgan.pth.tar')
    from torch_snippets import *
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

2. Load the model:

```
model = torch.load('srgan.pth.tar',
map_location='cpu')['generator'].to(device)
model.eval()
```

3. Fetch the image to convert to a high resolution:

```
!wget https://www.dropbox.com/s/nmzwu68nrl9j0lf/Hema6.JPG
```

4. Define the functions to `preprocess` and `postprocess` the image:

```
preprocess = T.Compose([
                T.ToTensor(),
                T.Normalize([0.485, 0.456, 0.406],
                            [0.229, 0.224, 0.225]),
                T.Lambda(lambda x: x.to(device))
            ])

postprocess = T.Compose([
                T.Lambda(lambda x: (x.cpu().detach()+1)/2),
                T.ToPILImage()
            ])
```

5. Load the image and preprocess it:

```
image = readPIL('Hema6.JPG')
image.size
# (260,181)
image = image.resize((130,90))
im = preprocess(image)
```

Note that, in the preceding code, we have performed an additional resize on the original image to further blur the image, but this is done only for illustration – as the improvement is more visible when we down-scale an image.

6. Pass the preprocessed image through the loaded `model` and `postprocess` the output of the model:

```
sr = model(im[None])[0]
sr = postprocess(sr)
```

7. Plot the original and the high-resolution images:

```
subplots([image, sr], nc=2, figsize=(10,10), \
         titles=['Original image','High resolution image'])
```

The preceding code results in the following output:

From the preceding image, we can see that the high-resolution image captured details that were blurred in the original image.

Note that the contrast between the original and the high-resolution image will be high if the original image is blurred. However, if the original image is not blurred, the contrast will not be that high. We encourage you to work with images of varying resolutions.

Summary

In this chapter, we have learned about generating images from a given contour using the Pix2Pix GAN. Further, we learned about the various loss functions in CycleGAN to convert images of one class to another. Next, we learned about how StyleGAN helps in generating realistic faces and also copying the style from one image to another based on the way in which the generator is trained. Finally, we learned about leveraging the pre-trained SRGAN model to generate high-resolution images.

In the next chapter, we will switch to learning about training an image classification model based on very few (typically less than 20) images.

Questions

1. Why do we need the Pix2Pix GAN where a supervised learning algorithm such as UNet could have worked to generate images from contours?
2. Why do we need to optimize for three different loss functions in CycleGAN?
3. How do the tricks leveraged in ProgressiveGAN help in building StyleGAN?
4. How do we identify latent vectors that correspond to a given custom image?

4

Section 4 - Combining Computer Vision with Other Techniques

In this final section, we will learn about merging computer vision techniques with techniques in other fields, such as NLP, reinforcement learning, and tools such as OpenCV, to come up with new ways of solving traditional problems.

This section comprises the following chapters:

14
Training with Minimal Data Points

So far, in the previous chapters, we have learned about classifying images where we have hundreds/ thousands of example images to train on per class. In this chapter, we will learn about various techniques that will help in classifying an image even when there are very few training examples per class. We will start by training a model to predict a class, even though the images corresponding to the class are not present during training. Next, we will move on to a scenario where only a few images of the class we are trying to predict are present during training. We will code Siamese networks, which fall into the category of few-shot learning, and understand the working details of relation networks and prototypical networks.

We will learn about the following topics in this chapter:

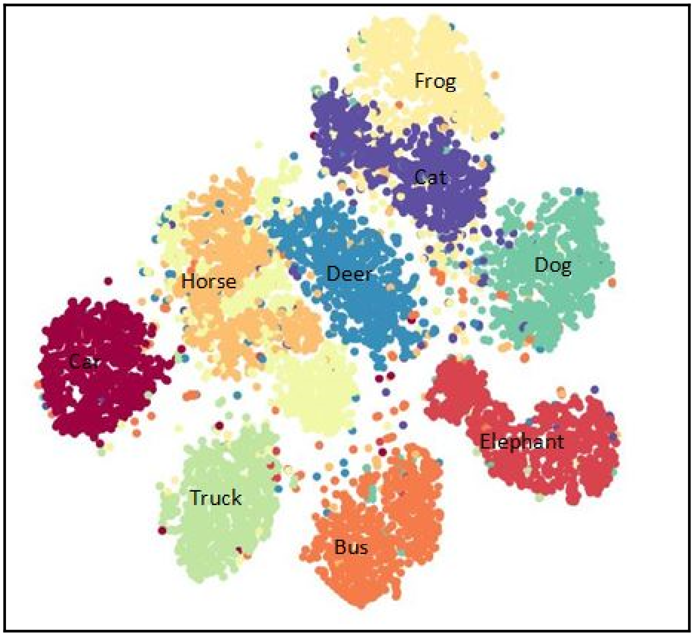
- Implementing zero-shot learning
- Implementing few-shot learning

Implementing zero-shot learning

Imagine a scenario where I ask you to predict the class of objects in an image where you have not seen an image of the object class earlier. How would you make predictions in such a scenario?

Intuitively, we resort to the attributes of the object in the image and then try to identify the object that matches the most attributes.

In one such scenario where we have to come up with attributes automatically (where the attributes are not given for training), we leverage word vectors. Word vectors encompass semantic similarity among words. For example, all animals would have similar word vectors and automobiles would have very different word vector representations. While the generation of word vectors is out of scope for this book, we will work on pre-trained word vectors. At a very high level, words that have similar surrounding words (context) will have similar vectors. Here's a sample t-SNE representation of word vectors:



From the preceding sample, we can see that the word vectors of automobiles fall to the left of the chart while the vectors corresponding to animals are on the right. Further, similar animals also have similar vectors.

This gives us the intuition that words, just like images, also have vector embeddings that help in obtaining similarity.

In the next section, as we code zero-shot learning, we will leverage this phenomenon to identify classes that are not seen by the model during training. Essentially, we will learn about mapping image features to word features, directly.

Coding zero-shot learning

The high-level strategy we adopt while coding zero-shot learning is as follows:

1. Import the dataset – which constitutes images and their corresponding classes.
2. Fetch the word vectors corresponding to each class from pre-trained word vector models.
3. Pass an image through a pre-trained image model such as VGG16.
4. We expect the network to predict the word vector corresponding to the object in the image.
5. Once we've trained the model, we predict the word vector on new images.
6. The class of word vector that is closest to the predicted word vector is the class of the image.

Let's code the preceding strategy as follows:

> The following code is available as `Zero_shot_learning.ipynb` in the `Chapter14` folder in this book's GitHub repository - `https://tinyurl.com/mcvp-packt` Be sure to copy the URL from the notebook in GitHub to avoid any issue while reproducing the results

1. Clone our GitHub repository that contains the dataset of this exercise and also import the relevant packages:

```
!git clone https://github.com/sizhky/zero-shot-learning/
!pip install -Uq torch_snippets
%cd zero-shot-learning/src
import gzip, _pickle as cPickle
from torch_snippets import *
from sklearn.preprocessing import LabelEncoder, normalize
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

2. Define the paths to features data (`DATAPATH`) and also the word2vec embeddings (`WORD2VECPATH`):

```
WORD2VECPATH = "../data/class_vectors.npy"
DATAPATH = "../data/zeroshot_data.pkl"
```

3. Extract the list of classes available:

```
with open('train_classes.txt', 'r') as infile:
    train_classes = [str.strip(line) for line in infile]
```

4. Load the feature vector data:

```
with gzip.GzipFile(DATAPATH, 'rb') as infile:
    data = cPickle.load(infile)
```

5. Define the training data and the data that belongs to zero-shot classes (the classes that are not present during training). Note that we will only show the classes belonging to training classes and hide the zero-shot model classes until the inference time:

```
training_data = [instance for instance in data if \
                instance[0] in train_classes]
zero_shot_data = [instance for instance in data if \
                instance[0] not in train_classes]
np.random.shuffle(training_data)
```

6. Fetch 300 training images per class for training and the remaining training class images for validation:

```
train_size = 300 # per class
train_data, valid_data = [], []
for class_label in train_classes:
    ctr = 0
    for instance in training_data:
        if instance[0] == class_label:
            if ctr < train_size:
                train_data.append(instance)
                ctr+=1
            else:
                valid_data.append(instance)
```

7. Shuffle the training and validation data and fetch the vectors corresponding to the classes into a dictionary – `vectors`:

```
np.random.shuffle(train_data)
np.random.shuffle(valid_data)
vectors = {i:j for i,j in np.load(WORD2VECPATH, \
                                allow_pickle=True)}
```

8. Fetch the image and word embedding features for training and validation data:

```
train_data=[(feat,vectors[clss]) for clss,feat in train_data]
valid_data=[(feat,vectors[clss]) for clss,feat in valid_data]
```

9. Fetch the training, validation, and zero-shot classes:

```
train_clss = [clss for clss,feat in train_data]
valid_clss = [clss for clss,feat in valid_data]
zero_shot_clss = [clss for clss,feat in zero_shot_data]
```

10. Define the input and output arrays of training data, validation data, and zero-shot data:

```
x_train, y_train = zip(*train_data)
x_train, y_train = np.squeeze(np.asarray(x_train)), \
                   np.squeeze(np.asarray(y_train))
x_train = normalize(x_train, norm='l2')

x_valid, y_valid = zip(*valid_data)
x_valid, y_valid = np.squeeze(np.asarray(x_valid)), \
                   np.squeeze(np.asarray(y_valid))
x_valid = normalize(x_valid, norm='l2')

y_zsl, x_zsl = zip(*zero_shot_data)
x_zsl, y_zsl = np.squeeze(np.asarray(x_zsl)), \
               np.squeeze(np.asarray(y_zsl))
x_zsl = normalize(x_zsl, norm='l2')
```

11. Define the training and validation datasets and dataloaders:

```
from torch.utils.data import TensorDataset

trn_ds = TensorDataset(*[torch.Tensor(t).to(device) for t in \
                         [x_train, y_train]])
val_ds = TensorDataset(*[torch.Tensor(t).to(device) for t in \
                         [x_valid, y_valid]])
trn_dl = DataLoader(trn_ds, batch_size=32, shuffle=True)
val_dl = DataLoader(val_ds, batch_size=32, shuffle=False)
```

12. Build a model that takes the 4,096-dimensional feature as input and predicts the 300-dimensional vector as output:

```
def build_model():
    return nn.Sequential(
        nn.Linear(4096, 1024), nn.ReLU(inplace=True),
        nn.BatchNorm1d(1024), nn.Dropout(0.8),
        nn.Linear(1024, 512), nn.ReLU(inplace=True),
        nn.BatchNorm1d(512), nn.Dropout(0.8),
        nn.Linear(512, 256), nn.ReLU(inplace=True),
        nn.BatchNorm1d(256), nn.Dropout(0.8),
        nn.Linear(256, 300)
    )
```

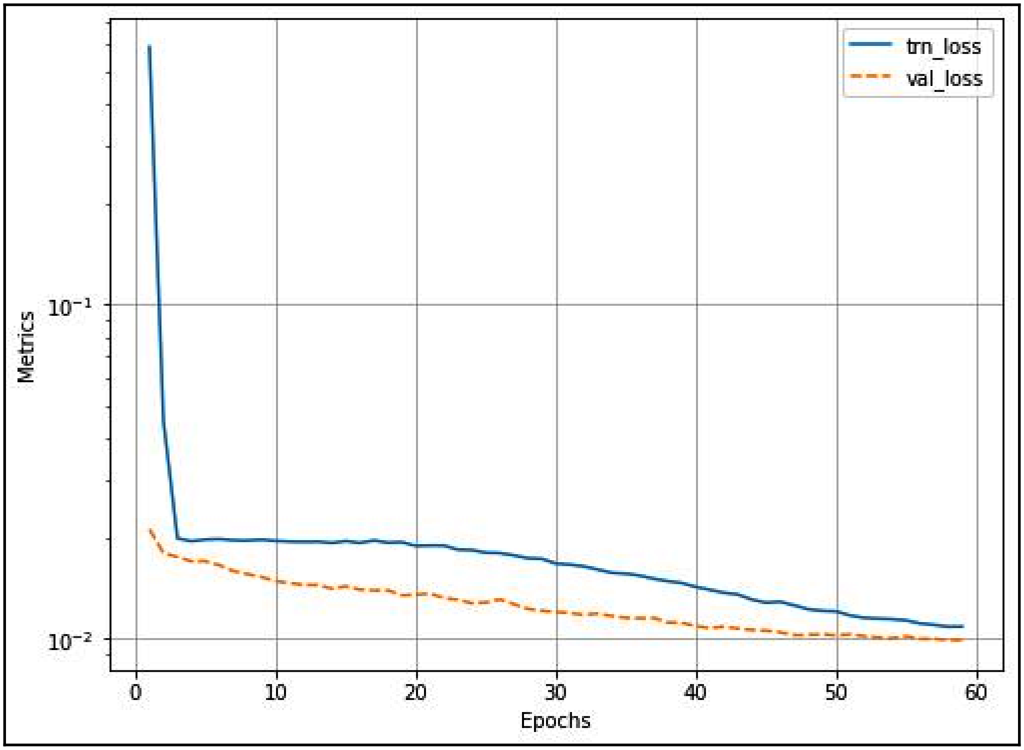
13. Define functions to train and validate on a batch of data:

```
def train_batch(model, data, optimizer, criterion):
    model.train()
    ims, labels = data
    _preds = model(ims)
    optimizer.zero_grad()
    loss = criterion(_preds, labels)
    loss.backward()
    optimizer.step()
    return loss.item()

@torch.no_grad()
def validate_batch(model, data, criterion):
    model.eval()
    ims, labels = data
    _preds = model(ims)
    loss = criterion(_preds, labels)
    return loss.item()
```

14. Train the model over increasing epochs:

```
model = build_model().to(device)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3)
n_epochs = 60

log = Report(n_epochs)
for ex in range(n_epochs):
    N = len(trn_dl)
    for bx, data in enumerate(trn_dl):
        loss = train_batch(model, data, optimizer, criterion)
        log.record(ex+(bx+1)/N, trn_loss=loss, end='\r')

    N = len(val_dl)
    for bx, data in enumerate(val_dl):
        loss = validate_batch(model, data, criterion)
        log.record(ex+(bx+1)/N, val_loss=loss, end='\r')

    if not (ex+1)%10: log.report_avgs(ex+1)

log.plot_epochs(log=True)
```

The preceding code results in the following output:



15. Predict on images (`x_zsl`) that contain the zero-shot classes (classes that the model has not seen) and also fetch the actual features (`vectors`) and `classnames` corresponding to all available classes:

```
pred_zsl = model(torch.Tensor(x_zsl).to(device)).cpu()\
                                    .detach().numpy()
class_vectors = sorted(np.load(WORD2VECPATH, \
                allow_pickle=True), key=lambda x: x[0])
classnames, vectors = zip(*class_vectors)
classnames = list(classnames)

vectors = np.array(vectors)
```

16. Calculate the distance between each predicted vector and the vector corresponding to the available classes and measure the number of zero-shot classes present in the top five predictions:

```
dists = (pred_zsl[None] - vectors[:,None])
dists = (dists**2).sum(-1).T

best_classes = []
for item in dists:
    best_classes.append([classnames[j] for j in \
                         np.argsort(item)[:5]])

np.mean([i in J for i,J in zip(zero_shot_clss, best_classes)])
```

From the preceding, we can see that we can predict correctly for ~73% of the images that contain an object whose class is not present during training, in the top 5 predictions of the model. Note that the percentages of correctly classified images will be 6%, 14%, and 40% for the top 1,2, and 3 predictions, respectively.

Now that we have seen a scenario on addressing predictions when no images of a class are present in training through zero-shot classification, in the next section, we will learn about building a model to predict the class of object in the image if there are only a few examples of a class in the training set.

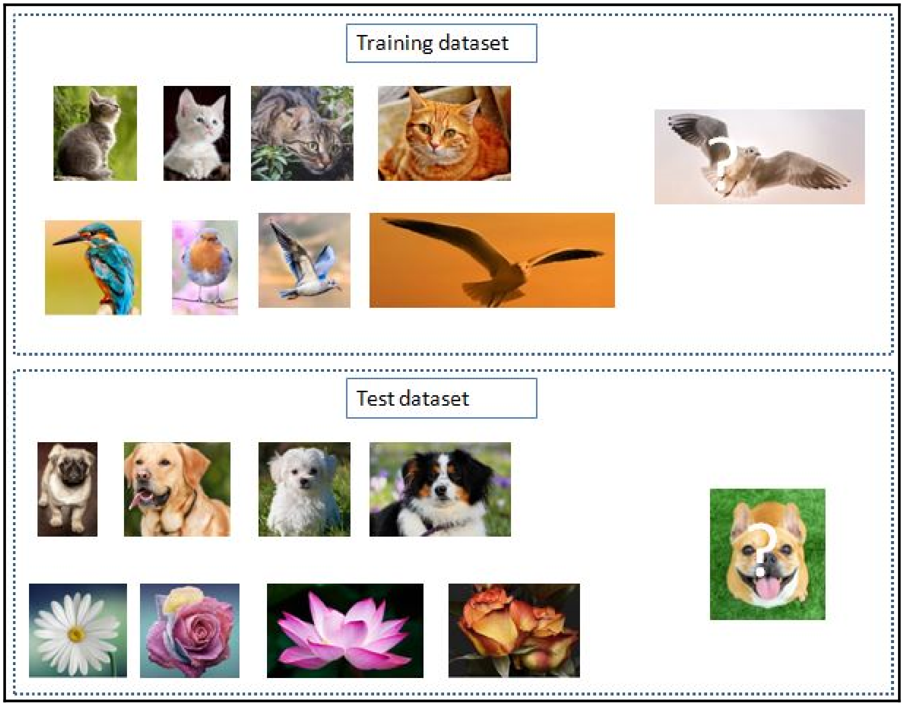
Implementing few-shot learning

Imagine a scenario where we give you only 10 images of a person and ask you to identify whether a new image is of the same person. As humans, we can classify such tasks with ease. However, the deep learning-based algorithms that we have learned so far would require hundreds/ thousands of labeled examples to classify accurately.

Multiple algorithms that fall in the meta-learning paradigm come in handy to solve this scenario. In this section, we will learn about Siamese networks, prototypical networks, and relation matching networks that work towards solving the few-images problem.

All three algorithms aim towards learning to compare two images to come up with a score for how similar the images are.

Here's an example of what to expect during few-shot classification:



In the preceding representative datasets, we have shown a few images of each class to the network while training and asked it to predict the class for a new image based on the images.

So far, we have been using pre-trained models to solve such problems. However, such models are likely to overfit soon, given the tiny amount of data that is available.

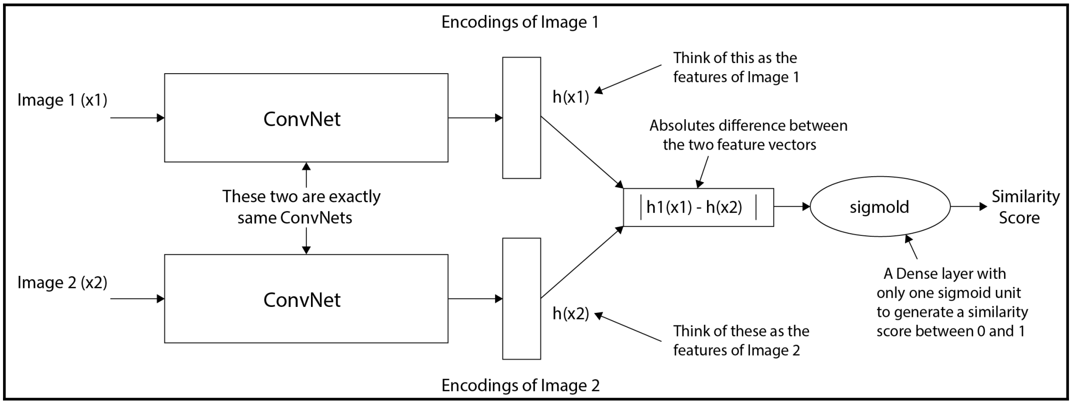
You can leverage multiple metrics, models, and optimization-based architectures to solve such scenarios. In this chapter, we will learn about metric-based architectures that come up with an optimal metric, either a Euclidean distance or cosine similarity, to group similar images together and then predict on a new image.

An N-shot k-class classification is where there are N images each for the k classes to train the network.

In the next sections, we will understand the working details and code Siamese networks, and also understand the working details of prototypical and relation networks.

Building a Siamese network

Here, it is the network through which our two images (one reference image and the query image) pass. Let's understand the working details of Siamese networks and how they help in identifying images of the same person with only a few images. First, let's go through the high-level overview of how Siamese networks work:



We go through the following steps:

1. Pass an image through a convolution network.
2. Pass another image through the same neural network as in step 1.
3. Calculate the encodings (features) of both images.
4. Calculate the difference between the two feature vectors.
5. Pass the difference vector through sigmoid activation, which represents whether the two images are similar.

The word Siamese in the preceding architecture relates to passing two images through a twin network (where we duplicate the network to handle two images) to fetch image encodings of each of the two images. Further, we are comparing the encodings of two images to fetch a similarity score for the two images. If the similarity score (or dissimilarity score) is beyond a threshold, we consider the images to be of the same person.

With this strategy in place, let's code the Siamese network to predict the class corresponding to the image – where the class of images occurred only a few times in training data.

Coding Siamese networks

In this section, we will learn about coding Siamese networks to predict whether the image of a person matches a reference image in our database.

The high-level strategy that we adopt is the following:

1. Fetch the dataset.
2. Create data in such a way that the dissimilarity of two images of the same person will be low and dissimilarity is high when two images are of different persons.
3. Build a **convolutional neural network** (**CNN**).
4. We expect the CNN to sum the loss values both corresponding to the classification loss if the images are of the same person, and the distance between the two images. We use contrastive loss for this exercise.
5. Train the model over increasing epochs.

Let's code the preceding strategy:

> The following code is available as `Siamese_networks.ipynb` in the `Chapter14` folder in this book's GitHub repository - `https://tinyurl.com/mcvp-packt` Be sure to copy the URL from the notebook in GitHub to avoid any issue while reproducing the results

1. Import the relevant packages and dataset:

```
!pip install torch_snippets
from torch_snippets import *
!wget
https://www.dropbox.com/s/ua1rr8btkmpqjxh/face-detection.zip
!unzip face-detection.zip
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

The training data comprises 38 folders (each corresponding to a different person) and each folder contains 10 sample images of the person. The testing data comprises 3 folders of 3 different persons with 10 images of each.

2. Define the dataset class – `SiameseNetworkDataset`:

   - The `__init__` method takes the `folder` containing images and the transformation to perform (`transform`) as inputs:

```
class SiameseNetworkDataset(Dataset):
    def __init__(self, folder, transform=None, \
                 should_invert=True):
        self.folder = folder
        self.items = Glob(f'{self.folder}/*/*')
        self.transform = transform
```

   - Define the `__getitem__` method:

```
    def __getitem__(self, ix):
        itemA = self.items[ix]
        person = fname(parent(itemA))
        same_person = randint(2)
        if same_person:
            itemB = choose(Glob(f'{self.folder}/{person}/*', \
                                silent=True))
        else:
            while True:
                itemB = choose(self.items)
                if person != fname(parent(itemB)):
                    break
        imgA = read(itemA)
        imgB = read(itemB)
        if self.transform:
            imgA = self.transform(imgA)
            imgB = self.transform(imgB)
        return imgA, imgB, np.array([1-same_person])
```

   In the preceding code, we are fetching two images—`imgA` and `imgB`, and returning the third output of 0 if it is the same person and 1 if it isn't.

   - Define the `__len__` method:

```
    def __len__(self):
        return len(self.items)
```

3. Define the transformations to perform, and prepare the dataset and data loaders for the training and validation data:

```
from torchvision import transforms

trn_tfms = transforms.Compose([
            transforms.ToPILImage(),
            transforms.RandomHorizontalFlip(),
            transforms.RandomAffine(5, (0.01,0.2), \
                                    scale=(0.9,1.1)),
            transforms.Resize((100,100)),
            transforms.ToTensor(),
            transforms.Normalize((0.5), (0.5))
        ])
val_tfms = transforms.Compose([
            transforms.ToPILImage(),
            transforms.Resize((100,100)),
            transforms.ToTensor(),
            transforms.Normalize((0.5), (0.5))
        ])

trn_ds=SiameseNetworkDataset(folder="./data/faces/training/" \
                             , transform=trn_tfms)
val_ds=SiameseNetworkDataset(folder="./data/faces/testing/", \
                              transform=val_tfms)

trn_dl = DataLoader(trn_ds, shuffle=True, batch_size=64)
val_dl = DataLoader(val_ds, shuffle=False, batch_size=64)
```

4. Define the neural network architecture:

   - Define the convolution block (`convBlock`):

```
def convBlock(ni, no):
    return nn.Sequential(
        nn.Dropout(0.2),
        nn.Conv2d(ni, no, kernel_size=3, padding=1, \
                  padding_mode='reflect'),
        nn.ReLU(inplace=True),
        nn.BatchNorm2d(no),
    )
```

- Define the `SiameseNetwork` architecture that returns a five-dimensional encoding given an input:

```
class SiameseNetwork(nn.Module):
    def __init__(self):
        super(SiameseNetwork, self).__init__()
        self.features = nn.Sequential(
            convBlock(1,4),
            convBlock(4,8),
            convBlock(8,8),
            nn.Flatten(),
            nn.Linear(8*100*100, 500), nn.ReLU(inplace=True),
            nn.Linear(500, 500), nn.ReLU(inplace=True),
            nn.Linear(500, 5)
        )

    def forward(self, input1, input2):
        output1 = self.features(input1)
        output2 = self.features(input2)
        return output1, output2
```

5. Define the `ContrastiveLoss` function:

```
class ContrastiveLoss(torch.nn.Module):
    """
    Contrastive loss function.
Based on:
http://yann.lecun.com/exdb/publis/pdf/hadsell-chopra-lecun-06.
pdf
    """

    def __init__(self, margin=2.0):
        super(ContrastiveLoss, self).__init__()
        self.margin = margin
```

Note that the margin here is like the margin in SVM, where we want the margin between datapoints belonging to two distinct classes to be as high as possible.

- Define the `forward` method:

```
    def forward(self, output1, output2, label):
        euclidean_distance = F.pairwise_distance(output1, \
                                    output2, keepdim = True)
        loss_contrastive = torch.mean((1-label) * \
                            torch.pow(euclidean_distance, 2) + \
                            (label) * torch.pow(torch.clamp( \
                            self.margin - euclidean_distance, \
```

```
                                                          min=0.0), 2))
            acc = ((euclidean_distance>0.6)==label).float().mean()
            return loss_contrastive, acc
```

In the preceding code, we are fetching encodings of two different images – `output1` and `output2` and calculating their `eucledian_distance`.

Next, we are calculating the contrastive loss – `loss_contrastive`, which penalizes for having a high Euclidean distance between images of the same label, and also for having a low Euclidean distance and `self.margin` for images of different labels.

6. Define the functions to train on a batch of data and validate:

```
def train_batch(model, data, optimizer, criterion):
    imgsA, imgsB, labels = [t.to(device) for t in data]
    optimizer.zero_grad()
    codesA, codesB = model(imgsA, imgsB)
    loss, acc = criterion(codesA, codesB, labels)
    loss.backward()
    optimizer.step()
    return loss.item(), acc.item()

@torch.no_grad()
def validate_batch(model, data, criterion):
    imgsA, imgsB, labels = [t.to(device) for t in data]
    codesA, codesB = model(imgsA, imgsB)
    loss, acc = criterion(codesA, codesB, labels)
    return loss.item(), acc.item()
```

7. Define the model, loss function, and optimizer:

```
model = SiameseNetwork().to(device)
criterion = ContrastiveLoss()
optimizer = optim.Adam(model.parameters(),lr = 0.001)
```

8. Train the model over increasing epochs:

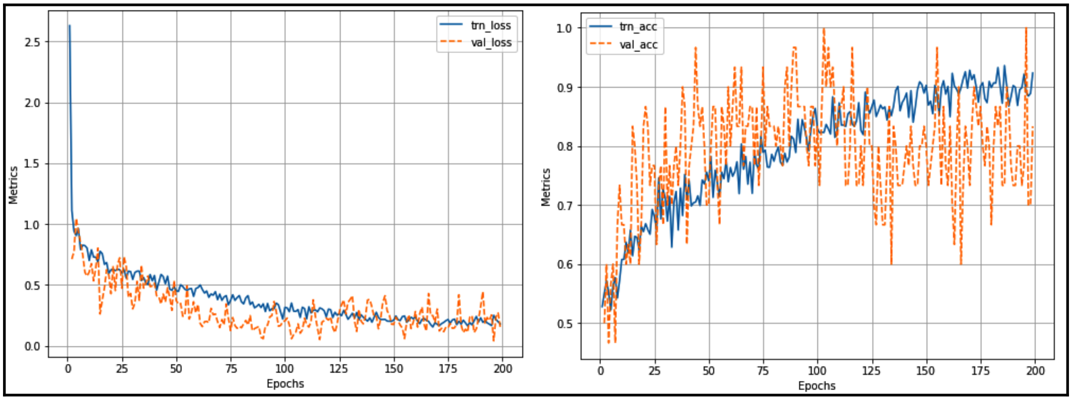
```
n_epochs = 200
log = Report(n_epochs)
for epoch in range(n_epochs):
    N = len(trn_dl)
    for i, data in enumerate(trn_dl):
        loss, acc = train_batch(model, data, optimizer, \
                                criterion)
        log.record(epoch+(1+i)/N,trn_loss=loss,trn_acc=acc, \
                   end='\r')
```

```
    N = len(val_dl)
    for i, data in enumerate(val_dl):
        loss, acc = validate_batch(model, data, \
                                   criterion)
        log.record(epoch+(1+i)/N,val_loss=loss,val_acc=acc, \
                   end='\r')
    if (epoch+1)%20==0: log.report_avgs(epoch+1)
```

- Plot the log of variation of training and validation loss accuracy over increasing epochs:

```
log.plot_epochs(['trn_loss','val_loss'])
log.plot_epochs(['trn_acc','val_acc'])
```

The preceding code results in the following output:

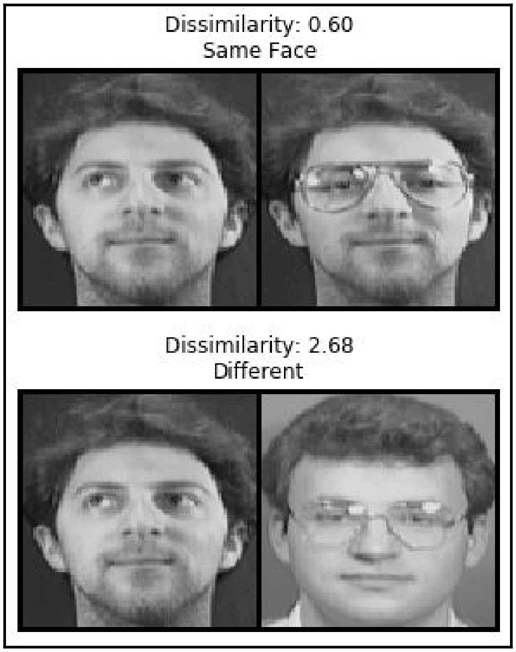


9. Test the model on new images. Note that the model has never seen these new images. While testing, we will fetch a random test image and compare it with other images in test data:

```
model.eval()
val_dl = DataLoader(val_ds,num_workers=6,batch_size=1, \
                    shuffle=True)
dataiter = iter(val_dl)
x0, _, _ = next(dataiter)

for i in range(2):
    _, x1, label2 = next(dataiter)
    concatenated = torch.cat((x0*0.5+0.5, x1*0.5+0.5),0)
    output1,output2 = model(x0.cuda(),x1.cuda())
    euclidean_distance = F.pairwise_distance(output1, output2)
    output = 'Same Face' if euclidean_distance.item() < 0.6 \
                 else 'Different'
```

```
    show(torchvision.utils.make_grid(concatenated), \
         title='Dissimilarity: {:.2f}\n{}'. \
         format(euclidean_distance.item(), output))
    plt.show()
```

The preceding results in the following output:



From the preceding, we can see that we can recognize persons in the image even when we have only a few images of a class.

TIP: In a realistic scenario (where you might use Siamese networks for attendance tracking), it would be a good idea to crop the face from the complete image before we train the model or infer it on new images.

Now that we understand how Siamese networks work, in the next sections, we will learn about other metric-based techniques – prototypical and relation networks.

Working details of prototypical networks

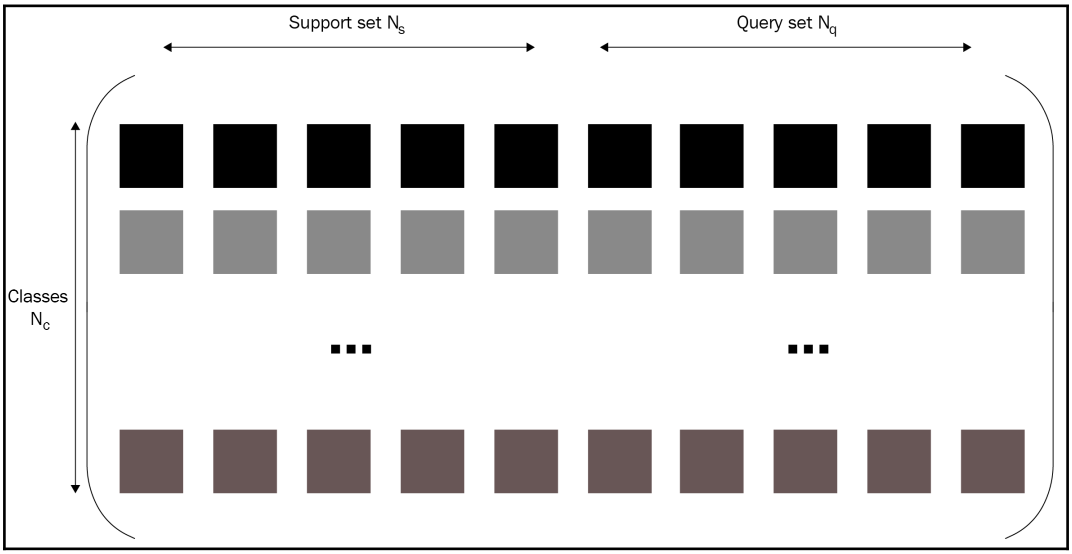
A prototype is the representative of a certain class. Imagine a scenario where we give you 10 images per class and there are 5 such classes. Prototypical networks come up with a representative embedding (a prototype) for each class by taking the average of embeddings of each image belonging to a class.

Here, let's understand a practical scenario:

Imagine you have 5 distinct classes of images with the dataset containing 10 images per class. Further, we give you 5 images per class in training and are testing your network's accuracy on the other 5 images. We will build our network with one image from each class and a randomly chosen test image as a query. Our task is to identify the class of the known image (training image) that has the highest similarity with the query image (test image).

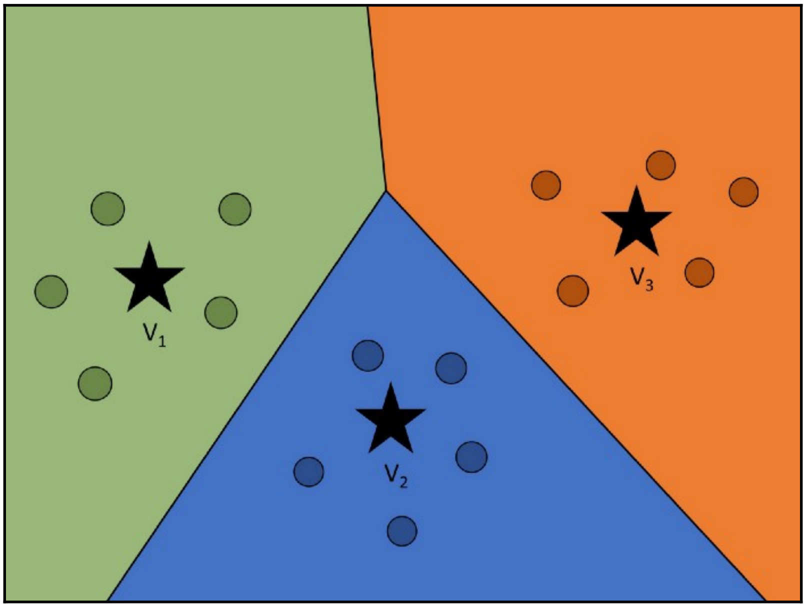
For facial recognition, the working details of prototypical networks are as follows:

- Choose N different persons at random for training.
- Choose k samples corresponding to each person as the data points available for training – this is our support set (images to compare).
- Choose q samples corresponding to each person as the data points to test – this is our query set (images to be compared):



For now, we have chosen N_c classes, with N_s images in the support set and N_q images in the query set:

- Fetch the embeddings of each data point within the support set (training images) and query set (test images) when passed through a CNN network, where we expect the CNN network to identify the index of the training image that has the highest similarity with the query image.
- Once you train the network, compute the prototype corresponding to the support set (training images) embeddings:
    - The prototype is the mean embedding of all images belonging to the same class:



In the preceding example illustration, there are three classes and each circle represents the embeddings of the images belonging to the class. Each star (prototype) is the average embedding across all the images (circles) present in the image:

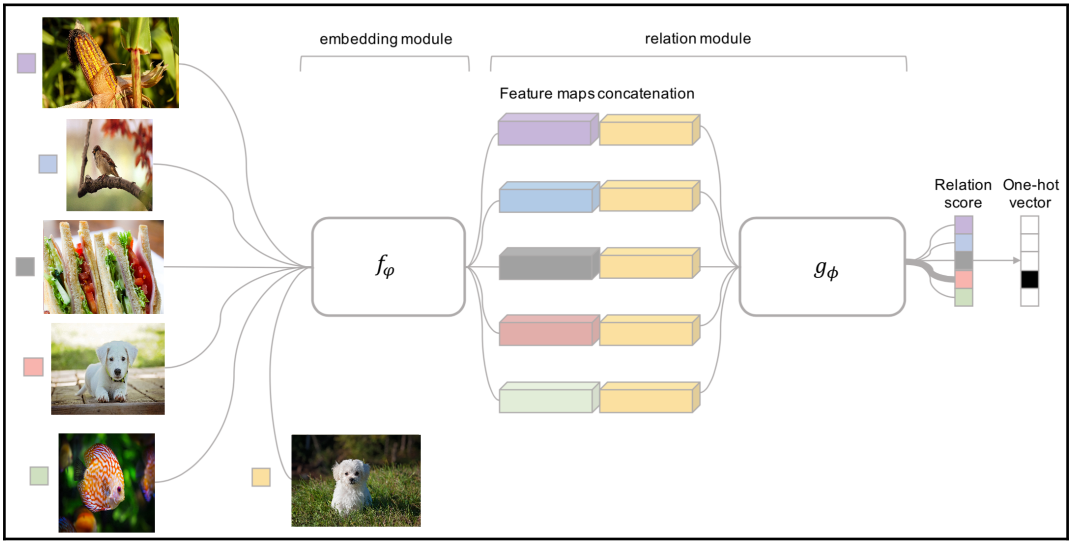
- Calculate the Euclidean distance between query embeddings and prototype embeddings:
    - If there are 5 query images and 10 classes, we will have 50 Euclidean distances.

- Perform a softmax on top of the Euclidean distances obtained earlier, to identify the probability corresponding to different support classes.
- Train a model to minimize the loss value in assigning the query image to the right class. Further, while looping over the dataset, choose a fresh set of persons at random in the next iteration.

By the end of the iterations, the model will have learned to identify the class a query image belongs to – given a few support-set images and query images.

Working details of relation networks

A relation network is fairly similar to a Siamese network, except that the metric we optimize for is not the L1 distance between embeddings but a relation score. Let's understand the working details of relation networks using the following diagram:



In the preceding diagram, the pictures on the left are the support set for five classes and the dog image at the bottom is the query image:

- Pass both the support and query images through an embedding module, which provides embeddings for the input image.
- Concatenate the feature maps of the support images with the feature maps of the query image.
- Pass the concatenated features through a CNN module to predict the relation score.

The class with the highest relation score is the predicted class of the query image.

With this, we have understood the different ways in which few-shot learning algorithms work. We compare a given query image with a support set of images to come up with the class of objects present in the support set that has the highest similarity with the query image.

Summary

In this chapter, we have learned about leveraging word vectors to come up with a way to address a scenario where the classes we want to predict are not present during training. Further, we learned about Siamese networks, which learn a distance function between two images to identify images of a similar person. Finally, we learned about prototypical networks and relation networks and how they learn to perform few-shot image classification.

In the next chapter, we will learn about combining computer vision and natural language processing-based techniques to come up with ways to solve annotating an image, detecting objects in an image, and handwriting transcription.

Questions

1. How are pre-trained word vectors obtained?
2. How do we map from an image feature embedding to a word embedding in zero-shot learning?
3. Why is the Siamese network called so?
4. How does the Siamese network come up with the similarity between two images?

15
Combining Computer Vision and NLP Techniques

In the previous chapter, we learned about leveraging novel architectures when there are a minimal number of data points. In this chapter, we will switch gears and learn about how a **Convolutional Neural Network** (**CNN**) can be used in conjunction with algorithms in the broad family of **Recurrent Neural Networks** (**RNNs**), which are heavily used (as of the time of writing this book) in **Natural Language Processing** (**NLP**) to develop solutions that leverage both computer vision and NLP.

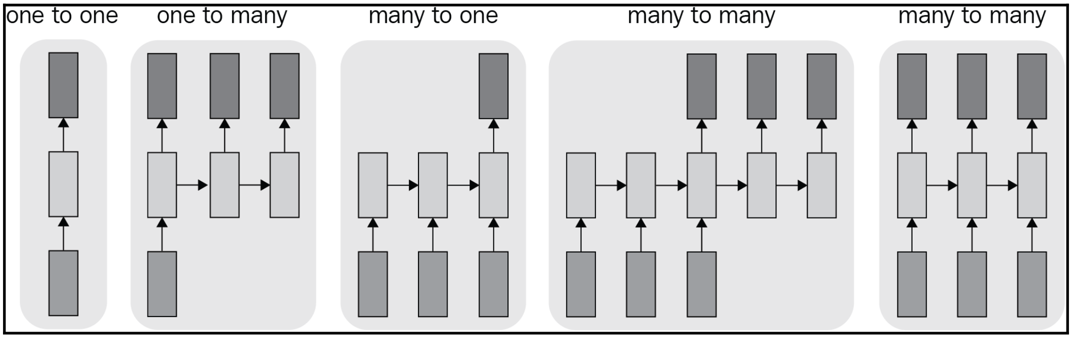
To understand combining CNNs and RNNs, we will first learn about how RNNs work and their variants – primarily **Long Short-Term Memory** (**LSTM**) – to understand how they are applied to predict annotations given an image as input. After that, we will learn about another important loss function, called the **Connectionist Temporal Classification** (**CTC**) loss function, before applying it in conjunction with a CNN and RNN to perform the transcription of handwritten images. Finally, we will learn about and leverage transformers to perform object detection using the **Detection with Transformers** (**DETR**) architecture.

By the end of this chapter, you will have learned about the following topics:

- Introducing RNNs
- Introducing LSTM architecture
- Implementing image captioning
- Transcribing handwritten images
- Object detection using DETR

Introducing RNNs

An RNN can have multiple architectures. Some of the possible ways of architecting an RNN are as follows:



In the preceding diagram, the boxes at the bottom are the input, followed by the hidden layer (the middle boxes), and then the boxes at the top are the output layer. The one-to-one architecture is a typical neural network with a hidden layer between the input and output layers. Examples of different architectures are as follows:

- **One-to-many**: The input is an image and the output is a caption of the image.
- **Many-to-one**: The input is a movie review (multiple words in input) and the output is the sentiment associated with the review.
- **Many-to-many**: Machine translation of a sentence in one language to a sentence in another language.

The idea behind the need for RNN architecture

RNNs are useful when we want to predict the next event given a sequence of events. An example of that could be to predict the word that comes after this: *This is an* ___.

Let's say that in reality, the sentence is *This is an example*.

Traditional text-mining techniques would solve the problem in the following way:

1. Encode each word while having an additional index for potential new words:

This: {1,0,0,0}

is: {0,1,0,0}

an: {0,0,1,0}

2. Encode the phrase *This is an*:

   This is an: {1,1,1,0}

3. Create the training dataset:

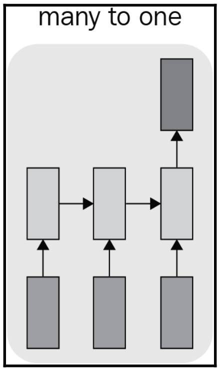
   Input --> {1,1,1,0}

   Output --> {0,0,0,1}

4. Build a model with the given input and output combination:

   One of the major drawbacks of the model is that the input representation does not change in the input sentence regardless of if it is in the form of *this is an*, *an is this*, or *this an is*.

   However, intuitively, we know that each of the preceding sentences is different and cannot be represented by the same structure mathematically. This calls for having a different architecture, which looks as follows:

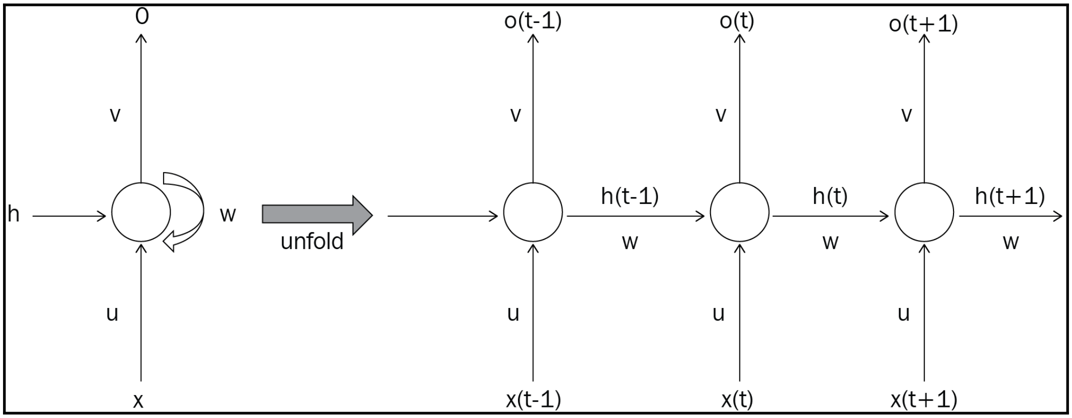


In the preceding architecture, each of the individual words from the sentence enters an individual box in the input boxes. This ensures that we preserve the structure of the input sentence; for example, *this* enters the first box, *is* enters the second box, and *an* enters the third box. The output box at the top will be the output – that is, *example*.

With an understanding of the need for RNN architecture in place, in the next section, let's learn about how to interpret the outputs of RNNs.

Exploring the structure of an RNN

You can think of an RNN as a mechanism to hold memory – where the hidden layer contains the memory. The unfolded version of an RNN is as follows:



The network on right is an unrolled version of the network on the left. The network on the right takes one input in each time step and extracts the output at each time step.

Note that while predicting the output of the third time step, we are incorporating values from the first two time steps through the hidden layer, which is connecting the values across time steps.

Let's explore the preceding diagram:

- The u weight represents the weights that connect the input layer to the hidden layer.
- The w weight represents the hidden layer to the hidden layer connection.
- The v weight represents the hidden layer to the output layer connection.

The output in a given time step depends on both the input in the current time step and the hidden layer value from the previous time step. With the introduction of the hidden layer of the previous time step being the input, along with the current time step's input, we are obtaining information from the previous time steps. This way, we are creating a pipeline of connections that enable memory storage.

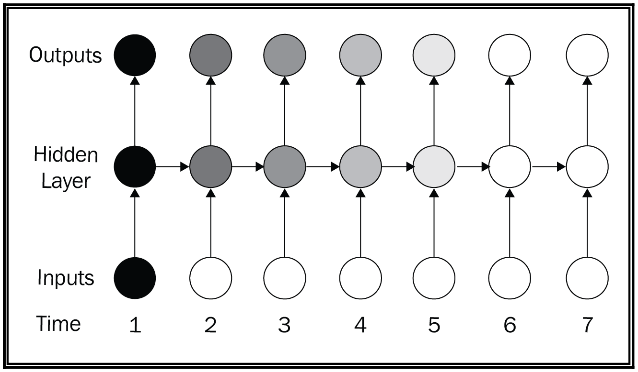
Why store memory?

There is a need to store memory as, in the preceding example, or even in text generation in general, the next word does not depend only on the preceding word, but also on the context of the words preceding the word to predict.

Given that we are looking at the preceding words, there should be a way to keep them in memory so that we can predict the next word more accurately.

We should also have the memory in order; more often than not, the recent words are more useful in predicting the next word than the words that are further away from the word to predict.

A traditional RNN that takes multiple time steps into account for giving predictions can be visualized as follows:



Notice that as the time step increases, the impact of the input present at a much earlier time step (time step 1) would be lower on the output at a much later time step (time step 7). An example of this can be seen here (for a moment, let's ignore the bias term and assume that the hidden layer input at time step 1 is `0` and we are predicting the value of the hidden layer at time step 5 – h_5):

$$
\begin{aligned}
h_5 &= WX_5 + Uh_4 \\
&= WX_5 + U(WX_4 + Uh_3) \\
&= WX_5 + U(WX_4 + U(WX_3 + Uh_2)) \\
&= WX_5 + U(WX_4 + U(WX_3 + U(WX_2 + Uh_1))) \\
&= WX_5 + U(WX_4 + U(WX_3 + U(WX_2 + U(WX_1 + h_0)))) \\
&= WX_5 + UWX_4 + U^2WX_3 + U^3WX_2 + U^4WX_1 + U^4h_0
\end{aligned}
$$

You can see that as the time step increases, the value of the hidden layer (h_5) highly depends on X_1 if U>1; however, it is much less dependent on X_1 if U<1.

The dependency on the U matrix can also result in the hidden layer (h_5) value being very small, hence resulting in a vanishing gradient when the value of U is very small, and can cause exploding gradients when the value of U is very high.

The preceding phenomenon results in an issue when there is a long-term dependency on predicting the next word. To solve this problem, we'll use the LSTM architecture.

Introducing LSTM architecture

In the previous section, we learned about how a traditional RNN faces a vanishing or exploding gradient problem resulting in it not being able to accommodate long-term memory. In this section, we will learn about how to leverage LSTM to get around this problem.

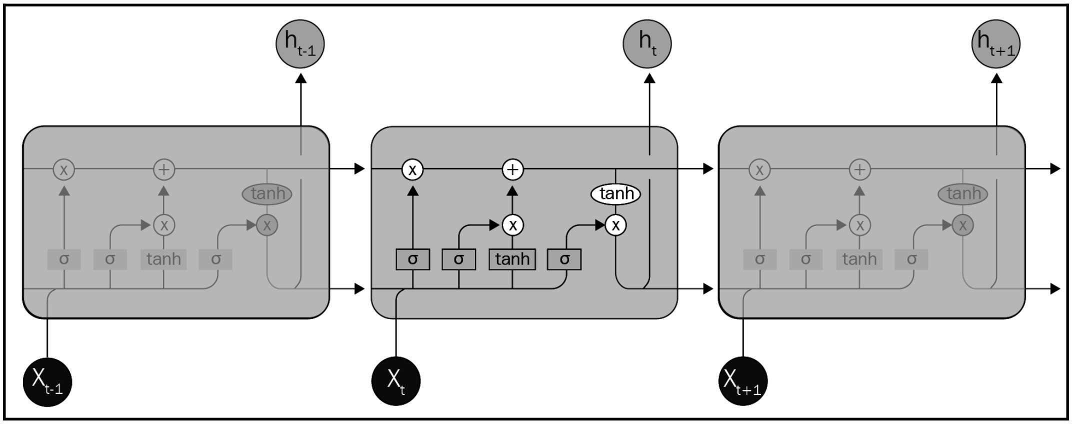
In order to further understand the scenario with an example, let's consider the following sentence:

I am from England. I speak __.

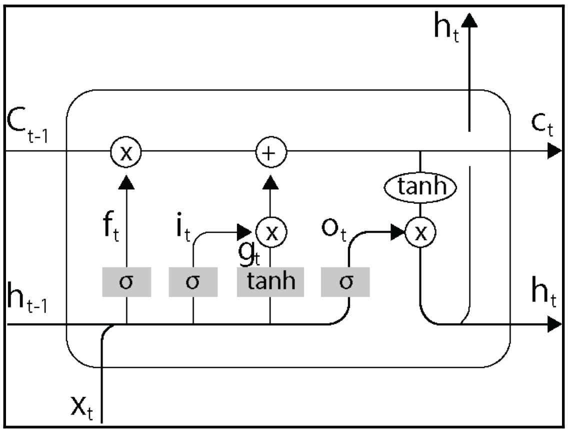
In the preceding sentence, intuitively, we know that the majority of the people from England speak English. The blank value to be filled (*English*) is obtained from the fact that the person is from England. While in this scenario we have the signaling word (England) closer to the blank value, in a realistic scenario, we might find that the signal word is far away from the blank space (the word we are trying to predict). When the distance between the signal word and blank value is large, the predictions through traditional RNNs might be wrong because of the vanishing or exploding gradient phenomenon. LSTM addresses this scenario – which we will learn about in the following section.

The working details of LSTM

A standard LSTM architecture is as follows:



In the preceding diagram, you can see that while input **X** and output **h** remain similar to what we saw in the *Exploring the structure of an RNN* section, the computations that happen between the input and output are different in LSTM. Let's understand the various activations that happen between the input and output:



In the preceding diagram, we can observe the following:

- **X** and **h** represent the input and output at time step **t**.
- **C** represents the cell state. This potentially helps in storing long-term memory.

- $\mathbf{C_{t-1}}$ is the cell state that is transferred from the previous time step.
- $\mathbf{h_{t-1}}$ represents the output of the previous time step.
- $\mathbf{f_t}$ represents activations that help with forgetting certain information.
- $\mathbf{i_t}$ represents the transformation corresponding to the input combined with the previous time step's output ($\mathbf{h_{t-1}}$).

The content that needs to be forgotten, f_t, is obtained as follows:

$$f_t = \sigma(W_{xf}\ X_t\ +\ W_{hf}\ h_{t-1}\ +\ b_f)$$

Note that W_{xf} and W_{hf} represent the weights associated with the input and the previous hidden layer, respectively.

The cell state is updated by multiplying the cell state from the previous time step, C_{t-1}, by the input content that helps in forgetting: f_t.

The updated cell state is as follows:

$$C_t\ =\ (C_{t-1} \otimes\ f)$$

Note that in the preceding step, we are performing element-to-element multiplication between C_{t-1} and f_t to obtain the modified cell state, C_t.

To understand how the preceding operations help, let's go through the input sentence: *I am from England. I speak __.*

Once we fill the blank with *English*, we no longer require the information that the person is from England and hence should erase it from memory. The combination of the cell state and forget gate helps in achieving that.

In the next step, we will include additional information from the current time step to the cell state as well as to the output. The modified cell state (after forgetting what is to be forgotten) is updated by the input activation (which is based on the current time step's input and also the previous time step's output) and the modulation gate, g_t (which helps in identifying the amount by which the cell state is to be updated).

The input activation is calculated as follows:

$$i_t = \sigma(W_{xi} X_t\ +\ W_{hi} h_{t-1}\ +\ b_i)$$

Note that W_{xi} and W_{hi} represent the weights associated with the input and the previous hidden layer, respectively.

The modified gate's activation is calculated as follows:

$$g_t = tanh(W_{xg}X_t + W_{hg}h_{t-1} + b_g)$$

Note that W_{xg} and W_{hg} represent the weights associated with the input and the previous hidden layer, respectively.

The modified gate can help in isolating the cell state values that are to be updated and not the rest, as well as identifying the magnitude of update that is to be done.

The modified cell state, C_t, which will be passed to the next time step, is now as follows:

$$C_t = (C_{t-1} \odot f_t) \oplus (i_t \odot g_t)$$

Finally, we multiply the activated updated cell state ($tanh(C_t)$) by the activated output values, O_t, to obtain the final output, h_t, at time step t:

$$O_t = \sigma(W_{xo}X_t + W_{ho}h_{t-1} + b_o)$$

$$h_t = O_t \odot tanh(C_t)$$

This way, we can leverage the various gates present in an LSTM to selectively memorize overly long time steps.

Implementing LSTM in PyTorch

In a typical text-related exercise, each word is an input to LSTM – one word per time step. To have LSTM work, we perform the following two steps:

1. Convert each word into an embedding vector.
2. Pass the embedding vector corresponding to the relevant word in the time step as input to LSTM.

Let's understand the reason why we have to convert an input word into an embedding vector. If there are 100K unique words in our vocabulary, we would have to one-hot encode them prior to passing them to the network. However, creating a one-hot-encoded vector for each word loses the semantic meaning of the word – for example, the words *like* and *enjoy* are similar and should have similar vectors. In order to address such a scenario, we leverage word embeddings, which help in learning word vector representation automatically (as they are a part of the network). Word embeddings are fetched as follows:

```
embed = nn.Embedding(vocab_size, embed_size)
```

In the preceding code, the `nn.Embedding` method takes `vocab_size` number of dimensions as input and returns `embed_size` dimensions of output. This way, if the vocabulary size is 100K and the embedding size is 128, each of the 100K words is represented as a 128-dimensional vector. One benefit of performing this exercise is that, in general, words that are similar would have similar embeddings.

Next, we pass the word embeddings through LSTM. LSTM is implemented in PyTorch using the `nn.LSTM` method, as follows:

```
hidden_state, cell_state = nn.LSTM(embed_size, \
                                   hidden_size, num_layers)
```

In the preceding code, `embed_size` represents the embedding size corresponding to each time step, `hidden_size` corresponds to the dimension of hidden layer output, and `num_layers` represents the number of times LSTM is stacked on top of each other.

Furthermore, the `nn.LSTM` method returns both the hidden state values and the cell state values.

Now that we understand the working details of LSTM and RNNs, let's understand how to leverage them in conjunction with a CNN when predicting captions given an image in the next section.

Implementing image captioning

Image captioning means generating a caption given an image. In this section, we will first learn about the preprocessing to be done to build an LSTM that can generate a text caption given an image, and then will learn how to combine a CNN and LSTM to perform image captioning. Before we learn about building a system that generates captions, let's understand how a sample input and output might look:

`In this image I can see few candles. The background is in black color.`

In the preceding example, the image is the input and the expected output is the caption of the image – **In this image I can see few candles. The background is in black color**.

The strategy that we will adopt to solve this problem is as follows:

1. Preprocess the output (ground truth annotations/captions) so that each unique word is represented by a unique ID.
2. Given that the output sentences can be of any length, let's assign a start and end token so that the model knows when to stop generating predictions. Furthermore, ensure that all input sentences are padded so that all inputs have the same length.
3. Pass the input image through a pre-trained model, such as VGG16, ResNet-18, and so on, to fetch features prior to the flattening layer.
4. Use the feature map of the image in conjunction with the text obtained in the previous step (the start token if it is the first word that we are predicting) to predict a word.
5. Repeat the preceding step until we obtain the end token.

Now that we understand what is to be done at a high level, let's implement the preceding steps in code in the next section.

Image captioning in code

Let's execute the strategy designed in the previous section in code:

The following code is available as `Image_captioning.ipynb` in the `Chapter15` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from and is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components from text.

1. Fetch the dataset from the Open Images dataset, which includes training images, their annotations, and the validation dataset:

    - Import the relevant packages, define the device, and fetch the JSON file that contains information about the images to download:

```
!pip install -qU openimages torch_snippets urllib3
!wget -O open_images_train_captions.jsonl -q
https://storage.googleapis.com/localized-narratives/annotation
s/open_images_train_v6_captions.jsonl
from torch_snippets import *
import json
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

    - Loop through the content of the JSON file and fetch the information of the first 100,000 images:

```
with open('open_images_train_captions.jsonl', 'r') as \
                                              json_file:
    json_list = json_file.read().split('\n')
np.random.shuffle(json_list)
data = []
N = 100000
for ix, json_str in Tqdm(enumerate(json_list), N):
    if ix == N: break
    try:
        result = json.loads(json_str)
        x = pd.DataFrame.from_dict(result, orient='index').T
        data.append(x)
    except:
        pass
```

A sample of the information obtained from the JSON file is as follows:

```
result

{'annotator_id': 32,
 'caption': 'In this image I can see a crocodile in the water.',
 'dataset_id': 'open_images',
 'image_id': '027963e9948e1082'}
```

From the preceding sample, we can see that `caption` and `image_id` are the key information we will use in the subsequent steps. `image_id` will be used to fetch the corresponding image and `caption` will be used to associate the output corresponding to the image obtained from a given image ID.

- Split the dataframe (`data`) into training and validation datasets:

```
np.random.seed(10)
data = pd.concat(data)
data['train'] = np.random.choice([True,False], \
                                  size=len(data),p=[0.95,0.05])
data.to_csv('data.csv', index=False)
```

- Download the images corresponding to the image IDs fetched from the JSON file:

```
from openimages.download import _download_images_by_id
!mkdir -p train-images val-images
subset_imageIds = data[data['train']].image_id.tolist()
_download_images_by_id(subset_imageIds, 'train', \
                       './train-images/')

subset_imageIds = data[~data['train']].image_id.tolist()
_download_images_by_id(subset_imageIds, 'train', \
                       './val-images/')
```

2. Create a vocabulary of all the unique words present in all the captions in the dataframe:

   - A vocabulary object is something that can map every word in all the captions to a unique integer and vice versa. We will take advantage of the `torchtext` library's `Field.build_vocab` functionality, which runs through all the words (annotations/captions) and accumulates them into two counters, `stoi` and `itos`, which are, respectively, "string to int" (a dictionary) and "int to string" (a list):

```
from torchtext.data import Field
from pycocotools.coco import COCO
from collections import defaultdict

captions = Field(sequential=False, init_token='<start>', \
                 eos_token='<end>')
all_captions = data[data['train']]['caption'].tolist()
all_tokens = [[w.lower() for w in c.split()] \
              for c in all_captions]
all_tokens = [w for sublist in all_tokens \
              for w in sublist]
captions.build_vocab(all_tokens)
```

     In the preceding code, `Field` for `captions` is a specialized object for building more complex NLP datasets in PyTorch. We cannot deal with text directly like we deal with images as strings are incompatible with tensors. So, we need to keep track of all unique occurrences of words (also called tokens), which will facilitate a one-to-one mapping of every word with a unique associated integer. For example, if the input caption is *Cat sat on the mat*, based on a mapping of words to integers, the sequence will be converted to, say, [5 23 24 4 29], where *cat* is uniquely associated with the integer 5. This mapping is typically called vocabulary, which may look like `{'<pad>': 0, '<unk'>: 1, '<start>': 2, '<end>': 3, 'the': 4, 'cat': 5, ...., 'on': 24, 'sat': 23, ... }`. The first few tokens are reserved for special functionalities, such as padding, unknown, the start of the sentence, and the end of the sentence.

   - We only need the `captions` vocabulary components, so in the following code, we create a dummy `vocab` object, which is lightweight and will have an extra `<pad>` token that was missing in `captions.vocab`:

```
class Vocab: pass
```

```
vocab = Vocab()
captions.vocab.itos.insert(0, '<pad>')
vocab.itos = captions.vocab.itos

vocab.stoi = defaultdict(lambda: \
                        captions.vocab.itos.index('<unk>'))
vocab.stoi['<pad>'] = 0
for s,i in captions.vocab.stoi.items():
    vocab.stoi[s] = i+1
```

Note that `vocab.stoi` is defined as `defaultdict` with a default function. Python uses this special dictionary to return a default value when a key does not exist. In our case, we will return an `'<unk>'` token when we try to call `vocab.stoi[<new-key/word>]`. This is handy during the validation phase where there might be some token that was not present in the training data.

3. Define the dataset class – `CaptioningDataset`:

   - Define the `__init__` method, where we provide the data frame obtained previously (`df`), the folder containing the images (`root`), `vocab`, and the image transformation pipeline (`self.transform`):

```
from torchvision import transforms
class CaptioningData(Dataset):
    def __init__(self, root, df, vocab):
        self.df = df.reset_index(drop=True)
        self.root = root
        self.vocab = vocab
        self.transform = transforms.Compose([
            transforms.Resize(224),
            transforms.RandomCrop(224),
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
            transforms.Normalize((0.485, 0.456, 0.406),
                                 (0.229, 0.224, 0.225))]
        )
```

   - Define the `__getitem__` method, where an image and its corresponding caption are fetched. Furthermore, the target is converted into a list of corresponding word IDs using `vocab`, which was built in the previous step:

```
    def __getitem__(self, index):
        """Returns one data pair (image and caption)."""
        row = self.df.iloc[index].squeeze()
```

```
        id = row.image_id
        image_path = f'{self.root}/{id}.jpg'
        image = Image.open(os.path.join(image_path))\
                                  .convert('RGB')

        caption = row.caption
        tokens = str(caption).lower().split()
        target = []
        target.append(vocab.stoi['<start>'])
        target.extend([vocab.stoi[token] for token in tokens])
        target.append(vocab.stoi['<end>'])
        target = torch.Tensor(target).long()
        return image, target, caption
```

- Define the __`choose`__ method:

```
    def choose(self):
        return self[np.random.randint(len(self))]
```

- Define the __`len`__ method:

```
    def __len__(self):
        return len(self.df)
```

- Define the `collate_fn` method to work on a batch of data:

```
    def collate_fn(self, data):
        data.sort(key=lambda x: len(x[1]), reverse=True)
        images, targets, captions = zip(*data)
        images = torch.stack([self.transform(image) \
                            for image in images], 0)
        lengths = [len(tar) for tar in targets]
        _targets = torch.zeros(len(captions), \
                             max(lengths)).long()
        for i, tar in enumerate(targets):
            end = lengths[i]
            _targets[i, :end] = tar[:end]
        return images.to(device), _targets.to(device), \
    torch.tensor(lengths).long().to(device)
```

In the `collate_fn` method, we are calculating the maximum length (the caption with the maximum number of words) of the captions in a batch and padding the rest of the captions in the batch to have the same length.

4. Define the training and validation dataset and data loaders:

```
trn_ds = CaptioningData('train-images', data[data['train']], \
                        vocab)
val_ds = CaptioningData('val-images', data[~data['train']], \
                        vocab)

image, target, caption = trn_ds.choose()
show(image, title=caption, sz=5); print(target)
```

A sample image and the corresponding caption and tokens' word indices are as follows:

```
In this image I can see few candles. The background is in black color.
tensor([   2,    6,   15,   17,   18,   11,   10,   23, 1689,    4,   19,    9,
           6,   50,   68,    3])
```

5. Create the dataloaders for the datasets:

```
trn_dl = DataLoader(trn_ds, 32, collate_fn=trn_ds.collate_fn)
val_dl = DataLoader(val_ds, 32, collate_fn=val_ds.collate_fn)
inspect(*next(iter(trn_dl)), names='images,targets,lengths')
```

A sample batch would have the following entities:

```
==================================================================
IMAGES:
Tensor  Shape: torch.Size([32, 3, 224, 224])    Min: -2.118     Max: 2.640      Mean: 0.026     dtype: torch.float32
==================================================================
TARGETS:
Tensor  Shape: torch.Size([32, 65])     Min: 0.000      Max: 12523.000  Mean: 105.936   dtype: torch.int64
==================================================================
LENGTHS:
Tensor  Shape: torch.Size([32]) Min: 12.000     Max: 65.000     Mean: 36.156    dtype: torch.int64
==================================================================
```

6. Define the network class:

    - Define the encoder architecture – `EncoderCNN`:

```
from torch.nn.utils.rnn import pack_padded_sequence
from torchvision import models
class EncoderCNN(nn.Module):
    def __init__(self, embed_size):
        """Load the pretrained ResNet-152 and replace
        top fc layer."""
        super(EncoderCNN, self).__init__()
        resnet = models.resnet152(pretrained=True)
        # delete the last fc layer.
        modules = list(resnet.children())[:-1]
        self.resnet = nn.Sequential(*modules)
        self.linear = nn.Linear(resnet.fc.in_features, \
                                embed_size)
        self.bn = nn.BatchNorm1d(embed_size, \
                                 momentum=0.01)
    def forward(self, images):
        """Extract feature vectors from input images."""
        with torch.no_grad():
            features = self.resnet(images)
        features = features.reshape(features.size(0), -1)
        features = self.bn(self.linear(features))
        return features
```

    In the preceding code, we are fetching the pre-trained ResNet-152 model, deleting the last `fc` layer, connecting it to a `Linear` layer of size `embed_size`, and then passing it through batch normalization (`bn`).

    - Fetch a summary of the `encoder` class:

```
encoder = EncoderCNN(256).to(device)
!pip install torch_summary
from torchsummary import summary
print(summary(encoder,torch.zeros(32,3,224,224).to(device)))
```

The preceding code gives the following output:

```
        Layer (type)               Output Shape         Param #     Tr. Param #
================================================================================
              Conv2d-1        [32, 64, 112, 112]           9,408           9,408
         BatchNorm2d-2        [32, 64, 112, 112]             128             128
                ReLU-3        [32, 64, 112, 112]               0               0
           MaxPool2d-4          [32, 64, 56, 56]               0               0
          Bottleneck-5         [32, 256, 56, 56]          75,008          75,008
          Bottleneck-6         [32, 256, 56, 56]          70,400          70,400
          Bottleneck-7         [32, 256, 56, 56]          70,400          70,400
          Bottleneck-8         [32, 512, 28, 28]         379,392         379,392
          Bottleneck-9         [32, 512, 28, 28]         280,064         280,064
         Bottleneck-10         [32, 512, 28, 28]         280,064         280,064
         Bottleneck-11         [32, 512, 28, 28]         280,064         280,064
         Bottleneck-12         [32, 512, 28, 28]         280,064         280,064
         Bottleneck-13         [32, 512, 28, 28]         280,064         280,064
         Bottleneck-14         [32, 512, 28, 28]         280,064         280,064
         Bottleneck-15         [32, 512, 28, 28]         280,064         280,064
         Bottleneck-16        [32, 1024, 14, 14]       1,512,448       1,512,448
         Bottleneck-17        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-18        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-19        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-20        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-21        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-22        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-23        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-24        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-25        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-26        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-27        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-28        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-29        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-30        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-31        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-32        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-33        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-34        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-35        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-36        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-37        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-38        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-39        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-40        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-41        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-42        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-43        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-44        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-45        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-46        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-47        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-48        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-49        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-50        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-51        [32, 1024, 14, 14]       1,117,184       1,117,184
         Bottleneck-52          [32, 2048, 7, 7]       6,039,552       6,039,552
         Bottleneck-53          [32, 2048, 7, 7]       4,462,592       4,462,592
         Bottleneck-54          [32, 2048, 7, 7]       4,462,592       4,462,592
 AdaptiveAvgPool2d-55          [32, 2048, 1, 1]               0               0
             Linear-56                 [32, 256]         524,544         524,544
        BatchNorm1d-57                 [32, 256]             512             512
================================================================================
Total params: 58,668,864
Trainable params: 58,668,864
Non-trainable params: 0
--------------------------------------------------------------------------------
```

- Define the decoder architecture – `DecoderRNN`:

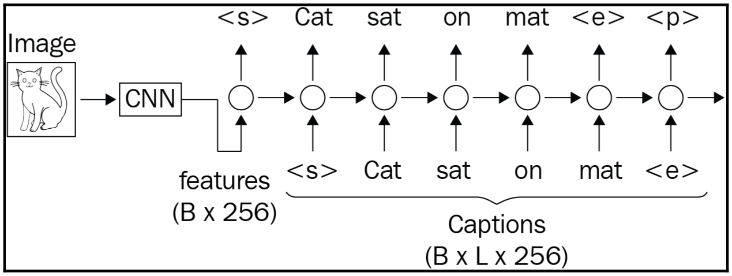
```
class DecoderRNN(nn.Module):
    def __init__(self, embed_size, hidden_size, vocab_size, \
                 num_layers, max_seq_length=80):
        """Set the hyper-parameters and build the layers."""
        super(DecoderRNN, self).__init__()
        self.embed = nn.Embedding(vocab_size, embed_size)
        self.lstm = nn.LSTM(embed_size, hidden_size, \
                            num_layers, batch_first=True)
        self.linear = nn.Linear(hidden_size, vocab_size)
        self.max_seq_length = max_seq_length
    def forward(self, features, captions, lengths):
        """Decode image feature vectors and
        generates captions."""
        embeddings = self.embed(captions)
        embeddings = torch.cat((features.unsqueeze(1), \
                                embeddings), 1)
        packed = pack_padded_sequence(embeddings, \
                            lengths.cpu(), batch_first=True)
        outputs, _ = self.lstm(packed)
        outputs = self.linear(outputs[0])
        return outputs
```

In the preceding decoder, let's understand what we are initializing:

- `self.embed`: A `vocab x embed_size` matrix that creates and learns a unique embedding for each word.
- `self.lstm` takes the output of `CNNEncoder` and the previous time step's word output embedding as input and returns a hidden state for each time step.
- `self.linear` converts each hidden state into a `V`-dimensional vector that we'll use softmax on, to get the likely word for a time step.

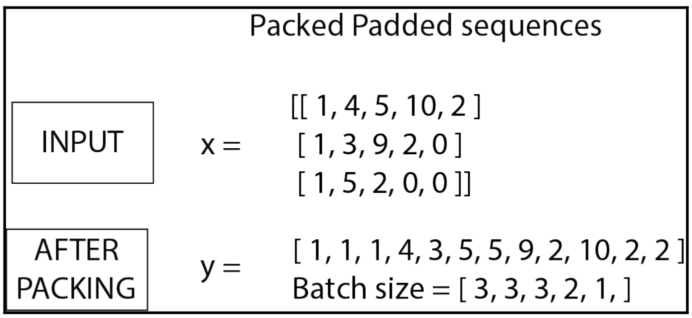
In the `forward` method, we see the following:

1. The captions (which are sent as integers) are converted into embedding using `self.embed`.
2. `features` from `EncoderCNN` is concatenated to `embeddings`. If the number of time steps (*L* in the following example) per caption is, say, 80, after concatenation, the number of time steps is going to be 81. See the following example for what is being fed and predicted in each time step:

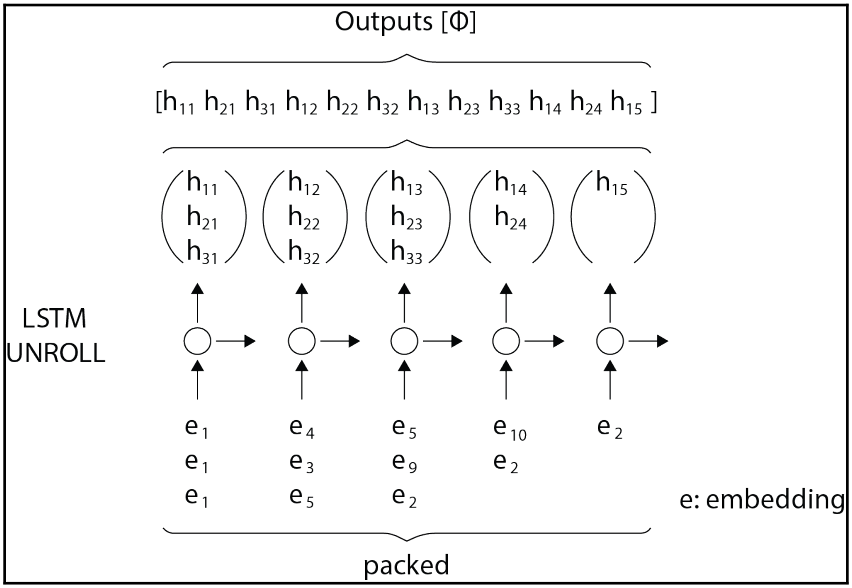


3. Using `pack_padded_sequences`, the concatenated embeddings are packed into a data structure that lets RNN computations be more efficient by not unrolling at time steps where padding is present. See the following diagram for an intuitive explanation:

    - In the following diagram, we have three sentences that are encoded with their corresponding word indices. A word index of `0` represents the padding index. After packing, the batch size is `1` in the last index as there is only one sentence where the last index in the sentence is not the padding index:



- The packed padding is now passed to LSTM, as follows:



The corresponding line for the previous illustration in code is `outputs, _ = self.lstm(packed)`. Finally, the outputs from LSTM are sent through a linear layer so that the number of dimensions changes from 512 to vocab size.

We will also add a `predict` method to the RNN that accepts features from `EncoderCNN` and returns the expected tokens for each feature. We will use this after training, to obtain captions on an image:

```
def predict(self, features, states=None):
    """Generate captions for given image
    features using greedy search."""
    sampled_ids = []
    inputs = features.unsqueeze(1)
    for i in range(self.max_seq_length):
        hiddens, states = self.lstm(inputs, states)
        # hiddens: (batch_size, 1, hidden_size)
        outputs = self.linear(hiddens.squeeze(1))
        # outputs: (batch_size, vocab_size)
        _, predicted = outputs.max(1)
        # predicted: (batch_size)
        sampled_ids.append(predicted)
```

```
            inputs = self.embed(predicted)
            # inputs: (batch_size, embed_size)
            inputs = inputs.unsqueeze(1)
            # inputs: (batch_size, 1, embed_size)

        sampled_ids = torch.stack(sampled_ids, 1)
        # sampled_ids: (batch_size, max_seq_length)
        # convert predicted tokens to strings
        sentences = []
        for sampled_id in sampled_ids:
            sampled_id = sampled_id.cpu().numpy()
            sampled_caption = []
            for word_id in sampled_id:
                word = vocab.itos[word_id]
                sampled_caption.append(word)
                if word == '<end>':
                    break
            sentence = ' '.join(sampled_caption)
            sentences.append(sentence)
        return sentences
```

7. Define the functions to train on a batch of data:

```
def train_batch(data, encoder, decoder, optimizer, criterion):
    encoder.train()
    decoder.train()
    images, captions, lengths = data
    images = images.to(device)
    captions = captions.to(device)
    targets = pack_padded_sequence(captions, lengths.cpu(), \
                                   batch_first=True)[0]
    features = encoder(images)
    outputs = decoder(features, captions, lengths)
    loss = criterion(outputs, targets)
    decoder.zero_grad()
    encoder.zero_grad()
    loss.backward()
    optimizer.step()
    return loss
```

Note that we create a tensor called `targets` from this, which has items packed into a vector. As you know from the previous diagram, `pack_padded_sequence` helps to pack the predictions in such a way that it is easier to call `nn.CrossEntropyLoss` on the output with the packed `target` values.

8. Define the function to validate on a batch of data:

```
@torch.no_grad()
def validate_batch(data, encoder, decoder, criterion):
    encoder.eval()
    decoder.eval()
    images, captions, lengths = data
    images = images.to(device)
    captions = captions.to(device)
    targets = pack_padded_sequence(captions, lengths.cpu(), \
                                   batch_first=True)[0]
    features = encoder(images)
    outputs = decoder(features, captions, lengths)
    loss = criterion(outputs, targets)
    return loss
```

9. Define the model objects and the loss function, and optimizer:

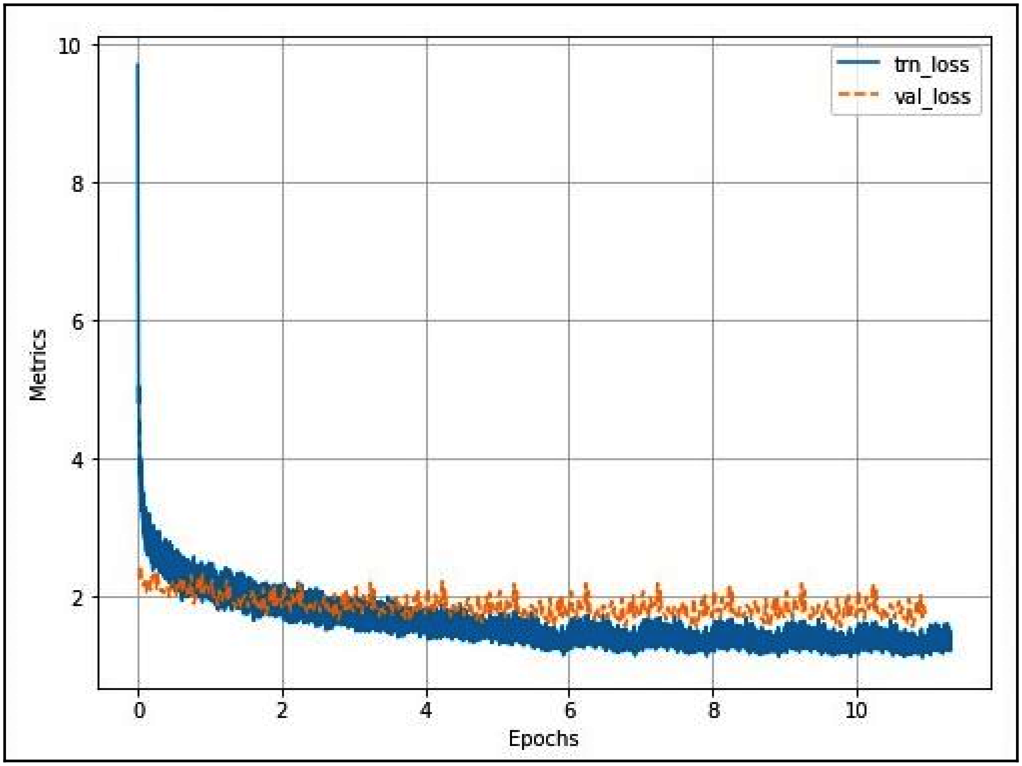
```
encoder = EncoderCNN(256).to(device)
decoder = DecoderRNN(256, 512, len(vocab.itos), 1).to(device)
criterion = nn.CrossEntropyLoss()
params = list(decoder.parameters()) + \
         list(encoder.linear.parameters()) + \
         list(encoder.bn.parameters())
optimizer = torch.optim.AdamW(params, lr=1e-3)
n_epochs = 10
log = Report(n_epochs)
```

10. Train the model over increasing epochs:

```
for epoch in range(n_epochs):
    if epoch == 5: optimizer = torch.optim.AdamW(params, \
                                                  lr=1e-4)
    N = len(trn_dl)
    for i, data in enumerate(trn_dl):
        trn_loss = train_batch(data, encoder, decoder, \
                               optimizer, criterion)
        pos = epoch + (1+i)/N
        log.record(pos=pos, trn_loss=trn_loss, end='\r')

    N = len(val_dl)
    for i, data in enumerate(val_dl):
        val_loss = validate_batch(data, encoder, decoder, \
                                  criterion)
        pos = epoch + (1+i)/N
        log.record(pos=pos, val_loss=val_loss, end='\r')
```

```
        log.report_avgs(epoch+1)

    log.plot_epochs(log=True)
```

The preceding code generates an output of variation of the training and validation loss over increasing epochs:



11. Define a function that generates predictions given an image:

```
def load_image(image_path, transform=None):
    image = Image.open(image_path).convert('RGB')
    image = image.resize([224, 224], Image.LANCZOS)
    if transform is not None:
        tfm_image = transform(image)[None]
    return image, tfm_image

def load_image_and_predict(image_path):
    transform = transforms.Compose([
                    transforms.ToTensor(),
                    transforms.Normalize(\
```

```
                                (0.485, 0.456, 0.406),
                                (0.229, 0.224, 0.225))
                            ])
        org_image, tfm_image = load_image(image_path, transform)
        image_tensor = tfm_image.to(device)
        encoder.eval()
        decoder.eval()
        feature = encoder(image_tensor)
        sentence = decoder.predict(feature)[0]
        show(org_image, title=sentence)
        return sentence

    files = Glob('val-images')
    load_image_and_predict(choose(files))
```

The preceding generates predictions given an image:

<start> in this image we can see a few boats on the water, in the background we can see the sky with clouds. <end>

From the preceding, we can see that we can generate reasonable captions given an image (which is presented as the title in the preceding example).

In this section, we learned about leveraging a CNN and RNN together to generate captions. In the next section, we will learn about using CNNs, RNNs, and CTC loss functions to transcribe images containing handwritten words.

Transcribing handwritten images

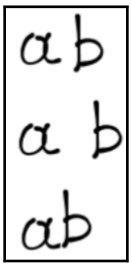
In the previous section, we learned about generating sequences of words from an input image. In this section, we will learn about generating sequences of characters with the image as input. Furthermore, we will learn about the CTC loss function, which helps in transcribing handwritten images.

Before we learn about the CTC loss function, let's understand the reason why the architecture that we saw in the image captioning section might not apply in handwritten transcription. Unlike in image captioning, where there is no straightforward correlation between the content in the image and the output words, in a handwritten image, there is a direct correlation between the sequence of characters present in the image and the sequence of output. Thus, we will follow a different architecture from what we designed in the previous section.

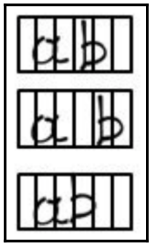
In addition, assume a scenario where an image is divided into 20 portions (assuming a scenario of a maximum of 20 characters per word in an image), where each portion corresponds to a character. One person's handwriting might ensure that each character perfectly fits into a box and another's handwriting might be mixed up such that each box contains two characters, and another where the spacing between two characters is so large that it is not possible to fit a word into 20 time steps (portions). This calls for a different way of solving this problem, which leverages the CTC loss function – which we will learn about in the next section.

The working details of CTC loss

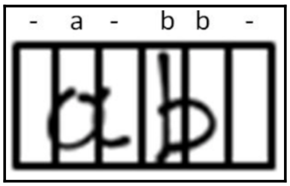
Imagine a scenario where we are transcribing an image that contains the word *ab*. The image could look like any of the following and the output is always *ab*, irrespective of which of the following three images we choose:



In the next step, we divide the preceding three examples into six time steps, as follows (where each box represents a time step):



Now, we'll predict the output character in each time step – where the output is the softmax of probabilities of words present within a vocabulary. Given that we are performing softmax, let's say the output character at each time step after running the image through our model (which we will define in the subsequent section) is as in the following (the output of each cell is provided above the image):



Note that – represents that nothing is present in the corresponding time step. Furthermore, note that the character *b* is repeated in two different time steps.

In the final step, we will squash the output (a sequence of characters) that is obtained by passing our image through the model in such a way that consecutive repeating characters are squashed into one.

The preceding step of squashing repeating characters' output if there are consecutive same-character predictions results in a final output as follows:

-a-b-

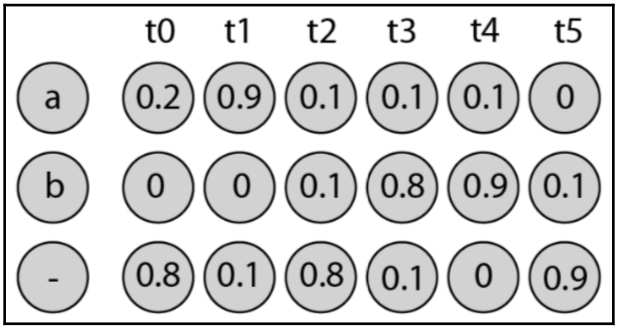
In another case, where the output is *abb*, the final output post squashing is expected to have a separator between the two *b* characters, an example of which is as follows:

-a-b-b-

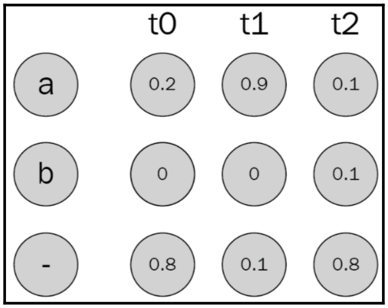
Now that we understand the concept of how the input and output values look, in the next section, let's learn about how we calculate the CTC loss value.

Calculating the CTC loss value

For the problem we are solving in the previous section, let's consider the following scenario – the probability of having the character in a given time step is provided in the circles in the following diagram (note that the probabilities add up to 1 in each time step from *t0* to *t5*):



However, to keep the calculation simple, for us to understand how the CTC loss value is calculated, let's take a scenario where the image contains only the character *a* and not the word *ab*. Furthermore, we'll assume that there are only **three time steps** for simplicity of calculation:



We can obtain the ground truth of *a* if the softmax in each time step is like any of the following seven scenarios:

Output in each time step	Prob. of character in t_0	Prob. of character in t_1	Prob. of character in t_2	Probability of combination	Final probability
--a	0.8	0.1	0.1	0.8 x 0.1 x 0.1	0.008
-aa	0.8	0.9	0.1	0.8 x 0.9 x 0.1	0.072
aaa	0.2	0.9	0.1	0.2 x 0.9 x 0.1	0.018

-a-	0.8	0.9	0.8	0.8 x 0.9 x 0.8	0.576
-aa	0.8	0.9	0.1	0.8 x 0.9 x 0.1	0.072
a--	0.2	0.1	0.8	0.2 x 0.1 x 0.8	0.016
aa-	0.2	0.9	0.8	0.2 x 0.9 x 0.8	0.144
				Overall probability	0.906

From the preceding results, we can see that the overall probability of obtaining the ground truth is 0.906.

The rest of the 0.094 corresponds to the probability of the outcome not obtaining the ground truth.

Let's calculate the binary cross-entropy loss corresponding to the summation of all the possible ground truths.

CTC loss is the negative logarithm of the overall probability summation of combinations that result in ground truth = -log(0.906) = 0.1.

Now that we understand how the CTC loss is calculated, let's implement this knowledge while building a model for handwriting transcription from an image in the next section.

Handwriting transcription in code

The strategy we will adopt to code up a network that can transcribe the content of an image of a handwritten word is as follows:

1. Import the dataset of images and their corresponding transcriptions.
2. Give an index to each character.
3. Pass the image through a convolutional network to fetch the feature map corresponding to the image.
4. Pass the feature maps through an RNN.
5. Fetch the probabilities in each time step.
6. Leverage the CTC loss function to squash outputs and provide transcriptions and the corresponding loss.
7. The weights of the network are optimized by minimizing the CTC loss function.

Let's execute the preceding strategy in code:

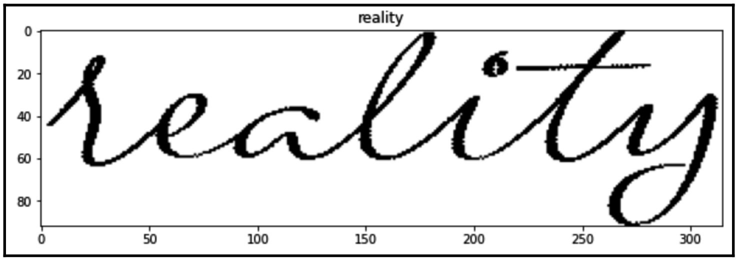
The following code is available as `Handwriting_transcription.ipynb` in the `Chapter15` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`.

1. Download and import the dataset of images:

```
!wget
https://www.dropbox.com/s/l2ul3upj7dkv4ou/synthetic-data.zip
!unzip -qq synthetic-data.zip
```

In the preceding code, we have downloaded the dataset where the images are provided and the filename of the image contains the ground truth of transcription corresponding to that image.

A sample from the images that were downloaded is as follows:



2. Install the required packages and import them:

```
!pip install torch_snippets torch_summary editdistance
```

- Import the packages:

```
from torch_snippets import *
from torchsummary import summary
import editdistance
```

3. Specify the location of the images and the function to fetch the ground truth from the images:

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
fname2label = lambda fname: stem(fname).split('@')[0]
images = Glob('synthetic-data')
```

Note that we are creating the `fname2label` function as the ground truth of an image is available after the `@` symbol in the filename. A sample of the filenames is as follows:

```
['synthetic-data/already@dPkHBT.png',
 'synthetic-data/bring@OzNTFr.png',
 'synthetic-data/few@EhVOvU.png',
 'synthetic-data/research@cgo7rI.png',
 'synthetic-data/fast@bQ8Wkm.png']
```

4. Define the vocabulary of characters (`vocab`), the batch size (`B`), the time steps of the RNN (`T`), the length of the vocabulary (`V`), the height (`H`), and the width (`W`) of the images:

```
vocab='QWERTYUIOPASDFGHJKLZXCVBNMqwertyuiopasdfghjklzxcvbnm'
B,T,V = 64, 32, len(vocab)
H,W = 32, 128
```

5. Define the `OCRDataset` dataset class:

   - Define the `__init__` method where we specify the mapping of character to character ID (`charList`) and the other way around (`invCharList`) by looping through `vocab`, as well as the number of time steps (`timesteps`) and the file paths of the images (`items`) that are to be fetched. We are using `charList` and `invCharList` instead of using `torchtext`'s `build` vocab here as the vocabulary is simpler to handle (contains fewer number of distinct characters):

```
class OCRDataset(Dataset):
    def __init__(self, items, vocab=vocab, \
                 preprocess_shape=(H,W), timesteps=T):
        super().__init__()
        self.items = items
        self.charList = {ix+1:ch for ix,ch \
                         in enumerate(vocab)}
        self.charList.update({0: '`'})
        self.invCharList = {v:k for k,v in \
                            self.charList.items()}
        self.ts = timesteps
```

   - Define the `__len__` and `__getitem__` methods:

```
    def __len__(self):
        return len(self.items)
```

```
    def sample(self):
        return self[randint(len(self))]
    def __getitem__(self, ix):
        item = self.items[ix]
        image = cv2.imread(item, 0)
        label = fname2label(item)
        return image, label
```

Note that in the `__getitem__` method, we are reading the image and creating the label using `fname2label`, which we defined earlier. In addition, we are defining a `sample` method that helps us in randomly sampling an image from the dataset.

- Define the `collate_fn` method, which takes a batch of images and appends them and their labels in different lists. Furthermore, it converts the characters of the ground truth corresponding to an image in their vector format (which converts each character into its corresponding ID) and finally, stores the label length and the input length (which is always the number of time steps) for every image. The label length and input length are leveraged by the CTC loss function while calculating the loss value:

```
    def collate_fn(self, batch):
        images, labels, label_lengths = [], [], []
        label_vectors, input_lengths = [], []
        for image, label in batch:
            images.append(torch.Tensor(self.\
                                   preprocess(image))[None,None])
            label_lengths.append(len(label))
            labels.append(label)
            label_vectors.append(self.str2vec(label))
            input_lengths.append(self.ts)
```

- Convert each of the preceding lists into a Torch tensor object and return `images`, `labels`, `label_lengths`, `label_vectors`, and `input_lengths`:

```
        images = torch.cat(images).float().to(device)
        label_lengths = torch.Tensor(label_lengths)\
                             .long().to(device)
        label_vectors = torch.Tensor(label_vectors)\
                             .long().to(device)
        input_lengths = torch.Tensor(input_lengths)\
                             .long().to(device)
        return images, label_vectors, label_lengths, \
                input_lengths, labels
```

- Define the `str2vec` function, which converts an input of character IDs into a string:

```
def str2vec(self, string, pad=True):
    string = ''.join([s for s in string if \
                      s in self.invCharList])
    val = list(map(lambda x: self.invCharList[x], \
                   string))
    if pad:
        while len(val) < self.ts:
            val.append(0)
    return val
```

In the `str2vec` function, we are fetching the characters from a string of character IDs and appending the vectors with a pad index of `0` if the length of the labels (`len(val)`) is less than the number of time steps (`self.ts`).

- Define the `preprocess` function, which takes an image (`img`) and `shape` as input to process it to a consistent shape of 32 x 128. Note that additional preprocessing is to be done other than resizing the image as images are to be resized while maintaining the aspect ratio.

Define the `preprocess` function and the target shape of the image, which for now is initialized as a blank image (white image – `target`):

```
def preprocess(self, img, shape=(32,128)):
    target = np.ones(shape)*255
```

Fetch the shape and the expected shape of the image:

```
    try:
        H, W = shape
        h, w = img.shape
```

Calculate how the image is to be resized to maintain the aspect ratio:

```
        fx = H/h
        fy = W/w
        f = min(fx, fy)
        _h = int(h*f)
        _w = int(w*f)
```

Resize the image and store it in the target variable defined earlier:

```
        _img = cv2.resize(img, (_w,_h))
        target[:_h,:_w] = _img
```

Return the normalized image (where we first convert the image to have a black background and then scale the pixels to a value between 0 and 1):

```
        except:
            ...
        return (255-target)/255
```

- Define the `decoder_chars` function to decode predictions into words:

```
    def decoder_chars(self, pred):
        decoded = ""
        last = ""
        pred = pred.cpu().detach().numpy()
        for i in range(len(pred)):
            k = np.argmax(pred[i])
            if k > 0 and self.charList[k] != last:
                last = self.charList[k]
                decoded = decoded + last
            elif k > 0 and self.charList[k] == last:
                continue
            else:
                last = ""
        return decoded.replace(" "," ")
```

In the preceding code, we are looping through the predictions (`pred`) one time step at a time, fetching the character that has the highest confidence (`k`), comparing that with the character that has the highest confidence in the previous time step (`last`), and appending it to the `decoded` characters so far if the character with the highest confidence in the previous time step is not the same as the character with the highest confidence in the current time step (equivalent to squashing, which we discussed in the CTC loss function section).

- Define the methods to calculate the character and word accuracies:

```
    def wer(self, preds, labels):
        c = 0
        for p, l in zip(preds, labels):
            c += p.lower().strip() != l.lower().strip()
        return round(c/len(preds), 4)
    def cer(self, preds, labels):
        c, d = [], []
        for p, l in zip(preds, labels):
            c.append(editdistance.eval(p, l) / len(l))
        return round(np.mean(c), 4)
```

- Define a method to evaluate the model on a set of images and return the word and character error rate:

```
def evaluate(self, model, ims, labels, lower=False):
    model.eval()
    preds = model(ims).permute(1,0,2) # B, T, V+1
    preds = [self.decoder_chars(pred) for pred in preds]
    return {'char-error-rate': self.cer(preds, labels), \
            'word-error-rate': self.wer(preds, labels), \
            'char-accuracy': 1-self.cer(preds, labels), \
            'word-accuracy' : 1-self.wer(preds, labels)}
```

In the preceding code, we are permuting channels of the input images so that we have the data preprocessed as expected by the model, decoding the predictions using the `decoder_chars` function and then returning the character error rate, word error rate, and their corresponding accuracies.

6. Specify the training and validation datasets and the dataloaders:

```
from sklearn.model_selection import train_test_split
trn_items,val_items=train_test_split(Glob('synthetic-data'), \
                              test_size=0.2, random_state=22)
trn_ds = OCRDataset(trn_items)
val_ds = OCRDataset(val_items)

trn_dl = DataLoader(trn_ds, batch_size=B, \
                    collate_fn=trn_ds.collate_fn, \
                    drop_last=True, shuffle=True)
val_dl = DataLoader(val_ds, batch_size=B, \
                collate_fn=val_ds.collate_fn, drop_last=True)
```

7. Build the network architecture:

- Build the basic blocks of a CNN:

```
from torch_snippets import Reshape, Permute
class BasicBlock(nn.Module):
    def __init__(self, ni, no, ks=3, st=1, \
                 padding=1, pool=2, drop=0.2):
        super().__init__()
        self.ks = ks
        self.block = nn.Sequential(
            nn.Conv2d(ni, no, kernel_size=ks, \
                      stride=st, padding=padding),
            nn.BatchNorm2d(no, momentum=0.3),
            nn.ReLU(inplace=True),
```

```
            nn.MaxPool2d(pool),
            nn.Dropout2d(drop)
        )
    def forward(self, x):
        return self.block(x)
```

- Build the neural network class OCR that has the CNN blocks and RNN blocks defined in the `__init__` method in `self.model` and `self.rnn`, respectively. Next, we define the `self.classification` layer, which takes the output of an RNN and passes it through a softmax activation after processing the RNN output through a dense layer:

```
class Ocr(nn.Module):
    def __init__(self, vocab):
        super().__init__()
        self.model = nn.Sequential(
                    BasicBlock( 1, 128),
                    BasicBlock(128, 128),
                    BasicBlock(128, 256, pool=(4,2)),
                    Reshape(-1, 256, 32),
                    Permute(2, 0, 1) # T, B, D
                )
        self.rnn = nn.Sequential(
            nn.LSTM(256, 256, num_layers=2, \
                    dropout=0.2, bidirectional=True),
        )
        self.classification = nn.Sequential(
            nn.Linear(512, vocab+1),
            nn.LogSoftmax(-1),
        )
```

- Define the `forward` method:

```
    def forward(self, x):
        x = self.model(x)
        x, lstm_states = self.rnn(x)
        y = self.classification(x)
        return y
```

In the preceding code, we are fetching the CNN output in the first step and then passing it through the RNN to fetch `lstm_states` and the RNN output `x`, before finally passing the output through the classification layer (`self.classification`) and returning it.

- Define the CTC loss function:

```
def ctc(log_probs, target, input_lengths, \
        target_lengths, blank=0):
    loss = nn.CTCLoss(blank=blank, zero_infinity=True)
    ctc_loss = loss(log_probs, target, \
                    input_lengths, target_lengths)
    return ctc_loss
```

In the preceding code, we are leveraging the `nn.CTCLoss` method to minimize `ctc_loss`, which takes the confidence matrix, `log_probs` (predictions in each time step), `target` (ground truth), `input_lengths`, and `target_lengths` as input to return the `ctc_loss` value.

- Fetch a summary of the defined model:

```
model = Ocr(len(vocab)).to(device)
summary(model, torch.zeros((1,1,32,128)).to(device))
```

The preceding code results in the following output:

```
--------------------------------------------------------------------------------------------
     Layer (type)                                 Output Shape         Param #     Tr. Param #
============================================================================================
        Conv2d-1                           [1, 128, 32, 128]           1,280           1,280
   BatchNorm2d-2                           [1, 128, 32, 128]             256             256
          ReLU-3                           [1, 128, 32, 128]               0               0
     MaxPool2d-4                            [1, 128, 16, 64]               0               0
     Dropout2d-5                            [1, 128, 16, 64]               0               0
        Conv2d-6                            [1, 128, 16, 64]         147,584         147,584
   BatchNorm2d-7                            [1, 128, 16, 64]             256             256
          ReLU-8                            [1, 128, 16, 64]               0               0
     MaxPool2d-9                             [1, 128, 8, 32]               0               0
   Dropout2d-10                             [1, 128, 8, 32]               0               0
      Conv2d-11                             [1, 256, 8, 32]         295,168         295,168
 BatchNorm2d-12                             [1, 256, 8, 32]             512             512
        ReLU-13                             [1, 256, 8, 32]               0               0
   MaxPool2d-14                             [1, 256, 2, 16]               0               0
   Dropout2d-15                             [1, 256, 2, 16]               0               0
     Reshape-16                                [1, 256, 32]               0               0
     Permute-17                                [32, 1, 256]               0               0
        LSTM-18     [32, 1, 512], [4, 1, 256], [4, 1, 256]       2,629,632       2,629,632
      Linear-19                                 [32, 1, 53]          27,189          27,189
  LogSoftmax-20                                 [32, 1, 53]               0               0
============================================================================================
Total params: 3,101,877
Trainable params: 3,101,877
Non-trainable params: 0
--------------------------------------------------------------------------------------------
```

Note that the output has 53 probabilities associated with each image in the batch as there is a vocabulary of 53 characters (26 x 2 = 52 letters and the separator character).

8. Define the function to train on a batch of data:

```
def train_batch(data, model, optimizer, criterion):
    model.train()
    imgs, targets, label_lens, input_lens, labels = data
    optimizer.zero_grad()
    preds = model(imgs)
    loss = criterion(preds, targets, input_lens, label_lens)
    loss.backward()
    optimizer.step()
    results = trn_ds.evaluate(model, imgs.to(device),labels)
    return loss, results
```

9. Define the function to validate on a batch of data:

```
@torch.no_grad()
def validate_batch(data, model):
    model.eval()
    imgs, targets, label_lens, input_lens, labels = data
    preds = model(imgs)
    loss = criterion(preds, targets, input_lens, label_lens)
    return loss, val_ds.evaluate(model, imgs.to(device), \
                                 labels)
```

10. Define the model object, optimizer, loss function, and the number of epochs:

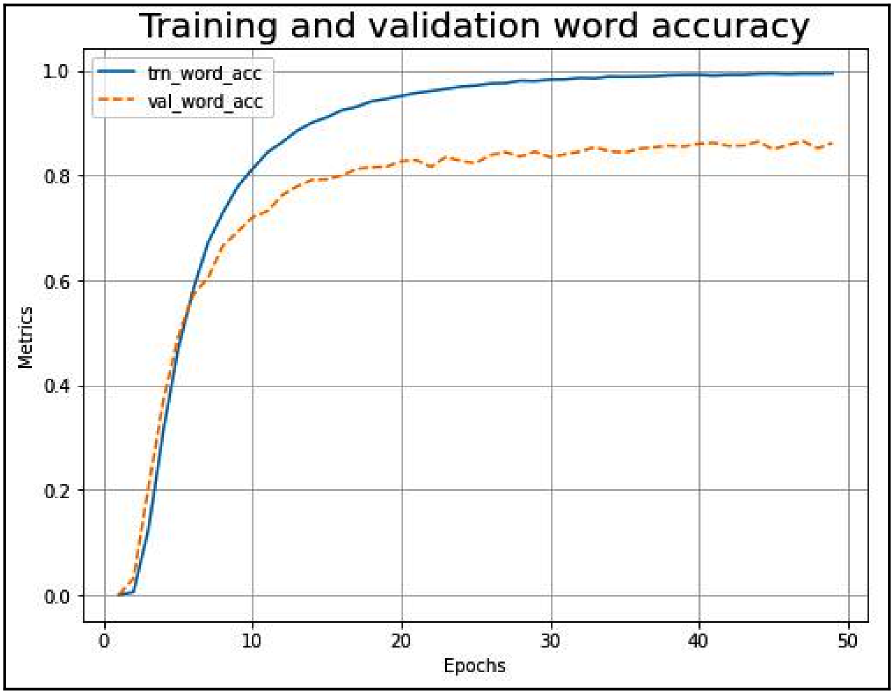
```
model = Ocr(len(vocab)).to(device)
criterion = ctc

optimizer = optim.AdamW(model.parameters(), lr=3e-3)

n_epochs = 50
log = Report(n_epochs)
```

11. Run the model over increasing epochs:

```
for ep in range( n_epochs):
    N = len(trn_dl)
    for ix, data in enumerate(trn_dl):
        pos = ep + (ix+1)/N
        loss, results = train_batch(data, model, optimizer, \
                                    criterion)
        ca, wa = results['char-accuracy'], \
                 results['word-accuracy']
        log.record(pos=pos, trn_loss=loss, trn_char_acc=ca, \
                   trn_word_acc=wa, end='\r')
    val_results = []
```

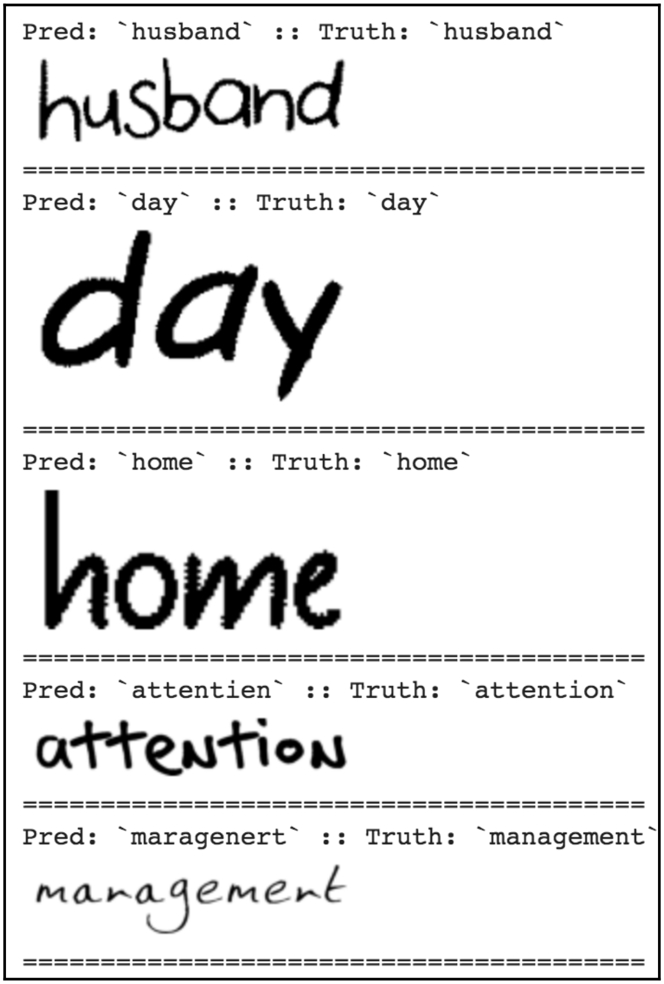
```
N = len(val_dl)
for ix, data in enumerate(val_dl):
    pos = ep + (ix+1)/N
    loss, results = validate_batch(data, model)
    ca, wa = results['char-accuracy'], \
             results['word-accuracy']
    log.record(pos=pos, val_loss=loss, val_char_acc=ca, \
               val_word_acc=wa, end='\r')

log.report_avgs(ep+1)
print()
for jx in range(5):
    img, label = val_ds.sample()
    _img=torch.Tensor(val_ds.preprocess(img)[None,None])\
                                    .to(device)
    pred = model(_img)[:,0,:]
    pred = trn_ds.decoder_chars(pred)
    print(f'Pred: `{pred}` :: Truth: `{label}`')
print()
```

The preceding code results in the following output:



From the graph, we can see that the model has a word accuracy of around 80% on the validation dataset.

Furthermore, the predictions at the end of training are as follows:



So far, we have learned about using a combination of CNNs and RNNs. In the next section, we will learn about leveraging transformer architecture to perform object detection of the trucks versus bus dataset we worked on in the previous chapters.

Object detection using DETR

In previous chapters on object detection, we learned about leveraging anchor boxes/region proposals to perform object classification and detection. However, it involved a pipeline of steps to come up with object detection. DETR is a technique that leverages transformers to come up with an end-to-end pipeline that simplifies the object detection network architecture considerably. Transformers are one of the more popular and more recent techniques to perform various tasks in NLP. In this section, we will learn about the working details of transformers, DETR, and code it up to perform our task of detecting trucks versus buses.

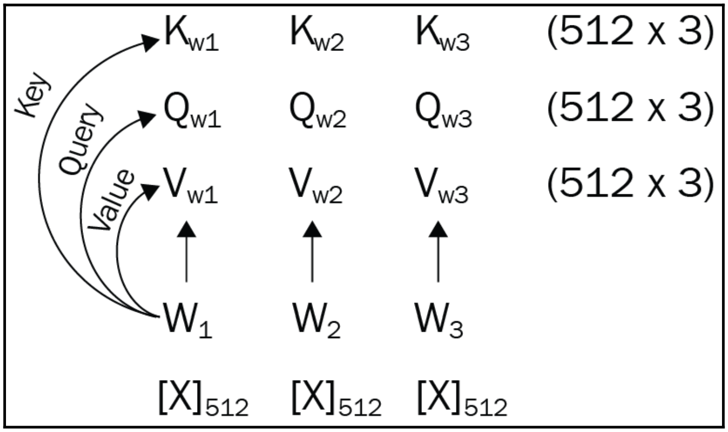
The working details of transformers

Transformers have proven to be a remarkable architecture for sequence-to-sequence problems. Almost all NLP tasks, as of the time of writing this book, have state-of-the-art implementations that come from transformers. This class of networks uses only linear layers and softmax to create **self-attention** (which will be explained in detail in the next sub-section). Self-attention helps in identifying the interdependency among words in the input text. The input sequence typically does not exceed 2,048 items as this is large enough for text applications. However, if images are to be used with transformers, they have to be flattened, which creates a sequence in the order of thousands/millions of pixels (as a 300 x 300 x 3 image would contain 270K pixels), which is not feasible. Facebook Research came up with a novel way to bypass this restriction by giving the feature map (which has a smaller size than the input image) as input to the transformer. Let's understand the basics of transformers in this section and understand the relevant code blocks later.

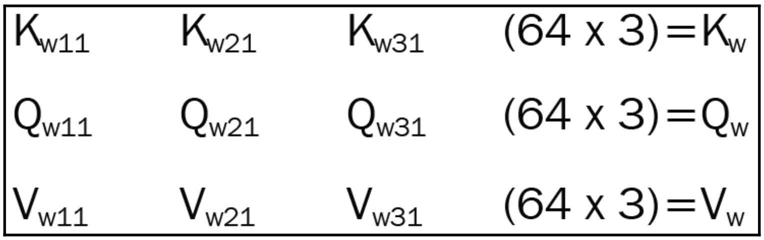
Basics of transformers

At the heart of a transformer is the **self-attention** module. It takes three two-dimensional matrices (called **query** (**Q**), **key** (**K**), and **value** (**V**) matrices) as input. The matrices can have very large embedding sizes (as they would contain text size x embedding size number of values), so they are split up into smaller components first (step 1 in the following diagram), before running through the scaled-dot-product-attention (step 2 in the following diagram).

Let's understand how self-attention works. In a hypothetical scenario where the sequence length is 3, we have three word embeddings (W_1, W_2, and W_3) as input. Say each embedding is of size 512. Each of these embeddings is individually converted into three additional vectors, which are the query, key, and value vectors corresponding to each input:



Since each vector is 512 in size, it is computationally expensive to do a matrix multiplication between them. So, we split each of these vectors into eight parts, having eight sets of (64 x 3) vectors for each of key, query, and value tensor, where 64 is obtained from 512 (embedding size) / 8 (multi-heads) and 3 is the sequence length:



Note that there will be eight sets of tensors of K_{w11}, K_{w12}, and so on because there are eight multi-heads.

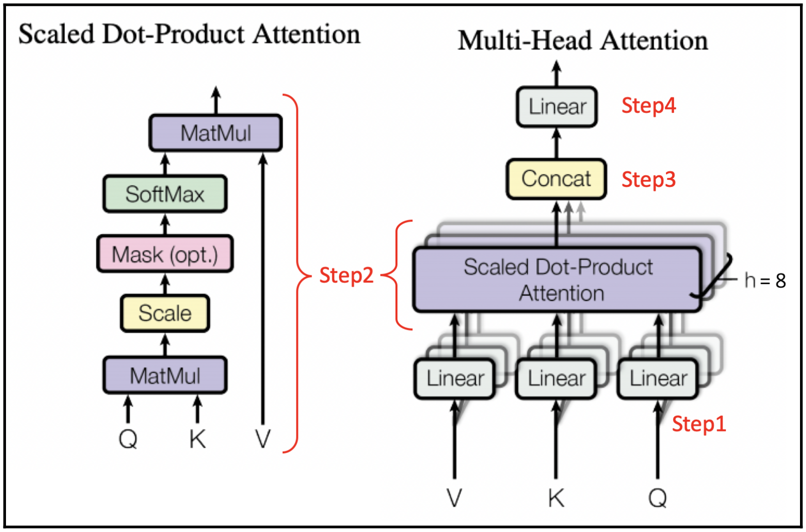
In each part, we first perform matrix multiplication between the key and query matrices. This way, we end up with a 3 x 3 matrix. Pass it through softmax activation. Now, we have a matrix showing how important each word is, in relation to every other word:

$$\begin{pmatrix} K^{T}_{W}\;Q_{W} \\ (3x64)\;(64x3) \\ 3x3 \end{pmatrix} \longrightarrow \begin{matrix} V^{*}_{W} \\ 3x3 \end{matrix} \longrightarrow \begin{matrix} \text{softmax}(V^{*}_{W}) \\ 3x3 \end{matrix}$$

Finally, we perform matrix multiplication of the preceding tensor output with the value tensor to get the output of our self-attention operation:

$$\begin{matrix} V_{W} & V^{*}_{W} & \longrightarrow & Z \\ (64 \text{ x } 3) & (3 \text{ x } 3) & & (64 \text{ x } 3) \end{matrix}$$

We then combine the eight outputs of this step, go back using concat layer (**step3** in the following diagram), and end up with a single tensor of size 512 x 3. Because of the splitting of the Q, K, and V matrices, the layer is also called **multi-head self-attention** (Source: `https://arxiv.org/pdf/1706.03762.pdf`):

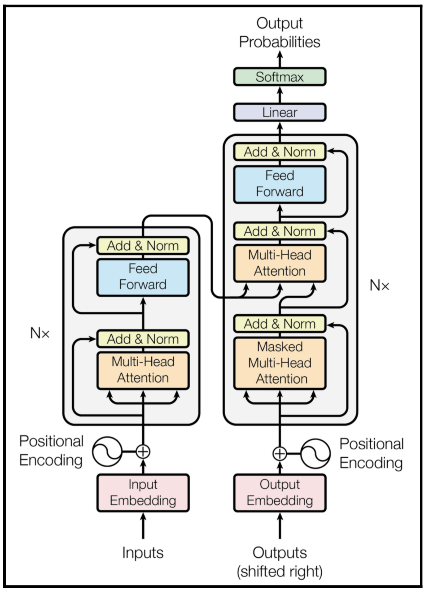


The idea behind such a complex-looking network is as follows:

- **Values** (**Vs**) are the processed embeddings that need to be learned for a given input, in its context of key and query matrices.
- **Queries** (**Qs**) and **Keys** (**Ks**) act in such a way that their combination will create the right mask so that only the important parts of the value matrix are fed to the next layer.

For our example in computer vision, when searching for an object such as a horse, the query should contain information to search for an object that is large in dimension and is brown, black, or white in general. The softmax output of scaled dot-product attention will reflect those parts of the key matrix that contain this color (brown, black, white, and so on) in the image. Thus, the values output from the self-attention layer will have those parts of the image that are roughly of the desired color and are present in the values matrix.

We use the self-attention block several times in the network, as illustrated in the following diagram. The transformer network contains an encoding network (the left part of the diagram) whose input is the source sequence. The output of the encoding half is used as the key and query inputs for the decoding half, while the value input is going to be learned by the neural network independently to the encoding half (Source: `https://arxiv.org/pdf/1706.03762.pdf`):



Finally, even though this is a sequence of inputs, there's no sign of which token (word) is first and which is next (since a linear layer has no positional indication). Positional encodings are learnable embeddings (and sometimes hardcoded vectors) that we add to each input as a function of its position in the sequence. This is done so that the network understands which word embedding is first in the sequence, which is second, and so on.

The way to create a transformer network in PyTorch is very simple. There is a built-in transformer block that you can create, like so:

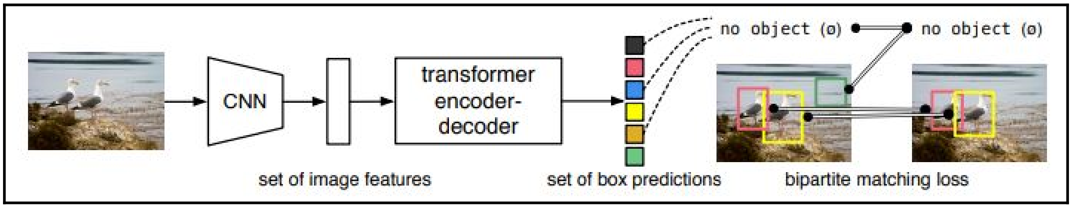
```
from torch import nn
transformer = nn.Transformer(hidden_dim, nheads, \
                        num_encoder_layers, num_decoder_layers)
```

Here, `hidden_dim` is the size of the embeddings, `nheads` is the number of heads in the multi-head self attention, and `num_encoder_layers` and `num_decoder_layers` are the number of encoding and decoding blocks in the network, respectively.

The working details of DETR

There are few key differences between a normal transformer network and DETR. Primarily, our input is an image, not a sequence. So, DETR passes the image through a ResNet backbone to get a vector of size 256 that can be then treated as a sequence. In our case, the inputs to the decoder are object-query embeddings, which are automatically learned during training. These act as the query matrices for all the decoder layers. Similarly, for every layer, the key and query matrices are going to be the final output matrix from the encoder block, replicated twice. The final output of the transformer is going to be a `Batch_Size` x 100 x `Embedding_Size` tensor, where the model has been trained with `100` as the sequence length; that is, it learned 100 object-query embeddings and returns 100 vectors per image, indicating whether there is an object or not. These 100 x `Embedding_Size` matrices are individually fed to an object classification module and object regression module, which independently predict whether there's an object (and what it is) and what the bounding box coordinates are, respectively. Both of these modules are simple `nn.Linear` layers.

At a high level, the architecture of DETR is as follows (Source: `https://arxiv.org/pdf/2005.12872.pdf`):



The definition of one of the smaller variants of DETR is as follows:

- Create the DETR model class:

```
from collections import OrderedDict
class DETR(nn.Module):
    def __init__(self,num_classes,hidden_dim=256,nheads=8, \
                 num_encoder_layers=6, num_decoder_layers=6):
        super().__init__()
        self.backbone = resnet50()
```

- We are going to take only a few layers from ResNet and discard the rest. These few layers contain the names given in the following list:

```
        layers = OrderedDict()
        for name,module in self.backbone.named_modules():
            if name in ['conv1','bn1','relu','maxpool', \
                    'layer1','layer2','layer3','layer4']:
                layers[name] = module
        self.backbone = nn.Sequential(layers)
        self.conv = nn.Conv2d(2048, hidden_dim, 1)
        self.transformer = nn.Transformer(\
                            hidden_dim, nheads, \
                            num_encoder_layers, \
                            num_decoder_layers)
        self.linear_class = nn.Linear(hidden_dim, \
                                      num_classes + 1)
        self.linear_bbox = nn.Linear(hidden_dim, 4)
```

In the preceding code, we are specifying the following:

- The layers of interest in sequential order (`self.backbone`)
- The convolution operation (`self.conv`)
- The transformer block (`self.transformer`)

    - The final connection to obtain the number of classes (`self.linear_class`)
    - The bounding box (`self.linear_box`)

- Define the positional embeddings for the encoder and decoder layers:

```
self.query_pos = nn.Parameter(torch.rand(100, \
                                        hidden_dim))
self.row_embed = nn.Parameter(torch.rand(50, \
                                        hidden_dim // 2))
self.col_embed = nn.Parameter(torch.rand(50, \
                                        hidden_dim // 2))
```

`self.query_pos` is the positional embedding input for the decoder layer, whereas `self.row_embed` and `self.col_embed` form the two-dimensional positional embeddings for the encoder layer.

- Define the `forward` method:

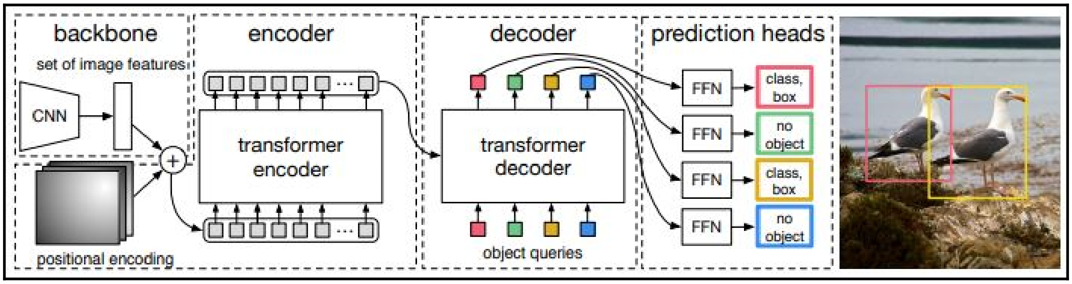
```
def forward(self, inputs):
    x = self.backbone(inputs)
    h = self.conv(x)
    H, W = h.shape[-2:]
    '''Below operation is rearranging the positional
    embedding vectors for encoding layer'''
    pos = torch.cat([\
        self.col_embed[:W].unsqueeze(0).repeat(H, 1, 1),\
        self.row_embed[:H].unsqueeze(1).repeat(1, W, 1),\
        ], dim=-1).flatten(0, 1).unsqueeze(1)
    '''Finally, predict on the feature map obtained
    from resnet using the transformer network'''
    h = self.transformer(pos+0.1*h.flatten(2)\
                         .permute(2, 0, 1), \
                self.query_pos.unsqueeze(1))\
                         .transpose(0, 1)
    '''post process the output `h` to obtain class
       probability and bounding boxes'''
    return {'pred_logits': self.linear_class(h), \
            'pred_boxes': self.linear_bbox(h).sigmoid()}
```

You can load a pre-trained model, trained on the COCO dataset, and use it for predicting generic classes. The prediction logic is explained in the next section and you can use the same function on this model as well (of course, with COCO classes):

```
detr = DETR(num_classes=91)
state_dict = torch.hub.load_state_dict_from_url(url=\
'https://dl.fbaipublicfiles.com/detr/detr_demo-da2a99e9.pth'\
,map_location='cpu', check_hash=True)
detr.load_state_dict(state_dict)
detr.eval();
```

Note that DETR can fetch predictions in a single shot when compared to the other object detection techniques that we learned in `Chapter 7`, *Basics of Object Detection*, and `Chapter 8`, *Advanced Object Detection*.

A more detailed version of the DETR architecture is as follows (Source: `https://arxiv.org/pdf/2005.12872.pdf`):



In the **backbone** segment, we are fetching the image features, which are then passed through an encoder, which concatenates the image features with positional embeddings.

Essentially, positional embeddings, present as `self.row_embed, self.col_embed`, in the `__init__` method help with encoding information about the position of various objects in the image. The encoder takes the concatenation of the positional embeddings and image features to obtain a hidden state vector, `h` (in the forward method), which is passed as input to the decoder. This transformer output is further fed to two linear networks, one for object identification and one for bounding box regression. All the transformer's complexity is hidden in the `self.transformer` module of the network.

The training uses a novel Hungarian loss, which is responsible for the identification of objects as a set and penalizes redundant predictions. This eliminates the need for a non-max suppression altogether. The details of Hungarian loss are out of scope for this book and we encourage you to go through the working details in the original paper.

The decoder takes a combination of the encoder hidden state vector and the object queries. An object query works in a similar fashion as that of positional embeddings/anchor boxes to come up with five predictions – one for the class of the object and the other four for the bounding box corresponding to the object.

With an intuition and high-level understanding of the working details of DETR, let's code it up in the following section.

Detection with transformers in code

In the following code, we will code up DETR to predict the objects of our interest – buses versus trucks:

> The following code is available as `Object_detection_with_DETR.ipynb` in the `Chapter15` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from and is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components from text.

1. Import the dataset and create a folder named `detr`:

```
import os
if not os.path.exists('open-images-bus-trucks'):
    !pip install -q torch_snippets torchsummary
    !wget --quiet
https://www.dropbox.com/s/agmzwk95v96ihic/open-images-bus-truc
ks.tar.xz
    !tar -xf open-images-bus-trucks.tar.xz
    !rm open-images-bus-trucks.tar.xz
    !git clone https://github.com/sizhky/detr/
%cd detr
```

- Move the annotation images to the `detr` folder:

```
%cd ../open-images-bus-trucks/annotations
!cp mini_open_images_train_coco_format.json\
 instances_train2017.json
!cp mini_open_images_val_coco_format.json\
 instances_val2017.json
%cd ..
!ln -s images/ train2017
!ln -s images/ val2017
%cd ../detr
```

- Define the classes of interest:

```
CLASSES = ['', 'BUS','TRUCK']
```

2. Import the pre-trained DETR model:

```
from torch_snippets import *
if not os.path.exists('detr-r50-e632da11.pth'):
    !wget
https://dl.fbaipublicfiles.com/detr/detr-r50-e632da11.pth
    checkpoint = torch.load("detr-r50-e632da11.pth", \
                            map_location='cpu')
    del checkpoint["model"]["class_embed.weight"]
    del checkpoint["model"]["class_embed.bias"]
    torch.save(checkpoint,"detr-r50_no-class-head.pth")
```

3. Train the model with the images and annotations present in the `open-images-bus-trucks` folder:

```
!python main.py --coco_path ../open-images-bus-trucks/\
  --epochs 10 --lr=1e-4 --batch_size=2 --num_workers=4\
  --output_dir="outputs" --resume="detr-r50_no-class-head.pth"
```

4. Once we train the model, load it from the folder:

```
from main import get_args_parser, argparse, build_model
parser=argparse.ArgumentParser('DETR training and \
            evaluation script', parents=[get_args_parser()])
args, _ = parser.parse_known_args()

model, _, _ = build_model(args)
model.load_state_dict(torch.load("outputs/checkpoint.pth")\
                      ['model']);
```

5. Postprocess the predictions to fetch the image and the bounding box around objects:

```
from PIL import Image, ImageDraw, ImageFont

# standard PyTorch mean-std input image normalization
# colors for visualization
COLORS = [[0.000, 0.447, 0.741], [0.850, 0.325, 0.098],
          [0.929, 0.694, 0.125], [0.494, 0.184, 0.556],
          [0.466, 0.674, 0.188], [0.301, 0.745, 0.933]]
transform = T.Compose([
    T.Resize(800),
    T.ToTensor(),
    T.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

# for output bounding box post-processing
def box_cxcywh_to_xyxy(x):
    x_c, y_c, w, h = x.unbind(1)
    b = [(x_c - 0.5 * w), (y_c - 0.5 * h), \
         (x_c + 0.5 * w), (y_c + 0.5 * h)]
    return torch.stack(b, dim=1)

def rescale_bboxes(out_bbox, size):
    img_w, img_h = size
    b = box_cxcywh_to_xyxy(out_bbox)
    b = b * torch.tensor([img_w, img_h, img_w, img_h], \
                         dtype=torch.float32)
    return b
def detect(im, model, transform):
    img = transform(im).unsqueeze(0)
    '''demo model only supports images up to 1600 pixels
     on each side'''
    assert img.shape[-2] <= 1600 and \
    img.shape[-1] <= 1600
    outputs = model(img)
    # keep only predictions with 0.7+ confidence
    probas=outputs['pred_logits'].softmax(-1)[0,:,:-1]
    keep = probas.max(-1).values > 0.7
    # convert boxes from [0; 1] to image scales
    bboxes_scaled = rescale_bboxes(outputs['pred_boxes']\
                               [0, keep], im.size)
    return probas[keep], bboxes_scaled

def plot_results(pil_img, prob, boxes):
    plt.figure(figsize=(16,10))
    plt.imshow(pil_img)
    ax = plt.gca()
```

```
        for p, (xmin, ymin, xmax, ymax), c in zip(prob, \
                                    boxes.tolist(), COLORS * 100):
            ax.add_patch(plt.Rectangle((xmin, ymin), \
                                xmax - xmin, ymax - ymin,\
                                fill=False, color=c, linewidth=3))
            cl = p.argmax()
            text = f'{CLASSES[cl]}: {p[cl]:0.2f}'
            ax.text(xmin, ymin, text, fontsize=15,\
                    bbox=dict(facecolor='yellow', alpha=0.5))
        plt.axis('off')
        plt.show()
```

6. Predict on new images:

```
for _ in range(2):
    image = Image.open(choose(Glob(\
                '../open-images-bus-trucks/images/*')))\
                .resize((800,800)).convert('RGB')
    scores, boxes = detect(image, model, transform)
    plot_results(image, scores, boxes)
```

The preceding code generates the following output:

From the preceding, we can see that we can now train the model that is able to predict objects within an image.

Note that we have trained the model on a small dataset and hence the accuracy of detection might not be very high in this particular case. However, the same methodology can be extended to large datasets. As an exercise, we suggest you apply the same technique to detect multiple objects as we did in `Chapter 10`, *Applications of Object Detection, and Segmentation*.

Summary

In this chapter, we learned about how RNNs work and specifically the variant of LSTM in detail. Furthermore, we learned about leveraging CNNs and RNNs together as we passed an image through a pre-trained model to extract features and passed the features as time steps to the RNN to extract the words one at a time, in our image captioning use case. We then took the combination of CNNs and RNNs a step further, where we leveraged the CTC loss function to transcribe handwritten images. The CTC loss function helped in ensuring that we squash the same character coming from subsequent time steps into a single character and also in ensuring that all possible combinations of output are considered, and then we evaluated the loss based on the combination resulting in the ground truth. Finally, we learned about leveraging transformers to perform object detection using DETR, during which we also understood how transformers work and how they can be leveraged in the context of object detection.

In the next chapter, we will learn about how to combine a CNN and reinforcement learning techniques to come up with a self-driving car prototype, an agent that is able to play the Atari Space Invaders game with no supervision after learning the Bellman equation, which enables assigning value to a given state.

Questions

1. Why are CNNs and RNNs combined in image captioning?
2. Why are start and end tokens provided in image captioning but not in handwritten transcription?
3. Why is the CTC loss function leveraged in handwriting transcription?
4. How do transformers help in object detection?

16 Combining Computer Vision and Reinforcement Learning

In the previous chapter, we learned about how to combine NLP techniques (LSTM and transformer) with computer vision-based techniques. In this chapter, we will learn how to combine reinforcement learning-based techniques (primarily deep Q-learning) with computer vision-based techniques.

We will start by learning about the basics of reinforcement learning and then about the terminology associated with identifying how to calculate the value (Q-value) associated with taking an action in a given state. Next, we will learn about filling a Q-table, which helps in identifying the value associated with various actions in a given state. Furthermore, we will learn about identifying the Q-values of various actions in scenarios where coming up with a Q-table is infeasible due to a high number of possible states; we'll do this using Deep Q-Network. This is where we will understand how to leverage neural networks in combination with reinforcement learning. Next, we will learn about scenarios where the Deep Q-Network model does not work and address this by using the Deep Q-Network alongside the fixed targets model. Here, we will play a video game known as Pong by leveraging CNN in conjunction with reinforcement learning. Finally, we will leverage what we've learned to build an agent that can drive a car autonomously in a simulated environment – CARLA.

In summary, in this chapter, we will cover the following topics:

- Learning the basics of reinforcement learning
- Implementing Q-learning
- Implementing deep Q-learning
- Implementing deep Q-learning with fixed targets
- Implementing an agent to perform autonomous driving

Learning the basics of reinforcement learning

Reinforcement learning (**RL**) is an area of machine learning concerned with how software **agents** ought to take **actions** in a given **state** of an **environment** to maximize the notion of cumulative **reward**.

To understand how RL helps, let's consider a simple scenario. Imagine that you are playing chess against a computer (in our case, the computer is an **agent** that has learned/is learning how to play chess). The setup (rules) of the game constitutes the **environment**. Furthermore, as we make a move (take an **action**), the **state** of the board (the location of various pieces on the chessboard) changes. At the end of the game, depending on the result, the agent gets a **reward**. The objective of the agent is to maximize the reward.

If the machine (agent1) is playing against a human, the number of games that it can play is finite (depending on the number of games the human can play). This might create a bottleneck for the agent to learn well. However, what if agent1 (the agent that is learning the game) can play against agent2 (agent2 could be another agent that is learning chess or it could be a piece of chess software that has been pre-programmed to play the game well)? Theoretically, the agents can play infinite games with each other, which results in maximizing the opportunity to learn to play the game well. This way, by playing multiple games with each other, the learning agent is likely to learn how to address the different scenarios/states of the game well.

Let's understand the process that the learning agent will follow to learn well:

1. Initially, the agent takes a random action in a given state.
2. The agent stores the action it has taken in various states within a game in **memory**.
3. Then, the agent associates the result of the action in various states with a reward.
4. After playing multiple games, the agent can correlate the action in a state to a potential reward by replaying its **experiences**.

Next comes the question of quantifying the **value** that corresponds to taking an action in a given state. We'll learn how to calculate this in the next section.

Calculating the state value

To understand how to quantify the value of a state, let's use a simple scenario where we will define the environment and objective as follows:

Start		
		+1

The environment is a grid with two rows and three columns. The agent starts at the Start cell and it achieves its objective (rewarded with a score of +1) if the agent reaches the bottom-right grid cell. The agent does not get a reward if it goes to any other cell. The agent can take an action by going to the right, left, bottom, or up, depending on the feasibility of the action (the agent can go to the right or to the bottom in the start grid cell, for example). The reward of reaching any of the remaining cells other than the bottom-right cell is 0.

By using this information, let's calculate the **value** of a cell (the state that the agent is in, in a given snapshot). Given that some energy is spent moving from one cell to another, we discount the value of reaching a cell by a factor of γ, where γ takes care of the energy that's spent in moving from one cell to another. Furthermore, the introduction of γ results in the agent learning to play well sooner. With this, let's formalize the Bellman equation, which helps in calculating the value of a cell:

$$\begin{aligned} value\ of\ action\ taken\ in\ a\ state\ =\ & reward\ of\ moving\ to\ the\ next\ cell \\ & +\ \gamma \times value\ of\ the\ best\ possible\ action\ in\ next\ state \end{aligned}$$

With the preceding equation in place, let's calculate the values of all cells (**once the optimal actions in a state have been identified**) with the value of γ being 0.9 (the typical value of γ is between 0.9 and 0.99):

$$\begin{aligned} V_{22} &= R_{23} + \gamma \times V_{23} &&= 1 + \gamma \times 0 &&= 1 \\ V_{13} &= R_{23} + \gamma \times V_{23} &&= 1 + \gamma \times 0 &&= 1 \\ V_{21} &= R_{22} + \gamma \times V_{22} &&= 0 + \gamma \times 1 &&= 0.9 \\ V_{12} &= R_{13} + \gamma \times V_{13} &&= 0 + \gamma \times 1 &&= 0.9 \\ V_{11} &= R_{12} + \gamma \times V_{12} &&= 0 + \gamma \times 0.9 &&= 0.81 \end{aligned}$$

From the preceding calculations, we can understand how to calculate the values in a given state (cell), when given the optimal actions in that state. These are as follows for our simplistic scenario of reaching the terminal state:

0.81	0.9	1
0.9	1	

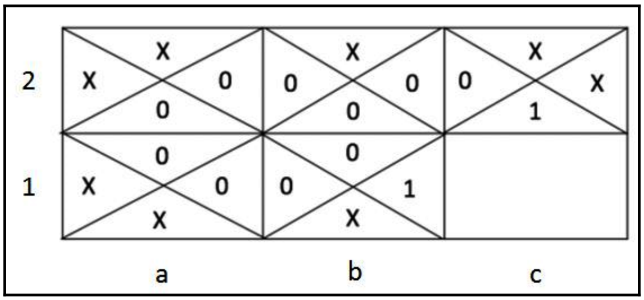
With the values in place, we expect the agent to follow a path of increasing value.

Now that we understand how to calculate the state value, in the next section, we will understand how to calculate the value associated with a state-action combination.

Calculating the state-action value

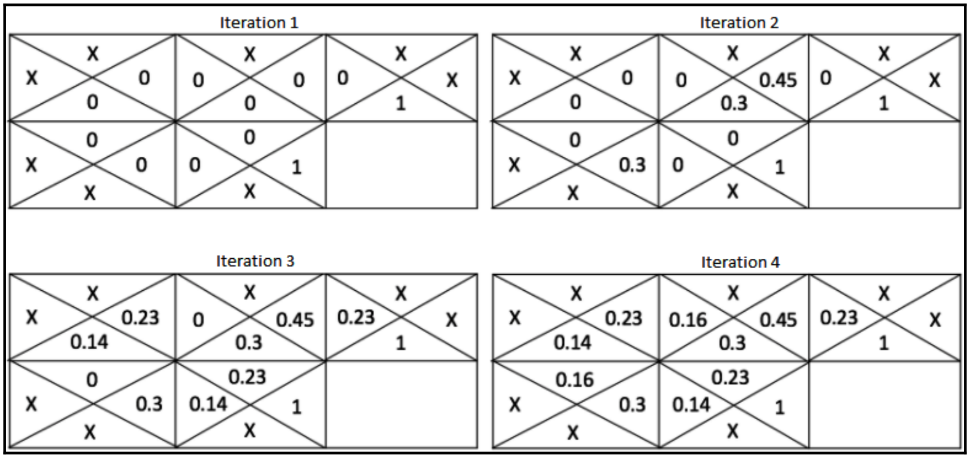
In the previous section, we provided a scenario where we already know that the agent is taking optimal actions (which is not realistic). In this section, we will look at a scenario where we can identify the value that corresponds to a state-action combination.

In the following image, each sub-cell within a cell represents the value of taking an action in the cell. Initially, the cell values for various actions are as follows:



Note that, in the preceding image, cell *b1* (2nd row and 2nd column) will have a value of 1 if the agent moves right from the cell (as it corresponds to the terminal cell); the other actions result in a value of 0. X represents that the action is not possible and hence no value is associated with it.

Over four iterations (steps), the updated cell values for the actions in the given state are as follows:



This would then go through multiple iterations to provide the optimal action that maximizes value at each cell.

Let's understand how to obtain the cell values in the second table (*Iteration 2* in the preceding image). Let's narrow this down to 0.3, which was obtained by taking the downward action when present in the 1st row and 2nd column of the second table. When the agent takes the downward action, there is a 1/3 chance of it taking the optimal action in the next state. Hence, the value of taking a downward action is as follows:

$$\begin{aligned} value\ of\ taking\ downward\ action\ &= immediate\ reward\ + \gamma * (1/3 * 0\ + 1/3 * 0\ +\ 1/3 * 1) \\ &= 0\ + 0.9 * 1/3 \\ &=\ 0.3 \end{aligned}$$

In a similar manner, we can obtain the values of taking different possible actions in different cells.

Now that we know how the values of various actions in a given state are calculated, in the next section, we will learn about Q-learning and how we can leverage it, along with the Gym environment, so that it can play various games.

Implementing Q-learning

In the previous section, we manually calculated the state-action values for all combinations. Technically, now that we have calculated the various state-action values we need, we can now identify the action that will be taken in every state. However, in the case of a more complex scenario – for example, when playing video games – it gets tricky to fetch state information. OpenAI's Gym environment comes in handy in this scenario. It contains a pre-defined environment for the game we're playing. Here, it fetches the next state information, given an action that's been taken in the current state. So far, we have considered the scenario of choosing the most optimal path. However, there can be scenarios where we are stuck at the local minima.

In this section, we will learn about Q-learning, which helps with calculating the value associated with the action in a state, as well as about leveraging the Gym environment so that we can play various games. For now, we'll take a look at a simple game called Frozen Lake. We'll also take a look at exploration-exploitation, which helps us avoid getting stuck at the local minima. However, before we do that, we will learn about the Q-value.

Q-value

The Q in Q-learning or Q-value represents the quality of an action. Let's learn how to calculate it:

$$\begin{aligned} value\ of\ action\ taken\ in\ a\ state \ =&\ reward\ of\ moving\ to\ the\ next\ cell \\ &+\ \gamma \times value\ of\ the\ best\ possible\ action\ in\ next\ state \end{aligned}$$

We already know that we must keep **updating** the state-action value of a given state until it is saturated. Hence, we'll modify the preceding formula like so:

$$\begin{aligned} value\ of\ action\ taken\ in\ a\ state \ =&\ Value\ of\ action\ taken\ in\ a\ state \\ &+\ 1 \times (reward\ of\ moving\ to\ the\ next\ cell \\ &+ \gamma \times Value\ of\ the\ best\ possible\ action\ in\ next\ state \\ &-\ Value\ of\ action\ taken\ in\ a\ state) \end{aligned}$$

In the preceding equation, we replace 1 with the learning rate so that we can update the value of the action that's taken in a state more gradually:

$$\begin{aligned} Value\ of\ action\ taken\ in\ a\ state\ (Q - value) = \ & Value\ of\ action\ taken\ in\ a\ state \\ & + \ learning\ rate * (reward\ of\ moving\ to\ the\ next\ cell \\ & + \gamma\ \times\ Value\ of\ the\ best\ possible\ action\ in\ next\ state \\ & -\ Value\ of\ action\ taken\ in\ a\ state) \end{aligned}$$

With this formal definition of Q-value in place, in the next section, we'll learn about the Gym environment and how it helps us fetch the Q-table (which stores information about the values of various actions that have been taken at various states) and thus come up with the optimal actions in a state.

Understanding the Gym environment

In this section, we will explore the Gym environment and the various functionalities present in it while playing the Frozen Lake game present in the Gym environment:

The following code is available as `Understanding_the_Gym_environment.ipynb` in the `Chapter16` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Import the relevant packages:

```
import numpy as np
import gym
import random
```

2. Print the various environments present in the Gym environment:

```
from gym import envs
print(envs.registry.all())
```

The preceding code prints a dictionary containing all the games available within Gym.

3. Create an environment for the chosen game:

```
env = gym.make('FrozenLake-v0', is_slippery=False)
```

4. Inspect the created environment:

```
env.render()
```

The preceding code results in the following output:

In the preceding image, the agent starts at **S**. Here, **F** specifies that the cell is frozen, while **H** specifies that the cell has a hole in it. The agent gets a reward of 0 if it goes to cell **H** and the game is terminated. The objective of the game is for the agent to reach **G**.

5. Print the size of the observation space (number of states) in the game:

```
env.observation_space.n
```

The preceding code gives us an output of 16. This represents the 16 cells that the game has.

6. Print the number of possible actions:

```
env.action_space.n
```

The preceding code results in a value of 4, which represents the four possible actions that can be taken.

7. Sample a random action at a given state:

```
env.action_space.sample()
```

`.sample()` specifies that we fetch one of the possible four actions in a given state. The scalar corresponding to each action can be associated with the name of the action. We can do this by inspecting the code in GitHub: `https://github.com/openai/gym/blob/master/gym/envs/toy_text/frozen_lake.py`.

8. Reset the environment to its original state:

```
env.reset()
```

9. Take (step) an action:

```
env.step(env.action_space.sample())
```

The preceding code fetches the next state, the reward, the flag that states whether the game was completed, and additional information. We can execute the game with `.step` since the environment readily provides the next state when it's given a step with an action.

These steps form the basis for us to build a Q-table that dictates the optimal action to be taken in each state. We'll do this in the next section.

Building a Q-table

In the previous section, we learned how to calculate Q-values for various state-action pairs manually. In this section, we will leverage the Gym environment and the various modules associated with it to populate the Q-table – where rows represent the states that an agent can be in and columns represent the actions the agent can take. The values of the Q-table represent the Q-values of taking an action in a given state.

We can populate the values of the Q-table using the following strategy:

1. Initialize the game environment and the Q-table with zeros.
2. Take a random action and fetch the next state, reward, the flag stating whether the game was completed, and additional information.
3. Update the Q-value using the Bellman equation we defined earlier.
4. Repeat *steps 2* and *3* so that there's a maximum of 50 steps in an episode.
5. Repeat *steps 2*, *3*, and *4* over multiple episodes.

Let's code up the preceding strategy:

> The following code is available as `Building_Q_table.ipynb` in the `Chapter16` folder in this book's GitHub repository - `https://tinyurl.com/mcvp-packt`

1. Initialize the game environment:

```
import numpy as np
import gym
import random
env = gym.make('FrozenLake-v0', is_slippery=False)
```

- Initialize the Q-table with zeros:

```
action_size=env.action_space.n
state_size=env.observation_space.n
qtable=np.zeros((state_size,action_size))
```

  The preceding code checks the possible actions and states that can be used to build a Q-table. The Q-table's dimension should be the number of states multiplied by the number of actions.

2. Play multiple episodes while taking a random action. Here, we reset the environment at the end of every episode:

```
episode_rewards = []
for i in range(10000):
    state=env.reset()
```

- Take a maximum of 50 steps per episode:

```
    total_rewards = 0
    for step in range(50):
```

  We are considering a maximum of 50 steps per episode as it's possible for the agent to keep oscillating between two states forever (think of left and right actions being performed consecutively forever). Thus, we need to specify the maximum number of steps an agent can take.

- Sample a random action and take (step) it:

```
        action=env.action_space.sample()
        new_state,reward,done,info=env.step(action)
```

- Update the Q-value that corresponds to the state and the action:

```
        qtable[state,action]+=0.1*(reward+0.9*np.max(\
                                    qtable[new_state,:]) \
                                   -qtable[state,action])
```

  In the preceding code, we specified that the learning rate is 0.1 and that were are updating the Q-value of a state-action combination by taking the maximum Q-value of the next state (`np.max(qtable[new_state,:])`) into consideration.

- Update the `state` value to `new_state`, which we obtained previously, and accumulate `reward` into `total_rewards`:

```
state=new_state
total_rewards+=reward
```

- Place the rewards in a list (`episode_rewards`) and print the Q-table (`qtable`):

```
    episode_rewards.append(total_rewards)
print(qtable)
```

The preceding code fetches the Q-values of various actions in a state:

```
[[0.531441   0.59049    0.59049    0.531441  ]
 [0.531441   0.         0.6561     0.59049   ]
 [0.59049    0.729      0.59049    0.6561    ]
 [0.6561     0.         0.59049    0.59049   ]
 [0.59049    0.6561     0.         0.531441  ]
 [0.         0.         0.         0.        ]
 [0.         0.81       0.         0.6561    ]
 [0.         0.         0.         0.        ]
 [0.6561     0.         0.729      0.59049   ]
 [0.6561     0.81       0.81       0.        ]
 [0.729      0.9        0.         0.729     ]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.         0.81       0.9        0.72899998]
 [0.80999997 0.9        1.         0.81      ]
 [0.         0.         0.         0.        ]]
```

We will learn about how the obtained Q-table is leveraged in the next section.

So far, we have kept taking a random action every time. However, in a realistic scenario, once we have learned that certain actions can't be taken in certain states and vice versa, we don't need to take a random action anymore. The concept of exploration-exploitation comes in handy in such a scenario.

Leveraging exploration-exploitation

In the previous section, we explored the possible actions we can take in a given space. In this section, we will learn about the concept of exploration-exploitation, which can be described as follows:

- **Exploration** is a strategy where we learn what needs to be done (what action to take) in a given state.
- **Exploitation** is a strategy where we leverage what has already been learned; that is, which action to take in a given state.

During the initial stages, it is ideal to have a high amount of exploration as the agent won't know what optimal actions to take initially. Through the episodes, as the agent learns the Q-values of various state-action combinations over time, we must leverage exploitation to perform the action that leads to a high reward.

With this intuition in place, let's modify the Q-value calculation that we built in the previous section so that it includes exploration and exploitation:

```
episode_rewards = []
epsilon=1
max_epsilon=1
min_epsilon=0.01
decay_rate=0.005
for episode in range(1000):
    state=env.reset()
    total_rewards = 0
    for step in range(50):
        exp_exp_tradeoff=random.uniform(0,1)
        ## Exploitation:
        if exp_exp_tradeoff>epsilon:
            action=np.argmax(qtable[state,:])
        else:
            ## Exploration
            action=env.action_space.sample()
        new_state,reward,done,info=env.step(action)
        qtable[state,action]+=0.9*(reward+0.9*np.max(\
                                  qtable[new_state,:])\
                                   -qtable[state,action])
        state=new_state
        total_rewards+=reward
    episode_rewards.append(total_rewards)
    epsilon=min_epsilon+(max_epsilon-min_epsilon)\
                            *np.exp(decay_rate*episode)
print(qtable)
```

The bold lines in the preceding code are what's been added to the code that was shown in the previous section. Within this code, we are specifying that, over increasing episodes, we perform more exploitation than exploration.

Once we've obtained the Q-table, we can leverage it to identify the steps that the agent needs to take to reach its destination:

```
env.reset()
for episode in range(1):
    state=env.reset()
    step=0
    done=False
    print("-----------------------")
    print("Episode",episode)
    for step in range(50):
        env.render()
        action=np.argmax(qtable[state,:])
        print(action)
        new_state,reward,done,info=env.step(action)
        if done:
            print("Number of Steps",step+1)
            break
        state=new_state
env.close()
```

In the preceding code, we are fetching the current `state` that the agent is in, identifying the `action` that results in a maximum value in the given state-action combination, taking the action (`step`) to fetch the `new_state` object that the agent would be in, and repeating these steps until the game is complete (terminated).

The preceding code results in the following output:

```
------------------
Episode 0

SFFF
FHFH
FFFH
HFFG
action:  2
  (Right)
SFFF
FHFH
FFFH
HFFG
action:  2
  (Right)
SFFF
FHFH
FFFH
HFFG
action:  1
  (Down)
SFFF
FHFH
FFFH
HFFG
action:  1
  (Down)
SFFF
FHFH
FFFH
HFFG
action:  1
  (Down)
SFFF
FHFH
FFFH
HFFG
action:  2
Number of Steps 6
```

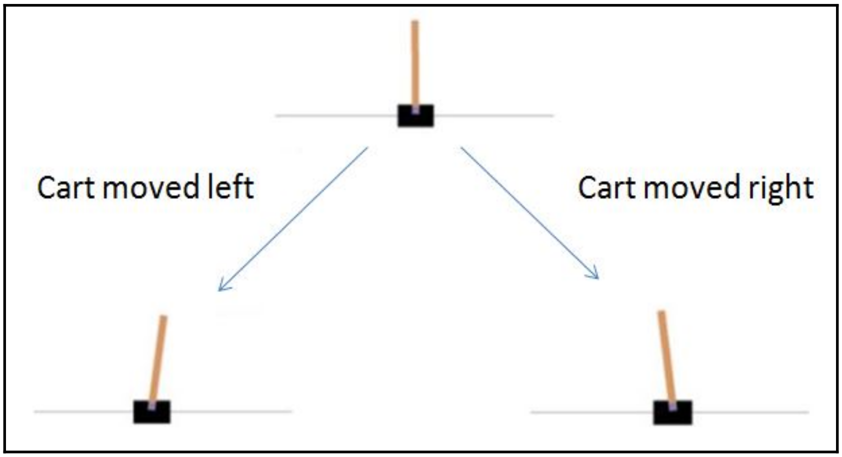
Note that this is a simplified example since the state spaces are discrete, resulting in us building a Q-table. What if the state spaces are continuous (for example, the state space is a snapshot image of a game's current state)? Building a Q-table becomes very difficult (as the number of possible states is very large). Deep Q-learning comes in handy in such a scenario. We'll learn about this in the next section.

Implementing deep Q-learning

So far, we have learned how to build a Q-table, which provides values that correspond to a given state-action combination by replaying a game – in this case, the Frozen Lake game – over multiple episodes. However, when the state spaces are continuous (such as a snapshot of a game of Pong), the number of possible state spaces becomes huge. We will address this in this section, as well as the ones to follow, using deep Q-learning. In this section, we will learn how to estimate the Q-value of a state-action combination without a Q-table by using a neural network – hence the term **deep** Q-learning.

Compared to a Q-table, deep Q-learning leverages a neural network to map any given state-action (where the state can be continuous or discrete) combination to Q-values.

For this exercise, we will work on the CartPole environment in Gym. Here, our task is to balance the CartPole for as long as possible. The following image shows what the CartPole environment looks like:



Note that the pole shifts to the left when the cart moves to the right and vice versa. Each state within this environment is defined using four observations, whose names and minimum and maximum values are as follows:

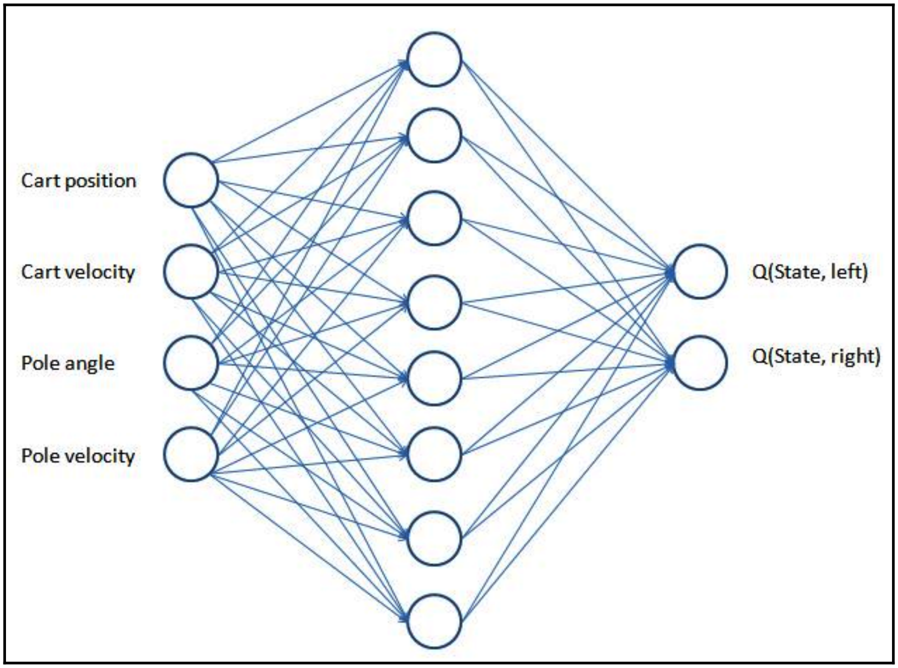
Observation	**Minimum Value**	**Maximum Value**
Cart position	-2.4	2.4
Cart velocity	-inf	inf
Pole angle	-41.8°	41.8°
Pole velocity at the tip	-inf	inf

Note that all the observations that represent a state have continuous values.

At a high level, deep Q-learning for the game of CartPole balancing works as follows:

1. Fetch the input values (image of the game/metadata of the game).
2. Pass the input values through a network that has as many outputs as there are possible actions.
3. The output layers predict the action values that correspond to taking an action in a given state.

A high-level overview of the network architecture is as follows:



In the preceding image, the network architecture uses the state (four observations) as input and the Q-value of taking left and right actions in the current state as output. We train the neural network as follows:

1. During the exploration phase, we perform a random action that has the highest value in the output layer.
2. Then, we store the action, the next state, the reward, and the flag stating whether the game was complete in memory.
3. In a given state, if the game is not complete, the Q-value of taking an action in a given state will be calculated; that is, reward + discount factor x maximum possible Q-value of all actions in the next state.
4. The Q-values of the current state-action combinations remain unchanged except for the action that is taken in *step 2*.
5. Perform *steps 1* to *4* multiple times and store the experiences.
6. Fit a model that takes the state as input and the action values as the expected outputs (from memory and replay experience) and minimize the MSE loss.
7. Repeat the preceding steps over multiple episodes while decreasing the exploration rate.

With the preceding strategy in place, let's code up deep Q-learning so that we can perform CartPole balancing:

> The following code is available as `Deep_Q_Learning_Cart_Pole_balancing.ipynb` in the `Chapter16` folder in this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from and is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components from text.

1. Import the relevant packages:

```
import gym
import numpy as np
import cv2
from collections import deque
import torch
import torch.nn as nn
import torch.nn.functional as F
import random
from collections import namedtuple, deque
```

```
import torch.optim as optim
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

2. Define the environment:

```
env = gym.make('CartPole-v1')
```

3. Define the network architecture:

```
class DQNetwork(nn.Module):
    def __init__(self, state_size, action_size):
        super(DQNetwork, self).__init__()
        self.fc1 = nn.Linear(state_size, 24)
        self.fc2 = nn.Linear(24, 24)
        self.fc3 = nn.Linear(24, action_size)
    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Note that the architecture is fairly simple since it only contains 24 units in the two hidden layers. The output layer contains as many units as there are possible actions.

4. Define the `Agent` class, as follows:

 - Define the `__init__` method with the various parameters, network, and experience defined:

```
class Agent():
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(0)

        ## hyperparameters
        self.buffer_size = 2000
        self.batch_size = 64
        self.gamma = 0.99
        self.lr = 0.0025
        self.update_every = 4

        # Q-Network
        self.local = DQNetwork(state_size, action_size)\
                                        .to(device)
        self.optimizer=optim.Adam(self.local.parameters(), \
                                        lr=self.lr)
```

```
        # Replay memory
        self.memory = deque(maxlen=self.buffer_size)
        self.experience = namedtuple("Experience", \
                          field_names=["state", "action", \
                           "reward", "next_state", "done"])
        self.t_step = 0
```

- Define the `step` function, which fetches data from memory and fits it to the model by calling the `learn` function:

```
    def step(self, state, action, reward, next_state, done):
        # Save experience in replay memory
        self.memory.append(self.experience(state, action, \
                                    reward, next_state, done))
        # Learn every update_every time steps.
        self.t_step = (self.t_step + 1) % self.update_every
        if self.t_step == 0:
        # If enough samples are available in memory,
        # get random subset and learn
            if len(self.memory) > self.batch_size:
                experiences = self.sample_experiences()
                self.learn(experiences, self.gamma)
```

- Define the `act` function, which predicts an action, given a state:

```
    def act(self, state, eps=0.):
        # Epsilon-greedy action selection
        if random.random() > eps:
            state = torch.from_numpy(state).float()\
                         .unsqueeze(0).to(device)
            self.local.eval()
            with torch.no_grad():
                action_values = self.local(state)
            self.local.train()
            return np.argmax(action_values.cpu().data.numpy())
        else:
            return random.choice(np.arange(self.action_size))
```

Note that in the preceding code, we are performing exploration-exploitation while determining the action to take.

- Define the `learn` function, which fits the model so that it predicts action values when given a state:

```
    def learn(self, experiences, gamma):
        states,actions,rewards,next_states,dones= experiences
        # Get expected Q values from local model
        Q_expected = self.local(states).gather(1, actions)
```

```
        # Get max predicted Q values (for next states)
        # from local model
        Q_targets_next = self.local(next_states).detach()\
                            .max(1)[0].unsqueeze(1)
        # Compute Q targets for current states
        Q_targets = rewards+(gamma*Q_targets_next*(1-dones))
        # Compute loss
        loss = F.mse_loss(Q_expected, Q_targets)

        # Minimize the loss
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

In the preceding code, we are fetching the sampled experiences and predicting the Q-value of the action we performed. Furthermore, given that we already know the next state, we can predict the best Q-value of the actions in the next state. This way, we now know the target value that corresponds to the action that was taken in a given state.

Finally, we'll compute the loss between the expected value (`Q_targets`) and the predicted value (`Q_expected`) of the Q-value of the action that was taken in the current state.

- Define the `sample_experiences` function in order to sample experiences from memory:

```
    def sample_experiences(self):
        experiences = random.sample(self.memory, \
                                    k=self.batch_size)
        states = torch.from_numpy(np.vstack([e.state \
                    for e in experiences if e is not None]))\
                        .float().to(device)
        actions = torch.from_numpy(np.vstack([e.action \
                    for e in experiences if e is not None]))\
                        .long().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward \
                    for e in experiences if e is not None]))\
                        .float().to(device)
        next_states=torch.from_numpy(np.vstack([e.next_state \
                    for e in experiences if e is not None]))\
                        .float().to(device)
        dones = torch.from_numpy(np.vstack([e.done \
                    for e in experiences if e is not None])\
                        .astype(np.uint8)).float().to(device)
        return (states, actions, rewards, next_states, dones)
```

5. Define the `agent` object:

```
agent = Agent(env.observation_space.shape[0], \
              env.action_space.n)
```

6. Perform deep Q-learning, as follows:

    - Initialize your lists:

```
scores = [] # list containing scores from each episode
scores_window = deque(maxlen=100) # last 100 scores
n_episodes=5000
max_t=5000
eps_start=1.0
eps_end=0.001
eps_decay=0.9995
eps = eps_start
```

    - Reset the environment in each episode and fetch the state's shape. Furthermore, reshape it so that we can pass it to a network:

```
for i_episode in range(1, n_episodes+1):
    state = env.reset()
    state_size = env.observation_space.shape[0]
    state = np.reshape(state, [1, state_size])
    score = 0
```

    - Loop through `max_t` time steps, identify the action to be performed, and perform (`step`) it. Next, reshape it so that the reshaped state is passed to the neural network:

```
    for i in range(max_t):
        action = agent.act(state, eps)
        next_state, reward, done, _ = env.step(action)
        next_state = np.reshape(next_state, [1, state_size])
```

    - Fit the model by specifying `agent.step` on top of the current state and resetting the state to the next state so that it can be useful in the next iteration:

```
        reward = reward if not done or score == 499 else -10
        agent.step(state, action, reward, next_state, done)
        state = next_state
        score += reward
        if done:
            break
```

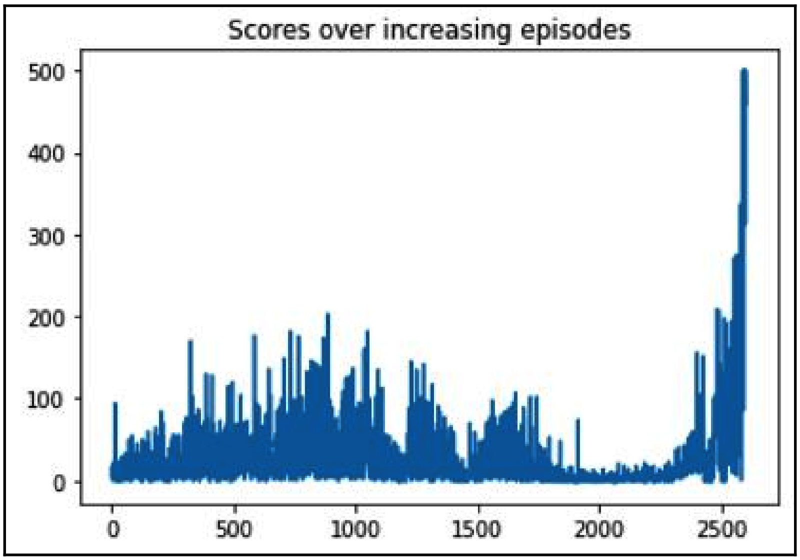
- Store, print periodically, and stop training if the mean of the scores in the previous 10 steps is greater than 450:

```
scores_window.append(score) # save most recent score
scores.append(score) # save most recent score
eps = max(eps_end, eps_decay*eps) # decrease epsilon
print('\rEpisode {}\tReward {} \tAverage Score: {:.2f} \
     \tEpsilon: {}'.format(i_episode,score, \
                          np.mean(scores_window), eps), end="")
if i_episode % 100 == 0:
    print('\rEpisode {}\tAverage Score: {:.2f} \
    \tEpsilon: {}'.format(i_episode, \
                          np.mean(scores_window), eps))
if i_episode>10 and np.mean(scores[-10:])>450:
    break
```

6. Plot the variation in scores over increasing episodes:

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(scores)
plt.title('Scores over increasing episodes')
```

A plot showing the variation of scores over episodes is as follows:



From the preceding image, we can see that, after episode 2000, the model attained a high score when balancing the CartPole.

Now that we have learned how to implement deep Q-learning, in the next section, we will learn how to work on a different state space – a video frame in Pong – instead of the four state spaces that define the state in the CartPole environment. We will also learn how to implement deep Q-learning with the fixed targets model.

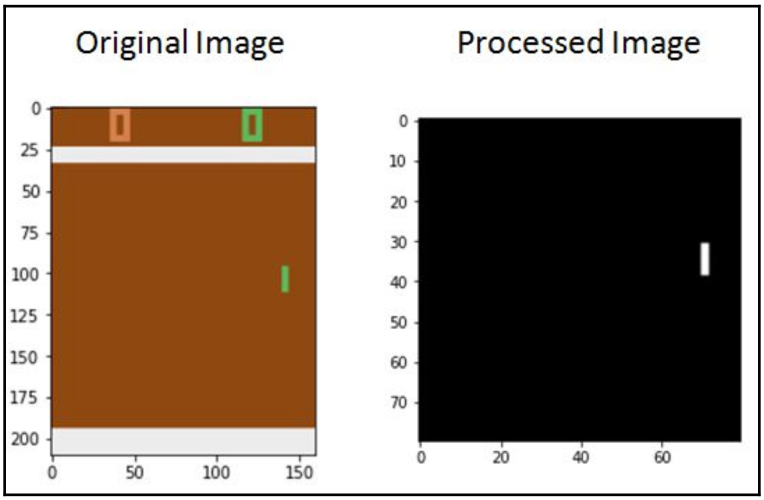
Implementing deep Q-learning with the fixed targets model

In the previous section, we learned how to leverage deep Q-learning to solve the CartPole environment in Gym. In this section, we will work on a more complicated game of Pong and understand how deep Q-learning, alongside the fixed targets model, can solve the game. While working on this use case, you will also learn how to leverage a CNN-based model (in place of the vanilla neural network we used in the previous section) to solve the problem.

The objective of this use case is to build an agent that can play against a computer (a pre-trained, non-learning agent) and beat it in a game of Pong, where the agent is expected to achieve a score of 21 points.

The strategy that we will adopt to solve the problem of creating a successful agent for the game of Pong is as follows:

Crop the irrelevant portion of the image in order to fetch the current frame (state):



Note that, in the preceding image, we have taken the original image and cropped the top and bottom pixels of the original image in the processed image:

- Stack four consecutive frames – the agent needs the sequence of states to understand whether the ball is approaching it or not.
- Let the agent play by taking random actions initially and keep collecting the current state, future state, action taken, and rewards in memory. Only keep information about the last 10,000 actions in memory and flush the historical ones beyond 10,000.
- Build a network (local network) that takes a sample of states from memory and predicts the values of the possible actions.
- Define another network (target network) that is a replica of the local network.
- Update the target network every 1,000 times the local network is updated. The weights of the target network at the end of every 1,000 epochs are the same as the weights of the local network.
- Leverage the target network to calculate the Q-value of the best action in the next state.
- For the action that the local network suggests, we expect it to predict the summation of the immediate reward and the Q-value of the best action in the next state.
- Minimize the MSE loss of the local network.
- Let the agent keep playing until it maximizes its rewards.

With the preceding strategy in place, we can now code up the agent so that it maximizes its rewards when playing Pong.

Coding up an agent to play Pong

Follow these steps to code up the agent so that it self-learns how to play Pong:

> The following code is available as `Pong_Deep_Q_Learning_with_Fixed_targets.ipynb` in the `Chapter16` folder in this book's GitHub repository - `https://tinyurl.com/mcvp-packt` The code contains URLs to download data from and is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components from text.

1. Import the relevant packages and set up the game environment:

    ```
    import gym
    import numpy as np
    import cv2
    from collections import deque
    import matplotlib.pyplot as plt
    import torch
    import torch.nn as nn
    import torch.nn.functional as F
    import random
    from collections import namedtuple, deque
    import torch.optim as optim
    import matplotlib.pyplot as plt
    %matplotlib inline

    device = 'cuda' if torch.cuda.is_available() else 'cpu'

    env = gym.make('PongDeterministic-v0')
    ```

2. Define the state size and action size:

    ```
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n
    ```

3. Define a function that will pre-process a frame so that it removes the bottom and top pixels that are irrelevant:

    ```
    def preprocess_frame(frame):
        bkg_color = np.array([144, 72, 17])
        img = np.mean(frame[34:-16:2,::2]-bkg_color,axis=-1)/255.
        resized_image = img
        return resized_image
    ```

4. Define a function that will stack four consecutive frames, as follows:

    - The function takes `stacked_frames`, the current `state`, and the flag of `is_new_episode` as input:

    ```
    def stack_frames(stacked_frames, state, is_new_episode):
        # Preprocess frame
        frame = preprocess_frame(state)
        stack_size = 4
    ```

    - If the episode is new, we will start with a stack of initial frames:

    ```
        if is_new_episode:
            # Clear our stacked_frames
    ```

```
        stacked_frames = deque([np.zeros((80,80), \
                         dtype=np.uint8) for i in \
                            range(stack_size)], maxlen=4)
        # Because we're in a new episode,
        # copy the same frame 4x
        for i in range(stack_size):
            stacked_frames.append(frame)
        # Stack the frames
        stacked_state = np.stack(stacked_frames, \
                                axis=2).transpose(2, 0, 1)
```

- If the episode is not new, we'll remove the oldest frame from `stacked_frames` and append the latest frame:

```
    else:
        # Append frame to deque,
        # automatically removes the #oldest frame
        stacked_frames.append(frame)
        # Build the stacked state
        # (first dimension specifies #different frames)
        stacked_state = np.stack(stacked_frames, \
                                axis=2).transpose(2, 0, 1)
    return stacked_state, stacked_frames
```

5. Define the network architecture; that is, `DQNetwork`:

```
class DQNetwork(nn.Module):
    def __init__(self, states, action_size):
        super(DQNetwork, self).__init__()
        self.conv1 = nn.Conv2d(4, 32, (8, 8), stride=4)
        self.conv2 = nn.Conv2d(32, 64, (4, 4), stride=2)
        self.conv3 = nn.Conv2d(64, 64, (3, 3), stride=1)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(2304, 512)
        self.fc2 = nn.Linear(512, action_size)
    def forward(self, state):
        x = F.relu(self.conv1(state))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = self.flatten(x)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

6. Define the `Agent` class, as we did in the previous section, as follows:

 - Define the `__init__` method:

```
class Agent():
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(0)

        ## hyperparameters
        self.buffer_size = 10000
        self.batch_size = 32
        self.gamma = 0.99
        self.lr = 0.0001
        self.update_every = 4
        self.update_every_target = 1000
        self.learn_every_target_counter = 0
        # Q-Network
        self.local = DQNetwork(state_size, \
                               action_size).to(device)
        self.target = DQNetwork(state_size, \
                                action_size).to(device)
        self.optimizer=optim.Adam(self.local.parameters(), \
                                  lr=self.lr)

        # Replay memory
        self.memory = deque(maxlen=self.buffer_size)
        self.experience = namedtuple("Experience", \
                    field_names=["state", "action", \
                    "reward", "next_state", "done"])
        # Initialize time step (for updating every few steps)
        self.t_step = 0
```

 Note that the only addition we've made to the `__init__` method in the preceding code, compared to the code provided in the previous section, is the `target` network and the frequency with which the target network is to be updated (these lines have been shown in bold in the preceding code).

 - Define the method that will update the weights (`step`), just like we did in the previous section:

```
    def step(self, state, action, reward, next_state, done):
        # Save experience in replay memory
        self.memory.append(self.experience(state[None], \
                                    action, reward, \
```

```
                                    next_state[None], done))
        # Learn every update_every time steps.
        self.t_step = (self.t_step + 1) % self.update_every
        if self.t_step == 0:
    # If enough samples are available in memory, get random
    # subset and learn
            if len(self.memory) > self.batch_size:
                experiences = self.sample_experiences()
                self.learn(experiences, self.gamma)
```

- Define the `act` method, which will fetch the action to be performed in a given state:

```
    def act(self, state, eps=0.):
        # Epsilon-greedy action selection
        if random.random() > eps:
            state = torch.from_numpy(state).float()\
                         .unsqueeze(0).to(device)
            self.local.eval()
            with torch.no_grad():
                action_values = self.local(state)
            self.local.train()
            return np.argmax(action_values.cpu()\
                                      .data.numpy())
        else:
            return random.choice(np.arange(self.action_size))
```

- Define the `learn` function, which will train the local model:

```
    def learn(self, experiences, gamma):
        self.learn_every_target_counter+=1
        states,actions,rewards,next_states,dones = experiences
        # Get expected Q values from local model
        Q_expected = self.local(states).gather(1, actions)

        # Get max predicted Q values (for next states)
        # from target model
        Q_targets_next = self.target(next_states).detach()\
                             .max(1)[0].unsqueeze(1)
        # Compute Q targets for current state
        Q_targets = rewards+(gamma*Q_targets_next*(1-dones))
        # Compute loss
        loss = F.mse_loss(Q_expected, Q_targets)

        # Minimize the loss
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

```
        # ----------- update target network ------------- #
        if self.learn_every_target_counter%1000 ==0:
            self.target_update()
```

Note that, in the preceding code, `Q_targets_next` is predicted using the target model instead of the local model that was used in the previous section. We are also updating the target network after every 1,000 steps, where `learn_every_target_counter` is the counter that helps in identifying whether we should update the target model.

- Define a function (`target_update`) that will update the target model:

```
    def target_update(self):
        print('target updating')
        self.target.load_state_dict(self.local.state_dict())
```

- Define a function that will sample experiences from memory:

```
    def sample_experiences(self):
        experiences = random.sample(self.memory, \
                                    k=self.batch_size)
        states = torch.from_numpy(np.vstack([e.state \
                    for e in experiences if e is not None]))\
                                    .float().to(device)
        actions = torch.from_numpy(np.vstack([e.action \
                    for e in experiences if e is not None]))\
                                    .long().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward \
                    for e in experiences if e is not None]))\
                                    .float().to(device)
        next_states=torch.from_numpy(np.vstack([e.next_state \
                     for e in experiences if e is not None]))\
                                    .float().to(device)
        dones = torch.from_numpy(np.vstack([e.done \
                    for e in experiences if e is not None])\
                        .astype(np.uint8)).float().to(device)
        return (states, actions, rewards, next_states, dones)
```

7. Define the `Agent` object:

```
agent = Agent(state_size, action_size)
```

8. Define the parameters that will be used to train the agent:

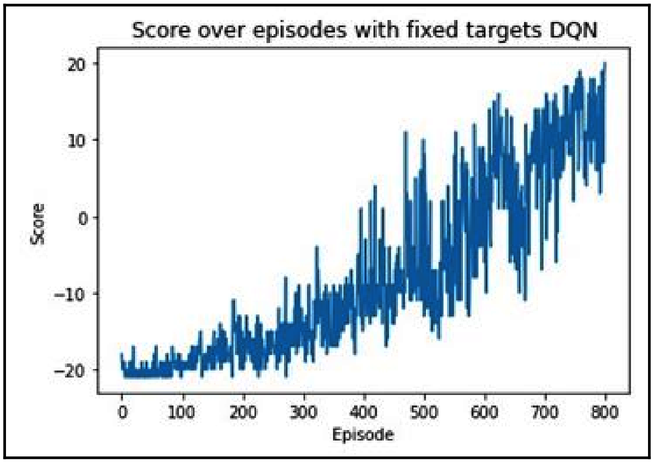
```
n_episodes=5000
max_t=5000
eps_start=1.0
eps_end=0.02
```

```
eps_decay=0.995
scores = [] # list containing scores from each episode
scores_window = deque(maxlen=100) # last 100 scores
eps = eps_start
stack_size = 4
stacked_frames = deque([np.zeros((80,80), dtype=np.int) \
                        for i in range(stack_size)], \
                       maxlen=stack_size)
```

9. Train the agent over increasing episodes, as we did in the previous section:

```
for i_episode in range(1, n_episodes+1):
    state = env.reset()
    state, frames = stack_frames(stacked_frames, \
                                 state, True)
    score = 0
    for i in range(max_t):
        action = agent.act(state, eps)
        next_state, reward, done, _ = env.step(action)
        next_state, frames = stack_frames(frames, \
                                          next_state, False)
        agent.step(state, action, reward, next_state, done)
        state = next_state
        score += reward
        if done:
            break
    scores_window.append(score) # save most recent score
    scores.append(score) # save most recent score
    eps = max(eps_end, eps_decay*eps) # decrease epsilon
    print('\rEpisode {}\tReward {} \tAverage Score: {:.2f} \
    \tEpsilon: {}'.format(i_episode,score,\
                          np.mean(scores_window),eps),end="")
    if i_episode % 100 == 0:
        print('\rEpisode {}\tAverage Score: {:.2f} \
        \tEpsilon: {}'.format(i_episode, \
                              np.mean(scores_window), eps))
```

The following plot shows the variation of scores over increasing episodes:



From the preceding image, we can see that the agent gradually learned to play Pong and that by the end of 800 episodes, it learned how to play Pong while receiving a high reward.

Now that we've trained an agent to play Pong well, in the next section, we will train an agent so that it can drive a car autonomously in a simulated environment.

Implementing an agent to perform autonomous driving

Now that you have seen RL working in progressively challenging environments, we will conclude this chapter by demonstrating that the same concepts can be applied to a self-driving car. Since it is impractical to see this working on an actual car, we will resort to a simulated environment. The environment is going to be a full-fledged city of traffic, with cars and additional details within the image of a road. The actor (agent) is a car. The inputs to the car are going to be various sensory inputs such as a dashcam, **Light Detection And Ranging** (**LIDAR**) sensors, and GPS coordinates. The outputs are going to be how fast/slow the car will move, along with the level of steering. This simulation will attempt to be an accurate representation of real-world physics. Thus, note that the fundamentals will remain the same, whether it is a car simulation or a real car.

Note that the environment we are going to install needs a **graphical user interface** (**GUI**) to display the simulation. Also, the training will take at least a day, if not more. Because of the non-availability of a visual setup and the time usage limits of Google-Colab, we will not be using Google-Colab notebooks as we have been doing so far. This is the only section of this book that requires an active Linux operating system, and preferably a GPU to achieve acceptable results in a few days of training.

Installing the CARLA environment

As we mentioned previously, we need an environment that can simulate complex interactions to make us believe that we are, in fact, dealing with a realistic scenario. CARLA is one such environment. The environment author stated the following about CARLA:

> *"CARLA has been developed from the ground up to support development, training, and validation of autonomous driving systems. In addition to open source code and protocols, CARLA provides open digital assets (urban layouts, buildings, and vehicles) that were created for this purpose and can be used freely. The simulation platform supports flexible specification of sensor suites, environmental conditions, full control of all static and dynamic actors, maps generation, and much more."*

There are two steps we need to follow to install the environment:

1. Install the CARLA binaries for the simulation environment.
2. Install the Gym version, which provides Python connectivity for the simulation environment.

The steps for this section have been presented as a video walkthrough here: `https://tinyurl.com/mcvp-self-driving-agent`.

Let's get started!

Install the CARLA binaries

In this section, we will learn how to install the necessary CARLA binaries:

1. Visit `https://github.com/carla-simulator/carla/releases/tag/0.9.6` and download the `CARLA_0.9.6.tar.gz` compiled version file.
2. Move it to a location where you want CARLA to live in your system and unzip it. Here, we are demonstrating this by downloading and unzipping CARLA into the `Documents` folder:

```
$ mv CARLA_0.9.6.tar.gz ~/Documents/
$ cd ~/Documents/
$ tar -xf CARLA_0.9.6.tar.gz
$ cd CARLA_0.9.6/
```

3. Add CARLA to `PYTHONPATH` so that any module on your machine can import CARLA:

```
$ echo "export
PYTHONPATH=$PYTHONPATH:/home/$(whoami)/Documents/CARLA_0.9.6/P
ythonAPI/carla/dist/carla-0.9.6-py3.5-linux-x86_64.egg" >>
~/.bashrc
```

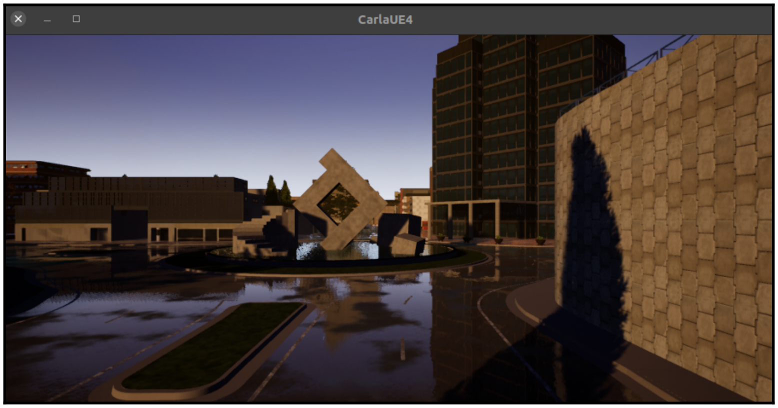
In the preceding code, we added the directory containing CARLA to a global variable called `PYTHONPATH`, which is an environment variable for accessing all Python modules. Adding it to `~/.bashrc` will ensure that every time a Terminal is opened, it can access this new folder. After running the preceding code, restart the Terminal and run `ipython -c "import carla; carla.__spec__"`. You should get the following output:

```
╰─ ipython -c "import carla; carla.__spec__"
Out[1]: ModuleSpec(name='carla', loader=<_frozen_importlib_external.SourceFileLo
ader object at 0x7fbb31646590>, origin='/home/yyr/anaconda3/lib/python3.7/site-p
ackages/carla-0.9.6-py3.5-linux-x86_64.egg/carla/__init__.py', submodule_search_
locations=['/home/yyr/anaconda3/lib/python3.7/site-packages/carla-0.9.6-py3.5-li
nux-x86_64.egg/carla'])
```

4. Finally, provide the necessary permissions and execute CARLA, as follows:

```
$ chmod +x /home/$(whoami)/Documents/CARLA_0.9.6/CarlaUE4.sh
$ ./home/$(whoami)/Documents/CARLA_0.9.6/CarlaUE4.sh
```

After a minute or two, you should see a window similar to the following showing CARLA running as a simulation, ready to take inputs:



In this section, we've verified that `CARLA` is a simulation environment whose binaries are working as expected. Let's move on to installing the Gym environment for it. Leave the Terminal running as-is since we need the binary to be running in the background throughout this exercise.

Installing the CARLA Gym environment

Since there is no official Gym environment, we will take advantage of a user implemented GitHub repository and install the Gym environment for CARLA from there. Follow these steps to install CARLA's Gym environment:

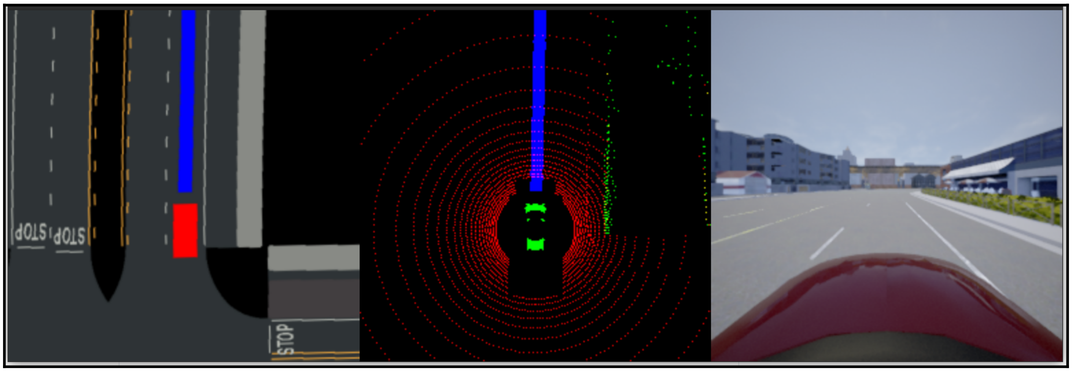
1. Clone the Gym repository to a location of your choice and install the library:

```
$ cd /location/to/clone/repo/to
$ git clone https://github.com/cjy1992/gym-carla
$ cd gym-carla
$ pip install -r requirements.txt
$ pip install -e .
```

2. Test your setup by running the following command:

```
$ python test.py
```

A window similar to the following should open, showing that we have added a fake car to the environment. From here, we can monitor the top view, the LIDAR sensor point cloud, and our dashcam:



Here, we can observe the following:

- The first view contains a view that is very similar to what vehicle GPS systems show in a car; that is, our vehicle, the various waypoints, and the road lanes. However, we shall not use this input for training as it also shows other cars in the view, which is unrealistic.
- The second view is more interesting. Some consider it as the eye of a self-driving car. LIDAR emits pulsed light into the surrounding environment (in all directions), multiple times every second. It captures the reflected light to determine how far the nearest obstacle is in that direction. The onboard computer collates all the nearest obstacle information to recreate a 3D point cloud that gives it a 3D understanding of its environment.
- In both the first and second views, we can see there is a strip ahead of the car. This is a waypoint indication of where the car is supposed to go.
- The third view is a simple dashboard camera.

Apart from these three, CARLA provides additional sensor data, such as the following:

- `lateral-distance` (deviation from the lane it should be in)
- `delta-yaw` (angle with respect to the road ahead)
- `speed`
- If there's a hazardous obstacle in front of the vehicle
- And many more...

We are going to use the first four sensors mentioned previously, along with LIDAR and our dashcam, to train the model.

We are now ready to understand the components of CARLA and create a DQN model for a self-driving car.

Training a self-driving agent

We will create two files before we start the training process in a notebook; that is, `model.py` and `actor.py`. These will contain the model architecture and the `Agent` class, respectively. The `Agent` class contains the various methods we'll use to train an agent.

> The code instructions for this section are present in the `Chapter16` folder of this book's GitHub repository as `Carla.md`.

model.py

This is going to be a PyTorch model that will accept the image that's provided to it, as well as other sensor inputs. It will be expected to return the most likely action:

```
from torch_snippets import *

class DQNetworkImageSensor(nn.Module):
    def __init__(self):
        super().__init__()
        self.n_outputs = 9
        self.image_branch = nn.Sequential(
                            nn.Conv2d(3, 32, (8, 8), stride=4),
                            nn.ReLU(inplace=True),
                            nn.Conv2d(32, 64, (4, 4), stride=2),
                            nn.ReLU(inplace=True),
                            nn.Conv2d(64,128,(3, 3),stride=1),
                            nn.ReLU(inplace=True),
                            nn.AvgPool2d(8),
                            nn.ReLU(inplace=True),
                            nn.Flatten(),
                            nn.Linear(1152, 512),
                            nn.ReLU(inplace=True),
                            nn.Linear(512, self.n_outputs)
                        )
```

```
        self.lidar_branch = nn.Sequential(
                            nn.Conv2d(3, 32, (8, 8), stride=4),
                            nn.ReLU(inplace=True),
                            nn.Conv2d(32,64,(4, 4),stride=2),
                            nn.ReLU(inplace=True),
                            nn.Conv2d(64,128,(3, 3),stride=1),
                            nn.ReLU(inplace=True),
                            nn.AvgPool2d(8),
                            nn.ReLU(inplace=True),
                            nn.Flatten(),
                            nn.Linear(1152, 512),
                            nn.ReLU(inplace=True),
                            nn.Linear(512, self.n_outputs)
                        )

        self.sensor_branch = nn.Sequential(
                                nn.Linear(4, 64),
                                nn.ReLU(inplace=True),
                                nn.Linear(64, self.n_outputs)
                            )

    def forward(self, image, lidar=None, sensor=None):
        x = self.image_branch(image)
        if lidar is None:
            y = 0
        else:
            y = self.lidar_branch(lidar)
        z = self.sensor_branch(sensor)

        return x + y + z
```

As you can see, there are more types of data being fed into the forward method than in the previous sections, where we were just accepting an image as input. `self.image_branch` is going to expect the image coming from the dashcam of the car, while `self.lidar_branch` will accept the image that's generated by the LIDAR sensor. Finally, `self.sensor_branch` will accept four sensor inputs in the form of a NumPy array. These four items are the lateral-distance (deviation from the lane it is supposed to be in), delta-yaw (angle with respect to the road ahead), speed, and if there's a hazardous obstacle at the front of the vehicle, respectively. See line number 543 in `gym_carla/envs/carla_env.py` (the repository that has been git cloned) for the same outputs. Using a different branch in the neural network will let the module provide different levels of importance for each sensor, and the outputs are summed up as the final output. Note that there are 9 outputs; we will look at these later.

actor.py

Much like the previous sections, we will use some code to store replay information and play it back when training is necessary:

1. Let's get the imports and hyperparameters in place:

```
import numpy as np
import random
from collections import namedtuple, deque
import torch
import torch.nn.functional as F
import torch.optim as optim
from model1 import DQNetworkImageSensor

BUFFER_SIZE = int(1e3) # replay buffer size
BATCH_SIZE = 256 # minibatch size
GAMMA = 0.99 # discount factor
TAU = 1e-2 # for soft update of target parameters
LR = 5e-4 # learning rate
UPDATE_EVERY = 50 # how often to update the network
ACTION_SIZE = 2

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

2. Next, we'll initialize the target and local networks. No changes have been made to the code from the previous section here, except for the module that is being imported:

```
class Actor():
    def __init__(self):
        # Q-Network
        self.qnetwork_local=DQNetworkImageSensor().to(device)
        self.qnetwork_target=DQNetworkImageSensor().to(device)
        self.optimizer = optim.Adam(self.qnetwork_local\
                                    .parameters(),lr=LR)

        # Replay memory
        self.memory= ReplayBuffer(ACTION_SIZE,BUFFER_SIZE, \
                                  BATCH_SIZE, 10)
        # Initialize time step
        # (for updating every UPDATE_EVERY steps)
        self.t_step = 0
    def step(self, state, action, reward, next_state, done):
        # Save experience in replay memory
        self.memory.add(state, action, reward, \
                        next_state, done)
        # Learn every UPDATE_EVERY time steps.
```

```
        self.t_step = (self.t_step + 1) % UPDATE_EVERY
        if self.t_step == 0:
  # If enough samples are available in memory,
  # get random subset and learn
            if len(self.memory) > BATCH_SIZE:
                experiences = self.memory.sample()
                self.learn(experiences, GAMMA)
```

3. Since there are more sensors to handle, we'll transport them as a dictionary of state. The state contains the `'image'`, `'lidar'`, and `'sensor'` keys, which we introduced in the previous section. We perform preprocessing before sending them to the neural network, as shown in the following code:

```
    def act(self, state, eps=0.):
        images,lidars sensors=state['image'], \
                              state['lidar'],state['sensor']
        images = torch.from_numpy(images).float()\
                      .unsqueeze(0).to(device)
        lidars = torch.from_numpy(lidars).float()\
                      .unsqueeze(0).to(device)
        sensors = torch.from_numpy(sensors).float()\
                       .unsqueeze(0).to(device)
        self.qnetwork_local.eval()
        with torch.no_grad():
            action_values = self.qnetwork_local(images, \
                                  lidar=lidars, sensor=sensors)
        self.qnetwork_local.train()
        # Epsilon-greedy action selection
        if random.random() > eps:
            return np.argmax(action_values.cpu().data.numpy())
        else:
            return random.choice(np.arange(\
                        self.qnetwork_local.n_outputs))
```

4. Now, we need to fetch items from replay memory. The following instructions are being executed in the following code:
    1. Obtain a batch of current and next states.
    2. Compute the expected reward, `Q_expected`, if a network performs actions in the current state.
    3. Compare it with the target reward, `Q_targets`, that would have been obtained when the next state was fed to the network.
5. Periodically update the target network with the local network:

```
    def learn(self, experiences, gamma):
        states,actions,rewards,next_states,dones= experiences
        images, lidars, sensors = states
```

```
        next_images, next_lidars, next_sensors = next_states
        # Get max predicted Q values (for next states)
        # from target model
        Q_targets_next = self.qnetwork_target(next_images, \
                       lidar=next_lidars,sensor=next_sensors)\
                            .detach().max(1)[0].unsqueeze(1)
        # Compute Q targets for current states
        Q_targets = rewards +(gamma*Q_targets_next*(1-dones))

        # Get expected Q values from local model
        # import pdb; pdb.set_trace()
        Q_expected=self.qnetwork_local(images,lidar=lidars, \
                   sensor=sensors).gather(1,actions.long())
        # Compute loss
        loss = F.mse_loss(Q_expected, Q_targets)
        # Minimize the loss
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # ------------ update target network ------------- #
        self.soft_update(self.qnetwork_local, \
                         self.qnetwork_target, TAU)

    def soft_update(self, local_model, target_model, tau):
        for target_param, local_param in \
            zip(target_model.parameters(), \
            local_model.parameters()):
            target_param.data.copy_(tau*local_param.data + \
                                (1.0-tau)*target_param.data)
```

6. The only major change in the `ReplayBuffer` class is going to be how the data is stored. Since we have multiple sensors, each memory (`states` and `next_states`) is stored as a tuple of data; that is, `states = [images, lidars, sensors]`:

```
class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""
    def __init__(self, action_size, buffer_size, \
                 batch_size, seed):
        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", \
                          field_names=["state", "action", \
                                     "reward","next_state", \
                                       "done"])
```

```
        self.seed = random.seed(seed)
    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, \
                            next_state, done)
        self.memory.append(e)
    def sample(self):
        experiences = random.sample(self.memory, \
                                    k=self.batch_size)
        images = torch.from_numpy(np.vstack(\
                    [e.state['image'][None] \
                 for e in experiences if e is not None]))\
                    .float().to(device)
        lidars = torch.from_numpy(np.vstack(\
                    [e.state['lidar'][None] \
                 for e in experiences if e is not None]))\
                    .float().to(device)
        sensors = torch.from_numpy(np.vstack(\
                    [e.state['sensor'] \
                 for e in experiences if e is not None]))\
                    .float().to(device)
        states = [images, lidars, sensors]
        actions = torch.from_numpy(np.vstack(\
                    [e.action for e in experiences \
                     if e is not None])).long().to(device)
        rewards = torch.from_numpy(np.vstack(\
                    [e.reward for e in experiences \
                     if e is not None])).float().to(device)
        next_images = torch.from_numpy(np.vstack(\
                    [e.next_state['image'][None] \
                     for e in experiences if e is not None]))\
                    .float().to(device)
        next_lidars = torch.from_numpy(np.vstack(\
                    [e.next_state['lidar'][None] \
                     for e in experiences if e is not None]))\
                    .float().to(device)
        next_sensors = torch.from_numpy(np.vstack(\
                    [e.next_state['sensor'] \
                     for e in experiences if e is not None]))\
                    .float().to(device)
        next_states = [next_images, next_lidars, next_sensors]
        dones = torch.from_numpy(np.vstack([e.done \
                     for e in experiences if e is not None])\
                    .astype(np.uint8)).float().to(device)

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
```

```
        """Return the current size of internal memory."""
        return len(self.memory)
```

Note that the lines of code in bold fetch the current states, actions, rewards, and next states' information.

Now that the critical components are in place, let's load the Gym environment into a Python notebook and start training.

Training DQN with fixed targets

There is no additional theory we need to learn here. The basics remain the same; we'll only be making changes to the Gym environment, the architecture of the neural network, and the actions our agent needs to take:

1. First, load the hyperparameters associated with the environment. Refer to each comment beside every key-value pair presented in the `params` dictionary in the following code. Since we are simulating a complex environment, we need to choose the environment's parameters, such as the number of cars in the city, number of walkers, which town to simulate, the resolution of the dashcam image, and LIDAR sensors:

```
import gym
import gym_carla
import carla
from model import DQNetworkState
from actor import Actor
from torch_snippets import *

params = {
    'number_of_vehicles': 10,
    'number_of_walkers': 0,
    'display_size': 256, # screen size of bird-eye render
    'max_past_step': 1, # the number of past steps to draw
    'dt': 0.1, # time interval between two frames
    'discrete': True, # whether to use discrete control space
    # discrete value of accelerations
    'discrete_acc': [-1, 0, 1],
    # discrete value of steering angles
    'discrete_steer': [-0.3, 0.0, 0.3],
    # define the vehicle
    'ego_vehicle_filter': 'vehicle.lincoln*',
    'port': 2000, # connection port
    'town': 'Town03', # which town to simulate
    'task_mode': 'random', # mode of the task
```

```
    'max_time_episode': 1000, # maximum timesteps per episode
    'max_waypt': 12, # maximum number of waypoints
    'obs_range': 32, # observation range (meter)
    'lidar_bin': 0.125, # bin size of lidar sensor (meter)
    'd_behind': 12, # distance behind the ego vehicle (meter)
    'out_lane_thres': 2.0, # threshold for out of lane
    'desired_speed': 8, # desired speed (m/s)
    'max_ego_spawn_times': 200, # max times to spawn vehicle
    'display_route': True, # whether to render desired route
    'pixor_size': 64, # size of the pixor labels
    'pixor': False, # whether to output PIXOR observation
}

# Set gym-carla environment
env = gym.make('carla-v0', params=params)
```

In the preceding `params` dictionary, the following are important for our simulation in terms of the action space:

- `'discrete': True`: Our actions lie in a discrete space.
- `'discrete_acc':[-1,0,1]`: All the possible accelerations the self-driven car is allowed to make during the simulation.
- `'discrete_steer':[-0.3,0,0.3]`: All the possible steering magnitudes, the self-driven car is allowed to make during the simulation.

As you can see, the `discrete_acc` and `discrete_steer` lists contain three items each. This means that there are 3 x 3 possible unique actions the car can take. This means that the network in the `model.py` file has nine discrete states.

Feel free to change the parameters once you've gone through the official documentation.

2. With that, we have all the components we need to train the model. Load a pre-trained model, if one exists:

```
load_path = None # 'car-v1.pth'
# continue training from an existing model
save_path = 'car-v2.pth'

actor = Actor()
if load_path is not None:
    actor.qnetwork_local.load_state_dict(\
                                torch.load(load_path))
```

```
    actor.qnetwork_target.load_state_dict(\
                                torch.load(load_path))
else:
    pass
```

3. Fix the number of episodes and define the `dqn` function to train the agent, as follows:

    - Reset the state:

```
n_episodes = 100000
def dqn(n_episodes=n_episodes, max_t=1000, eps_start=1, \
        eps_end=0.01, eps_decay=0.995):
    scores = [] # list containing scores from each episode
    scores_window = deque(maxlen=100) # last 100 scores
    eps = eps_start # Initialize epsilon
    for i_episode in range(1, n_episodes+1):
        state = env.reset()
```

    - Wrap the state into a dictionary (as discussed in the `actor.py:Actor` class) and `act` on it:

```
        image, lidar, sensor = state['camera'], \
                               state['lidar'], \
                               state['state']
        image, lidar = preprocess(image), preprocess(lidar)
        state_dict = {'image': image, 'lidar': lidar, \
                      'sensor': sensor}
        score = 0
        for t in range(max_t):
            action = actor.act(state_dict, eps)
```

    - Store the next state that's obtained from the environment and then store the `state, next_state` pair (along with the rewards and other state information) to train the actor using DQN:

```
            next_state, reward, done, _ = env.step(action)
            image, lidar, sensor = next_state['camera'], \
                                   next_state['lidar'], \
                                   next_state['state']
            image,lidar = preprocess(image), preprocess(lidar)
            next_state_dict = {'image':image,'lidar':lidar, \
                               'sensor': sensor}
            actor.step(state_dict, action, reward, \
                       next_state_dict, done)
            state_dict = next_state_dict
            score += reward
```

```
    if done:
        break
scores_window.append(score) # save most recent score
scores.append(score) # save most recent score
eps = max(eps_end, eps_decay*eps) # decrease epsilon
if i_episode % 100 == 0:
    log.record(i_episode, \
               mean_score=np.mean(scores_window))
    torch.save(actor.qnetwork_local.state_dict(), \
               save_path)
```

We must repeat the loop until we get a done signal, after which we reset the environment and start storing actions once again. After every 100 episodes, store the model.

4. Call the `dqn` function to train the model:

```
dqn()
```

Since this is a more complex environment, training can take a few days, so be patient and continue training a few hours at a time using the `load_path` and `save_path` arguments. With enough training, the vehicle can maneuver and learn how to drive by itself. Here's a video of the training result we were able to achieve after two days of training: `https://tinyurl.com/mcvp-self-driving-agent-result`.

Summary

In this chapter, we learned how the values of various actions in a given state are calculated. We then learned how the agent updates the Q-table using the discounted value of taking an action in a given state. In the process of doing this, we learned how the Q-table is infeasible in a scenario where the number of states is high. We also learned how to leverage deep Q-networks to address the scenario where the number of possible states is high. Next, we moved on to leveraging CNN-based neural networks while building an agent that learned how to play Pong using DQN based on fixed targets. Finally, we learned how to leverage DQN with fixed targets to perform self-driving using the CARLA simulator. As we have seen repeatedly in this chapter, you can use deep Q-learning to learn very different tasks – such as CartPole balancing, playing Pong, and self-driving navigation – with almost the same code. While this is not the end of our journey into exploring RL, at this point, we should be able to appreciate how we can use CNN-based and reinforcement learning-based algorithms together to solve complex problems and build learning agents.

So far, we have learned how to combine computer vision-based techniques with techniques from other prominent areas of research, including meta-learning, natural language processing, and reinforcement learning. Apart from this, we've also learned how to perform object classification, detection, segmentation, and image generation using GANs. In the next chapter, we will switch gears and learn how to move a deep learning model to production.

Questions

1. How is a value calculated for a given state?
2. How is a Q-table populated?
3. Why do we have a discount factor in the state-action value calculation?
4. What do we need the exploration-exploitation strategy?
5. Why do we need to use deep Q-learning?
6. How is the value of a given state-action combination calculated using deep Q-learning?
7. Once the agent has maximized the reward in the CartPole environment, is there a chance that it can learn a sub-optimal policy later?

17
Moving a Model to Production

Moving a model to production is a step toward enabling the consumption of our model by an external party. We should expose our model to the world and start rendering predictions on real, unseen input.

It is not sufficient to have a trained PyTorch model for deployment. We need additional server components for creating communication channels from the real world to the PyTorch model and back to the real world. It is important that we know how to create an API (through which the user can interact with the model), wrap it as a self-contained application (so that it can be deployed on any computer), and ship it to the cloud – so that anybody with the required URL and credentials can interact with the model. To successfully move a model to production, all these steps are necessary. In this chapter, we will deploy a simple application that is accessible from anywhere on the internet. We will also learn about deploying the **Fashion MNIST** (**FMNIST**) model and letting any user upload the picture they want to classify and fetch results.

The following topics will be covered in this chapter:

- Understanding the basics of an API
- Creating an API and making predictions on a local server
- Moving the API to the cloud

Understanding the basics of an API

By now, we know how to create a deep learning model for various tasks. It accepts/returns tensors as input/output. But an outsider such as a client/end user would talk only in terms of images and classes. Furthermore, they would expect to send and receive input/output over channels that might have nothing to do with Python. The internet is the easiest channel to communicate on. Hence, for a client, the best-case deployment scenario would be if we can set up a publically available URL and ask them to upload their images there. One such paradigm is called an **Application Programming Interface** (**API**), which has standard protocols that accept input and post output over the internet while abstracting the user away from how the input is processed or the output is generated.

Some common protocols in APIs are POST, GET, PUT, and DELETE, which are sent as **requests** by the client to the host server along with relevant data. Based on the request and data, the server performs the relevant task and returns appropriate data in the form of a **response** – which the client can use in their downstream tasks. In our case, the client will send a POST request with an image of clothing as a file attachment. We should save the file, process it, and return the appropriate FMNIST class as a response to the request, and our job is done.

Requests are organized data packets sent over the internet to communicate with API servers. Typically, the components in a request are as follows:

- **An endpoint URL**: This would be the address of the API service. For example, `https://www.packtpub.com/` would be an endpoint to connect to the Packt Publishing service and browse through the catalog of their latest books.
- **A collection of headers**: This information helps the API server return output; if the header contained information that the client is on mobile, then the API can return an HTML page with a layout that is mobile-friendly.
- **A collection of queries** so that only related items from the server database are fetched. For example, a search string of `PyTorch` will return only PyTorch-related books, in the previous example. (In this chapter, we will not work on queries as a prediction on images does not require querying – it requires the filename.)
- **A list of files** that could be uploaded to the server, or in our case, be used to make deep learning predictions.

`cURL` is a computer software project providing a library and command-line tool for transferring data using various network protocols. It is one of the most lightweight, commonly used, and simple applications to call API requests and get back responses.

We will use a readily available Python module called `Fast-API` that will enable us to do the following:

1. Set up a communication URL.
2. Accept input from a wide variety of environments/formats when it is sent to the URL.
3. Convert every form of input into the exact format that the machine learning model needs as input.
4. Make predictions with the trained deep learning-based model.
5. Convert predictions into the right format and respond to the client's request with the prediction.

We will use the FMNIST classifier as an example to demonstrate these concepts.

After understanding the basic setup and code, you can create APIs for any kind of deep learning task and serve predictions through a URL on your local machine. While this is a logical end to creating an application, it is equally important that we deploy it somewhere that is accessible by anyone who does not have access to our computer or the model.

In the next two sections, we will cover how to wrap the application in a self-contained Docker image that can be shipped and deployed anywhere on the cloud. Once a Docker image is prepared, a container can be created from it and be deployed on any major cloud service provider, as all of them accept Docker as standard input. We will specifically walk through the example of deploying the FMNIST classifier on an **Amazon Web Services** (**AWS**) **Elastic Compute Cloud** (**EC2**) instance in the last section of this chapter. Let's use FastAPI, a Python library, in the next section to create the API and verify that we can make predictions directly from the terminal (without Jupyter notebooks).

Creating an API and making predictions on a local server

In this section, we will learn about making predictions on a local server (that has nothing to do with the cloud). At a high level, this involves the following steps:

1. Installing FastAPI
2. Creating a route to accept incoming requests
3. Saving an incoming request on disk
4. Loading the requested image, then preprocessing and predicting with the trained model
5. Postprocessing the results and sending back the predictions as a response to the same incoming request

All of the steps in this section are summarized as a video walk-through here: `https://tinyurl.com/MCVP-Model2FastAPI`.

Let's begin by installing FastAPI in the following subsection.

Installing the API module and dependencies

Since FastAPI is a Python module, we can use `pip` for installation, and be ready to code an API. We will now open a new terminal and run the following command:

```
$pip install fastapi uvicorn aiofiles jinja2
```

We have installed a couple more dependencies that are needed with FastAPI. `uvicorn` is a minimal low-level server/application interface for setting up APIs. `aiofiles` enables the server to work asynchronously with requests, such as accepting and responding to multiple independent parallel requests at the same time. These two modules are dependencies for FastAPI and we will not directly interact with them.

Let's create the required files and code them in the next section.

Serving an image classifier

The first step is to set up a folder structure as shown here:

The setup is quite minimal, as shown here:

- The `files` folder is going to act as the download location for incoming requests.
- `fmnist.weights.pth` contains the weights of our trained FMNIST model.
- `fmnist.py` will contain logic to load the weights, accept incoming images, preprocess, predict, and postprocess the predictions.
- `server.py` will contain FastAPI functionalities that can set up a URL, accept client requests from the URL, send/receive input/output from `fmnist.py`, and send the output as responses to the client requests.

Note the following:
The `files` folder is empty and is only used to store uploaded files.
We are assuming we have the weights of the trained model as `fmnist.weights.pth`.

Let's understand what `fmnist.py` and `server.py` constitute and code them now.

fmnist.py

As discussed, the `fmnist.py` file should have the logic to load the model and return predictions of a given image.

We are already familiar with how to create a PyTorch model. The only additional component to the class is the `predict` method, which is there for doing any necessary preprocessing on the image and postprocessing on the results.

In the following code, we are first creating the model class that constitutes the architecture of the model, which is initialized with the optimal weights through `torch.load`:

```
from torch_snippets import *

device = 'cuda' if torch.cuda.is_available() else 'cpu'

class FMNIST(nn.Module):
    classes = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress',
    'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
    def __init__(self, fpath='fmnist.weights.pth'):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(28 * 28, 1000),
            nn.ReLU(),
            nn.Linear(1000, 10)
        ).to(device)
        self.model.load_state_dict(torch.load(fpath))
        logger.info('Loaded FMNIST Model')
```

The following code block highlights the `forward` method:

```
    @torch.no_grad()
    def forward(self, x):
        x = x.view(1, -1).to(device)
        pred = self.model(x)
        pred = F.softmax(pred, -1)[0]
        conf, clss = pred.max(-1)
        clss = self.classes[clss.cpu().item()]
        return conf.item(), clss
```

The following code block highlights the `predict` method to do the necessary preprocessing and postprocessing:

```
    def predict(self, path):
        image = cv2.imread(path,0)
        x = np.array(image)
        x = cv2.resize(x, (28,28))
        x = torch.Tensor(255 - x)/255.
        conf, clss = self.forward(x)
        return {'class': clss, 'confidence': f'{conf:.4f}'}
```

In the `__init__` method, we are initializing the model and loading the pre-trained weights. In the `forward` method, we are passing an image through the model and fetching predictions. In the `predict` method, we are loading an image from a pre-defined path, preprocessing the image before passing it through the `forward` method of the model, and wrapping the output in a dictionary while returning the predicted class and its confidence.

server.py

This is the portion of code in the API that connects the user's request with the PyTorch model. Let's create the file step by step:

1. Load the libraries:

   ```
   import os, io
   from fmnist import FMNIST
   from PIL import Image
   from fastapi import FastAPI, Request, File, UploadFile
   ```

   `FastAPI` is the base server class that will be used to create an API.

   `Request`, `File`, and `UploadFile` are proxy placeholders for a client request and the files they will upload. For more details, you are encouraged to go through the official FastAPI documentation.

2. Load the model:

   ```
   # Load the model from fmnist.py
   model = FMNIST()
   ```

3. Create an `app` model that can supply us with a URL for uploading and displaying:

   ```
   app = FastAPI()
   ```

4. Create a URL at `"/predict"` so that the client can send `POST` requests to "<hosturl>/predict" (we will learn about <hosturl>, which is the server, in the next section) and receive responses:

   ```
   @app.post("/predict")
   def predict(request: Request, file:UploadFile=File(...)):
       content = file.file.read()
       image = Image.open(io.BytesIO(content)).convert('L')
       output = model.predict(image)
       return output
   ```

That's it! We have all the components to leverage our image classifier to make predictions over our local server. Let's set up the server and make some predictions over the local server.

Running the server

Now that we have set all the components up, we are ready to run the server. Open a new terminal and `cd` the folder that contains `fmnist.py`, `server.py`:

1. Run the server:

   ```
   $ uvicorn server:app
   ```

   You will see a message like so:

```
2020-10-06 06:54:27.748 | INFO     | fmnist:__init__:16 - Loaded FMNIST Model
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:5000 (Press CTRL+C to quit)
```

   The `Uvicorn running on ...` message indicates that the server is up and running.

2. To fetch predictions, we will run the following in the terminal to fetch predictions for a sample image present in `/home/me/Pictures/shirt.png`:

   ```
   $ curl -X POST "http://127.0.0.1:8000/predict" -H "accept:
   application/json" -H "Content-Type: multipart/form-data" -F
   "file=@/home/me/Pictures/shirt.png;type=image/png"
   ```

The major components of the preceding line of code are as follows:

- **API protocol**: The protocol we are calling is `POST`, which indicates that we want to send our own data to the server.
- **URL – server address**: The server host URL is `http://127.0.0.1:8000/` (which is the local server and `8000` is the default port) and `/predict/` is the route given to the client to create `POST` requests; future clients must upload their data to the URL – `http://127.0.0.1:8000/predict`.

- **Headers**: The request has components in the form of `-H` flags. These explain additional information, such as the following:
    - What the input content type is going to be – `multipart/form-data` – which is API jargon for saying the input data is in the form of a file.
    - What the expected output type is – `application/json` – which means the JSON format. There are other formats, such as XML, text, and octet-stream, which are applicable based on the complexity of the output being generated.
- **Files**: The final `-F` flag is pointing to the location where the file that we want to upload exists, and what its type is.

The output dictionary, once we run the preceding code, will be printed in the terminal:

```
└─ curl -X POST "http://127.0.0.1:8000/predict" -H "accept: application/json" -H "Content-Type: multipart/form-data" -F "file=@/home/yyr/Pictures/shirt.png;type=image/png"
{"class":"Coat","confidence":"0.6488"}%
```

We can now fetch model predictions from our local server. In the next section, we will look at fetching model predictions from the cloud so that any user can get model predictions.

Moving the API to the cloud

So far, we have learned about making predictions on a local server (`http://127.0.0.1` is a URL of the local server that cannot be accessed on the web) – so, only the owner of the local machine can use the model. In this section, we will learn about moving this model to the cloud so that anyone can predict using an image.

In general, companies deploy services in redundant machines to ensure reliability and there is little control over the hardware provided by the cloud provider. It is not convenient to keep track of all folders and their code, or copy-paste all the code, then install all the dependencies, ensuring the code works as expected on the new environment, and forward ports on all the cloud machines. There are too many steps to be followed for the same code on every new machine. Repeating these steps is a waste of time for the developer and such a process is highly prone to mistakes.

We would rather install one package that has everything than install multiple individual packages (such as the individual modules and code required to run the application) and connect them later. Thus, it becomes important that we are able to wrap the entire code base and modules into a single package (something like a `.exe` file in Windows) so that the package can be deployed with as little as one command and still ensure it works exactly the same on all hardware. To this end, we need to learn how to work with Docker – which is essentially a condensed operating system with code. The created Docker containers are lightweight and will perform only the tasks that we want them to perform. In our example, the Docker image we will create will run the API for the task of predicting the class of FMNIST images. But first, let's understand some Docker jargon.

Comparing Docker containers and Docker images

A **Docker image** is a standard unit of software that packages up code and all its dependencies. This way, the application runs quickly and reliably from one computing environment to another. A Docker image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings.

A **Docker container** is a snapshot of the image that will be instantiated wherever it needs to be deployed. We can create any number of Docker image copies from a single image and they are all expected to perform the same task. Think of an image as the parent copy and a container as the child copy.

At a high level, we will perform the following tasks:

1. Create a Docker image. Create a Docker container out of it and test it.
2. Push the Docker image to the cloud.
3. Build the Docker container on the cloud.
4. Deploy the Docker container on the cloud.

Creating a Docker container

In the previous section, we built an API that takes an image and returns the class of the image and the probability associated with that class of image on a local server. Now, it's time to wrap our API in a package that can be shipped and deployed anywhere.

Ensure Docker is installed on your machine. You can refer to `https://docs.docker.com/get-docker/` for instructions on the installation.

There are four steps to this process of creating a Docker container:

1. Create a `requirements.txt` file.
2. Create a Dockerfile.
3. Build a Docker image.
4. Create a Docker container from the image and test it.

The code in the following sections is also summarized as a video walkthrough here: `https://tinyurl.com/MCVP-Model2FastAPI`. The relevant part of this section starts at the 2-minute mark in the video.

We will go through and understand these four steps now, and in the next section, we will learn how to ship the image to AWS servers.

Creating the requirements.txt file

We need to tell the Docker image which Python modules to install to run the application. The `requirements.txt` file contains a list of all these Python modules:

1. Open a terminal and go to the folder that contains `fmnist.py`, `server.py`. Next, we will create a blank virtual environment and activate it in our local terminal in the root folder:

```
$ python3 -m venv fastapi-venv
$ source fastapi-env/bin/activate
```

   The reason why we create a blank virtual environment is to ensure that *only* the required modules are installed in the environment so that when shipping, we don't waste valuable space.

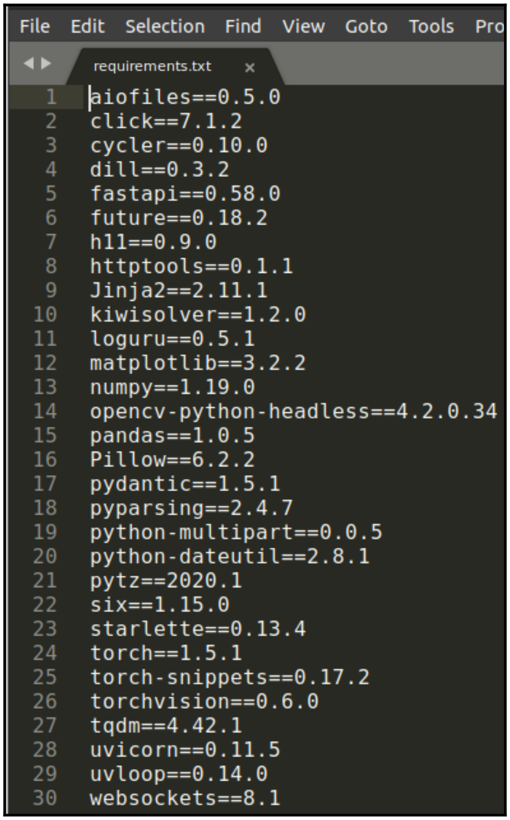
2. Install the required packages (`fastapi`, `uvicorn`, `aiofiles`, `torch`, and `torch_snippets`) to run the FMNIST app:

```
$ pip install fastapi uvicorn aiofiles torch torch_snippets
```

3. In the same terminal, run the following command to install all the required Python modules:

   ```
   $ pip freeze > requirements.txt
   ```

   The preceding code fetches all the Python modules and their corresponding version numbers into the `requirements.txt` file, which will be used for installing dependencies in the Docker image:




```
aiofiles==0.5.0
click==7.1.2
cycler==0.10.0
dill==0.3.2
fastapi==0.58.0
future==0.18.2
h11==0.9.0
httptools==0.1.1
Jinja2==2.11.1
kiwisolver==1.2.0
loguru==0.5.1
matplotlib==3.2.2
numpy==1.19.0
opencv-python-headless==4.2.0.34
pandas==1.0.5
Pillow==6.2.2
pydantic==1.5.1
pyparsing==2.4.7
python-multipart==0.0.5
python-dateutil==2.8.1
pytz==2020.1
six==1.15.0
starlette==0.13.4
torch==1.5.1
torch-snippets==0.17.2
torchvision==0.6.0
tqdm==4.42.1
uvicorn==0.11.5
uvloop==0.14.0
websockets==8.1
```

We can open the text file, which would look similar to the previous screenshot. Now that we have all the pre-requisites, let's create the Dockerfile in the next section.

Creating a Dockerfile

As introduced in the preceding section, the Docker image is a self-contained application, complete with its own operating system and dependencies. Given a computation platform (such as an EC2 instance), the image can act independently and perform the tasks that it is designed to perform. For this, we need to provide a Docker application with the necessary instructions – dependencies, code, and commands – to launch applications. Let's create these instructions in a text file called `Dockerfile` in the root directory that contains `server.py`, `fmnist.py` – which we already placed after creating the project folder – of our FMINST project. The file needs to be named `Dockerfile` (no extension) as a convention. The content of the text file is as follows:

```
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.7
COPY ./requirements.txt /app/requirements.txt
RUN pip install --no-cache-dir -r requirements.txt
WORKDIR /app
COPY . /app
EXPOSE 5000
CMD ["uvicorn", "server:app", "--host", "0.0.0.0"]
```

Let's understand the preceding code step by step:

1. `FROM` is instructing which operating system base to use. The `tiangolo/uvicorn-gunicorn-fastapi:python3.7` location is an address that is parsed by Docker from the internet and it fetches a base image that has already installed Python and other FastAPI modules.
2. Next, we are copying the `requirements.txt` file that we created. This provides the packages that we want to install. In the next line, we are asking the image to `pip install` the packages.
3. `WORKDIR` is the folder where our application will be running. Hence, we are creating a new folder named `/app` in the Docker image and copying the contents of the root folder into the `/app` folder of the image.
4. Finally, we run the server as we did in the previous section.

This way, we have set up a blueprint to create a completely new operating system and filesystem (think of it as a new Windows installable CD) from scratch, which is going to contain only the code that we specify and run only one application, which is FastAPI.

Building a Docker image and creating a Docker container

Note that so far, we have only created a blueprint for the Docker image. Let's build the image and create a container out of it.

Run the following commands from the same terminal (where we are in the root directory containing the application files):

1. Build the Docker image and tag it as `fmnist:latest`:

   ```
   $ docker build -t fmnist:latest .
   ```

   After a long list of outputs, we get the following, telling us that the image is built:

   ```
   Step 11/12 : EXPOSE 5000
    ---> Running in 8b3bec49d6ea
   Removing intermediate container 8b3bec49d6ea
    ---> 862971ca0081
   Step 12/12 : CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "5000"]
    ---> Running in 4c94059a61ac
   Removing intermediate container 4c94059a61ac
    ---> 94fc46d82744
   Successfully built 94fc46d82744
   Successfully tagged fmnist:latest
   ```

   We have successfully created a Docker image with the name `fmnist:latest` (where `fmnist` is the image name and `latest` is a tag that we gave, indicating its version number). Docker maintains a registry in the system from which all these images are accessible. This Docker registry now contains a standalone image with all the code and logic to run the FMNIST API.

   We can always check in the Docker registry by typing out `$ docker image ls` in Command Prompt:

```
└ docker image ls
REPOSITORY          TAG       IMAGE ID       CREATED            SIZE
fmnist              latest    be6e6a4cdc99   8 minutes ago      6.38GB
```

2. Run the built image with `-p 5000:5000` forwarding port `5000` from inside the image to port `5000` on our local machine. The last argument is the name of the container being created from the image:

   ```
   $ docker run -p 5000:5000 fmnist:latest
   ```

   Port forwarding is important. Often, we don't have a say on which ports the cloud is exposing. Hence, as a matter of demonstration, even though our `uvicorn` model created a `5000` port for the `POST` operation, we are still using Docker's functionality to route external requests from `5000` to `5000`, which is where `uvicorn` is listening.

   This should give a prompt with the last few lines as follows:

```
└─ docker run -p 5000:5000 fmnist:latest
2020-11-08 14:13:50.095 | WARNING  | torch_snippets:<module>:5 - torch library is not found. Skipping to
rch imports and loading only utilities
2020-11-08 14:13:50.216 | INFO     | fmnist:__init__:16 - Loaded FMNIST Model
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:5000 (Press CTRL+C to quit)
INFO:     172.17.0.1:51788 - "POST /predict HTTP/1.1" 200 OK
```

3. Now, run a `curl` request from a new terminal and access the API as described in the previous section, but this time, the application is being served from Docker instead:

```
└─ curl -X POST "http://127.0.0.1:5000/predict" -H "accept: application/json" -H "Content-Type: multipar
t/form-data" -F "file=@/home/yyr/Pictures/boot.png;type=image/png"
{"class":"Ankle boot","confidence":"0.7668"}%
```

Even though we have not moved anything to the cloud so far, wrapping the API in Docker enables us to not have to worry about `pip install` or copy-pasting code ever again. Next, we'll ship it to a cloud provider and make the app available to the world.

You can now ship the image to any computer that also has Docker. No matter what type of computer we ship it to, calling `docker run` will always create a container that will work exactly the way we intend it to. We need not worry about `pip install` or copy-pasting code anymore.

Shipping and running the Docker container in the cloud

We will rely on AWS for our cloud requirements. We will use two of AWS's free offerings for our purpose:

- **Elastic Container Registry** (**ECR**): Here, we will store our Docker image.
- **EC2**: Here, we will create a Linux system to run our API Docker image.

For this section, let's focus only on the ECR part of it. A high-level overview of the steps we will follow to push the Docker image to the cloud is as follows:

1. Configure AWS on the local machine.
2. Create a Docker repository on AWS ECR and push the `fmnist:latest` image.
3. Create an EC2 instance.
4. Install dependencies on the EC2 instance.
5. Create and run the Docker image on the EC2 instance.

> The code in the following sections is also summarized as a video walkthrough here: `https://tinyurl.com/MCVP-FastAPI2AWS`.

Let's implement the preceding steps, starting with configuring AWS in the next section.

Configuring AWS

We are going to log in to AWS from Command Prompt and push our Docker image. Let's do it step by step:

1. Create an AWS account at `https://aws.amazon.com/` and log in.
2. Install the AWS CLI on your local machine (which contains the Docker image).

> The AWS CLI is a command-line interface application for all Amazon services. It should be installed from the official website for your operating system first. Visit `https://docs.aws.amazon.com/cli/latest/userguide/install-cliv2.html` for more details.

3. Verify that it is installed by running `aws --version` in your local terminal.
4. Configure the AWS CLI. Get the following tokens from `https://aws.amazon.com/`:
    - `aws_account_id`
    - Access key ID
    - Secret access key
    - Region

We can find all the preceding variables in the **Identity and Access Management** (**IAM**) section in AWS. Run `aws configure` in the terminal and give the appropriate credentials when asked:

```
$ aws configure
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key [None]:wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [None]: region
Default output format [None]:json
```

We have now logged in to Amazon's services from our computer. We can, in principle, access any of their services directly from the terminal. In the next section, let's connect to ECR and push the Docker image.

Creating a Docker repository on AWS ECR and pushing the image

Now, we will create the Docker repository, as follows:

1. After configuring, log in to AWS ECR using the following command (the following code is all one line), providing the preceding region and account ID details at the places that are given in bold in the following code:

    ```
    $ aws ecr get-login-password --region region | docker login --
    username AWS --password-stdin
    aws_account_id.dkr.ecr.region.amazonaws.com
    ```

    The preceding line of code creates and connects you to your own Docker registry in the Amazon cloud. Much like the Docker registry in the local system, this is where the images are going to reside, but instead, it will be in the cloud.

2. Create a repository from the CLI by running the following:

   ```
   $ aws ecr create-repository --repository-name fmnist_app
   ```

   With the preceding code, a location is now created in the cloud that can hold your Docker images.

3. Tag your local image by running the following command so that when you push the image, it will be pushed to the tagged repository. Remember to give your own `aws_accound_id` and `region` values in the bolded part of the following code:

   ```
   $ docker tag fmnist:latest
   aws_account_id.dkr.ecr.region.amazonaws.com/fmnist_app
   ```

4. Run the following command to push the local Docker image to the AWS repository in the cloud:

   ```
   $ docker push
   aws_account_id.dkr.ecr.region.amazonaws.com/fmnist_app
   ```

We have successfully created a location in the cloud for our API and pushed the Docker image to this location. As you are now aware, this image already has all the components to run the API. The only remaining aspect is to create a Docker container out of it in the cloud, and we will have successfully moved our application to production!

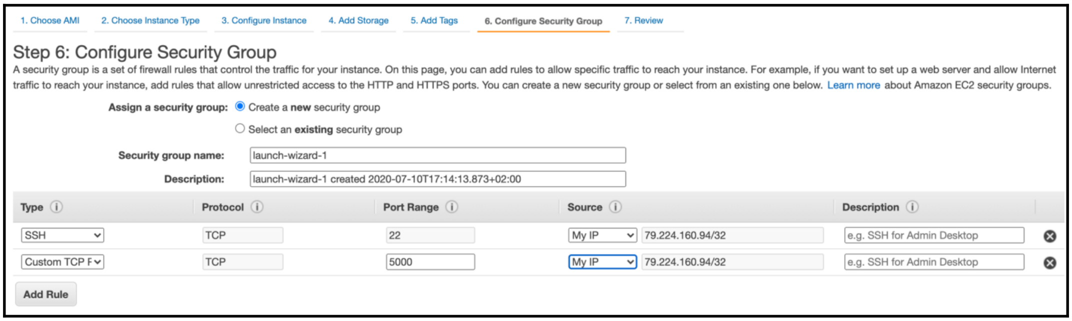
Creating an EC2 instance

Pushing the Docker image to AWS ECR is like pushing code to a GitHub repository. It just resides in one place and we still need to build the application out of it.

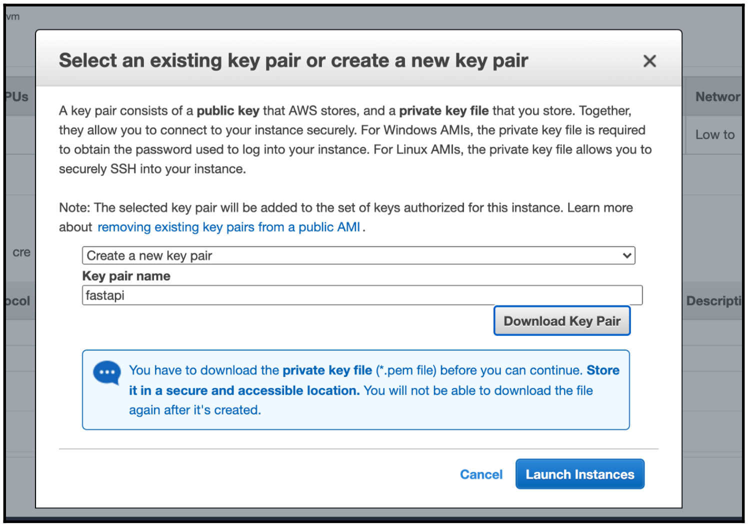
For this, you have to create an Amazon EC2 instance that can serve your web application:

1. Go to the search bar of the AWS Management Console and search for `EC2`.
2. Select **Launch Instance**.
3. You will be given a list of the instances that are available. AWS offers many instances in the free tier. We chose the **Amazon Linux 2 AMI - t2.micro** instance with 20 GB of space here (you can use other instances as well but remember to change the configuration accordingly).

4. While configuring the instance creation, in the **Configure Security Group** section, add a rule with **Custom TCP** set, and set **Port Range** as `5000` (as we have exposed port `5000` in the Docker image), as shown here:



5. In the **Launch Instances** popup (see the following screenshot), which is the last step, create a new key pair (this will download a `.pem` file, which is needed for logging into the instance). This is as good as a password, so do not lose this file:



6. Move the `.pem` file to a secure location and change its permissions by running `chmod 400 fastapi.pem`.

   You should see an instance running in your EC2 dashboard at this point:

7. Copy the EC2 instance name that looks like this:

   ```
   ec2-18-221-11-226.us-east-2.compute.amazonaws.com
   ```

8. Log in to the EC2 instance by using the following command in your local terminal:

   ```
   $ ssh -i fastapi.pem ec2-user@ec2-18-221-11-226.us-
   east-2.compute.amazonaws.com
   ```

We have created an EC2 instance with the necessary space and operating system. Furthermore, we were able to expose port `8000` from the machine and could also take note of the public URL for this machine (this URL will be used by the client for sending `POST` requests). Finally, we were able to log in to it by successfully using the downloaded `.pem` file, treating the EC2 machine like any other machine that can install software.

Pulling the image and building the Docker container

Let's install the dependencies for running the Docker image on the EC2 machine, and then we'll be ready to run the API. The following commands are all needed to be run in the EC2 console that we have logged in to in the previous section (*step 8* of the previous section):

1. Install and configure the Docker image on a Linux machine:

   ```
   $ sudo yum install -y docker
   $ sudo groupadd docker
   $ sudo gpasswd -a ${USER} docker
   $ sudo service docker restart
   ```

   `groupadd` and `gpasswd` ensure that Docker has all the permissions required to run.

2. Configure AWS in an EC2 instance, as you did earlier, and reboot the machine:

```
$ aws configure
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key
[None]:wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [None]: us-west-2
Default output format [None]:json
$ reboot
```

3. Log in again to the instance from the local terminal using the following command:

```
$ ssh -i fastapi.pem ec2-user@ec2-18-221-11-226.us-
east-2.compute.amazonaws.com
```

4. Now, from the EC2 logged-in console (which has Docker installed), log in to AWS ECR (change the region that is present in bold in the following code):

```
$ aws ecr get-login --region region --no-include-email
```

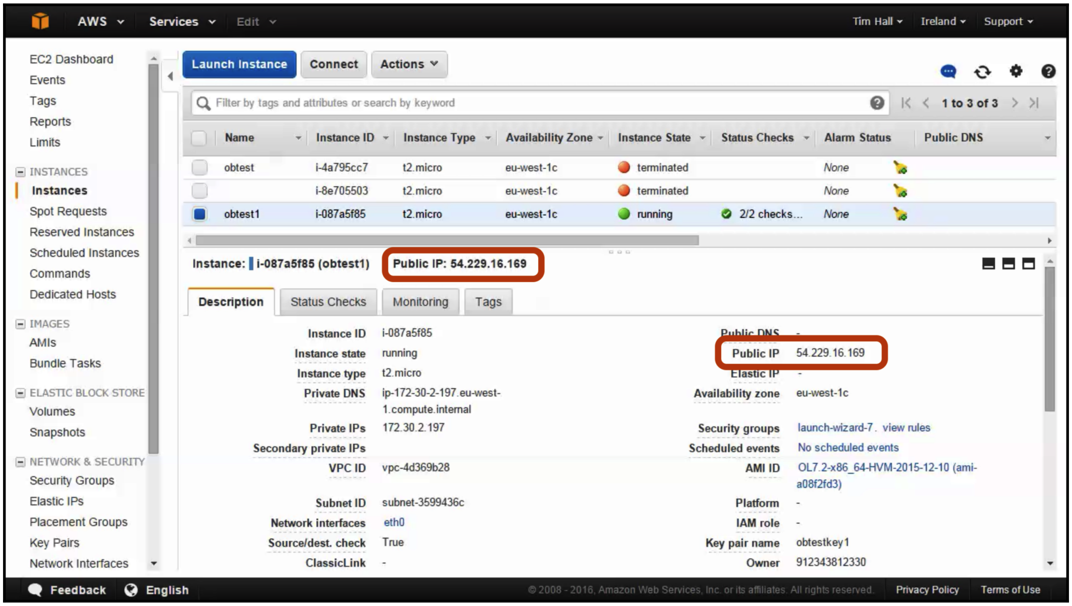
5. Copy the output from the preceding code, then paste and run it in the command line. Once you are successfully logged in to AWS ECR, you will see **Login Succeeded** in the console.
6. Pull the Docker image from AWS ECR:

```
$ docker pull
aws_account_id.dkr.ecr.region.amazonaws.com/fmnist_app:latest
```

7. Finally, run the pulled Docker image in the EC2 machine:

```
docker run -p 5000:5000
aws_account_id.dkr.ecr.region.amazonaws.com/fmnist_app
```

We have our API running on EC2. All we have to do is get the public IP address for the machine and run the `curl` request with this address in place of `127.0.0.1`. You can find this address on the EC2 dashboard at the right of the page:



8. You can now call a `POST` request from any computer, and the EC2 instance will respond to it, giving us predictions for what type of clothing image we have uploaded:

```
$ curl -X POST "http://54.229.16.169:5000/predict" -H "accept:
application/json" -H "Content-Type: multipart/form-data" -F
"file=@/home/me/Pictures/shirt.png;type=image/png"
```

The preceding code results in the following output:

```
{"class":"Coat","confidence":"0.6488"}
```

In this section, we were able to install the dependencies for EC2, pull the Docker image, and run the Docker container, to enable any user with the URL to make predictions on a new image.

Summary

In this chapter, we learned what additional steps are required in moving a model to production. We learned what an API is and what its components are. After creating an API, with the use of FastAPI, we glanced at the core steps of creating a Docker image of the API. Using AWS, we created our own Docker registry in the cloud and went through the steps to push our Docker image there. We saw what it takes to create an EC2 instance and install the required libraries to pull the Docker image from ECR, build a Docker container from it, and deploy it for any user to make predictions.

In the next and final chapter, we will learn about OpenCV, which has utilities that help in addressing some of the image-related problems in a constrained environment. We will go through five different use cases to gain an understanding of leveraging OpenCV for image analysis. Learning the functionalities of OpenCV will further strengthen our computer vision repertoire.

18
Using OpenCV Utilities for Image Analysis

So far, in previous chapters, we have learned about leveraging various techniques to perform object classification, localization, and segmentation, as well as generating images. While all these techniques leverage deep learning to solve tasks, for relatively simple and well-defined tasks, we can leverage specific functionalities provided in the OpenCV package. For example, we don't need YOLO if the object that needs to be detected is always the same object with the same background. In cases where images come from a constrained environment, there is a high chance that one of the OpenCV utilities can help solve the problem to a large extent.

We are going to cover only a few use cases in this chapter as there are just so many utilities to cover that it would warrant a separate book focusing on OpenCV. In doing word detection, you will learn about image dilation, erosion, and extracting contours around connected components. After that, you will learn about Canny edge detection to identify edges of objects within an image. Furthermore, you will understand the advantage of having a green screen in the background of videos/images while performing a bitwise operation on images to identify the color space of interest. Then, you will understand a technique that helps in creating a panoramic view of two images by stitching them together. Finally, you will learn about leveraging pre-trained cascade filters to identify objects such as number plates.

In this chapter, we will learn about the following topics:

- Drawing bounding boxes around words in an image
- Detecting lanes in an image of a road
- Detecting objects based on color
- Building a panoramic view of images
- Detecting the number plate of a car

Drawing bounding boxes around words in an image

Imagine a scenario where you are building a model that performs word transcription from the image of a document. The first step would be to identify the location of words within the image. Primarily, there are two ways of identifying words within an image:

- Using deep learning techniques such as CRAFT, EAST, and more
- Using OpenCV-based techniques

In this section, we will learn about how machine-printed words can be identified in a clean image without leveraging deep learning. As the contrast between the background and foreground is high, you do not need an overkill solution such as YOLO to identify the location of individual words. Using OpenCV is going to be especially handy in these scenarios because we can arrive at a solution with very limited computational resources and, consequently, even the inference time will be very small. The only drawback is that the accuracy may not be 100%, but that is also subject to how clean the scanned images are. If the scans are guaranteed to be very, very clear, then you can expect a near 100% accuracy.

At a high level, let's understand how we can identify/isolate words within an image:

1. Convert the image to grayscale, as the color does not affect identifying words within an image.
2. Dilate the content in the image slightly. Dilation bleeds the black pixels into the immediate neighborhood and hence connects the black pixels between characters of the same word. This helps in ensuring that characters that belong to the same word are connected. However, do not dilate so much that characters that belong to different adjacent words also get connected.
3. Once the characters are connected, leverage the `cv2.findContours` method to draw a bounding box around each word.

Let's code up the preceding strategy:

The following code is available as `Drawing_bounding_boxes_around_words_in_an_image.ipynb` in the `Chapter18` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` Be sure to copy the URL from the notebook in GitHub to avoid any issue while reproducing the results

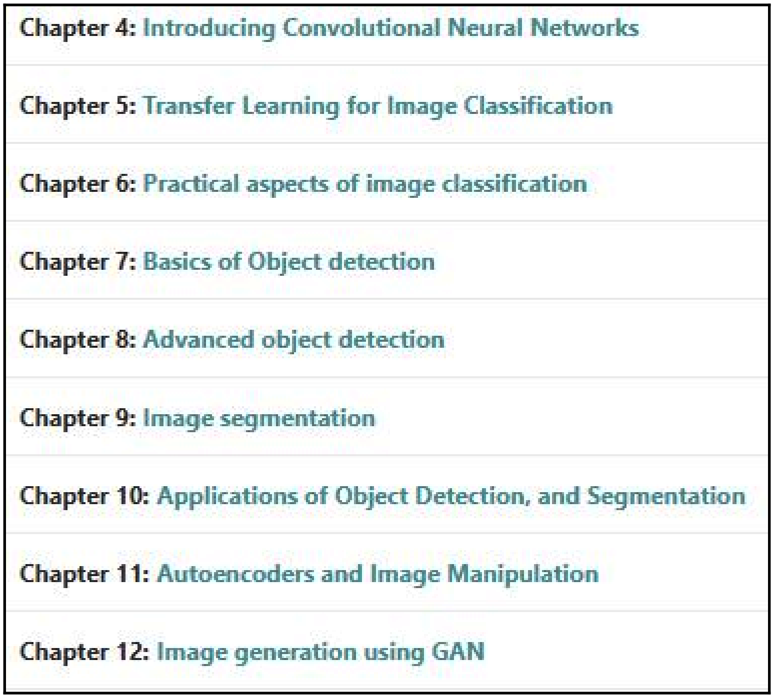
1. Let's start by downloading a sample image:

```
!wget https://www.dropbox.com/s/3jkwy16m6xdlktb/18_5.JPG
```

2. View the downloaded image using the following lines of code:

```
import cv2, numpy as np
img = cv2.imread('18_5.JPG')
img1 = cv2.cvtColor(img, cv2.COLOR_RGB2BGR)
import matplotlib.pyplot as plt,cv2
%matplotlib inline
plt.imshow(img1)
```

The preceding code will return the following output:



Chapter 4: Introducing Convolutional Neural Networks

Chapter 5: Transfer Learning for Image Classification

Chapter 6: Practical aspects of image classification

Chapter 7: Basics of Object detection

Chapter 8: Advanced object detection

Chapter 9: Image segmentation

Chapter 10: Applications of Object Detection, and Segmentation

Chapter 11: Autoencoders and Image Manipulation

Chapter 12: Image generation using GAN

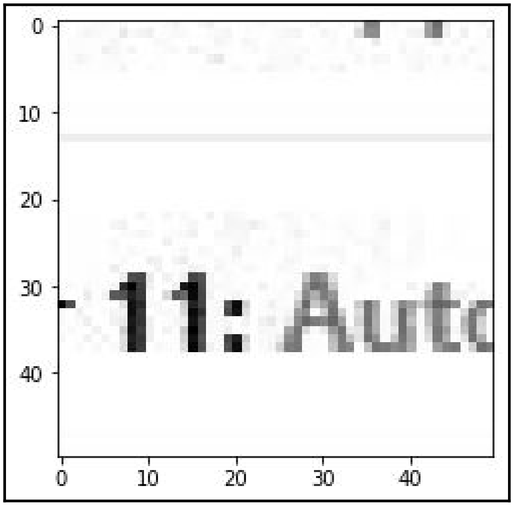
3. Convert the input image into a grayscale image:

```
img_gray = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
```

4. Fetch a random crop of the original image:

```
crop = img_gray[250:300,50:100]
plt.imshow(crop,cmap='gray')
```

The preceding code results in the following output:



From the preceding output, we can see that there are a few pixels that contain noise. Next, we will remove the noise present in the original image.

5. Binarize the input grayscale image:

```
_img_gray = np.uint8(img_gray < 200)*255
```

The preceding code results in the pixels that have a value of less than 200 having a value of `0`, while the pixels that are bright (have a pixel intensity greater than 200) have a value of `255`.

6. Find contours of the various characters present in the image:

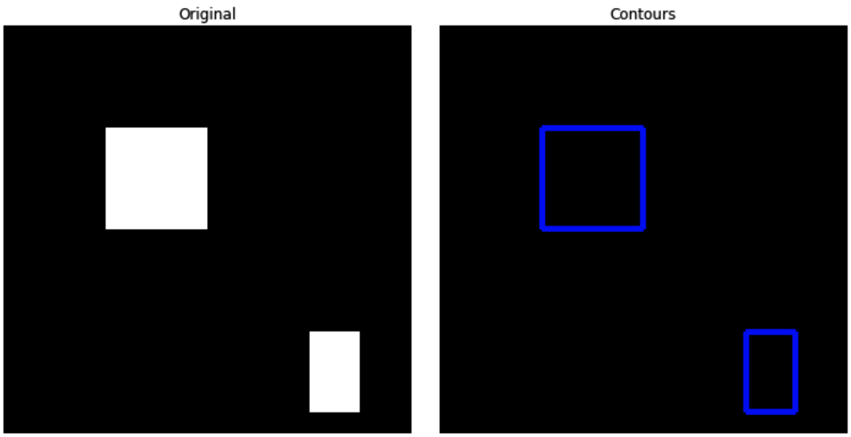
```
contours,hierarchy=cv2.findContours(_img_gray, \
                    cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)
```

`cv2` finds contours by creating a collection of a continuous set of pixels as a single blob of the object. Refer to the following screenshot for an idea of how `cv2.findContours` works:

```
from torch_snippets import *

im = np.zeros((200,200))
im[50:100,50:100] = 255
im[150:190,150:175] = 255

im = uint(im)
contours,hierarchy = cv2.findContours(im,cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)
im_tmp = C(np.zeros_like(im)) # create a fake 3 channel image
cv2.drawContours(im_tmp, contours, -1, (0,0,255), thickness=2) # draw contours on the image
subplots([im, im_tmp], titles='Original,Contours'.split(','), nc=2,figsize=(10,5))
```

executed in 122ms, finished 14:49:46 2020-09-22

```
2020-09-22 14:49:46.554 | INFO     | torch_snippets.loader:subplots:347 - plotting 2 images in a grid of 1x2 @ (10, 5)
```



7. Convert the threshold image obtained previously to have three channels so that we can plot the colored bounding boxes around characters:

   ```
   thresh1 = np.stack([_img_gray]*3,axis=2)
   ```

8. Create a blank image so that we can copy the relevant content from `thresh1` into the new image:

   ```
   thresh2 = np.zeros((thresh1.shape[0],thresh1.shape[1]))
   ```

9. Fetch the contours obtained in the previous step and draw a bounding box with a rectangle where the contour is mentioned. Also, copy the content corresponding to the bounding rectangle from the `thresh1` image to `thresh2`:

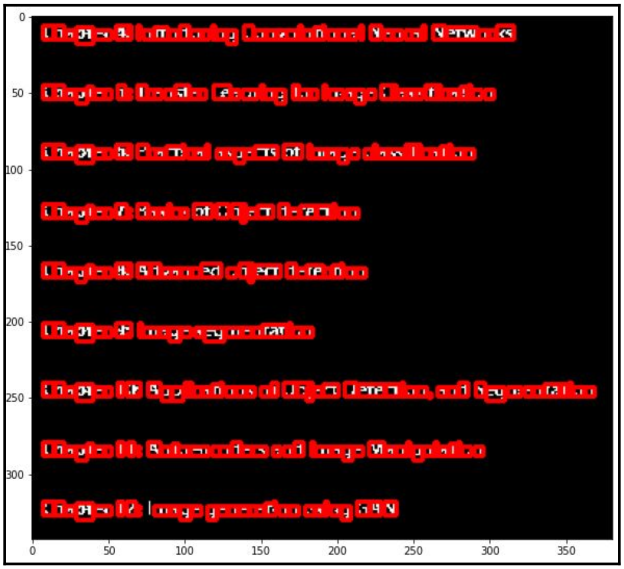
```
for cnt in contours:
    if cv2.contourArea(cnt)>0:
        [x,y,w,h] = cv2.boundingRect(cnt)
        if ((h>5) & (h<100)):
            thresh2[y:(y+h),x:(x+w)] = thresh1[y:(y+h), \
                                              x:(x+w),0].copy()
            cv2.rectangle(thresh1,(x,y),(x+w,y+h),(255,0,0),2)
```

In the preceding lines of code, we are fetching only those contours that have an area greater than 5 pixels and also fetching only those where the height of the bounding box is between 5 and 100 pixels (this way, we eliminate the boxes that are too small, which are likely to be noise, and large bounding boxes that might encompass the whole image).

10. Plot the resulting image:

```
fig = plt.figure()
fig.set_size_inches(20,20)
plt.imshow(thresh1)
```

The preceding code fetches the following output:

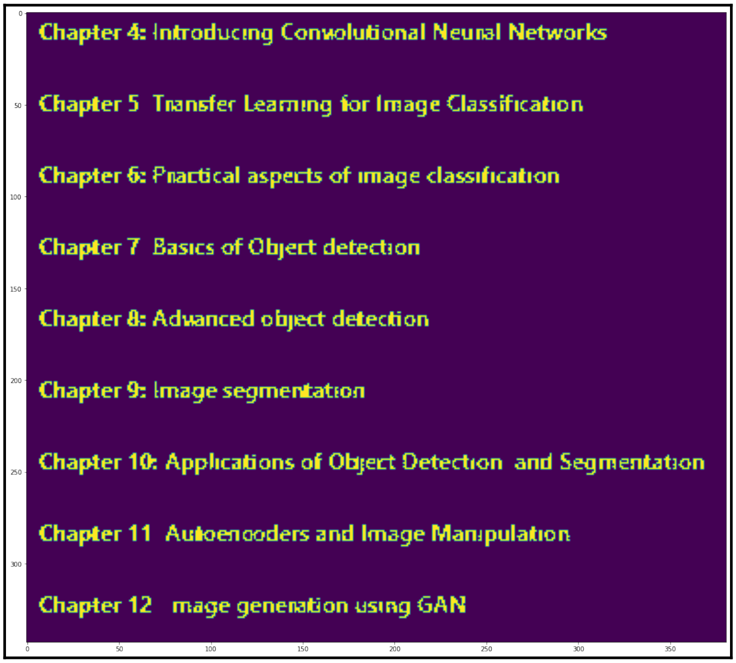


So far, we can draw bounding boxes around characters, but if we want to draw boxes around words, we need to combine pixels within the word into a single contiguous unit. Next, we will look at drawing bounding boxes around words, leveraging word dilation techniques.

11. Inspect the filled image, `thresh2`:

```
fig = plt.figure()
fig.set_size_inches(20,20)
plt.imshow(thresh2)
```

The resulting image would look as follows:



Now, the problem to be solved is how to connect the pixels of different characters into one so that a continuous collection of pixels constitutes a word.

We use a technique called dilation (using `cv2.dilate`), which bleeds white pixels into the surrounding pixels. The amount of bleeding is dictated by the kernel size. If the kernel size is, say, 5, then all the borders of the white regions move outward by 5 pixels. Refer to the following screenshot for an intuitive explanation:

12. Dilate with a kernel size of 1 row and 2 columns:

```
dilated = cv2.dilate(thresh2, np.ones((1,2),np.uint8), \
                        iterations=1)
```

Note that we are specifying a kernel size of 1 row and 2 columns (`np.ones((1,2),np.uint8)`) so that adjacent characters are very likely to have some intersection. This way, `cv2.findContours` can now encompass the characters that are very close to each other.

However, if the kernel size is bigger, the dilated words can have some intersection, resulting in the combined words being captured in one bounding box.

13. Fetch the contours of the dilated image:

```
contours,hierarchy = cv2.findContours(np.uint8(dilated), \
                                      cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)
```

14. Draw contours of the dilated image on the original image:

```
for cnt in contours:
    if cv2.contourArea(cnt)>5:
        [x,y,w,h] = cv2.boundingRect(cnt)
        if ((h>5) & (h<100)):
            cv2.rectangle(img1,(x,y),(x+w,y+h),(255,0,0),2)
```

15. Plot the original image with contours:

```
fig = plt.figure()
fig.set_size_inches(20,20)
plt.imshow(img1)
```

The preceding code results in the following output:

From this, you can see that we fetched a bounding box corresponding to each word.

The key aspect to learn is how we can identify that a collection of pixels forms a single connected unit, and if the collection of pixels did not form a unit, how to manipulate them using dilation. While dilation bleeds the black pixels, there is a similar function called `erode` that bleeds white pixels. We encourage you to perform erosion and understand how it works by yourself.

So far, we have learned about finding contours around characters (objects) in an image. In the next section, we will learn about identifying lines within an image.

Detecting lanes in an image of a road

Imagine a scenario where you have to detect the lanes within an image of a road. One way to solve this is by leveraging semantic segmentation techniques in deep learning. One of the traditional ways of solving this problem using OpenCV has been using edge and line detectors. In this section, we will learn about how edge detection followed by line detection can help in identifying lanes within an image of a road.

Here, we will have outlined a high-level understanding of the strategy:

1. Find the edges of various objects present in the image.
2. Identify the edges that follow a straight line and are also connected.
3. Extend the identified lines from one end of the image to the other end.

Let's code up our strategy:

> The following code is available as `detecting_lanes_in_the_image_of_a_road.ipynb` in the `Chapter18` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` Be sure to copy the URL from the notebook in GitHub to avoid any issue while reproducing the results

1. Download a sample image:

```
!wget
https://www.dropbox.com/s/0n5cs04sb2y98hx/road_image3.JPG
```

2. Import the packages and inspect the image:

```
!pip install torch_snippets
from torch_snippets import show, read, subplots, cv2, np
IMG = read('road_image3.JPG',1)
img = np.uint8(IMG.copy())
```

The imported image looks like so:

There's too much information in the image, and we are only interested in the straight lines. One quick way to get the edges in an image is using a Canny edge detector, which identifies something as an edge when there is a drastic change in the color. The color change technically depends on the gradient of pixels within the image. The more the difference of two pixels, the higher the likelihood that the pixels represent the edge of an object.

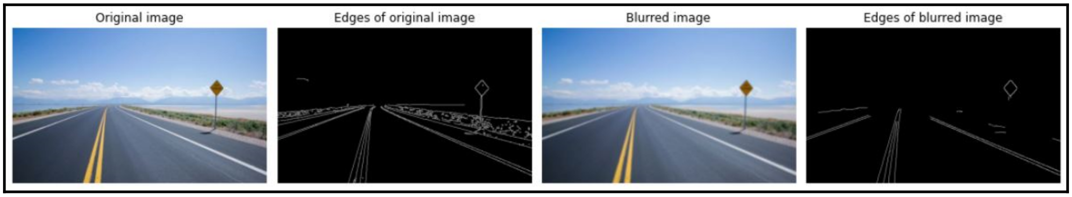
3. Fetch the edges corresponding to content in the image using the `cv2.Canny` edge detection technique:

```
blur_img = cv2.blur(img, (5,5))
edges = cv2.Canny(blur_img,150,255)
edges_org = cv2.Canny(img,150,255)
subplots([img,edges_org,blur_img,edges],nc=4, \
        titles=['Original image','Edges of original image', \
        'Blurred image','Edges of blurred image'],sz=15)
```

In the preceding code, we are first blurring the original image using `cv2.blur` in such a way that we look at a patch of 5 x 5, fetch the average of the pixel values in that patch, and replace the central element with the average of the pixel values surrounding every pixel.

When calculating edges using the `cv2.Canny` method, the values `150` and `255` represent the minimum and maximum possible gradient values corresponding to the edges. Note that a pixel is an edge if one side of the pixel has a certain pixel value and another side has a pixel value that is considerably different from the pixel on the other side.

The image and edges for the original and blurred images look as follows:



From the preceding, we can see that the edges are more logical when we perform blurring on the original image. Now that the edges are identified, we need to get only the straight ones from the image. This is done using the `HoughLines` technique.

4. Identify the lines that have a length of at least 100 pixels using the `cv2.HoughLines` method:

```
lines = cv2.HoughLines(edges,1,np.pi/180,100)
```

Note that the parameter value of `100` specifies that the length of the identified line should be at least 100 pixels.

The obtained lines, in this case, have a shape of 9 x 1 x 2; that is, there are nine lines in the image, each with its own distance from the bottom-left corner of the image and a corresponding angle (typically referred to as `[rho, theta]` in polar coordinates).

5. Plot the lines that are less horizontal:

```
lines = lines[:,0,:]
for rho,theta in lines:
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = a*rho
    y0 = b*rho
    x1 = int(x0 + 10000*(-b))
    y1 = int(y0 + 10000*(a))
    x2 = int(x0 - 10000*(-b))
    y2 = int(y0 - 10000*(a))
    if theta < 75*3.141/180 or theta > 105*3.141/180:
        cv2.line(blur_img,(x1,y1),(x2,y2),(255,0,0),1)

show(blur_img,sz=10, grid=True)
```

The preceding code generates the following output:

To summarize, we first filtered out all the possible noise from the image by performing blurring and edge detection. Only a few pixels remained as likely candidates for lanes. Next, using `HoughLines`, we further filtered out candidates that are not straight lines of at least 100 pixels. While the lanes on the road are detected reasonably well in this image, it is not guaranteed that the preceding logic works on every image of a road. As an exercise, try out the preceding process on a few different road images. Here is where you will appreciate the power of deep learning over lane detection using OpenCV, where the model learns to predict accurately on a wide variety of images (provided we train the model on a reasonably wide variety of images).

Detecting objects based on color

Green screen is a classic video editing technique where we can make someone look like they are standing in front of a completely different background. This is widely used in weather reports, where reporters point to backgrounds of moving clouds and maps. The trick in this technique is that the reporter never wears a certain color of clothing (say, green) and stands in front of a background that is only green. Then, identifying green pixels will identify what is the background and helps replace content at only those pixels.

In this section, we will learn about leveraging the `cv2.inRange` and `cv2.bitwise_and` methods to detect the green color in any given image.

The strategy that we will adopt is as follows:

1. Convert the image from RGB into HSV space.
2. Specify the upper and lower limits of HSV space that correspond to the color green.
3. Identify the pixels that have a green color – this will be the mask.
4. Perform a `bitwise_and` operation between the original image and the mask image.

The preceding strategy is implemented in code as follows:

> The following code is available as `Detecting_objects_based_on_color.ipynb` in the `Chapter18` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` Be sure to copy the URL from the notebook in GitHub to avoid any issue while reproducing the results

1. Fetch the image and install the required packages:

```
!wget https://www.dropbox.com/s/utrkdooh08y9mvm/uno_card.png
!pip install torch_snippets
from torch_snippets import *
import cv2, numpy as np
```

2. Read the image and convert it into **HSV** (**Hue-Saturation-Value**) space. Converting to HSV space from RGB will let us decouple the brightness out of the color so that we can easily extract the color information of every pixel:

```
img = read('uno_card.png', 1)
show(img)
hsv = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
```

Here is the image in RGB space:

3. Define the upper and lower thresholds of the green color in HSV space:

```
lower_green = np.array([45,100,100])
upper_green = np.array([80,255,255])
```

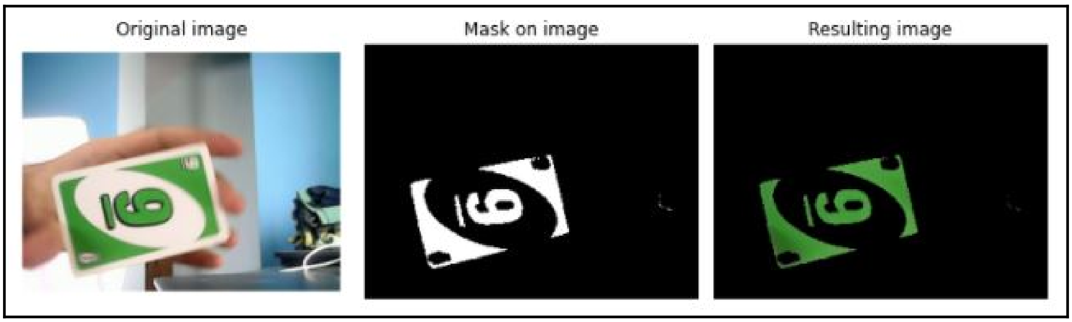
4. Generate the mask, which activates only the pixels that fall within the defined upper and lower thresholds. `cv2.inRange` is a comparison operation to check whether a pixel value is between the minimum and maximum but on an HSV scale:

```
mask = cv2.inRange(hsv, lower_green, upper_green)
```

5. Perform the `cv2.bitwise_and` operation between the original image and mask to fetch the resulting image:

```
res = cv2.bitwise_and(img, img, mask=mask)
subplots([img, mask, res], nc=3, figsize=(10,5), \
        titles=['Original image','Mask on image', \
                'Resulting image'])
```

The original image, the mask, and the resulting image are as follows:



From the preceding plot, we can see that the algorithm has ignored the rest of the content in the image and concentrated only on the color of interest. Using this, we can extend the logic to come up with a foreground mask that is exclusively *not* green using the `cv2.bitwise_not` operation and perform the green screen technique.

In summary, we can identify color spaces in an image, and if we want to project/overlay another image onto the identified green screen, we pick the pixels from the other image that correspond to the green pixels in the original one.

Next, we will learn about matching features from one image to another using keypoint detection techniques.

Building a panoramic view of images

In this section, we will learn about one of the techniques that helps in creating a panoramic view by combining multiple images.

Imagine a scenario where you are capturing the panorama of a place using your camera. Essentially, you are taking multiple shots, and in the backend, the algorithm is mapping the common elements present across the images (moving from the leftmost to the rightmost side) into a single image.

To perform the stitching of images, we will leverage the **ORB** (**Oriented FAST and Rotated BRIEF**) method available in `cv2`. Getting into the details of how these algorithms work is beyond the scope of this book – we encourage you to go through the documentation and the paper available at `https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_orb/py_orb.html`.

At a high level, the method identifies keypoints within a **query image** (`image1`) and then associates them with the keypoints identified in another **training image** (`image2`) if the keypoints match.

The strategy that we will adopt to perform image stitching is as follows:

1. Calculate the keypoints and extract them in both images.
2. Identify the common features in both images using the brute-force method.
3. Leverage the `cv2.findHomoGraphy` method to transform the training image to match the orientation of the query image.
4. Finally, we leverage the `cv2.warpperspective` method to fetch a view that looks like a standard view.

Now, we will implement the preceding strategy using the following code:

> The following code is available as `Building_a_panoramic_view_of_images.ipynb` in the `Chapter18` folder of the book's GitHub repository - `https://tinyurl.com/mcvp-packt` Be sure to copy the URL from the notebook in GitHub to avoid any issue while reproducing the results

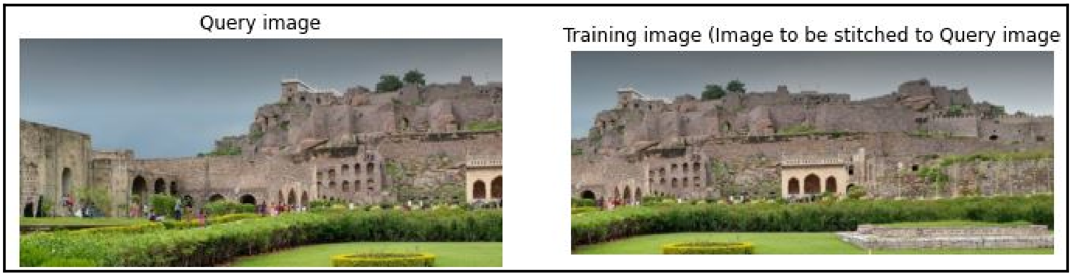
1. Fetch the images and import the relevant packages:

```
!pip install torch_snippets
from torch_snippets import *
!wget https://www.dropbox.com/s/mfg1codtc2rue84/g1.png
!wget https://www.dropbox.com/s/4yhui8s1xjndavm/g2.png
```

2. Load the query and train images and convert them into grayscale images:

```
queryImg = read('g1.png', 1)
queryImg_gray = read('g1.png')

trainImg = read('g2.png', 1)
trainImg_gray = read('g2.png')

subplots([trainImg, queryImg], nc=2, figsize=(10,5), \
        titles = ['Query image', \
    'Training image (Image to be stitched to Query image)'])
```

The query and train images look as follows:

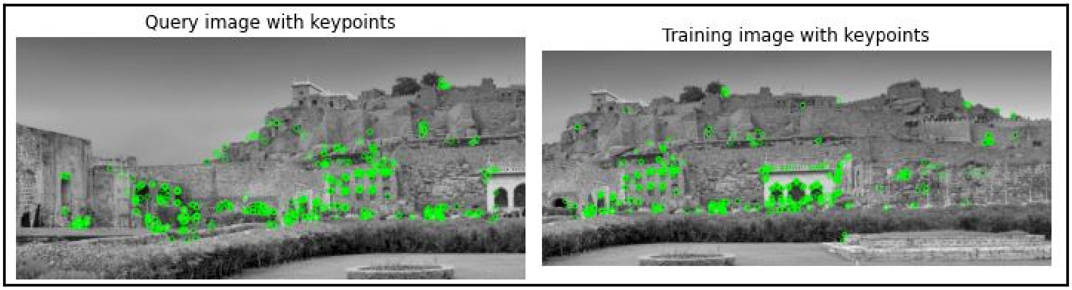


3. Extract the keypoints and features in both images using the ORB feature detector:

```
# Fetch the keypoints and features corresponding to the images
descriptor = cv2.ORB_create()
kpsA, featuresA = descriptor.detectAndCompute(trainImg_gray, \
```

```
                                                        None)
kpsB, featuresB = descriptor.detectAndCompute(queryImg_gray, \
                                                        None)
# Draw the keypoints obtained on images
img_kpsA = cv2.drawKeypoints(trainImg_gray,kpsA,None, \
                            color=(0,255,0))
img_kpsB = cv2.drawKeypoints(queryImg_gray,kpsB,None, \
                            color=(0,255,0))
subplots([img_kpsB, img_kpsA], nc=2, figsize=(10,5), \
            titles=['Query image with keypoints', \
                'Training image with keypoints'])
```

A plot of the extracted keypoints in both images is as follows:



ORB or any other feature detector works in two steps:

1. First, it identifies interesting keypoints in both the images. One of the standard keypoint detectors is Harris Corner Detector, which identifies intersections of lines to tell whether something is a sharp corner or not.
2. Second, all the pairs of keypoints from both images are compared with each other to see whether there is a high correlation around patches of images near the keypoints. If there's a high match, it must mean that both the keypoints are referring to the same location in the images.

For an in-depth understanding of ORB, refer to *ORB: An efficient alternative to SIFT or SURF* (`https://ieeexplore.ieee.org/document/6126544`).

4. Find the best matches in features of both images using the `cv2.BFMatcher` method:

```
bf = cv2.BFMatcher(cv2.NORM_HAMMING)
best_matches = bf.match(featuresA,featuresB)
matches = sorted(best_matches, key = lambda x:x.distance)
```

The output of the matches is a list of `DMatch` objects. The `DMatch` objects have the following attributes:

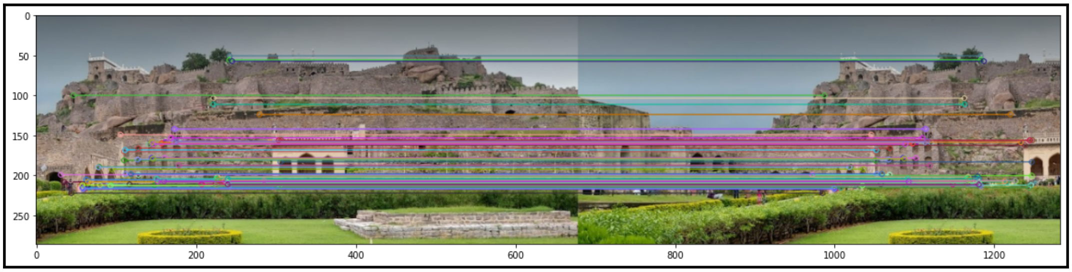
- `DMatch.distance`: The distance between descriptors. The lower, the better
- `DMatch.trainIdx`: Index of the descriptor in the train descriptors
- `DMatch.queryIdx`: Index of the descriptor in the query descriptors
- `DMatch.imgIdx`: Index of the train image

Note that we have sorted the matches between the two image features based on their distance.

5. Plot the matches using the following code:

```
img3 = cv2.drawMatches(trainImg,kpsA,queryImg,kpsB, \
                       matches[:100],None, \
        flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
show(img3)
```

The preceding code results in the following output:



Now, we need to find the right set of translation, rotation, and scaling to superimpose the second image on top of the first. This set of transformations is obtained as a homography matrix.

6. Fetch the homography corresponding to the two images:

```
kpsA = np.float32([kp.pt for kp in kpsA])
kpsB = np.float32([kp.pt for kp in kpsB])
ptsA = np.float32([kpsA[m.queryIdx] for m in matches])
ptsB = np.float32([kpsB[m.trainIdx] for m in matches])

(H, status) = cv2.findHomography(ptsA, ptsB, cv2.RANSAC,4)
```

Note that we are considering only those points that are identified as a match between the two images. Furthermore, by performing homography, we have come up with a matrix, `H`, that is able to transform `ptsA` with its associated points in `ptsB` using the following equation:

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = H \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

7. Perform image stitching:

   Given the `H` matrix, you can do the actual translation, rotation, and scaling using the `cv2.warpPerspective` function. After doing this, on `trainImg`, we will superimpose `queryImg` on it and we will have our panoramic image!

```
width = trainImg.shape[1] + queryImg.shape[1]
height = trainImg.shape[0] + queryImg.shape[0]

result = cv2.warpPerspective(trainImg, H, (width, height))
result[0:queryImg.shape[0], 0:queryImg.shape[1]] = queryImg

_x = np.nonzero(result.sum(0).sum(-1) == 0)[0][0]
_y = np.nonzero(result.sum(1).sum(-1) == 0)[0][0]

show(result[:_y,:_x])
```

   The preceding results in the following output:

From the preceding, we can see that we have successfully combined the two images using the keypoints that were detected to have a match between the two images. The key insight in this section should be that there are several keypoint-matching techniques that identify whether two local features in two different images are the same or not.

Once the common keypoints are identified, we leveraged homography to identify the transformations to perform. Finally, we perform the transformation that will align the two images by leveraging the `cv2.warpperspective` technique and stitch the two images together. In addition to image stitching, this pipeline of techniques (keypoint identification, identifying the matching keypoint between the two images, identifying the transformation to be performed, and performing the transformation) is immensely useful in applications such as image registration, where one image needs to be superimposed on top of another.

Next, we will learn about leveraging pre-trained cascade classifiers when identifying the location of the number plate on a car.

Detecting the number plate of a car

Imagine a scenario where we ask you to identify the location of a number plate in the image of a car. One way we have learned how to do this in the chapters on object detection is to come up with anchor box-based techniques to identify the location of the number plate. This would require us to train the model on a few hundred images before we leverage the model.

However, there is a cascade classifier that is readily available as a pre-trained file that we can use to identify the location of the number plate in an image of a car. A classifier is a **cascade** classifier if it consists of several simpler classifiers (*stages*) that are applied subsequently to a region of interest until at some stage, the candidate region is rejected or all the stages are passed. These are analogous to convolution kernels that we have learned how to use so far. Instead of having a deep neural network that learns kernels from other kernels, this is a list of kernels that have been identified to give a good classification score when all of their classifications are voted for.

For example, a face cascade can have up to 6,000 kernels that deal with some part of the face. A couple of these kernels might look like so:

These cascades are also referred to as Haar Cascade.

With this high-level understanding, let's chalk out the strategy that we will adopt in leveraging pre-trained cascade classifiers to identify the location of the number plate in an image of a car:

1. Import the relevant cascade.
2. Convert the image into a grayscale image.
3. Specify the minimum and maximum scale of the object of interest present within our image.
4. Fetch the region proposals coming from the cascade classifier.
5. Draw bounding boxes around the region proposals.

Let's implement the preceding strategy in code:

The following code is available as `Detecting_the_number_plate_of_a_car.ipynb` in the `Chapter18` folder of this book's GitHub repository - `https://tinyurl.com/mcvp-packt` Be sure to copy the URL from the notebook in GitHub to avoid any issue while reproducing the results

1. Fetch the number plate recognition cascade:

```
!wget
https://raw.githubusercontent.com/zeusees/HyperLPR/master/mode
l/cascade.xml
```

2. Fetch the image:

```
!wget https://www.dropbox.com/s/4hbem2kxzqcwo0y/car1.jpg
```

3. Load the image and cascade classifier:

```
!pip install torch_snippets
from torch_snippets import *
plate_cascade = cv2.CascadeClassifier('cascade.xml')
image = read("car1.jpg", 1)
```

4. Convert the image to grayscale and plot it:

```
image_gray = cv2.cvtColor(image,cv2.COLOR_RGB2GRAY)
```

5. Leverage the cascade to detect the number plate of multiple scales:

```
plates = plate_cascade.detectMultiScale(image_gray, 1.08, \
                                        2, minSize=(40, 40), \
                                        maxSize=(1000, 100))
```

`plate_cascade.detectMultiScale` will return all possible rectangular regions that have a high convolve match with the cascade kernels – which helps in identifying the location of the number plate within an image. Furthermore, we are specifying the minimum and maximum size of the width and height.

6. Loop through the plate region proposals (plates) and fetch the region that is a little bigger than the region proposal:

```
image2 = image.astype('uint8')
for (x, y, w, h) in plates:
    print(x,y,w,h)
    x -= w * 0.14
    w += w * 0.75
    y -= h * 0.15
    h += h * 0.3
    cv2.rectangle(image2, (int(x), int(y)), \
                 (int(x + w), int(y + h)), (0, 255, 0), 10)
show(image2, grid=True)
```

The preceding code generates the following output:

From the preceding screenshot, we can see that the pre-trained cascade classifier can identify the location of the number plate accurately. Similar to the road lane detection exercise, even in the case of number plate detection, we might encounter a scenario where our strategy is not working on a different set of images. We encourage you to try out the preceding steps on different custom images.

Summary

In this chapter, we learned about leveraging some of the OpenCV-based techniques to identify contours, edges, and lines, and track colored objects. While we discussed a few use cases in this chapter, these techniques have a much broader application across the various use cases. Then, we learned about identifying similarities between two images using the keypoint and feature extraction techniques when stitching two images related to each other. Finally, we learned about cascade classifiers and leveraging the pre-trained ones to arrive at an optimal solution with little development effort, and also generating predictions in real time.

Broadly, through this chapter, we wanted to show that not all problems need neural networks and, especially in constrained environments, we can use a vast library of historical knowledge and techniques to quickly solve those problems. Where it is not possible to solve with OpenCV, we have already delved deep into neural networks.

Images are fascinating. Storing them has been one of humanity's earliest endeavors and is one of the most powerful ways to capture content. The ease of capturing images in the 21st century has opened up multitudes of problems that can be solved with or without human intervention. We have covered some of the most common as well as modern tasks using PyTorch – image classification, object detection, image segmentation, image embedding, image generation, manipulating the generated image, training with very few data points, combining computer vision with NLP techniques, and reinforcement learning. We have covered the working details of various algorithms from scratch. We have also learned how to formulate a problem, capture the data, create networks, and infer from the trained models, and learned how to train and validate them. We understood how to pick up code bases/pre-trained models and customize them for our tasks, and finally, we learned about deploying our model.

We hope you have picked up the skills to handle images like it's second nature and solve your own tasks of interest.

Most importantly, we hope this has been a joyful journey for you and that you have enjoyed reading the book as much as we have enjoyed writing it!

Appendix

Chapter 1 - Artificial Neural Network Fundamentals

1. What are the various layers in a neural network?
   Input, Hidden, and Output Layers
2. What is the output of a feed-forward propagation?
   Predictions that help in calculating loss value
3. How is the loss function of a continuous dependent variable different from that of a binary dependent variable and also of a categorical dependent variable?
   MSE is the generally used loss function for a continuous dependent variable and binary cross-entropy for a binary dependent variable. Categorical cross-entropy is used for categorical dependent variables.
4. What is stochastic gradient descent?
   It is a process of reducing loss, by adjusting weights in the direction of decreasing gradient
5. What does a backpropagation exercise do?
   It computes gradients of all weights with respect to loss using the chain rule
6. How does the weight update of all the weights across layers happen during back-propagation?
   It happens using the formula dW = W – alpha*(dW/dL)
7. What all functions of a neural network happen within each epoch of training a neural network?
   For each batch in an epoch, perform forward-prop -> back prop -> update weights -> repeat with next batch until the end of all the epochs
8. Why is training a network on a GPU faster when compared to training it on a CPU?
   More matrix operations can be performed in parallel on GPU hardware

9. How does the learning rate impact training a neural network?
   Too high a learning rate will explode the weights, and too small a learning rate will not change the weights at all
10. What is the typical value of the learning rate parameter?
   1e-2 to 1e-5

Chapter 2 - PyTorch Fundamentals

1. Why should we convert integer inputs into float values during training?
   `nn.Linear` (and almost all torch layers) only accepts floats as inputs
2. What are the various methods to reshape a tensor object?
   reshape, view
3. Why is computation faster with tensor objects over NumPy arrays?
   Capability to run on GPUs in parallel is only available on tensor objects
4. What constitutes the init magic function in a neural network class?
   Calling `super().__init__()` and specifying the neural network layers
5. Why do we perform zero gradients before performing back-propagation?
   To ensure gradients from previous calculations are flushed out
6. What magic functions constitute the dataset class?
   `__len__` and `__getitem__`
7. How do we make predictions on new data points?
   By calling the model on the tensor as if it is a function – model(x)
8. How do we fetch the intermediate layer values of a neural network?
   By creating a custom method
9. How does the `Sequential` method help in simplifying defining the architecture of a neural network?
   We can avoid creating `__init__` and the `forward` method by connecting a sequence of layers

Chapter 3 - Building a Deep Neural Network with PyTorch

1. What is the issue if the input values are not scaled in the input dataset?
   It takes longer to adjust weights to optimal value because input values vary so widely when they are unscaled
2. What could be the issue if the background has a white pixel color while the content has a black pixel color when training a neural network?
   The neural network has to learn to ignore a majority of the not so useful content that is white in color
3. What is the impact of batch size on the model's training time, accuracy over a given number of epochs?
   The larger the batch size more is the time taken to converge and more iterations required to attain a high accuracy
4. What is the impact of the input value range on weight distribution at the end of the training?
   If the input value is not scaled to a certain range, certain weights can aid in over-fitting
5. How does batch normalization help in improving accuracy?
   Just like how it is important that we scale the inputs for better convergence of the ANN, batch normalization scales the activations for better convergence of its next layer
6. How do we know if a model has over-fit on training data?
   When validation loss is constant or keeps increasing with more epochs while training loss keeps decreasing over increasing epochs
7. How does regularization help in avoiding over-fitting?
   Regularization techniques help the model to train in a constrained environment thereby forcing ANN to adjust its weights in a less biased fashion
8. How do L1 and L2 regularization differ from each other?
   L1 = sum of the absolute value of weights, L2 = sum of the square of weights are added to the loss value in addition to the typical loss
9. How does Dropout help in reducing over-fitting?
   By dropping some connections in ANN we are forcing networks to learn from fewer data. This forces the model to generalize.

Chapter 4 - Introducing Convolutional Neural Networks

1. Why is the prediction on a translated image low when using traditional neural networks?
   All images were centered in the original dataset, so the ANN learned the task for only centered images.
2. How is Convolution done?
   Convolution is a multiplication between two matrices.
3. How are optimal weight values in a filter identified?
   Through backpropagation.
4. How does the combination of convolution and pooling help in addressing the issue of image translation?
   While convolution gives important image features, pooling takes the most prominent features in a patch of the image. This makes pooling a robust operation over the vicinity, i.e., even if something is translated by a few pixels, pooling will still return the expected output.
5. What do the filters in layers closer to the input layer learn?
   Low-level features like edges.
6. What functionality does pooling do that helps in building a model?
   It reduces input size by reducing feature map size and makes model translation invariant.
7. Why can we not take the input image, flatten just like we did on the FashionMNIST dataset, and train a model for real-world images?
   If the image size is even modestly large, the number of parameters connecting two layers will be in millions.
8. How does data augmentation help in improving image translation?
   Data augmentation creates copies of images that are translated by a few pixels. Thus the model is forced to learn the right classes even if the object in the image is off-center.
9. In what scenario do we leverage `collate_fn` for dataloaders?
   When we need to perform batch level transformations, which are difficult/slow to perform in `__getitem__`.
10. What is the impact of varying the number of training data points on classification accuracy on the validation dataset?
    In general, the larger the dataset size, the better will the model accuracy.

Chapter 5 - Transfer Learning for Image Classification

1. What are VGG and ResNet pre-trained architectures trained on?
   The images in the Imagenet dataset.
2. Why does VGG11 have an inferior accuracy to VGG16?
   VGG11 has fewer layers when compared to VGG16.
3. What does the number 11 in VGG11 represent?
   11 layers.
4. What is residual in the residual network?
   The layer returns input in addition to the layer's transformation.
5. What is the advantage of a residual network?
   It helps in avoiding vanishing gradients and also helps in increasing model depth.
6. What are the various popular pre-trained models?
   VGG, ResNet, Inception, AlexNet.
7. During transfer learning, why should images be normalized with the same mean and standard deviation as those which were used during training of a pre-trained model?
   Models are trained such that they expect input images to be normalized with a specific mean and standard deviation.
8. Why do we freeze certain parameters in a model?
   We freeze so that the parameters will not be updated during backpropagation. They are not updated as they are already well trained.
9. How do we know the various modules present in a pre-trained model?
   `print(model)`
10. How do we train a model which predicts categorical and numeric values together?
    By having multiple prediction heads and training with a separate loss for each head.
11. Why might age and gender prediction code not work always for an image of your own interest if we execute the same code as we wrote in the age and gender estimation section?
    An Image that does not have similar distribution as training data can give unexpected results.
12. How can we further improve the accuracy of the facial key-points recognition model that we wrote in the facial keypoints prediction section?
    We can add color and geometric augmentations to the training process.

Chapter 6 - Practical Aspects of Image Classification

1. How are class activation maps obtained?
   Refer to the 8 steps provided in the Generating CAMs section

2. How do batch normalization and data augmentation help when training a model?
   They help reduce over-fitting

3. What are the common reasons why a CNN model overfits?
   No batch normalization, data augmentation, dropout

4. What are the various scenarios where the CNN model works with training and validation data at the data scientists' end but not in the real world?
   Real-world data can have a different distribution from the data used to train and validate the model. Additionally, the model might have over-fitted on training data

5. What are the various scenarios where we leverage OpenCV packages?
   While working in constrained environments, and also when speed to infer is more important

Chapter 7 - Basics of Object Detection

1. How does the region proposal technique generate proposals?
   It identifies regions that are similar in color, texture, size, and shape.
2. How is IoU calculated if there are multiple objects in an image?
   IoU is calculated for each object with the ground truth, using Intersection Over Union metric
3. Why does R-CNN take a long time to generate predictions?
   Because we create as many forward propagations as there are proposals
4. Why is Fast R-CNN faster when compared to R-CNN?
   For all proposals, extracting the feature map from the VGG backbone is common. This reduces almost 90% of the computations as compared to Fast RCNN

5. How does RoI Pooling work?
   All the `selectivesearch` crops are passed through adaptive pooling kernel so that the final output is of the same size
6. What is the impact of not having multiple layers, post obtaining feature map, when predicting the bounding box corrections?
   You might not notice that the model did not learn to predict the bounding boxes accurately
7. Why do we have to assign a higher weightage to regression loss when calculating overall loss?
   Classification loss is cross-entropy which is generally of the order log(n) resulting in outputs that can have a high range. However, bounding box regression losses are between 0 and 1. Hence regression losses have to be scaled up.
8. How does Non-max suppression work?
   By combining boxes of the same classes and with high IoUs, we eliminate redundant bounding box predictions.

Chapter 8 - Advanced Object Detection

1. Why is Faster R-CNN faster when compared to Fast R-CNN?
   We do not need to feed a lot of unnecessary proposals every time using the `selectivesearch` technique. Instead, Faster R-CNN automatically finds them using the region proposal network.
2. How are YOLO and SSD faster when compared to Faster R-CNN?
   We don't need to rely on a new proposal network. The network directly finds the proposals in a single go.
3. What makes YOLO and SSD single shot detector algorithms?
   Networks predict all the proposals and predictions in one shot
4. What is the difference between the objectness score and class score?
   Objectness identifies if an object exists or not. But class score predicts what is the class for an anchor box whose objectness is non zero

Chapter 9 - Image Segmentation

1. How does up-scaling help in U-Net architecture?
   Upscaling helps the feature map to increase in size so that the final output is the same size as the input size.
2. Why do we need to have a fully convolutional network in U-Net?
   Because the outputs are also images, and it is difficult to predict an image shaped tensor using the Linear layer.
3. How does RoI Align improve over RoI pooling in Mask R-CNN?
   RoI Align takes offsets of predicted proposals to fine-align the feature map.
4. What is the major difference between U-Net and Mask R-CNN for segmentation?
   U-Net is fully convolutional and with a single end2end network, whereas Mask R-CNN uses mini networks such as Backbone, RPN, etc to do different tasks. Mask R-CNN is capable of identifying and separating several objects of the same type, but U-Net can only identify (but not separate them into individual instances).
5. What is instance segmentation?
   If there are different objects of the same class in the same image then each such object is called an instance. Applying image segmentation to predict, at a pixel level, all the instances separately is called instance segmentation.

Chapter 11 - Autoencoders and Image Manipulation

1. What is an encoder in autoencoder?
   A smaller neural network that converts an image into a vector representation.
2. What loss function does autoencoder optimize for?
   Pixel level mean square error, directly comparing prediction with input.
3. How do autoencoders help in grouping similar images?
   Similar images will return similar encodings, which are easier to cluster.
4. When is the Convolutional autoencoder useful?
   When the inputs are images.

5. Why do we get non-intuitive images if we randomly sample from vector space of embeddings obtained from vanilla/convolutional autoencoder?
   The range of values in encodings is unconstrained, so proper outputs are highly dependent on the right range of values. Random sampling, in general, assumes a 0 mean and 1 standard deviation.
6. What are the loss functions that the Variational autoencoder optimizes for?
   Pixel level MSE and KL-Divergence of the distribution of mean and standard deviation from the encoder.
7. How does the Variational autoencoder overcome the limitation of vanilla/convolutional auto-encoders to generate new images?
   By constraining predicted encodings to have a normal distribution, all encodings fall in the region of mean-0 and standard deviation 1, which is easy to sample from.
8. During an adversarial attack, why do we modify the input image pixels and not the weight values?
   We do not have control over the neural network in adversarial attacks.
9. In a neural style transfer what are the losses that we optimize for?
   Perceptual (VGG) loss of generated image with the original image, and the style-loss coming from the gram matrices of generated and style images.
10. Why do we consider the activation of different layers and not the original image when calculating style and content loss?
    Using more intermediate layers ensures, the generated image is preserving finer details about the image. Also, using more losses makes the gradient ascent more stable.
11. Why do we consider gram matrix loss and not the difference between images when calculating style loss?
    Gram matrix gives an indication of the style of the image, i.e., how the textures shapes, and colors are arranged and will ignore the actual content. That is why it is more convenient for style loss.
12. Why do we warp images while building a model to generate deep fakes?
    Warping images helps act as a regularize. Further, it helps in generating as many images as required.

Chapter 12 - Image Generation Using GANs

1. What happens if the learning rate of generator and discriminator models is high?
   Empirically, it is observed that the model stability is lower.
2. In a scenario where the generator and discriminator are very well trained, what is the probability of a given image being real?
   0.5.
3. Why do we use convtranspose2d in generating images?
   We cannot upscale/ generate images using a linear layer.
4. Why do we have embeddings with high embedding size than the number of classes in Conditional GANs?
   Using more parameters gives the model more degrees of freedom to learn the important features of each class.
5. How can we generate images of men that have a beard?
   By using a conditional GAN. Just like we had male and female images, we can have bearded males and other such classes while training model.
6. Why do we have Tanh activation at the last layer in the generator and not ReLU or Sigmoid?
   The pixel range of normalized images is [-1,1] and hence we use Tanh
7. Why did we get realistic images even though we did not de-normalize the generated data?
   Even though pixel values are not between [0,255], the relative values are sufficient for the `make_grid` utility to de-normalize input
8. What happens if we do not crop faces corresponding to images before training the GAN?
   If there is too much background, the GAN can get wrong signals as to what is a face and what is not, so it might focus on generating more realistic backgrounds
9. Why do the weights of the discriminator not get updated when the training generator does (as `generator_train_step` function involves the discriminator network)?
   It is a step by step process. When updating the generator we assume the discriminator is able to do its best.
10. Why do we fetch loss on real and fake images while training discriminator but only the loss on fake images while training generator?
    Because whatever generator creates are only fake images.

Chapter 13 - Advanced GANs to Manipulate Images

1. Why do we need a Pix2Pix GAN where a supervised learning algorithm like U-Net could have worked to generate images from contours?
   U-net only uses pixel-level loss during training. We needed pix2pix since there is no loss for realism when a U-net generates images.
2. Why do we need to optimize for 3 different loss functions in CycleGAN?
   Answer provided in the 7 points in CycleGAN section.
3. How do the tricks leverage in ProgressiveGAN help in building a StyleGAN?
   ProgressiveGAN helps the network to learn a few upsampling layers at a time so that when the image has to be increased in size, the networks responsible for generating current size images are optimal.
4. How do we identify latent vectors corresponding to a given custom image?
   By adjusting the randomly generated noise in such a way that the MSE loss between the generated image and the image of interest is as minimal as possible.

Chapter 14 - Training with Minimal Data Points

1. How are pre-trained word vectors obtained?
   From an existing database such as GLOVE or word2vec
2. How do we map from an image feature embedding to word embedding in Zero-shot learning?
   By creating a suitable neural network that returns a vector of the same shape as word-embedding and training with mse-loss (comparing prediction with actual word-embedding)
3. Why is the Siamese network called so?
   Because we always produce and compare two outputs with each other, for identicalness. Siamese stands for twins.
4. How does the Siamese network come up with the similarity between the two images?
   The loss function forces the network to predict that the outputs have a smaller distance if the images are similar.

Chapter 15 - Combining Computer Vision and NLP Techniques

1. Why are CNN and RNN combined in image captioning?
   CNN is needed for capturing image features, whereas, RNN is needed for creating the language output.
2. Why are start and end tokens provided in image captioning but not in handwritten transcription?
   CTC loss does not need such tokens, and moreover, in OCR, we generate tokens in all time-steps in one shot.
3. Why is the CTC loss function leveraged in handwriting transcription?
   We cannot delineate timesteps in the image. CTC takes care of aligning key image features with timesteps.
4. How do transformers help in object detection?
   By treating anchor boxes as embedding inputs for transformer decoders DETR learns dynamic anchor boxes thereby helping object detection.

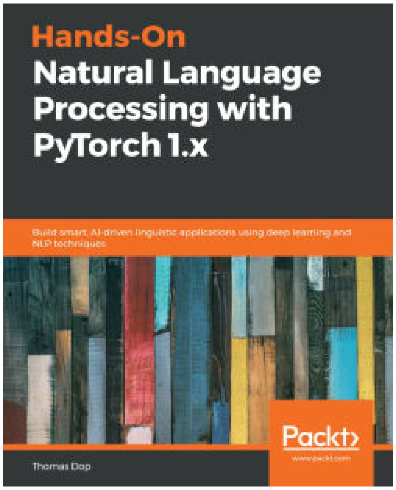
Chapter 16 - Combining Computer Vision and Reinforcement Learning

1. How is the value calculated for a given state?
   By computing the expected reward at that state
2. How is the Q-table populated?
   By computing expected reward for all states
3. Why do we have a discount factor in state action value calculation?
   Due to uncertainty, we are unsure of how the future might work. Hence we reduce future rewards' weightage which is done by the way of discounting
4. What is the need for an exploration-exploitation strategy?
   Only exploitation will make the model stagnant and predictable and hence model should be able to explore and find unseen steps that can be even more rewarding than what the model already has already learned.

5. What is the need for Deep Q-Learning?
   We let the neural network learn the likely reward system without the need for costly algorithms that may take too much time or demand visibility of the entire environment.
6. How is the value of a given state action combination calculated using Deep Q-Learning?
   It is simply the output of the neural network. The input is the state and the network predicts one expected reward for every action in the given state.
7. What are the possible actions in CARLA environment?
   accelerate – [-1,0,1] (brake, no-acceleration, accelerate)
   steer – [-1,0,1] (left, no-steer, right)

Other Books You May Enjoy

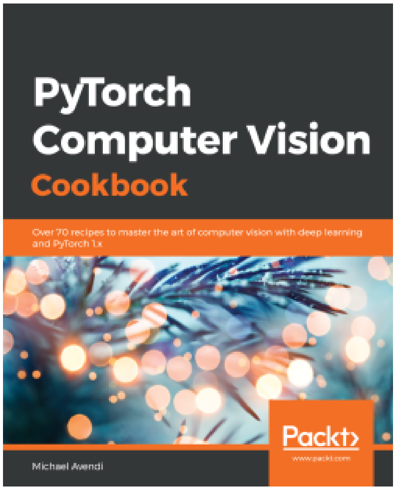
If you enjoyed this book, you may be interested in these other books by Packt:



Hands-On Natural Language Processing with PyTorch 1.x
Thomas Dop

ISBN: 978-1-78980-274-0

- Use NLP techniques for understanding, processing, and generating text
- Understand PyTorch, its applications, and how it can be used to build deep linguistic models
- Explore the wide variety of deep learning architectures for NLP
- Develop the skills you need to process and represent both structured and unstructured NLP data
- Become well-versed with state-of-the-art technologies and exciting new developments in the NLP domain
- Create chatbots using attention-based neural networks



PyTorch Computer Vision Cookbook
Michael Avendi

ISBN: 978-1-83864-483-3

- Develop, train and deploy deep learning algorithms using PyTorch 1.x
- Understand how to fine-tune and change hyperparameters to train deep learning algorithms
- Perform various CV tasks such as classification, detection, and segmentation
- Implement a neural style transfer network based on CNNs and pre-trained models
- Generate new images and implement adversarial attacks using GANs
- Implement video classification models based on RNN, LSTM, and 3D-CNN
- Discover best practices for training and deploying deep learning algorithms for CV applications

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

D

M

N

O

P

Q

R

S

T

U

V

Y

Z