

Deep Learning with PyTorch 1.x

Second Edition

Implement deep learning techniques and neural network architecture variants using Python



Packt

www.packt.com

Laura Mitchell, Sri. Yogesh K.
and Vishnu Subramanian

**Deep Learning with
PyTorch 1.x
*Second Edition***

Implement deep learning techniques and neural network architecture variants using Python

Laura Mitchell
Sri. Yogesh K.
Vishnu Subramanian



BIRMINGHAM - MUMBAI

Deep Learning with PyTorch 1.x

Second Edition

Copyright © 2019 Packt Publishing

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

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

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

Commissioning Editor: Sunith Shetty

Acquisition Editor: Devika Battike

Content Development Editor: Athikho Sapuni Rishana

Senior Editor: Sofi Rogers

Technical Editor: Joseph Sunil

Copy Editor: Safis Editing

Project Coordinator: Aishwarya Mohan

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Production Designer: Jyoti Chauhan

First published: February 2018

Second edition: November 2019

Production reference: 1291119

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-83855-300-5

www.packt.com



[Packt.com](https://www.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.

About the authors

Laura Mitchell graduated with a degree in mathematics from the University of Edinburgh. With 15 years of experience in the tech and data science space, Laura is the lead data scientist at MagicLab whose brands have connected the lives of over 500 million people through dating, social and business. Laura has hands-on experience in the delivery of projects surrounding natural language processing, image classification and recommender systems, from initial conception to production. She has a passion for learning new technologies and keeping herself up to date with industry trends.

Sri. Yogesh K. is an experienced data scientist with a history of working in higher education. He is skilled in Python, Apache Spark, deep learning, Hadoop, and machine learning. He is a strong engineering professional with a Certificate of Engineering Excellence from the International School of Engineering (INSOFE) and is focused on big data analytics. Sri has trained over 500 working professionals in data science and deep learning from companies including Flipkart, Honeywell, GE, and Rakuten. Additionally, he has worked on various projects that involved deep learning and PyTorch.

Vishnu Subramanian has experience in leading, architecting, and implementing several big data analytical projects using artificial intelligence, machine learning, and deep learning. He specializes in machine learning, deep learning, distributed machine learning, and visualization. He has experience in retail, finance, and travel domains. Also, he is good at understanding and coordinating between businesses, AI, and engineering teams.

About the reviewers

Mingfei Ma is a senior deep learning software engineer from Intel Asia-Pacific Research & Development Ltd and he has plenty of experience in high-performance computation. Mingfei contributed extensively to the CPU performance optimization of PyTorch and its predecessor, Torch. He also has expertise in computer graphics, heterogeneous computing, microarchitecture detection, high-performance computation libraries, and more.

Ajit Pratap Kundan is at the forefront of innovative technologies in the world of IT. He's worked with HPE, VMware, Novell, Redington, and PCS to help their customers in transforming their data centers through software-defined services. Ajit is an innovative pre-sales tech enthusiast with over 19 years of experience in technologies such as Lotus, SUSE Linux, Platespin, and all VMware solutions. Ajit is a valued author on cloud technologies and has authored two books, *VMware Cross-Cloud Architecture* and *Intelligent Automation with VMware*, published by Packt.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [author](https://s.packtpub.com) s.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

Title Page
Copyright and Credits
Deep Learning with PyTorch 1.x Second Edition
About Packt
Why subscribe?
Contributors
About the authors
About the reviewers
Packt is searching for authors like you
Preface
Who this book is for
What this book covers
To get the most out of this book
Download the example code files
Download the color images
Conventions used
Get in touch
Reviews
1. Section 1: Building Blocks of Deep Learning with PyTorch 1.x
1. Getting Started with Deep Learning Using PyTorch
Exploring artificial intelligence
The history of AI
Machine learning in the real world
So, why DL?
Applications of deep learning
Automatic translation of text from images
Object detection in self-driving cars
Deep learning frameworks
Why PyTorch?
What's new in PyTorch v1.x?
CPU versus GPU
What is CUDA?
Which GPUs should we use?

What should you do if you don't have a GPU?

Setting up PyTorch v1.x

Installing PyTorch

Summary

2. Building Blocks of Neural Networks

What is a neural network?

 Understanding the structure of neural networks

Building a neural network in PyTorch

PyTorch sequential neural network

Building a PyTorch neural network using nn.Module

Understanding PyTorch Tensors

Understanding Tensor shapes and reshaping Tensors

Understanding tensor operations

Understanding Tensor types in PyTorch

Importing our dataset as a PyTorch Tensor

Training neural networks in PyTorch

Summary

2. Section 2: Going Advanced with Deep Learning

3. Diving Deep into Neural Networks

Diving into the building blocks of neural networks

Layers – the fundamental blocks of neural networks

Non-linear activations

Sigmoid

Tanh

ReLU

Leaky ReLU

PyTorch non-linear activations

The PyTorch way of building deep learning algorithms

Model architecture for different machine learning problems

Loss functions

Optimizing network architecture

Image classification using deep learning

Loading data into PyTorch tensors

Loading PyTorch tensors as batches

Building the network architecture

Training the model

Summary

4. Deep Learning for Computer Vision

- Introduction to neural networks
- MNIST – getting data
- Building a CNN model from scratch
 - Conv2d
 - Pooling
 - Nonlinear activation – ReLU
 - View
 - Linear layer
 - Training the model
 - Classifying dogs and cats – CNN from scratch
 - Classifying dogs and cats using transfer learning
- Creating and exploring a VGG16 model
 - Freezing the layers
 - Fine-tuning VGG16
 - Training the VGG16 model
- Calculating pre-convoluted features
- Understanding what a CNN model learns
 - Visualizing outputs from intermediate layers
 - Visualizing the weights of the CNN layer
- Summary

5. Natural Language Processing with Sequence Data

- Working with text data
 - Tokenization
 - Converting text into characters
 - Converting text into words
 - N-gram representation
 - Vectorization
 - One-hot encoding
 - Word embedding
- Training word embedding by building a sentiment classifier
 - Downloading IMDb data and performing text tokenization
 - Tokenizing with torchtext.data
 - Tokenizing with torchtext.datasets
 - Building vocabulary
 - Generating batches of vectors
 - Creating a network model with embedding

- Training the model
- Using pretrained word embeddings
 - Downloading the embeddings
 - Loading the embeddings in the model
 - Freezing the embedding layer weights
- Recursive neural networks
 - Understanding how RNN works with an example
- Solving text classification problem using LSTM
 - Long-term dependency
 - LSTM networks
 - Preparing the data
 - Creating batches
 - Creating the network
 - Training the model
- Convolutional network on sequence data
 - Understanding one-dimensional convolution for sequence data
 - Creating the network
 - Training the model
- Language modeling
 - Pretrained models
 - Embeddings from language models
 - Bidirectional Encoder Representations from Transformers
 - Generative Pretrained Transformer 2
 - PyTorch implementations
 - GPT-2 playground
- Summary

3. Section 3: Understanding Modern Architectures in Deep Learning

6. Implementing Autoencoders

- Applications of autoencoders
 - Bottleneck and loss functions
 - Coded example – standard autoencoder
- Convolutional autoencoders
 - Coded example – convolutional autoencoder
- Denoising autoencoders
- Variational autoencoders
 - Training VAEs
 - Coded example – VAE

- Restricted Boltzmann machines
 - Training RBMs
 - Theoretical example – RBM recommender system
 - Coded example – RBM recommender system
- DBN architecture
 - Fine-tuning
- Summary
- Further reading

7. Working with Generative Adversarial Networks

- Neural style transfer
 - Loading the data
 - Creating the VGG model
 - Content loss
 - Style loss
 - Extracting the losses
 - Creating a loss function for each layer
 - Creating the optimizer
 - Training the model
- Introducing GANs
- DCGAN
 - Defining the generator network
 - Transposed convolutions
 - Batch normalization
 - Generator
 - Defining the discriminator network
 - Defining loss and optimizer
 - Training the discriminator
 - Training the discriminator with real images
 - Training the discriminator with fake images
 - Training the generator network
 - Training the complete network
 - Inspecting the generated images
- Summary

8. Transfer Learning with Modern Network Architectures

- Modern network architectures
 - ResNet
 - Creating PyTorch datasets

```
Creating loaders for training and validation
Creating a ResNet model
Extracting convolutional features
Creating a custom PyTorch dataset class for the pre-convoluted
features and loader
Creating a simple linear model
Training and validating the model

Inception
The Inception architecture
Creating an Inception model
Extracting convolutional features using register_forward_hook
Creating a new dataset for the convoluted features
Creating a fully connected model
Training and validating the model

Densely connected convolutional networks &#x2013; DenseNet
The _DenseBlock object
The _DenseLayer object
Creating a DenseNet model
Extracting DenseNet features
Creating a dataset and loaders
Creating a fully connected model and training it

Model ensembling
Creating models
Extracting the image features
Creating a custom dataset, along with data loaders
Creating an ensembling model
Training and validating the model

Encoder-decoder architecture&#xA0;
Encoder&#xA0;
Decoder
Encoder-decoder with attention&#xA0;

Summary
9. Deep Reinforcement Learning
Introduction to RL
Model-based RL
Model-free RL
Comparing on-policy and off-policy
```

Q-learning

- Value methods
- Value iteration
- Coded example – value iteration

Policy methods

- Policy iteration
- Coded example – policy iteration
- Value iteration versus policy iteration
- Policy gradient algorithm
- Coded example – policy gradient algorithm

Deep Q-networks

- DQN loss function
- Experience replay
- Coded example – DQN
- Double deep Q-learning

Actor-critic methods

- Coded example – actor-critic model
- Asynchronous actor-critic algorithm

Practical applications

Summary

Further reading

10. What's Next?

What's next?

- Overview of the book
- Reading and implementing research papers

Interesting ideas to explore

- Object detection
- Image segmentation
- OpenNMT in PyTorch
- Allen NLP
- fast.ai – making neural nets uncool again
- Open neural network exchange
- How to keep yourself updated

Summary

Other Books You May Enjoy

Leave a review - let other readers know what you think

Preface

PyTorch is grabbing the attention of deep learning researchers and data science professionals due to its accessibility, efficiency, and the fact of it being more native to the Python way of development. This book will get you up and running with one of the most cutting-edge deep learning libraries—PyTorch.

In this second edition, you'll learn about the various fundamental building blocks that power modern deep learning using the new features and offerings of the PyTorch 1.x library. You will learn how to solve real-world problems using **convolutional neural networks (CNNs)**, **recurrent neural networks (RNNs)**, and **long short-term memory (LSTM)** networks. You will then get to grips with the concepts of various state-of-the-art modern deep learning architectures, such as ResNet, DenseNet, and Inception. You will learn how to apply neural networks to various domains, such as computer vision, **natural language processing (NLP)**, and more. You will see how to build, train, and scale a model with PyTorch and dive into complex neural networks such as generative networks and autoencoders for producing text and images. Furthermore, you will learn about GPU computing and how GPUs can be used to perform heavy computations. Lastly, you will learn how to work with deep learning-based architectures for transfer learning and reinforcement learning problems.

By the end of the book, you'll be able to implement deep learning applications in PyTorch with ease.

Who this book is for

This book is for data scientists and machine learning engineers who are looking to explore deep learning algorithms using PyTorch 1.x. Those who wish to migrate to PyTorch 1.x will find this book insightful. To make the most out of this book, working knowledge of Python programming and some knowledge of machine learning will be helpful.

What this book covers

[Chapter 1](#), *Getting Started with Deep Learning Using PyTorch*, introduces you to the history of deep learning, machine learning, and AI. This chapter covers how they are related to neuroscience and other areas of science, such as statistics, information theory, probability, and linear algebra.

[Chapter 2](#), *Building Blocks of Neural Networks*, covers the various math concepts that are required to understand and appreciate neural networks using PyTorch.

[Chapter 3](#), *Diving Deep into Neural Networks*, shows you how to apply neural networks to various real-world scenarios.

[Chapter 4](#), *Deep Learning for Computer Vision*, covers the various building blocks of modern CNN architectures.

[Chapter 5](#), *Natural Language Processing with Sequence Data*, shows you how to handle sequence data, particularly text data, and teaches you how to create a network model.

[Chapter 6](#), *Implementing Autoencoders*, introduces the idea of semi-supervised learning algorithms through an introduction of autoencoders. It also covers how to use restricted Boltzmann machines to understand the probability distribution of data.

[Chapter 7](#), *Working with Generative Adversarial Networks*, shows you how to build generative models capable of producing text and images.

[Chapter 8](#), *Transfer Learning with Modern Network Architectures*, covers modern architectures such as ResNet,

Inception, DenseNet, and Seq2Seq, and also shows you how to use pre-trained weights for transfer learning.

[Chapter 9](#), *Deep Reinforcement Learning*, starts with a basic introduction to reinforcement learning, including coverage of agents, state, action, reward, and policy. It also contains hands-on code for deep learning-based architectures for reinforcement learning problems, such as Deep Q networks, policy gradient methods, and actor-critic models.

[Chapter 10](#), *What Next?*, gives you a quick overview of what the book covered along with information on how you can keep up to date with the latest advances in the field.

To get the most out of this book

Working knowledge of Python will be useful.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the Support tab.
3. Click on Code Downloads.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Deep-Learning-with-PyTorch-1.x>. 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: [here](#)

http://www.packtpub.com/sites/default/files/downloads/9781838553005_ColorImages.pdf.

Conventions used

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

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Let's use simple Python functions, such as `split` and `list`, to convert the text into tokens."

A block of code is set as follows:

```
toy_story_review = "Just perfect. Script, character, animation....this  
manages to break free of the yoke of 'children's movie' to simply be one  
of the best movies of the 90's, full-stop."  
  
print(list(toy_story_review))
```

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

```
['J', 'u', 's', 't', ' ', 'p', 'e', 'r', 'f', 'e', 'c', 't', '.', ' ',  
'S', 'c', 'r', 'i', 'p', 't', ' ', ' ', 'c', 'h', 'a', 'r', 'a', 'c',  
't', 'e', 'r', ' ', ' ', 'a', 'n', 'i', 'm', 'a', 't', 'i', 'o', 'n',  
'.', ' ', ' ', ' ', 't', 'h', 'i', 's', ' ', 'm', 'a', 'n', 'a', 'g',  
'e', 's', ' ', 't', 'o', ' ', 'b', 'r', 'e', 'a', 'k', ' ', 'f', 'r',  
'e', 'e', ' ', 'o', 'f', ' ', 't', 'h', 'e', ' ', 'y', 'o', 'k', 'e',  
' ', 'o', 'f', ' ', "", 'c', 'h', 'i', 'l', 'd', 'r', 'e', 'n', "", 's',  
' ', 'm', 'o', 'v', 'i', 'e', "", ' ', 't', 'o', ' ', 's', 'i', 'm',  
'p', 'l', 'y', ' ', 'b', 'e', ' ', 'o', 'n', 'e', ' ', 'o', 'f', ' ',  
't', 'h', 'e', ' ', 'b', 'e', 's', 't', ' ', 'm', 'o', 'v', 'i', 'e',  
's', ' ', 'o', 'f', ' ', 't', 'h', 'e', ' ', '9', '0', "", 's', ' ',  
' ', 'f', 'u', 'l', 'l', ' ', 's', 't', 'o', 'p', '.']
```

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

```
| pip install torchtext
```

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 be helping you to understand **recurrent neural networks (RNNs)**."



Warnings or important notes appear like this.



Tips and tricks appear like this.

TIP

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.

Section 1: Building Blocks of Deep Learning with PyTorch 1.x

In this section, you will be introduced to the concepts of deep learning and the various deep learning frameworks.

This section contains the following chapters:

- [Chapter 1](#), *Getting Started with Deep Learning Using PyTorch*
- [Chapter 2](#), *Building Blocks of Neural Networks*

Getting Started with Deep Learning Using PyTorch

Deep learning (DL) has revolutionized industry after industry. It was once famously described by Andrew Ng on Twitter as follows:

"Artificial intelligence is the new electricity!"

Electricity transformed countless industries; now, **artificial intelligence (AI)** will do the same.

AI and DL are used as synonyms, but there are substantial differences between the two. Let's demystify the terminology that's used in the industry so that you, as a practitioner, will be able to differentiate between signal and noise.

In this chapter, we will cover the following different parts of AI:

- Exploring artificial intelligence
- Machine learning in the real world
- Applications of deep learning
- Deep learning frameworks
- Setting up PyTorch 1.x

Exploring artificial intelligence

Countless articles discussing AI are published every day. The trend has increased in the last 2 years. There are several definitions of AI floating around the web, with my favorite being *the automation of intellectual tasks normally performed by humans.*

The history of AI

Since you've picked up this book, you may be well aware of the recent hype in AI. But it all started when John McCarthy, then a young assistant professor at Dartmouth, coined the term *artificial intelligence* in 1955, which he defined as a field pertaining to the science and engineering of intelligent machines. This kick-started the first wave of AI, which was primarily driven by symbolic reasoning; its outcomes were astonishing, to say the least. AI that was developed during this time was capable of reading and solving high-school Algebra problems [STUDENT], proving theorems in Geometry [SAINT], and learning the English language [SHRDLU]. Symbolic reasoning is the use of complex rules nested in if-then statements.

The most promising work in this era, though, was the perceptron, which was introduced in 1958 by Frank Rosenblatt. The perceptron, when combined with intelligent optimization techniques that were discovered later, laid the foundations for deep learning as we know it today.

It wasn't plain sailing for AI, though, since the funding in the field significantly reduced during lean periods, mostly due to overpromising initial discoveries and, as we were yet to discover, a lack of data and compute power. The rise in prominence of **machine learning (ML)** in the early nineties bucked the trend and created significant interest in the field. First, we need to understand the paradigm of ML and its relationship with DL.

Machine learning in the real world

ML is a subfield of AI that uses algorithms and statistical techniques to perform a task without the use of any explicit instructions. Instead, it relies on underlying statistical patterns in the data.

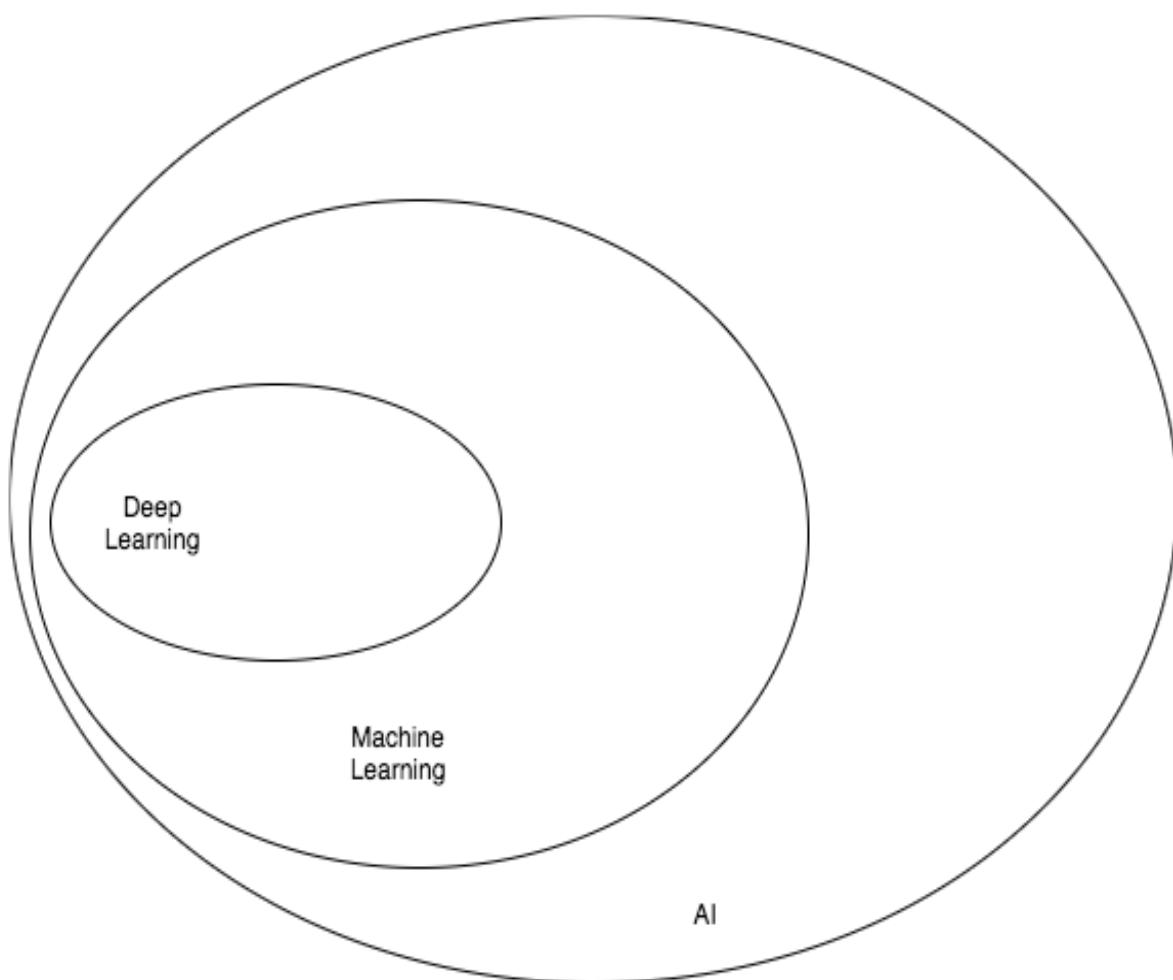
To build successful machine learning models, we need to provide ML algorithms with labeled data. The success of this approach was heavily dependent on the available data and compute power so that large amounts of data could be used.

So, why DL?

Most ML algorithms perform well on structured data, such as sales predictions, recommendation systems, and marketing personalization. An important factor for any ML algorithm is feature engineering and data scientists need to spend a lot of time exploring possible features with high predictive power for ML algorithms. In certain domains, such as computer vision and **natural language processing (NLP)**, feature engineering is challenging as features that are important for one task may not hold up well for other tasks. This is where DL excels—the algorithm itself engineers features in a non-linear space so that they are important for a particular task.

Traditional ML algorithms still outperform DL methods when there is a paucity of data, but as data increases, the performance of traditional machine learning algorithms tends to plateau and deep learning algorithms tend to significantly outperform other learning strategies.

The following diagram shows the relationship DL has with ML and AI:



To summarize this, DL is a subfield of machine learning; feature engineering is where the algorithm non-linearly explores its space.

Applications of deep learning

DL is at the center of the most important innovations of the 21st century, from detecting tumors with a lower error rate than radiologists to self-driving cars. Let's quickly look at a few DL applications.

Automatic translation of text from images

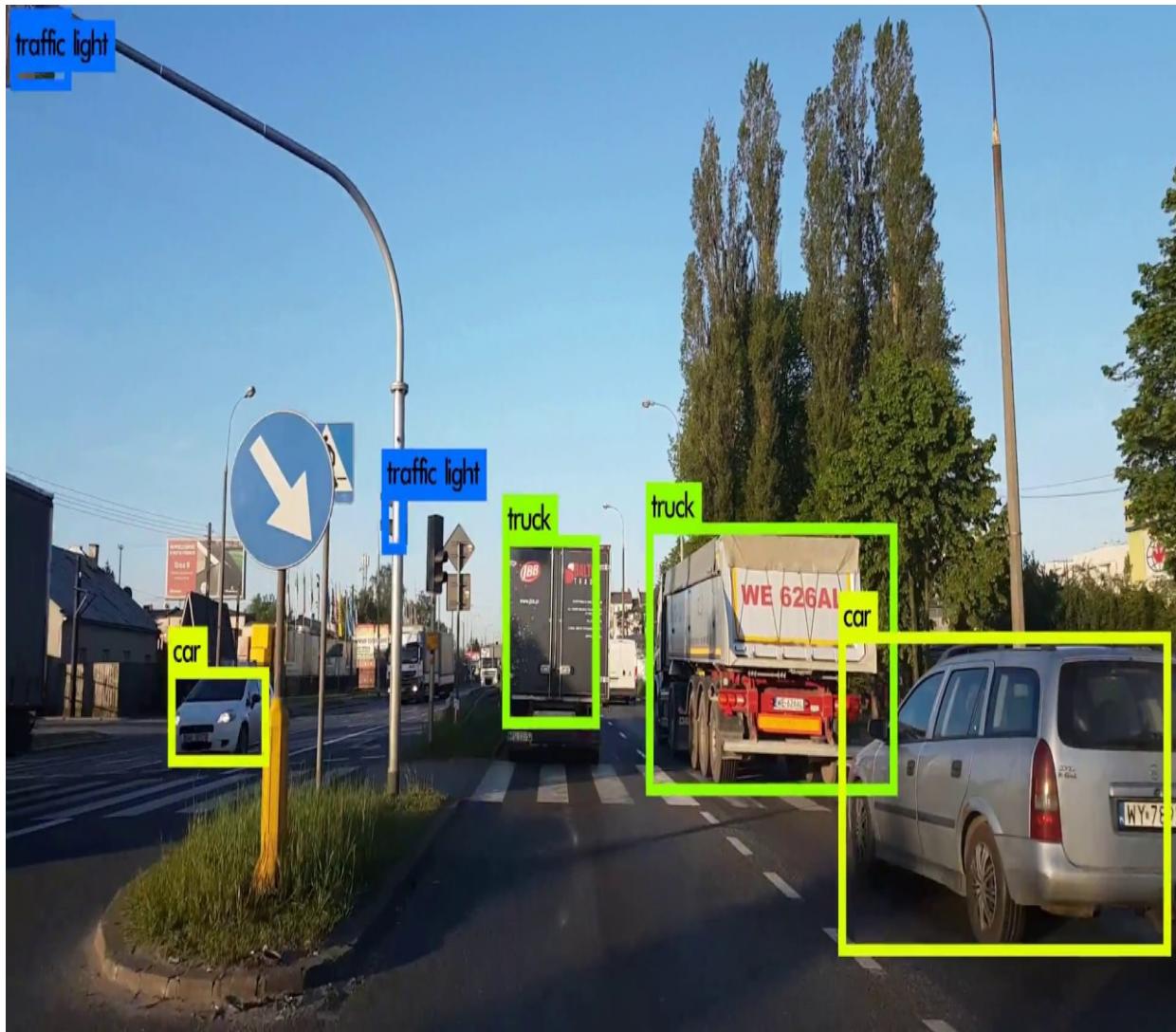
A 2015 blog from Google details how the team at Google can translate text from images. The following image shows the steps involved:



First, a DL algorithm is used to perform **optical character recognition (OCR)** and recognize the text from the image. Later, another DL algorithm is used to translate the text from the source language to the language of choice. The improvements we see today in machine translation are attributed to the switch to DL from traditional methods.

Object detection in self-driving cars

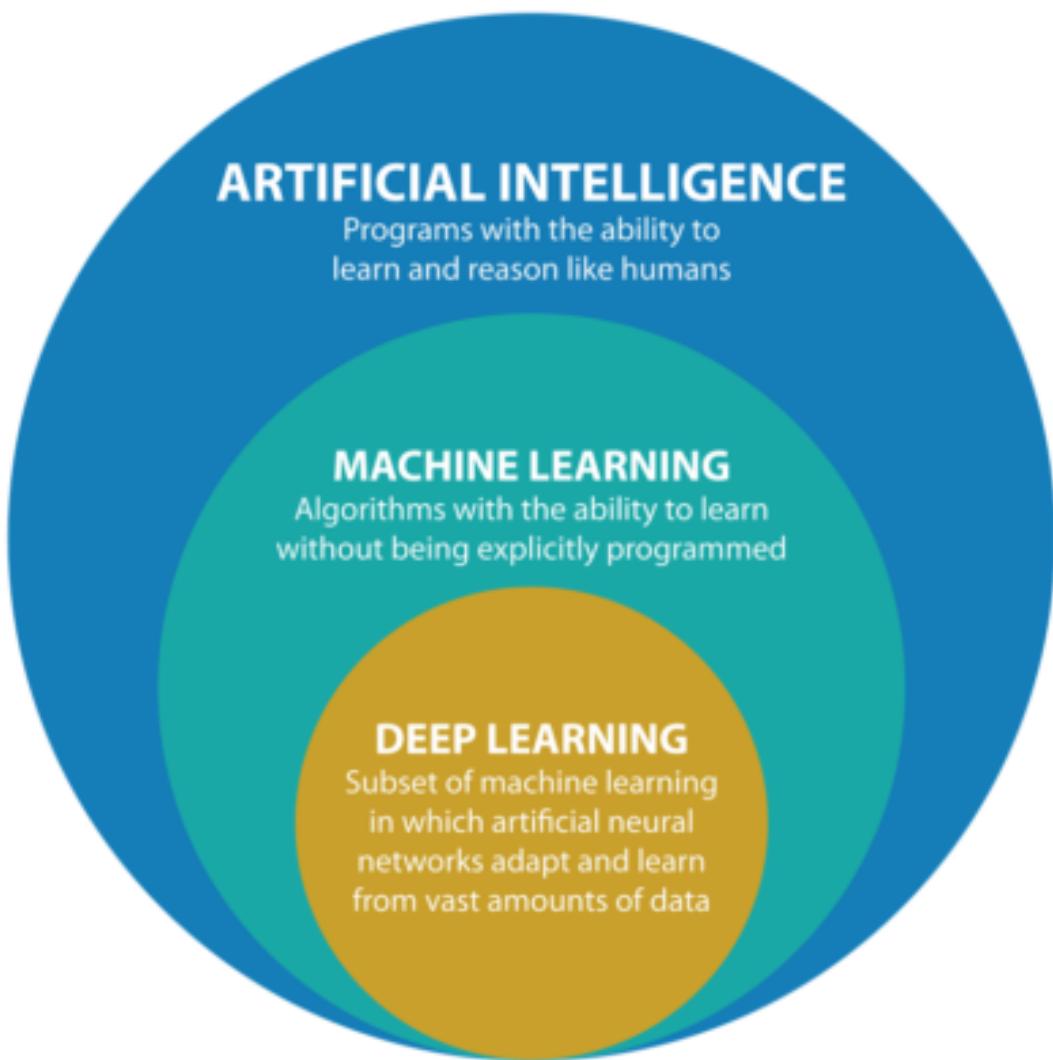
Tesla did a deep dive into their autonomous driving system for investors in 2019, where they mentioned how they use deep neural networks to detect objects from cameras in the car. The output of this algorithm is used by the proprietary self-driving policy developed by Tesla:



The preceding image is the output of an object detection deep learning network. The semantic information it has captured from the visual image is crucial for self-driving tasks.

Deep learning frameworks

It used to be extremely hard to write code for deep learning algorithms since writing code for the learning step, which involved chaining complex derivatives, was extremely error-prone and lengthy. DL frameworks used ingenious heuristics to automate the computation of these complex derivatives. The choice of such heuristics significantly changes the way these frameworks work. The following diagram shows the current ecosystem of DL frameworks:



TensorFlow is the most popular deep learning framework but the simplicity and usefulness of PyTorch has made DL research accessible to a lot of people. Let's look at why using PyTorch can speed up our DL research and development time significantly.

Why PyTorch?

To compute complex chained derivatives, TensorFlow uses a **Define and Run** paradigm, whereas PyTorch uses a more ingenuous **Define by Run** paradigm. Let's delve deeper into this by looking at the following image, where we will be computing the sum of the series $1 + 1/2 + 1/4 + 1/8 \dots$, which should add up to 2:

Computing $1 + 1/2 + 1/4 + 1/8 + \dots = 2$



```
import torch

x = torch.Tensor([0.])
y = torch.Tensor([1.])

for iteration in range(50):
    x = x + y
    y = y / 2

print(x)

tensor([2.])
```

We can immediately see how succinct and simple it is to write code to perform operations in PyTorch. This difference is more widely noticeable in more complex scenarios.

As the head of AI at Tesla and one of the biggest thought leaders in computer vision at the moment, Andrej Karpathy tweeted—*I've been using PyTorch for a few months now and I've never felt better. I have more energy. My skin is clearer. My eyesight has improved.* PyTorch definitely makes the lives of people writing DL code better.

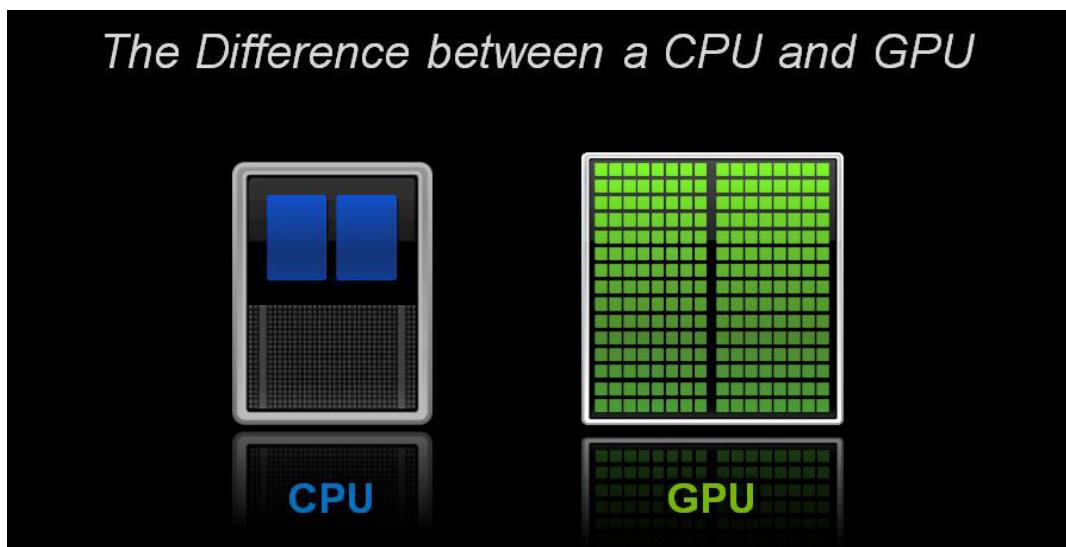
This **Define by Run** paradigm also has many advantages other than just creating cleaner and simpler code. Debugging also becomes extremely easy and all of the tools that you currently use to debug Python code can be used with PyTorch as well. This is a significant advantage because, as networks get more and more complex, debugging your networks with ease will be a lifesaver.

What's new in PyTorch v1.x?

PyTorch 1.x expands on its flexibility and tries to unify research and production capabilities into a single framework. Caffe2, a production-grade deep learning framework, is integrated into PyTorch, allowing us to deploy PyTorch models to mobile operating systems and high-performance C++ services. PyTorch v1.0 also natively supports exporting models into the ONNX format, which allows PyTorch models to be imported into other DL frameworks. It truly is an exciting time to be a PyTorch developer!

CPU versus GPU

CPUs have fewer but more powerful compute cores, whereas GPUs have a large number of lower-performance cores. CPUs are more suited to sequential tasks, whereas GPUs are suitable for tasks with significant parallelization. In summary, a CPU can execute large, sequential instructions but can only execute a small number of instructions in parallel in contrast to a GPU, which can execute hundreds of small instructions in parallel:



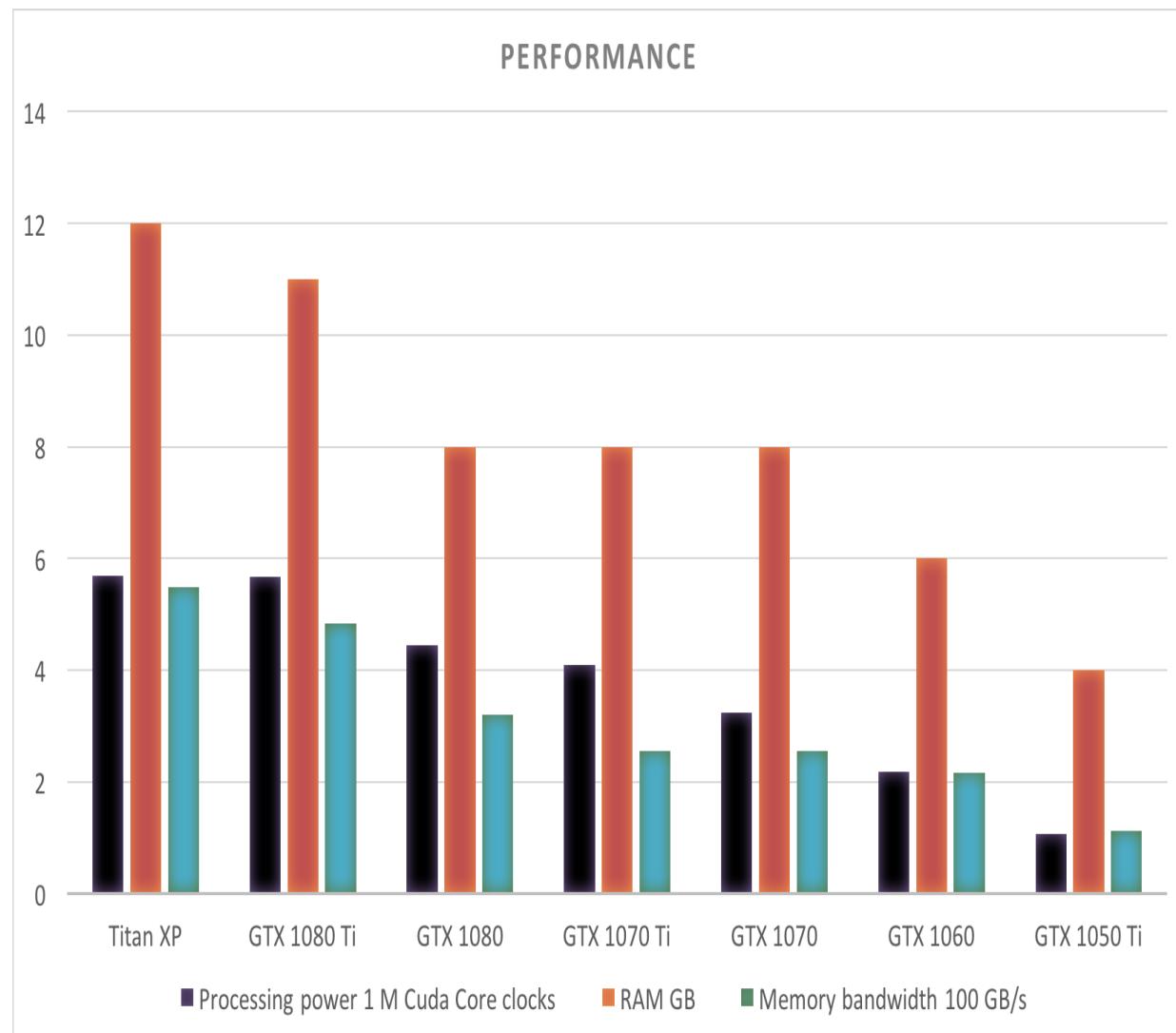
While using DL, we will be performing a large number of linear algebraic operations that are more suited to a GPU and can provide a significant boost in terms of the time it takes to train a neural network.

What is CUDA?

CUDA is a framework developed by NVIDIA that allows us to use **General Purpose Computing on Graphics Processing Units (GPGPU)**. It is a widely used framework written in C++ that allows us to write general-purpose programs that run on GPUs. Almost all deep learning frameworks leverage CUDA to execute instructions on GPUs.

Which GPUs should we use?

Since most deep learning frameworks, including PyTorch, use NVIDIA's CUDA framework, it is highly recommended that you buy and use a NVIDIA GPU for deep learning. Let's do a quick comparison of a few NVIDIA GPU models:



What should you do if you don't have a GPU?

There are a lot of Cloud services such as Azure, AWS, and GCP that provide instances that have GPUs and all the required deep learning software preinstalled. FloydHub is a great tool for running deep learning models in the cloud. However, the single most important tool you should definitely check out is Google's Colaboratory, which provides high-performance GPUs for free so that you can run deep learning models.

Setting up PyTorch v1.x

Throughout this book, we will be using the Anaconda Distribution for Python and PyTorch 1.x. You can follow along with the code by executing the relevant command based on your current configuration by going to the official PyTorch website (<https://pytorch.org/get-started/locally/>).

Installing PyTorch

PyTorch is available as a Python package and you can either use `pip` or `conda` to build it. Alternatively, you can build it from the source. The recommended approach for this book is to use the Anaconda Python 3 distribution. To install Anaconda, please refer to the Anaconda official documentation at <https://conda.io/docs/user-guide/install/index.html>. All the examples will be available as Jupyter Notebooks in this book's GitHub repository. I would strongly recommend that you use Jupyter Notebook since it allows you to experiment interactively. If you already have Anaconda Python installed, then you can proceed with the following instructions for PyTorch installation.

For GPU-based installation with Cuda 8, use the following command:

```
| conda install pytorch torchvision cuda80 -c soumith
```

For GPU-based installation with Cuda 7.5, use the following command:

```
| conda install pytorch torchvision -c soumith
```

For non-GPU-based installation, use the following command:

```
| conda install pytorch torchvision -c soumith
```

At the time of writing, PyTorch does not work on Windows machines, so you can try a **virtual machine (VM)** or Docker image.

Summary

In this chapter, we've learned about the history of AI, why we use deep learning, multiple frameworks in the deep learning ecosystem, why PyTorch is an important tool, why we use GPUs for deep learning, and setting up PyTorch v1.0.

In the next chapter, we will delve into the building blocks of neural networks and learn how to write PyTorch code to train them.

Building Blocks of Neural Networks

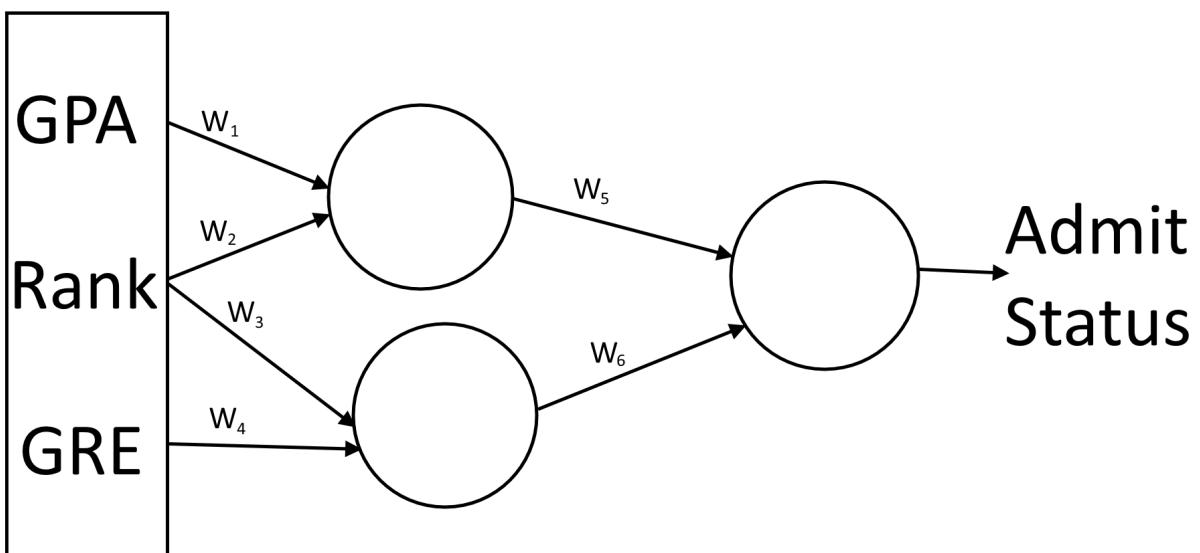
Understanding the basic building blocks of a neural network, such as tensors, tensor operations, and gradient descent, is important for building complex neural networks. In this chapter, we will provide a general overview of neural networks and at the same time dive deep into the foundations of the PyTorch API. The original idea of a neural network was inspired by biological neurons that are found in the human brain, but, at the time of writing, the similarities between them are only superficial and any comparisons between the two systems may lead to incorrect assumptions about either of these systems. So, we won't ponder the similarities between the two systems and, instead, directly dive into the anatomy of the neural networks that are used in AI.

In this chapter, we will cover the following topics:

- What is a neural network?
- Building a neural network in PyTorch
- Understanding PyTorch tensors
- Understanding tensor operations

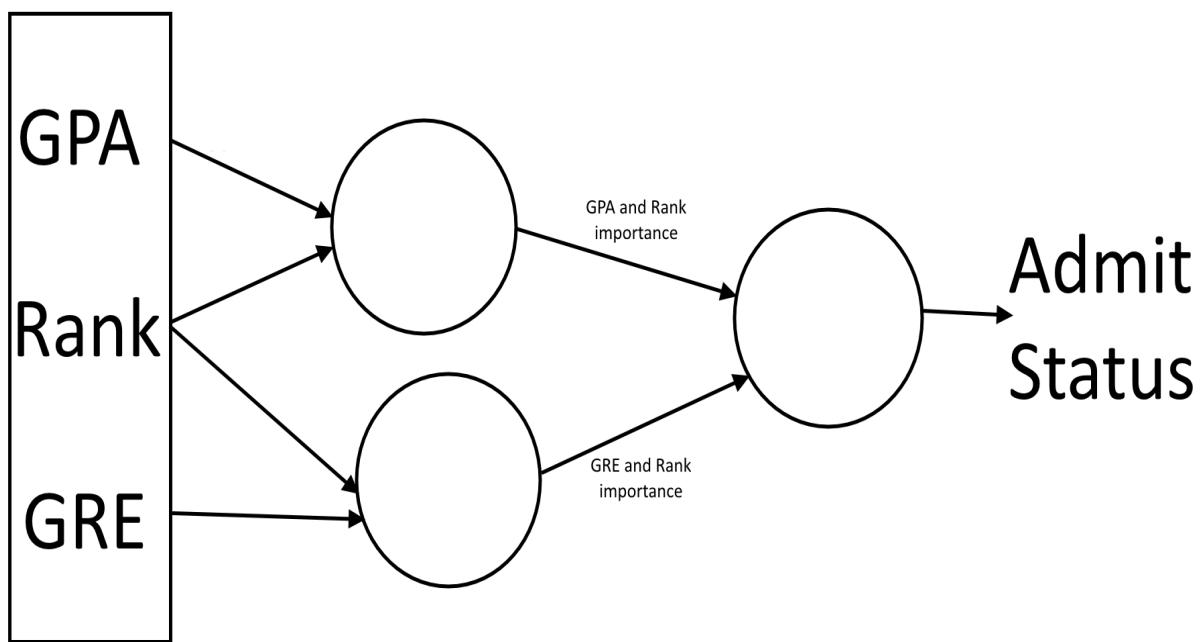
What is a neural network?

In short, a neural network is an algorithm that learns about the relationship between the input variables and their associated target variables. For example, if you have a dataset consisting of students' GPAs, GRE scores, the rank of the university, and the students' admit status into a college, we can use a neural network to predict the admit status (target variable) of a student given their GPA, GRE scores, and the rank of the university (input variables):



In the preceding diagram, each of the arrows represents a weight. These weights are learned from the instances of training data, $\{ (x_1, y_1), (x_2, y_2), \dots, (x_m, y_m) \}$, so that the composite feature that was created from the operations can predict the admit status of the student.

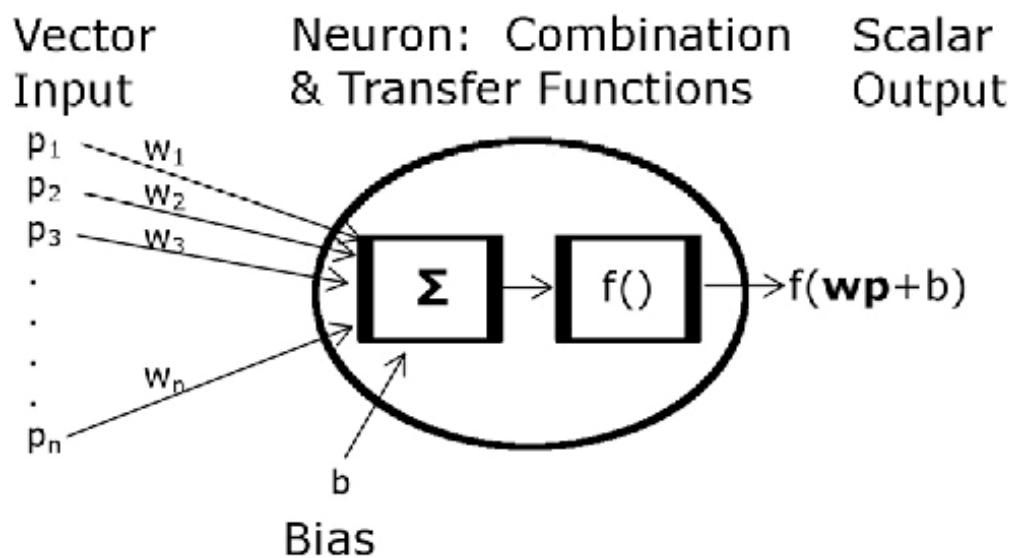
For example, the network may learn the importance of GPA/GRE in terms of the rank of the institution, as shown in the following diagram:



Understanding the structure of neural networks

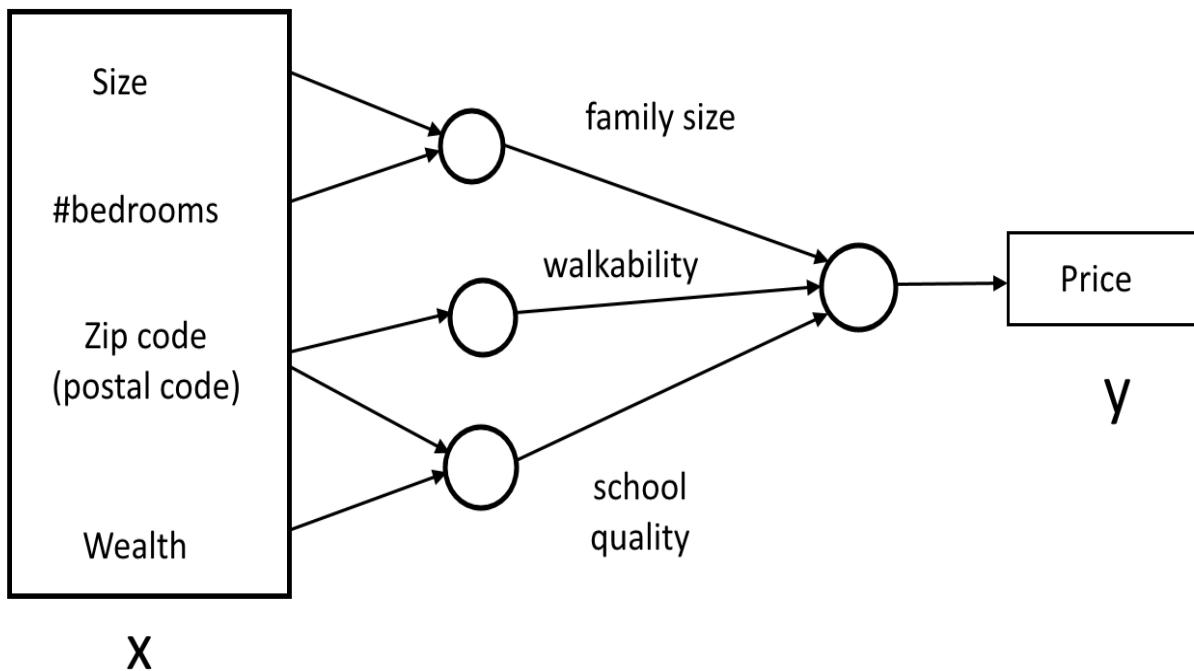
Operations in a neural network are built from two underlying computations. One is a dot product between the weight vector and their corresponding input variable vector, while the other is a function that transforms the product into a non-linear space. We will learn about several types of these functions in the next chapter.

Let's break this down further: The first dot product learns of a mixed concept since it creates a mixture of input variables that depend on the importance of each input variable. Passing these important features into a non-linear function allows us to build an output that is more powerful than just using a traditional linear combination:



By using these operations as building blocks, we can build robust neural networks. Let's break down the neural network example from earlier; the neural network learns about features that, in turn, are the best predictors of the target variable. So, each layer of a neural

network learns features that can help the neural network predict the target variable better:

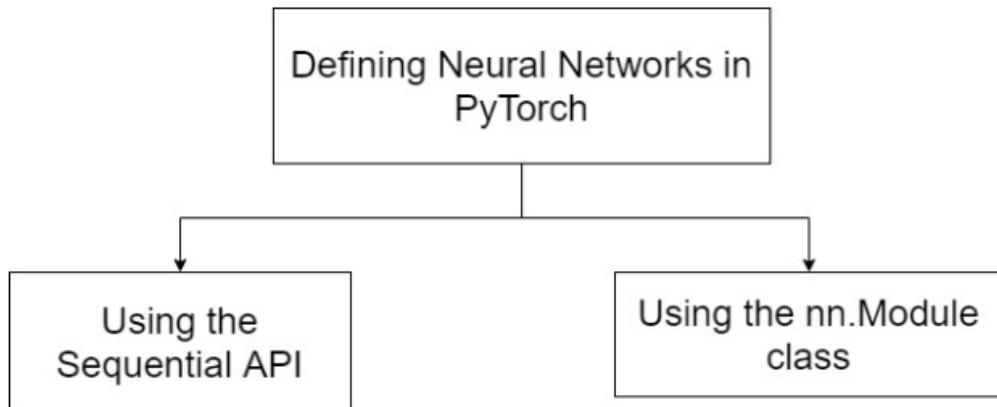


In the preceding diagram, we can see how the price of a house can be predicted using a neural network.

Building a neural network in PyTorch

Let's start off by building a neural network in PyTorch that will help us predict the admit status of a college student. There are two ways to build a neural network in PyTorch. First, we can use the simpler `torch.nn.Sequential` class, which allows us to pass the sequence of operations in our desired neural network as the argument, while instantiating our network.

The other way, which is a more complex and powerful yet elegant approach, is to define our neural network as a class that inherits from the `torch.nn.Module` class:



We will build out our neural network using these two patterns, both of which are defined by the PyTorch API.

PyTorch sequential neural network

All the commonly used operations in a neural network are available in the `torch.nn` module. Therefore, we need to start by importing the required modules:

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

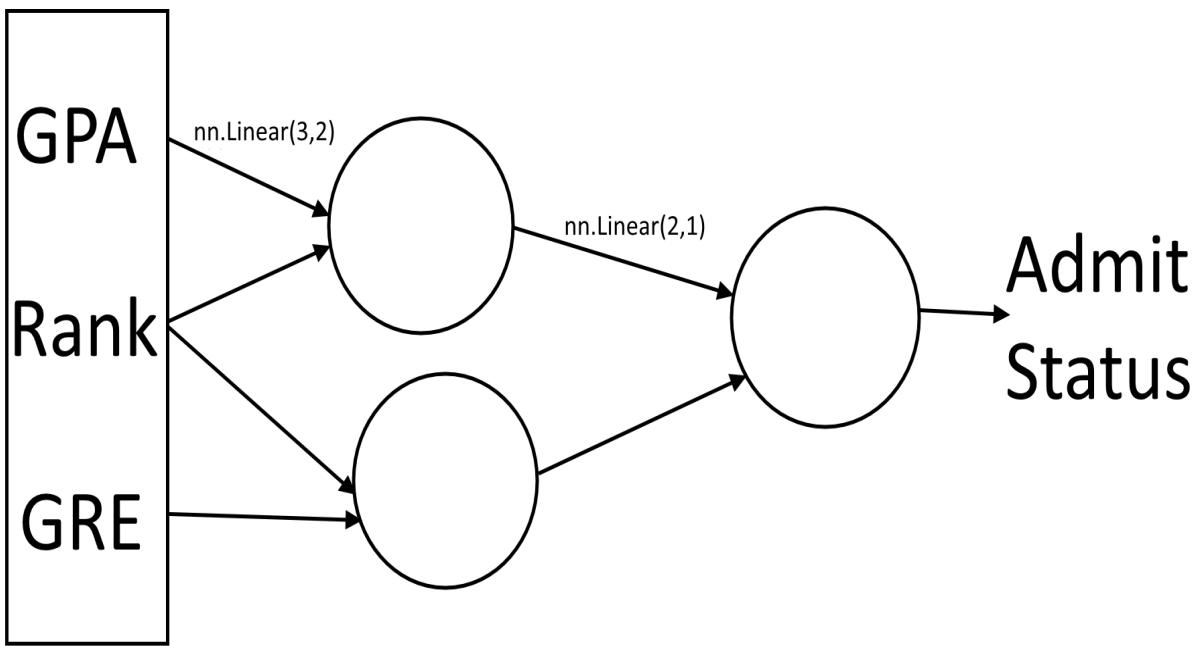
Now, let's look at how we build a neural network using the `torch.nn.Sequential` class. We use operations defined in the `torch.nn` module and pass them as arguments in a sequential fashion to the `torch.nn.Sequential` class to instantiate our neural network. After we import our operations, our neural network code should look as follows:

```
| My_neuralnet = nn.Sequential(operationOne, operationTwo...)
```

The most commonly used operation while building a neural network is the `nn.Linear()` operation. It takes two arguments: `in_features` and `out_features`. The `in_features` argument is the size of the input. In our case, we have three input features: GPA, GRE, and the rank of the university. The `out_features` argument is the size of the output, which, in our case, is two as we want to learn about two features from the input that can help us predict the admit status of a student. Essentially, the `nn.Linear(in_features, out_features)` operation takes the input and creates weight vectors to perform the dot product.

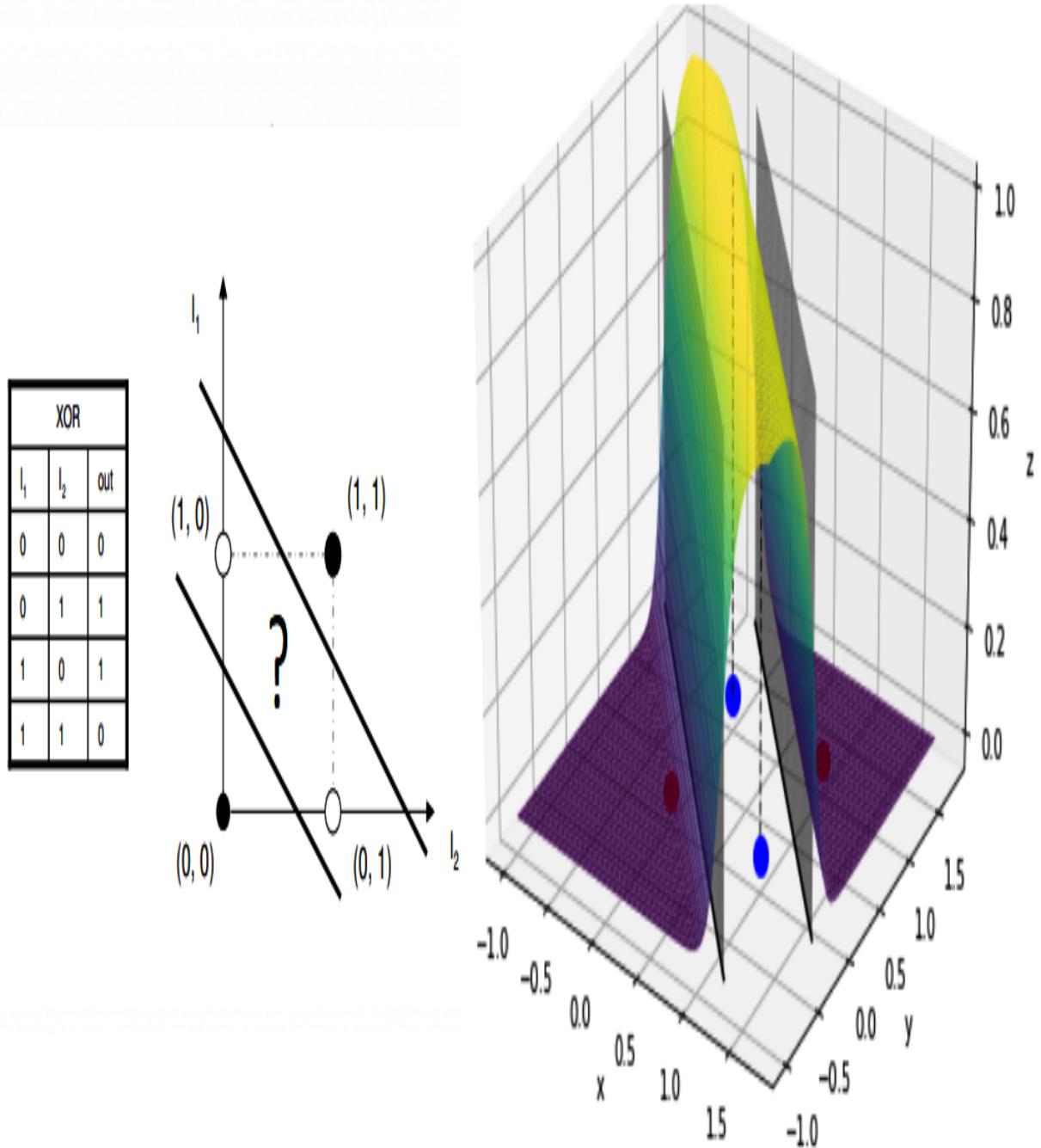
In our case, `nn.Linear(in_features = 3, out_features = 2)` would create two vectors: [w11, w12, w13] and [w21, w22, w23]. When the input, [xGRE, xGPA, xrank], is passed to the neural network, we would create a vector of two outputs, [h1, h2], which would be the result of [w11 . xGRE + w12. xGPA + w13 . xrank , w21 . xGRE + w22 . xGPA + w23 . xrank].

This pattern continues downstream when you want to keep adding more layers to your neural network. The following diagram shows the neural network's structure after it's been translated into the `nn.Linear()` operations:



Great! But adding more linear operations does not exploit the power of neural networks. We also have to transform these outputs into a non-linear space using one of several non-linear functions. The types of these functions and the benefits and pitfalls of each of them will be described in more detail in the next chapter. For now, let's use one of the most commonly used non-linear functions, that is, the **Rectified Linear Unit**, also known as **ReLU**. PyTorch provides us with an inbuilt ReLU operator that can be accessed by calling `nn.ReLU()`. The following diagram shows how non-linear functions can classify or solve learning problems where linear functions fail:

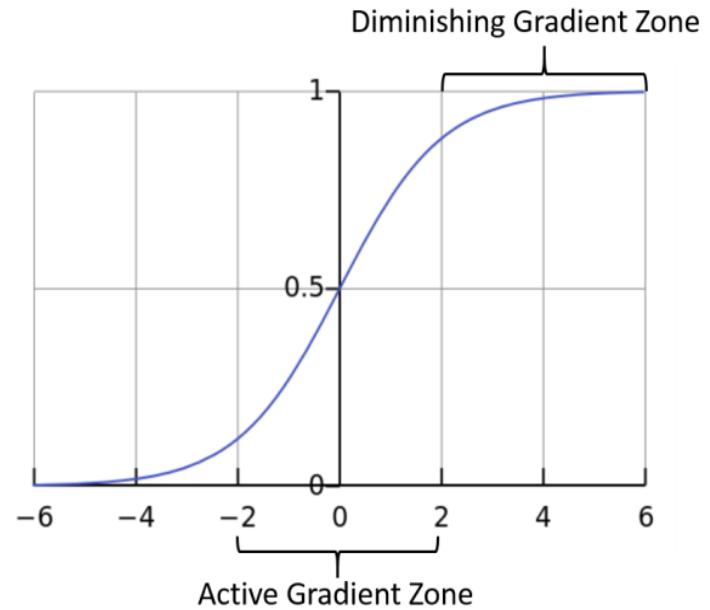
https://www.tensorflow.org/tutorials/quick_start/guide



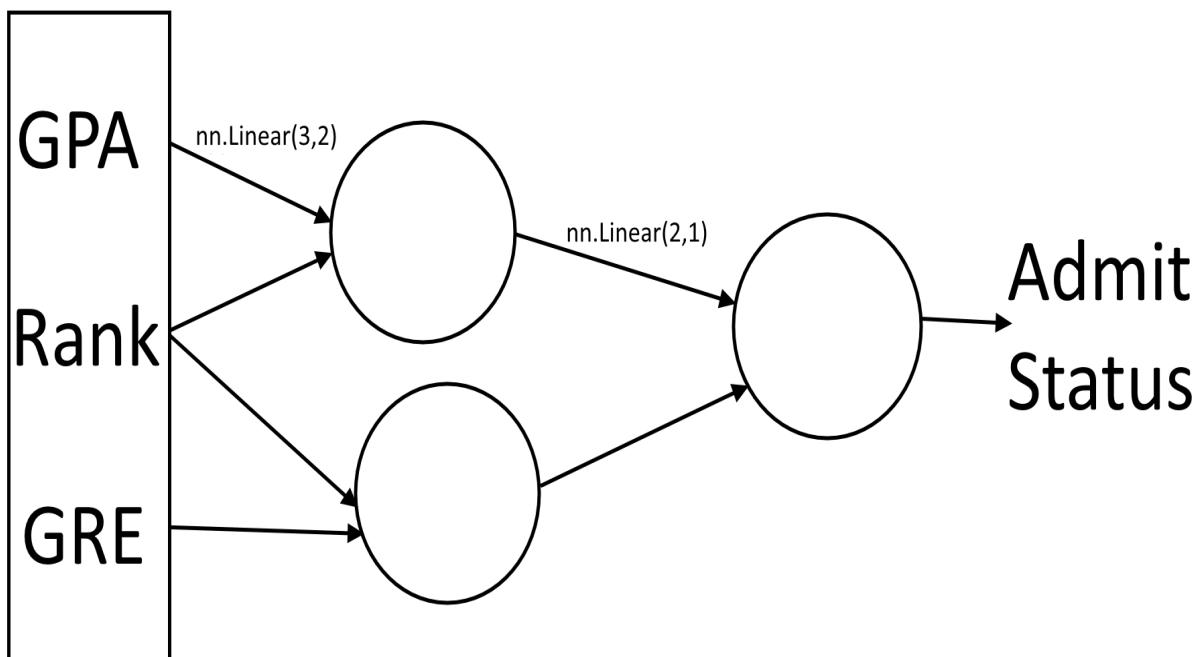
Finally, to get our predictions, we need to squash the output between 0 and 1. These states refer to a non-admit and an admit, respectively. The sigmoid function, as shown in the following diagram, is the most commonly used function to transform a continuous quantity between

negative and positive infinity into a value between 0 and 1. In PyTorch, we simply need to call the `nn.Sigmoid()` operation:

$$A = \frac{1}{1+e^{-x}}$$



Now, let's put our neural network code together in PyTorch so that we get a network that looks like what's shown in the following diagram in terms of its structure:



The code for doing this is as follows:

```
import torch
import torch.nn as nn
my_neuralnet = nn.Sequential(nn.Linear(3,2),
    nn.ReLU(),
    nn.Linear(2, 1),
    nn.Sigmoid())
```

That's it! It's that simple to put a neural network together in PyTorch. The `my_neuralnet` Python object contains our neural network. We will look at how to use it a little later. For now, let's look at how to build neural networks using the more advanced API, which is based on defining a class that inherits from the `nn.Module` class.

Building a PyTorch neural network using nn.Module

Defining a neural network using the `nn.Module` class is also simple and elegant. It starts by defining a class that will inherit from the `nn.Module` class and overrides two methods: the `__init__()` and `forward()` methods. The `__init__()` method should include the operations that are a part of the layers in our desired neural network. On the other hand, the `forward()` method should describe the flow of data through these desired layer operations. So, the structure of the code should look similar to the following:

```
class MyNeuralNet(nn.Module):
    # define the __init__() method
    def __init__(self, other_features_for_initialization):
        # Initialize Operations for Layers
        # define the forward() method
    def forward(self, x):
        # Describe the flow of data through the layers
```

Let's understand this pattern in more detail. The `class` keyword helps define a Python class, followed by any arbitrary name you want your class to be. In this case, it is `MyNeuralNet`. Then, the argument that's passed in the parenthesis is the class that our currently defined class would inherit from. So, we always start off with the `MyNeuralNet(nn.Module)` class.

`self` is the arbitrary first argument that is passed to every method defined in the class. It represents the instance of the class and can be used to access the attributes and methods defined in the class.

The `__init__()` method is a reserved method in Python classes. It is also known as a constructor. Whenever an object of the class is instantiated, the code wrapped inside the `__init__()` method is run. This helps us set up all our neural network operations once an object of our neural network class has been instantiated.

One thing to note is that, once we define the `__init__()` method inside our neural network class, we lose access to all of the code defined inside the `__init__()` method of the `nn.Module` class. Luckily, the Python `super()` function can help us run the code in the `__init__()` method of the `nn.Module` class. All we need to do is use the `super()` function in the first line inside our new `__init__()` method. Using the `super()` function to gain access to the `__init__()` method is pretty straightforward; we just use `super(ClassName, self).__init__()`. In our case, this would be `super(MyNeuralNet, self).__init__()`.

Now that we know what goes into writing the first line of code for our `__init__()` method, let's look at what other code we need to include in the definition of the `__init__()` method. We have to store the operations defined in PyTorch as attributes of `self`. In our case, we have two `nn.Linear` operations: one from the input variables to the two nodes in the neural network layer and another from these nodes to the output node. So, our `__init__()` method would look as follows:

```
| class MyNeuralNet(nn.Module):
|   def __init__(self):
|     super(MyNeuralNet, self).__init__()
|     self.operationOne = nn.Linear(3, 2)
|     self.operationTwo = nn.Linear(2, 1)
```

In the preceding code, we store the operations for our desired neural network as attributes inside `self`. You should be comfortable with storing operations from PyTorch as attributes in `self`. The pattern that we use to do this is as follows:

```
|   self.desiredOperation = PyTorchOperation
```

However, there is one glaring mistake in the preceding code: the inputs to `nn.Linear` are hardcoded, so if the input size changes, we have to rewrite our neural network class again. Therefore, it's good practice to use variable names instead and pass them as arguments when instantiating the object. The code would look as follows:

```
| def __init__(self, input_size, n_nodes, output_size):
|   super(MyNeuralNet, self).__init__()
```

```
| self.operationOne = nn.Linear(input_size, n_nodes)
| self.operationTwo = nn.Linear(n_nodes, output_size)
```

Now, let's dive into the implementation of the `forward()` method. This method takes two arguments: the `self` argument and the arbitrary `x` argument, which is a placeholder for our real data.

We have already looked at the `nn.ReLU` operation, but there is also the more convenient functional interface defined in PyTorch that allows us to better describe the flow of data. It is important to note that these functional equivalents cannot be used inside the Sequential API. Our first job is to pass the data, which is represented by the `x` argument, to the first operation in our neural network. In PyTorch, passing the data to the first operation in our network is as simple as using `self.operationOne(x)`.

Then, using the PyTorch functional interface, we can pass the output of this operation to the nonlinear ReLU function by using `torch.nn.functional.relu(self.operationOne(x))`. Let's put everything together and define the `forward()` method. It's important to remember that the final output must be accompanied with the `return` keyword:

```
def forward(self, x):
    x = self.operationOne(x)
    x = nn.functional.relu(x)
    x = self.operationTwo(x)
    output = nn.functional.sigmoid(x)
    return output
```

Now, let's polish and compile everything so that we can define our neural network in PyTorch using the class-based API. The following code is how you would find most PyTorch code in open source communities:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
class MyNeuralNet(nn.Module):
    def __init__(self, input_size, n_nodes, output_size):
        super(MyNeuralNet, self).__init__()
        self.operationOne = nn.Linear(input_size, n_nodes)
        self.operationTwo = nn.Linear(n_nodes, output_size)
    def forward(self, x):
```

```
|     x = F.relu(self.operationOne(x))
|     x = self.operationTwo(x)
|     x = F.sigmoid(x)
| return x
```

Finally, to get access to our neural network, we have to instantiate an object of the `MyNeuralNet` class. We can do that as follows:

```
| my_network = MyNeuralNet(input_size = 3, n_nodes = 2, output_size = 1)
```

Now, we have access to our desired neural network with the `my_network` Python variable. We have built our neural network, so what's next? Can it predict the admit status of a student now? No. But we will get there. Before that, we need to understand how data should be represented in PyTorch so that our neural network understands it. That's where PyTorch Tensors come into play.

Understanding PyTorch Tensors

PyTorch Tensors is the engine that drives computation in PyTorch. If you have prior experience with Numpy, it'll be a breeze understanding PyTorch Tensors. Most of the patterns that you learned about with Numpy Arrays can be transformed into PyTorch Tensors.

Tensors are data containers and are a generalized representation of vectors and matrices. A vector is a first-order tensor since it only has one axis and would look like [x₁, x₂, x₃,..]. A matrix is a second-order tensor that has two axes and looks like [[x₁₁, x₁₂, x₁₃,..] , [x₂₁, x₂₂, x₂₃,..]]. On the other hand, a scalar is a zero-order tensor that only contains a single element, such as x₁. This is shown in the following diagram:

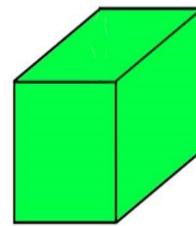
1D TENSOR /
VECTOR

5
7
4 5
1 2
- 6
3
2 2
1
6
3
- 9

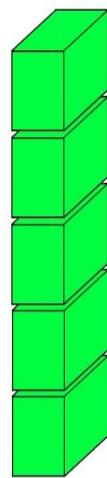
2D TENSOR /
MATRIX

- 9	4	2	5	7
3	0	1 2	8	6 1
1	2 3	- 6	4 5	2
2 2	3	- 1	7 2	6

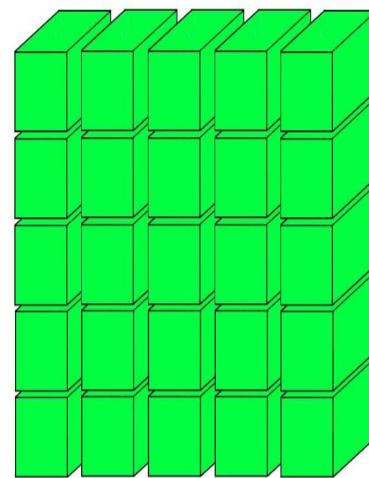
3D TENSOR /
CUBE



- 9	4	2	5	7
3	0	1 2	8	6 1
1	2 3	- 6	4 5	2
2 2	3	- 1	7 2	6



4D TENSOR
VECTOR OF CUBES



5D TENSOR
MATRIX OF CUBES

We can immediately observe that our dataset, which has columns of GPA, GRE, Rank, and Admit Status and rows of various observations, can be represented as a tensor of the second order:

admit	gre	gpa	rank	2 D TENSOR / MATRIX
0	380	3.61	3	
1	660	3.67	3	
1	800	4	1	
1	640	3.19	4	
0	520	2.93	4	
1	760	3	2	
1	560	2.98	1	
0	400	3.08	2	
1	540	3.39	3	

- 9	4	2	5	7
3	0	1 2	8	6 1
1	2 3	- 6	4 5	2
2 2	3	- 1	7 2	6

Let's quickly look at how to create PyTorch Tensors from Python lists:

```
import torch
first_order_tensor = torch.tensor([1, 2, 3])
print(first_order_tensor)
#tensor([1, 2, 3])
```

Accessing elements from this container is also straightforward and the indexing starts at 0 and ends at $n - 1$, where n is the number of elements in the container:

```
print(first_order_tensor[0])
#tensor(1)
```

`tensor(1)`, which we printed previously, is a zero-order tensor.

Accessing multiple elements is similar to how we'd do this in NumPy and Python, where `0:2` extracts elements starting at the element from index 0, up to, but not including, the element at index 2:

```
print(first_order_tensor[0:2])
#tensor([1, 2])
```

If you want to access all the elements of the tensor starting at a particular index, you could use `k:1`, where k is the index of the first element you want to extract:

```
print(first_order_tensor[1:])
#tensor([2, 3])
```

Now, let's understand how second-order tensors work:

```
second_order_tensor = torch.tensor([ [ 11, 22, 33 ],
                                     [ 21, 22, 23 ]
                                    ])

print(second_order_tensor)

#tensor([[11, 12, 13],
        [21, 22, 23]])
```

Accessing elements from a second-order tensor is a little more complex. Now, let's access element 12 from the tensor we created previously. It is important to view a second-order tensor as a tensor that's being built from two first-order tensors, for example, [[tensor of order one], [tensor of order one]]. Element 12 is inside the first tensor of order one and, within that tensor, it is at the second position, or index 1. Therefore, we can access element 22 by using [0, 1], where 0 describes the index of the tensor of order one and 1 describes the index of the element within the tensor of order one:

```
print(second_order_tensor[0, 1])
#tensor(12)
```

Now, let's do a small mental exercise: how would you access element 23 from the tensor we created? Yes! You are right! We can access it using [1, 2].

This same pattern holds for tensors of higher dimensions as well. It is important to note that the number of index positions you need to use to access the desired scalar element is equal to the order of the tensor. Let's carry out an exercise with a tensor of order 4!

Before we begin, let's visualize a tensor of order 4; it must be composed of tensors of order 3. Therefore, it must look similar to [[tensor of order 3] , [tensor of order 3], [tensor of order 3]...]. Each of these tensors of order 3 must, in turn, be composed of tensors of order 2, which would look like [[tensor of order 2], [tensor of order 2], [tensor of order 2], ...], and so on.

Here, you will find a fourth-order tensor. It has been graciously spaced for the convenience of visualization. In this exercise, we need to access the elements 1112, 1221, 2122, and 2221:

```
fourth_order_tensor = torch.tensor(  
[  
    [  
        [  
            [  
                [1111, 1112],  
                [1121, 1122]  
            ],  
            [  
                [1211, 1212],  
                [1221, 1222]  
            ]  
        ],  
        [  
            [  
                [2111, 2112],  
                [2121, 2122]  
            ],  
            [  
                [2211, 2212],  
                [2221, 2222]  
            ]  
        ]  
    ]  
)
```

Here, the tensor is composed of two third-order tensors, each of which has two tensors of order 2, which in turn contain two tensors of order one. Let's look at how to access element 2122; the others are left for you to do in your spare time. Element 2122 is contained in the second tensor of order three in our original tensor [[tensor of order 3], [*tensor of order 3]]. So, the first indexing position is 1. Next in the tensor of order 3, our desired element is in the first tensor of order 2 [[*tensor of order 2], [tensor of order 2]]. So, the second indexing position is 0. Inside the tensor of order 2, our desired element is in the second tensor of order one [[tensor of order 1], [* tensor of order 1], so the indexing position is 1. Finally, in the tensor of order one, our desired element is the second element [2121, 2122], for which the index position is 1. When we put this all together, we can index element 2122 by using `fourth_order_tensor[1, 0, 1, 1]`.

Understanding Tensor shapes and reshaping Tensors

Now that we know how to access elements from a Tensor, it is easy to understand Tensor shapes. All PyTorch Tensors have a `size()` method that describes the size of the Tensor across each of its axes. A PyTorch Tensor of order zero, or a scalar, does not have any axes, and so it doesn't have a quantifiable size. Let's look at the sizes of a few tensors in PyTorch:

```
my_tensor = torch.tensor([1, 2, 3, 4, 5])
print(my_tensor.size())
# torch.Size([5])
```

Since there are five elements in the tensor along the first axis, the size of the Tensor is [5]:

```
my_tensor = torch.tensor([[11, 12, 13], [21, 22, 23]])
print(my_tensor.size())
# torch.Size([2, 3])
```

Since the tensor of order two consists of two tensors of order one, the size of the first axis is 2 and each of the two tensors of order one consists of 3 scalar elements, where the size along the second axis is 3. Therefore, the size of the tensor is [2, 3].

This pattern scales to tensors of higher orders. Let's complete a quick exercise with the `fourth_order_tensor` that we created in the preceding subsection. There are two tensors of order three, each of which has two tensors of order one, which in turn have two tensors of order one containing two scalar elements. Therefore, the size of the tensor is [2, 2, 2, 2]:

```
print(fourth_order_tensor.size())
# torch.Size([2, 2, 2, 2])
```

Now that we understand tensor sizes, we can quickly generate tensors with random elements of our desired shape using `torch.rand()` and pass the desired tensor size as an argument to it. There are also other ways to generate tensors in PyTorch, which we'll look at later in this book. The elements that are created in your tensor may vary from what you can see here:

```
random_tensor = torch.rand([4, 2])
print(random_tensor)
#tensor([[0.9449,  0.6247],
        [0.1689,  0.4221],
        [0.9565,  0.0504],
        [0.5897,  0.9584]])
```

There will also be times when you want to reshape the tensor, that is, you would want to shift elements in the tensor to a different axis. We use the `.view()` method to reshape tensors. Let's delve into a quick example of how we can do this in PyTorch:

```
random_tensor.view([2, 4])
#tensor([[0.9449,  0.6247,  0.1689,  0.4221],
        [0.9565,  0.0504,  0.5897,  0.9584]])
```

It is important to note that this is not an in-place operation and that the original `random_tensor` is still of the size [4, 2]. You will need to assign the returned value to actually store the result. Sometimes, when you have a lot of axes, you can use -1 so that PyTorch computes the size of one particular axis:

```
random_tensor = torch.rand([4, 2, 4])
random_tensor.view([2, 4, -1])
#tensor([[[0.1751,  0.2434,  0.9390,  0.4585],
        [0.5018,  0.5252,  0.8161,  0.9712],
        [0.7042,  0.4778,  0.2127,  0.3466],
        [0.6339,  0.4634,  0.8473,  0.8062]],
       [[0.3456,  0.0725,  0.0054,  0.4665],
        [0.9140,  0.2361,  0.4009,  0.4276],
        [0.3073,  0.9668,  0.0215,  0.5560],
        [0.4939,  0.6692,  0.9476,  0.7543]]])

random_tensor.view([2, -1, 4])
#tensor([[[0.1751,  0.2434,  0.9390,  0.4585],
        [0.5018,  0.5252,  0.8161,  0.9712],
        [0.7042,  0.4778,  0.2127,  0.3466],
        [0.6339,  0.4634,  0.8473,  0.8062]],
       [[0.3456,  0.0725,  0.0054,  0.4665],
```

```
| [0.9140, 0.2361, 0.4009, 0.4276],  
| [0.3073, 0.9668, 0.0215, 0.5560],  
| [0.4939, 0.6692, 0.9476, 0.7543]]])
```

Understanding tensor operations

So far, we've looked at the basic tensor properties, but what makes them so special is their ability to perform vectorized operations, which are extremely important to performant neural networks. Let's take a quick look at a few tensor operations that are available in PyTorch.

The addition, subtraction, multiplication, and division operations are performed elementwise:

$$\mathbf{u} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \mathbf{v} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$
$$\mathbf{u} + \mathbf{v} = \begin{bmatrix} 1+0 \\ 0+1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Let's take a quick look at these operations:

```
x = torch.tensor([5, 3])
y = torch.tensor([3, 2])
torch.add(x, y)
# tensor([8, 5])
torch.sub(x, y)
# tensor([2, 1])
torch.mul(x, y)
# tensor([15, 6])
```

You can also use the `+`, `-`, `*`, and `/` operators to perform these operations on torch Tensors:

```
x + y
# tensor([8, 5])
```

Let's take a quick look at matrix multiplication in torch Tensors, which can be performed either using `torch.matmul()` or the `@` operator:

```
| torch.matmul(x, y)
| # tensor(21)
| x @ y
| # tensor(21)
```

There is a specific reason why we haven't performed the division operation on the two tensors yet. Let's do this now:

```
| torch.div(x, y)
| # tensor([1, 1])
```

What? How was that possible? $5 / 3$ should be approximately 1.667 and $3 / 2$ should be 1.5. But why did we get `tensor([1, 1])` as the result? If you guessed that it was because of the data type of the elements stored in the tensor, then you are absolutely right!

Understanding Tensor types in PyTorch

PyTorch Tensors can only store elements of a single data type in a Tensor. There are also methods defined in PyTorch that require specific data types. Therefore, it is important to understand the data types a PyTorch Tensor can store. According to the PyTorch documentation, here are the datatypes that a PyTorch Tensor can store:

Data type	dtype	tensor
32-bit floating point	<code>torch.float32</code> or <code>torch.float</code>	<code>torch.FloatTensor</code>
64-bit floating point	<code>torch.float64</code> or <code>torch.double</code>	<code>torch.DoubleTensor</code>
16-bit floating point	<code>torch.float16</code> or <code>torch.half</code>	<code>torch.HalfTensor</code>
8-bit integer (unsigned)	<code>torch.uint8</code>	<code>torch.ByteTensor</code>
8-bit integer (signed)	<code>torch.int8</code>	<code>torch.CharTensor</code>
16-bit integer (signed)	<code>torch.int16</code> or <code>torch.short</code>	<code>torch.ShortTensor</code>
32-bit integer (signed)	<code>torch.int32</code> or <code>torch.int</code>	<code>torch.IntTensor</code>
64-bit integer (signed)	<code>torch.int64</code> or <code>torch.long</code>	<code>torch.LongTensor</code>

Every PyTorch Tensor has a `dtype` attribute. Let's look at the `dtype` of the tensors we created previously:

```
| x.dtype  
| # torch.int64  
y.dtype  
# torch.int64
```

Here, we can see that the data type of the elements stored in the tensors that we created was an int64. Hence, the division that was performed between the elements was an integer division!

Let's recreate the PyTorch tensors with 32-bit floating-point elements by passing the `dtype` argument to `torch.tensor()`:

```
| x_float = torch.tensor([5, 3], dtype = torch.float32)  
y_float = torch.tensor([3, 2], dtype = torch.float32)  
print(x_float / y_float)  
# tensor([1.6667, 1.5000])
```

You can also use `torch.FloatTensor()`, or the other names under the `tensor` column in the preceding screenshot, to directly create tensors of your desired type. You can also cast tensors to other data types using the `.type()` method:

```
| torch.FloatTensor([5, 3])  
# tensor([5., 3.])  
x.type(torch.DoubleTensor)  
# tensor([5., 3.], dtype=torch.float64)
```

Importing our dataset as a PyTorch Tensor

Now, let's import our `admit_status.csv` dataset as a PyTorch Tensor so that we can feed it to our neural network. To import our dataset, we will be using the NumPy library in Python. The dataset that we will work with can be seen in the following image:

admit	gre	gpa	rank
0	380	3.61	3
1	660	3.67	3
1	800	4	1
1	640	3.19	4
0	520	2.93	4
1	760	3	2
1	560	2.98	1
0	400	3.08	2
1	540	3.39	3

When we import the dataset, we don't want to import the first row, which are the column names. We will be using `np.genfromtext()` from the NumPy library to read our data as a numpy array:

```
import numpy as np
admit_data = np.genfromtxt('../datasets/admit_status.csv',
delimiter = ',', skip_header = 1)
print(admit_data)
```

This will give us the following output:

```
[[ 0.    380.     3.61   3.    ]
 [ 1.    660.     3.67   3.    ]
 [ 1.    800.     4.     1.    ]
 ...
 [ 0.    460.     2.63   2.    ]
 [ 0.    700.     3.65   2.    ]
 [ 0.    600.     3.89   3.    ]]
```

We can directly import a numpy array as a PyTorch Tensor using

```
torch.from_numpy():
```

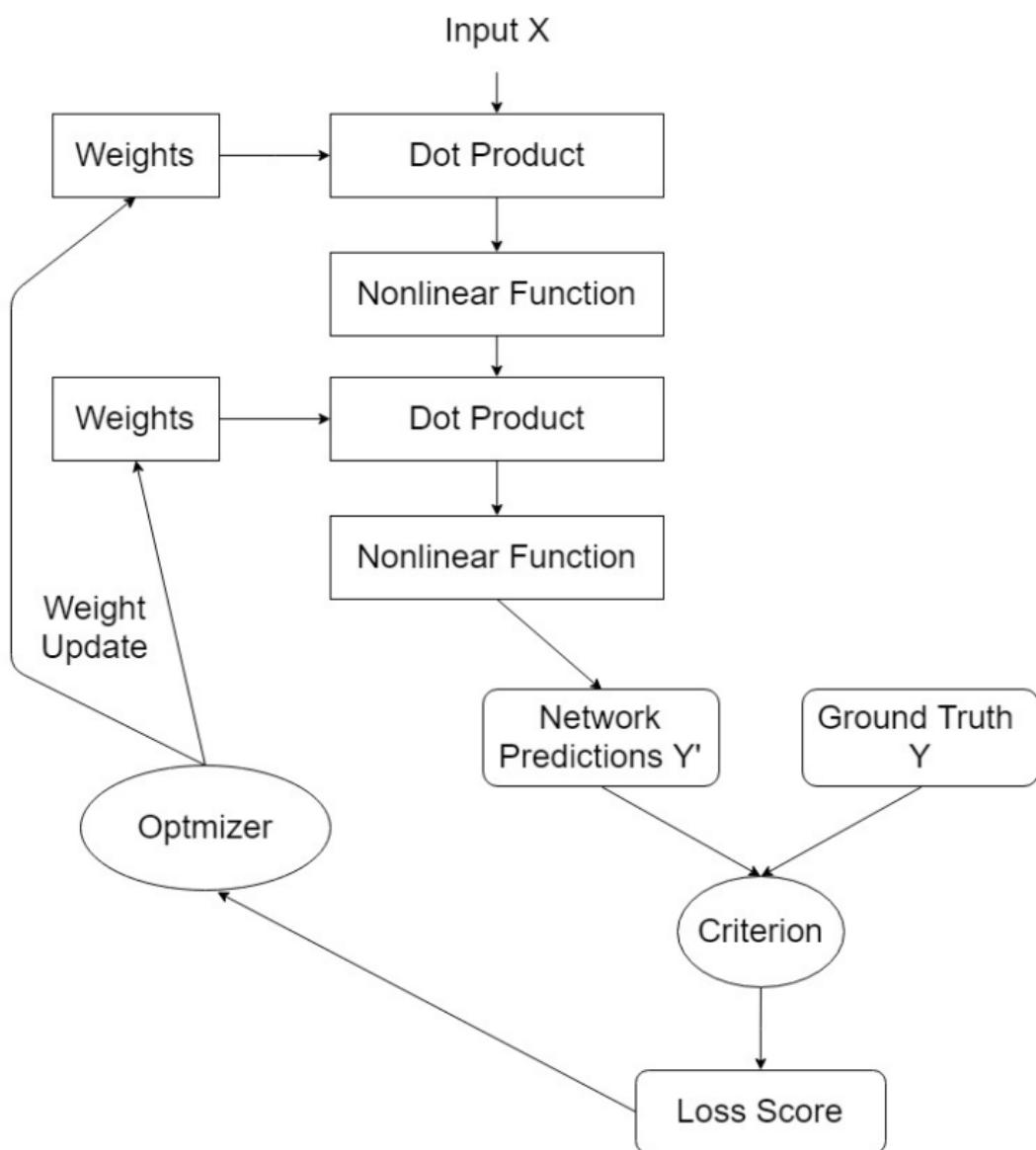
```
| admit_tensor = torch.from_numpy(admit_data)
| print(admit_tensor)
```

This will give us the following output:

```
tensor([[ 0.0000, 380.0000,  3.6100,  3.0000],
       [ 1.0000, 660.0000,  3.6700,  3.0000],
       [ 1.0000, 800.0000,  4.0000,  1.0000],
       ...,
       [ 0.0000, 460.0000,  2.6300,  2.0000],
       [ 0.0000, 700.0000,  3.6500,  2.0000],
       [ 0.0000, 600.0000,  3.8900,  3.0000]], dtype=torch.float64)
```

Training neural networks in PyTorch

We have our data as a PyTorch Tensor and we have our PyTorch neural network. Can we predict the admit status of a student now? No, not yet. First, we need to learn the specific weights that can help us predict the admit status:



The neural network we defined previously randomly generates weights at first. So, if we pass our data to the neural network directly, we would get meaningless predictions.

The two important components in a neural network that help during the training process are the **Criterion** and the **Optimizer**. The Criterion generates a loss score that is proportional to how far away the predictions from the neural network are compared to the real ground truth, that is, the target variable, which in our case is the admit status.

The Optimizer takes this score to adjust the weights in the neural network so that the predictions from the network are as close to the ground truth as possible.

This iterative process of the Optimizer using the loss score from the Criterion to update the weights of the neural network is known as the training phase of a neural network. Now, we can train our neural network.

Before we proceed and train our neural network, we have to split our dataset into inputs, x , and targets, y :

```
| x_train = admit_tensor[:300, 1:]
| y_train = admit_tensor[:300, 0]
| x_test = admit_tensor[300:, 1:]
| y_test = admit_tensor[300:, 0]
```

We need to create an instance of the criterion and the optimizer so that we can train our neural network. There are multiple criteria built into PyTorch that are accessible from the `torch.nn` module. In this case, we will be using `BCELoss()`, also known as **binary cross-entropy loss**, which is used for binary classification:

```
| criterion = nn.BCELoss()
```

There are several optimizers built into the `torch.optim` module in PyTorch. Here, we will be using the **SGD optimizer**, also known as the **stochastic gradient descent optimizer**. The optimizer takes

the parameters, or the weights, of the neural network as an argument that can be accessed by using the `parameters()` method on the instance of the neural network we created previously:

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

We have to write a for loop that iterates over the process of updating the weights. First, we need to pass our data to get predictions from the neural network. This is very simple: we just need to pass the input data as an argument to the instance of the neural network with `y_pred = my_neuralnet(x_train)`. Then, we need to compute the loss score, which we arrive at by passing the predictions from the neural network and the ground truth to the criterion. We can do this with

```
loss_score = criterion(y_pred, y_train).
```

Before we proceed to update the weights in our neural network, it is important to clear any gradients that have been accumulated from when we used the `zero_grad()` method on the optimizer. Then, to perform a backpropagation step, we use the `backward()` method on the computed `loss_score`. Finally, we update the parameters, or the weights, using the `step()` method on the optimizer.

All of the previous logic must be in a loop, where we iterate over the training process until our network learns the best parameters. So, let's put everything together into workable code:

```
for epoch in range(100):
    # Forward Propagation
    y_pred = my_network(x_train)

    # Compute and print loss
    loss_score = criterion(y_pred, y_train)
    print('epoch: ', epoch, ' loss: ', loss.item())

    # Zero the gradients
    optimizer.zero_grad()

    # perform a backward pass (backpropagation)
    loss_score.backward()

    # Update the parameters
    optimizer.step()
```

Voila! We have trained our neural network and it is ready to perform predictions. In the next chapter, we will delve into the various non-linear functions that are used in neural networks, the idea of validating what the neural network has learned, and dive deeper into the philosophy of building robust neural networks.

Summary

In this chapter, we explored various data structures and operations provided by PyTorch. We implemented several components using the fundamental blocks of PyTorch. For the data preparation stage, we created the tensors that will be used by our algorithm. Our network architecture was a model that would learn to predict the average hours spent by users on our Wondermovies platform. We used the loss function to check the standard of our model and used the `optimize` function to adjust the learnable parameters of our model to make it perform better.

We also looked at how PyTorch makes it easier for us to create data pipelines by abstracting away several complexities that would require us to parallelize and augment data.

In the next chapter, we will take a deep dive into how neural networks and deep learning algorithms work. We will explore various built-in PyTorch modules for building network architectures, loss functions, and optimizations. We will also learn how to use them on real-world datasets.

Section 2: Going Advanced with Deep Learning

In this section, you will learn how to apply neural networks to various real-world scenarios.

This section contains the following chapters:

- [Chapter 3](#), *Diving Deep into Neural Networks*
- [Chapter 4](#), *Deep Learning for Computer Vision*
- [Chapter 5](#), *Natural Language Processing with Sequence Data*

Diving Deep into Neural Networks

In this chapter, we will explore the different modules of deep learning architectures that are used to solve real-world problems. In the previous chapter, we used low-level operations of PyTorch to build modules such as a network architecture, a loss function, and an optimizer. In this chapter, we will explore some of the important components of neural networks required to solve real-world problems, along with how PyTorch abstracts away a lot of complexity by providing a lot of high-level functions. Toward the end of the chapter, we will build algorithms that solve real-world problems, such as regression, binary classification, and multi-class classification.

In this chapter, we will go through the following topics:

- Diving into the various building blocks of neural networks
- Non-linear activations
- PyTorch non-linear activations
- Image classification using deep learning

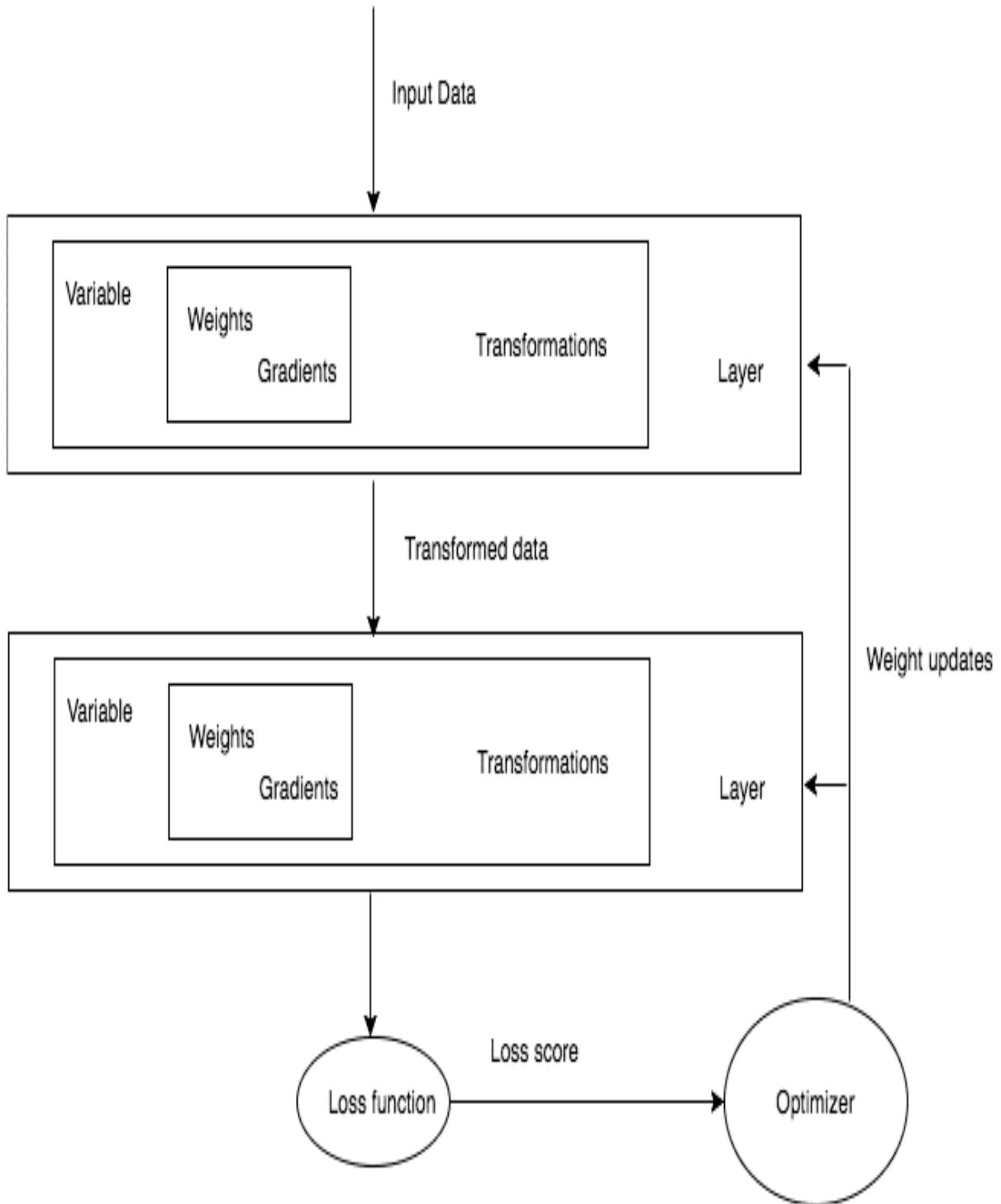
Diving into the building blocks of neural networks

As we learned in the previous chapter, training a deep learning algorithm requires the following steps:

1. Building a data pipeline
2. Building a network architecture
3. Evaluating the architecture using a loss function
4. Optimizing the network architecture weights using an optimization algorithm

In the previous chapter, the network was composed of a simple linear model built using PyTorch numerical operations. Though building a neural architecture for a dummy problem using numerical operations is easier, it quickly becomes complicated when we try to build architectures required to solve complex problems in different areas, such as computer vision and **natural language processing (NLP)**.

Most of the deep learning frameworks, such as PyTorch, TensorFlow, and Apache MXNet, provide higher-level functionalities that abstract a lot of this complexity. These higher-level functionalities are called **layers** across the deep learning frameworks. They accept input data, apply transformations like the ones we saw in the previous chapter, and output the data. To solve real-world problems, deep learning architectures consist of a number of layers ranging from 1 to 150, or sometimes more than that. Abstracting low-level operations and training deep learning algorithms would look like the following diagram:



Any deep learning training involves getting data, building an architecture (which, in general, means putting a bunch of layers together), evaluating the accuracy of the model using a loss function,

and then optimizing the algorithm by optimizing the weights of our network. Before looking at solving some real-world problems, we will come to understand higher-level abstractions provided by PyTorch for building layers, loss functions, and optimizers.

Layers – the fundamental blocks of neural networks

Throughout the rest of the chapter, we will come across different types of layers. To begin, let's try to understand one of the most important layers, the linear layer, which does exactly what our network architecture from the previous chapter does. The linear layer applies a linear transformation:

$$Y = Wx + b$$

What makes it powerful is the fact that the entire function that we wrote in the previous chapter can be written in a single line of code, as follows:

```
| from torch.nn import Linear  
| linear_layer = Linear(in_features=5,out_features=3,bias=True)
```

The `linear_layer` function, in the preceding code, will accept a tensor of size 5 and outputs a tensor of size 3 after applying linear transformation. Let's look at a simple example of how to do that:

```
| inp = Variable(torch.randn(1,5))  
| linear_layer(inp)
```

We can access the trainable parameters of the layer using the weights:

```
| Linear_layer.weight
```

This will give the following output:

Parameter containing:

```
tensor([[-0.3530, -0.3377,  0.3623, -0.4423,  0.1586],  
       [-0.1048,  0.1233, -0.0642,  0.1204, -0.2941],  
       [ 0.4393, -0.1655, -0.0738,  0.1208, -0.1913]], requires_grad=True)
```

In the same way, we can access the trainable parameters of the layer using the `bias` attributes:

```
| linear_layer.bias
```

This will give the following output:

Parameter containing:

```
tensor([-0.3194, -0.2299, -0.0302], requires_grad=True)
```

Linear layers are called by different names, such as **dense** or **fully connected** layers, across different frameworks. Deep learning architectures used for solving real-world use cases generally contain more than one layer. In PyTorch, we can do it in a simple approach by passing the output of one layer to another layer:

```
| linear_layer = Linear(5, 3)  
| linear_layer_2 = Linear(3, 2)  
| linear_layer_2(linear_layer(inp))
```

This will give the following output:

```
tensor([[0.3003, 0.4173]], grad_fn=<ThAddmmBackward>)
```

Each layer will have its own learnable parameters. The idea behind using multiple layers is that each layer will learn some kind of pattern that the later layers will build on. There is a problem with adding just linear layers together, as they fail to learn anything new beyond a simple representation of a linear layer. Let's see, through a simple

example, why it does not make sense to stack multiple linear layers together.

Let's say we have two linear layers with the following weights:

Layers	Weight1
Layer1	3.0
Layer2	2.0

The preceding architecture with two different layers can be simply represented as a single layer with a different layer. Hence, just stacking multiple linear layers will not help our algorithms to learn anything new. Sometimes, this can be unclear, so we can visualize the architecture with the following mathematical formulas:

$$Y = 2(3X_1) - 2 \text{ Linear layers}$$

$$Y = 6(X_1) - 1 \text{ Linear layers}$$

To solve this problem, we have different non-linearity functions that help in learning different relationships, rather than only focusing on linear relationships.

There are many different non-linear functions available in deep learning. PyTorch provides these non-linear functionalities as layers and we will be able to use them the same way we used the linear layer.

Some of the popular non-linear functions are as follows:

- Sigmoid
- Tanh
- ReLU
- Leaky ReLU

Non-linear activations

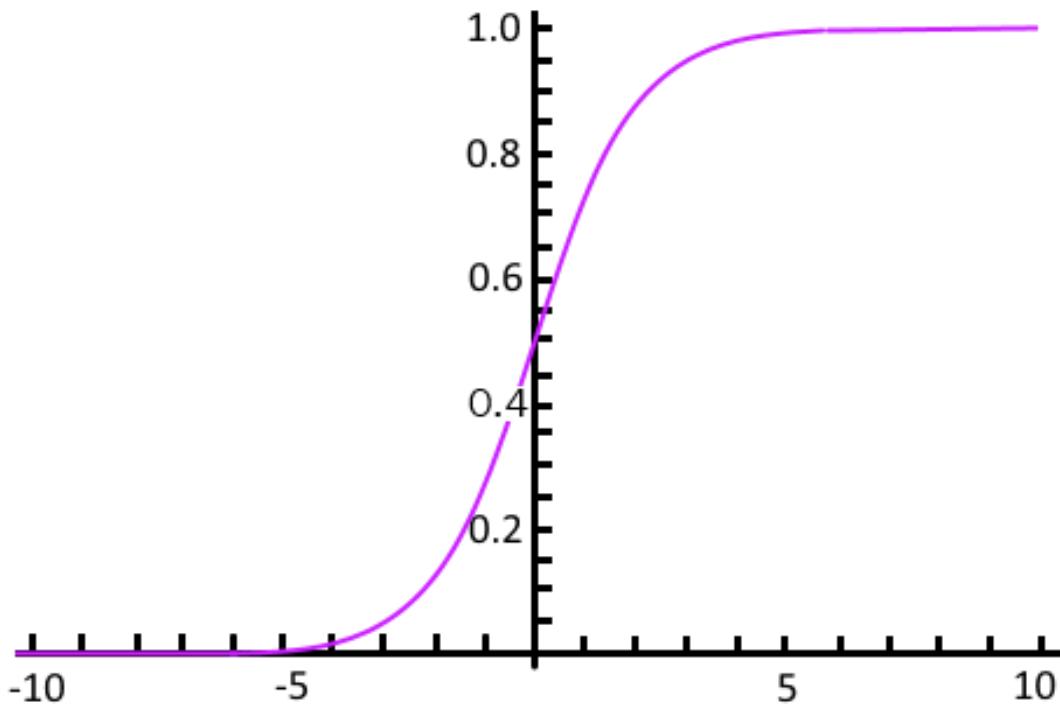
Non-linear activations are functions that take inputs and then apply a mathematical transformation and produce an output. There are several non-linear operations that we come across in practice. We will go through some of the popular non-linear activation functions.

Sigmoid

The sigmoid activation function has a simple mathematical form, as follows:

$$\sigma(x) = 1/(1 + e^{-x})$$

The sigmoid function intuitively takes a real-valued number and outputs a number in the range between 0 and 1. For a large negative number, it returns close to 0 and for a large positive number, it returns close to 1. The following plot represents different sigmoid function outputs:

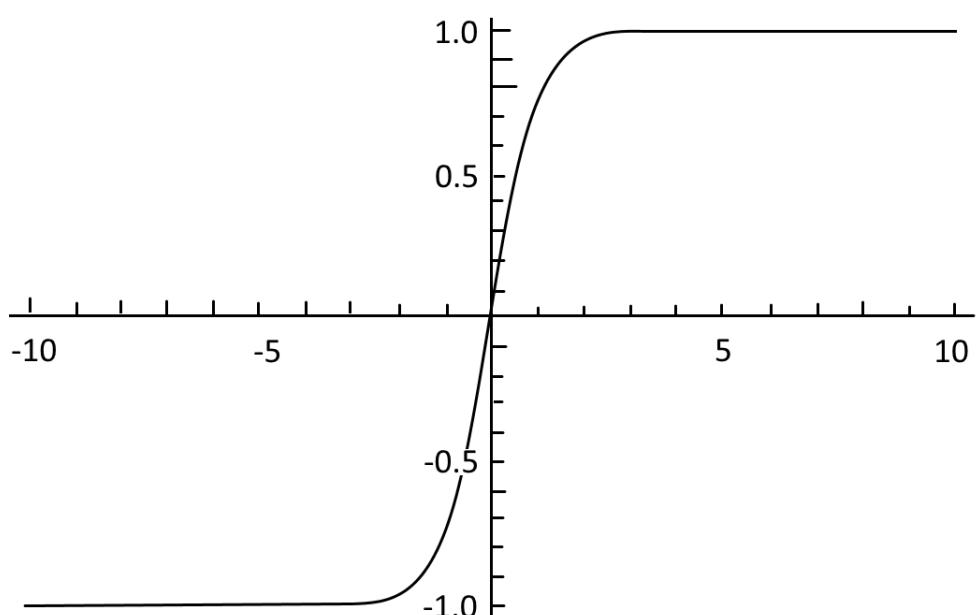


The sigmoid function has been historically used across different architectures, but in recent times, it has gone out of popularity as it has one major drawback. When the output of the sigmoid function is close to 0 or 1, the gradients for the layers before the sigmoid

function are close to 0 and, hence, the learnable parameters of the previous layer get gradients close to 0 and the weights do not get adjusted often, resulting in dead neurons.

Tanh

The tanh non-linearity function squashes a real-valued number in the range of -1 and 1. The tanh also faces the same issue of saturating gradients when tanh outputs extreme values close to -1 and 1. However, it is preferred to sigmoid, as the output of tanh is zero-centered:

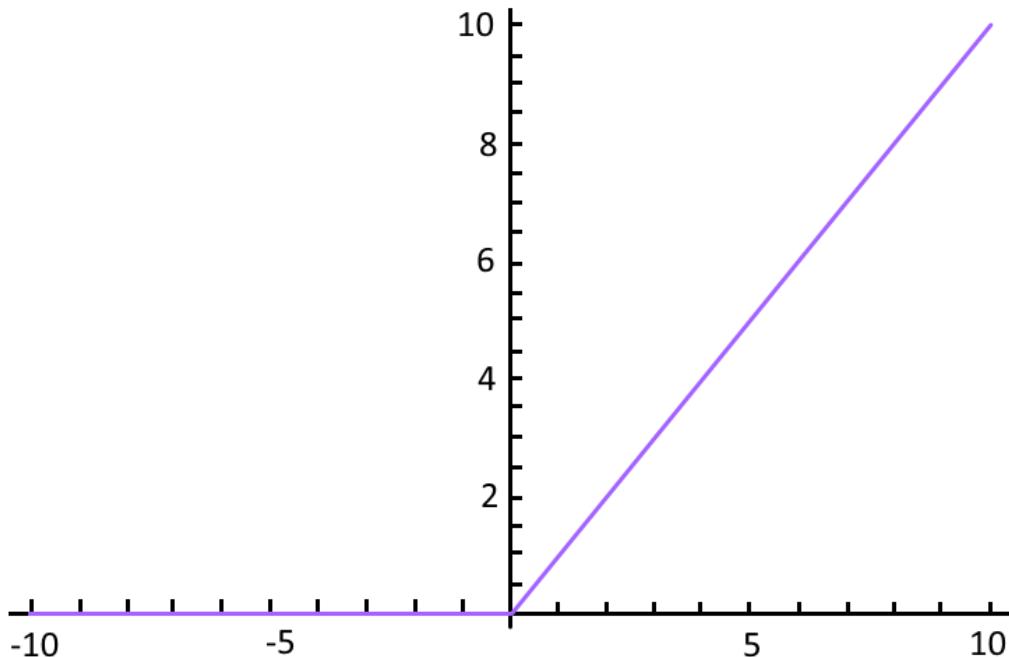


ReLU

ReLU has become more popular in recent years; we can find either its usage or one of its variants' usages in almost any modern architecture. It has a simple mathematical formulation:

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

Simply put, ReLU squashes any input that is negative to 0 and leaves positive numbers as they are. We can visualize the ReLU function as follows:



Some of the pros and cons of using ReLU are as follows:

- It helps the optimizer to find the right set of weights sooner. More technically, it makes the convergence of stochastic gradient descent faster.

- It is computationally inexpensive, as we are just thresholding and not calculating anything, as we did for the sigmoid and tangent functions.
- ReLU has one disadvantage: when a large gradient passes through it during backward propagation, it often becomes non-responsive; these are called **dead neutrons**, which can be controlled by carefully choosing the learning rate. We will discuss how to choose learning rates when we discuss the different ways to adjust the learning rate in [Chapter 4](#), *Deep Learning for Computer Vision*.

Leaky ReLU

Leaky ReLU is an attempt to solve a dying problem where, instead of saturating to 0, we saturate to a very small number such as 0.001. For some use cases, this activation function provides a superior performance to others, but it is not consistent.

PyTorch non-linear activations

PyTorch has most of the common non-linear activation functions implemented for us already and it can be used like any other layer. Let's look at a quick example of how to use the ReLU function in PyTorch:

```
| example_data = Variable(torch.Tensor([[10,2,-1,-1]]))  
| example_relu = ReLU()  
| example_relu(example_data)
```

This will result in the following output:

```
tensor([[10.,  2.,  0.,  0.]])
```

In the preceding example, we take a tensor with two positive values and two negative values and apply a ReLU on it, which thresholds the negative numbers to 0 and retains the positive numbers as they are.

Now we have covered most of the details required for building a network architecture, let's build a deep learning architecture that can be used to solve real-world problems. In the previous chapter, we used a simple approach so that we could focus only on how a deep learning algorithm works. We will not be using that style to build our architecture anymore; rather, we will be building the architecture in the way it is supposed to be built in PyTorch.

The PyTorch way of building deep learning algorithms

All networks in PyTorch are implemented as classes, subclassing a PyTorch class called `nn.Module`, and should implement the `__init__` and `forward` methods. Inside the `init` function, we initialize any layers, such as the linear layer, which we covered in the previous section. In the `forward` method, we pass our input data into the layers that we initialized in our `init` method and return our final output. The non-linear functions are often directly used in the `forward` function and some use it in the `init` method too. The following code snippet shows how a deep learning architecture is implemented in PyTorch:

```
class NeuralNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(NeuralNetwork, self).__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.layer2 = nn.Linear(hidden_size, output_size)
    def forward(self, input):
        out = self.layer1(input)
        out = nn.ReLU(out)
        out = self.layer2(out)
        return out
```

If you are new to Python, some of the preceding code could be difficult to understand, but all it is doing is inheriting a parent class and implementing two methods in it. In Python, we subclass by passing the parent class as an argument to the class name. The `init` method acts as a constructor in Python and `super` is used to pass on arguments of the child class to the parent class, which in our case is `nn.Module`.

Model architecture for different machine learning problems

The kind of problem we are solving will decide mostly what layers we will use, starting from a linear layer to **long short-term memory (LSTM)** for sequential data. Based on the type of the problem you are trying to solve, your last layer is determined. There are three problems that we generally solve using any machine learning or deep learning algorithms. Let's look at what the last layer would look like:

- For a regression problem, such as predicting the price of a t-shirt to sell, we would use the last layer as a linear layer with an output of 1, which outputs a continuous value.
- To classify a given image as a t-shirt or shirt, you would use a sigmoid activation function, as it outputs values either closer to 1 or 0, which is generally called a **binary classification problem**.
- For multi-class classification, where we have to classify whether a given image is a t-shirt, a jeans, a shirt, or a dress, we would use a softmax layer at the end of our network. Let's try to understand intuitively what softmax does without going into the math of it. It takes inputs from the previous linear layer, for example, and outputs the probabilities for a given number of examples. In our example, it would be trained to predict four probabilities for each type of image. Remember, all these probabilities always add up to 1.

Loss functions

Once we have defined our network architecture, we are left with two important steps. One is calculating how good our network is at performing a particular task of regression, classification, and the next is optimizing the weight.

The optimizer (gradient descent) generally accepts a scalar value, so our loss function should generate a scalar value that has to be minimized during our training. Certain use cases, such as predicting where an obstacle is on the road and classifying it as a pedestrian or not, would require two or more loss functions. Even in such scenarios, we need to combine the losses into a single scalar for the optimizer to minimize. We will discuss examples of combining multiple losses into a single scalar in detail with a real-world example in [Chapter 8](#), *Transfer Learning with Modern Network Architectures*.

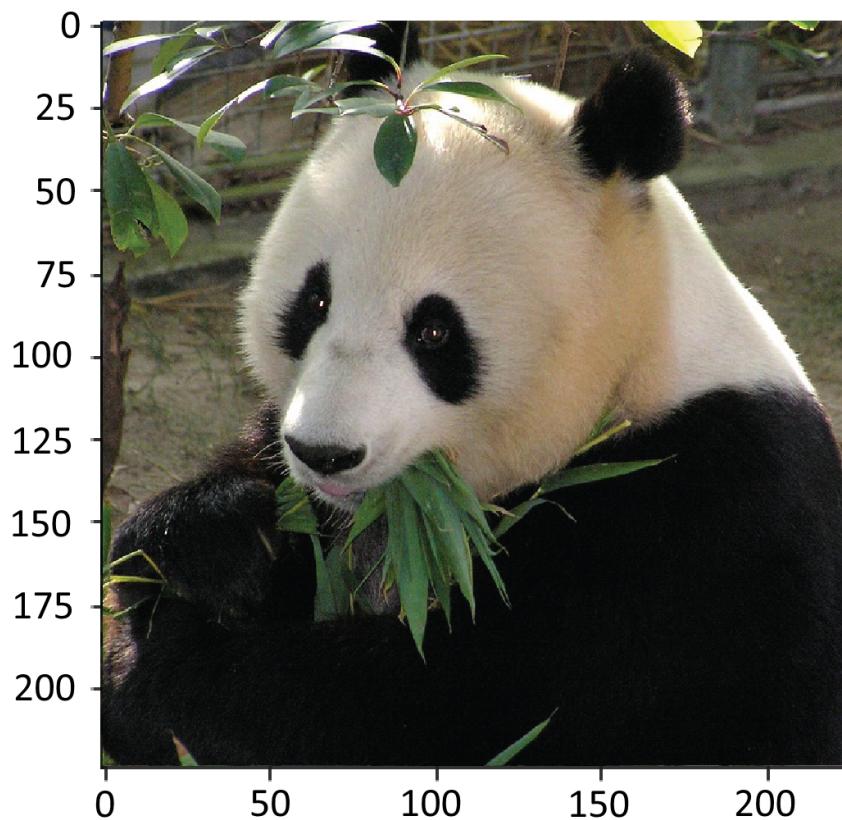
In the previous chapter, we defined our own loss function. PyTorch provides several implementations of commonly used loss functions. Let's take a look at the loss functions used for regression and classification.

The commonly used loss function for regression problems is **mean square error (MSE)**. It is the same loss function we implemented in our previous chapter. We can use the loss function implemented in PyTorch, as follows:

```
loss = nn.MSELoss()  
input = Variable(torch.randn(2, 6), requires_grad=True)  
target = Variable(torch.randn(2, 6))  
output = loss(input, target)  
output.backward()
```

For classification, we use cross-entropy loss. Before looking at the math for cross-entropy, let's understand what cross-entropy loss does. It calculates the loss of a classification network, predicting the

probabilities, which should sum up to 1, like our softmax layer. Cross-entropy loss increases when the predicted probability diverges from the correct probability. For example, if our classification algorithm predicts 0.1 probability of the following image being a cat, but it is actually a panda, then the cross-entropy loss will be higher. If it predicts similar to the actual labels, then the cross-entropy loss will be lower:



Let's look at a sample implementation of how this actually happens in Python code:

```
def cross_entropy_function(true_label, prediction):
    if true_label == 1:
        return -log(prediction)
    else:
        return -log(1 - prediction)
```

To use cross-entropy loss in a classification problem, we really do not need to be worried about what happens inside—all we have to

remember is that the loss will be high when our predictions are bad and low when predictions are good. PyTorch provides us with an implementation of the loss, which we can use, as follows:

```
loss = nn.CrossEntropyLoss()  
input = Variable(torch.randn(2, 6), requires_grad=True)  
target = Variable(torch.LongTensor(2).random_(6))  
output = loss(input, target)  
output.backward()
```

Some of the other loss functions that come as part of PyTorch are as follows:

L1 loss	Mostly used as a regularizer; we will discuss it further in Chapter 4 , <i>Deep Learning for Computer Vision</i>
MSE loss	Used as a loss function for regression problems
Cross-entropy loss	Used for binary and multi-class classification problems
NLL Loss	Used for classification problems and allows us to use specific weights to handle imbalanced datasets
NLL Loss2d	Used for pixel-wise classification, mostly for problems related to image segmentation

Optimizing network architecture

Once we have calculated the loss of our network, we will optimize the weights to reduce the loss, thus improving the accuracy of the algorithm. For the sake of simplicity, let's see these optimizers as black boxes that take loss functions and all the learnable parameters and move them slightly to improve our performance. PyTorch provides most of the commonly used optimizers required in deep learning. If you want to explore what happens inside these optimizers and have a mathematical background, I would strongly recommend the following blogs:

- <http://ruder.io/optimizing-gradient-descent/>
- <https://medium.com/datadriveninvestor/overview-of-different-optimizers-for-neural-networks-e0ed119440c3>

Some of the optimizers that PyTorch provides are as follows:

- ASGD
- Adadelta
- Adagrad
- Adam
- Adamax
- LBFGS
- RMSprop
- Rprop
- SGD
- SparseAdam

We will get into the details of some of the algorithms in [Chapter 4](#), *Deep Learning for Computer Vision*, along with some of the

advantages and trade-offs. Let's walk through some of the important steps in creating any optimizer:

```
| sgd_optimizer = optim.SGD(model.parameters(), lr = 0.01)
```

In the preceding example, we created an SGD optimizer that takes all the learnable parameters of your network as the first argument and a learning rate that determines what ratio of change can be made to the learnable parameters. In [Chapter 4](#), *Deep Learning for Computer Vision*, we will get into more details of learning rates and momentum, which is an important parameter of optimizers. Once you create an optimizer object, we need to call `zero_grad()` inside our loop, as the parameters will accumulate the gradients created during the previous optimizer call:

```
| for input, target in dataset:  
|     sgd_optimizer.zero_grad()  
|     output = model(input)  
|     loss = loss_fn(output, target)  
|     loss.backward()  
|     sgd_optimizer.step()
```

Once we call `backward` on the loss function, which calculates the gradients (the quantity by which learnable parameters need to change), we call `optimizer.step()`, which makes the actual changes to our learnable parameter.

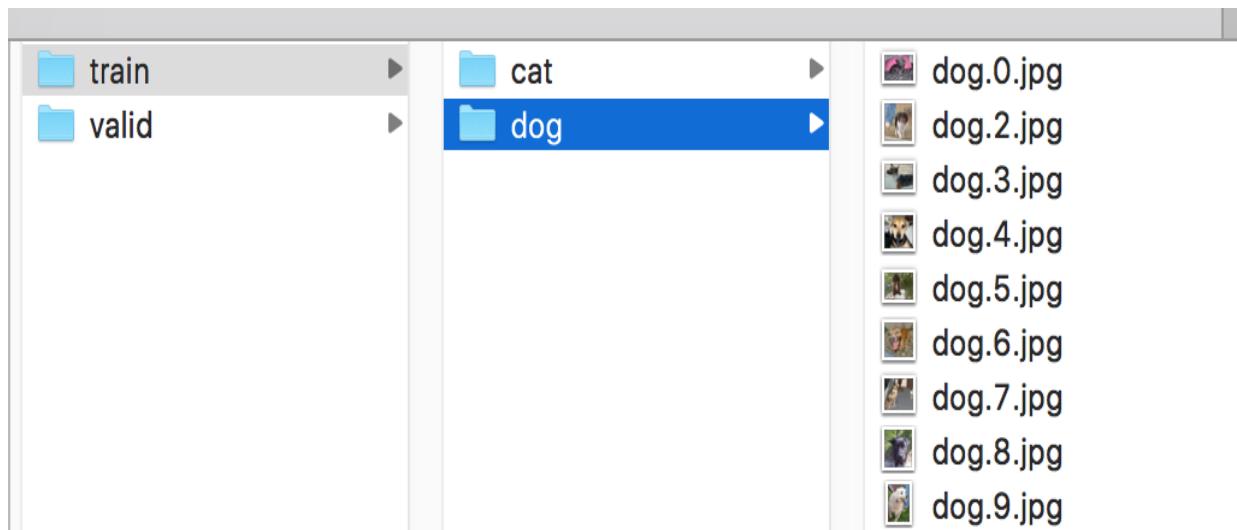
Now we have covered most of the components required to help a computer see or recognize images. Let's build a complex deep learning model that can differentiate between dogs and cats to put all the theory into practice.

Image classification using deep learning

The most important step in solving any real-world problem is getting the data. In order to test our deep learning algorithms in this chapter, we will use a dataset provided in a GitHub repository from an user called `ardamavi`. We will use this dataset again in [Chapter 4](#), *Deep Learning for Computer Vision*, which will be on **convolution neural networks (CNNs)** and some of the advanced techniques that we can use to improve the performance of our image recognition models.

You can download the data from the following link: https://github.com/ardamavi/Dog-Cat-Classifier/tree/master/Data/Train_Data. The dataset contains images of dogs and cats. The preprocessing of data and the creation of training, validation, and testing splits are some of the important steps that need to be performed before we can implement an algorithm.

Most frameworks make it easier to read images and tag them to their labels when provided in the following format. That means that each class should have a separate folder of its images. Here, all cat images should be in the `cat` folder and dog images in the `dog` folder:



Python makes it easy to put the data into the right format. Let's quickly take a look at the code and then we will go through the important parts of it:

```
path = 'Dog-Cat-Classifier/Data/Train_Data/'  
#Read all the files inside our folder.  
dog_files = [f for f in glob.glob('Dog-Cat-  
Classifier/Data/Train_Data/dog/*.jpg')]  
cat_files = [f for f in glob.glob('Dog-Cat-  
Classifier/Data/Train_Data/cat/*.jpg')]  
files = dog_files + cat_files  
print(f'Total no of images {len(files)}')  
no_of_images = len(files)
```

Create a shuffled index that can be used to create a validation dataset:

```
| shuffle = np.random.permutation(no_of_images)
```

Create a validation directory for holding training and validation images:

```
os.mkdir(os.path.join(path,'train'))  
os.mkdir(os.path.join(path,'valid'))  
Create directories with label names.  
for t in ['train','valid']:  
    for folder in ['dog/','cat/']:  
        os.mkdir(os.path.join(path,t,folder))
```

Copy a small subset of images into the validation folder:

```
| for i in shuffle[:250]:  
|     folder = files[i].split('/')[-2].split('.')[0]  
|     image = files[i].split('/')[-1]  
|     os.rename(files[i],os.path.join(path,'valid',folder,image))
```

Copy a small subset of images into the training folder:

```
| for i in shuffle[250:]:  
|     folder = files[i].split('/')[-2].split('.')[0]  
|     image = files[i].split('/')[-1]  
|     os.rename(files[i],os.path.join(path,'train',folder,image))
```

All the preceding code does is retrieve all the files and pick a sample of images for creating a testing and validation set. It segregates all the images into the two categories of cats and dogs. It is a common and important practice to create a separate validation set, as it is not fair to test our algorithms on the same data it is trained on. To create a dataset, we create a list of numbers that are in the range of the length of the images in a shuffled order. The shuffled numbers act as an index for us to pick a bunch of images to create our dataset. Let's go through each section of the code in detail.

We use the `glob` method to return all the files in the particular path:

```
| dog_files = [f for f in glob.glob('Dog-Cat-  
| Classifier/Data/Train_Data/dog/*.jpg')]  
| cat_files = [f for f in glob.glob('Dog-Cat-  
| Classifier/Data/Train_Data/cat/*.jpg')]
```

When there are a huge number of images, we can also use `iglob`, which returns an iterator, instead of loading the names into memory. In our case, the volume of images we are working with is low and we can easily fit them into memory so it is not necessary.

We can shuffle our files using the following code:

```
| shuffle = np.random.permutation(no_of_images)
```

The preceding code returns numbers in the range of 0 to 1,399 in a shuffled order, which we will use as an index for selecting a subset of images to create a dataset.

We can create testing and validation code, as follows:

```
os.mkdir(os.path.join(path,'train'))
os.mkdir(os.path.join(path,'valid'))
for t in ['train','valid']:
    for folder in ['dog/','cat/']:
        os.mkdir(os.path.join(path,t,folder))
```

The preceding code creates folders based on categories (cats and dogs) inside the `train` and `valid` directories.

We can shuffle an index with the following code:

```
for i in shuffle[:250]:
    folder = files[i].split('/')[-2].split('.')[0]
    image = files[i].split('/')[-1]
    os.rename(files[i],os.path.join(path,'valid',folder,image))
```

In the preceding code, we use our shuffled index to randomly pick 250 different images for our validation set. We do something similar for the training data to segregate the images in the `train` directory.

As we have the data in the format we need, let's quickly look at how to load the images as PyTorch tensors.

Loading data into PyTorch tensors

The PyTorch `torchvision.datasets` package provides a utility class called `ImageFolder` that can be used to load images along with their associated labels when data is presented in the aforementioned format. It is a common practice to perform the following preprocessing steps:

1. Resize all the images to the same size. Most deep learning architectures expect the images to be of the same size.
2. Normalize the dataset with the mean and standard deviation of the dataset.
3. Convert the image dataset to a PyTorch tensor.

PyTorch makes a lot of these preprocessing steps easier by providing a lot of utility functions in the `transforms` module. For our example, let's apply three transformations:

- Scale to a 256 x 256 image size
- Convert to a PyTorch tensor
- Normalize the data (we will talk about how we arrived at the mean and standard deviation in the next section)

The following code demonstrates how transformations can be applied and images are loaded using the `ImageFolder` class:

```
transform = transforms.Compose([transforms.Resize((224,224))
                               ,transforms.ToTensor()
                               ,transforms.Normalize([0.12, 0.11,
0.40], [0.89, 0.21, 0.12])])
train = ImageFolder('Dog-Cat-
Classifier/Data/Train_Data/train/',transform)
valid = ImageFolder('Dog-Cat-
Classifier/Data/Train_Data/valid/',transform)
```

The `train` object holds all the images and associated labels for the dataset. It contains two important attributes: one that gives a

mapping between classes and the associated index used in the dataset and another one that gives a list of classes:

- `train.class_to_idx = {'cat': 0, 'dog': 1}`
- `train.classes = ['cat', 'dog']`

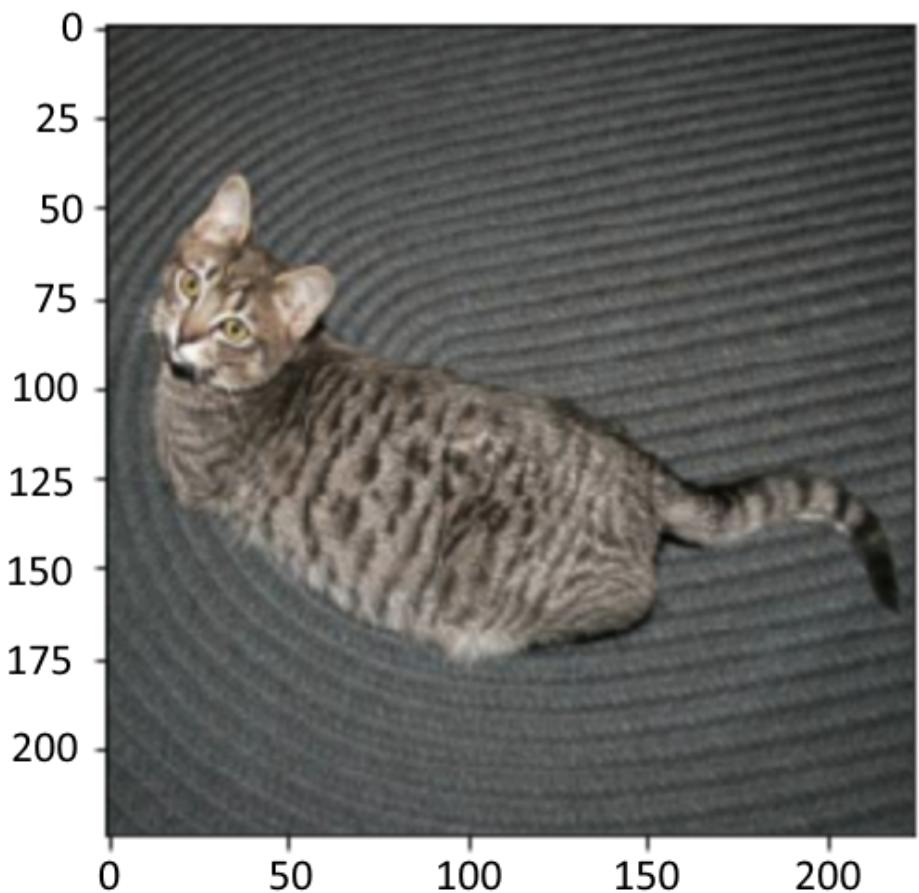
It is often a best practice to visualize the data loaded into tensors. To visualize the tensors, we have to reshape the tensors and denormalize the values. The following function does that for us:

```
import matplotlib.pyplot as plt
def imshow(inp):
    """Imshow for Tensor."""
    inp = inp.numpy().transpose((1, 2, 0))
    mean = np.array([0.12, 0.12, 0.40])
    std = np.array([0.22, 0.20, 0.20])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    plt.imshow(inp)
```

Now we can pass our tensor to the preceding `imshow` function, which converts it into an image:

```
| imshow(train[30][0])
```

The preceding code generates the following output:



Loading PyTorch tensors as batches

It is a common practice in deep learning or machine learning to batch samples of images, as modern **graphics processing units (GPUs)** and CPUs are optimized to run operations faster on a batch of images. The batch size generally varies depending on the kind of GPU we use. Each GPU has its own memory, which can vary from 2 GB to 12 GB, and sometimes more for commercial GPUs. PyTorch provides the `DataLoader` class, which takes in a dataset and returns a batch of images. It abstracts a lot of complexities in batching, such as the usage of multi-workers for applying transformation. The following code converts the previous `train` and `valid` datasets into data loaders:

```
train_data_generator =  
    torch.utils.data.DataLoader(train, shuffle=True, batch_size=64, num_workers=  
        8)  
valid_data_generator =  
    torch.utils.data.DataLoader(valid, batch_size=64, num_workers=8)
```

The `DataLoader` class provides us with a lot of options and some of the most commonly used ones are as follows:

- `shuffle`: When true, this shuffles the images every time the data loader is called.
- `num_workers`: This is responsible for parallelization. It is common practice to use a number of workers fewer than the number of cores available in your machine.

Building the network architecture

For most real-world use cases, particularly in computer vision, we rarely build our own architecture. There are different architectures that can be quickly used to solve our real-world problems. For our example, we'll use a popular deep learning algorithm called **ResNet**, which won the first prize in 2015 in different competitions, such as ImageNet, related to computer vision.

For a simpler understanding, let's assume that this algorithm is a bunch of different PyTorch layers carefully tied together and not focus on what happens inside this algorithm. We will see some of the key building blocks of the ResNet algorithm when we learn about CNNs. PyTorch makes it easier to use a lot of these popular algorithms by providing them off the shelf in the `torchvision.models` module. So, for this example, let's quickly take a look at how to use this algorithm and then walk through each line of code:

```
pretrained_resnet = models.resnet18(pretrained=True)
number_features = pretrained_resnet.fc.in_features
pretrained_resnet.fc = nn.Linear(number_features, 4)
```

The `models.resnet18(pretrained = True)` object creates an instance of the algorithm, which is a collection of PyTorch layers. We can take a quick look at what constitutes the ResNet algorithm by printing `pretrained_resnet`. A small portion of the algorithm looks like the following screenshot (I am not including the full algorithm as it could run for several pages):

```

ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
  )
)

```

As we can see, the ResNet architecture is a collection of layers, namely `Conv2d`, `BatchNorm2d`, and `MaxPool2d`, stitched in a particular way. All these algorithms will accept an argument called `pretrained`. When `pretrained` is `True`, the weights of the algorithm are already tuned for a

particular ImageNet classification problem of predicting 1,000 different categories, which include cars, ships, fish, cats, and dogs. This algorithm is trained to predict the 1,000 ImageNet categories and the weights are adjusted to a certain point where the algorithm achieves state-of-the-art accuracy. These weights are stored and shared with the model that we are using for the use case. Algorithms tend to work better when started with fine-tuned weights, rather than when started with random weights. So, for our use case, we'll start with pretrained weights.

The ResNet algorithm cannot be used directly, as it is trained to predict one of the 1,000 categories. For our use case, we need to predict only one of the two categories of dogs and cats. To achieve this, we take the last layer of the ResNet model, which is a linear layer, and change the output features to 4, as shown in the following code:

```
| pretrained_resnet.fc = nn.Linear(number_features, 4)
```

If you are running this algorithm on a GPU-based machine, then to make the algorithm run on a GPU, we call the `cuda` method on the model. It is strongly recommended that you run these programs on a GPU-powered machine; it is easy to spin a cloud instance with a GPU for less than a dollar. The last line in the following code snippet tells PyTorch to run the code on the GPU:

```
| if is_cuda:  
|     pretrained_resnet = pretrained_resnet.cuda()
```

Training the model

In the previous sections, we created some `DataLoader` instances and algorithms. Now let's train the model. To do this, we need a loss function and an optimizer:

```
learning_rate = 0.005
criterion = nn.CrossEntropyLoss()
fit_optimizer = optim.SGD(pretrained_resnet.parameters(), lr=0.005,
momentum=0.6)
exp_learning_rate_scheduler = lr_scheduler.StepLR(fit_optimizer,
step_size=2, gamma=0.05)
```

In the preceding code, we created our loss function based on `CrossEntropyLoss` and the optimizer based on `SGD`. The `StepLR` function helps in dynamically changing the learning rate. We will discuss different strategies available to tune the learning rate in [Chapter 4, Deep Learning for Computer Vision](#).

The following `train_my_model` function takes in a model and tunes the weights of our algorithm by running multiple epochs and reducing the loss:

```
def train_my_model(model, criterion, optimizer, scheduler,
number_epochs=20):
    since = time.time()
    best_model_weights = model.state_dict()
    best_accuracy = 0.0
    for epoch in range(number_epochs):
        print('Epoch {}/{}'.format(epoch, number_epochs - 1))
        print('-' * 10)
```

Each epoch has a training and validation phase:

```
for each_phase in ['train', 'valid']:
    if each_phase == 'train':
        scheduler.step()
        model.train(True)
    else:
        model.train(False)

    running_loss = 0.0
    running_corrects = 0
```

Iterate over the data:

```
for data in dataloaders[each_phase]:
    input_data, label_data = data
    if torch.cuda.is_available():
        input_data = Variable(inputs.cuda())
        label_data = Variable(labels.cuda())
    else:
        input_data, label_data = Variable(input_data),
Variable(label_data)
    optimizer.zero_grad()
    outputs = model(input_data)
    _, preds = torch.max(outputs.data, 1)
    loss = criterion(outputs, label_data)
    if each_phase == 'train':
        loss.backward()
        optimizer.step()
        running_loss += loss.data[0]
        running_corrects += torch.sum(preds == label_data.data)
    epoch_loss = running_loss / dataset_sizes[each_phase]
    epoch_acc = running_corrects / dataset_sizes[each_phase]
    print('{}) Loss: {:.4f} Acc: {:.4f}'.format(each_phase,
epoch_loss, epoch_acc))
    if each_phase == 'valid' and epoch_acc > best_acc:
        best_accuracy = epoch_acc
        best_model_weights = model.state_dict()
    print()
time_elapsed = time.time() - since
print('Training complete in {:.0f}m {:.0f}s'.format(time_elapsed // 60, time_elapsed % 60))
print('Best val Acc: {:.4f}'.format(best_accuracy))
model.load_state_dict(best_model_weights)
return model
```

The function can be run as follows:

```
train_my_model(pretrained_resnet, criterion, fit_optimizer,
exp_learning_rate_scheduler, number_epochs=20)
```

The preceding function does the following:

- It passes the images through the model and calculates the loss.
- It backpropagates during the training phase. For the validation/testing phase, it does not adjust the weights.
- The loss is accumulated across batches for each epoch.
- The best model is stored and validation accuracy is printed.

The preceding model, after running for 20 epochs, results in a validation accuracy of 87%.

In the coming chapters, we will learn more advanced techniques that will help us in training more accurate models in a much faster way. The preceding model took around 30 minutes to run on a Titan X GPU. We will cover different techniques that will help in training the model faster.

Summary

In this chapter, we explored the complete life cycle of a neural network in PyTorch, starting from constituting different types of layers, adding activations, calculating cross-entropy loss, and finally, optimizing network performance (that is, minimizing loss), by adjusting the weights of layers using the SGD optimizer.

We studied how to apply the popular ResNet architecture to binary or multi-class classification problems.

While doing this, we tried to solve the real-world image classification problem of classifying a cat image as a cat and a dog image as a dog. This knowledge can be applied to classify different categories/classes of entities, such as classifying species of fish, identifying different kinds of dogs, categorizing plant seedlings, grouping together cervical cancer into Type 1, Type 2, and Type 3, and much more.

In the next chapter, we will go through the fundamentals of machine learning.

Deep Learning for Computer Vision

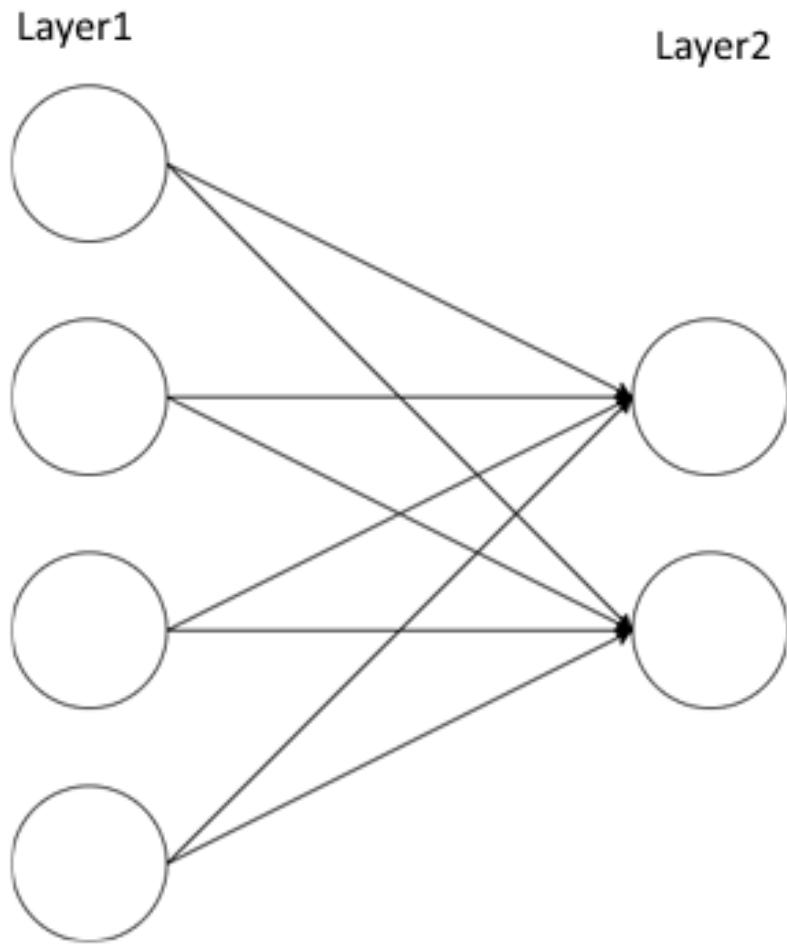
In [Chapter 3](#), *Diving Deep into Neural Networks*, we built an image classifier using a popular **convolutional neural network (CNN)** architecture called **ResNet**, but we used this model as a black box. In this chapter, we will explore how we can build an architecture from scratch to solve image classification problems, which are the most common use cases. We will also learn how to use transfer learning, which will help us build image classifiers using a very small dataset. Apart from learning how to use CNNs, we will also explore what these convolutional networks learn.

In this chapter, we will cover the important building blocks of convolutional networks. Some of the important topics that we will be covering in this chapter are as follows:

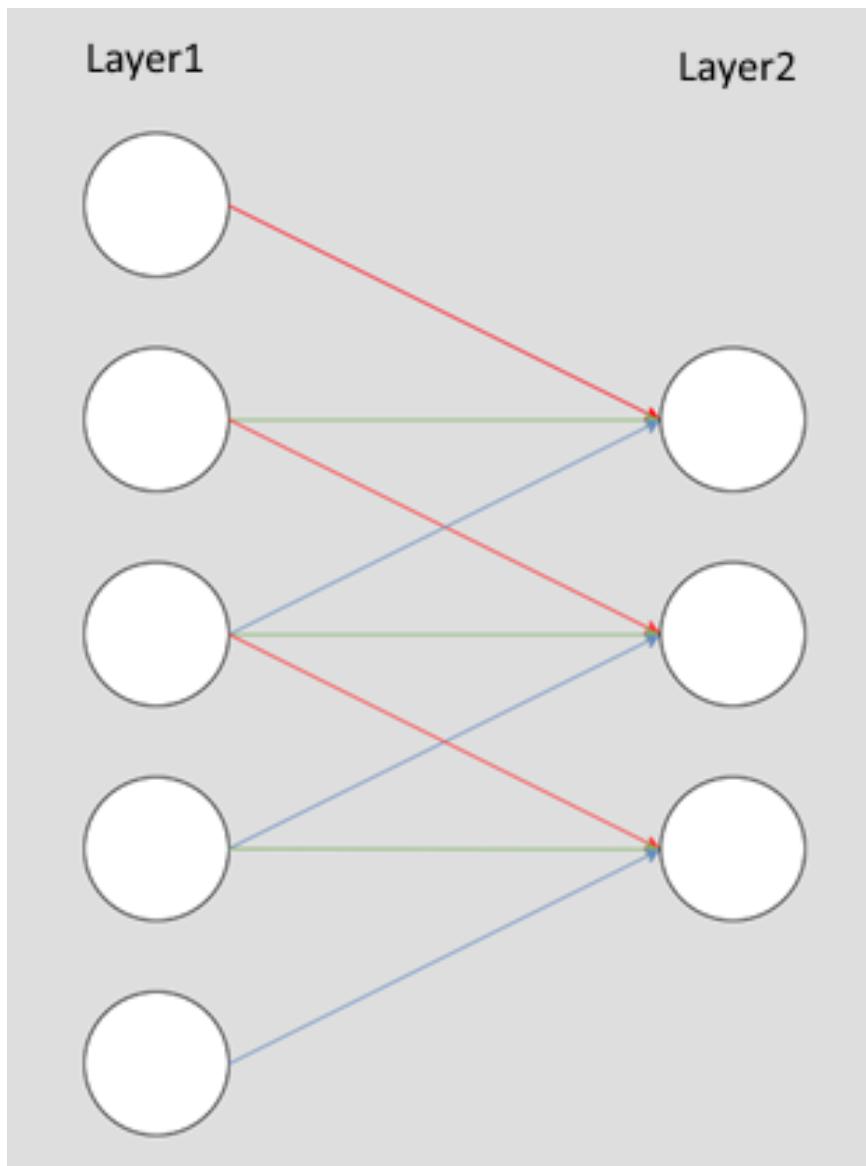
- Introduction to neural networks
- Building a CNN model from scratch
- Creating and exploring a VGG16 model
- Calculating pre-convoluted features
- Understanding what a CNN model learns
- Visualizing the weights of the CNN layer

Introduction to neural networks

In the last few years, CNNs have become popular in image recognition, object detection, segmentation, and many other areas in the field of computer vision. They are also becoming popular in the field of **natural language processing (NLP)**, though they are not commonly used yet. The fundamental difference between fully connected layers and convolution layers is the way the weights are connected to each other in the intermediate layers. Let's take a look at the following diagram, which shows how fully connected, or linear, layers work:



One of the biggest challenges of using a linear layer or fully connected layers for computer vision is that they lose all spatial information, and the complexity in terms of the number of weights that are used by fully connected layers is too big. For example, when we represent a 224-pixel image as a flat array, we would end up with 150,528 ($224 \times 224 \times 3$ channels). When the image is flattened, we lose all the spatial information. Let's look at what a simplified version of a CNN looks like:



All the convolution layer is doing is applying a window of weights called **filters** across the image. Before we try to understand convolutions and other building blocks in detail, let's build a simple yet powerful image classifier for the MNIST dataset. Once we've built this, we will walk through each component of the network. We will break down building our image classifier into the following steps:

1. Getting data
2. Creating a validation dataset
3. Building our CNN model from scratch
4. Training and validating the model

MNIST – getting data

The MNIST dataset contains 60,000 handwritten digits from 0 to 9 for training and 10,000 images for a test set. The PyTorch `torchvision` library provides us with an MNIST dataset, which downloads the data and provides it in a readily usable format. Let's use the dataset `MNIST` function to pull the dataset to our local machine and then wrap it around `DataLoader`. We will use `torchvision` transformations to convert the data into PyTorch tensors and do data normalization. The following code takes care of downloading, wrapping data around `DataLoader`, and normalizing the data:

```
transformation = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.14,), (0.32,))])
training_dataset =
datasets.MNIST('dataset/',train=True,transform=transformation,
download=True) test_dataset =
datasets.MNIST('dataset/',train=False,transform=transformation,
download=True)
training_loader =
torch.utils.data.DataLoader(training_dataset,batch_size=32,shuffle=True)
test_loader =
torch.utils.data.DataLoader(test_dataset,batch_size=32,shuffle=True)
```

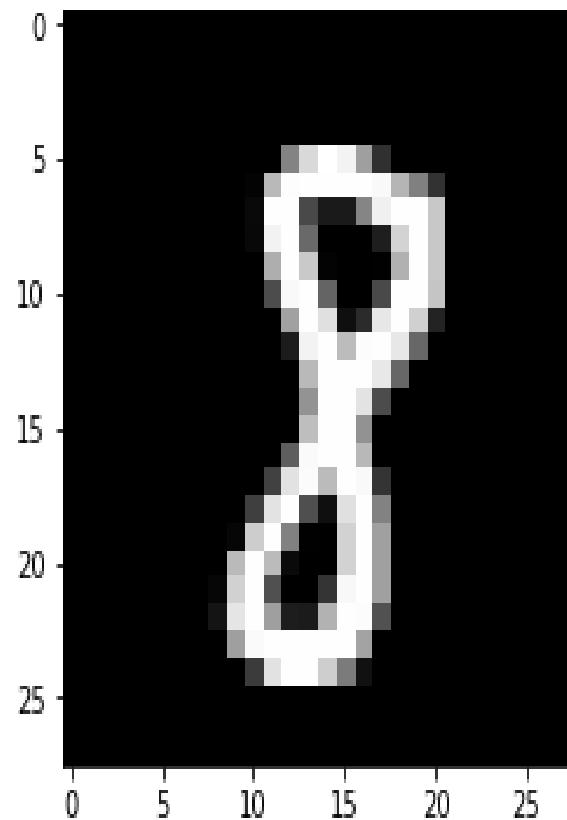
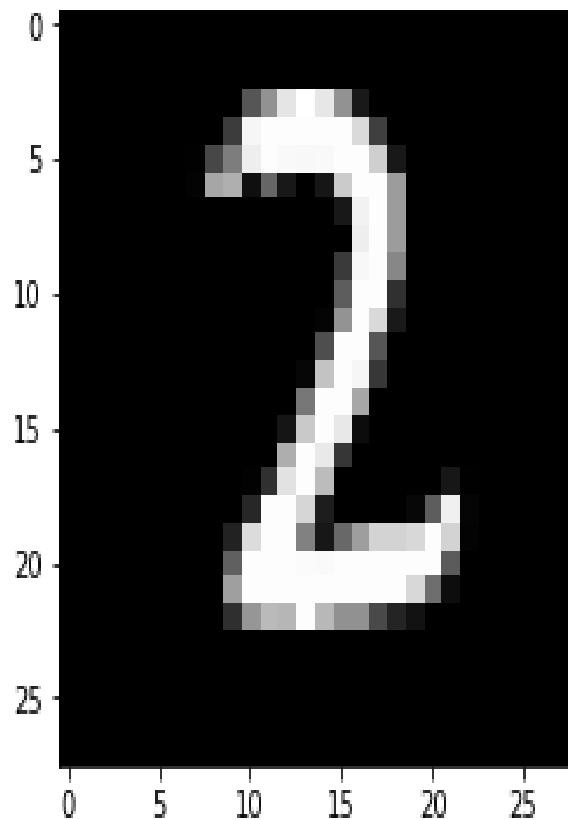
The previous code provides us with a `DataLoader` variable for the train and test datasets. Let's visualize a few images to get an understanding of what we are dealing with. The following code will help us visualize the MNIST images:

```
def plot_img(image):
image = image.numpy()[0] mean = 0.1307
std = 0.3081
image = ((mean * image) + std) plt.imshow(image,cmap='gray')
```

Now, we can pass the `plot_img` method to visualize our dataset. We will pull a batch of records from the `DataLoader` variable using the following code and plot the images:

```
sample_data = next(iter(training_loader)) plot_img(sample_data[0][1])
plot_img(sample_data[0][2])
```

The images can be visualized as follows:

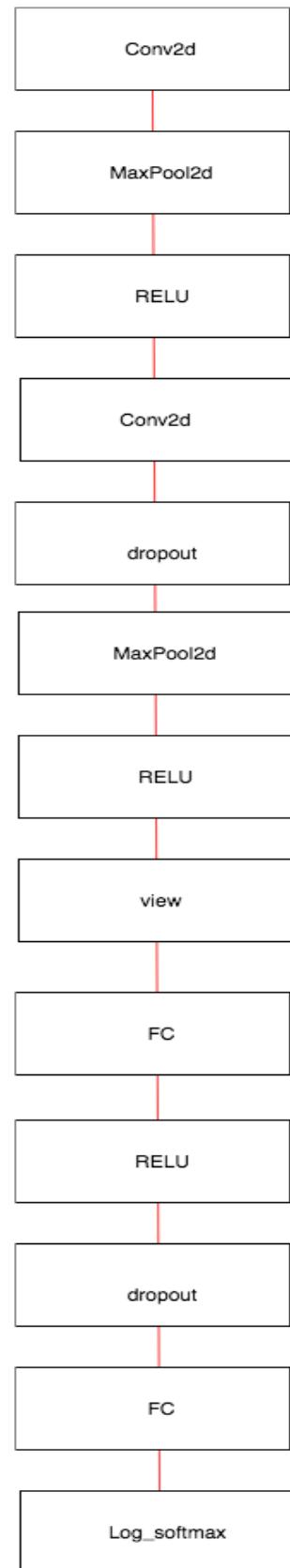


Building a CNN model from scratch

In this section, we'll build our own architecture from scratch. Our network architecture will contain a combination of different layers, as follows:

- Conv2d
- MaxPool2d
- **Rectified linear unit (ReLU)**
- View
- Linear layer

Let's look at a pictorial representation of the architecture we are going to implement:



Let's implement this architecture in PyTorch and then walk through what each individual layer does:

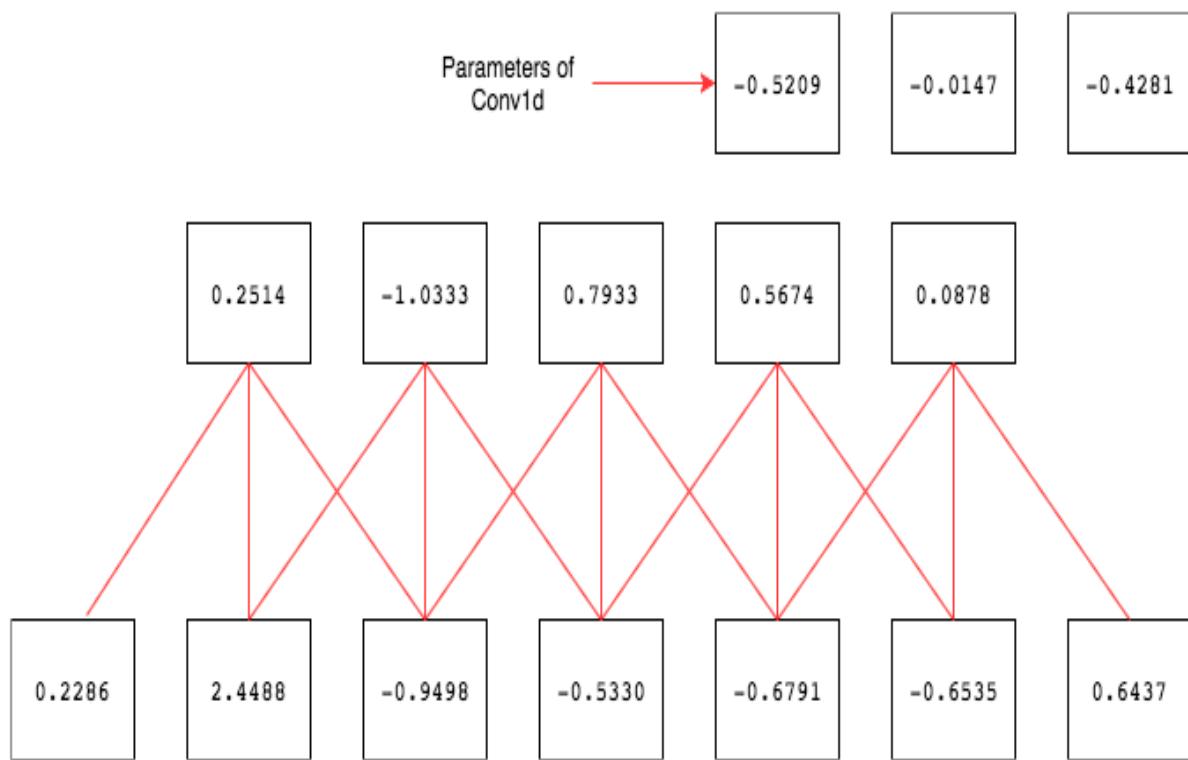
```
class Network(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=3)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=3)
        self.conv2_drop = nn.Dropout2d()
        self.fullyconnected1 = nn.Linear(320, 50)
        self.fullyconnected2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fullyconnected1(x))
        x = F.dropout(x, training=self.training)
        x = self.fullyconnected2(x)
        return F.log_softmax(x)
```

Let's understand what each layer does in detail.

Conv2d

Conv2d takes care of applying a convolutional filter on our MNIST images. Let's try to understand how convolution is applied on a one-dimensional array and then learn how a two-dimensional convolution is applied to an image. Take a look at the following diagram. Here, we will apply a **Conv1d** of a filter (or kernel) that's size 3 to a tensor of length 7:



The bottom boxes represent our input tensor of seven values, while the connected boxes represent the output after we apply our convolution filter of size three. At the top right corner of the image, the three boxes represent the weights and parameters of the **Conv1d** layer. The convolution filter is applied like a window and it moves to the following values by skipping one value. The number of values to be skipped is called the **stride** and is set to 1 by default. Let's

understand how the output values are being calculated by writing down the calculation for the first and last outputs:

Output 1 \rightarrow $(-0.5209 \times 0.2286) + (-0.0147 \times 2.4488) + (-0.321 \times -0.9498)$

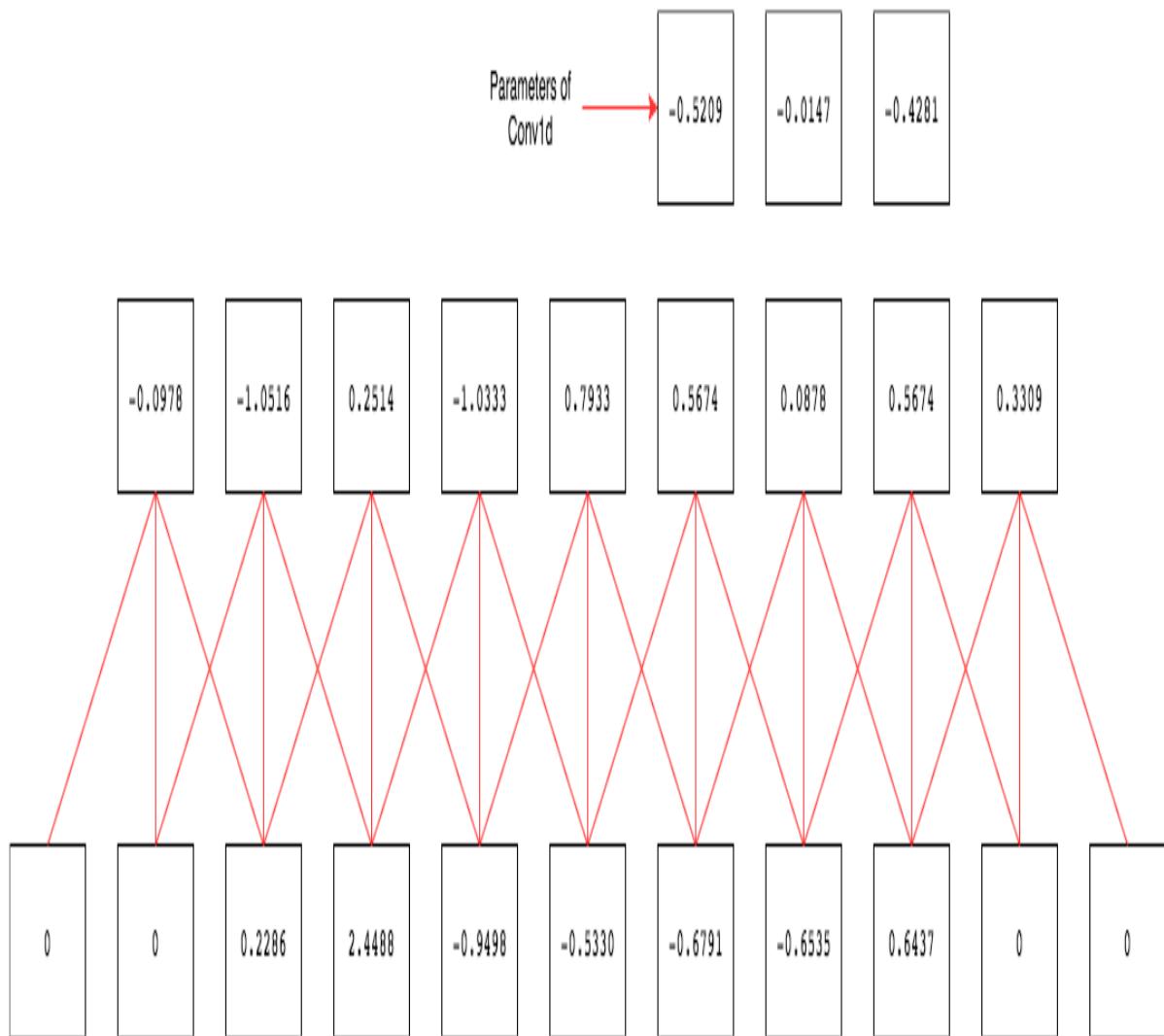
Output 5 \rightarrow $(-0.5209 \times -0.6791) + (-0.0147 \times -0.6535) + (-0.321 \times 0.6437)$

So, by now, it should be clear what a convolution does. It applies a filter (or kernel), that is, a bunch of weights, on the input by moving it based on the value of the stride. In the preceding example, we moved our filter one point at a time. If the stride value is 2, then we would move two points at a time. Let's look at a PyTorch implementation of this to understand how it works:

```
conv = nn.Conv1d(1,1,3,bias=False)
sample = torch.randn(1,1,7)
conv(Variable(sample))

#Check the weights of our convolution filter by
conv.weight
```

There is another important parameter, called **padding**, that is often used with convolutions. As shown in the previous example, if the filter is not applied until the end of the data, that is, when there are not enough elements for the data to stride, it stops. Padding prevents this by adding zeros to both ends of a tensor. Let's look at a one-dimensional example of how padding works:



In the preceding diagram, we applied a **Conv1d** layer with padding 2 and stride 1. Let's look at how Conv2d works on an image.



Before we understand how Conv2d works, I would strongly recommend that you check out an amazing blog (<http://setosa.io/ev/image-kernels/>) that contains a live demo of how convolution works. After you have spent a few minutes playing with the demo, continue reading.

Let's understand what happened in the demo. In the center box of the image, we have two different sets of numbers: one represented in the boxes and the other beneath the boxes. The ones represented in the boxes are pixel values, as highlighted by the white box on the left-hand photo in the demo. The numbers denoted beneath the boxes

are the filter (or kernel) values that are being used to sharpen the image. The numbers are handpicked to do a particular job. In this case, they are sharpening the image. Just like in our previous example, we are doing an element-to-element multiplication and summing up all the values to generate the value of the pixel in the right-hand image. The generated value is highlighted by the white box on the right-hand side of the image.

Though the values in the kernel are handpicked in this example, in CNNs, we do not handpick the values; instead, we initialize them randomly and let gradient descent and backpropagation tune the values of the kernels. The learned kernels will be responsible for identifying different features, such as lines, curves, and eyes. Take a look at the following screenshot, where we can see a matrix of numbers and see how convolution works:

In the preceding screenshot, we assume that the 6×6 matrix represents an image and we apply the convolution filter of size 3×3 . Then, we show how the output is generated. To keep it simple, we are just calculating for the highlighted portion of the matrix. The output is generated by doing the following calculation:

Output → $0.86 \times 0 + -0.92 \times 0 + -0.61 \times 1 + -0.32 \times -1 + -1.69 \times -1 + \dots$

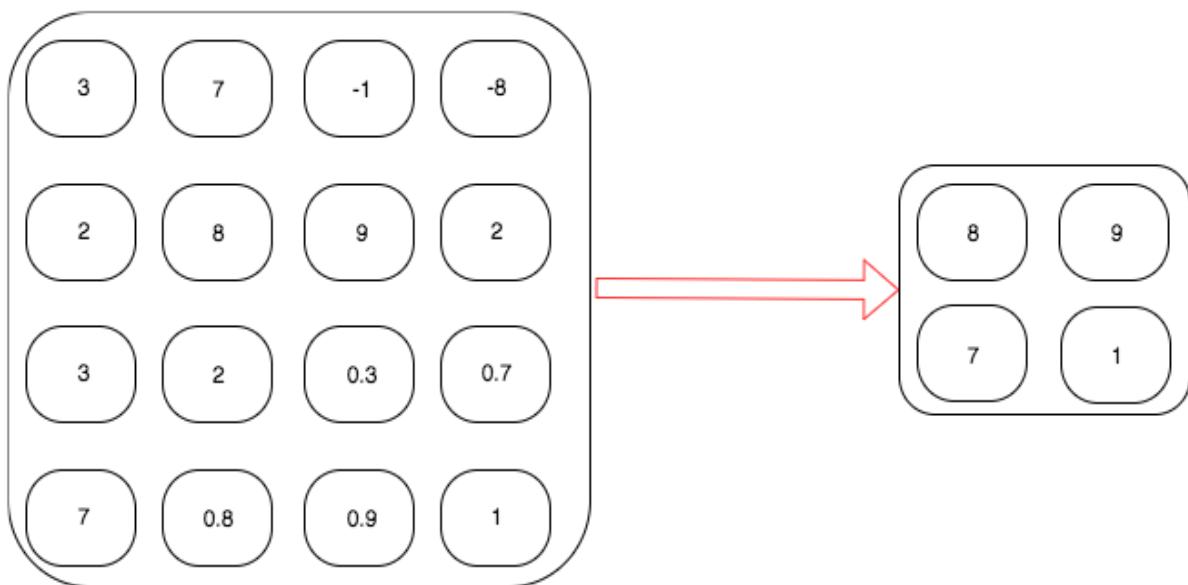
The other important parameter that's used in the Conv2d function is `kernel_size`, which decides the size of the kernel. Some of the commonly used kernel sizes are 1, 3, 5, and 7. The larger the kernel's size, the larger the area that a filter can cover, so it is common to observe filters of 7 or 9 being applied to the input data in the early layers.

Pooling

It is a common practice to add pooling layers after convolution layers since they reduce the size of feature maps and the outcomes of convolution layers.

Pooling offers two different features: one is reducing the size of the data to process and the other is forcing the algorithm to not focus on small changes in the position of an image. For example, a face detection algorithm should be able to detect a face in the picture, irrespective of the position of the face in the photo.

Let's look at how MaxPool2d works. It also uses the same concept of kernel size and strides. It differs from convolutions as it does not have any weights and just acts on the data generated by each filter from the previous layer. If the kernel size is 2×2 , then it considers that size in the image and picks the maximum of that area. Let's look at the following diagram, which will make it clear how MaxPool2d works:

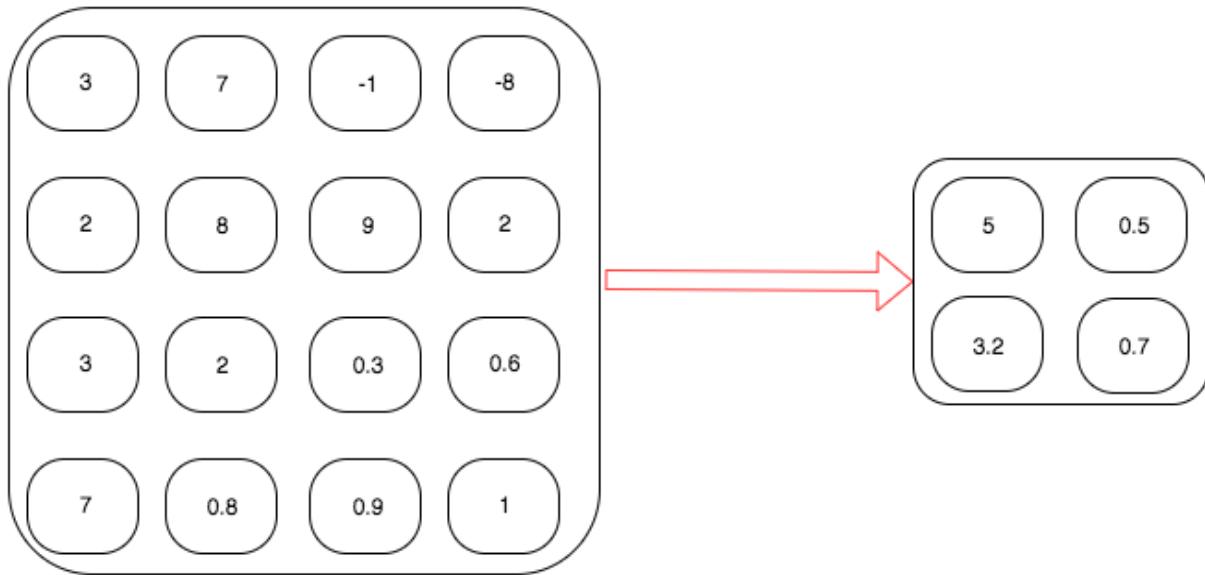


The box on the left-hand side contains the values of feature maps. After applying max pooling, the output is stored on the right-hand side of the box. Let's look at how the output is calculated by writing down the calculation for the values in the first row of the output:

$$Output1 -> \text{Maximum}(3, 7, 2, 8) -> 8$$

$$Output2 -> \text{Maximum}(-1, -8, 9, 2) -> 9$$

The other commonly used pooling technique is **average pooling**. The maximum function is replaced with the average function. The following diagram explains how average pooling works:



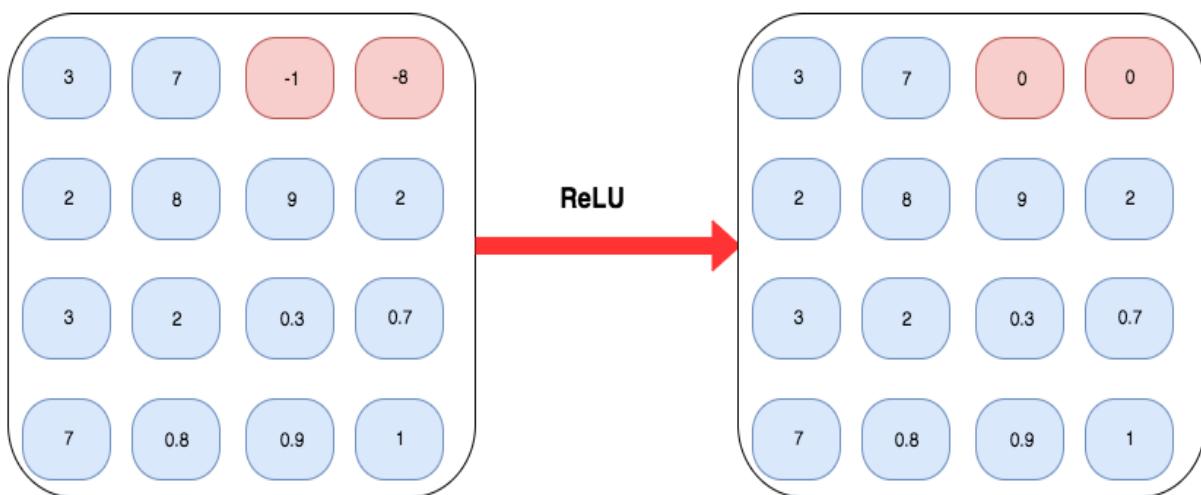
In this example, instead of taking a maximum of four values, we are taking the average of four values. Let's write down the calculation to make it easier to understand:

$$Output1 -> \text{Average}(3, 7, 2, 8) -> 84$$

$$Output2 -> \text{Average}(-1, -8, 9, 2) -> -37$$

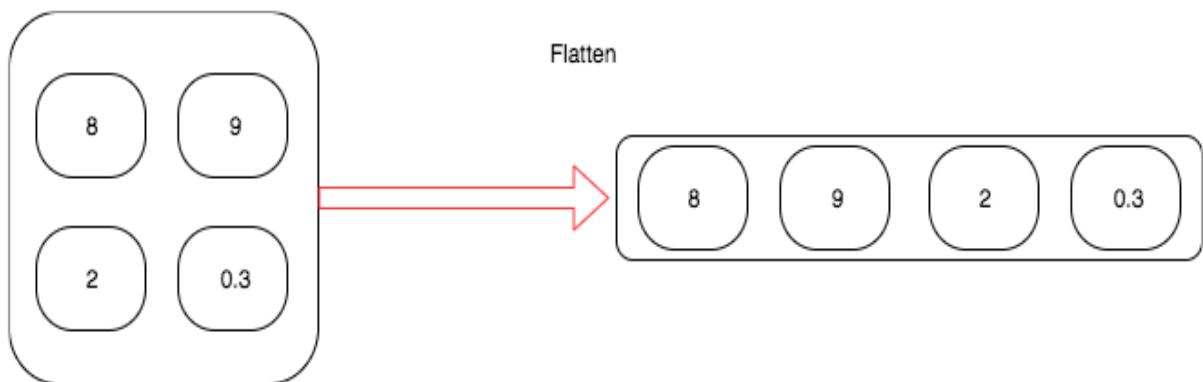
Nonlinear activation – ReLU

It is a common and best practice to have a nonlinear layer after max pooling, or after convolution is applied. Most network architectures tend to use ReLU or different flavors of ReLU. Whatever nonlinear function we choose, it gets applied to each element of the feature map. To make it more intuitive, let's look at an example where we apply ReLU on the same feature map that we applied max pooling and average pooling to:



View

It is a common practice to use a fully connected, or linear, layer at the end of most networks for image classification problems. Here, we are using a two-dimensional convolution that takes a matrix of numbers as input, and outputs another matrix of numbers. To apply a linear layer, we need to flatten the matrix, which is a tensor of two dimensions, into a vector of one dimension. The following diagram shows how view works:

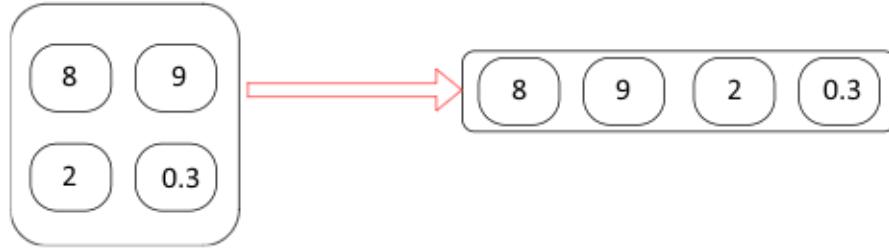
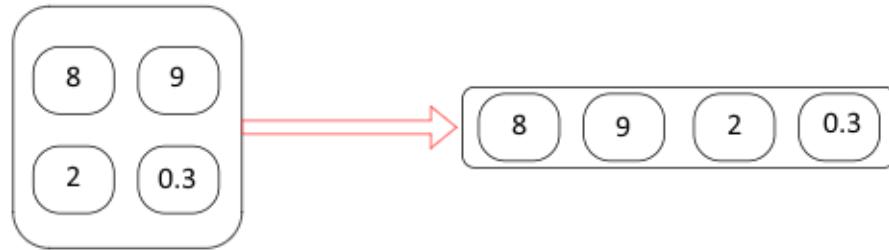


Let's look at the code that's used in our network, which does exactly the same:

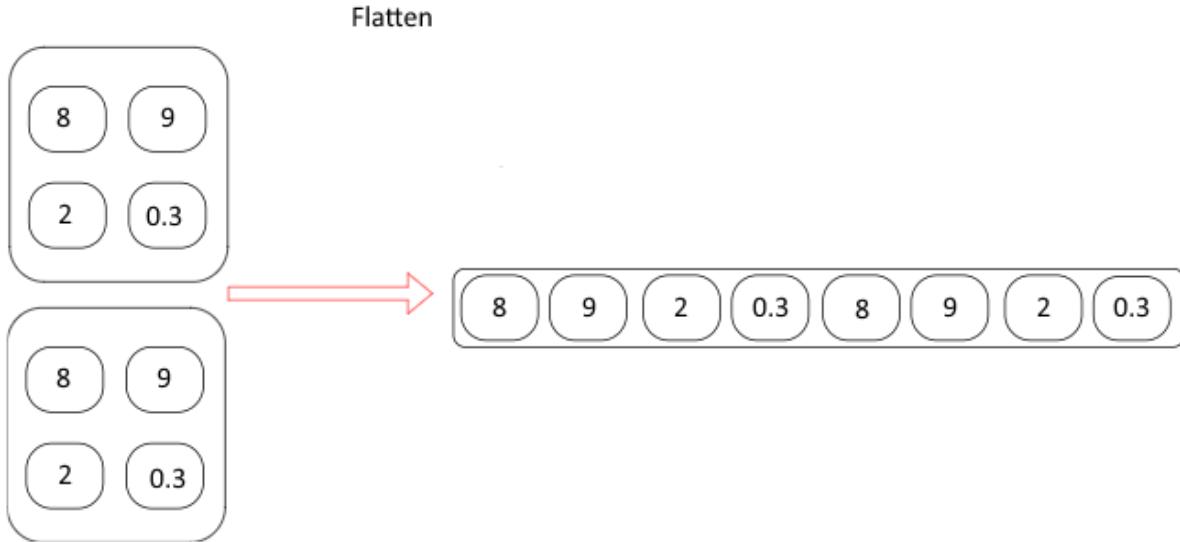
```
| x.view(-1, 320)
```

As we saw earlier, the view method will flatten an n -dimension tensor into a one-dimensional tensor. In our network, the first dimension is of each image. The input data after batching will have a dimension of $32 \times 1 \times 28 \times 28$, where the first number, 32, will denote that there are 32 images of size 28 height, 28 width, and 1 channel since it is a black and white image. When we flatten, we don't want to flatten or mix the data for different images. So, the first argument that we pass to the view function will instruct PyTorch to avoid flattening the data on the first dimension. The following diagram shows how this works:

When view(1,4)



In the preceding diagram, we have data of size $2 \times 1 \times 2 \times 2$; after we apply the `view` function, it converts it into a tensor of size $2 \times 1 \times 4$. Let's look at another example where we don't mention the - 1:



If we ever forget to mention which dimension to flatten, we may end up with unexpected results, so be extra careful at this step.

Linear layer

After we have converted the data from a two-dimensional tensor into a one-dimensional tensor, we pass the data through a linear layer, followed by a nonlinear activation layer. In our architecture, we have two linear layers, one followed by ReLU and the other followed by a `log_softmax` function, which predicts what digit is contained in the given image.

Training the model

To train the model, we need to follow the same process that we followed for our previous dogs and cats image classification problem. The following code snippet trains our model on the provided dataset:

```
def fit_model(epoch,model,data_loader,phase='training',volatile=False):
    if phase == 'training':
        model.train()
    if phase == 'validation': model.eval() volatile=True
    running_loss = 0.0
    running_correct = 0
    for batch_idx , (data,target) in enumerate(data_loader): if is_cuda:
        data,target = data.cuda(),target.cuda()
        data , target = Variable(data,volatile),Variable(target) if phase ==
        'training':
            optimizer.zero_grad() output = model(data)
            loss = F.null_loss(output,target) running_loss +=
            F.null_loss(output,target,size_average=False).data[0] predictions =
            output.data.max(dim=1,keepdim=True) [1]
            running_correct += preds.eq(target.data.view_as(predictions)).cpu().sum()
        if phase == 'training':
            loss.backward() optimizer.step()
            loss = running_loss/len(data_loader.dataset)
            accuracy = 100. * running_correct/len(data_loader.dataset)
            print(f'{phase} loss is {loss:.2f} and {phase} accuracy is
            {running_correct}/{len(data_loader.dataset)}{accuracy:.4f}') return
            loss,accuracy
```

This method has different logic for training and validation. There are primarily two reasons for using different modes:

- In training mode, dropout removes a percentage of values, which shouldn't happen in the validation or testing phase.
- In training mode, we calculate gradients and change the model's parameter value, but backpropagation is not required during the testing or validation phases.

Most of the code in the previous function is self-explanatory. At the end of the function, we return the loss and accuracy of the model for that particular epoch.

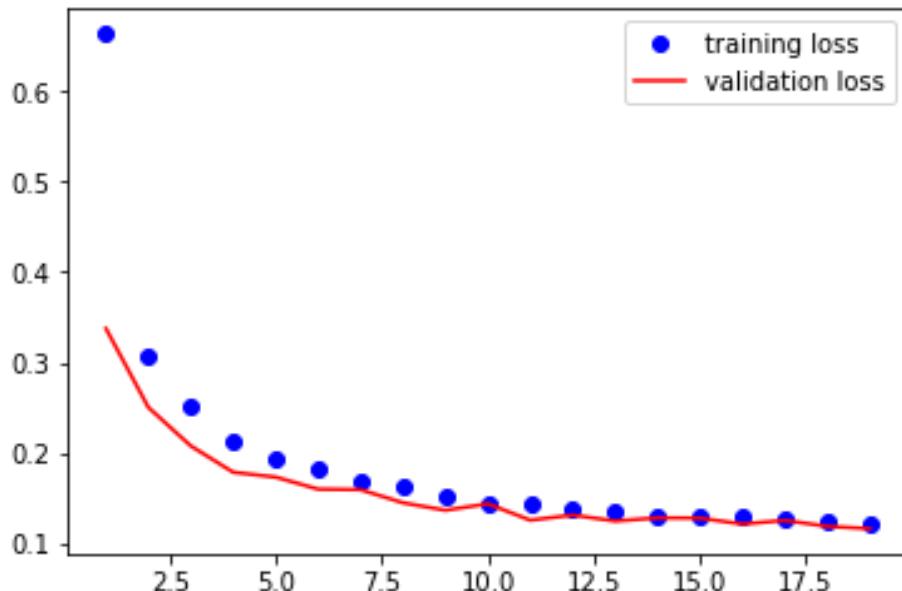
Let's run the model through the preceding function for 20 iterations and plot the loss and accuracy of train and validation to understand how our network performed. The following code runs the `fit` method for the train and test dataset for 20 iterations:

```
model = Network() if is_cuda:  
    model.cuda()  
  
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)  
training_losses, training_accuracy = [], []  
validation_losses, validation_accuracy = [], []  
for epoch in range(1, 20):  
    epoch_loss, epoch_accuracy =  
        fit(epoch, model, training_loader, phase='training')  
    validation_epoch_loss, validation_epoch_accuracy =  
        fit(epoch, model, test_loader, phase='validation')  
    training_losses.append(epoch_loss)  
    training_accuracy.append(epoch_accuracy)  
    validation_losses.append(validation_epoch_loss)  
    validation_accuracy.append(validation_epoch_accuracy)
```

The following code plots the training and test loss:

```
plt.plot(range(1, len(training_losses) + 1), training_losses, 'bo', label =  
         'training loss')  
plt.plot(range(1, len(validation_losses) + 1), validation_losses, 'r', label =  
         'validation loss')  
plt.legend()
```

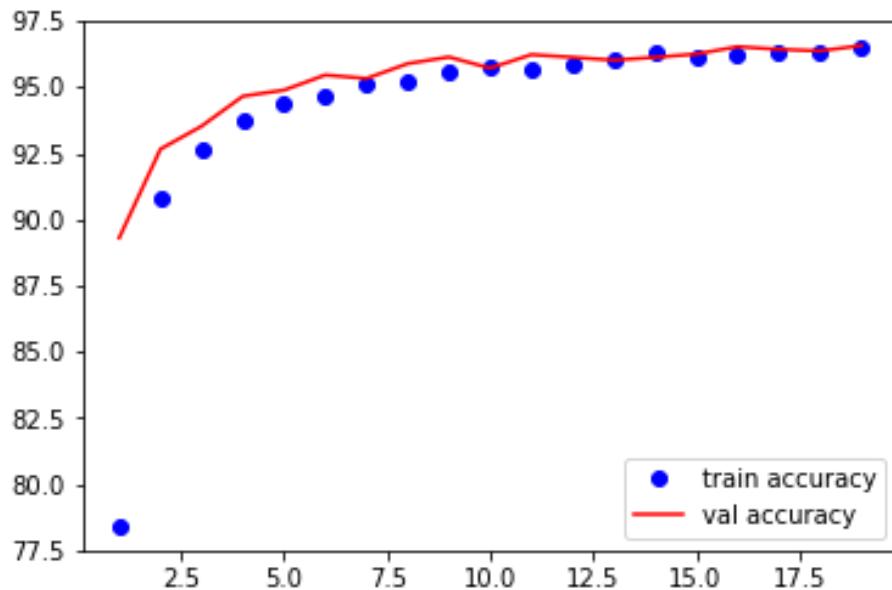
The preceding code generates the following graph:



The following code plots the training and test accuracy:

```
plt.plot(range(1,len(training_accuracy)+1),training_accuracy,'bo',label =  
'train accuracy')  
plt.plot(range(1,len(validation_accuracy)+1),validation_accuracy,'r',labe  
l = 'val accuracy')  
plt.legend()
```

The preceding code generates the following graph:



At the end of the 20th epoch, we achieve a test accuracy of 98.9%. We have got our simple convolutional model working and almost achieving state of the art results. Let's take a look at what happens when we try the same network architecture on our dogs versus cats dataset. We will use the data from the previous chapter, Chapter 3, *Building Blocks of Neural Networks*, and the architecture from the MNIST example with some minor changes. Once we've trained the model, we can evaluate it to understand how well our simple architecture performs.

Classifying dogs and cats – CNN from scratch

We will use the same architecture but with a few minor changes, as listed here:

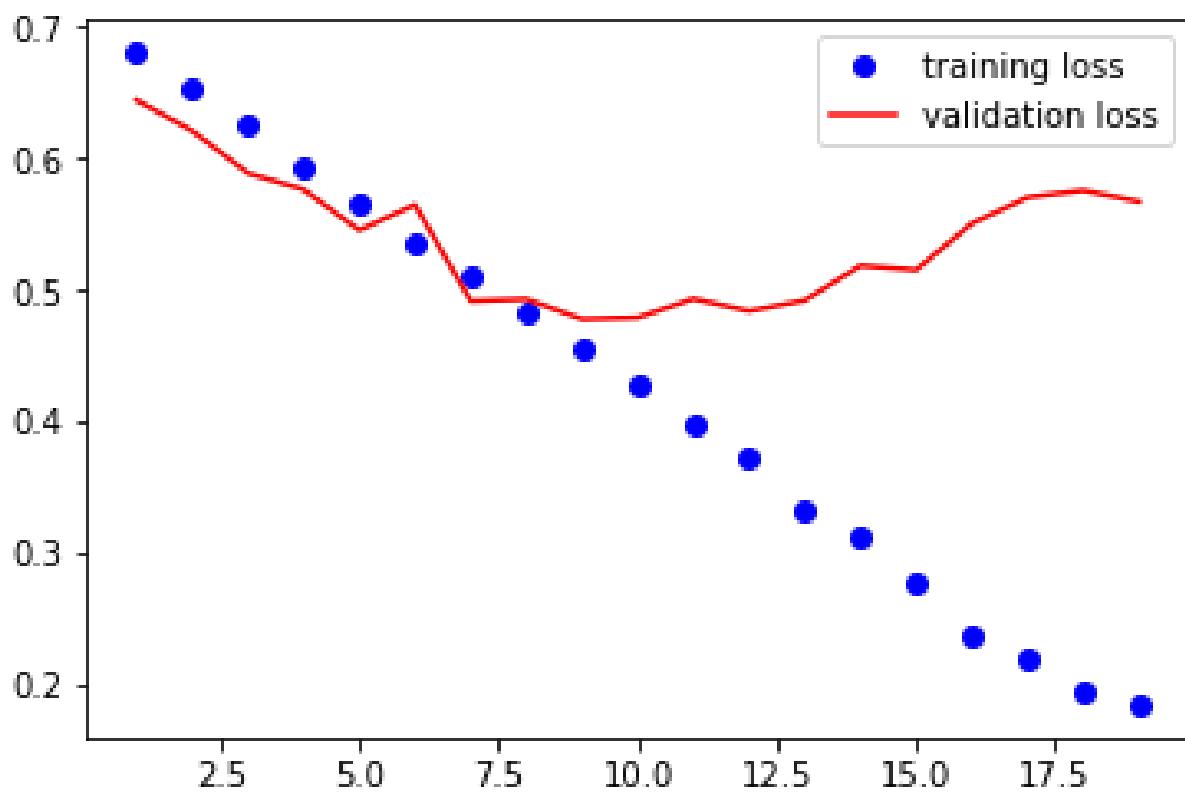
- The input dimensions for the first linear layer will need to change, since the dimensions for our cat and dog images are 256, 256.
- We will add another linear layer to allow the model to learn more flexibly.

Let's look at the code that implements the network architecture:

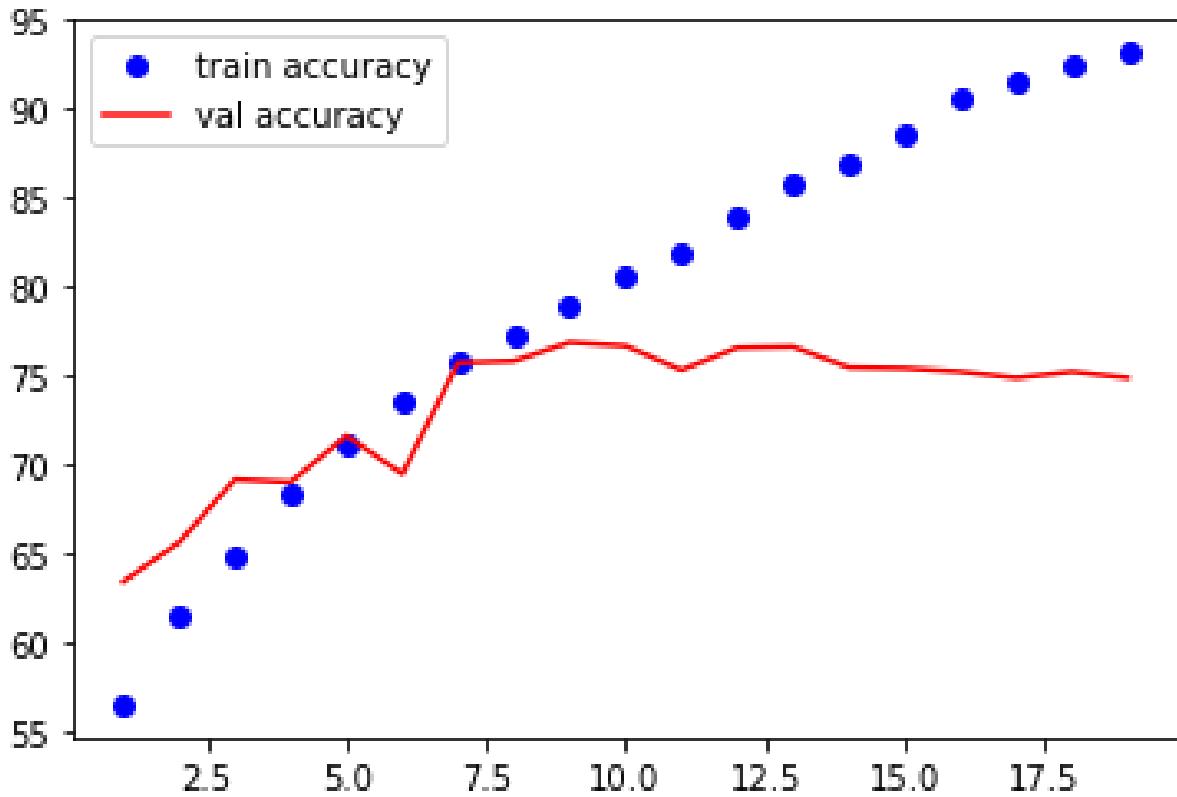
```
class Network(nn.Module): def    init   (self):  
super(). init  ()  
self.conv1 = nn.Conv2d(3, 10, kernel_size=3) self.conv2 = nn.Conv2d(10,  
20, kernel_size=3) self.conv2_drop = nn.Dropout2d()  
self.fc1 = nn.Linear(56180, 500) self.fc2 = nn.Linear(500,50) self.fc3 =  
nn.Linear(50, 2)  
  
def forward(self, x):  
x = F.relu(F.max_pool2d(self.conv1(x), 2))  
x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2)) x =  
x.view(x.size(0),-1)  
x = F.relu(self.fc1(x))  
x = F.dropout(x, training=self.training) x = F.relu(self.fc2(x))  
x = F.dropout(x,training=self.training) x = self.fc3(x)  
return F.log_softmax(x,dim=1)
```

We will use the same training function that we used for the MNIST example, so I'm not going to include the code here. However, let's look at the plots that are generated when the model is trained for 20 iterations.

The loss of the training and validation datasets is plotted as follows:



The accuracy for the training and validation datasets is plotted as follows:

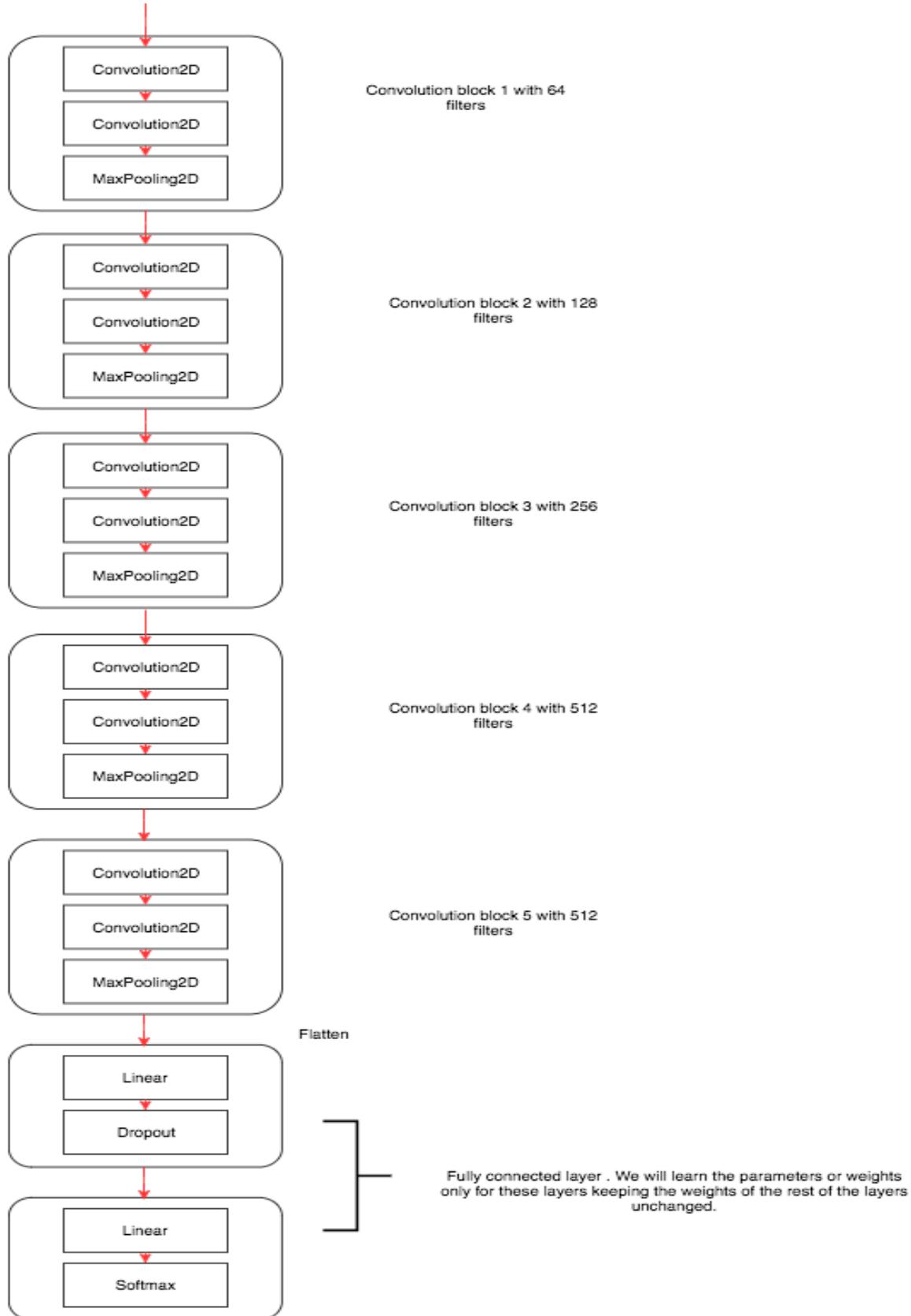


From these plots, it is clear that the training loss is decreasing for every iteration, but the validation loss gets worse. Accuracy also increases during the training, but almost saturates at 75%. This is a clear example showing that the model is not generalizing. In the next section, we will look at another technique called **transfer learning**, which helps us train more accurate models and provides tricks that we can use to make the training faster.

Classifying dogs and cats using transfer learning

Transfer learning is the ability to reuse a trained algorithm on a similar dataset without training it from scratch. We humans don't learn to recognize new images by analyzing thousands of similar images. We just understand the different features that actually differentiate a particular animal, say, a fox, from a dog. We don't need to learn what a fox is from understanding what lines, eyes, and other smaller features are like. Therefore, we will learn how to use a pretrained model to build state of the art image classifiers with very little data.

The first few layers of a CNN architecture focus on smaller features, such as how a line or curve looks. The filters in the later layers of a CNN learn higher-level features, such as eyes and fingers, and the last few layers learn to identify the exact category. A pretrained model is an algorithm that is trained on a similar dataset. Most of the popular algorithms are pretrained on the popular ImageNet dataset to identify 1,000 different categories. Such a pretrained model will have filter weights tuned to identify various patterns. So, let's understand how can we take advantage of these pretrained weights. We will look into an algorithm called **VGG16**, which was one of the earliest algorithms to find success in ImageNet competitions. Though there are more modern algorithms, this algorithm is still popular as it is simple to understand and use for transfer learning. Let's take a look at the architecture of the VGG16 model and then try to understand the architecture and how we can use it to train our image classifier:



The VGG16 architecture contains five VGG blocks. A block is a set of convolution layers, a nonlinear activation function, and a max-pooling function. All the algorithm parameters are tuned to achieve state of the art results when it comes to classifying 1,000 categories. The algorithm takes input data in the form of batches, which are normalized by the mean and standard deviation of the ImageNet dataset.

In transfer learning, we try to capture what the algorithm learns by freezing the learned parameters of most of the layers of the architecture. It is often good practice to fine-tune only the last layers of the network. In this example, we'll train only the last few linear layers and leave the convolutional layers intact since the features that are learned by the convolutional features are mostly used for all kinds of image problems where the images share similar properties. Let's train a VGG16 model using transfer learning to classify dogs and cats. We'll walk through the steps to implement this in the next section.

Creating and exploring a VGG16 model

PyTorch provides a set of trained models in its `torchvision` library. Most of them accept an argument called `pretrained` when `True`, which downloads the weights that have been tuned for the **ImageNet** classification problem. We can use the following code to create a VGG16 model:

```
| from torchvision import models  
| vgg = models.vgg16(pretrained=True)
```

Now, we have our VGG16 model and all the pretrained weights ready to be used. When the code is run for the first time, it could take several minutes, depending on your internet speed. The size of the weights could be around 500 MB. We can take a quick look at the VGG16 model by printing it. Understanding how these networks are implemented turns out to be very useful when we use modern architectures. Let's take a look at the model:

```
VGG (  
    features): Sequential (  
        (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
        (1): ReLU (inplace)  
        (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,  
1))  
        (3): ReLU (inplace)  
        (4): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))  
        (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,  
1))  
        (6): ReLU (inplace)  
        (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,  
1))  
        (8): ReLU (inplace)  
        (9): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))  
        (10):Conv2d(128, 256, kernel_size=(3, 3), stride=(1,1), padding=(1,  
1))  
        (11): ReLU (inplace)  
        (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,  
1))  
        (13): ReLU (inplace)  
        (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
```

```

    1))
    (15): ReLU (inplace)
    (16): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (18): ReLU (inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (20): ReLU (inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (22): ReLU (inplace)
    (23): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1,1))
    (25): ReLU (inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (27): ReLU (inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (29): ReLU (inplace)
    (30): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
)
(classifier): Sequential (
    (0): Linear (25088 -> 4096)
    (1): ReLU (inplace)
    (2): Dropout (p = 0.5)
    (3): Linear (4096 -> 4096)
    (4): ReLU (inplace)
    (5): Dropout (p = 0.5)
    (6): Linear (4096 -> 1000)
)
)

```

The model summary contains two sequential models: `features` and `classifiers`. The `features` sequential model contains the layers that we are going to freeze.

Freezing the layers

Let's freeze all the layers of the features model, which contains the convolutional block. Freezing the weights in the layers will prevent the weights of these convolutional blocks from updating. Since the weights of the model are trained to recognize a lot of important features, our algorithm will be able to do the same from the very first iteration. The ability to use model's weights, which were initially trained for a different use case, is called **transfer learning**.

Now, let's look at how we can freeze the weights, or parameters, of layers:

```
| for param in vgg.features.parameters(): param.requires_grad = False
```

This code prevents the optimizer from updating the weights.

Fine-tuning VGG16

The VGG16 model has been trained to classify 1,000 categories, but not trained to classify dogs and cats. Therefore, we need to change the output features of the last layer from 1,000 to 2. We can use the following code to do this:

```
| vgg.classifier[6].out_features = 2
```

The `vgg.classifier` function gives us access to all the layers inside the sequential model, and the sixth element will contain the last layer. When we train the VGG16 model, we only need the classifier parameters to be trained. Therefore, we only pass `classifier.parameters` to the optimizer, as follows:

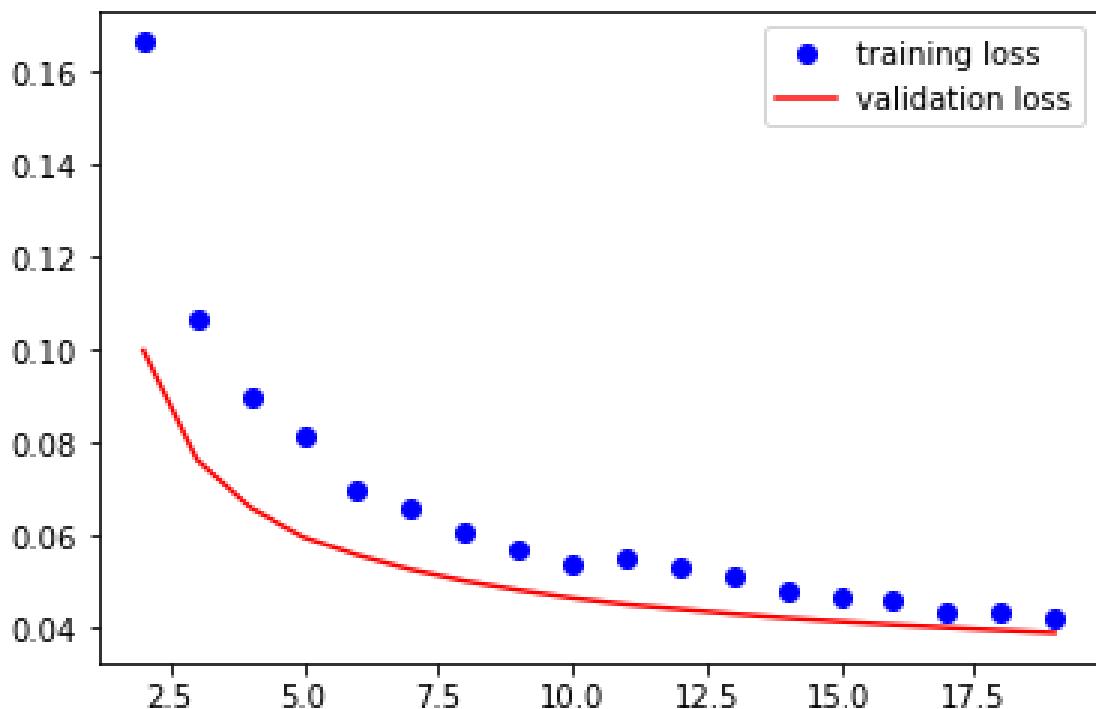
```
| optimizer = optim.SGD(vgg.classifier.parameters(), lr=0.0001, momentum=0.5)
```

Training the VGG16 model

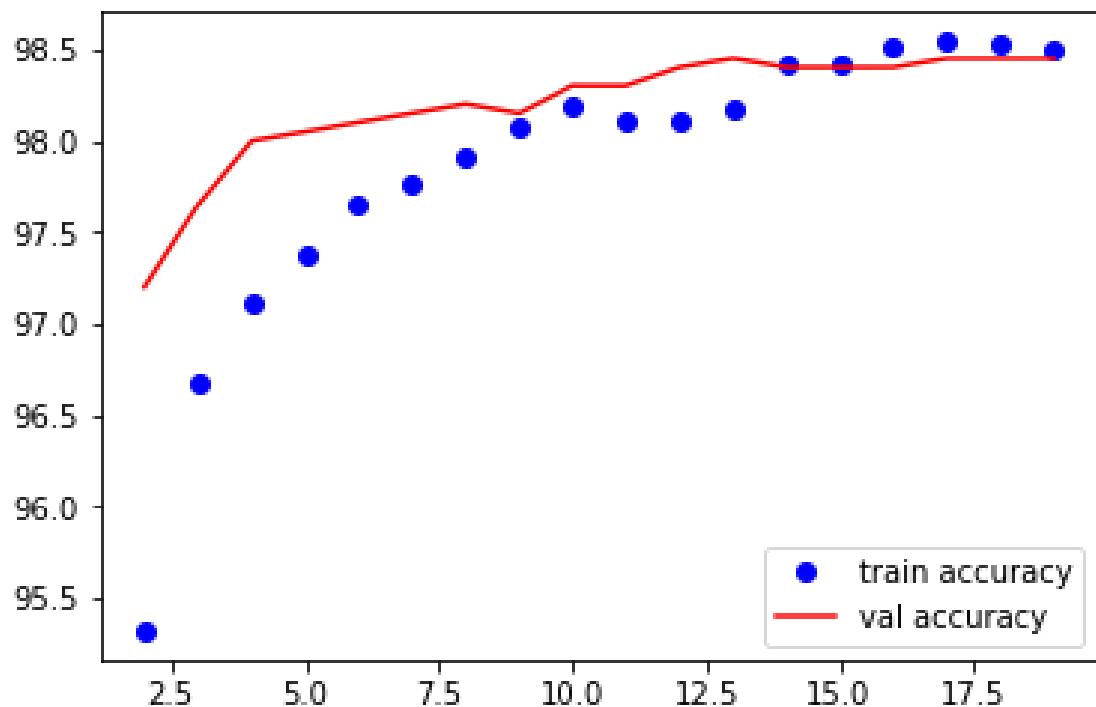
So far, we have created the model and optimizer. Since we are using the Dogs versus Cats dataset, we can use the same data loaders and the train function to train our model. Remember: when we train the model, only the parameters inside the classifier change. The following code snippet will train the model for 20 epochs, thereby reaching a validation accuracy of 98.45%:

```
training_losses , training_accuracy = [], []
validation_losses , validation_accuracy = [], []
for epoch in range(1,20):
    epoch_loss, epoch_accuracy =
        fit(epoch, vgg, training_data_loader, phase='training')
    validation_epoch_loss , validation_epoch_accuracy =
        fit(epoch, vgg, valid_data_loader, phase='validation')
    training_losses.append(epoch_loss)
    training_accuracy.append(epoch_accuracy)
    validation_losses.append(validation_epoch_loss)
    validation_accuracy.append(validation_epoch_accuracy)
```

Let's visualize the training and validation loss:



Let's visualize the training and validation accuracy:



We can apply a couple of tricks, such as data augmentation, and play with different values of the dropout to improve the model's generalization capabilities. The following code snippet changes the dropout value in the classifier module of VGG to 0.2 from 0.5 and trains the model:

```
for layer in vgg.classifier.children(): if(type(layer) == nn.Dropout):
    layer.p = 0.2

#Training
training_losses , training_accuracy = [],[] validation_losses ,
validation_accuracy = [],[]
for epoch in range(1,3):
    epoch_loss, epoch_accuracy =
    fit(epoch,vgg,training_data_loader,phase='training')
    validation_epoch_loss , validation_epoch_accuracy =
    fit(epoch,vgg,valid_data_loader,phase='validation')
    training_losses.append(epoch_loss)
    training_accuracy.append(epoch_accuracy)
    validation_losses.append(validation_epoch_loss)
    validation_accuracy.append(validation_epoch_accuracy)
```

Training this for a few epochs gave me a slight improvement; you can try playing with the different dropout values yourself and see if you can get better results. Another important trick we can use to improve model generalization is to add more data or do data augmentation. We can perform data augmentation by randomly flipping the image horizontally or rotating the image by a small angle. The `torchvision transforms` module provides different functionalities for doing data augmentation and they do so dynamically, changing for every epoch. We can implement data augmentation using the following code:

```
training_transform =transforms.Compose([transforms.Resize((224,224)),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.RandomRotation(0.2),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485,
 0.32, 0.406], [0.229, 0.224, 0.225])
])

train = ImageFolder('dogsandcats/train/',training_transform) valid =
ImageFolder('dogsandcats/valid/',simple_transform)

#Training
```

```
training_losses , training_accuracy = [], []
validation_losses , validation_accuracy = [], []
for epoch in range(1,3):
    epoch_loss, epoch_accuracy =
fit(epoch,vgg,training_data_loader,phase='training')
    validation_epoch_loss , validation_epoch_accuracy =
fit(epoch,vgg,valid_data_loader,phase='validation')
    training_losses.append(epoch_loss)
    training_accuracy.append(epoch_accuracy)
    validation_losses.append(validation_epoch_loss)
    validation_accuracy.append(validation_epoch_accuracy)
```

The output of the preceding code is as follows:

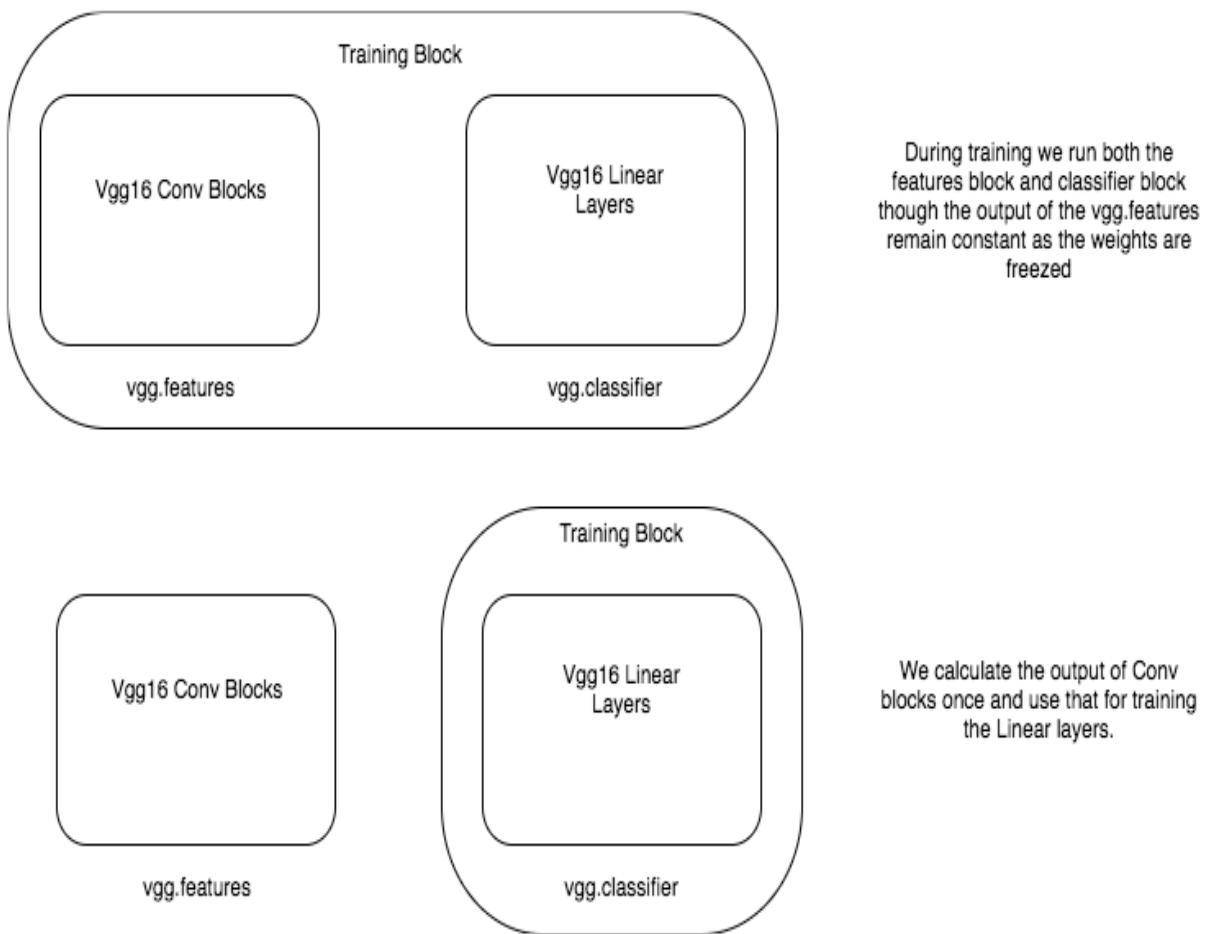
```
#Results
training loss is 0.041 and training accuracy is 22657/23000 98.51
validation loss is 0.043 and validation accuracy is 1969/2000 98.45
training loss is 0.04 and training accuracy is 22697/23000 98.68
validation loss is 0.043 and validation accuracy is 1970/2000 98.5
```

Training the model with augmented data improved the model's accuracy by 0.1% by running just two epochs; we can run it for a few more epochs to improve further. If you have been training these models while reading this book, you will have realized that training each epoch could take more than a couple of minutes, depending on the GPU you are running. Let's look at a technique where we can train each epoch in a few seconds.

Calculating pre-convoluted features

When we freeze the convolution layers and the training model, the input to the fully connected layers, or dense layers, (`vgg.classifier`) is always the same. To understand this better, let's treat the convolution block – which in our example is the `vgg.features` block – as a function that has learned weights and doesn't change during training. So, calculating the convolution features and storing them will help us improve the training speed. The time to train the model will decrease since we only calculate these features once, instead of calculating for each epoch.

Let's visually understand this and implement it:



The first box depicts how training is done in general, which could be slow since we calculate the convolutional features for every epoch, though the values don't change. In the bottom box, we calculate the convolutional features once and train only the linear layers. To calculate the pre-convoluted features, we need to pass all the training data through the convolution blocks and store them. To perform this, we need to select the convolution blocks of the VGG model. Fortunately, the PyTorch implementation of VGG16 has two sequential models, so just picking the first sequential model's features is enough. The following code does this for us:

```

vgg = models.vgg16(pretrained=True)  vgg = vgg.cuda()
features = vgg.features

training_data_loader =
torch.utils.data.DataLoader(train,batch_size=32,num_workers=3,shuffle=False)
    
```

```

valid_data_loader =
torch.utils.data.DataLoader(valid,batch_size=32,num_workers=3,shuffle=False)

def preconvfeat(dataset,model):
    conv_features = []
    labels_list = []
    for data in dataset:
        inputs,labels = data
        if is_cuda:
            inputs , labels = inputs.cuda(),labels.cuda()
            inputs , labels = Variable(inputs),Variable(labels)
            output = model(inputs)
            conv_features.append(output.data.cpu().numpy())
            labels_list.append(labels.data.cpu().numpy())
    conv_features = np.concatenate([[feat] for feat in conv_features])

    return (conv_features,labels_list)
conv_feat_train,labels_train = preconvfeat(training_data_loader,features)
conv_feat_val,labels_val = preconvfeat(valid_data_loader,features)

```

In the preceding code, the `preconvfeat` method takes in the dataset and the `vgg` model and returns the convoluted features, along with the labels associated with it. The rest of the code is similar to what we used in the preceding examples to create data loaders and datasets.

Once we have the convolutional features for the train and validation sets, we can create a PyTorch dataset and `DataLoader` classes, which will ease up our training process. The following code creates the dataset and `DataLoader` for our convolutional features:

```

class CustomDataset(Dataset):
    def __init__(self,feat,labels): self.conv_feat = feat self.labels = labels
    def __len__(self):
        return len(self.conv_feat) def __getitem__(self,idx):
        return self.conv_feat[idx],self.labels[idx]

training_feat_dataset = CustomDataset(conv_feat_train,labels_train)
validation_feat_dataset = CustomDataset(conv_feat_val,labels_val)

training_feat_loader =
DataLoader(training_feat_dataset,batch_size=64,shuffle=True)
validation_feat_loader =
DataLoader(validation_feat_dataset,batch_size=64,shuffle=True)

```

Since we have our new data loaders, which generate batches of convoluted features along with labels, we can use the same `train` function that we have been using in the other examples. Now, we will use `vgg.classifier` as the model to create the optimizer and fit methods.

The following code trains the classifier module to identify dogs and cats. On a Titan X GPU, each epoch takes less than five seconds, which would otherwise take a few minutes:

```
training_losses , training_accuracy = [],[] validation_losses ,
validation_accuracy = [],[]
for epoch in range(1,20): epoch_loss, epoch_accuracy =
fit_numpy(epoch,vgg.classifier,training_feat_loader,phase='training')
validation_epoch_loss , validation_epoch_accuracy =
fit_numpy(epoch,vgg.classifier,validation_feat_loader,phase='validation')
training_losses.append(epoch_loss)
training_accuracy.append(epoch_accuracy)
validation_losses.append(validation_epoch_loss)
validation_accuracy.append(validation_epoch_accuracy)
```

Understanding what a CNN model learns

Deep learning models are often said to not be interpretable. However, there are different techniques that we can use to interpret what happens inside these models. For images, the features that are learned by convnets are interpretable. In this section, we will explore two popular techniques so that we can understand convnets.

Visualizing outputs from intermediate layers

Visualizing the outputs from intermediate layers will help us understand how the input image is being transformed across different layers. Often, the output from each layer is called an **activation**. To do this, we should extract the output from intermediate layers, which can be done in different ways. PyTorch provides a method called `register_forward_hook`, which allows us to pass a function that can extract the outputs of a particular layer.

By default, PyTorch models only store the output of the last layer so that they use memory optimally. So, before we inspect what the activations from the intermediate layers look like, let's learn how to extract outputs from the model. Take a look at the following code snippet, which extracts outputs. We will walk through it to understand what happens:

```
vgg = models.vgg16(pretrained=True).cuda()

class LayerActivations(): features=None
def __init__(self,model,layer_num):
    self.hook = model[layer_num].register_forward_hook(self.hook_fn) def
    hook_fn(self,module,input,output):
        self.features = output.cpu() def remove(self):
            self.hook.remove()

conv_out = LayerActivations(vgg.features,0) o = vgg(Variable(img.cuda()))
conv_out.remove()
act = conv_out.features
```

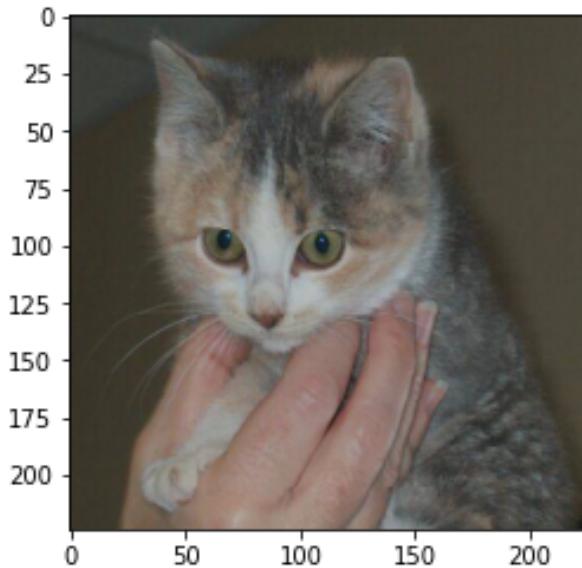
We start by creating a pre-trained VGG model, from which we extract the outputs of a particular layer. The `LayerActivations` class instructs PyTorch to store the output of a layer in the `features` variable. Let's walk through each function inside the `LayerActivations` class.

The `_init_` function takes a model and the layer number of the layer that the outputs need to be extracted from as arguments. We call the `register_forward_hook` method on the layer and pass in a function.

PyTorch, when doing a forward pass – that is, when the images are passed through the layers – calls the function that is passed to the `register_forward_hook` method. This method returns a handle, which can be used to deregister the function that is passed to the `register_forward_hook` method.

The `register_forward_hook` method passes three values to the function that we pass to it. The `module` parameter allows us to access the layer itself. The second parameter is `input`, which refers to the data that is flowing through the layer. The third parameter is `output`, which allows us to access the transformed inputs, or activations, of the layer. We store the output of the `features` variable in the `LayerActivations` class.

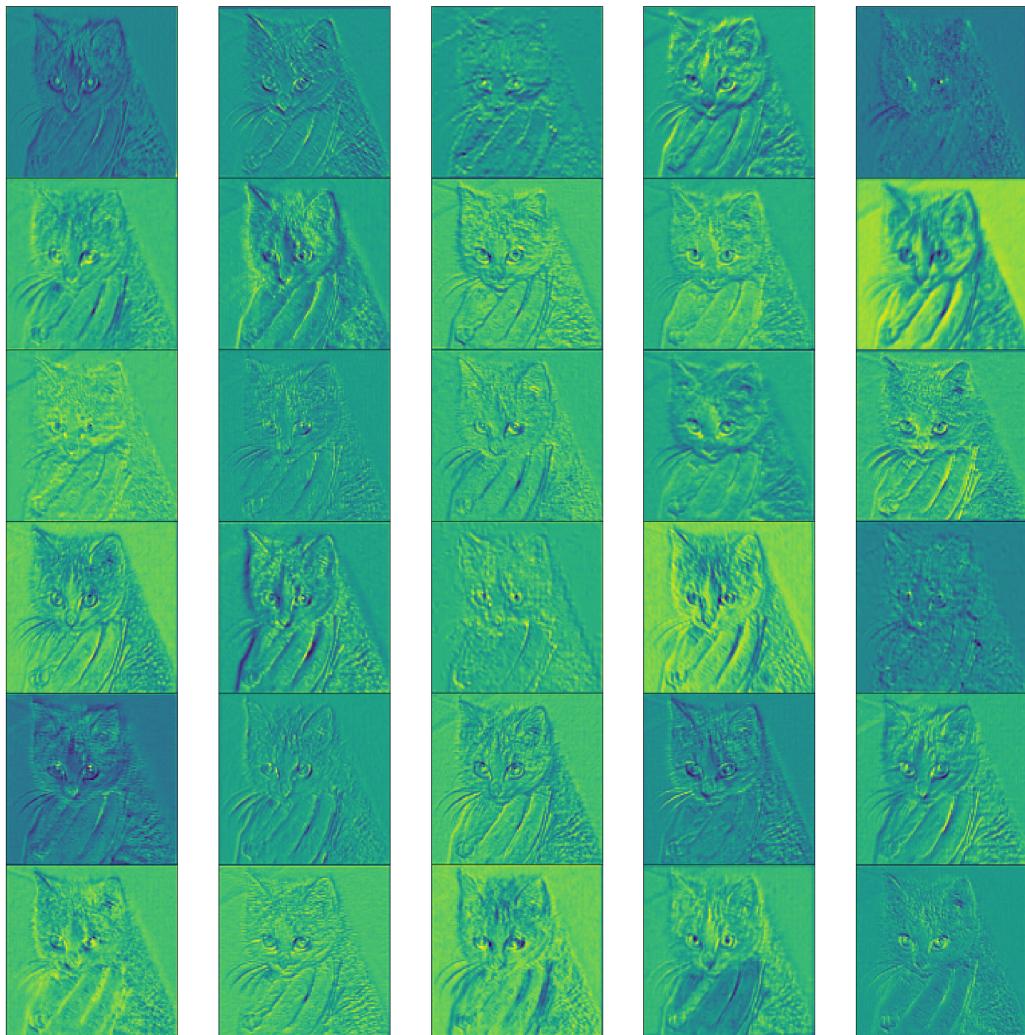
The third function takes the hook from the `_init_` function and deregisters the function. Now, we can pass the model and the layer numbers of the activations we are looking for. Let's look at the activations that will be created for the following image for different layers:



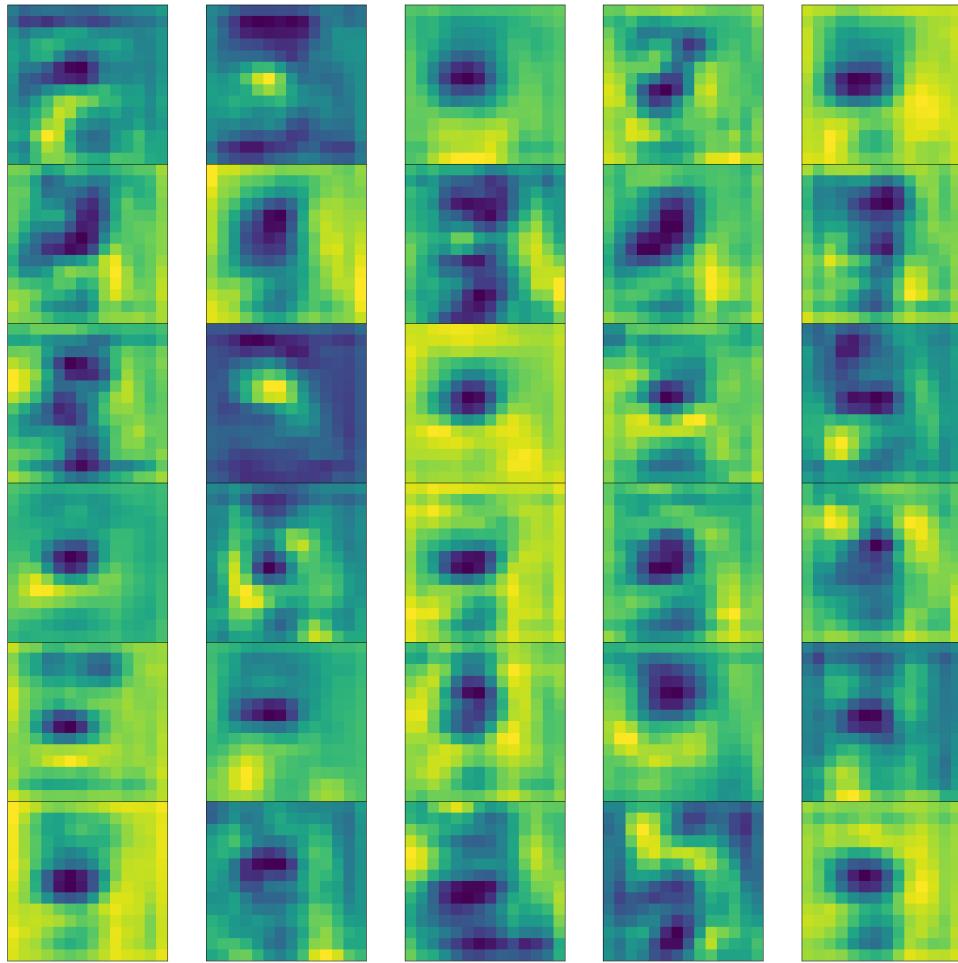
Let's visualize some of the activations that are created by the first convolution layer and the code that's used for this:

```
fig = plt.figure(figsize=(20,50))
fig.subplots_adjust(left=0,right=1,bottom=0,top=0.8,hspace=0,
wspace=0.2)
for i in range(30):
    ax = fig.add_subplot(12,5,i+1,xticks=[],yticks=[]) ax.imshow(act[0][i])
```

Let's visualize some of the activations that are created by the fifth convolutional layer:



Let's look at the last CNN layer:



By looking at what the different layers generate, we can see that the early layers detect lines and edges, while the final layers tend to learn high-level features and are less interpretable.

Before we move on to visualizing weights, let's learn how the features maps or activations represent themselves after the ReLU layer. So, let's visualize the outputs of the second layer.

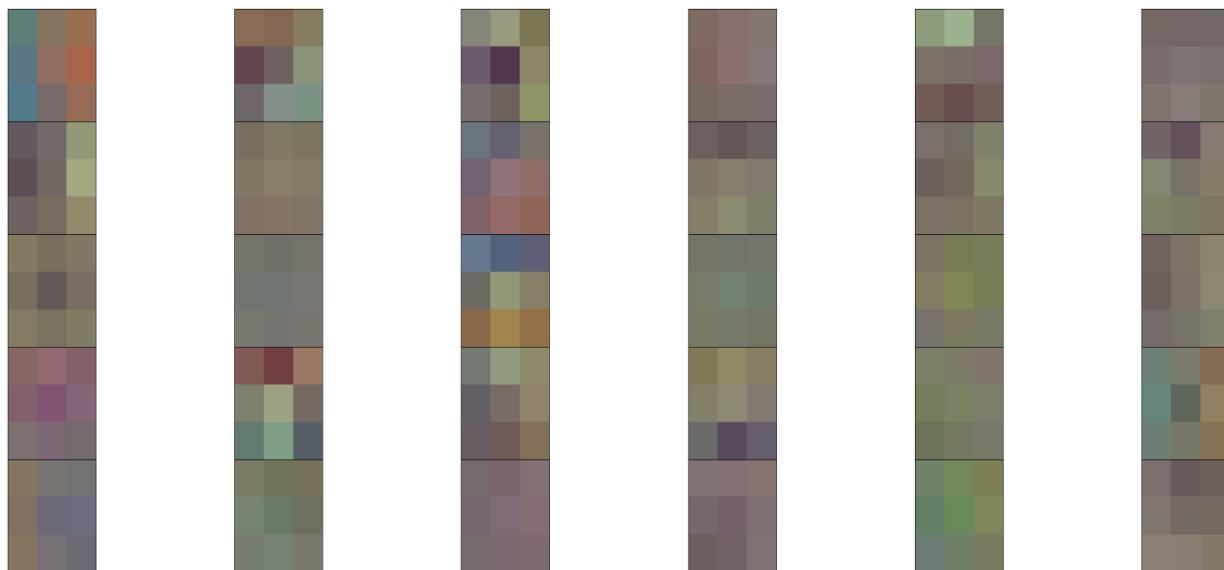
If you take a quick look at the fifth image in the second row of the preceding image, it looks like the filter is detecting the eyes in the image. When the models do not perform, these visualization tricks can help us understand why the model may not be working.

Visualizing the weights of the CNN layer

Getting the model weights for a particular layer is straightforward. All the model weights can be accessed through the `state_dict` function. The `state_dict` function returns a dictionary, with `keys` as its layers and `weights` as its values. The following code demonstrates how we can pull weights for a particular layer and visualize them:

```
| vgg.state_dict().keys()  
| cnn_weights = vgg.state_dict()['features.0.weight'].cpu()
```

The preceding code provides us with the following output:



Each box represents the weights of a filter that are 3×3 in size. Each filter is trained to identify certain patterns in the images.

Summary

In this chapter, we learned how to build an image classifier using convnets, as well as how to use a pretrained model. We covered tricks on how to speed up the process of training by using pre-convoluted features. We also looked at the different techniques we can use to understand what goes on inside a CNN.

In the next chapter, we will learn how to handle sequential data using recurrent neural networks.

Natural Language Processing with Sequence Data

In this chapter, we will be looking at different representations of text data that are useful for building deep learning models. We will be helping you to understand **recurrent neural networks (RNNs)**. This chapter will cover different implementations of RNNs, such as **long short-term memory (LSTM)** and **gated recurrent unit (GRU)**, which power most of the deep learning models for text and sequential data. We will be looking at different representations of text data and how they are useful for building deep learning models. In addition, this chapter will address one-dimensional convolutions that can be used for sequential data.

Some of the applications that can be built using RNNs are as follows:

- **Document classifiers:** Identifying the sentiment of a tweet or review, classifying news articles
- **Sequence-to-sequence learning:** For tasks such as language translations, converting English to French
- **Time-series forecasting:** Predicting the sales of a store when given details about previous days' sales records

The following topics will be covered in this chapter:

- Working with text data
- Training embedding by building a sentiment classifier
- Using pretrained word embeddings
- Recursive neural networks
- Solving text classification problem using LSTM
- Convolutional network on sequence data
- Language modeling

Working with text data

Text is one of the most commonly used sequential data types. Text data can be seen as either a sequence of characters or a sequence of words. It is common to see text as a sequence of words for most problems. Deep learning sequential models such as RNNs and its variants are able to learn important patterns from text data that can solve problems in areas such as the following:

- Natural language understanding
- Document classification
- Sentiment classification

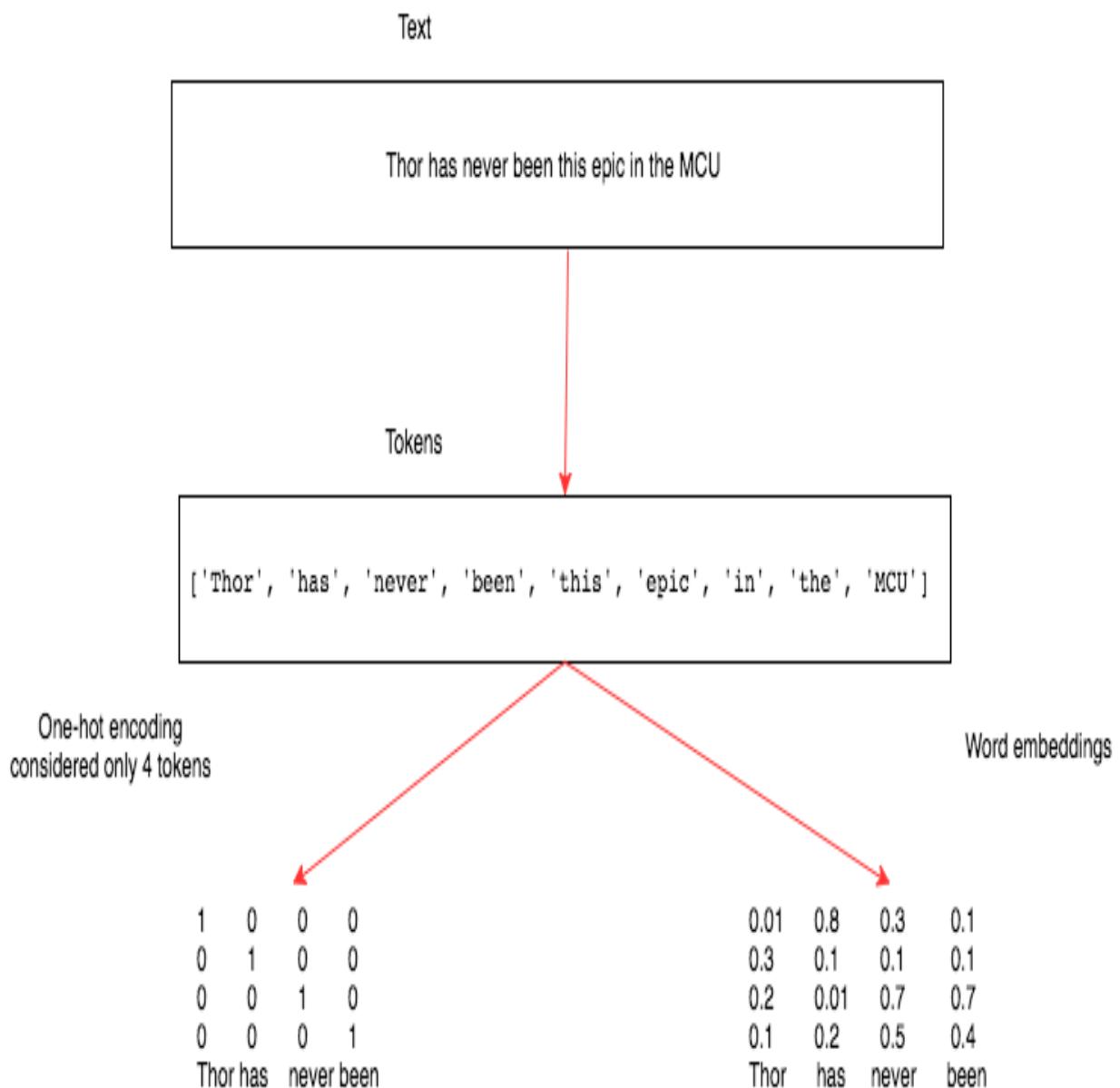
These sequential models also act as important building blocks for various systems, such as **question and answer (QA)** systems.

Though these models are highly useful in building these applications, they do not have an understanding of human language, due to its inherent complexities. These sequential models are able to successfully find useful patterns that are then used for performing different tasks. Applying deep learning to text is a fast-growing field, and a lot of new techniques arrive every month. We will cover the fundamental components that power most modern-day deep learning applications.

Deep learning models, like any other machine learning model, do not understand text, so we need to convert text into numerical representations. The process of converting text into numerical representations is called **vectorization** and can be done in different ways, as outlined here:

- Convert text into words and represent each word as a vector
- Convert text into characters and represent each character as a vector
- Create n-grams of words and represent them as vectors

Text data can be broken down into one of these representations. Each smaller unit of text is called a token, and the process of breaking text into tokens is called **tokenization**. There are a lot of powerful libraries available in Python that can help us in tokenization. Once we convert the text data into tokens, we then need to map each token to a vector. One-hot encoding and word embedding are the two most popular approaches for mapping tokens to vectors. The following diagram summarizes the steps for converting text into vector representations:



Let's look in more detail at tokenization, n-gram representation, and vectorization.

Tokenization

Given a sentence, splitting it into either characters or words is called tokenization. There are libraries, such as spaCy, that offer complex solutions to tokenization. Let's use simple Python functions such as `split` and `list` to convert the text into tokens.

To demonstrate how tokenization works on characters and words, let's consider a small review of the movie *Toy Story*. We will work with the following text:

Just perfect. Script, character, animation....this manages to break free of the yoke of 'children's movie' to simply be one of the best movies of the 90's, full-stop.

Converting text into characters

The Python `list` function takes a string and converts it into a list of individual characters. This does the job of converting the text into characters. The following code block shows the code used and the results:

```
toy_story_review = "Just perfect. Script, character, animation....this  
manages to break free of the yoke of 'children's movie' to simply be one  
of the best movies of the 90's, full-stop."  
  
print(list(toy_story_review))
```

The result is as follows:

```
['J', 'u', 's', 't', ' ', 'p', 'e', 'r', 'f', 'e', 'c', 't', '.', ' ',  
's', 'c', 'r', 'i', 'p', 't', ' ', ' ', 'c', 'h', 'a', 'r', 'a', 'c',  
't', 'e', 'r', ' ', ' ', 'a', 'n', 'i', 'm', 'a', 't', 'i', 'o', 'n',  
'.', ' ', ' ', ' ', 't', 'h', 'i', 's', ' ', 'm', 'a', 'n', 'a', 'g',  
'e', 's', ' ', 't', 'o', ' ', 'b', 'r', 'e', 'a', 'k', ' ', 'f', 'r',  
'e', 'e', ' ', 'o', 'f', ' ', 't', 'h', 'e', ' ', 'y', 'o', 'k', 'e',  
' ', 'o', 'f', ' ', "", 'c', 'h', 'i', 'l', 'd', 'r', 'e', 'n', "", 's',  
' ', 'm', 'o', 'v', 'i', 'e', "", ' ', 't', 'o', ' ', 's', 'i', 'm',  
'p', 'l', 'y', ' ', 'b', 'e', ' ', 'o', 'n', 'e', ' ', 'o', 'f', ' ',  
't', 'h', 'e', ' ', 'b', 'e', 's', 't', ' ', 'm', 'o', 'v', 'i', 'e',  
's', ' ', 'o', 'f', ' ', 't', 'h', 'e', ' ', '9', '0', "", 's', ' ',  
' ', 'f', 'u', 'l', 'l', ' ', 's', 't', 'o', 'p', '.']
```

This result shows how our simple Python function has converted text into tokens.

Converting text into words

We will use the `split` function available in the Python string object to break the text into words. The `split` function takes an argument, and based on this, it splits the text into tokens. For our example, we will use spaces as delimiters. The following code block demonstrates how we can convert text into words using the Python `split` function:

```
| print(list(toy_story_review.split()))
```

This results in the following output:

```
| ['Just', 'perfect.', 'Script,', 'character,', 'animation....this',
| 'manages', 'to', 'break', 'free', 'of', 'the', 'yoke', 'of',
| "'children's", "movie'", 'to', 'simply', 'be', 'one', 'of', 'the',
| 'best', 'movies', 'of', 'the', "90's,", 'full-stop.']}
```

In the preceding code, we did not use any separator; by default, the `split` function splits on white spaces.

N-gram representation

We have seen how text can be represented as characters and words. Sometimes, it is useful to look at two, three, or more words together. **N-grams** are groups of words extracted from the given text. In an n-gram, n represents the number of words that can be used together. Let's look at an example of what a bigram ($n=2$) looks like. We used the Python `nltk` package to generate a bigram for `toy_story_review`. The following code block shows the result of the bigram and the code used to generate it:

```
| from nltk import ngrams  
| print(list(ngrams(toy_story_review.split(), 2)))
```

This results in the following output:

```
[('Just', 'perfect.'), ('perfect.', 'Script,'), ('Script,',  
'character,'), ('character,', 'animation....this'), ('animation....this',  
'manages'), ('manages', 'to'), ('to', 'break'), ('break', 'free'),  
('free', 'of'), ('of', 'the'), ('the', 'yoke'), ('yoke', 'of'), ('of',  
'children's'), ("children's", "movie'"), ("movie'", 'to'), ('to',  
'simply'), ('simply', 'be'), ('be', 'one'), ('one', 'of'), ('of', 'the'),  
('the', 'best'), ('best', 'movies'), ('movies', 'of'), ('of', 'the'),  
('the', "90's,"), ("90's,", 'full-stop.')]
```

The `ngrams` function accepts a sequence of words as its first argument and the number of words to be grouped as the second argument. The following code block shows how a trigram representation would look, and the code used for it:

```
| print(list(ngrams(toy_story_review.split(), 3)))
```

This results in the following output:

```
[('Just', 'perfect.', 'Script,'), ('perfect.', 'Script,', 'character,'),  
('Script,', 'character,', 'animation....this'), ('character,',  
'animation....this', 'manages'), ('animation....this', 'manages', 'to'),  
('manages', 'to', 'break'), ('to', 'break', 'free'), ('break', 'free',  
'of'), ('free', 'of', 'the'), ('of', 'the', 'yoke'), ('the', 'yoke',  
'of'), ('yoke', 'of', "children's"), ('of', "children's", "movie'"),  
("children's", "movie'", 'to'), ("movie'", 'to', 'simply'), ('to',  
'simply', 'be'), ('simply', 'be', 'one'), ('be', 'one', 'of'), ('one',
```

```
| 'of', 'the'), ('of', 'the', 'best'), ('the', 'best', 'movies'), ('best',
| 'movies', 'of'), ('movies', 'of', 'the'), ('of', 'the', "90's"), ('the',
| "90's," , 'full-stop.')]
```

The only thing that changed in the preceding code is the n value, the second argument to the function.

Many supervised machine learning models, for example, Naive Bayes, use n-grams to improve their feature space. N-grams are also used for spelling correction and text summarization tasks.

One challenge with n-gram representation is that it loses the sequential nature of text. It is often used with shallow machine learning models. This technique is rarely used in deep learning, as architectures such as RNN and Conv1D, learn these representations automatically.

Vectorization

There are two popular approaches to mapping the generated tokens to vectors of numbers, called one-hot encoding and word embedding. Let's understand how tokens can be converted to these vector representations by writing a simple Python program. We will also discuss the various pros and cons of each method.

One-hot encoding

In one-hot encoding, each token is represented by a vector of length N , where N is the size of the vocabulary. The vocabulary is the total number of unique words in the document. Let's take a simple sentence and observe how each token would be represented as one-hot encoded vectors. The following is the sentence and its associated token representation:

An apple a day keeps doctor away said the doctor.

One-hot encoding for the preceding sentence can be represented in a tabular format as follows:

An	100000000
apple	10000000
a	1000000
day	100000
keeps	10000
doctor	1000
away	100
said	10
the	1

This table describes the tokens and their one-hot encoded representation. The vector length is nine, as there are nine unique words in the sentence. A lot of machine learning libraries have eased the process of creating one-hot encoding variables. We will write our own implementation to make it easier to understand, and we can use

the same implementation to build other features required for later examples. The following code contains a `Dictionary` class, which contains functionality to create a dictionary of unique words along with a function to return a one-hot encoded vector for a particular word. Let's take a look at the code and then walk through each functionality:

```
class Dictionary(object):
    def __init__(self):
        self.word2index = {}
        self.index2word = []
        self.length = 0

    def add_word(self, word):
        if word not in self.index2word:
            self.index2word.append(word)
            self.word2index[word] = self.length + 1
            self.length += 1
        return self.word2index[word]

    def len(self):
        return len(self.index2word)

    def onehot_encoded(self, word):
        vec = np.zeros(self.length)
        vec[self.word2index[word]] = 1
        return vec
```

The preceding code provides three important functionalities:

- The initialization, `__init__`, function creates a `word2index` dictionary, which will store all unique words along with the index. The `index2word` list stores all the unique words and the `length` variable contains the total number of unique words in our document.
- The `add_word` function takes a word and adds it to `word2index` and `index2word`, and increases the length of the vocabulary, provided the word is unique.
- The `onehot_encoded` function takes a word and returns a vector of length N with zeros throughout, except at the index of the word. If the index of the passed word is two, then the value of the vector at index two will be one, and all the remaining values will be zeros.

As we have defined our `Dictionary` class, let's use it on our `toy_story_review` data. The following code demonstrates how the `word2index` dictionary is built and how we can call our `onehot_encoded` function:

```
| dic = Dictionary()  
| for tok in toy_story_review.split(): dic.add_word(tok)  
| print(dic.word2index)
```

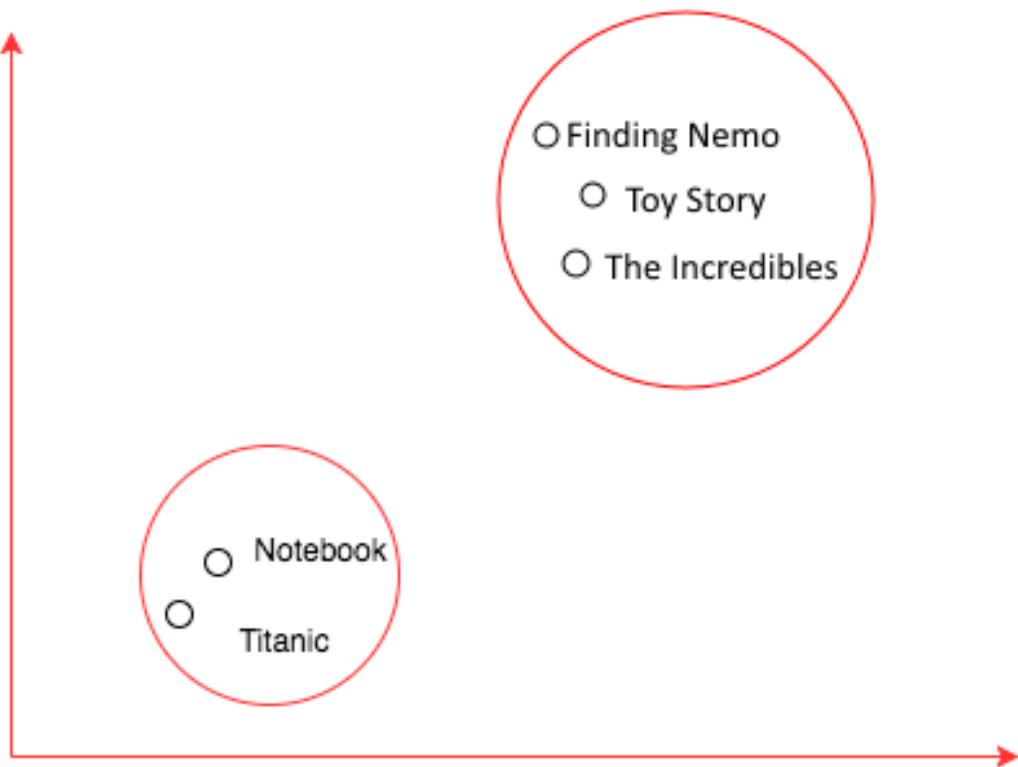
One of the challenges with one-hot representation is that the data is too sparse, and the size of the vector quickly grows as the number of unique words in the vocabulary increases. Another limitation is that one-hot has no representation of internal relations between words. For these reasons, one-hot is rarely used with deep learning.

Word embedding

Word embedding is a very popular way of representing text data in problems that are solved by deep learning algorithms. Word embedding provides a dense representation of a word filled with floating numbers. The vector dimension varies according to the vocabulary size. It is common to use a word embedding of dimension size 50, 100, 256, 300, and sometimes 1,000. The dimension size is a hyperparameter that we need to play with during the training phase.

If we are trying to represent a vocabulary of size 20,000 in one-hot representation, then we will end up with $20,000 \times 20,000$ numbers, most of which will be zero. The same vocabulary can be represented in a word embedding as $20,000 \times \text{dimension size}$, where the dimension size could be 10, 50, 300, and so on.

One way to create word embeddings is to start with dense vectors for each token containing random numbers and then train a model, such as a document classifier, for sentiment classification. The floating point numbers, which represent the tokens, will get adjusted in such a way that semantically closer words will have similar representation. To understand it, let's look at the following diagram, where we plotted the word embedding vectors on a two-dimensional plot of five movies:



The preceding diagram shows how the dense vectors are tuned in order to have smaller distances for words that are semantically similar. Since movie titles such as *Finding Nemo*, *Toy Story*, and *The Incredibles* are fictional movies with cartoons, the embedding for such words is closer. On the other hand, the embedding for the movie *Titanic* is far from the cartoons and closer to the movie title *Notebook*, since they are romantic movies.

Learning word embedding may not be feasible when you have too little data, and in those cases, we can use word embeddings that are trained by some other machine learning algorithm. An embedding generated from another task is called a pretrained word embedding. We will learn how to build our own word embeddings and use pretrained word embeddings.

Training word embedding by building a sentiment classifier

In the last section, we briefly learned about word embedding without implementing it. In this section, we will download a dataset called IMDb, which contains reviews, and build a sentiment classifier that calculates whether a review's sentiment is positive, negative, or unknown. In the process of building, we will also train word embedding for the words present in the IMDb dataset.

We will use a library called `torchtext`, which makes a lot of processes such as downloading, text vectorization, and batching, much easier. Training a sentiment classifier will involve the following steps:

1. Downloading IMDb data and performing text tokenization
2. Building a vocabulary
3. Generating batches of vectors
4. Creating a network model with embeddings
5. Training the model

We shall see these steps in more detail in the following sections.

Downloading IMDb data and performing text tokenization

For applications related to computer vision, we used the `torchvision` library, which provides us with a lot of utility functions, helping to build computer vision applications. In the same way, there is a library called `torchtext`, which is built to work with PyTorch and eases a lot of activities related to **natural language processing (NLP)** by providing different data loaders and abstractions for text. At the time of writing, `torchtext` does not come with the standard PyTorch installation and requires a separate installation. You can run the following code in the command line of your machine to get `torchtext` installed:

```
| pip install torchtext
```

Once it is installed, we will be able to use it. The `torchtext` download provides two important modules called `torchtext.data` and `torchtext.datasets`.



We can download the IMDb Movies dataset from the following link:
<https://grouplens.org/datasets/movielens/>.

Tokenizing with torchtext.data

The `torchtext.data` instance defines a `Field` class, which helps us to define how the data has to be read and tokenized. Let's look at the following example, which we will use for preparing our IMDb dataset:

```
| from torchtext import data
| text = data.Field(lower=True, batch_first=True, fix_length=20)
| label = data.Field(sequential=False)
```

In the preceding code, we define two `Field` objects, one for actual text and another for the label data. For actual text, we expect `torchtext` to lowercase all the text, tokenize the text, and trim it to a maximum length of 20. If we are building an application for a production environment, we may fix the length to a much larger number. However, for the toy example, this works well. The `Field` constructor also accepts another argument called `tokenize`, which by default uses the `str.split` function. We can also specify `spaCy` as the argument, or any other tokenizer. For our example, we will stick with `str.split`.

Tokenizing with torchtext.datasets

The `torchtext.datasets` instance provides wrappers for using different datasets such as IMDb, TREC (question classification), language modeling (WikiText-2), and a few other datasets. We will use `torch.datasets` to download the IMDb dataset and split it into train and test datasets. The following code does that and when you run it for the first time, it could take several minutes (depending on your broadband connection) as it downloads the IMDb dataset from the internet:

```
|train, test = datasets.IMDB.splits(text, label)
```

The previous dataset's `IMDB` class abstracts away all the complexity involved in downloading, tokenizing, and splitting the database into train and test datasets. The `train.fields` download contains a dictionary where `TEXT` is the key and `LABEL` is the value. Let's look at `train.fields` and what each element of `train` contains:

```
|print('train.fields', train.fields)
```

This results in the following output:

```
#Results
train.fields {'text': <torchtext.data.field.Field object at 0x1129db160>,
'label': <torchtext.data.field.Field object at 0x1129db1d0>}
```

Similarly, the variance for the training dataset is as follows:

```
|print(vars(train[0]))
```

This results in the following output:

```
#Results
vars(train[0]) {'text': ['for', 'a', 'movie', 'that', 'gets', 'no',
'respect', 'there', 'sure', 'are', 'a', 'lot', 'of', 'memorable',
'quotes', 'listed', 'for', 'this', 'gem.', 'imagine', 'a', 'movie',
```

```
'where', 'joe', 'piscopo', 'is', 'actually', 'funny!', 'maureen',
'stapleton', 'is', 'a', 'scene', 'stealer.', 'the', 'moroni',
'character', 'is', 'an', 'absolute', 'scream.', 'watch', 'for', 'alan',
'"the', 'skipper"', 'hale', 'jr.', 'as', 'a', 'police', 'sgt.'], 'label':
'pos'})
```

We can see from these results that a single element contains a field and text, along with all the tokens representing the text, and a `label` field that contains the label of the text. Now we have the IMDb dataset ready for batching.

Building vocabulary

When we created one-hot encoding for `toy_story_review`, we created a `word2index` dictionary, which is referred to as the vocabulary since it contains all the details of the unique words in the documents. The `torchtext` instance makes that easier for us. Once the data is loaded, we can call `build_vocab` and pass the necessary arguments that will take care of building the vocabulary for the data. The following code shows how the vocabulary is built:

```
| text.build_vocab(train, vectors=GloVe(name='6B',
| dim=300), max_size=10000, min_freq=10)
| label.build_vocab(train)
```

In the preceding code, we pass in the `train` object on which we need to build the vocabulary, and we also ask it to initialize vectors with pretrained embeddings of dimensions `300`. The `build_vocab` object just downloads and creates the dimension that will be used later when we train the sentiment classifier using pretrained weights. The `max_size` instance limits the number of words in the vocabulary, and `min_freq` removes any word that has not occurred more than 10 times, where `10` is configurable.

Once the vocabulary is built, we can obtain different values such as frequency, word index, and the vector representation for each word. The following code demonstrates how to access these values:

```
| print(text.vocab.freqs)
```

This results in the following output:

```
# A sample result
Counter({'i'm': 4174,
         'not': 28597,
         'tired': 328,
         'to': 133967,
         'say': 4392,
         'this': 69714,
         'is': 104171,
```

```
'one': 22480,  
'of': 144462,  
'the': 322198,
```

The following code demonstrates how to access the results:

```
| print(text.vocab.vectors)
```

This results in the following output:

```
#Results displaying the 300 dimension vector for each word.  
0.0000 0.0000 0.0000 ... 0.0000 0.0000 0.0000  
0.0000 0.0000 0.0000 ... 0.0000 0.0000 0.0000  
0.0466 0.2132 -0.0074 ... 0.0091 -0.2099 0.0539  
...  
0.0000 0.0000 0.0000 ... 0.0000 0.0000 0.0000  
0.7724 -0.1800 0.2072 ... 0.6736 0.2263 -0.2919  
0.0000 0.0000 0.0000 ... 0.0000 0.0000 0.0000  
[torch.FloatTensor of size 10002x300]
```

Similarly, we will print the values for a dictionary containing words and their indexes as follows:

```
| print(TEXT.vocab.stoi)
```

This results in the following output:

```
# Sample results  
defaultdict(<function torchtext.vocab._default_unk_index>,  
{'<unk>': 0,  
'<pad>': 1,  
'the': 2,  
'a': 3,  
'and': 4,  
'of': 5,  
'to': 6,  
'is': 7,  
'in': 8,  
'i': 9,  
'this': 10,  
'that': 11,  
'it': 12,}
```

The `stoi` value gives access to a dictionary containing words and their indexes.

Generating batches of vectors

The `torchtext` download provides `BucketIterator`, which helps in batching all the text and replacing the words with the index number of the words. The `BucketIterator` instance comes with a lot of useful parameters such as `batch_size`, `device` (GPU or CPU), and `shuffle` (whether data has to be shuffled). The following code demonstrates how to create iterators that generate batches for the train and test datasets:

```
| train_iter, test_iter = data.BucketIterator.splits((train, test),
| batch_size=128, device=-1, shuffle=True)
| #device = -1 represents cpu , if you want gpu leave it to None.
```

The preceding code gives a `BucketIterator` object for both train and test datasets. The following code shows how to create a batch and display the results of the batch:

```
| batch = next(iter(train_iter)) batch.text
```

This results in the following output:

```
#Results
Variable containing:
 5128 427 19 ... 1688 0 542
 58 2 0 ... 2 0 1352
 0 9 14 ... 2676 96 9
 ...
 129 1181 648 ... 45 0 2
 6484 0 627 ... 381 5 2
 748 0 5052 ... 18 6660 9827
 [torch.LongTensor of size 128x20]
```

We will print the labels as follows:

```
| batch.label
```

This results in the following output:

```
#Results
Variable containing:
 2
```

```
| 1  
| 2  
| 1  
| 2  
| 1  
| 1  
| 1  
[torch.LongTensor of size 128]
```

From the results in the preceding code block, we can see how the text data is converted into a matrix of size $(\text{batch_size} * \text{fix_len})$, which is (128×20) .

Creating a network model with embedding

We discussed word embeddings briefly earlier. In this section, we create word embeddings as part of our network architecture and train the entire model to predict the sentiment of each review. At the end of the training, we will have a sentiment classifier model and also the word embeddings for the IMDB datasets. The following code demonstrates how to create a network architecture to predict the sentiment using word embeddings:

```
class EmbeddingNetwork(nn.Module):
    def __init__(self, emb_size, hidden_size1, hidden_size2=400):
        super().__init__()
        self.embedding = nn.Embedding(emb_size, hidden_size1)
        self.fc = nn.Linear(hidden_size2, 3)
    def forward(self, x):
        embeds = self.embedding(x).view(x.size(0), -1)
        out = self.fc(embeds)
        return F.log_softmax(out, dim=-1)
```

In the preceding code, `EmbeddingNetwork` creates the model for sentiment classification. Inside the `__init__` function, we initialize an object of the `nn.Embedding` class, which takes two arguments, namely, the size of the vocabulary and the dimensions that we wish to create for each word. As we have limited the number of unique words, the vocabulary size will be 10,000 and we can start with a small embedding size of 10. For running the program quickly, a small embedding size is useful, but when you are trying to build applications for production systems, use embeddings of a large size. We also have a linear layer that maps the word embeddings to the category (positive, negative, or unknown).

The forward function determines how the input data is processed. For a batch size of 32 and sentences of a maximum length of 20 words, we will have inputs of the shape 32 x 20. The first embedding

layer acts as a lookup table, replacing each word with the corresponding embedding vector. For an embedding dimension of 10, the output becomes $32 \times 20 \times 10$ as each word is replaced with its corresponding embedding. The `view()` function will flatten the result from the embedding layer. The first argument passed to `view` will keep that dimension intact.

In our case, we do not want to combine data from different batches, so we preserve the first dimension and flatten the rest of the values in the tensor. After the `view()` function is applied, the tensor shape changes to 32×200 . A dense layer maps the flattened embeddings to the number of categories. Once the network is defined, we can train the network as usual.



Remember that in this network, we lose the sequential nature of the text and we just use the text as a bag of words.

Training the model

Training the model is very similar to what we saw for building image classifiers, so we will be using the same functions. We pass batches of data through the model, calculate the outputs and losses, and then optimize the model weights, which includes the embedding weights. The following code does this:

```
def fit(epoch,model,data_loader,phase='training',volatile=False):
    if phase == 'training':
        model.train()
    if phase == 'validation':
        model.evaluation()
volatile=True
running_loss = 0.0
running_correct = 0
```

Now we iterate over the dataset:

```
for batch_idx , batch in enumerate(data_loader):
    text, target = batch.text , batch.label
    if is_cuda:
        text,target = text.cuda(),target.cuda()
    if phase == 'training':
        optimizer.zero_grad()
        output = model(text)
        loss = F.nll_loss(output,target)
        running_loss += F.nll_loss(output,target,size_average=False).data[0]
        predictions = output.data.max(dim=1,keepdim=True)[1]
        running_correct +=
predictions.eq(target.data.view_as(predictions)).cpu().sum()
    if phase == 'training':
        loss.backward()
        optimizer.step()
        loss = running_loss/len(data_loader.dataset)
        accuracy = 100. * running_correct/len(data_loader.dataset)
        print(f'{phase} loss is {loss:.2f} and {phase} accuracy is
{running_correct}/{len(data_loader.dataset)}{accuracy:.10}.
{4}}').format(loss,accuracy)
```

From here, we can train the model over each epoch:

```
train_losses , train_accuracy = [],[]
validation_losses , validation_accuracy = [],[]

train_iter.repeat = False
```

```
test_iter.repeat = False
for epoch in range(1,10):
    epoch_loss, epoch_accuracy =
fit(epoch,model,train_iter,phase='training')
    validation_epoch_loss, validation_epoch_accuracy =
fit(epoch,model,test_iter,phase='validation')
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    validation_losses.append(validation_epoch_loss)
    validation_accuracy.append(validation_epoch_accuracy)
```

In the preceding code, we call the `fit` method by passing the `BucketIterator` object that we created for batching the data. The iterator, by default, does not stop generating batches, so we have to set the `repeat` variable of the `BucketIterator` object to `False`. If we don't set the `repeat` variable to `False`, then the `fit` function will run indefinitely. Training the model for around 10 epochs gives a validation accuracy of approximately 70%. Now that you've learned how to train word embedding by building sentiment classifier, let us learn how to use pretrained word embeddings in the next section.

Using pretrained word embeddings

Pretrained word embeddings are useful when we are working in specific domains, such as medicine and manufacturing, where we have a lot of data to train the embeddings. When we have little data and we cannot meaningfully train the embeddings, we can use embeddings that are trained on different data corpuses such as Wikipedia, Google News, and Twitter tweets. A lot of teams have open source word embeddings trained using different approaches. In this section, we will explore how `torchtext` makes it easier to use different word embeddings, and how to use them in our PyTorch models. It is similar to transfer learning, which we use in computer vision applications. Typically, using pretrained embedding would involve the following steps:

1. Downloading the embeddings
2. Loading the embeddings in the model
3. Freezing the embedding layer weights

Let's explore in detail how each step is implemented.

Downloading the embeddings

The `torchtext` library abstracts away a lot of complexity involved in downloading the embeddings and mapping them to the right word. The `torchtext` library provides three classes in the `vocab` module, namely, GloVe, FastText, CharNGram, which ease the process of downloading embeddings and mapping them to our vocabulary. Each of these classes provides different embeddings trained on different datasets and using different techniques. Let's look at some of the different embeddings provided:

- `charngram.100d`
- `fasttext.en.300d`
- `fasttext.simple.300d`
- `glove.42B.300d`
- `glove.840B.300d`
- `glove.twitter.27B.25d`
- `glove.twitter.27B.50d`
- `glove.twitter.27B.100d`
- `glove.twitter.27B.200d`
- `glove.6B.50d`

- `glove.6B.100d`
- `glove.6B.200d`
- `glove.6B.300d`

The `build_vocab` method of the `Field` object takes in an argument for the embeddings. The following code explains how we download the embeddings:

```
from torchtext.vocab import GloVe
TEXT.build_vocab(train, vectors=GloVe(name='6B',
dim=300), max_size=10000, min_freq=10)
LABEL.build_vocab(train,)
```

The value to the argument vector denotes what embedding class is to be used. The `name` and `dim` arguments determine which embeddings can be used. We can easily access the embeddings from the `vocab` object. The following code demonstrates it, along with a view of how the results will look:

```
| TEXT.vocab.vectors
```

This results in the following output:

```
| #Output
| 0.0000 0.0000 0.0000 ... 0.0000 0.0000 0.0000
| 0.0000 0.0000 0.0000 ... 0.0000 0.0000 0.0000
| 0.0466 0.2132 -0.0074 ... 0.0091 -0.2099 0.0539
| ...
| [torch.FloatTensor of size 10002x300]
```

Now we have downloaded and mapped the embeddings to our vocabulary. Let's understand how we can use them with a PyTorch model.

Loading the embeddings in the model

The `vectors` variable returns a torch tensor of shape `vocab_size x dimensions` containing the pretrained embeddings. We have to store the embeddings to the weights of our embedding layer. We can assign the weights of the embeddings by accessing the weights of the embeddings layer as demonstrated by the following code:

```
| model.embedding.weight.data = TEXT.vocab.vectors
```

The `model` download represents the object of our network, and `embedding` represents the embedding layer. As we are using the embedding layer with new dimensions, there will be a small change in the input to the linear layer that comes after the embedding layer. The following code has the new architecture, which is similar to the previously used architecture where we trained our embeddings:

```
class EmbeddingNetwork(nn.Module):
    def __init__(self, embedding_size, hidden_size1, hidden_size2=400):
        super().__init__()
        self.embedding = nn.Embedding(embedding_size, hidden_size1)
        self.fc1 = nn.Linear(hidden_size2, 3)

    def forward(self, x):
        embeds = self.embedding(x).view(x.size(0), -1)
        out = self.fc1(embeds)
        return F.log_softmax(out, dim=-1)

model = EmbeddingNetwork(len(TEXT.vocab.stoi), 300, 12000)
```

Once the embeddings are loaded, we have to ensure that, during training, we do not change the embedding weights. Let's discuss how to achieve that.

Freezing the embedding layer weights

It is a two-step process to tell PyTorch not to change the weights of the embedding layer:

1. Set the `requires_grad` attribute to `False`, which instructs PyTorch that it does not need gradients for these weights.
2. Remove the passing of the embedding layer parameters to the optimizer. If this step is not done, then the optimizer throws an error, as it expects all the parameters to have gradients.

The following code demonstrates how easy it is to freeze the embedding layer weights and instruct the optimizer not to use those parameters:

```
| model.embedding.weight.requires_grad = False  
| optimizer = optim.SGD([ param for param in model.parameters() if  
| param.requires_grad == True], lr=0.001)
```

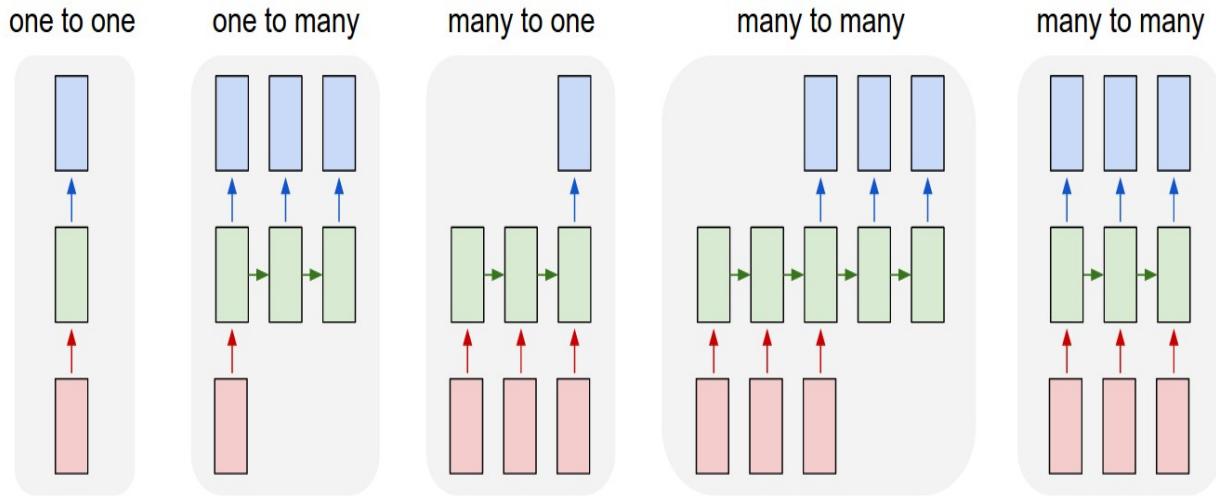
We generally pass all the model parameters to the optimizer, but in the previous code, we passed parameters that have `requires_grad` as `True`.

We can train the model using this exact code and should achieve similar accuracy. All these model architectures fail to take advantage of the sequential nature of the text. In the next section, we explore two popular techniques, namely, RNN and Conv1D, which take advantage of the sequential nature of the data.

Recursive neural networks

RNNs are among the most powerful models that enable us to take on applications such as classification, labeling of sequential data, generating sequences of text (such as with the SwiftKey Keyboard app, which predicts the next word), and converting one sequence to another, such as when translating a language (for example, from French to English). Most of the model architectures, such as feedforward neural networks, do not take advantage of the sequential nature of data. For example, we need the data to present the features of each example in a vector, say all the tokens that represent a sentence, paragraph, or documents. Feedforward networks are designed just to look at all the features once and map them to output. Let's look at a text example that shows why the order, or sequential nature of text, is important. *I had cleaned my car* and *I had my car cleaned* are two English sentences with the same set of words, but they mean different things when we consider the order of the words.

In most modern languages, humans make sense of text data by reading words from left to right and building a powerful model that kind of understands all the different things the text says. RNN works similarly by looking at one word in text at a time. RNN is also a neural network that has a special layer in it, which loops over the data instead of processing all at once. As RNNs can process data in sequence, we can use vectors of different lengths and generate outputs of different lengths. Some of the different representations are provided in the following diagram:

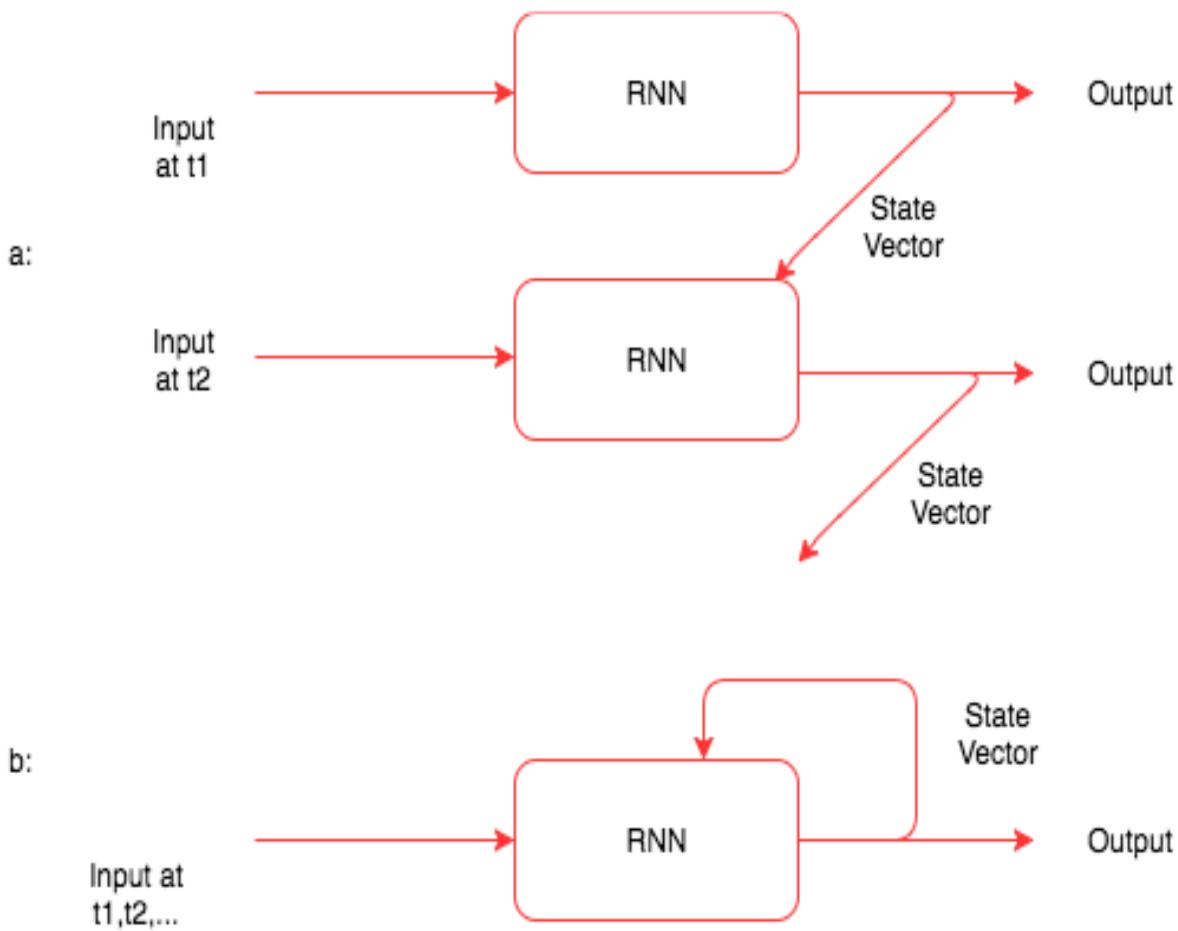


The previous diagram is from one of the famous blogs on RNN (<http://karpathy.github.io/2015/05/21/rnn-effectiveness>), in which the author, Andrej Karpathy, writes about how to build an RNN from scratch with Python and using it as a sequence generator.

Understanding how RNN works with an example

Let's start with the assumption that we have an RNN model already built and let's try to understand what functionality it provides. Once we understand what an RNN does, let's then explore what happens inside an RNN.

Let's consider the *Toy Story* review as input to the RNN model. The example text we are looking at is *Just perfect. Script, character, animation....this manages to break free.....* We start by passing the first word, *just* to our model, and the model generates two different things: a **State Vector** and an **Output** vector. The **State Vector** is passed to the model when it processes the next word in the review, and a new **State Vector** is generated. We just consider the **Output** of the model generated during the last sequence. The following diagram summarizes it:



The preceding diagram demonstrates the following:

- How RNN works by unfolding the text input and the image
- How the state is recursively passed to the same model

By now, you will have an idea of what RNN does, but not how it works. Before we get into how it works, let's look at a code snippet that showcases in more detail what we have learned. We will still view RNN as a black box:

```
rnn = RNN(input_size, hidden_size, output_size)
for i in range(len(toy_story_review)):
    output, hidden = rnn(toy_story_review[i], hidden)
```

In the preceding code, the `hidden` variable represents the state vector, sometimes called the **hidden state**. By now, we should have an idea

of how RNN is used. Now, let's look at the code that implements RNN and understand what happens inside the RNN. The following code contains the `RNN` class:

```
import torch.nn as nn
from torch.autograd import Variable

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

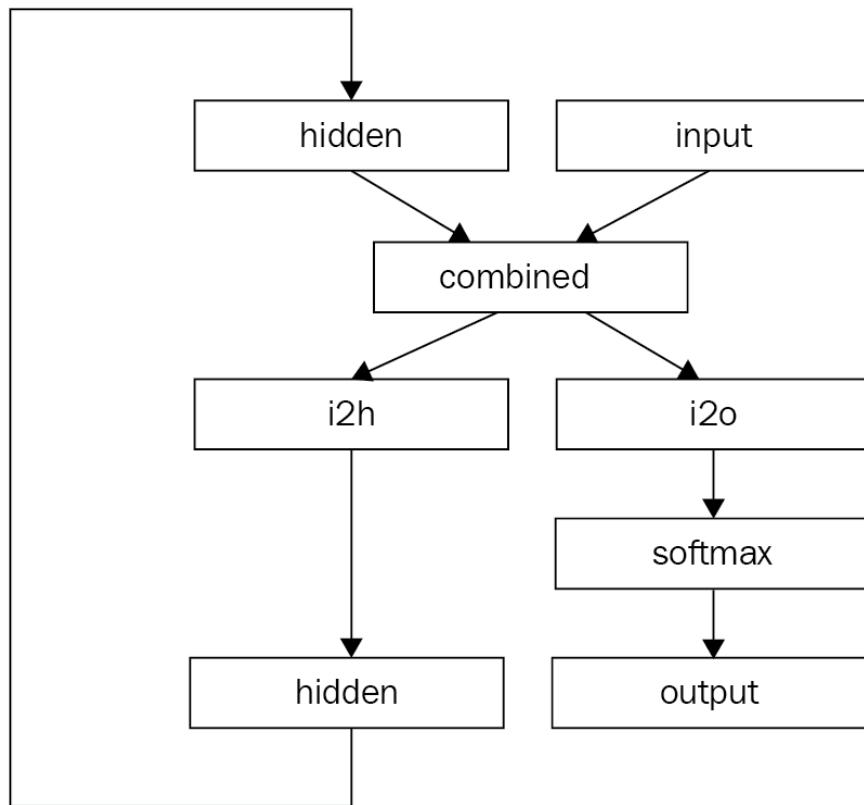
    def initHidden(self):
        return Variable(torch.zeros(1, self.hidden_size))
```

Except for the word RNN in the preceding code, everything else would sound pretty similar to what we have used in the previous chapters, as PyTorch hides a lot of complexity of backpropagation. Let's walk through the `__init__` function and the `forward` function to understand what is happening.

The `__init__` function initializes two linear layers, one for calculating the output and the other for calculating the state or hidden vector.

The `forward` function combines the input vector and the hidden vector and passes it through the two linear layers, which generates an output vector and a hidden state. For the output layer, we apply a `log_softmax` function.

The `initHidden` function helps in creating hidden vectors with no state for calling RNN the very first time. Let's take a visual look into what the RNN class does in the following diagram:



The preceding diagram shows how an RNN works.



The concepts of RNN are sometimes tricky to understand when you meet them for the first time, so I would strongly recommend some of the amazing blogs provided in the following links: <http://karpathy.github.io/2015/05/21/rnn-effectiveness/> and <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>.

In the next section, we will learn how to use a variant of RNN called LSTM to build a sentiment classifier on the IMDB dataset.

Solving text classification problem using LSTM

RNNs are quite popular in building real-world applications, such as language translation, text classification, and many more sequential problems. However, in reality, we would rarely use a vanilla version of RNN, such as the one we saw in the previous section. The vanilla version of RNN has problems, such as vanishing gradients and gradient explosion when dealing with large sequences. In most of the real-world problems, variants of RNN such as LSTM or GRU are used, which solve the limitations of plain RNN and also have the ability to handle sequential data better. We will try to understand what happens in LSTM and build a network based on LSTM to solve the text classification problem on the IMDB datasets.

Long-term dependency

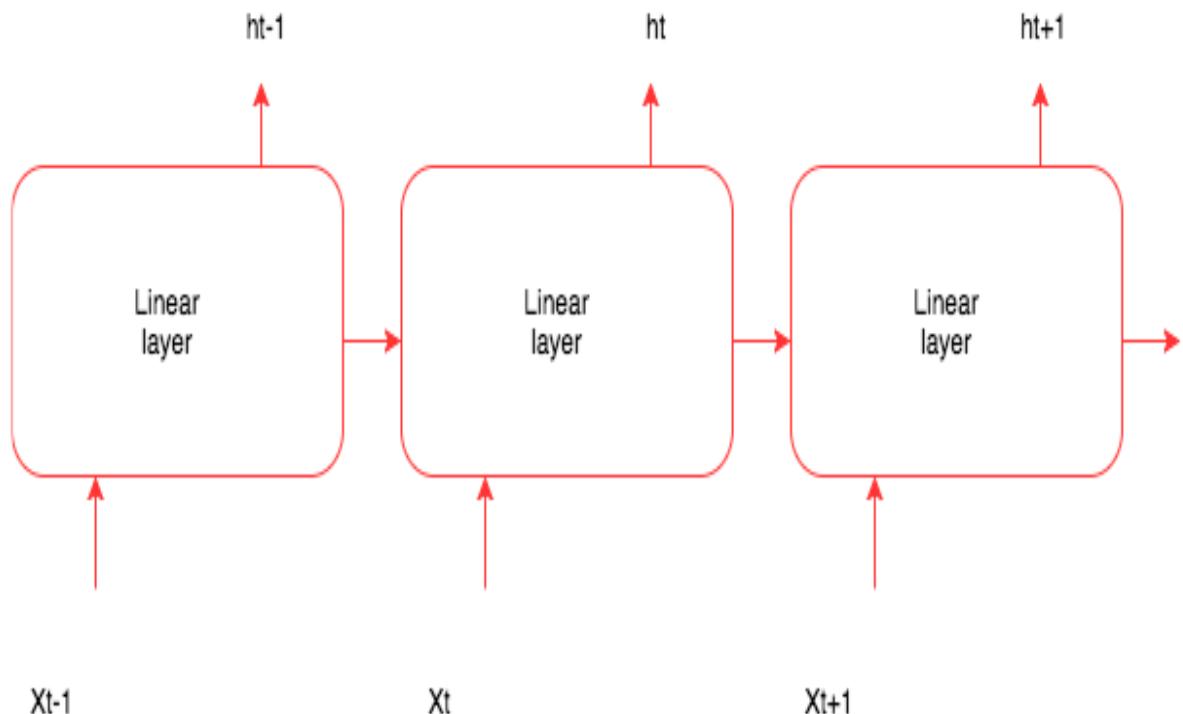
RNNs, in theory, should learn all the dependency required from the historical data to build a context of what happens next. Say, for example, we are trying to predict the last word in the sentence *The clouds are in the sky*. RNN would be able to predict it, as the information (clouds) is just a few words behind. Let's take another long paragraph where the dependency need not be that close, and we want to predict the last word in it. The sentence is: *I am born in Chennai a city in Tamilnadu. Did schooling in different states of India and I speak...* The vanilla version of RNN, in practice, finds it difficult to remember the context that happened in the earlier parts of sequences. LSTMs, and other different variants of RNN, solve this problem by adding different neural networks inside the LSTM, which later decide how much, or what data can be remembered.

LSTM networks

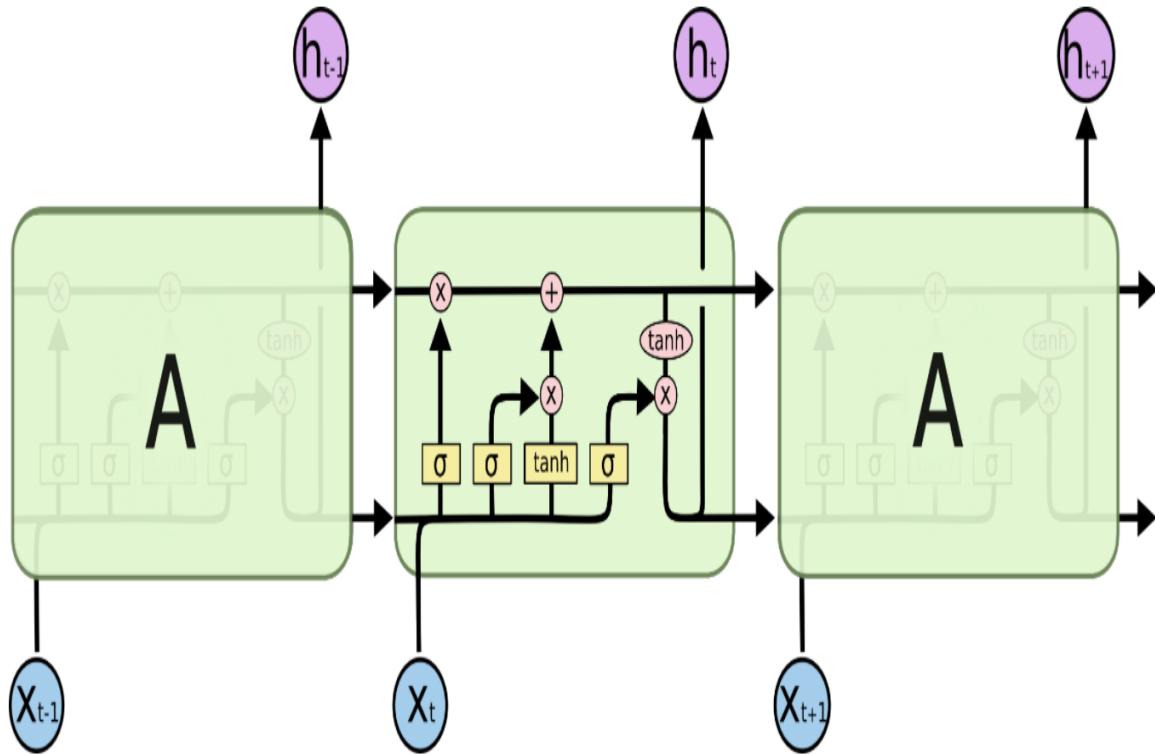
LSTMs are a special kind of RNN, capable of learning long-term dependency. They were introduced in 1997 and got popular in the last few years with advancements in available data and hardware. They work tremendously well on a large variety of problems and are widely used.

LSTMs are designed to avoid long-term dependency problems by having a design by which it is natural to remember information for a long period of time. In RNNs, we saw how they repeat themselves over each element of the sequence. In standard RNNs, the repeating module will have a simple structure like a single linear layer.

The following diagram shows how a simple RNN repeats itself:



Inside LSTM, instead of using a simple linear layer, we have smaller networks inside the LSTM, which do an independent job. The following diagram showcases what happens inside an LSTM:



The repeating module in an LSTM contains four interacting layers.

Image source: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/> (diagram by Christopher Olah)

Each of the small rectangular (yellow) boxes in the second box in the preceding diagram represent a PyTorch layer, the circles represent an element matrix or vector addition, and the merging lines represent that the two vectors are being concatenated. The good part is, we need not implement all of this manually. Most of the modern deep learning frameworks provide an abstraction that will take care of what happens inside an LSTM. PyTorch provides abstraction of all the functionality inside the `nn.LSTM` layer, which we can use like any other layer.

The most important thing in the LSTM is the cell state that passes through all the iterations, represented by the horizontal line across the cells in the preceding diagram. Multiple networks inside LSTM control what information travels across the cell state. The first step in LSTM (a small network represented by the symbol σ) is to decide what information is going to be thrown away from the cell state. This network is called a **forget gate** and has a sigmoid as an activation function, which outputs values between 0 and 1 for each element in the cell state. The network (PyTorch layer) is represented using the following formula:

$$f_t = \sigma(W_f \cdot [h_{t-1}, X_t] + b_f)$$

The values from the network decide which values are to be held in the cell state and which are to be thrown away. The next step is to decide what information we are going to add to the cell state. This has two parts; a sigmoid layer, called the input gate, which decides what values to be updated, and a *tanh* layer, which creates new values to be added to the cell state. The mathematical representation looks like this:

$$\begin{aligned} i_t &= \sigma(W_i \cdot [h_{t-1}, X_t] + b_i) \\ \dot{C}_t &= \tanh(W_{\dot{C}} \cdot [h_{t-1}, X_t] + b_{\dot{C}}) \end{aligned}$$

In the next step, we combine the two values generated by the input gate and *tanh*. Now we can update the cell state by doing an element-wise multiplication between the forget gate and the sum of the product of it and C_t , as represented by the following formula:

$$C_t = f_t * C_t + i_t * \dot{C}_t$$

Finally, we need to decide on the output, which will be a filtered version of the cell state. There are different versions of LSTM available and most of them work on similar principles. As developers

or data scientists, we rarely need to worry about what goes on inside LSTM.

If you want to learn more about them, go through the following blog links, which cover a lot of theory in a very intuitive way.



Look at Christopher Olah's amazing blog on LSTM (<http://colah.github.io/posts/2015-08-Understanding-LSTMs>), and another blog from Brandon Rohrer (https://brohrer.github.io/how_rnns_lstm_work.html) where he explains LSTM in a nice video.

Since we understand LSTM, let's implement a PyTorch network that we can use to build a sentiment classifier. As usual, we will follow these steps for creating the classifier:

1. Preparing the data
2. Creating the batches
3. Creating the network
4. Training the model

We will see the steps in detail in the following section.

Preparing the data

We use the same `torchtext` library for downloading, tokenizing, and building vocabulary for the IMDB dataset. When creating the `Field` object, we leave the `batch_first` argument at `False`. RNNs expect the data to be in the form of `sequence_length`, `batch_size`, and `features`. The following is used for preparing the dataset:

```
TEXT = data.Field(lower=True, fix_length=200, batch_first=False)
LABEL = data.Field(sequential=False, )
train, test = IMDB.splits(TEXT, LABEL)
TEXT.build_vocab(train, vectors=GloVe(name='6B',
dim=300), max_size=10000, min_freq=10)
LABEL.build_vocab(train, )
```

Creating batches

We use the `torchtext BucketIterator` function for creating batches, and the size of the batches will be sequence length and batches. For our case, the size will be [200, 32], where 200 is the sequence length and 32 is the batch size.

The following is the code used for batching:

```
| train_iter, test_iter = data.BucketIterator.splits((train, test),  
| batch_size=32, device=-1)  
| train_iter.repeat = False  
| test_iter.repeat = False
```

Creating the network

Let's look at the code and then walk through it. You may be surprised at how familiar the code looks:

```
class IMDBRnn(nn.Module):

    def __init__(self, vocab, hidden_size, n_cat, bs=1, nl=2):
        super().__init__()
        self.hidden_size = hidden_size
        self.bs = bs
        self.nl = nl
        self.e = nn.Embedding(n_vocab, hidden_size)
        self.rnn = nn.LSTM(hidden_size, hidden_size, nl)
        self.fc2 = nn.Linear(hidden_size, n_cat)
        self.softmax = nn.LogSoftmax(dim=-1)

    def forward(self, inp):
        bs = inp.size()[1]
        if bs != self.bs:
            self.bs = bs
        e_out = self.e(inp)
        h0 = c0 = Variable(e_out.data.new(*
(self.nl, self.bs, self.hidden_size)).zero_())
        rnn_o, _ = self.rnn(e_out, (h0, c0))
        rnn_o = rnn_o[-1]
        fc = F.dropout(self.fc2(rnn_o), p=0.8)
        return self.softmax(fc)
```

The `__init__` method creates an embedding layer of the size of the vocabulary and `hidden_size`. It also creates an LSTM and a linear layer. The last layer is a `LogSoftmax` layer for converting the results from the linear layer to probabilities.

In the `forward` function, we pass the input data of size [200, 32], which gets passed through the embedding layer and each token in the batch gets replaced by embedding and the size turns to [200, 32, 100], where 100 is the embedding dimensions. The LSTM layer takes the output of the embedding layer along with two hidden variables. The hidden variables should be the same type of the embeddings output, and their size should be [`num_layers, batch_size, hidden_size`]. The LSTM processes the data in a sequence and

generates the output of the shape `[Sequence_length, batch_size, hidden_size]`, where each sequence index represents the output of that sequence. In this case, we just take the output of the last sequence, which is of the shape `[batch_size, hidden_dim]`, and pass it on to a linear layer to map it to the output categories. Since the model tends to overfit, add a dropout layer. You can play with the dropout probabilities.

Training the model

Once the network is created, we can train the model using the same code as seen in the previous examples. The following is the code for training the model:

```
model = IMDBRnn(n_vocab,n_hidden,3,bs=32)
model = model.cuda()

optimizer = optim.Adam(model.parameters(),lr=1e-3)

def fit(epoch,model,data_loader,phase='training',volatile=False):
    if phase == 'training':
        model.train()
    if phase == 'validation':
        model.eval()
        volatile=True
    running_loss = 0.0
    running_correct = 0
    for batch_idx , batch in enumerate(data_loader):
        text , target = batch.text , batch.label
        if is_cuda:
            text,target = text.cuda(),target.cuda()

        if phase == 'training':
            optimizer.zero_grad()
        output = model(text)
        loss = F.nll_loss(output,target)

        running_loss +=
F.nll_loss(output,target,size_average=False).data[0]
        preds = output.data.max(dim=1,keepdim=True) [1]
        running_correct +=
preds.eq(target.data.view_as(preds)).cpu().sum()
        if phase == 'training':
            loss.backward()
            optimizer.step()

        loss = running_loss/len(data_loader.dataset)
        accuracy = 100. * running_correct/len(data_loader.dataset)

        print(f'{phase} loss is {loss:.2f} and {phase} accuracy is
{running_correct}/{len(data_loader.dataset)}{accuracy:.4f}')
    return loss,accuracy

train_losses , train_accuracy = [],[]
validation_losses , validation_accuracy = [],[]

for epoch in range(1,5):
```

```
    epoch_loss, epoch_accuracy =
fit(epoch,model,train_iter,phase='training')
    validation_epoch_loss , validation_epoch_accuracy =
fit(epoch,model,test_iter,phase='validation')
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    validation_losses.append(validation_epoch_loss)
    validation_accuracy.append(validation_epoch_accuracy)
```

The following is the result of the training model:

```
training loss is 0.7 and training accuracy is 12564/25000 50.26
validation loss is 0.7 and validation accuracy is 12500/25000 50.0
training loss is 0.66 and training accuracy is 14931/25000 59.72
validation loss is 0.57 and validation accuracy is 17766/25000 71.06
training loss is 0.43 and training accuracy is 20229/25000 80.92
validation loss is 0.4 and validation accuracy is 20446/25000 81.78
training loss is 0.3 and training accuracy is 22026/25000 88.1
validation loss is 0.37 and validation accuracy is 21009/25000 84.04
```

Training the model for four epochs gave an accuracy of 84%.

Training for more epochs resulted in an overfitted model, as the loss started increasing. We can try some of the techniques that we tried such as decreasing the hidden dimensions, increasing sequence length, and training with smaller learning rates to further improve the accuracy.

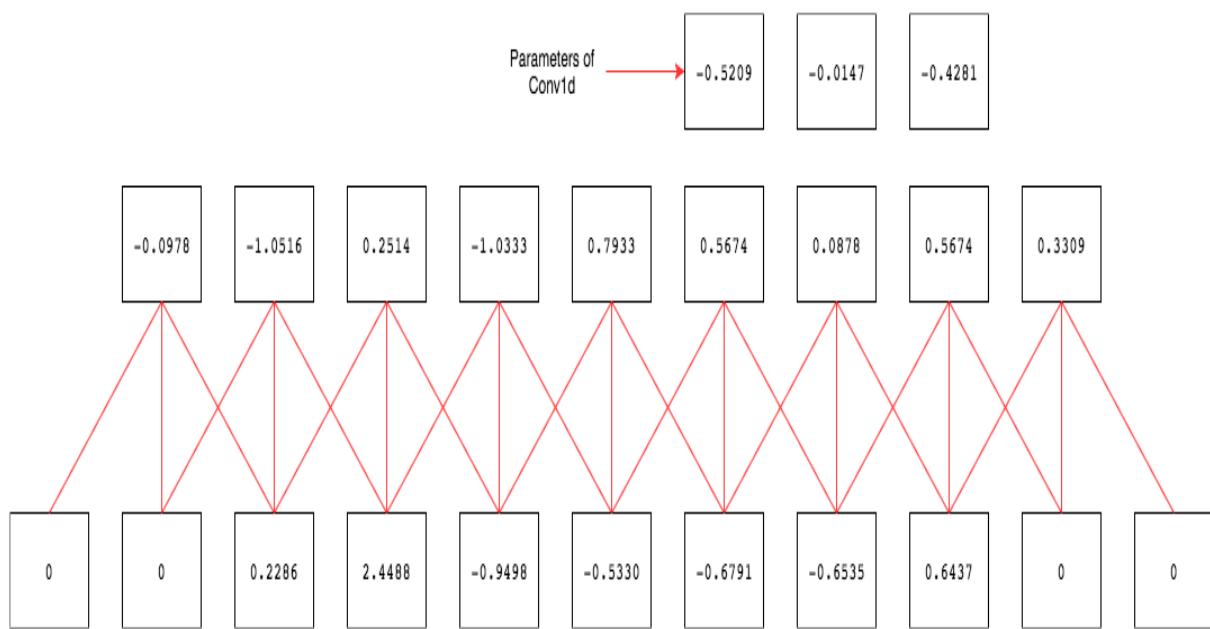
We will also explore how we can use one-dimensional convolutions for training on sequence data.

Convolutional network on sequence data

We learned how CNNs solve problems in computer vision by learning features from the images in [Chapter 4](#), *Deep Learning for Computer Vision*. In images, CNNs work by convolving across height and width. In the same way, time can be treated as a convolutional feature. One-dimensional convolutions sometimes perform better than RNNs and are computationally cheaper. In the last few years, companies such as Facebook have shown success in audio generation and machine translation. In this section, we will learn how CNNs can be used to build a text classification solution.

Understanding one-dimensional convolution for sequence data

In [Chapter 4](#), *Deep Learning for Computer Vision*, we have seen how two-dimensional weights are learned from the training data. These weights move across the image to generate different activations. In the same way, one-dimensional convolution activations are learned during the training of our text classifier, where these weights learn patterns by moving across the data. The following diagram explains how one-dimensional convolutions will work:



For training a text classifier on the IMDB dataset, we will follow the same steps as we followed for building the classifier using LSTMs. The only thing that changes is that we use `batch_first = True`, unlike in our LSTM network. So, let's look at the network, the training code, and its results.

Creating the network

Let's look at the network architecture and then walk through the code:

```
class IMDBCnn(nn.Module):

    def
    __init__(self,vocab,hidden_size,n_cat,bs=1,kernel_size=3,max_len=200):
super().__init__()
    self.hidden_size = hidden_size
    self.bs = bs
    self.e = nn.Embedding(n_vocab,hidden_size)
    self.cnn = nn.Conv1d(max_len,hidden_size,kernel_size)
    self.avg = nn.AdaptiveAvgPool1d(10)
    self.fc = nn.Linear(1000,n_cat)
    self.softmax = nn.LogSoftmax(dim=-1)

    def forward(self,inp):
        bs = inp.size()[0]
        if bs != self.bs:
            self.bs = bs
        e_out = self.e(inp)
        cnn_o = self.cnn(e_out)
        cnn_avg = self.avg(cnn_o)
        cnn_avg = cnn_avg.view(self.bs,-1)
        fc = F.dropout(self.fc(cnn_avg),p=0.5)
        return self.softmax(fc)
```

In the preceding code, instead of an LSTM layer, we have a `Conv1d` layer and an `AdaptiveAvgPool1d` layer. The convolution layer accepts the sequence length as its input size, and the output size to the hidden size, and the kernel size as three. Since we have to change the dimensions of the linear layer, every time we try to run it with different lengths, we use an `AdaptiveAvgpool1d` layer, which takes input of any size and generates an output of the given size. So, we can use a linear layer whose size is fixed. The rest of the code is similar to what we have seen in most of the network architectures.

Training the model

The training steps for the model are the same as the previous example. Let's just look at the code to call the `fit` method and the results it generated:

```
train_losses , train_accuracy = [], []
validation_losses , validation_accuracy = [], []

for epoch in range(1,5):

    epoch_loss, epoch_accuracy =
    fit(epoch,model,train_iter,phase='training')
    validation_epoch_loss , validation_epoch_accuracy =
    fit(epoch,model,test_iter,phase='validation')
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    validation_losses.append(validation_epoch_loss)
    validation_accuracy.append(validation_epoch_accuracy)
```

We ran the model for four epochs, which gave approximately 83% accuracy. Here are the results of running the model:

```
training loss is 0.59 and training accuracy is 16724/25000 66.9
validation loss is 0.45 and validation accuracy is 19687/25000 78.75
training loss is 0.38 and training accuracy is 20876/25000 83.5
validation loss is 0.4 and validation accuracy is 20618/25000 82.47
training loss is 0.28 and training accuracy is 22109/25000 88.44
validation loss is 0.41 and validation accuracy is 20713/25000 82.85
training loss is 0.22 and training accuracy is 22820/25000 91.28
validation loss is 0.44 and validation accuracy is 20641/25000 82.56
```

Since the validation loss started increasing after three epochs, I stopped running the model. A couple of things that we could try to improve the results are using pretrained weights, adding another convolution layer, and trying a `MaxPool1d` layer between the convolutions. I leave it to you to try this and see if that helps improve the accuracy. Now that we have learned all about the various neural networks to process sequence data, let us see language modeling in the next section.

Language modeling

Language modeling is the task of predicting the next word in a piece of text given the previous words. The ability to generate this sequential data has applications in many different areas such as the following:

- Image captioning
- Speech recognition
- Language translation
- Automatic email reply
- Writing stories, news articles, poems, and so on

The focus in this area was initially held by RNNs, in particular, LSTMs. However, after the introduction of the Transformer architecture in 2017 (<https://arxiv.org/pdf/1706.03762.pdf>) it became prevalent in NLP tasks. A number of modifications of the transformer have since appeared, some of which we will cover in this chapter.

Pretrained models

In recent times there has been a lot of interest in the use of pretrained models for NLP tasks. A key benefit to using pretrained language models is that they are able to learn with significantly less data. These models are particularly beneficial for languages where labeled data is scarce as they only require labeled data.

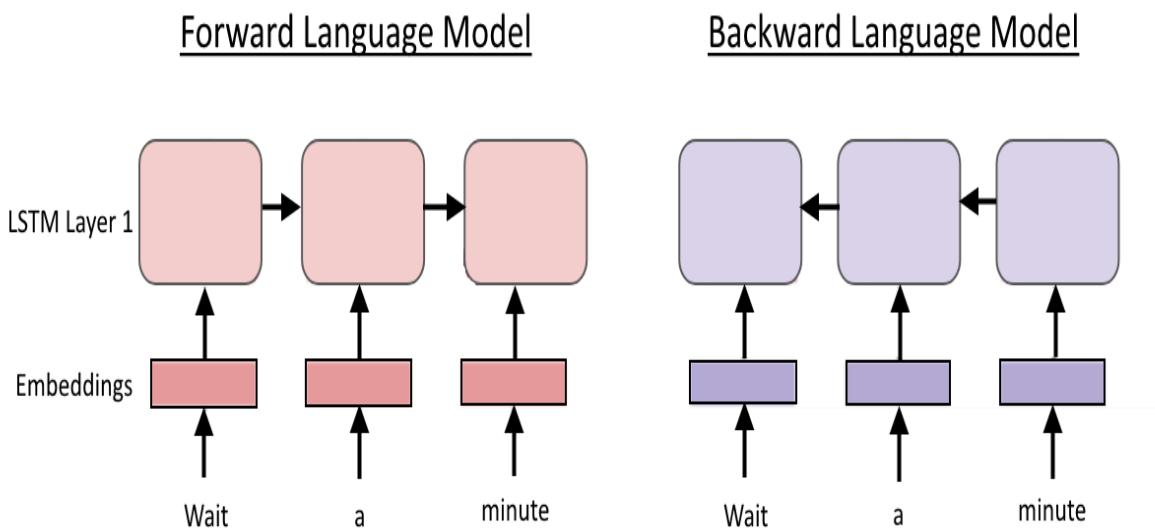
Pretrained models for sequence learning were first proposed in 2015 by A. M. Dai and Q. V. Le, in a paper titled *Semi-supervised Sequence Learning* (<http://arxiv.org/abs/1511.01432>). However, it is only recently that they have shown to be beneficial across a broad range of tasks. We will now consider some of the most noteworthy advances in this field in recent times, which include, but are by no means limited to the following:

- **Embeddings from Language Models (ELMo)**
- **Bidirectional Encoder Representations from Transformers (BERT)**
- **Generative Pretrained Transformer 2 (GPT-2)**

Embeddings from language models

In February 2018, the *Deep contextualized word representations* paper by M. Peters and others (<https://arxiv.org/abs/1802.05365>) introduced ELMo. It essentially demonstrated that language model embeddings can be used as features in a target model as shown in the following diagram:

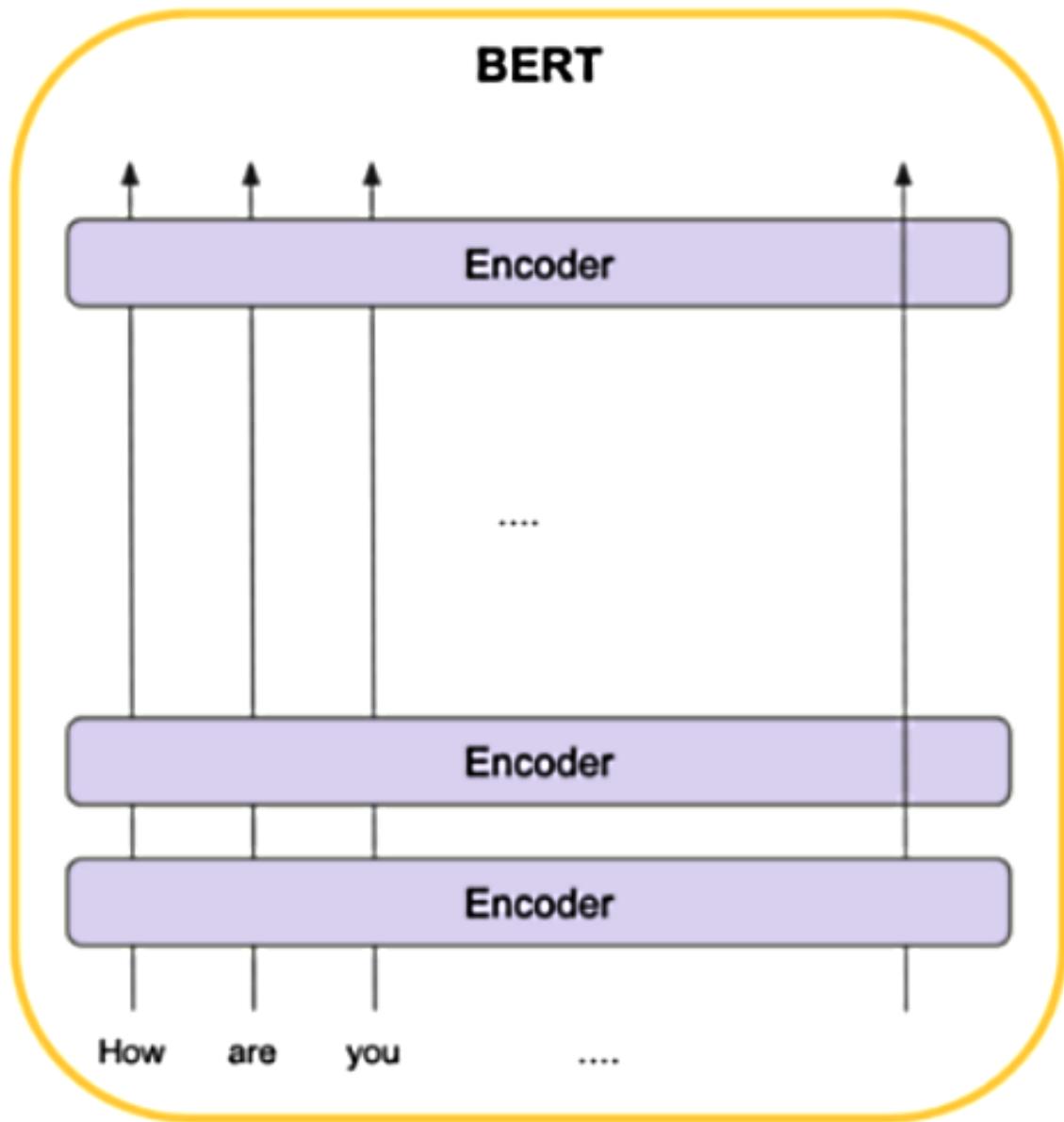
Embedding of "Wait a minute"



ELMo uses a bidirectional language model to learn both word and context. The internal states of both the forward and the backward pass are concatenated at each word to produce an intermediate vector. It is the bidirectional nature of the model that gives it information about both the next word in the sentence and the words before it.

Bidirectional Encoder Representations from Transformers

In a later paper published by Google in November 2018 (<https://arxiv.org/pdf/1810.04805.pdf>), **Bidirectional Encoder Representations from Transformers (BERT)** was proposed, which incorporates an attention mechanism that learns the contextual relations between words and text:



Unlike ELMo where the text input is read sequentially (left to right or right to left), with BERT, the entire sequence of words is read at once. Essentially, BERT is a trained transformer encoder stack.

Generative Pretrained Transformer 2

At the time of writing, OpenAI's GTP-2 is one of the most state-of-the-art language models designed to improve on the realism and coherence of generated text. It was introduced in the paper *Language Models are Unsupervised Multi-task Learners* (https://d4mucf.pksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf) in February 2019. It was trained to predict the next word on 8 million web pages (which totalled 40 GB of text) with 1.5 billion parameters, which is four times as many as BERT. Here is what OpenAI has to say about GPT-2:

GPT-2 generates coherent paragraphs of text, achieves state-of-the-art performance on many language modeling benchmarks, and performs rudimentary reading comprehension, machine translation, question answering, and summarization—all without task-specific training.

Initially, OpenAI said they would not release the dataset, code, or the full GPT-2 model weights. The reason for this was due to concerns about them being used to generate deceptive, biased, or abusive language at scale. Examples of some of the applications of these models for malicious purposes are as follows:

- Realistic fake news articles
- Realistically impersonate others online
- Abusive or faked content that could be posted on social media
- Automate the production of spam or phishing content

The team then decided to do a staged release of the model in order to give people time to assess the societal implications and evaluate the impacts of release after each stage.

PyTorch implementations

There is a popular GitHub repository from the developer Hugging Face that has implemented BERT and GPT-2 in PyTorch. It can be found at the following web link: https://github.com/huggingface/pytorch-pretrained_bert. The repository was first made available in November 2018 and permits the user to generate sentences from their own data. It also includes a variety of classes that can be used to test different models when applied to different tasks, such as question answering, token classification, and sequence classification.

The following code snippet demonstrates how the code from the GitHub repository can be used to generate text from the GPT-2 model. Firstly, we import the associated libraries and initialize the pretrained information as follows:

```
import torch
from torch.nn import functional as F
from pytorch_pretrained_bert import GPT2Tokenizer, GPT2LMHeadModel

tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
gpt2model = GPT2LMHeadModel.from_pretrained('gpt2')
```

In this example, we feed the model the 'We like unicorns because they' sentence and it generates words from there like so:

```
input_text = tokenizer.encode('We like unicorns because they')
input, past = torch.tensor([input_text]), None
for _ in range(25):
    logits, past = gpt2model(input, past=past)
    input = torch.multinomial(F.softmax(logits[:, -1]), 1)
    input_text.append(input.item())
```

The following is the output:

"We like unicorns because they have power. We like unicorns because our communities are hierarchical, communities that are structured around competitive hierarchy. We're a hom"

GPT-2 playground

There is another useful GitHub repository from developer ilopezfr, which can be found at the following link: <https://github.com/ilopezfr/gpt-2>. It also provides a notebook in Google Colab that allows the user to play around and experiment with the OpenAI GPT-2 model (https://colab.research.google.com/github/ilopezfr/gpt-2/blob/master/gpt-2-playground_.ipynb).

The following are some examples from different sections of the playground:

- The **Text Completion** section:

1. Text Completion

- Context: random unseen text

Sample prompt 1:

In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

Sample prompt 2: ([Voight-Kampff test](#))

You're in a desert, walking along in the sand, when all of a sudden you look down and see a tortoise, Leon. It's crawling toward you. You reach down, you flip the tortoise over on its back. The tortoise lays on its back, its belly baking in the hot sun, beating its legs trying to turn itself over, but it can't, not without your help. But you're not helping. Why is that?

Sample prompt 3:

I've seen things you people wouldn't believe. Attack ships on fire off the shoulder of Orion. I watched C-beams glitter in the dark near the Tannhäuser Gate. All those moments will be lost in time, like tears in rain. Time to die.

- The **Question-Answering** section:

2. Question-Answering

- Context: passage, some question/answer pairs, and token A:
- For a single word answer (i.e.: Yes/No, city), set flag length=1

Sample prompt 1 ([The Baseline test](#))

Q: What's it like to hold the hand of someone you love?

A: Interlinked.

Q: Do they teach you how to feel finger to finger?

A: Interlinked.

Q: Do you long for having your heart interlinked?

A:

Sample prompt 2:

The 2008 Summer Olympics torch relay was run from March 24 until August 8, 2008, prior to the 2008 Summer Olympics, with the theme of "one world, one dream". Plans for the relay were announced on April 26, 2007, in Beijing, China. The relay, also called by the organizers as the "Journey of Harmony", lasted 129 days and carried the torch 137,000 km (85,000 mi) – the longest distance of any Olympic torch relay since the tradition was started ahead of the 1936 Summer Olympics.

After being lit at the birthplace of the Olympic Games in Olympia, Greece on March 24, the torch traveled to the Panathinaiko Stadium in Athens, and then to Beijing, arriving on March 31. From Beijing, the torch was following a route passing through six continents. The torch has visited cities along the Silk Road, symbolizing ancient links between China and the rest of the world. The relay also included an ascent with the flame to the top of

Mount Everest on the border of Nepal and Tibet, China from the Chinese side, which was closed specially for the event.

Q: What was the length of the race?

A: 137,000 km

Q: Was it larger than previous ones?

A: No

Q: Where did the race begin?

A: Olympia, Greece

Q: Where did they go after?

A: Athens

- The Translation section:

4. Translation

- Context: a few example pairs of the format `english_sentence = spanish_sentence`, and then `english_sentence = at the end`.

Sample prompt:

Good morning. = Buenos días.

I am lost. Where is the restroom? = Estoy perdido. ¿Dónde está el baño?

How much does it cost? = ¿Cuánto cuesta?

How do you say maybe in Spanish? = ¿Cómo se dice maybe en Español?

Would you speak slower, please. = Por favor, habla mas despacio.

Where is the book store? = ¿Dónde está la librería?

At last a feminist comedian who makes jokes about men. = Por fin un cómico feminista que hace chistes sobre hombres.

Summary

In this chapter, we learned different techniques to represent text data in deep learning. We learned how to use pretrained word embeddings and our own trained embeddings when working on a different domain. We built a text classifier using LSTMs and one-dimensional convolutions. We also learned about how to generate text using state-of-the-art language modeling architectures.

In the next chapter, we will learn how to train deep learning algorithms to generate stylish images, and new images, and to generate text.

Section 3: Understanding Modern Architectures in Deep Learning

In this section, you will become well versed in the various modern architectures of deep learning.

This section contains the following chapters:

- [Chapter 6](#), *Implementing Autoencoders*
- [Chapter 7](#), *Working with Generative Adversarial Networks*
- [Chapter 8](#), *Transfer Learning with Modern Network Architectures*
- [Chapter 9](#), *Deep Reinforcement Learning*
- [Chapter 10](#), *What Next?*

Implementing Autoencoders

This chapter addresses the notion of semi-supervised learning algorithms through the introduction of autoencoders, and then moves on to **restricted Boltzmann machines (RBMs)** and **deep belief networks (DBNs)** in order to understand the probability distribution of data. The chapter will give you an overview as to how these algorithms have been applied to some real-world problems. Coded examples implemented in PyTorch will also be provided.

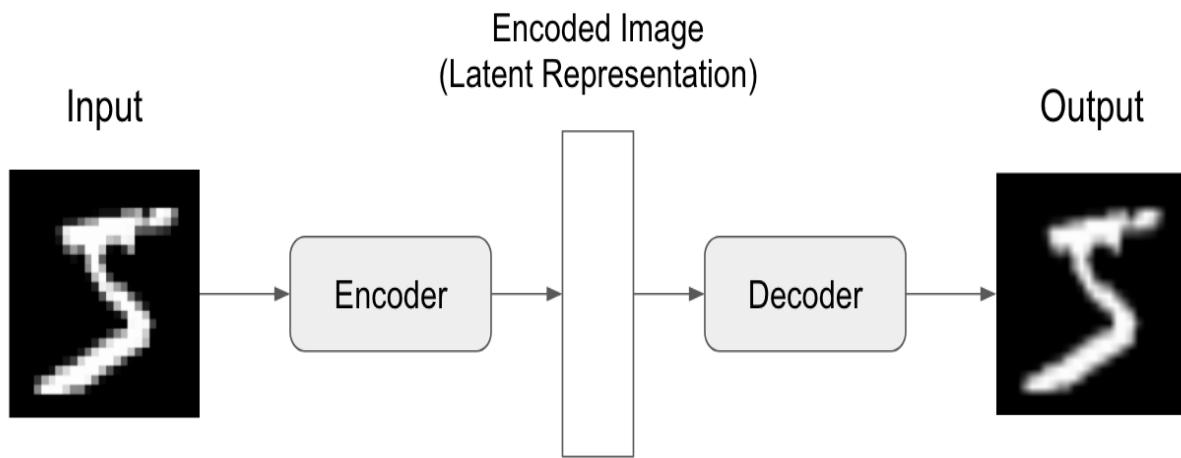
Autoencoders are an unsupervised learning technique. They can take an unlabeled dataset and task it with reconstructing the original input by modeling it using an unsupervised learning problem as opposed to a supervised one. The goal of the autoencoder is for the input to be as similar as possible to the output.

Specifically, the following topics will be covered in this chapter:

- An overview of autoencoders and their applications
- Bottleneck and loss functions
- Different types of autoencoders
- Restricted Boltzmann machines
- Deep belief networks

Applications of autoencoders

Autoencoders fall under representational learning and are used to find a compressed representation of the inputs. They are composed of an encoder and a decoder. The following diagram shows the structure of an autoencoder:

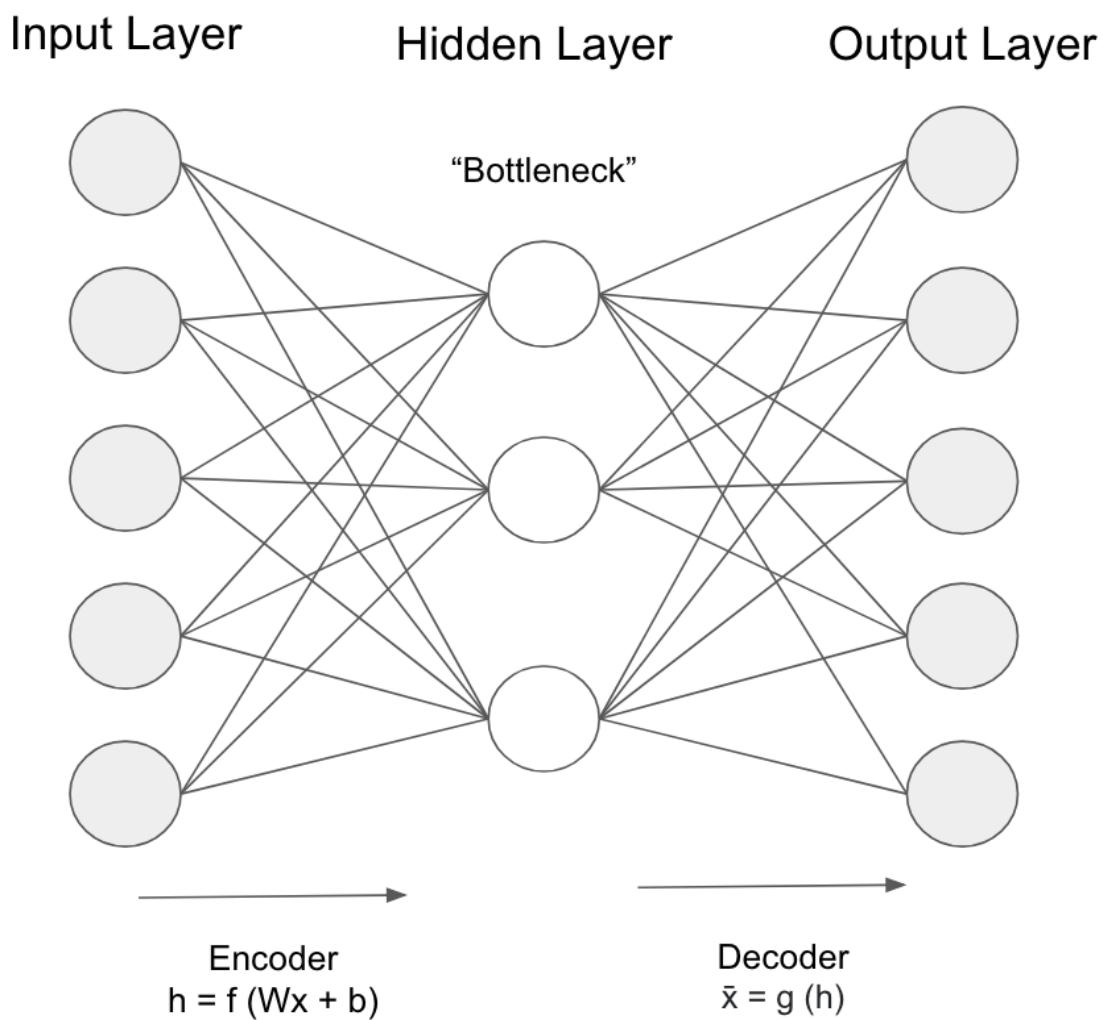


Examples of applications of autoencoders include the following:

- Data denoising
- Dimensionality reduction for data visualization
- Image generation
- Interpolating text

Bottleneck and loss functions

Autoencoders impose a bottleneck on the network, enforcing a compressed knowledge representation of the original input. The network would simply learn to memorize the input values if the bottleneck were not present. As such, this would mean that the model wouldn't generalize well on unseen data:



In order for the model to detect a signal, we need it to be sensitive to the input but not so much that it simply memorizes them and doesn't

predict well on unseen data. In order to determine the optimal trade-off, we need to construct a loss/cost function:

$$Loss = L(x, \bar{x}) + \text{regularizer}$$

There are some commonly used autoencoder architectures for imposing these two constraints and ensuring there is an optimal trade-off between the two.

Coded example – standard autoencoder

In this example, we will show you how to compile an autoencoder model in PyTorch:

1. Firstly, import the relevant libraries:

```
import os
from torch import nn
from torch.autograd import Variable
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.datasets import MNIST
from torchvision.utils import save_image
```

2. Now, define the model parameters:

```
number_epochs = 10
batch_size = 128
learning_rate = 1e-4
```

3. Then, initiate a function to transform the images in the MNIST dataset:

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

dataset = MNIST('./data', transform=transform_image)
data_loader = DataLoader(dataset, batch_size=batch_size,
shuffle=True)
```

4. Define the autoencoder class in which to feed the data and initiate the model:

```
class autoencoder_model(nn.Module):
    def __init__(self):
        super(autoencoder_model, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(28 * 28, 128),
            nn.ReLU(True),
```

```

        nn.Linear(128, 64),
        nn.ReLU(True), nn.Linear(64, 12), nn.ReLU(True),
nn.Linear(12, 3))
    self.decoder = nn.Sequential(
        nn.Linear(3, 12),
        nn.ReLU(True),
        nn.Linear(12, 64),
        nn.ReLU(True),
        nn.Linear(64, 128),
        nn.ReLU(True), nn.Linear(128, 28 * 28), nn.Tanh())

def forward(self, x):
    x = self.encoder(x)
    x = self.decoder(x)
    return x

model = autoencoder_model()
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(
    model.parameters(), lr=learning_rate, weight_decay=1e-5)

```

5. Define a function that will output the images from the model after each epoch:

```

def to_image(x):
    x = 0.5 * (x + 1)
    x = x.clamp(0, 1)
    x = x.view(x.size(0), 1, 28, 28)
    return x

```

6. Now run the model over each epoch and review the results of the reconstructed images:

```

for epoch in range(number_epochs):
    for data in data_loader:
        image, i = data
        image = image.view(image.size(0), -1)
        image = Variable(image)

        # Forward pass
        output = model(image)
        loss = criterion(output, image)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        print('Epoch [{}/{}], Loss:{:.4f}'.format(epoch + 1,
number_epochs, loss.data[0]))
        if epoch % 10 == 0:
            pic = to_image(output.cpu().data)
            save_image(pic, './mlp_img/image_{}.png'.format(epoch))

```

```
| torch.save(model.state_dict(), './sim_autoencoder.pth')
```

This will give the following output:

```
Epoch [1/10], Loss:0.2003
Epoch [2/10], Loss:0.1963
Epoch [3/10], Loss:0.1896
Epoch [4/10], Loss:0.1767
Epoch [5/10], Loss:0.1853
Epoch [6/10], Loss:0.1709
Epoch [7/10], Loss:0.1838
Epoch [8/10], Loss:0.1776
Epoch [9/10], Loss:0.1611
Epoch [10/10], Loss:0.1614
```

And the following image shows the output of the autoencoder at each epoch:



The more epochs that pass, the clearer the images become as the model continues to learn.

Convolutional autoencoders

Autoencoders can be used with convolutions instead of fully connected layers. This can be done using 3D vectors instead of 1D vectors. In the context of images, downsampling the image forces the autoencoder to learn a compressed version of it.

Coded example – convolutional autoencoder

In this example, we will show you how to compile a convolutional autoencoder:

1. As before, you obtain the train and test datasets from the MNIST dataset and define the model parameters:

```
number_epochs = 10
batch_size = 128
learning_rate = 1e-4

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

dataset = MNIST('./data', transform=transform_image)
data_loader = DataLoader(dataset, batch_size=batch_size,
shuffle=True)
```

2. From here, initiate the model for the convolutional autoencoder:

```
class conv_autoencoder(nn.Module):
    def __init__(self):
        super(conv_autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, stride=3, padding=1),
            nn.ReLU(True),
            nn.MaxPool2d(2, stride=2),
            nn.Conv2d(16, 8, 3, stride=2, padding=1),
            nn.ReLU(True),
            nn.MaxPool2d(2, stride=1)
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(8, 16, 3, stride=2),
            nn.ReLU(True),
            nn.ConvTranspose2d(16, 8, 5, stride=3, padding=1),
            nn.ReLU(True),
            nn.ConvTranspose2d(8, 1, 2, stride=2, padding=1),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.encoder(x)
```

```

        x = self.decoder(x)
        return x

model = conv_autoencoder()
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate,
weight_decay=1e-5)

```

3. Finally, run the model over each epoch while saving the output images for reference:

```

for epoch in range(number_epochs):
    for data in data_loader:
        img, i = data
        img = Variable(img)

        # Forward pass
        output = model(img)
        loss = criterion(output, img)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Print results
    print('epoch [{}/{}], loss:{:.4f}'
          .format(epoch+1, number_epochs, loss.data[0]))
    if epoch % 10 == 0:
        pic = to_image(output.cpu().data)
        save_image(pic, './dc_img/image_{}.png'.format(epoch))

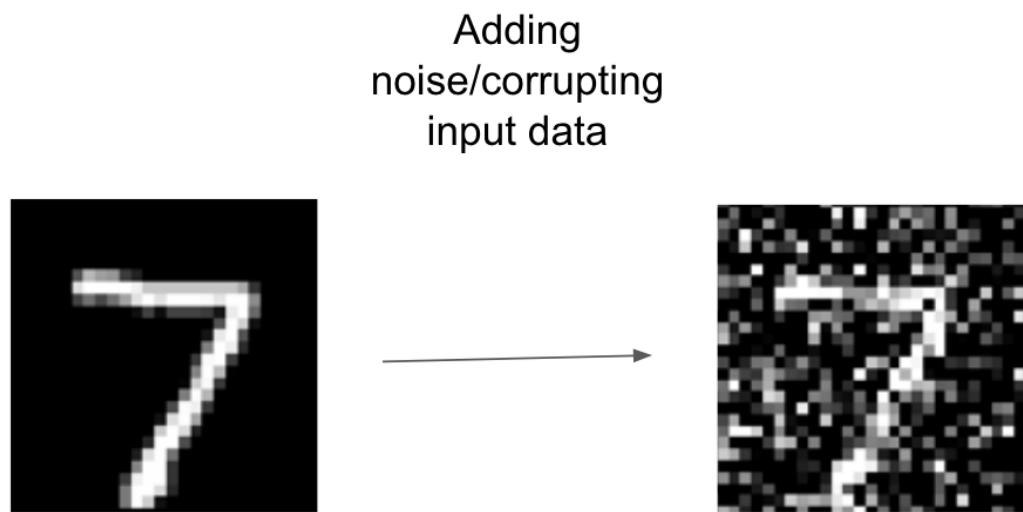
torch.save(model.state_dict(), './convolutional_autoencoder.pth')

```

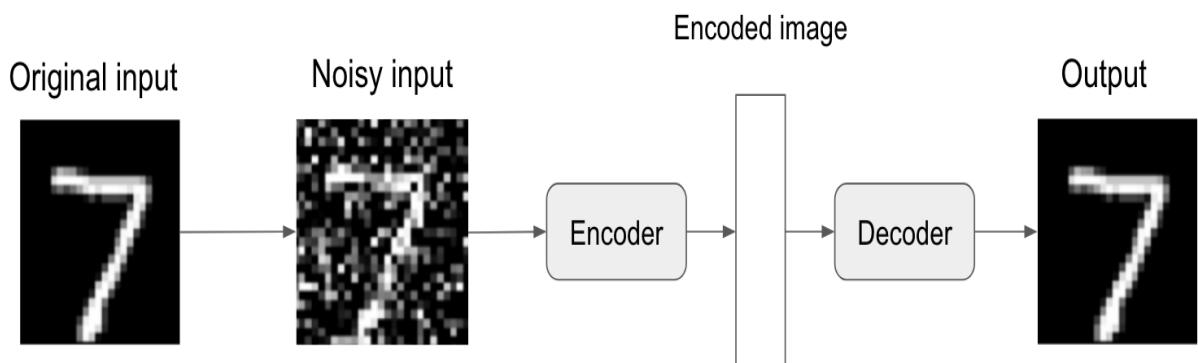
We can look at the saved images after every epoch in the folder that is mentioned in the code.

Denoising autoencoders

Denoising encoders deliberately add noise to the input of the network. These autoencoders essentially create a corrupted copy of the data. In doing so, this helps the encoder to learn the latent representation in the input data, making it more generalizable:



This corrupted image is fed into the network in the same way as other standard autoencoders:



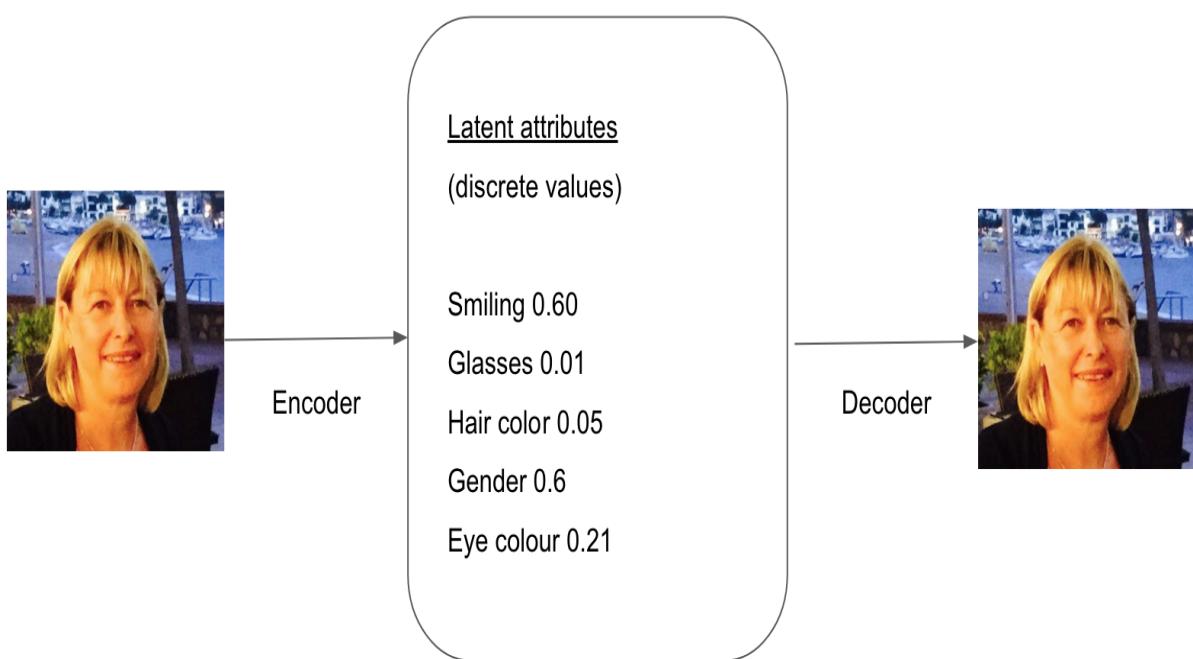
As we can see, noises were added in the original input and the encoder encodes the input and sends it to the decoder, which then decodes the noisy input into the cleaned output. Thus, we have looked at various applications that autoencoders can be used for. We will now look at a specific type of autoencoder, which is a **variational autoencoder (VAE)**.

Variational autoencoders

VAEs are different from the standard autoencoders we have considered so far as they describe an observation in latent space in a probabilistic manner rather than deterministic. A probability distribution for each latent attribute is output, rather than a single value.

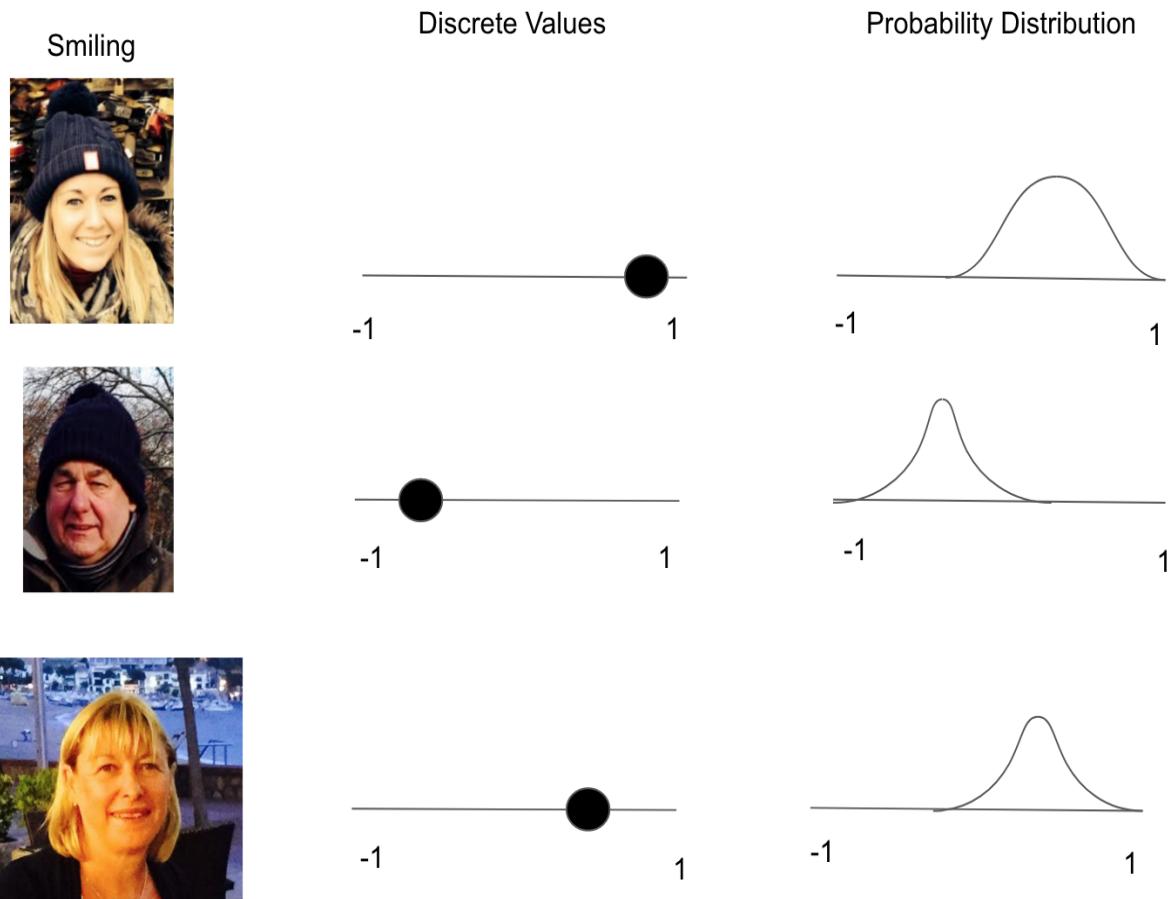
Standard autoencoders have somewhat limited applications in the real world as they are only really useful when you want to replicate the data that has been put into it. Since VAEs are generative models, they can be applied to cases where you don't want to output data that is the same as the input.

Let's consider this in a real-world context. When training an autoencoder model on a dataset of faces, you would hope that it would learn latent attributes, such as whether the person is smiling, their skin tone, whether they are wearing glasses, and more:



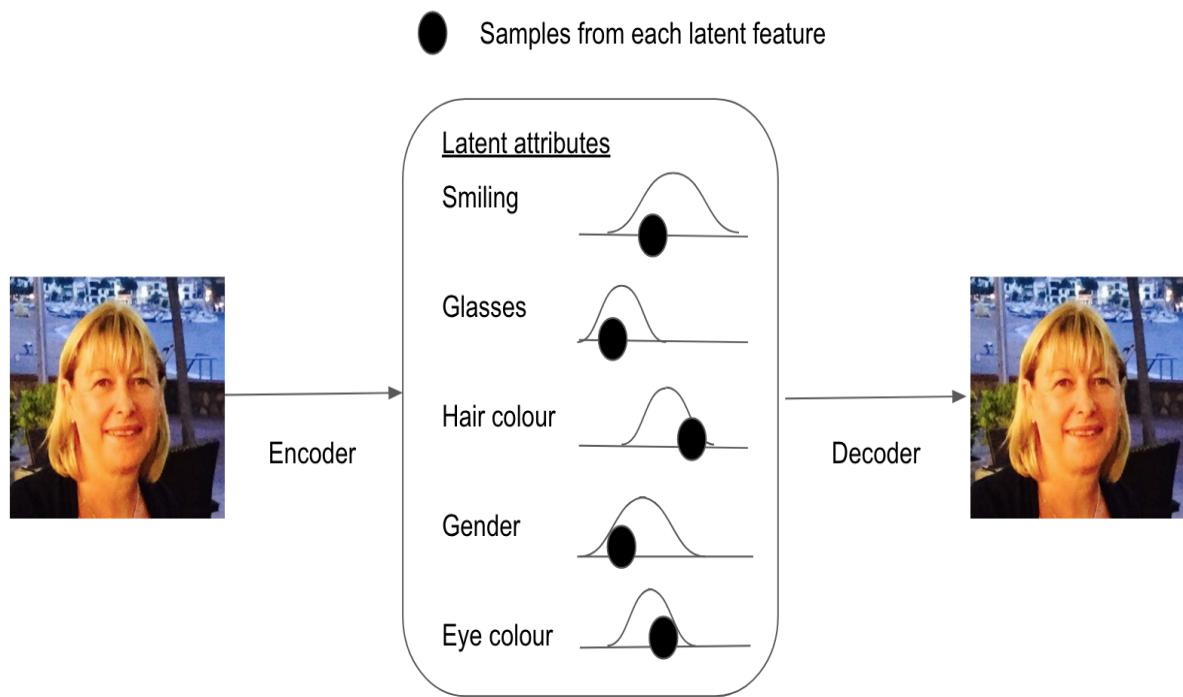
As we can see in the preceding diagram, standard autoencoders represent these latent attributes as discrete values.

If we allow each feature to be within a range of possible values rather than a single value, we can use VAEs to describe the attributes in probabilistic terms:

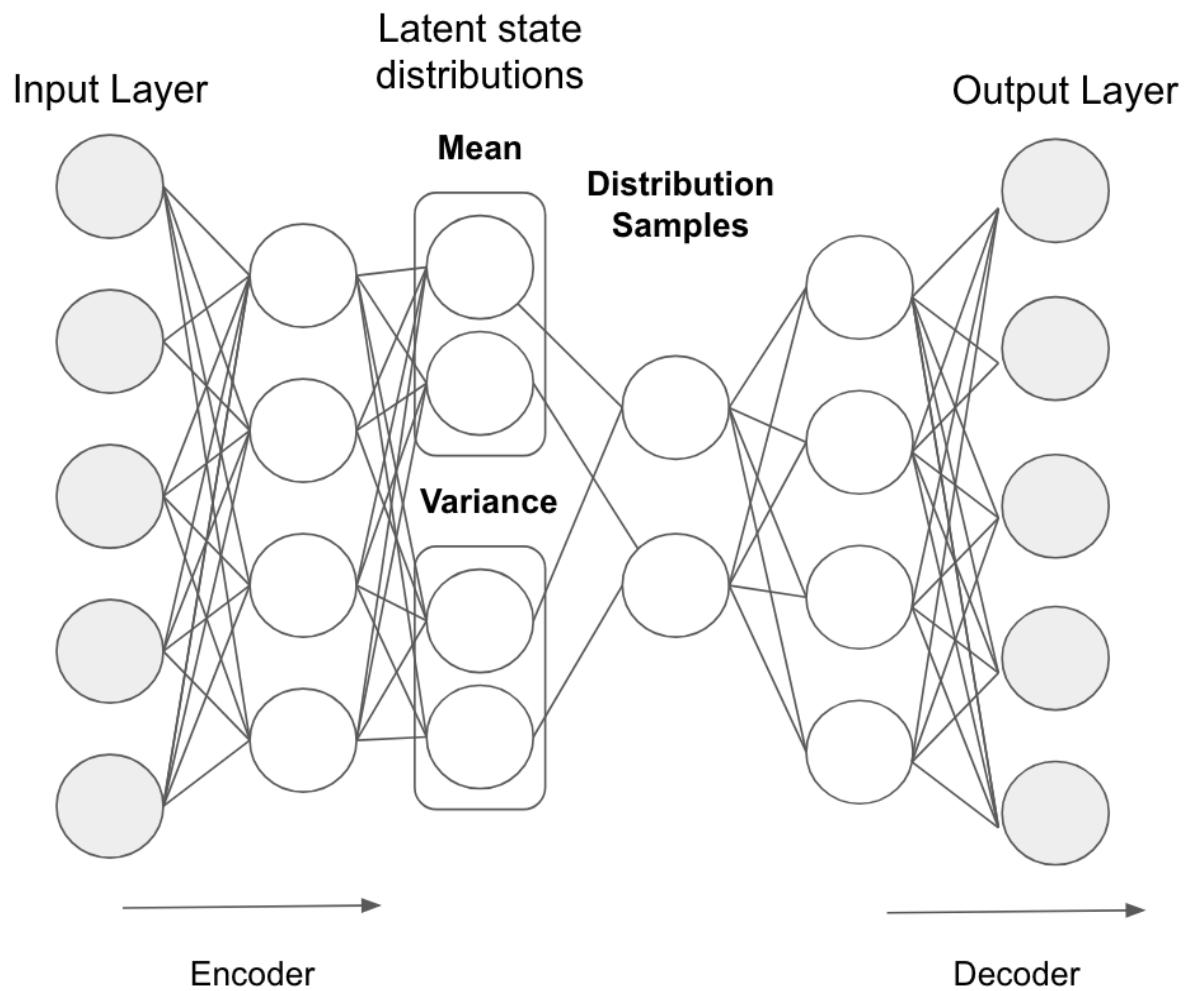


The preceding diagram depicts how we can represent whether the person is smiling as either a discrete value or as a probability distribution.

The distribution of each latent attribute is sampled from the image in order to generate the vector that is used as the input for the decoder model:



Two vectors are output, as shown in the following diagram:

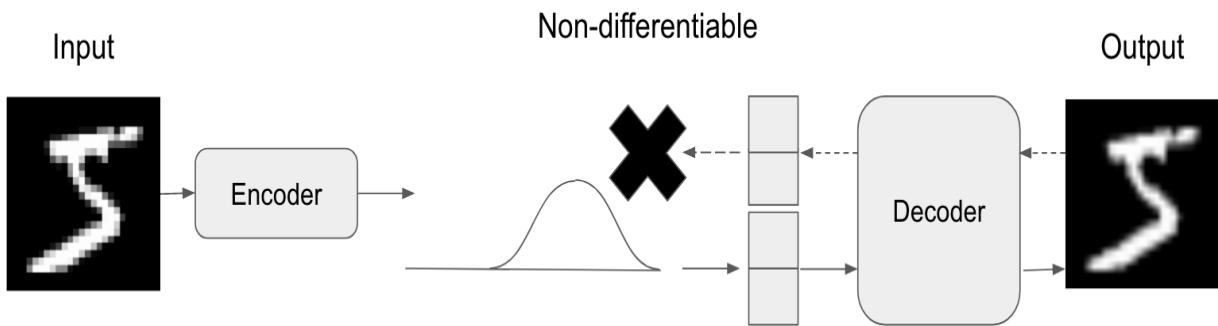


One describes the mean and the other describes the variance of the distributions.

Training VAEs

During training, we calculate the relationship of each parameter in the network with respect to the overall loss using a process called **backpropagation**.

Standard autoencoders use backpropagation in order to reconstruct the loss value across the weights of the network. As the sampling operation in VAEs is not differentiable, the gradients cannot be propagated from the reconstruction error. The following diagram explains this further:

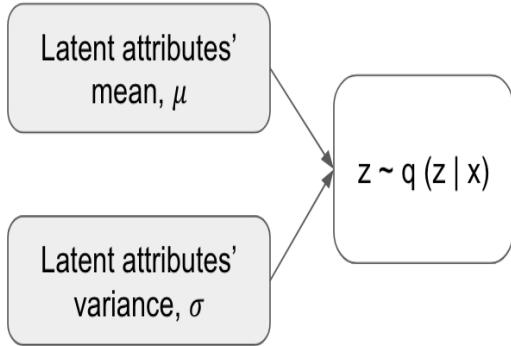


In order to overcome this limitation, the reparameterization trick can be used. The reparameterization trick samples ϵ from a unit normal distribution, shifts it by the mean μ of the latent attribute, and, then scales it by the latent attribute's variance σ :

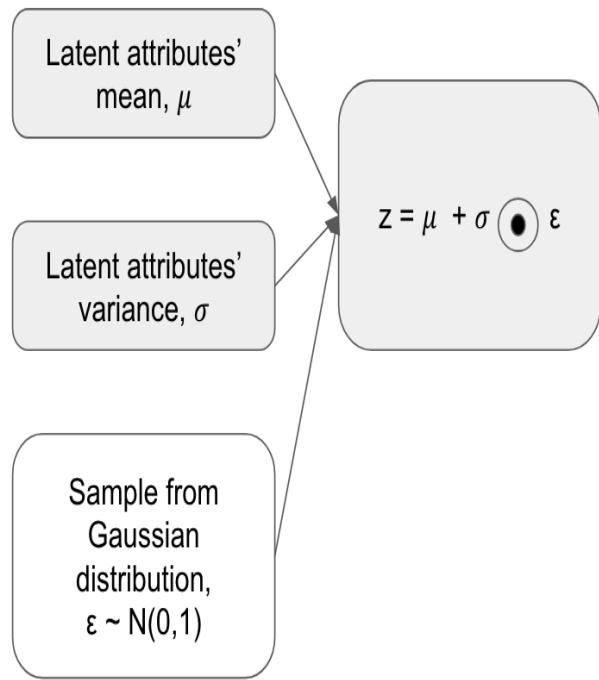
$$z = \mu + \sigma\epsilon$$

This removes the sampling process from the flow of gradients as it is now outside of the network. As such, the sampling process doesn't depend on anything in the network. We can now optimize the parameters of the distribution while maintaining the ability to randomly sample from it:

No reparameterization



With reparameterization

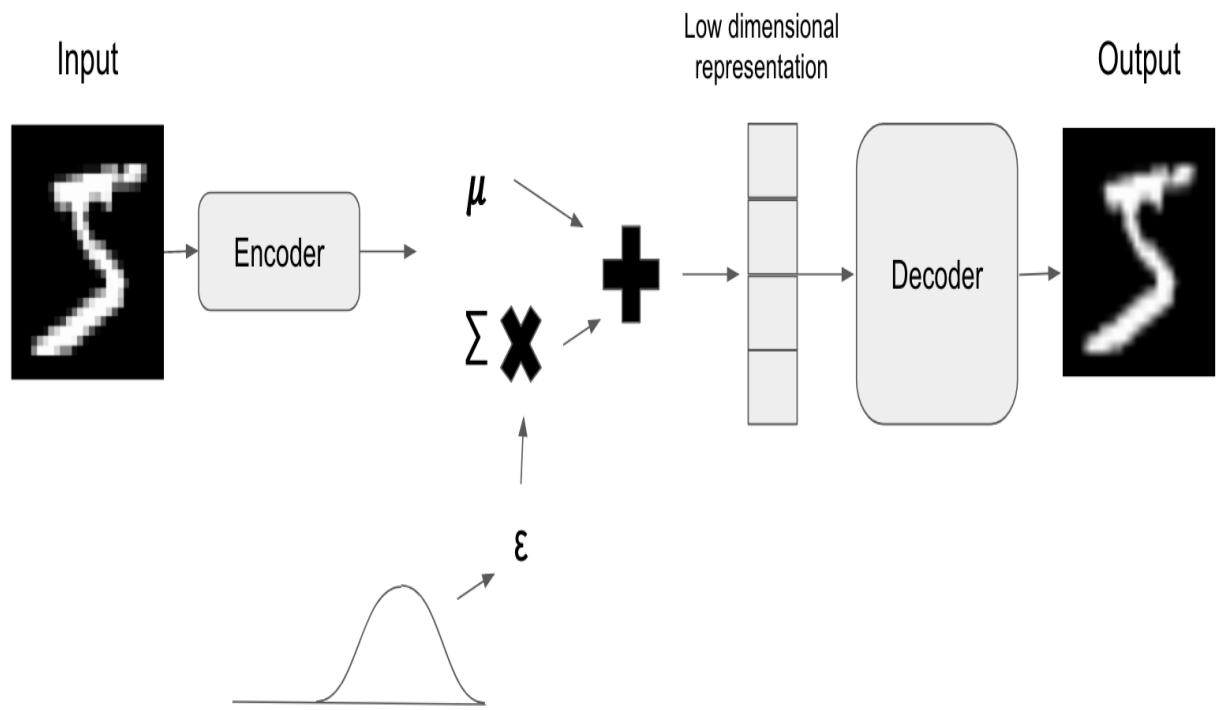


We can transform with a mean, μ , and covariance matrix Σ as per the following, as the distribution of each attribute is Gaussian:

$$z = \mu + \sum^{1/2} \varepsilon$$

Here, $\varepsilon \sim N(0,1)$.

We can now train the model using simple backpropagation with the introduction of the reparameterization trick:



As seen in the preceding diagram, we have trained the autoencoder to smooth out the image.

Coded example – VAE

In order to code a VAE in PyTorch, we can load the libraries and dataset like we did in the previous examples. From here, we can define the VAE class:

```
class VariationalAutoEncoder(nn.Module):
    def __init__(self):
        super(VariationalAutoEncoder, self).__init__()

        self.fc1 = nn.Linear(784, 400)
        self.fc21 = nn.Linear(400, 20)
        self.fc22 = nn.Linear(400, 20)
        self.fc3 = nn.Linear(20, 400)
        self.fc4 = nn.Linear(400, 784)

    def encode_function(self, x):
        h1 = F.relu(self.fc1(x))
        return self.fc21(h1), self.fc22(h1)

    def reparametrize(self, mu, logvar):
        std = logvar.mul(0.5).exp_()
        if torch.cuda.is_available():
            eps = torch.cuda.FloatTensor(std.size()).normal_()
        else:
            eps = torch.FloatTensor(std.size()).normal_()
        eps = Variable(eps)
        return eps.mul(std).add_(mu)

    def decode_function(self, z):
        h3 = F.relu(self.fc3(z))
        return F.sigmoid(self.fc4(h3))

    def forward(self, x):
        mu, logvar = self.encode_function(x)
        z = self.reparametrize(mu, logvar)
        return self.decode_function(z), mu, logvar
```

We then define the loss function with the help of KL divergence and initiate the model:

```
def loss_function(reconstruction_x, x, mu, latent_log_variance):
    """
    reconstruction_x: generating images
    x: original images
    mu: latent mean
    """
    BCE = reconstruction_function(reconstruction_x, x)
```

```

    # KL loss = 0.5 * sum(1 + log(sigma^2) - mu^2 - sigma^2)
    KLD_aspect =
mu.pow(2).add_(latent_log_variance.exp()).mul_(-1).add_(1).add_(logvar)
    KLD = torch.sum(KLD_aspect).mul_(-0.5)
    # KL divergence
    return BCE + KLD

optimizer = optim.Adam(model.parameters(), lr=1e-4)

```

From here, we can run the model over each epoch and save the output:

```

for epoch in range(number_epochs):
    model.train()
    train_loss = 0
    for batch_idx, data in enumerate(data_loader):
        img, _ = data
        img = img.view(img.size(0), -1)
        img = Variable(img)
        if torch.cuda.is_available():
            img = img.cuda()
        optimizer.zero_grad()
        recon_batch, mu, logvar = model(img)
        loss = loss_function(recon_batch, img, mu, logvar)
        loss.backward()
        train_loss += loss.data[0]
        optimizer.step()
        if batch_idx % 100 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch,
                batch_idx * len(img),
                len(data_loader.dataset), 100. * batch_idx /
len(data_loader),
                loss.data[0] / len(img)))

    print('Epoch: {} Average loss: {:.4f}'.format(epoch, train_loss /
len(data_loader.dataset)))
    if epoch % 10 == 0:
        save = to_image(recon_batch.cpu().data)
        save_image(save, './vae_img/image_{}.png'.format(epoch))

torch.save(model.state_dict(), './vae.pth')

```

Now that we have seen the various autoencoders and how to compile them, let us learn how to implement them in recommender systems.

Restricted Boltzmann machines

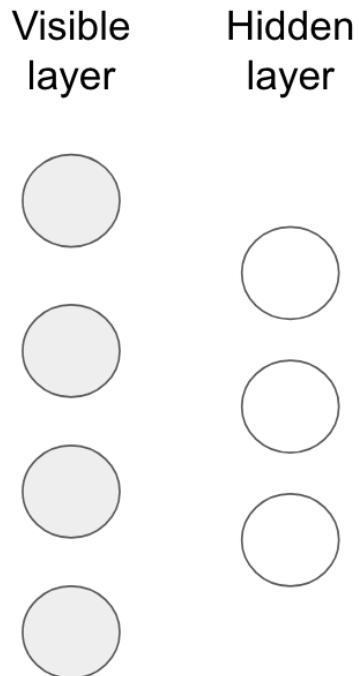
An **RBM** is an algorithm that has been widely used for tasks such as collaborative filtering, feature extraction, topic modeling, and dimensionality reduction. They can learn patterns in a dataset in an unsupervised fashion.

For example, if you watch a movie and say whether you liked it or not, we could use an RBM to help us determine the reason why you made this decision.

The goal of RBM is to minimize energy defined by the following formula, which depends on the configurations of visible/input states, hidden states, weights, and biases:

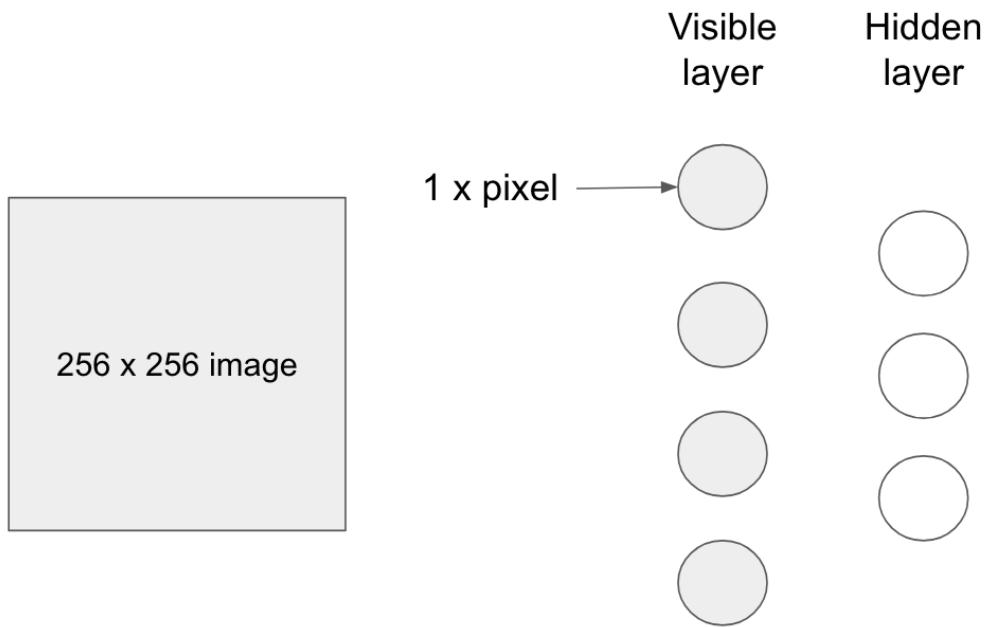
$$E(v, h) = - \sum_i a_i v_i - \sum_j b_j h_j - \sum_{ij} v_i h_j w_{ij}$$

RBMs are two-layer networks that are the fundamental building blocks of a DBN. The first layer of an RBM is a visible/input layer of neurons and the second is the hidden layer of neurons:

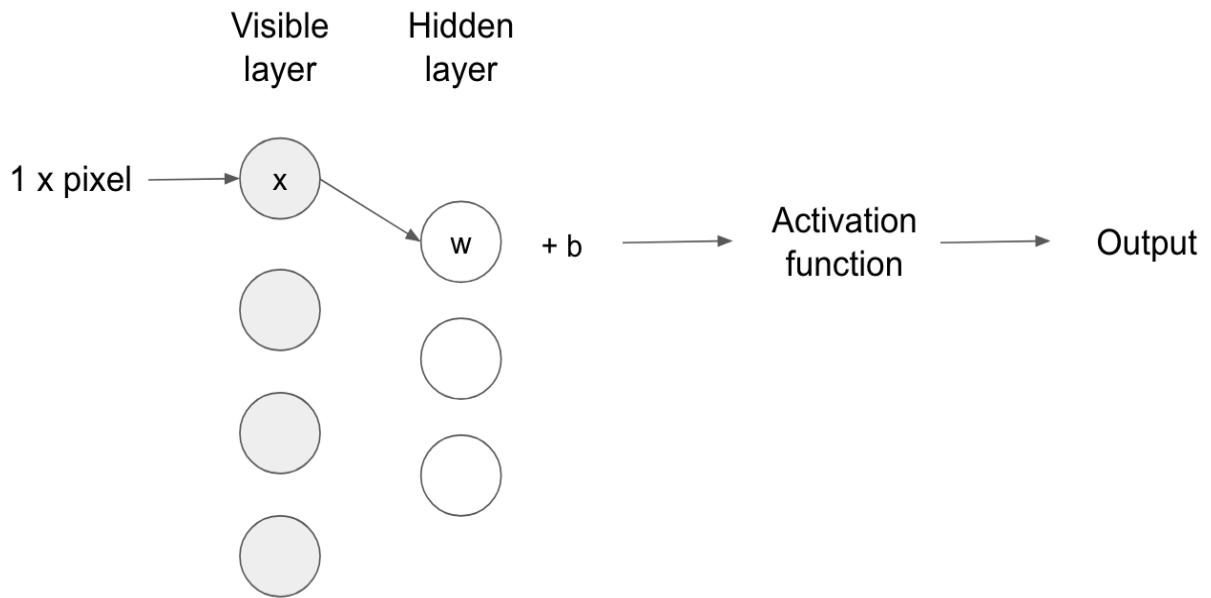


The RBM translates the inputs from the visible layer and translates them into a set of numbers. Through several forward and backward passes, the number is then translated back to reconstruct the inputs. The restriction in the RBM is such that nodes in the same layer are not connected.

A low-level feature is fed into each node of the visible layer from the training dataset. In the case of image classification, each node would receive one pixel value for each pixel in an image:

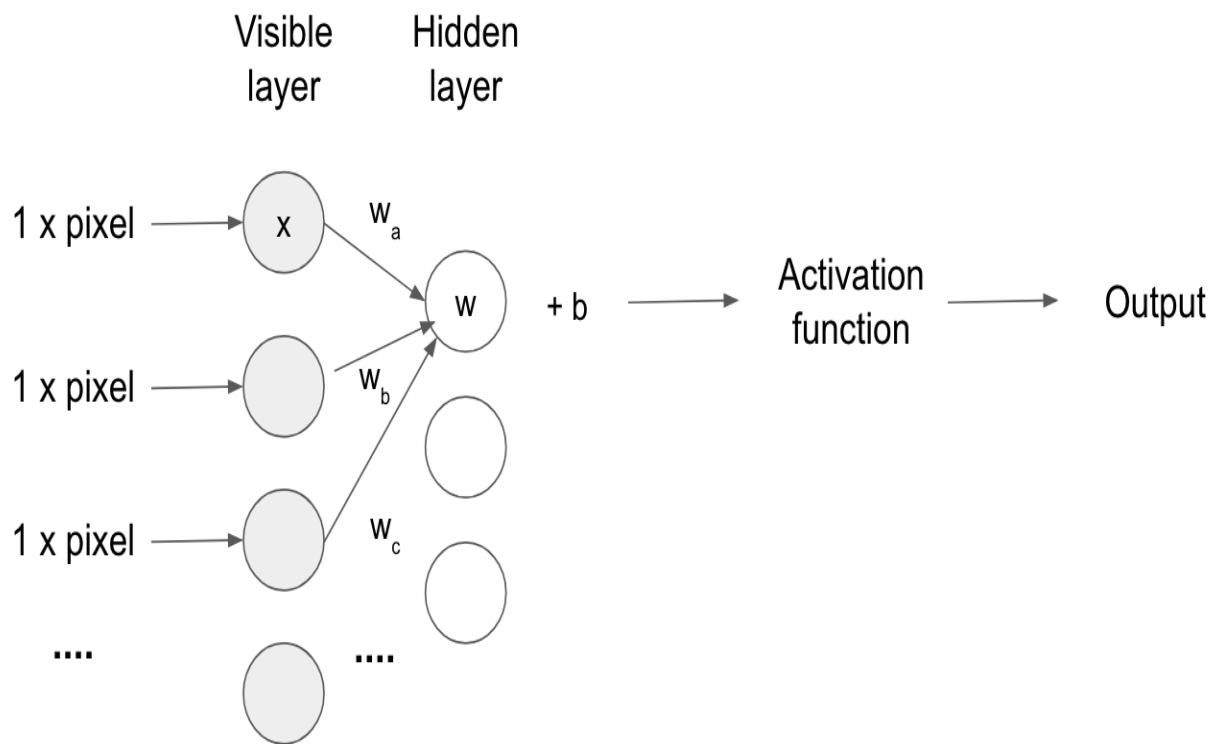


Following one pixel through the network, the input x is multiplied by the weight from the hidden layer and a bias is then added. From here, this is then fed into an activation function, which produces the output, which is essentially the strength of the signal passing through it given the input x , as shown in the following diagram:



Activation Function ((input $x \times$ weight $w) + bias b) = Output$

At each node in the hidden layer, x from each pixel value is multiplied by a separate weight. The products are then summed, and a bias is added. The output of this is then passed through an activation function, producing the output at that single node:



At each point in time, the RBM is in a certain state, which refers to the values of the neurons in the visible v and hidden h layers. The probability of such a state can be given by the following joint distribution function:

$$p(v, h) = \frac{1}{Z} e^{-E(v, h)}$$

$$Z = \sum_{v, h} e^{-E(v, h)}$$

Here, Z is the partition function that is the summation over all possible pairs of visible and hidden vectors.

Training RBMs

There are two main steps an RBM carries out during training:

1. **Gibbs sampling:** The first step in the training process uses Gibbs sampling, which repeats the following process k times:
 - Probability of hidden vector given the input vector; prediction of the hidden values.
 - Probability of the input vector given the hidden vector; prediction of the input values. From this, we obtain another input vector, which was recreated from the original input values.
2. **Contrastive divergence:** RBMs adjust their weights through contrastive divergence. During this process, weights for visible nodes are randomly generated and used to generate hidden nodes. The hidden nodes then use the same weights to reconstruct visible nodes. The weights used to reconstruct the visible nodes are the same throughout. However, the generated nodes are not the same because they aren't connected to each other.

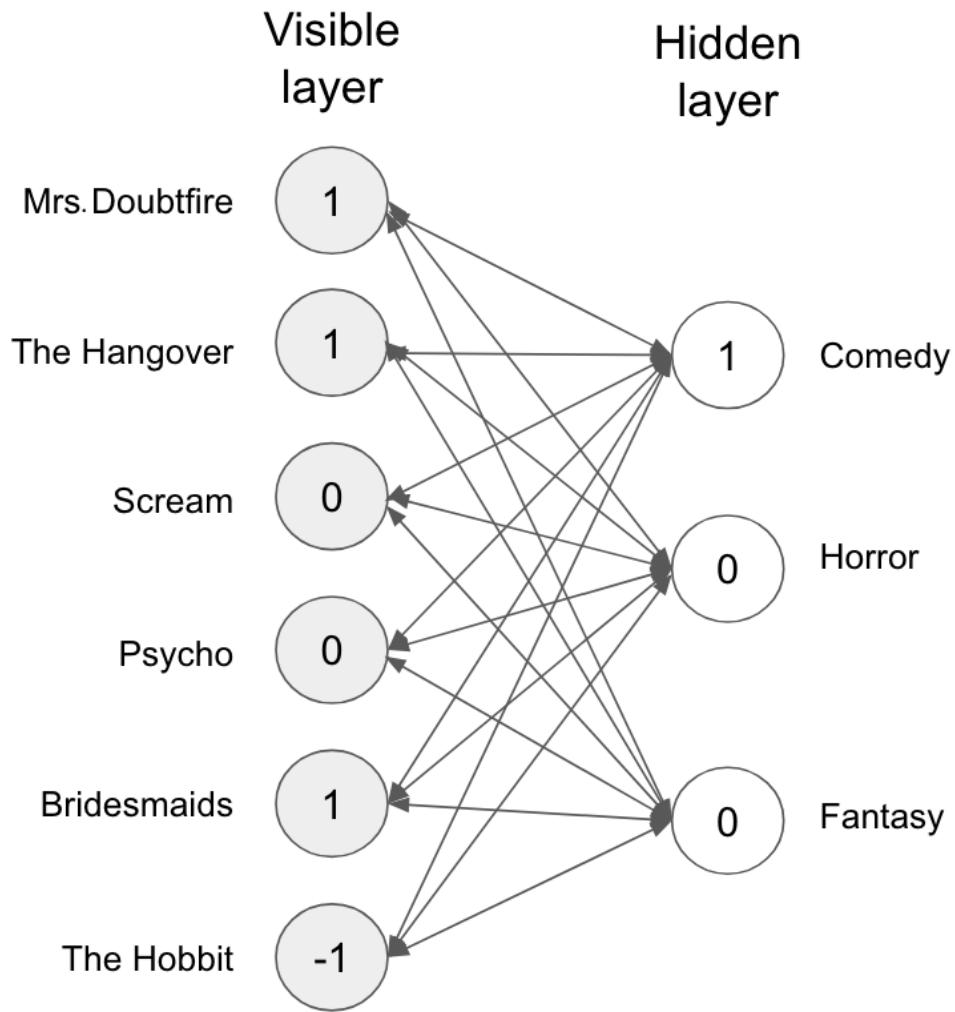
Once an RBM has been trained, it is essentially able to express two things:

- The interrelationship between the features of the input data
- Which features are the most important when identifying patterns

Theoretical example – RBM recommender system

In the context of movies, we can use RBMs to uncover a set of latent factors that represent their genre, and consequently determine which genre of film a person likes. For example, if we were to ask someone to tell us which movies they have watched and whether they liked them or not, we can then represent them as binary inputs (1 or 0) to the RBM. For those movies they haven't seen or haven't told us about, we need to assign a value of -1 so that the network can identify those during training and ignore their associated weights.

Let's consider an example where a user likes *Mrs. Doubtfire*, *The Hangover*, and *Bridesmaids*, does not like *Scream* or *Psycho*, and has not yet seen *The Hobbit*. Given these inputs, the RBM may identify three hidden factors: comedy, horror, and fantasy, which correspond to the genres of the films:



For each hidden neuron, the RBM assigns a probability of the hidden neuron given the input neuron. The final binary values of the neurons are obtained by sampling from the Bernoulli distribution.

In the preceding example, the only hidden neuron that represents the comedy genre becomes active. As such, given the movie ratings input into the RBM, it predicts that the user likes comedy films the most.

For the trained RBM to make predictions on movies the user has not yet seen, based on their preference, the RBM uses the probability of the visible neurons given the hidden neurons. It samples from the

Bernoulli distribution to find out which one of the visible neurons can then become active.

Coded example – RBM recommender system

Continuing in the context of movies, we will now give an example of how we can build an RBM recommender system using the PyTorch library. The goal of the example is to train a model to determine whether a user will like a movie or not.

In this example, we use the MovieLens dataset (<https://grouplens.org/datasets/movielens/>) with 1 million ratings, which was created by the GroupLens Research Group at the University of Minnesota:

1. Firstly, download the datasets. This can be done through terminal commands as follows:

```
|| wget -O moviedataset.zip  
|| http://files.grouplens.org/datasets/movielens/ml-1m.zip  
|| unzip -o moviedataset.zip -d ./data  
|| unzip -o moviedataset.zip -d ./data
```

2. Now import the libraries that we will use:

```
|| import numpy as np  
|| import pandas as pd  
|| import torch  
|| import torch.nn as nn  
|| import torch.nn.parallel  
|| import torch.optim as optim  
|| import torch.utils.data  
|| from torch.autograd import Variable
```

3. Then import the data:

```
|| movies = pd.read_csv('ml-1m/movies.dat', sep = '::', header = None,  
|| engine = 'python', encoding = 'latin-1')  
|| users = pd.read_csv('ml-1m/users.dat', sep = '::', header = None,  
|| engine = 'python', encoding = 'latin-1')  
|| ratings = pd.read_csv('ml-1m/ratings.dat', sep = '::', header =  
|| None, engine = 'python', encoding = 'latin-1')
```

The following screenshot illustrates the structure of our dataset:

movie_id	List Index	user_id	rating
1	0	1	5
1	0	6	4

4. Prepare the testing and training datasets:

```
training_dataset = pd.read_csv('ml-100k/u1.base', delimiter = '\t')
training_dataset = np.array(training_set, dtype = 'int')
test_dataset = pd.read_csv('ml-100k/u1.test', delimiter = '\t')
test_dataset = np.array(test_dataset, dtype = 'int')
```

5. Now we need to prepare a matrix with the users' ratings. The matrix will have users as rows and movies as columns. Zeros are used to represent cases where a user didn't rate a particular movie. We define the `no_users` and `no_movies` variables then consider the `max` value in the training and testing datasets as follows:

```
no_users = int(max(max(training_dataset[:,0]),
max(test_dataset[:,0])))
no_movies = int(max(max(training_dataset[:,1]),
max(test_dataset[:,1])))
```

6. Now we define a function named `convert_dataset` that converts the dataset into a matrix. It does so by creating a loop that will run through the dataset and fetch all of the movies rated by a specific user along with the ratings by that same user. We firstly create a matrix of zeros since there are movies the user didn't rate:

```
def convert_dataset(data):
    converted_data = []
    for id_users in range(1, no_users + 1):
        id_movies = data[:,1][data[:,0] == id_users]
        id_ratings = data[:,2][data[:,0] == id_users]
        movie_ratings = np.zeros(no_movies)
        ratings[id_movies - 1] = id_ratings
        converted_data.append(list(movie_ratings))
    return converted_data
```

```
    | training_dataset = convert_dataset(training_dataset)
    | test_dataset = convert_dataset(test_dataset)
```

7. Now we convert the data into Torch tensors by using the `FloatTensor` utility. This will convert the dataset into PyTorch arrays:

```
    | training_dataset = torch.FloatTensor(training_dataset)
    | test_dataset = torch.FloatTensor(test_dataset)
```

8. In this example, we want to make a binary classification, which is whether the user will like the movie or not. As such, we convert the ratings into zeros and ones. First, however, we replace the existing zeros with -1 in order to represent movies that a user never rated:

```
    | training_dataset[training_dataset == 0] = -1
    | training_dataset[training_dataset == 1] = 0
    | training_dataset[training_dataset == 2] = 0
    | training_dataset[training_dataset >= 3] = 1
    | test_dataset[test_dataset == 0] = -1
    | test_dataset[test_dataset == 1] = 0
    | test_dataset[test_dataset == 2] = 0
    | test_dataset[test_dataset >= 3] = 1
```

9. Now, we need to create a class in order to define the architecture of the RBM. The class initializes the weight and bias by using a random normal distribution. Two types of biases are also defined, where `a` is the probability of the hidden nodes given the visible nodes and `b` is the probability of the visible nodes given the hidden nodes. The class creates a `sample_hidden_nodes` function that takes `x` as an argument and represents the visible neurons. From here, we compute the probability of `h` given `v`, where `h` and `v` represent the hidden and visible nodes respectively. This represents the sigmoid activation function. It is computed as the product of the vector of the weights and `x` plus the bias `a`. Since we are considering the model for binary classification, we return Bernoulli samples of the hidden neurons. From here, we create a `sample_visible_function` function that will sample the visible nodes. Finally, we create the training function. It takes the input vector containing the movie ratings, the visible nodes obtained after k samplings, the vector of probabilities, and the probabilities of the hidden nodes after k samplings:

```

class RBM():
    def __init__(self, num_visible_nodes, num_hidden_nodes):
        self.W = torch.randn(num_hidden_nodes, num_visible_nodes)
        self.a = torch.randn(1, num_hidden_nodes)
        self.b = torch.randn(1, num_visible_nodes)

    def sample_hidden_nodes(self, x):
        wx = torch.mm(x, self.W.t())
        activation = wx + self.a.expand_as(wx)
        p_h_given_v = torch.sigmoid(activation)
        return p_h_given_v, torch.bernoulli(p_h_given_v)

    def sample_visible_nodes(self, y):
        wy = torch.mm(y, self.W)
        activation = wy + self.b.expand_as(wy)
        p_v_given_h = torch.sigmoid(activation)
        return p_v_given_h, torch.bernoulli(p_v_given_h)

    def train(self, v0, vk, ph0, phk):
        self.W += torch.mm(v0.t(), ph0) - torch.mm(vk.t(), phk)
        self.b += torch.sum((v0 - vk), 0)
        self.a += torch.sum((ph0 - phk), 0)

```

10. Now we define our model parameters:

```

num_visible_nodes = len(training_dataset[0])
num_hidden_nodes = 200
batch_size = 100
rbm = RBM(num_visible_nodes, num_hidden_nodes)

```

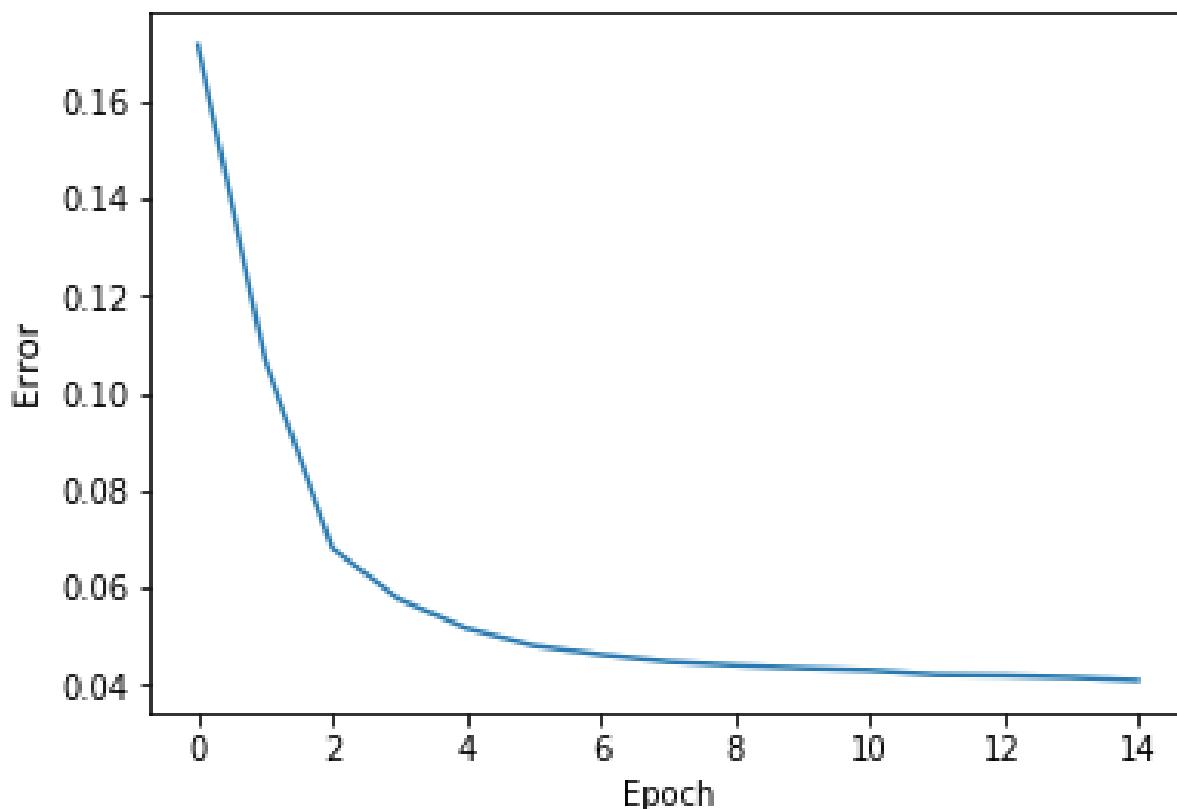
11. From here, we can train the model for each epoch:

```

nb_epoch = 10
for epoch in range(1, nb_epoch + 1):
    train_loss = 0
    s = 0.
    for id_user in range(0, nb_users - batch_size, batch_size):
        vk = training_dataset[id_user:id_user+batch_size]
        v0 = training_dataset[id_user:id_user+batch_size]
        ph0, _ = rbm.sample_hidden_nodes(v0)
        for k in range(10):
            _, hk = rbm.sample_hidden_nodes(vk)
            _, vk = rbm.sample_visible_nodes(hk)
            vk[v0<0] = v0[v0<0]
        phk, _ = rbm.sample_hidden_nodes(vk)
        rbm.train(v0, vk, ph0, phk)
        train_loss += torch.mean(torch.abs(v0[v0>=0] - vk[v0>=0]))
        s += 1.
    print('epoch: '+str(epoch)+ ' loss: '+str(train_loss/s))

```

We can plot the error across epochs during training:

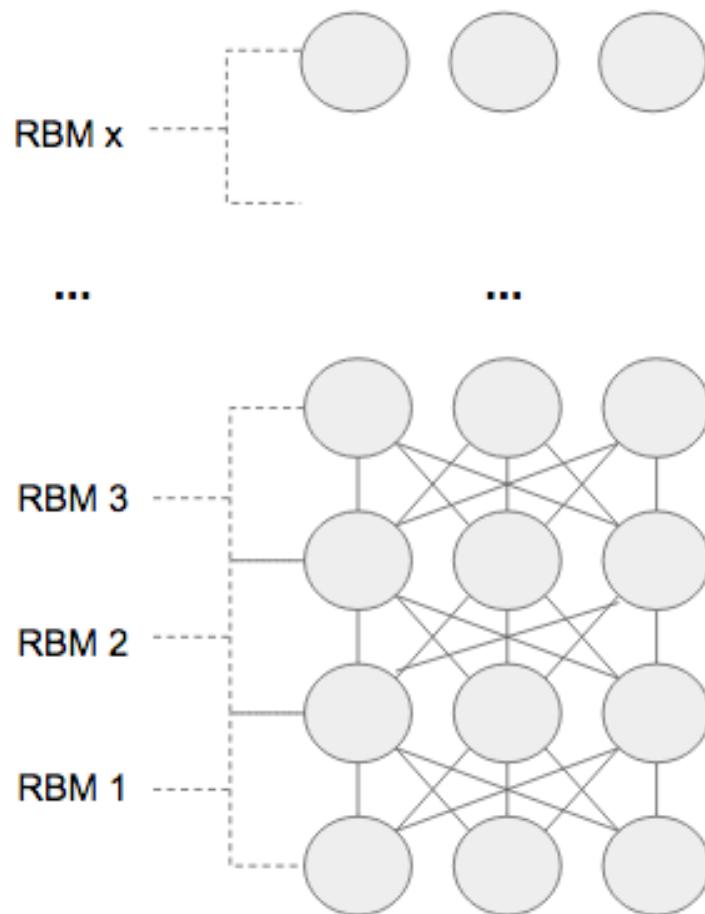


This can help us to determine how many epochs we should run the training for. It shows that after six epochs, the improved performance rate drops and hence we should consider stopping the training at this stage.

We have seen the coded example of implementing a recommender system in RBM, now let's briefly walk through a DBN architecture.

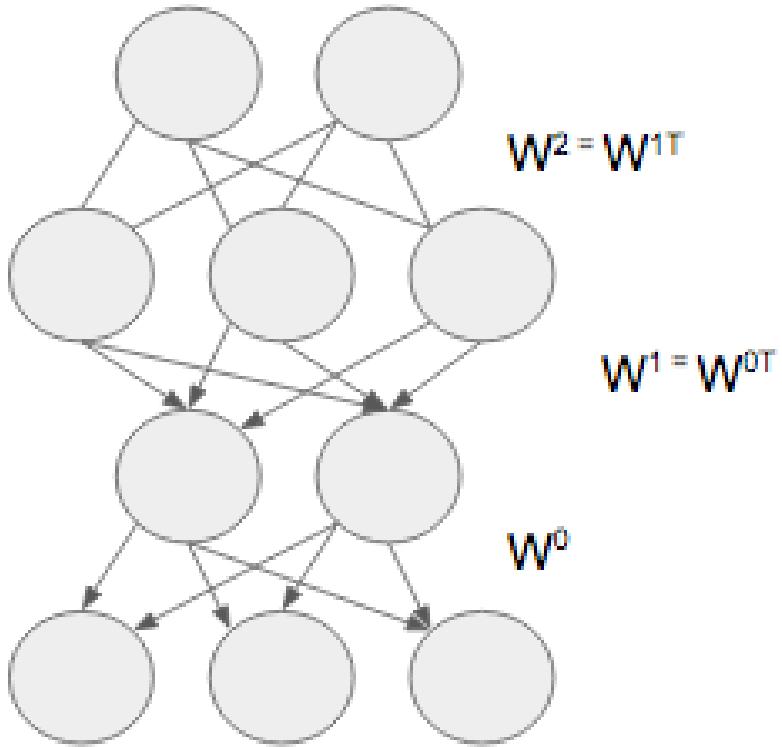
DBN architecture

A DBN is a multilayer belief network where each layer is an RBM stacked against one another. Apart from the first and final layers of the DBN, each layer serves as both a hidden layer to the nodes before it and as the input layer to the nodes that come after it:



Two layers in the DBN are connected by a matrix of weights. The top two layers of a DBN are undirected, which gives a symmetric connection between them, forming an associative memory. The lower two layers have direct connections to the layers above. The

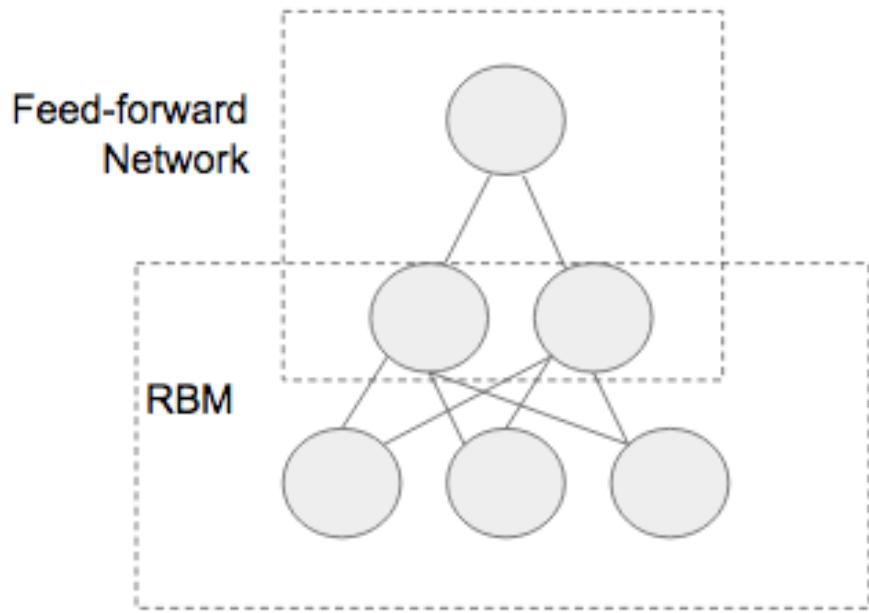
sense of direction converts associative memory into observed variables:



The two most significant properties of DBNs are as follows:

- A DBN learns top-down, generative weights via an efficient, layer-by-layer procedure. These weights determine how the variables in one layer depend on the layer above.
- Once training is complete, the values of the hidden variables in each layer can be inferred by a single bottom-up pass. The pass begins with a visible data vector in the lower layer and uses its generative weights in the opposite direction.

The probability of a joint configuration network over both visible and hidden layers depends on the joint configuration network's energy compared with the energy of all other joint configuration networks:



Once the pretraining phase of the DBN has been completed by the RBM stack, a feedforward network can then be used for the fine-tuning phase in order to create a classifier or simply help cluster unlabeled data in an unsupervised learning scenario.

Fine-tuning

Fine-tuning aims to find the optimal values of the weights between the layers. It tweaks the original features in order to obtain more precise boundaries of the classes. In order to help the model associate patterns and features to the datasets, a small labeled dataset is used.

Fine-tuning can be applied as a stochastic bottom-up pass and then used to adjust the top-down weights. Once the top is reached, recursion is applied to the top layer. In order to fine-tune further, we can do a stochastic top-down pass and adjust the bottom-up weights.

Summary

In this chapter, we explained what autoencoders are and different variations of them. Throughout the chapter, we gave some coded examples of how they can be applied to the MNIST dataset. We later introduced RBMs and explained how these can be developed into a DBN along with some additional examples.

In the next chapter, we will introduce generative adversarial networks. We will show how they can be used to generate both images and text.

Further reading

Refer to the following for further information:

- *Tutorial on Variational Autoencoders*: <https://arxiv.org/abs/1606.05908>
- *CS598LAZ – Variational Autoencoders*: http://slazebni.cs.illinois.edu/spring17/lec12_vae.pdf
- *Auto-Encoding Variational Bayes*: <https://arxiv.org/abs/1312.6114>
- *Deep Learning Book*: <https://www.deeplearningbook.org/contents/autoencoders.html>
- *A Fast Learning Algorithm for Deep Belief Nets*: <http://www.cs.toronto.edu/~fritz/absps/ncfast.pdf>
- *Training restricted Boltzmann machines: An introduction*: <https://www.sciencedirect.com/science/article/abs/pii/S0031320313002495>
- *Deep Boltzmann Machines*: <http://proceedings.mlr.press/v5/salakhutdinov09a/salakhutdinov09a.pdf>
- *A Practical Guide to Training Restricted Boltzmann Machines*: <https://www.cs.toronto.edu/~hinton/absps/guideTR.pdf>
- *Deep Belief Networks*: https://link.springer.com/chapter/10.1007/978-3-319-06938-8_8
- *Hands-On Neural Networks*: <https://www.amazon.co.uk/Hands-Neural-Networks-ebook/dp/B07SKDSGB6/>

Working with Generative Adversarial Networks

All the examples that we saw in the previous chapters were focused on solving problems such as classification or regression. This chapter is very interesting and important for understanding how deep learning is evolving to solve problems in unsupervised learning.

In this chapter, we will train networks that learn how to create the following:

- Images based on content and a particular artistic style, popularly called **style transfer**
- Generating faces of new people using a particular type of **generative adversarial network (GAN)**

These techniques form the basis of most of the advanced research that is happening in the deep learning space. Going into the exact specifics of each of the subfields, such as GANs and language modeling is beyond the scope of this book, as they deserve a separate book for themselves. We will learn how they work in general and the process of building them in PyTorch.

The following topics will be covered in this chapter:

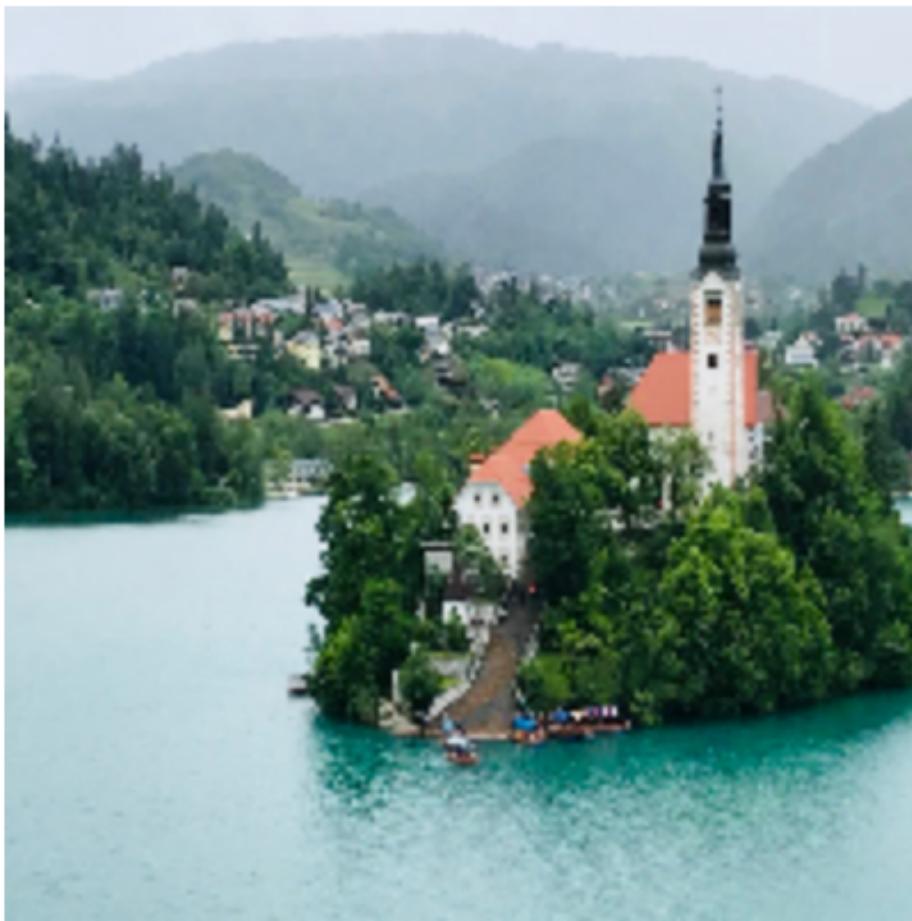
- Neural style transfer
- Introducing GANs
- DCGANs

Neural style transfer

We humans generate artwork with different levels of accuracy and complexity. Though the process of creating art can be a very complex process, it can be seen as a combination of the two most important factors, namely, what to draw and how to draw. What to draw is inspired by what we see around us, and how we draw will also take influences from certain things that are found around us. This could be an oversimplification from an artist's perspective, but for understanding how we can create artwork using deep learning algorithms, it is very useful.

We will train a deep learning algorithm to take content from one image and then draw it according to a specific artistic style. If you are an artist or in the creative industry, you can directly use the amazing research that has gone on in recent years to improve this and create something cool within the domain you work in. Even if you are not, it still introduces you to the field of generative models, where networks generate new content.

Let's understand what is done in neural style transfer at a high-level, and then dive into details, along with the PyTorch code required to build it. The style transfer algorithm is provided with a content image (C) and a style image (S)—the algorithm has to generate a new image (O) that has the content from the content image and the style from the style image. This process of creating neural style transfer was introduced by Leon Gatys and others in 2015 in their paper, *A Neural Algorithm of Artistic Style* (<https://arxiv.org/pdf/1508.06576.pdf>). The following is the content image (C) that we will be using:

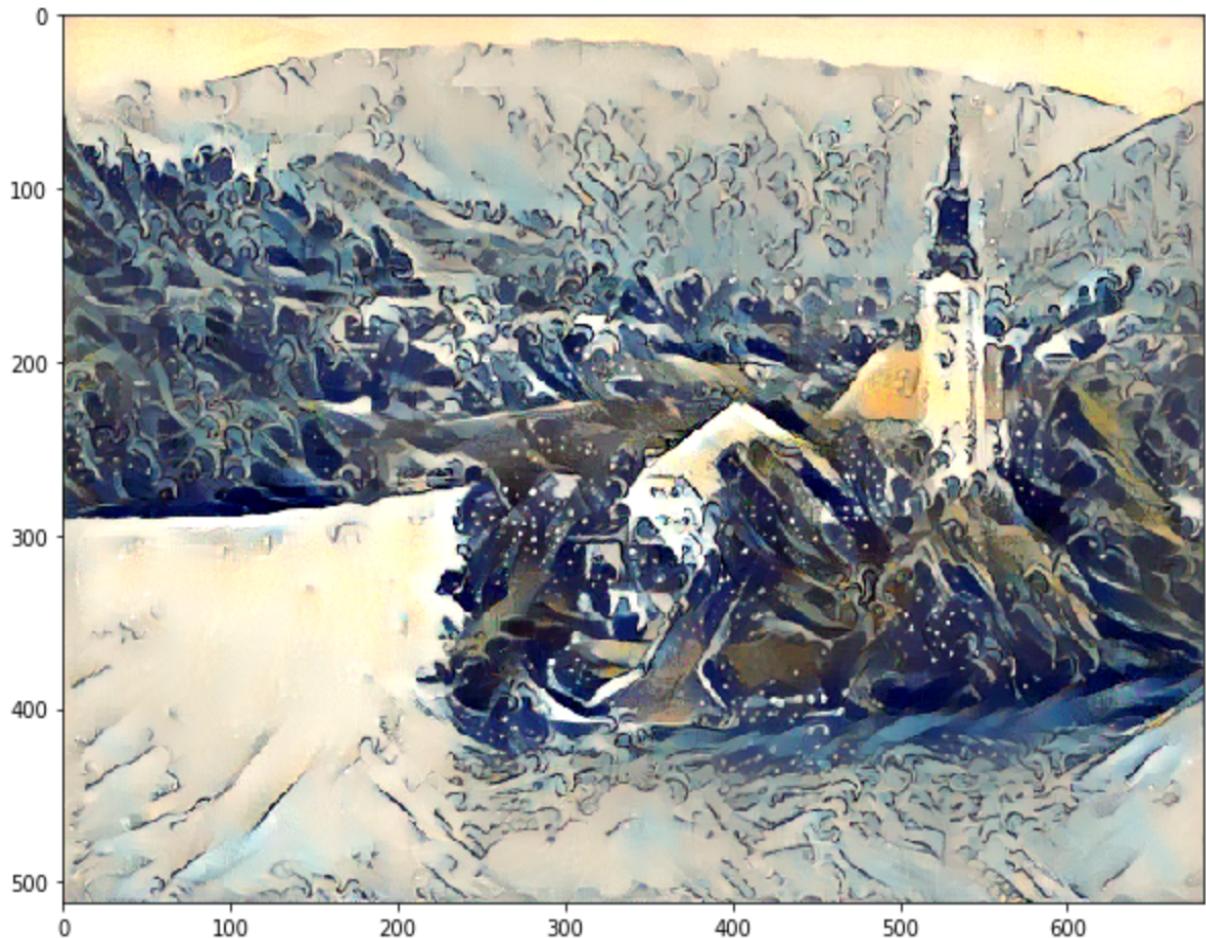


And the following is the style image (S):



Source of the preceding image: *The Great Wave Off Kanagawa* from by Katsushika Hokusai (http://commons.wikimedia.org/wiki/File:The_Great_Wave_off_Kanagawa.jpg)

And this is the image that we will get as the result:



The idea behind style transfer becomes clear when you understand how **convolutional neural networks (CNNs)** work. When CNNs are trained for object recognition, the early layers of a trained CNN learn very generic information like lines, curves, and shapes. The last layers in a CNN capture the higher-level concepts from an image, such as eyes, buildings, and trees. So the values of the last layers of similar images tend to be closer. We take the same concept and apply it for content loss. The last layer for the content image and the generated image should be similar, and we calculate the similarity using the mean square error (MSE). We use our optimization algorithms to bring down the loss value.

The style of the image is generally captured across multiple layers in a CNN by a technique called the gram matrix. The gram matrix

calculates the correlation between the feature maps captured across multiple layers. The gram matrix gives a measure of calculating the style. Similarly styled images have similar values for the gram matrix. The style loss is also calculated using the MSE between the gram matrix of the style image and the generated image.

We will use a pretrained VGG19 model, provided in the TorchVision models. The steps required for training a style transfer model are similar to any other deep learning models, except for the fact that calculating losses is more involved than for a classification or regression model. The training of the neural style algorithm can be broken down to the following steps:

1. Loading data.
2. Creating a VGG19 model.
3. Defining content loss.
4. Defining style loss.
5. Extracting losses across layers from the VGG model.
6. Creating an optimizer.
7. Training—generating an image similar to the content image, and a style similar to the style image.

Loading the data

Loading data is similar to what we saw for solving image classification problems in Chapter 3, *Diving Deep into Neural Networks*. We will be using the pretrained VGG model, so we have to normalize the images using the same values on which the pretrained model is trained.

The following code shows how we can do this. The code is mostly self-explanatory as we already discussed it in detail in the previous chapters:

```
image_size = 512
is_cuda = torch.cuda.is_available()
preprocessing = transforms.Compose([transforms.Resize(image_size),
                                    transforms.ToTensor(),
                                    transforms.Lambda(lambda x:
x[torch.LongTensor([2,1,0])]),
                                    transforms.Normalize(mean=[0.40760392,
0.45795686, 0.48501961],
                                             std=[1,1,1]),
                                    transforms.Lambda(lambda x: x.mul_(255)),
])
processing = transforms.Compose([transforms.Lambda(lambda x:
x.mul_(1./255)),
                                    transforms.Normalize(mean=[-0.40760392,
-0.45795686, -0.48501961],
                                             std=[1,1,1]),
                                    transforms.Lambda(lambda x:
x[torch.LongTensor([2,1,0])]),
])
postprocess = transforms.Compose([transforms.ToPILImage()])

def postprocess_b(tensor):
    t = processing(tensor)
    t[t>1] = 1
    t[t<0] = 0
    img = postprocess(t)
    return img
```

In this code, we defined three functionalities, preprocess does all the preprocessing required and uses the same values for normalization as those with which the VGG model was trained. The output of the model needs to be normalized back to its original values; the

`processing` function does the processing required. The generated model may be out of the range of accepted values, and the `postprocess_b` function limits all the values greater than one to one, and values that are less than zero to zero.

Now we define the `loader` function, which loads the image, applies the `preprocessing` transformation, and converts it into a variable:

```
def loader(image_name):
    image = Image.open(image_name)
    image = Variable(preprocessing(image))
    # fake batch dimension required to fit network's input dimensions
    image = image.unsqueeze(0)
    return image
```

The following function loads the style and content image:

```
style_image = loader("Images/style_image.jpg")
content_image = loader("Images/content_image.jpg")
```

We can either create an image with noise (random numbers), or we can use the same content image. We will use the content image in this case. The following code creates the content image:

```
output_image = Variable(content_image.data.clone(), requires_grad=True)
```

We will use an optimizer to tune the values of the `output_image` variable in order for the image to be closer to the content image and style image. For that reason, we are asking PyTorch to maintain the gradients by mentioning `requires_grad=True`.

Creating the VGG model

We will load a pretrained model from `torchvisions.models`. We will be using this model only for extracting features, and the PyTorch VGG model is defined in such a way that all the convolutional blocks will be in the features module and the fully connected, or linear, layers are in the classifier module. Since we will not be training any of the weights or parameters in the VGG model, we will also freeze the model, as the following code demonstrates:

```
| vgg = vgg19(pretrained=True).features
| for param in vgg.parameters():
|     param.requires_grad = False
```

In this code, we created a VGG model, used only its convolution blocks, and froze all of the parameters of the model as we will be using it only for extracting features.

Content loss

The **content loss** is the distance between the input and the output images. The aim is to preserve the original content of the image. It is the MSE calculated on the output of a particular layer, extracted by passing two images through the network. We extract the outputs of the intermediate layers from the VGG by using the `register_forward_hook` functionality, passing in the content image and the image to be optimized.

We calculate the MSE obtained from the outputs of these layers, as described in the following code:

```
| target_layer = dummy_fn(content_img)
| noise_layer = dummy_fn(noise_img)
| criterion = nn.MSELoss()
| content_loss = criterion(target_layer,noise_layer)
```

We will implement `dummy_fn` for this code in the coming sections. For now, all we know is that the `dummy_fn` function returns the outputs of particular layers by passing an image. We pass the outputs generated by passing the content image and noise image to the MSE loss function.

Style loss

The style loss is calculated across multiple layers. Style loss is the MSE of the gram matrix generated for each feature map. The gram matrix represents the correlation value of its features. Let's understand how gram matrix works by using the following diagram and a code implementation.

The following table shows the output of a feature map of dimension [2, 3, 3, 3], having the column attributes **Batch_size**, **Channels**, and **Values**:

Batch_size	Channels	Values		
1	1	0.1	0.1	0.1
		0.2	0.2	0.2
		0.3	0.3	0.3
	2	0.2	0.2	0.2
		0.2	0.2	0.2
		0.2	0.2	0.2
	3	0.3	0.3	0.3
		0.3	0.3	0.3
		0.3	0.3	0.3
2	1	0.1	0.1	0.1
		0.2	0.2	0.2
		0.3	0.3	0.3
	2	0.2	0.2	0.2
		0.2	0.2	0.2
		0.2	0.2	0.2
	3	0.3	0.3	0.3
		0.3	0.3	0.3
		0.3	0.3	0.3

To calculate the gram matrix, we flatten all the values per channel and then find its correlation by multiplying with its transpose, as shown in the following table:

Batch_size	Channels	BMM(Gram Matrix, Transpose(Gram Matrix))
1	1	(0.1,0.1,0.1,0.2,0.2,0.2,0.3,0.3,0.3,)
	2	(0.2,0.2,0.2,0.2,0.2,0.2,0.2,0.2,0.2)
	3	(0.3,0.3,0.3,0.3,0.3,0.3,0.3,0.3,0.3,0.3)
2	1	(0.1,0.1,0.1,0.2,0.2,0.2,0.3,0.3,0.3,)
	2	(0.2,0.2,0.2,0.2,0.2,0.2,0.2,0.2,0.2)
	3	(0.3,0.3,0.3,0.3,0.3,0.3,0.3,0.3,0.3,0.3)

All we did is flatten all the values, with respect to each channel, to a single vector or tensor. The following code implements this:

```
| class GramMatrix(nn.Module):
|     def forward(self, input):
|         b,c,h,w = input.size()
|         features = input.view(b,c,h*w)
|         gram_matrix = torch.bmm(features, features.transpose(1,2))
|         gram_matrix.div_(h*w)
|         return gram_matrix
```

We implement the `GramMatrix` function as another PyTorch module with a `forward` function so that we can use it like a PyTorch layer. We are extracting the different dimensions from the input image in this line:

```
| b,c,h,w = input.size()
```

Here, `b` represents batch, `c` represents filters or channels, `h` represents height, and `w` represents width. In the next step, we will use the following code to keep the batch and channel dimensions intact and flatten all the values along the height and width dimension as shown in the preceding diagram:

```
| features = input.view(b,c,h*w)
```

The gram matrix is calculated by multiplying the flattening values along with its transposed vector. We can do it by using the PyTorch batch matrix multiplication function, provided as `torch.bmm()`, as shown in the following code:

```
| gram_matrix = torch.bmm(features, features.transpose(1,2))
```

We finish normalizing the values of the gram matrix by dividing it by the number of elements. This prevents a particular feature map with a lot of values dominating the score. Once `GramMatrix` is calculated, it becomes simple to calculate the style loss, which is implemented in this code:

```
| class StyleLoss(nn.Module):
|     def forward(self, inputs, targets):
|         out = nn.MSELoss()(GramMatrix()(inputs), targets)
|         return (out)
```

The `StyleLoss` class is implemented as another PyTorch layer. It calculates the MSE between the input `GramMatrix` values and the style image `GramMatrix` values.

Extracting the losses

Just like we extracted the activation of a convolution layer using the `register_forward_hook()` function, we can extract losses of different convolutional layers required to calculate style loss and content loss. The one difference in this case is that instead of extracting from one layer, we need to extract outputs of multiple layers. The following class integrates the required change:

```
class LayerActivations():
    features=[]

    def __init__(self,model,layer_numbers):
        self.hooks = []
        for layer_num in layer_numbers:
            self.hooks.append(model[layer_numbers].register_forward_hook(self.hook_fn))

    def hook_fn(self,module,input,output):
        self.features.append(output)

    def remove(self):
        for hook in self.hooks:
            hook.remove()
```

The `__init__` method takes the model on which we need to call the `register_forward_hook` method and the layer numbers for which we need to extract the outputs. The `for` loop in the `__init__` method iterates through the layer numbers and registers the forward hook required to pull the outputs.

The `hook_fn` function passed to the `register_forward_hook` method is called by PyTorch after that layer on which the `hook_fn` function is registered. Inside the function, we capture the output and store it in the `features` array.

We need to call the `remove` function once when we don't want to capture the outputs. Forgetting to invoke the `remove` methods can

cause out-of-memory exceptions as all the outputs get accumulated.

Let's write another utility function that can extract the outputs required for the style and content images. The following function does the same:

```
def extract_layers(layers,image,model=None):  
    la = LayerActivations(model,layers)  
    la.features = []  
    out = model(image)  
    la.remove()  
    return la.features
```

Inside the `extract_layers` function, we create objects for the `LayerActivations` class by passing in the model and the layer numbers. The features list may contain outputs from previous runs, so we are reinitializing to an empty list. Then we pass in the image through the model, and we are not going to use the outputs. We are more interested in the outputs generated in the features array. We call the remove method to remove all the registered hooks from the model and return the features. The following code shows how we extract the targets required for style and content image:

```
content_targets = extract_layers(content_layers,content_img,model=vgg)  
style_targets = extract_layers(style_layers,style_img,model=vgg)
```

Once we extract the targets, we need to detach the outputs from the graphs that created them. Remember that all these outputs are PyTorch variables, which maintain information on how they are created. But, for our case, we are interested in only the output values and not the graph, as we are not going to update either the style image or the content image. The following code illustrates this technique:

```
content_targets = [t.detach() for t in content_targets]  
style_targets = [GramMatrix()(t).detach() for t in style_targets]
```

Once we have detached, let's add all the targets into one list. The following code illustrates this technique:

```
| targets = style_targets + content_targets
```

When calculating the style loss and content loss, we passed on two lists called content layers and style layers. Different layer choices will have an impact on the quality of the image generated. Let's pick the same layers as the authors of the paper mentioned. The following code shows the choice of layers that we are using here:

```
| style_layers = [1, 6, 11, 20, 25]
| content_layers = [21]
| loss_layers = style_layers + content_layers
```

The optimizer expects a single scalar quantity to minimize. To achieve a single scalar value, we sum up all the losses that have arrived at different layers. It is common practice to do a weighted sum of these losses, and again we pick the same weights as used in the paper's implementation in the GitHub repository (<https://github.com/leongatys/PytorchNeuralStyleTransfer>). Our implementation is a slightly modified version of the author's implementation. The following code describes the weights being used, which are calculated by the number of filters in the selected layers:

```
| style_weights = [1e3/n**2 for n in [64, 128, 256, 512, 512]]
| content_weights = [1e0]
| weights = style_weights + content_weights
```

To visualize this, we can print the VGG layers. Take a minute to observe which layers we are picking, and you can experiment with different layer combinations. We will use the following code to print the VGG layers:

```
| print(vgg)
```

This results in the following output:

```
#Results
Sequential(
  (0): Conv2d (3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace)
  (2): Conv2d (64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace)
  (4): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
```

```

    (5) : Conv2d (64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (6) : ReLU(inplace)
    (7) : Conv2d (128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (8) : ReLU(inplace)
    (9) : MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
   (10) : Conv2d (128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
   (11) : ReLU(inplace)
   (12) : Conv2d (256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
   (13) : ReLU(inplace)
   (14) : Conv2d (256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
   (15) : ReLU(inplace)
   (16) : Conv2d (256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
   (17) : ReLU(inplace)
   (18) : MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
   (19) : Conv2d (256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
   (20) : ReLU(inplace)
   (21) : Conv2d (512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
   (22) : ReLU(inplace)
   (23) : Conv2d (512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
   (24) : ReLU(inplace)
   (25) : Conv2d (512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
   (26) : ReLU(inplace)
   (27) : MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
   (28) : Conv2d (512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
   (29) : ReLU(inplace)
   (30) : Conv2d (512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
   (31) : ReLU(inplace)
   (32) : Conv2d (512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
   (33) : ReLU(inplace)
   (34) : Conv2d (512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
   (35) : ReLU(inplace)
   (36) : MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
)

```

We have to define the loss functions and the optimizer to generate artistic images. We will initialize both of them in the following section.

Creating a loss function for each layer

We have already defined loss functions as PyTorch layers. So, let's create the loss layers for different style losses and content losses. The following code defines the function:

```
loss_fns = [StyleLoss()] * len(style_layers) + [nn.MSELoss()] *  
len(content_layers)
```

The `loss_fns` function is a list containing a bunch of style loss objects and content loss objects based on the length of the arrays created.

Creating the optimizer

In general, we pass in the parameters of a network like VGG to be trained. But, in this example, we are using VGG models as feature extractors, and so we cannot pass the VGG parameters. Here, we will only provide the parameters of the `opt_img` variable that we will optimize to make the image have the required content and style. The following code creates the optimizer that optimizes its values:

```
| optimizer = optim.LBFGS([output_image]);
```

Now we have all the components for training.

Training the model

The training method is different compared to the other models that we have trained till now. Here, we need to calculate loss at multiple layers, and every time the optimizer is called, it will change the input image so that its content and style gets close to the target's content and style. Let's look at the code used for training, and then we will walk through the important steps in the training:

```
maximum_iterations = 500
show_iteration-1 = 50
n_iter=[0]

optimizer = optim.LBFGS([output_image]);
n_iteration=[0]

while n_iteration[0] <= maximum_iterations:

    def closure():
        optimizer.zero_grad()

        out = extract_layers(loss_layers,output_image,model=vgg)
        layer_losses = [weights[a] * loss_fns[a](A, targets[a]) for a,A in enumerate(out)]
        loss = sum(layer_losses)
        loss.backward()
        n_iteration[0]+=1
        if n_iteration[0]%show_iteration == (show_iteration-1):
            print('Iteration: %d, loss: %f'%(n_iteration[0]+1,
loss.data[0]))

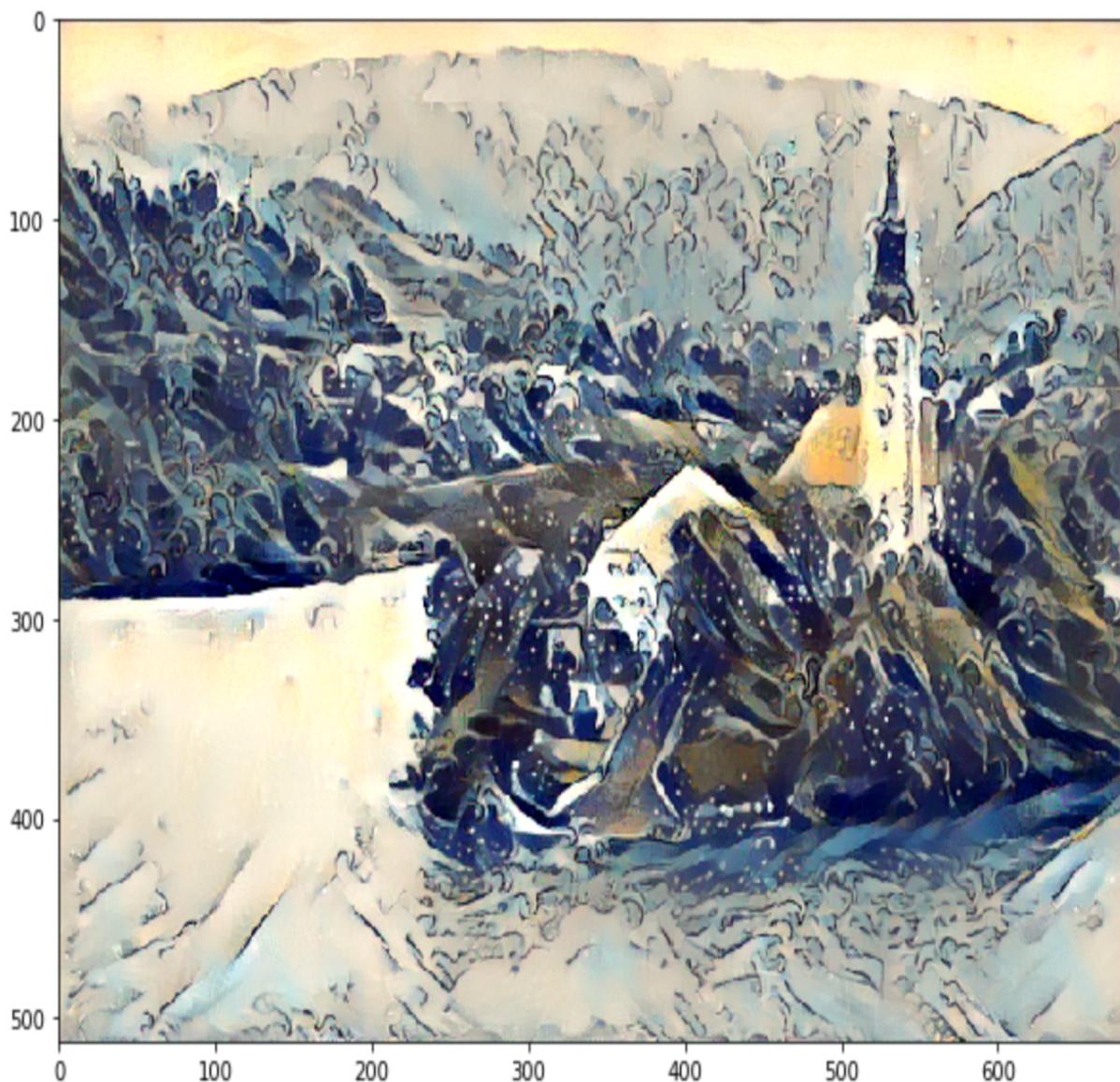
    return loss

optimizer.step(closure)
```

We are running the training loop for 500 iterations. For every iteration, we calculate the output from different layers of the VGG model using our `extract_layers` function. In this case, the only thing that changes is the values of `output_image`, which will contain our style image. Once the outputs are calculated, we calculate the losses by iterating through the outputs and passing them to the corresponding loss functions, along with their respective targets. We sum up all the losses and call the backward function. At the end of the closure function, the loss is

returned. The closure method is called along with the `optimizer.step` method for `max_iterations`. If you are running on a GPU, it could take a few minutes to run; if you are running on a CPU, try reducing the size of the image to make it run faster.

After running for 500 epochs, the resulting image on my machine looks as shown here. Try different combinations of content and style to generate interesting images:

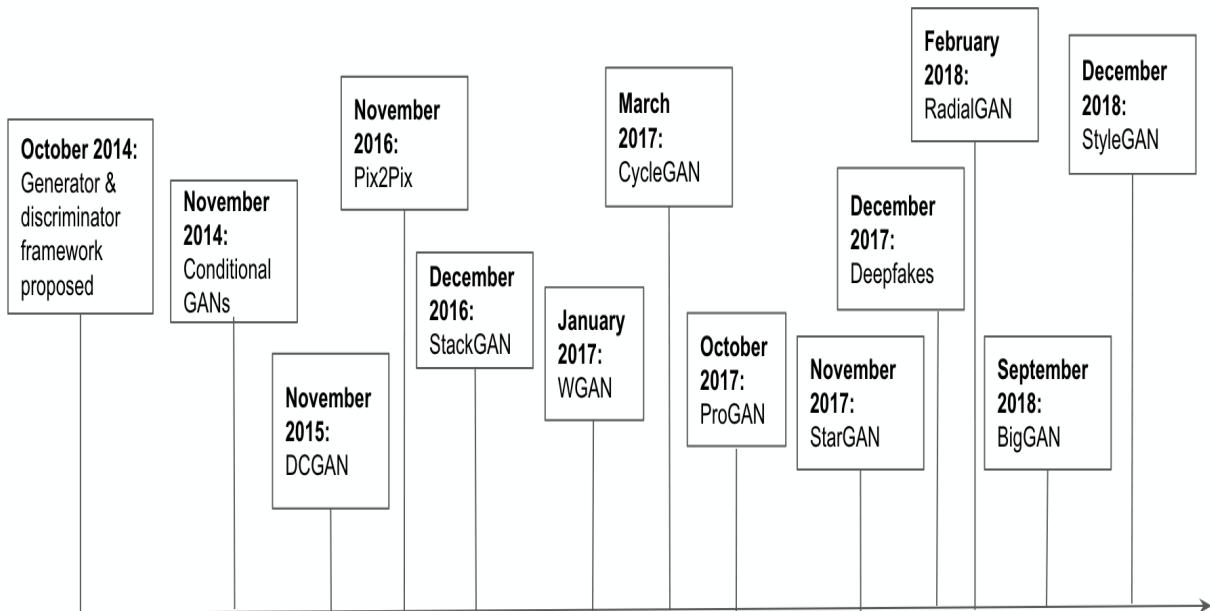


In the next section, let's go ahead and generate human faces using **deep convolutional generative adversarial networks (DCGANs)**.

Introducing GANs

GANs were introduced by Ian Goodfellow in 2014 and have become very popular. There have been many significant developments to GAN research in recent times, and the following timeline shows some of the most noteworthy advances and key developments in GAN research:

Timeline: key developments in GAN research



In this chapter, we will focus on the PyTorch implementation of DCGAN. However, there is a very useful GitHub repository that provides a host of PyTorch implementation examples of the GANs shown in the timeline along with others. It can be accessed via the following link: <https://github.com/eriklindernoren/PyTorch-GAN>.

The GAN addresses the problem of unsupervised learning by training two deep neural networks, called a generator and discriminator, which compete with each other. In the course of training, both eventually become better at the tasks that they perform.

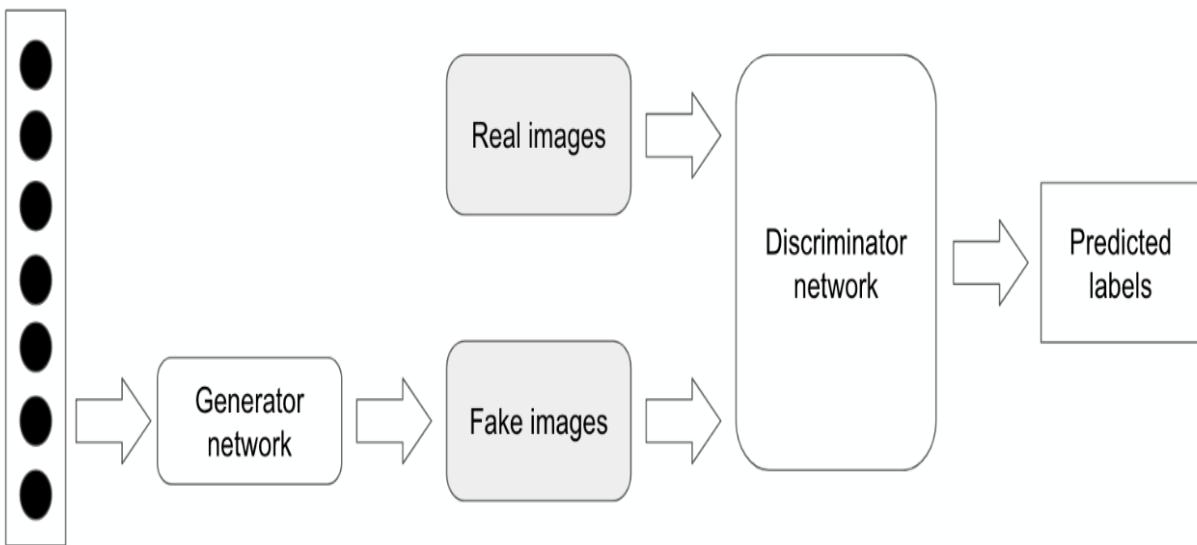
GANs are intuitively understood using the case of a counterfeiter (generator) and the police (discriminator). Initially, the counterfeiter shows the police fake money. The police identifies it as fake and explains to the counterfeiter why it is fake. The counterfeiter makes new fake money based on the feedback it received. The police finds it's fake and informs the counterfeiter why it is fake. It repeats this a huge number of times until the counterfeiter is able to make fake money, which the police is unable to recognize. In the GAN scenario, we end up with a generator that generates fake images that are quite similar to the real ones, and a classifier becomes great at identifying a fake from the real thing.

GAN is a combination of a forger network and an expert network, each being trained to beat the other. The generator network takes a random vector as input and generates a synthetic image. The discriminator network takes an input image, and predicts whether the image is real or fake. We pass the discriminator network either a real image or a fake image.

The generator network is trained to produce images and fool the discriminator network into believing they are real. The discriminator network is also constantly improving at not getting fooled, as we pass the feedback while training it.

The following diagram depicts the architecture of a GAN model:

D-dimensional
noise vector



Though the idea of GANs sounds simple in theory, training a GAN model that actually works is very difficult as there are two deep neural networks that need to be trained in parallel.



The DCGAN is one of the early models that demonstrated how to build a GAN that learns by itself and generates meaningful images. You can learn more about it here: <https://arxiv.org/pdf/1511.06434.pdf>. We will walk through each of the components of this architecture along with some of the reasoning behind it and how this can be implemented in PyTorch.

DCGAN

In this section, we will implement different parts of training a GAN architecture, based on the DCGAN paper I mentioned in the preceding information box. Some of the important parts of training a DCGAN include the following:

- A generator network, which maps a latent vector (list of numbers) of some fixed dimension to images of some shape. In our implementation, the shape is (3, 64, 64).
- A discriminator network, which takes as input an image generated by the generator or from the actual dataset, and maps to that a score estimating if the input image is real or fake.
- Defining loss functions for the generator and discriminator.
- Defining an optimizer.

Let's explore each of these sections in detail. The implementation provides a more detailed explanation of the code that is available in the PyTorch GitHub repository: <https://github.com/pytorch/examples/tree/master/dcgan>.

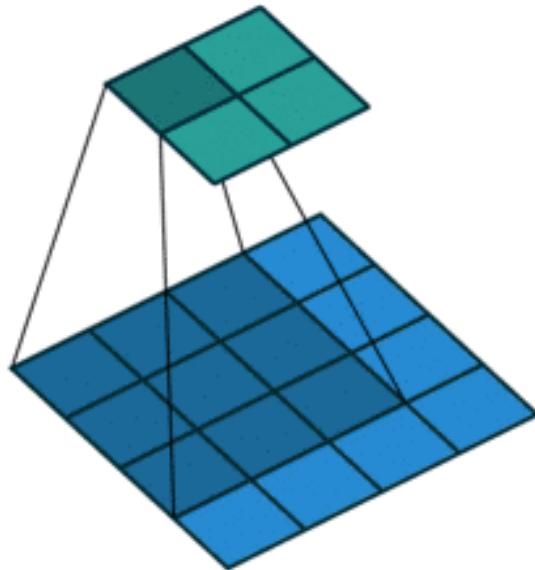
Defining the generator network

The generator network takes a random vector of fixed dimension as input, and applies a set of transposed convolutions, batch normalization, and ReLU activation to it, and generates an image of the required size. Before looking into the generator implementation, let's look at defining transposed convolution and batch normalization.

Transposed convolutions

Transposed convolutions are also called fractionally strided convolutions. They work in the opposite way to how convolution works. Intuitively, they try to calculate how the input vector can be mapped to higher dimensions.

Let's look at the following diagram to understand it better:



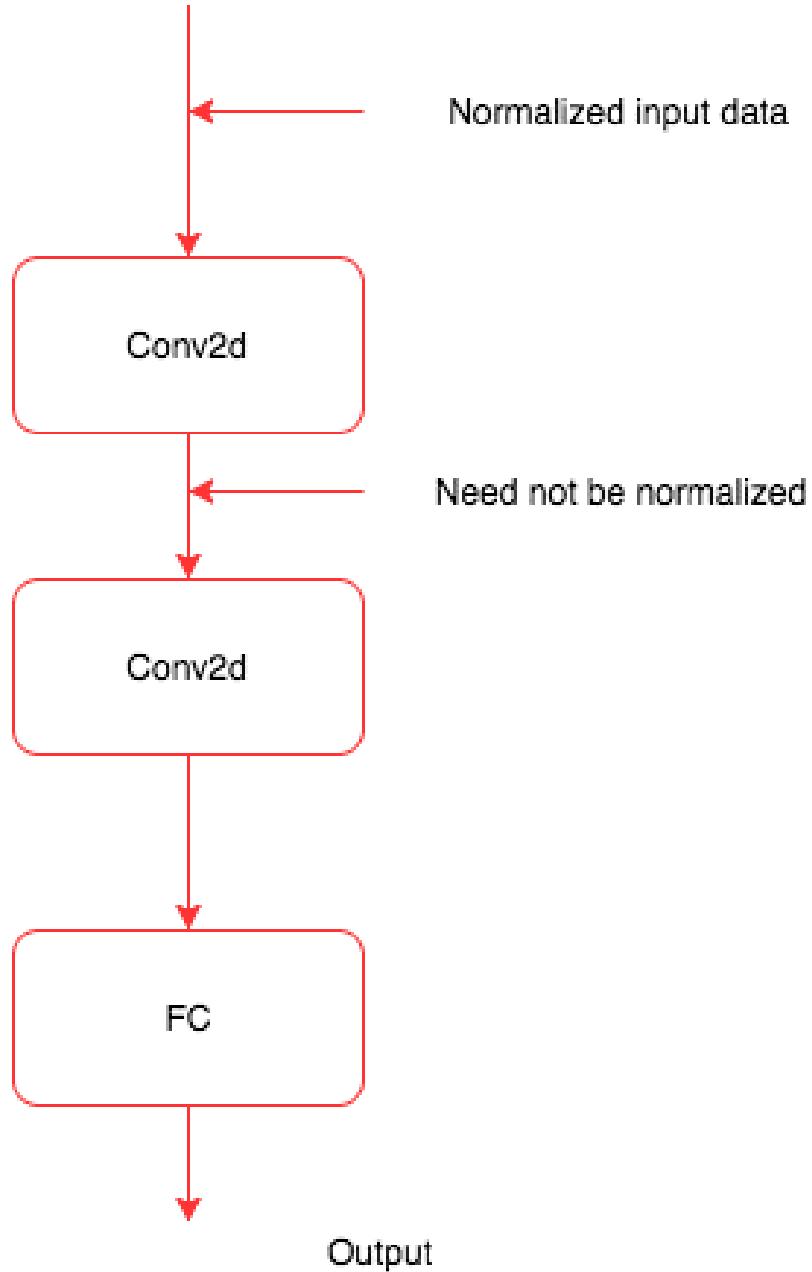
This diagram is referenced in the Theano documentation (another popular deep learning framework—http://deeplearning.net/software/theano/tutorial/conv_arithmetic.html). If you want to explore more about how strided convolutions work, I strongly recommend you read this article from the Theano documentation. What is important for us is that it helps to convert a vector to a tensor of the required dimensions, and we can train the values of the kernels by backpropagation.

Batch normalization

We have already observed a couple of times that all the features that are being passed to either machine learning or deep learning algorithms are normalized; that is, the values of the features are centered to zero by subtracting the mean from the data and giving the data a unit standard deviation by dividing the data by its standard deviation. We would generally do this by using the PyTorch `torchvision.Normalize` method. The following code shows an example:

```
| transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
```

In all the examples we have seen, the data is normalized just before it enters a neural network; there is no guarantee that the intermediate layers get a normalized input. The following diagram shows how the intermediate layers in the neural network fail to get normalized data:

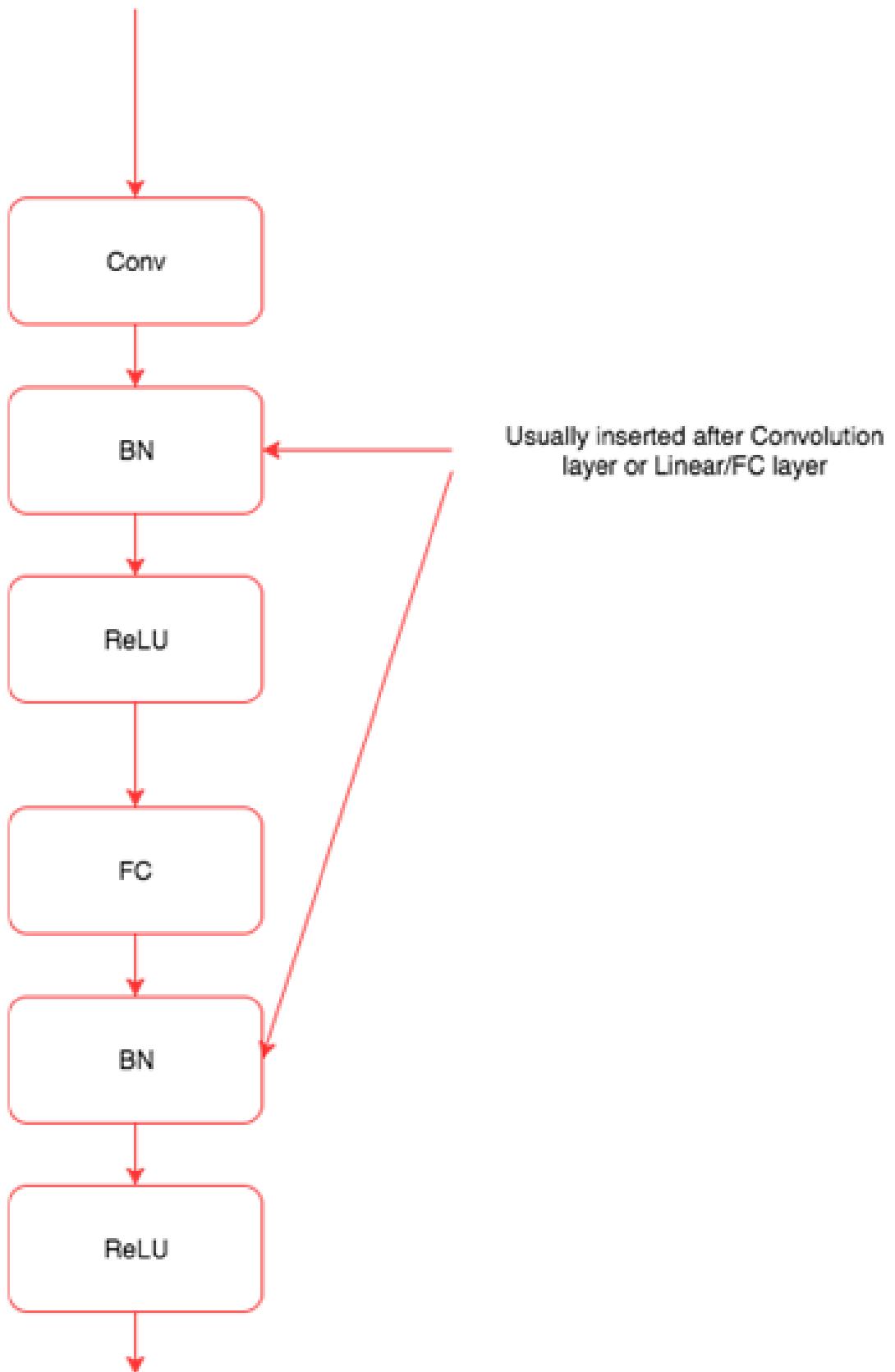


Batch normalization acts like an intermediate function or a layer, which normalizes the intermediate data when the mean and variance change over time during training. Batch normalization was introduced in 2015 by Ioffe and Szegedy (<https://arxiv.org/abs/1502.03167>). Batch normalization behaves differently during training and validation or testing. During training, the mean and variance are calculated for the data in the batch. For validation

and testing, the global values are used. All we need to understand in order to use it is that it normalizes the intermediate data. Some of the key advantages of using batch normalization are that it does the following:

- Improves gradient flow through the network, thus helping us build deeper networks
- Allows higher learning rates
- Reduces the strong dependency of initialization
- Acts as a form of regularization and reduces the dependency of dropout

Most of the modern architectures, such as ResNet and Inception, extensively use batch normalization in their architectures. We will be diving deeper into these architectures in the next chapter. Batch normalization layers are introduced after a convolution layer or linear/fully connected layers, as shown in the following diagram:



By now, we have an intuitive understanding of the key components of a generator network.

Generator

Let's quickly look at the following generator network code, and then discuss the key features of the generator network:

```
class _net_generator(nn.Module):
    def __init__(self):
        super(_net_generator, self).__init__()

        self.main = nn.Sequential(
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            nn.ConvTranspose2d(ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
        )

    def forward(self, input):
        output = self.main(input)
        return output

net_generator = _net_generator()
net_generator.apply(weights_initialisation)
print(net_generator)
```

In most of the code examples we have seen, we use a bunch of different layers and then define the flow in the forward method. In the generator network, we define the layers and the flow of the data inside the `__init__` method using a sequential model. The model takes as input a tensor of size `nz`, and then passes it on to a transposed convolution to map the input to the image size that it needs to generate. The forward function passes on the input to the sequential module and returns the output. The last layer of the generator

network is a tanh layer, which limits the range of values the network can generate.

Instead of using the same random weights, we initialize the model with weights as defined in the paper. The following is the weight initialization code:

```
| def weights_inititalisation(m):
|     class_name = m.__class__.__name__
|     if class_name.find('Conv') != -1:
|         m.weight.data.normal_(0.0, 0.02)
|     elif class_name.find('BatchNorm') != -1:
|         m.weight.data.normal_(1.0, 0.02)
|         m.bias.data.fill_(0)
```

We call the `weight` function by passing the function to the generator object, `net_generator`. Each layer is passed on to the function; if the layer is a convolution layer we initialize the weights differently, and if it is `BatchNorm`, then we initialize it a bit differently. We call the function on the `network` object using the following code:

```
| net_generator.apply(weights_inititalisation)
```

Defining the discriminator network

Let's quickly look at the following discriminator network code, and then discuss the key features of the discriminator network:

```
class _net_discriminator(nn.Module):
    def __init__(self):
        super(_net_discriminator, self).__init__()
        self.main = nn.Sequential(
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        output = self.main(input)
        return output.view(-1, 1).squeeze(1)

net_discriminator = _net_discriminator()
net_discriminator.apply(weights_inititalisation)
print(net_discriminator)
```

There are two important things in the previous network, namely, the usage of leaky ReLU as an activation function, and the usage of sigmoid as the last activation layer. First, let's understand what leaky ReLU is.

Leaky ReLU is an attempt to fix the dying ReLU problem. Instead of the function returning zero when the input is negative, leaky ReLU will output a very small number like 0.001. In the paper, it is shown that using leaky ReLU improves the efficiency of the discriminator.

Another important difference is not using fully connected layers at the end of the discriminator. It is common to see the last fully connected layers being replaced by global average pooling. But using global average pooling reduces the rate of the convergence speed (number of iterations to build an accurate classifier). The last convolution layer is flattened and passed to a sigmoid layer.

Other than these two differences, the rest of the network is similar to the other image classifier networks we have seen in the book.

Defining loss and optimizer

We will define a binary cross-entropy loss and two optimizers, one for the generator and another one for the discriminator, in the following code:

```
criterion = nn.BCELoss()  
  
optimizer_discriminator = optim.Adam(net_discriminator.parameters(), lr,  
betas=(beta1, 0.95))  
optimizer_generator = optim.Adam(net_generator.parameters(), lr, betas=  
(beta1, 0.95))
```

Up to this point, it is very similar to what we have seen in all our previous examples. Let's explore how we can train the generator and discriminator.

Training the discriminator

The loss of the discriminator network depends on how it performs on real images and how it performs on fake images generated by the generator network. The loss can be defined as follows:

$$\text{loss} = \max \log(D(x)) + \log(1 - D(G(z)))$$

So, we need to train the discriminator with real images and the fake images generated by the generator network.

Training the discriminator with real images

Let's pass some real images as direct information to train the discriminator.

First, we will take a look at the code for doing the same and then explore the important features:

```
| output = net_discriminator(inputv)
| err_discriminator_real = criterion(output, labelv)
| err_discriminator_real.backward()
```

In the previous code, we calculate the loss and the gradients required for the discriminator image. The `inputv` and `labelv` values represents the input image from the CIFAR10 dataset and labels, which is one for real images. It is pretty straightforward, as it is similar to what we do for other image classifier networks.

Training the discriminator with fake images

Now pass some random images to train the discriminator.

Let's look at the code for it and then explore the important features:

```
|fake = net_generator(noisev)
|output = net_discriminator(fake.detach())
|err_discriminator_fake = criterion(output, labelv)
|err_discriminator_fake.backward()
|optimizer_discriminator.step()
```

The first line in this code passes a vector with a size of 100, and the generator network (`net_generator`) generates an image. We pass on the image to the discriminator for it to identify whether the image is real or fake. We do not want the generator to get trained, as the discriminator is getting trained. So, we remove the fake image from its graph by calling the `detach` method on its variable. Once all the gradients are calculated, we call the optimizer to train the discriminator.

Training the generator network

Let's look at the following code for training the generator network and then explore the important features:

```
net_generator.zero_grad()
labelv = Variable(label.fill_(real_label)) # fake labels are real for
generator cost
output = net_discriminator(fake)
err_generator = criterion(output, labelv)
err_generator.backward()
optimizer_generator.step()
```

It looks similar to what we did while we trained the discriminator on fake images, except for some key differences. We are passing the same fake images created by the generator, but this time we are not detaching it from the graph that produced it, because we want the generator to be trained. We calculate the loss (`err_generator`) and calculate the gradients. Then we call the generator optimizer, as we only want the generator to be trained, and we repeat this entire process for several iterations before we have the generator producing slightly realistic images.

Training the complete network

We have looked at individual pieces of how a GAN is trained. Let's summarize them as follows and look at the complete code that will be used to train the GAN network we created:

- Train the discriminator network with real images
- Train the discriminator network with fake images
- Optimize the discriminator
- Train the generator based on the discriminator feedback
- Optimize the generator network alone

We will use the following code to train the network:

```
for epoch in range(niter):
    for i, data in enumerate(dataloader, 0):
        # train with real
        net_discriminator.zero_grad()
        real_cpu, _ = data
        batch_size = real_cpu.size(0)
        if torch.cuda.is_available():
            real_cpu = real_cpu.cuda()
        input.resize_as_(real_cpu).copy_(real_cpu)
        label.resize_(batch_size).fill_(real_label)
        inputv = Variable(input)
        labelv = Variable(label)

        output = net_discriminator(inputv)
        err_discriminator_real = criterion(output, labelv)
        err_discriminator_real.backward()
        D_x = output.data.mean()

        noise.resize_(batch_size, nz, 1, 1).normal_(0, 1)
        noisev = Variable(noise)
        fake = net_generator(noisev)
        labelv = Variable(label.fill_(fake_label))
        output = net_discriminator(fake.detach())
        err_discriminator_fake = criterion(output, labelv)
        err_discriminator_fake.backward()
        D_G_z1 = output.data.mean()
        err_discriminator = err_discriminator_real +
err_discriminator_fake
        optimizer_discriminator.step()

        net_generator.zero_grad()
        labelv = Variable(label.fill_(real_label)) # fake labels are real
```

```

for generator_cost
    output = net_discriminator(fake)
    err_generator = criterion(output, labelv)
    err_generator.backward()
    D_G_z2 = output.data.mean()
    optimizer_generator.step()

    print('[%d/%d] [%d/%d] Loss_Discriminator: %.4f Loss_Generator:
%.4f D(x): %.4f D(G(z)): %.4f / %.4f'
          % (epoch, niter, i, len(dataloader),
             err_discriminator.data[0], err_generator.data[0], D_x,
             D_G_z1, D_G_z2))
    if i % 100 == 0:
        vutils.save_image(real_cpu,
                           '%s/real_samples.png' % outf,
                           normalize=True)
        fake = net_generator(fixed_noise)
        vutils.save_image(fake.data,
                           '%s/fake_samples_epoch_%03d.png' % (outf, epoch),
                           normalize=True)

```

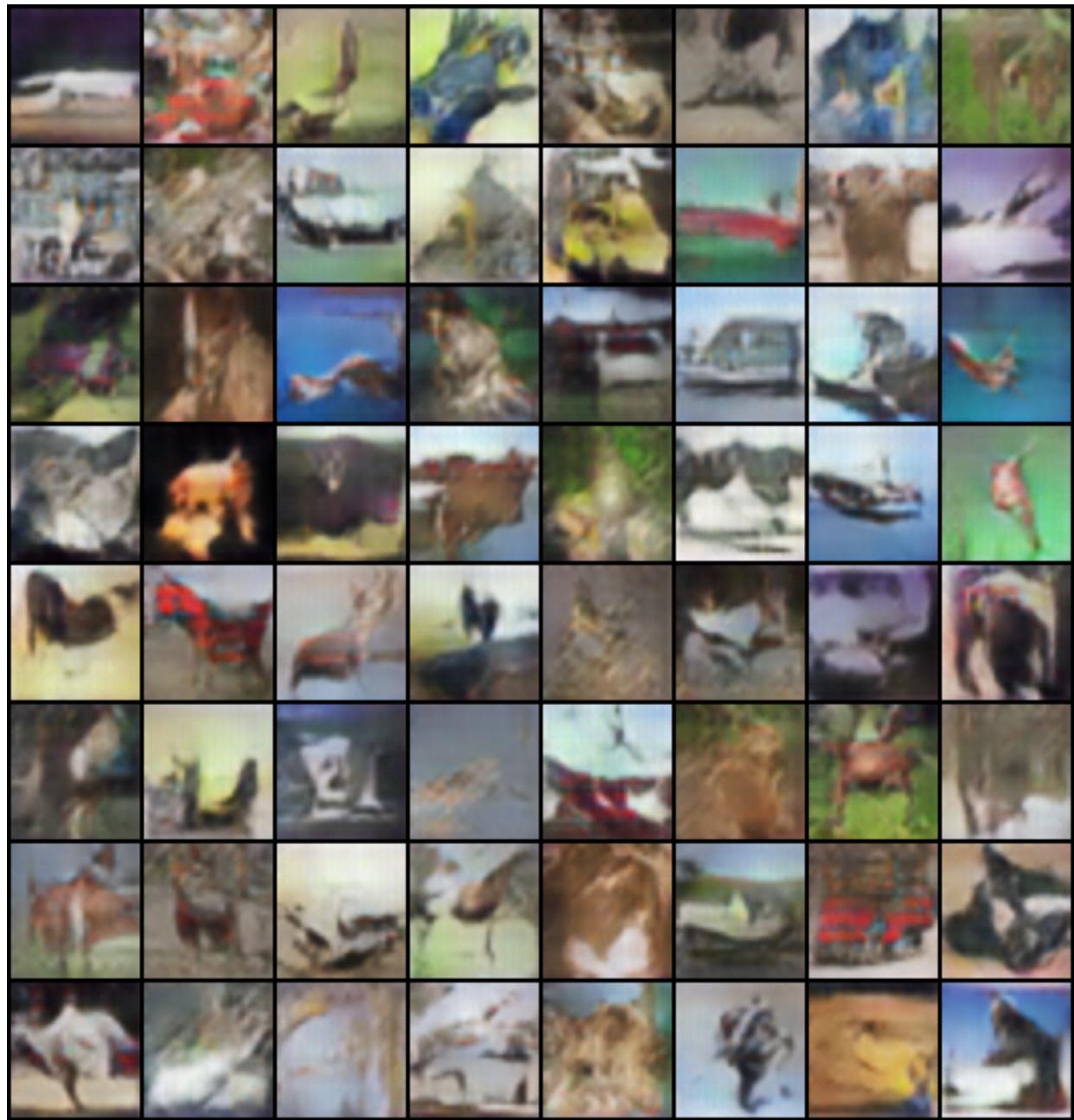
The `vutils.save_image` will take a tensor and save it as an image. If provided with a mini-batch of images, then it saves them as a grid of images.

In the following sections, we will take a look at what the generated images and real images look like.

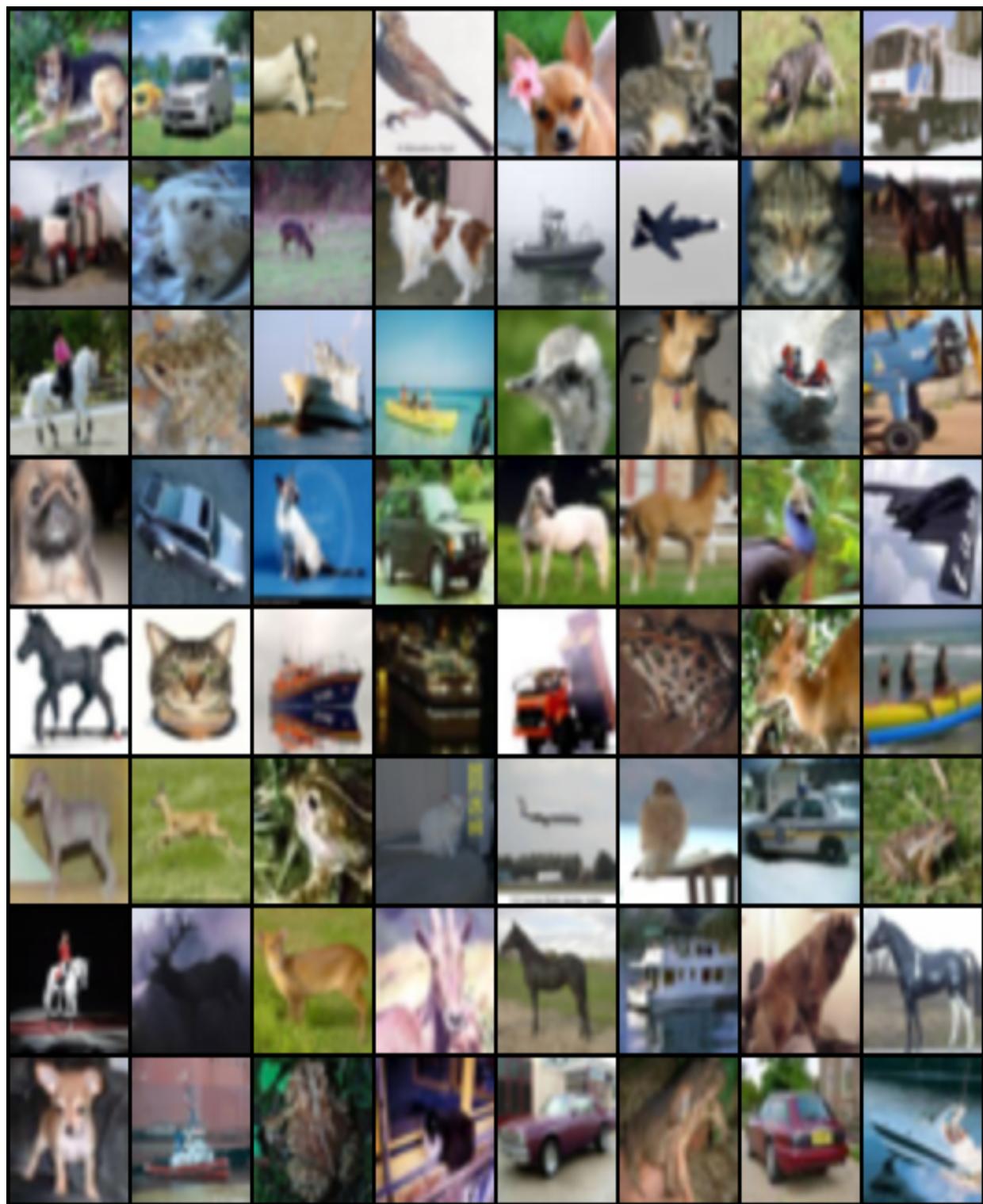
Inspecting the generated images

So, let's compare the generated images and real images.

The generated images will be as follows:



The real images are as follows:



Comparing both sets of images, we can see that our GAN was able to learn how to generate images.

Summary

In this chapter, we covered how to train deep learning algorithms that can generate artistic style transfers using generative networks. We also learned how to generate new images using GAN and DCGAN. In DCGAN, we explored training the discriminator with real and fake images and inspected the generated images. Apart from training to generate new images, we also have a discriminator, which can be used for classification problems. The discriminator learns important features about the images that can be used for classification tasks when there is a limited amount of labeled data available. When there is limited labeled data, we can train a GAN that will give us a classifier, which can be used to extract features—and a classifier module can be built on top of it.

In the next chapter, we will cover some of the modern architectures, such as ResNet and Inception, for building better computer vision models and models such as sequence-to-sequence, which can be used for building language translation and image captioning.

Transfer Learning with Modern Network Architectures

In the previous chapter, we explored how deep learning algorithms can be used to create artistic images, create new images based on existing datasets, and generate text. In this chapter, we will introduce you to different network architectures that power modern computer vision applications and natural language systems. We will also cover how transfer learning can be incorporated into these models.

Transfer learning is a method in machine learning where a model that's been developed for a particular task is reused for another. If, for example, we wanted to learn how to drive a motorbike but we already know how to drive a car, we would transfer our knowledge about what driving a car involves to the new task rather than starting from scratch.

To transfer such knowledge from one task to another, some of the layers in the network need to be frozen. Freezing a layer means that the weights of the layer won't update during training. The benefit of transfer learning is that it can speed up the time that's taken to develop and train a new model by reusing what was learned by the pretrained model, thereby helping to accelerate the results.

Some of the architectures that we will look at in this chapter are as follows:

- **Residual network (ResNet)**
- Inception
- DenseNet
- Encoder-decoder architecture

The following topics will be covered in this chapter:

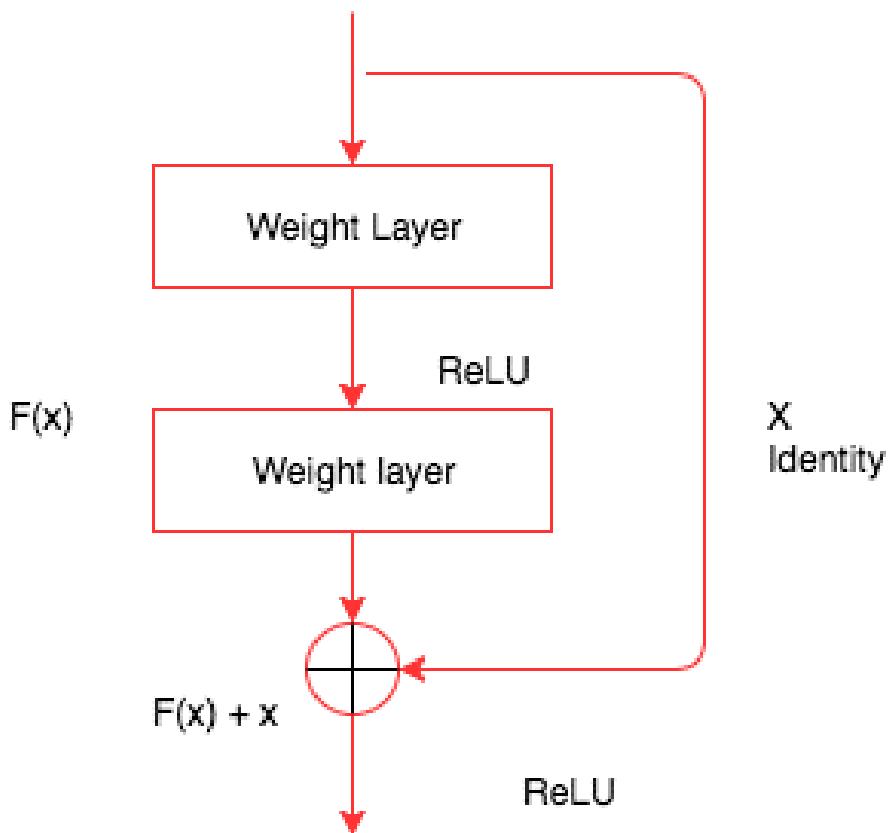
- Modern network architectures
- Densely connected convolutional networks – DenseNet
- Model ensembling
- Encoder-decoder architecture

Modern network architectures

One of the best things we do when the deep learning model fails to learn is add more layers to the model. As you add layers, the model's accuracy improves and then starts saturating. However, adding more layers beyond a certain number will introduce certain challenges, such as vanishing or exploding gradients. This is partially solved by carefully initializing weights and introducing intermediate normalizing layers. Modern architectures, such as ResNet and Inception, try to solve this problem by introducing different techniques, such as residual connections.

ResNet

ResNet was first introduced in 2015 in a paper called *Deep Residual Learning for Image Recognition* by Kaiming He and et al. (<https://arxiv.org/pdf/1512.03385.pdf>). It makes it possible for us to train thousands of layers and achieve high performance. The core concept of ResNet is to introduce an identity shortcut connection that skips one or more of the layers. The following diagram depicts how ResNet works:



This identity mapping doesn't have any parameters. It is simply there to add the output from the previous layer to the next layer. However, sometimes, x and $F(x)$ will not have the same dimensions. A convolution operation typically shrinks the spatial resolution of an image. For example, a 3×3 convolution on a 32×32 image results

in a 30×30 image. This identity mapping is multiplied by a linear projection, W , in order to expand the channels of the shortcut to match the residual. As such, the input, x , and $F(x)$ need to be combined to create the input for the next layer:

$$y = F(x, \{W_i\}) + W_s x.$$

The following code demonstrates what a simple ResNet block would look like in PyTorch:

```
class ResNetBlock(nn.Module):
    def __init__(self,in_channels,output_channels,stride):
        super().__init__()
        self.convolutional_1 =
            nn.Conv2d(input_channels,output_channels,kernel_size=3,stride=stride,padding=1,bias=False)
        self.bn1 = nn.BatchNorm2d(output_channels)
        self.convolutional_2 =
            nn.Conv2d(output_channels,output_channels,kernel_size=3,stride=stride,padding=1,bias=False)
        self.bn2 = nn.BatchNorm2d(output_channels)
        self.stride = stride
    def forward(self,x):
        residual = x
        out = self.convolutional_1(x)
        out = F.relu(self.bn1(out),inplace=True)
        out = self.convolutional_2(out)
        out = self.bn2(out)
        out += residual
        return F.relu(out)
```

The `ResNetBasicBlock` class contains an `__init__` method that initializes all the different layers, such as the convolution layer and batch normalization. The `forward` method is almost identical to what we have seen up until now, except that the input is added to the layer's output just before it is returned.

The PyTorch `torchvision` package provides an out-of-the-box ResNet model with different layers. Some of the different models that are available are as follows:

- ResNet-18
- ResNet-34

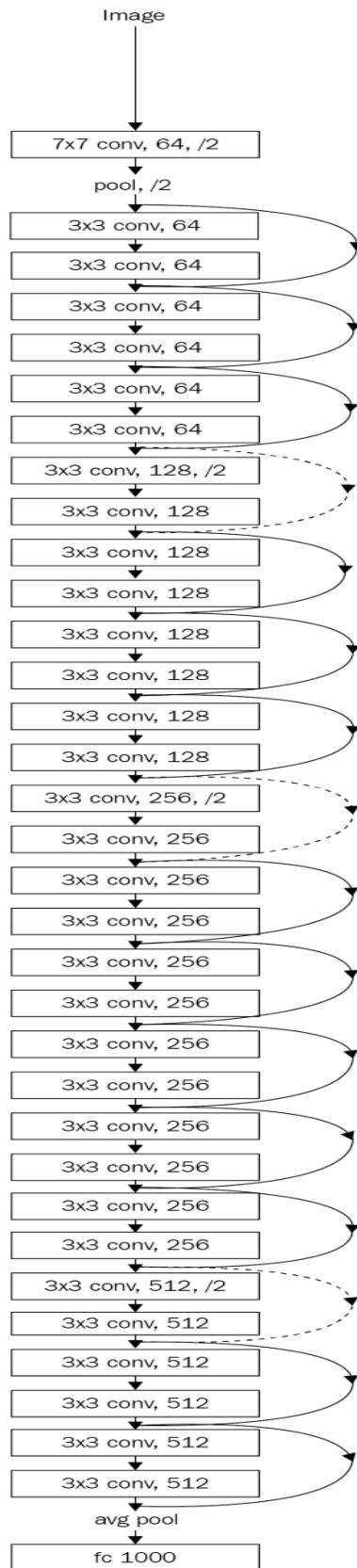
- ResNet-50
- ResNet-101
- ResNet-152

We can also use any of these models for transfer learning. The `torchvision` instance allows us to simply create one of these models and use it as shown in the following code:

```
| from torchvision.models import resnet18  
| resnet_model = resnet18(pretrained=False)
```

The following diagram shows what a 34-layer ResNet model would look like:

34-layer residual



Here, we can see that this network consists of multiple ResNet blocks. A key advantage of these modern networks is that they need very few parameters compared to models such as VGG since they avoid using fully connected layers that need lots of parameters to train.

Now, we will train a ResNet model on the dogs and cats dataset. We will use the data that we used in [Chapter 3](#), *Diving Deep into Neural Networks*, and will quickly train a model based on the features that have been calculated from the ResNet. As usual, we will follow these steps to train the model:

1. Create the PyTorch datasets.
2. Create the loaders for training and validation.
3. Create the ResNet model.
4. Extract the convolutional features.
5. Create a custom PyTorch dataset class for the pre-convoluted features and loader.
6. Create a simple linear model.
7. Train and validate the model.

Once done, we are going to repeat these steps for Inception and DenseNet. Finally, we will explore the ensembling technique, where we combine these powerful models to build a new model.

Creating PyTorch datasets

First, we need to create a transformation object that contains all the basic transformations required and use the `ImageFolder` function to load the images from the data directory that we created in [Chapter 3](#), *Diving Deep into Neural Networks*. In the following code, we create the datasets:

```
transform_data = transforms.Compose([
    transforms.Resize((299,299)),
    transforms.ToTensor(),
    transforms.Normalize([0.30, 0.40, 0.40], [0.20, 0.20, 0.20])
])

train_dataset = ImageFolder('../Chapter03/Dog-Cat-
Classifier/Data/Train_Data/train/', transform=transform_data)
validation_dataset = ImageFolder('../Chapter03/Dog-Cat-
Classifier/Data/Train_Data/valid/', transform=transform_data)
classes=2
```

By now, most of the preceding code will be self-explanatory.

Creating loaders for training and validation

We use PyTorch loaders to load the data provided by the dataset in the form of batches, along with all of its advantages, such as shuffling the data and using multithreads, to speed up the process. The following code demonstrates this:

```
training_data_loader =  
    DataLoader(train_dataset, batch_size=32, shuffle=False, num_workers=4)  
validation_data_loader =  
    DataLoader(validation_dataset, batch_size=32, shuffle=False, num_workers=4)
```

We need to maintain the exact sequence of the data while calculating the pre-convoluted features. When we allow the data to be shuffled, we won't be able to maintain the labels. So, ensure that `shuffle` is `False`; otherwise, the required logic needs to be handled inside the code.

Creating a ResNet model

Here, we will consider a coded example for creating a ResNet model. First, we initiate the pretrained `resnet34` model:

```
| resnet_model = resnet34(pretrained=True)
```

Then, we discard the last linear layer:

```
| m = nn.Sequential(*list(resnet_model.children())[:-1])
```

Once the model has been created, we set the `requires_grad` parameter to `False` so that PyTorch doesn't have to maintain any space for holding gradients:

```
| for p in resnet_model.parameters():
|     p.requires_grad = False
```

Extracting convolutional features

Here, we pass the data from the train and validation data loaders through the model and store the results in a list for further computation. By calculating the pre-convoluted features, we can save a lot of time in training the model as we won't be calculating these features in every iteration:

```
# Stores the labels of the train data
training_data_labels = []
# Stores the pre convoluted features of the train data
training_features = []
```

Iterate through the train data and store the calculated features and the labels using the following code:

```
for d,la in training_data_loader:
    o = m(Variable(d))
    o = o.view(o.size(0),-1)
    training_data_labels.extend(la)
    training_features.extend(o.data)
```

For validation data, iterate through the validation data and store the calculated features and the labels:

```
validation_data_labels = []
validation_features = []
for d,la in validation_data_loader:
    o = m(Variable(d))
    o = o.view(o.size(0),-1)
    validation_data_labels.extend(la)
    validation_features.extend(o.data)
```

Creating a custom PyTorch dataset class for the pre-convoluted features and loader

Now that we have calculated the pre-convoluted features, we need to create a custom dataset that can select data from them. Here, we will create a custom dataset and loader for the pre-convoluted features:

```
class FeaturesDataset(Dataset):
    def __init__(self,features_list,labels_list):
        self.features_list = features_list
        self.labels_list = labels_list
    def __getitem__(self,index):
        return (self.features_lst[index],self.labels_list[index])
    def __len__(self):
        return len(self.labels_list)
#Creating dataset for train and validation
train_features_dataset =
FeaturesDataset(training_features,training_data_labels)
validation_features_dataset =
FeaturesDataset(validation_features,validation_data_labels)
```

Once the custom dataset for the pre-convoluted features has been created, we can use the `DataLoader` function, as follows:

```
train_features_loader =
DataLoader(train_features_dataset,batch_size=64,shuffle=True)
validation_features_loader =
DataLoader(validation_features_dataset,batch_size=64)
```

This will create a data loader for training and validation.

Creating a simple linear model

Now, we need to create a simple linear model that will map the pre-convoluted features to the respective categories. In this example, there are two categories (dogs and cats):

```
class FullyConnectedLinearModel(nn.Module):
    def __init__(self, input_size, output_size):
        super().__init__()
        self.fc = nn.Linear(input_size, output_size)

    def forward(self, inp):
        out = self.fc(inp)
        return out

fully_connected_in_size = 8192

fc = FullyConnectedLinearModel(fully_connected_in_size, classes)
if is_cuda:
    fc = fc.cuda()
```

Now, we are ready to train our new model and validate the dataset.

Training and validating the model

The following code shows how we can train the model. Note that the `fit` function is the same as the one discussed in [Chapter 3](#), *Diving Deep into Neural Networks*:

```
train_losses , train_accuracy = [], []
validation_losses , validation_accuracy = [], []
for epoch in range(1,20):
    epoch_loss, epoch_accuracy =
        fit(epoch,fc,train_features_loader,phase='training')
    validation_epoch_loss , validation_epoch_accuracy =
        fit(epoch,fc,validation_features_loader,phase='validation')
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    validation_losses.append(validation_epoch_loss)
    validation_accuracy.append(validation_epoch_accuracy)
```

Inception

The Inception network was an important milestone in the development of CNN classifiers as it was shown to improve both speed and accuracy. There have been a number of versions of Inception, with some of the most noteworthy being the following:

- Inception v1 (<https://arxiv.org/pdf/1409.4842v1.pdf>), commonly known as GoogLeNet
- Inception v2 and v2 (<https://arxiv.org/pdf/1512.00567v3.pdf>)
- Inception v4 and Inception-ResNet (<https://arxiv.org/pdf/1602.07261.pdf>)

The following diagram shows how the naive Inception network is structured (v1):

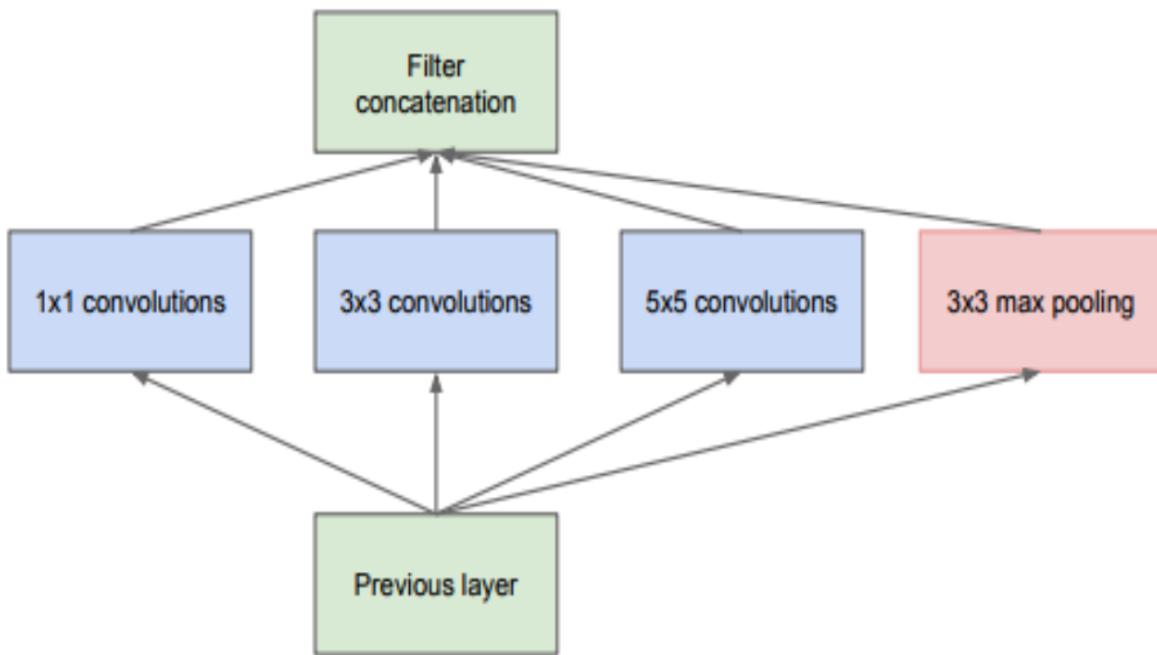


Image source: <https://arxiv.org/pdf/1409.4842.pdf>

Here, the convolution of different sizes is applied to the input, and the outputs of all these layers are concatenated. This is the simplest

version of an Inception module. There is another variant of an Inception block where we pass the input through a 1×1 convolution before passing it through 3×3 and 5×5 convolutions. A 1×1 convolution is used for dimensionality reduction. It helps in solving computational bottlenecks. A 1×1 convolution looks at one value at a time and across the channels. For example, using a $10 \times 1 \times 1$ filter on an input size of $100 \times 64 \times 64$ would result in $10 \times 64 \times 64$. The following diagram shows the Inception block with dimensionality reductions:

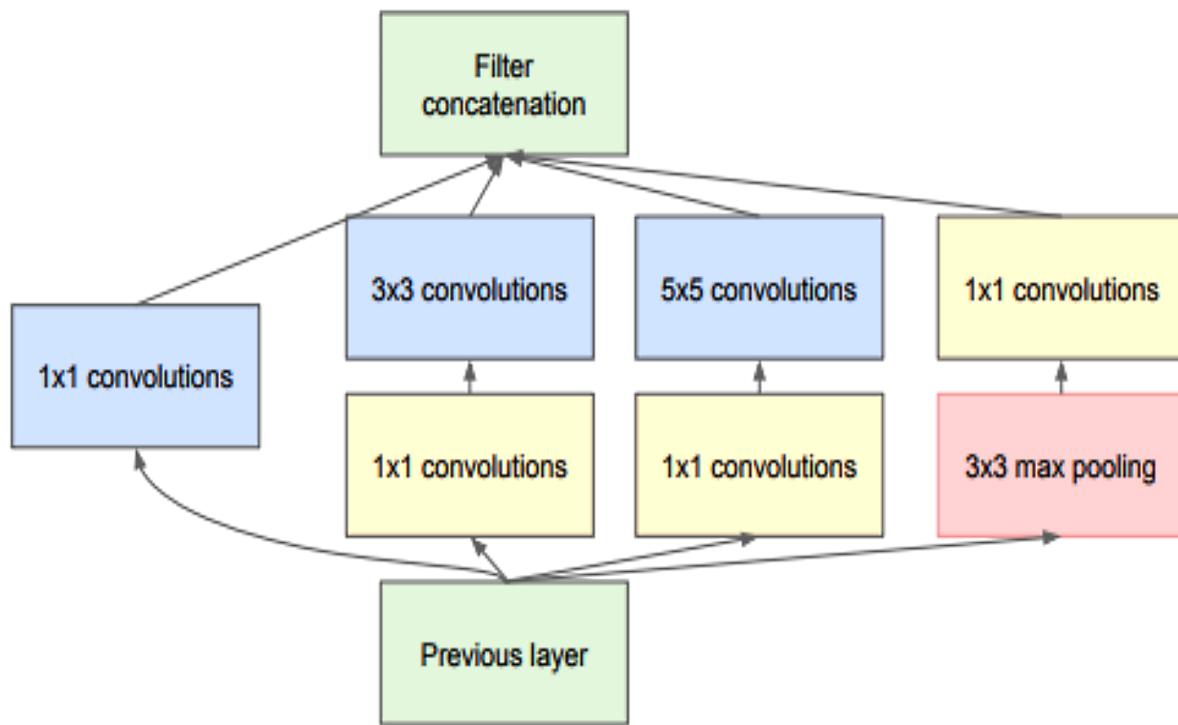


Image source: <https://arxiv.org/pdf/1409.4842.pdf>

Now, let's look at a PyTorch example of what the preceding Inception block would look like:

```

class BasicConvolutional2d(nn.Module):

    def __init__(self, input_channels, output_channels, **kwargs):
        super(BasicConv2d, self).__init__()
        self.conv = nn.Conv2d(input_channels, output_channels, bias=False,
                           **kwargs)
        self.bn = nn.BatchNorm2d(output_channels)

    def forward(self, x):
        x = self.bn(self.conv(x))
        return x
  
```

```

        x = self.conv(x)
        x = self.bn(x)
        return F.relu(x, inplace=True)

class InceptionBlock(nn.Module):

    def __init__(self, input_channels, pool_features):
        super().__init__()
        self.inception_branch_1x1 = BasicConv2d(input_channels, 64,
kernel_size=1)

        self.inception_branch_5x5_1 = BasicConv2d(input_channels, 48,
kernel_size=1)
        self.inception_branch_5x5_2 = BasicConv2d(48, 64, kernel_size=5,
padding=2)

        self.inception_branch_3x3dbl_1 = BasicConv2d(input_channels, 64,
kernel_size=1)
        self.inception_branch_3x3dbl_2 = BasicConv2d(64, 96,
kernel_size=3, padding=1)

        self.inception_branch_pool = BasicConv2d(input_channels,
pool_features, kernel_size=1)

    def forward(self, x):
        inception_branch_1x1 = self.inception_branch1x1(x)

        inception_branch_5x5 = self.inception_branch_5x5_1(x)
        inception_branch_5x5 = self.inception_branch_5x5_2(branch5x5)

        inception_branch_3x3dbl = self.inception_branch_3x3dbl_1(x)
        inception_branch_3x3dbl =
self.inception_branch_3x3dbl_2(inception_branch3x3dbl)

        branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
        branch_pool = self.branch_pool(branch_pool)

        outputs = [inception_branch_1x1, inception_branch_5x5,
inception_branch_3x3dbl, inception_branch_pool]
        return torch.cat(outputs, 1)

```

The preceding code contains two classes: `BasicConv2d` and `InceptionBasicBlock`. `BasicConv2d` acts like a custom layer that applies a two-dimensional convolution layer, batch normalization, and a ReLU layer to the input that is passed through. It is good practice to create a new layer when we have a repeating code structure, in order to make the code look elegant.

The `InceptionBasicBlock` class implements what we have in the second Inception diagram. Let's go through each smaller snippet and try to understand how it is implemented:

```
| inception_branch_1x1 = self.inception_branch_1x1(x)
```

The preceding code transforms the input by applying a 1 x 1 convolution block:

```
| inception_branch_5x5 = self.inception_branch_5x5_1(x)
| inception_branch_5x5 = self.inception_branch_5x5_2(inception_branch5x5)
```

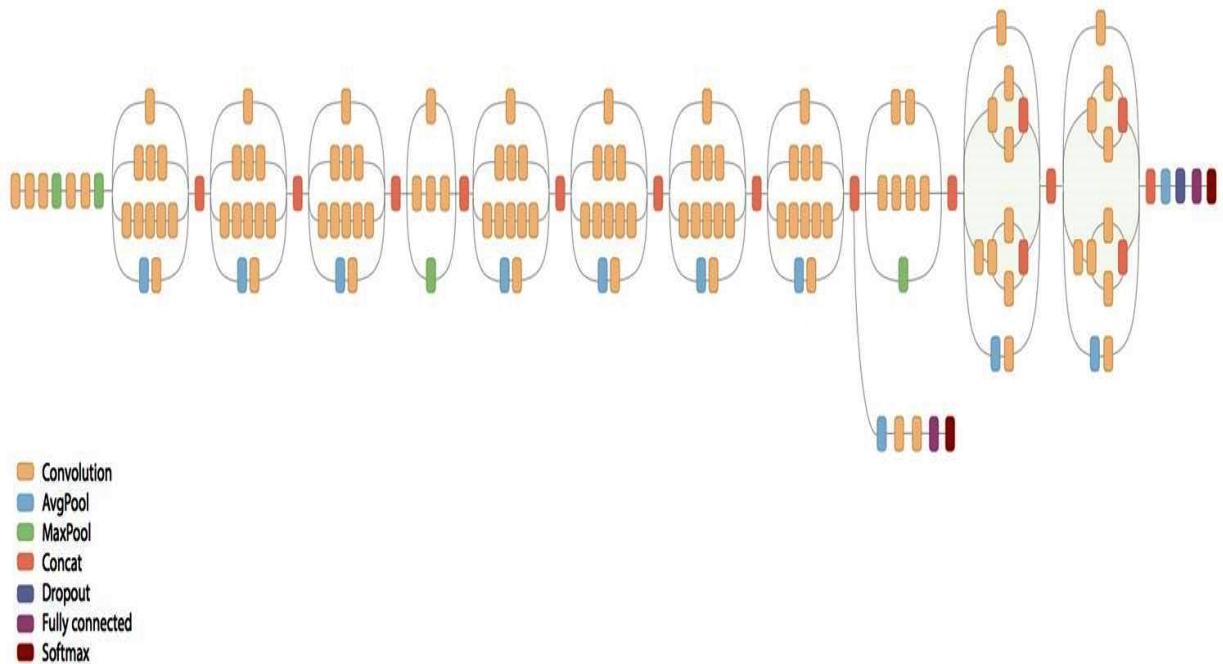
In the preceding code, we transform the input by applying a 1 x 1 convolution block, followed by a 5 x 5 convolution block:

```
| inception_branch_3x3dbl = self.inception_branch_3x3dbl_1(x)
| inception_branch_3x3dbl =
|   self.inception_branch_3x3dbl_2(inception_branch3x3dbl)
```

In the preceding code, we transform the input by applying a 1 x 1 convolution block, followed by a 3 x 3 convolution block:

```
| branch_pool = F.avg_pool2d(x, kernel_size=3, stride=1, padding=1)
| branch_pool = self.branch_pool(branch_pool)
```

In the preceding code, we apply an average pool, along with a 1 x 1 convolution block. Finally, we concatenate all the results together. An Inception network consists of several Inception blocks. The following diagram shows what an Inception architecture would look like:



The Inception architecture

The `torchvision` package has an Inception network that can be used in the same way as we used the ResNet network. Many improvements were made to the initial Inception block, and the current implementation that's available from PyTorch is Inception v3. Let's look at how we can use the Inception v3 model from `torchvision` to calculate pre-computed features. We won't go through the data loading process as we will be using the same data loaders from the *Creating a ResNet model* section. We will look at the following important topics:

- Creating an Inception model
- Extracting convolutional features using `register_forward_hook`
- Creating a new dataset for the convoluted features
- Creating a fully connected model
- Training and validating the model

Creating an Inception model

The Inception v3 model has two branches, each of which generates an output, and in the original model training, we would merge the losses, just like we did for style transfer. At the moment, we are interested in using only one branch to calculate pre-convoluted features using Inception. Going into the details of this is outside the scope of this book. If you are interested in finding out more about how this works, then going through the paper and the source code ([h
`https://github.com/pytorch/vision/blob/master/torchvision/models/inception.py`](https://github.com/pytorch/vision/blob/master/torchvision/models/inception.py)) of the Inception model would help. We can disable one of the branches by setting the `aux_logits` parameter to `False`. The following code explains how to create a model and how to set the `aux_logits` parameter to `False`:

```
| inception_model = inception_v3(pretrained=True)
| inception_model.aux_logits = False
| if is_cuda:
|     inception_model = inception_model.cuda()
```

Extracting the convolution features from the Inception model isn't straightforward, so we will use the `register_forward_hook` function to extract the activations.

Extracting convolutional features using register_forward_hook

We will be using the same techniques that we used to calculate activations for style transfer. The following is the `LayerActivations` class with some minor modifications since we are only interested in extracting the outputs of a particular layer:

```
class LayerActivations():
    features=[]

    def __init__(self,model):
        self.features = []
        self.hook = model.register_forward_hook(self.hook_function)

    def hook_function(self,module,input,output):
        self.features.extend(output.view(output.size(0),-1).cpu().data)

    def remove(self):
        self.hook.remove()
```

Apart from the `hook` function, the rest of the code is similar to what we used for style transfer. Since we are capturing the outputs of all the images and storing them, we won't be able to hold the data on the **graphics processing unit (GPU)** memory. Therefore, we need to extract the tensors from the GPU and CPU and just store the tensors instead of `Variable`. We are converting them back into tensors since the data loaders will only work with tensors. In the following code, we're using the objects of `LayerActivations` to extract the output of the Inception model from the last layer, excluding the average pooling layer, the dropout layer, and the linear layer. We are skipping the average pooling layer to avoid losing useful information in the data:

```
# Create LayerActivations object to store the output of inception model  
at a particular layer.  
train_features = LayerActivations(inception_model.Mixed_7c)  
train_labels = []  
  
# Passing all the data through the model , as a side effect the outputs  
will get stored  
# in the features list of the LayerActivations object.  
for da,la in train_loader:  
    _ = inception_model(Variable(da.cuda()))  
    train_labels.extend(la)  
train_features.remove()  
  
# Repeat the same process for validation dataset .  
  
validation_features = LayerActivations(inception_model.Mixed_7c)  
validation_labels = []  
for da,la in validation_loader:  
    _ = inception_model(Variable(da.cuda()))  
    validation_labels.extend(la)  
validation_features.remove()
```

Let's create the datasets and loaders that are required for the new convoluted features.

Creating a new dataset for the convoluted features

We can use the same `FeaturesDataset` class to create the new dataset and data loaders. In the following code, we're creating the datasets and the loaders:

```
#Dataset for pre computed features for train and validation data sets

train_feat_dset = FeaturesDataset(train_features.features,train_labels)
validation_feat_dset =
FeaturesDataset(validation_features.features,validation_labels)

#Data loaders for pre computed features for train and validation data
sets

train_feat_loader =
DataLoader(train_feat_dset,batch_size=64,shuffle=True)
validation_feat_loader = DataLoader(validation_feat_dset,batch_size=64)
```

Let's create a new model that we can train on the pre-convoluted features.

Creating a fully connected model

A simple model may end up overfitting, so let's include dropout in the model. Dropout will help us avoid overfitting. In the following code, we are creating our model:

```
class FullyConnectedModel(nn.Module):

    def __init__(self, input_size, output_size, training=True):
        super().__init__()
        self.fully_connected = nn.Linear(input_size, output_size)

    def forward(self, input):
        output = F.dropout(input, training=self.training)
        output = self.fully_connected(output)
        return output

# The size of the output from the selected convolution feature
fc_in_size = 131072

fc = FullyConnectedModel(fc_in_size, classes)
if is_cuda:
    fc = fc.cuda()
```

Once the model has been created, we can train the model.

Training and validating the model

Here, we will use the same fit and training logic that we used in our ResNet example. We're only going to look at the training code and the results it returns:

```
for epoch in range(1,10):
    epoch_loss, epoch_accuracy =
    fit(epoch,fc,train_feat_loader,phase='training')
    validation_epoch_loss , validation_epoch_accuracy =
    fit(epoch,fc,validation_feat_loader,phase='validation')
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    validation_losses.append(validation_epoch_loss)
    validation_accuracy.append(validation_epoch_accuracy)
```

This will result in the following output:

```
training loss is 0.78 and training accuracy is 22825/23000 99.24
validation loss is 5.3 and validation accuracy is 1947/2000 97.35
training loss is 0.84 and training accuracy is 22829/23000 99.26
validation loss is 5.1 and validation accuracy is 1952/2000 97.6
training loss is 0.69 and training accuracy is 22843/23000 99.32
validation loss is 5.1 and validation accuracy is 1951/2000 97.55
training loss is 0.58 and training accuracy is 22852/23000 99.36
validation loss is 4.9 and validation accuracy is 1953/2000 97.65
training loss is 0.67 and training accuracy is 22862/23000 99.4
validation loss is 4.9 and validation accuracy is 1955/2000 97.75
training loss is 0.54 and training accuracy is 22870/23000 99.43
validation loss is 4.8 and validation accuracy is 1953/2000 97.65
training loss is 0.56 and training accuracy is 22856/23000 99.37
validation loss is 4.8 and validation accuracy is 1955/2000 97.75
training loss is 0.7 and training accuracy is 22841/23000 99.31
validation loss is 4.8 and validation accuracy is 1956/2000 97.8
training loss is 0.47 and training accuracy is 22880/23000 99.48
validation loss is 4.7 and validation accuracy is 1956/2000 97.8
```

Looking at the results, the Inception model achieves 99% accuracy on the training dataset and 97.8% accuracy on the validation dataset. Since we are precomputing and holding all the features in memory, it takes less than a few minutes to train the models. If you are running out of memory when you run the program on your

machine, then you may need to avoid holding the features in memory.

In the next section, we will look at another interesting architecture, DenseNet, which has become very popular in the last year.

Densely connected convolutional networks – DenseNet

Some of the most successful and popular architectures, such as ResNet and Inception, have shown the importance of deeper and wider networks. ResNet uses shortcut connections to build deeper networks. DenseNet has taken this to a whole new level by allowing connections to be made from each layer to the subsequent layers, that is, the layers where we can receive all the feature maps from the previous layers. Symbolically, this would look as follows:

$$X_l = H_l(x_0, x_1, x_2, \dots, x_{l-1})$$

The following diagram describes what a five-layer dense block would look like:

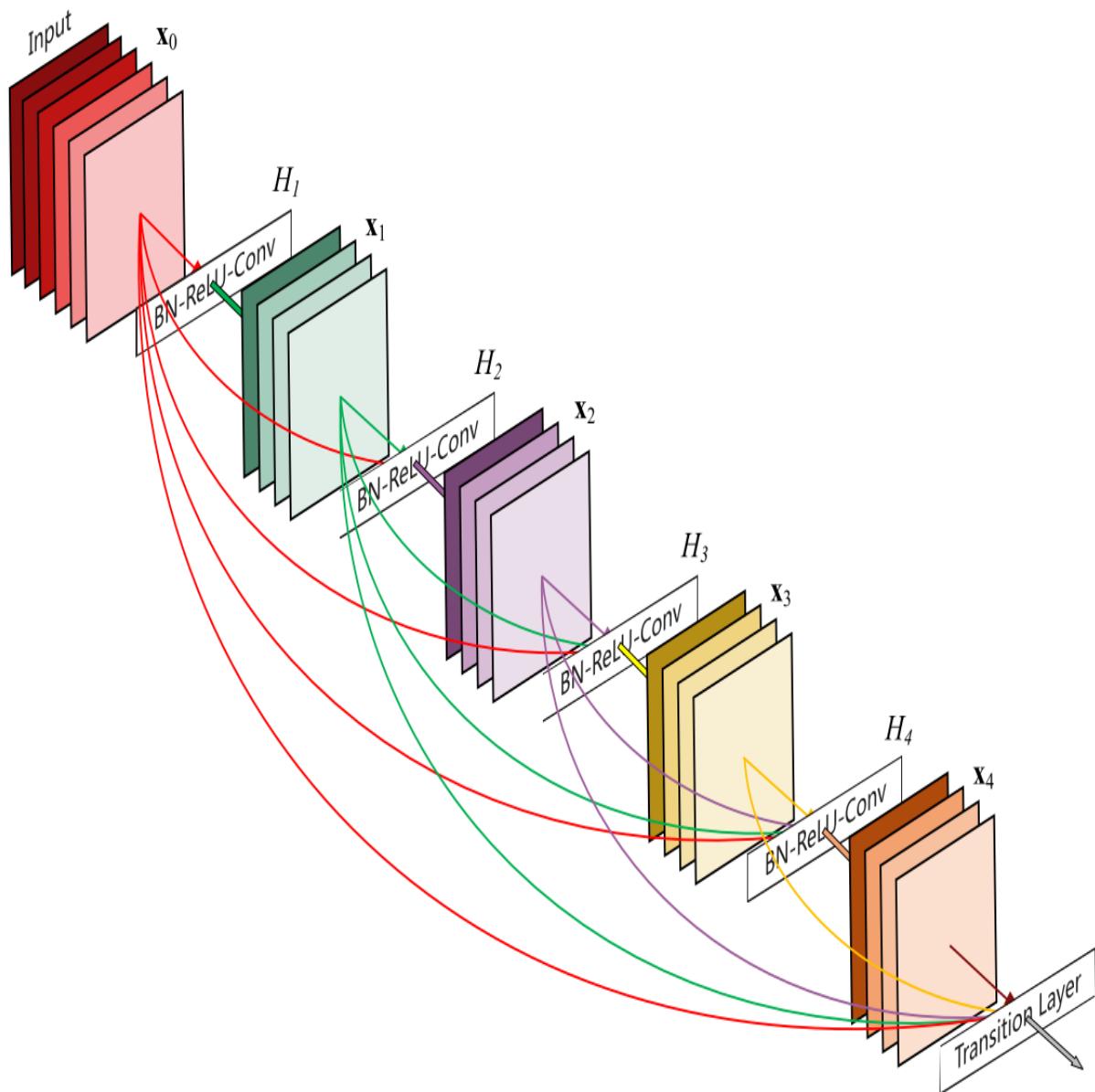


Image source: <https://arxiv.org/abs/1608.06993>

There is also a DenseNet implementation of `torchvision` (<https://github.com/pytorch/vision/blob/master/torchvision/models/densenet.py>). Let's look at two of its major functionalities, that is, `_DenseBlock` and `_DenseLayer`.

The `_DenseBlock` object

Let's look at the code for `_DenseBlock` and then walk through it:

```
class _DenseBlock(nn.Sequential):
    def __init__(self, number_layers, number_input_features, bn_size,
growth_rate, drop_rate):
        super(_DenseBlock, self).__init__()
        for i in range(number_layers):
            layer = _DenseLayer(number_input_features + i * growth_rate,
growth_rate, bn_size, drop_rate)
            self.add_module('denselayer%d' % (i + 1), layer)
```

`_DenseBlock` is a sequential module where we add layers in a sequential order. Based on the number of layers (`number_layers`) in the block, we add that number of `_DenseLayer` objects, along with a name, to it. All the magic happens inside the `_DenseLayer` object. Let's look at what goes on inside the `DenseLayer` object.

The `_DenseLayer` object

One way to learn how a particular network works is to look at the source code. PyTorch has a very clean implementation and most of the time it is easily readable. Let's look at the `_DenseLayer` implementation:

```
class _DenseLayer(nn.Sequential):
    def __init__(self, number_input_features, growth_rate, bn_size,
drop_rate):
        super(_DenseLayer, self).__init__()
        self.add_module('norm.1', nn.BatchNorm2d(number_input_features)),
        self.add_module('relu.1', nn.ReLU(inplace=True)),
        self.add_module('conv.1', nn.Conv2d(number_input_features, bn_size
*)
                                growth_rate, kernel_size=1, stride=1,
bias=False)),
        self.add_module('norm.2', nn.BatchNorm2d(bn_size * growth_rate)),
        self.add_module('relu.2', nn.ReLU(inplace=True)),
        self.add_module('conv.2', nn.Conv2d(bn_size * growth_rate,
growth_rate,
                                kernel_size=3, stride=1, padding=1, bias=False)),
        self.drop_rate = drop_rate

    def forward(self, x):
        new_features = super(_DenseLayer, self).forward(x)
        if self.drop_rate > 0:
            new_features = F.dropout(new_features, p=self.drop_rate,
training=self.training)
        return torch.cat([x, new_features], 1)
```

If you are new to inheritance in Python, then the preceding code may not look intuitive. The `_DenseLayer` object is a subclass of `nn.Sequential`; let's look at what goes on inside each method.

In the `__init__` method, we add all the layers that the input data needs to be passed to. It is quite similar to all the other network architectures we have seen.

The magic happens in the `forward` method. We pass the input to the `forward` method of the super class, which is `nn.Sequential`. Let's look at what happens in the `forward` method of the sequential class (<https://github.com>

[ub.com/pytorch/pytorch/blob/409b1c8319ecde4bd62fcf98d0a6658ae7a4ab23/torch/nn/module.py](https://github.com/pytorch/pytorch/blob/409b1c8319ecde4bd62fcf98d0a6658ae7a4ab23/torch/nn/module.py)):

```
| def forward(self, input):
|     for module in self._modules.values():
|         input = module(input)
|     return input
```

The input is passed through all the layers that were previously added to the sequential block and the output is concatenated to the input. This process is repeated for the required number of layers in a block.

Now that we understand how a DenseNet block works, let's explore how we can use DenseNet to calculate pre-convoluted features and build a classifier model on top of it. At a high level, the DenseNet implementation is similar to the VGG implementation. The DenseNet implementation also has a features module, which contains all the dense blocks, and a classifier module, which contains the fully connected model. We will be going through the following steps for building the model in this section but will be skipping most of the parts that are similar to what we have seen for Inception and ResNet, such as creating the data loader and datasets.

We will discuss the following steps in detail:

- Creating a DenseNet model
- Extracting DenseNet features
- Creating a dataset and loaders
- Creating a fully connected model and training it

By now, most of the code will be self-explanatory.

Creating a DenseNet model

Torchvision has a pretrained DenseNet model with different layer options (121, 169, 201, and 161). Here, we have chosen the model with 121 layers. As we mentioned previously, the DenseNet model has two modules: `features` (containing the dense blocks) and `classifier` (fully connected block). Since we are using DenseNet as an image feature extractor, we will only use the `features` module:

```
densenet_model = densenet121(pretrained=True).features
if is_cuda:
    densenet_model = densenet_model.cuda()

for p in densenet_model.parameters():
    p.requires_grad = False
```

Let's extract the DenseNet features from the images.

Extracting DenseNet features

This process is similar to what we did for Inception, except we aren't using `register_forward_hook` to extract features. The following code shows how the DenseNet features are extracted:

```
#For training data
train_labels = []
train_features = []

#code to store densenet features for train dataset.
for d,la in train_loader:
    o = densenet_model(Variable(d.cuda()))
    o = o.view(o.size(0),-1)
    train_labels.extend(la)
    train_features.extend(o.cpu().data)

#For validation data
validation_labels = []
validation_features = []

#Code to store densenet features for validation dataset.
for d,la in validation_loader:
    o = densenet_model(Variable(d.cuda()))
    o = o.view(o.size(0),-1)
    validation_labels.extend(la)
    validation_features.extend(o.cpu().data)
```

The preceding code is similar to what we have seen for Inception and ResNet.

Creating a dataset and loaders

We will use the same `FeaturesDataset` class that we created for ResNet and use it to create data loaders for the train and validation datasets. We will use the following code to do so:

```
# Create dataset for train and validation convolution features
train_feat_dset = FeaturesDataset(train_features,train_labels)
validation_feat_dset =
FeaturesDataset(validation_features,validation_labels)

# Create data loaders for batching the train and validation datasets
train_feat_loader =
DataLoader(train_feat_dset,batch_size=64,shuffle=True,drop_last=True)
validation_feat_loader = DataLoader(validation_feat_dset,batch_size=64)
```

Now, it's time to create the model and train it.

Creating a fully connected model and training it

Now, we will use a simple linear model, similar to what we used in ResNet and Inception. The following code shows the network architecture that we will be using to train the model:

```
class FullyConnectedModel(nn.Module):

    def __init__(self, input_size, output_size):
        super().__init__()
        self.fc = nn.Linear(input_size, output_size)

    def forward(self, input):
        output = self.fc(input)
        return output

fc = FullyConnectedModel(fc_in_size, classes)
if is_cuda:
    fc = fc.cuda()
```

We will use the same `fit` method to train the preceding model. The following code snippet shows the training code, along with the results:

```
train_losses, train_accuracy = [], []
validation_losses, validation_accuracy = [], []
for epoch in range(1,10):
    epoch_loss, epoch_accuracy =
    fit(epoch,fc,train_feat_loader,phase='training')
    validation_epoch_loss, validation_epoch_accuracy =
    fit(epoch,fc,validation_feat_loader,phase='validation')
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    validation_losses.append(validation_epoch_loss)
    validation_accuracy.append(validation_epoch_accuracy)
```

The result of the preceding code is as follows:

```
training loss is 0.057 and training accuracy is 22506/23000 97.85
validation loss is 0.034 and validation accuracy is 1978/2000 98.9
training loss is 0.0059 and training accuracy is 22953/23000 99.8
validation loss is 0.028 and validation accuracy is 1981/2000 99.05
training loss is 0.0016 and training accuracy is 22974/23000 99.89
```

```
validation loss is 0.022 and validation accuracy is 1983/2000 99.15
training loss is 0.00064 and training accuracy is 22976/23000 99.9
validation loss is 0.023 and validation accuracy is 1983/2000 99.15
training loss is 0.00043 and training accuracy is 22976/23000 99.9
validation loss is 0.024 and validation accuracy is 1983/2000 99.15
training loss is 0.00033 and training accuracy is 22976/23000 99.9
validation loss is 0.024 and validation accuracy is 1984/2000 99.2
training loss is 0.00025 and training accuracy is 22976/23000 99.9
validation loss is 0.024 and validation accuracy is 1984/2000 99.2
training loss is 0.0002 and training accuracy is 22976/23000 99.9
validation loss is 0.025 and validation accuracy is 1985/2000 99.25
training loss is 0.00016 and training accuracy is 22976/23000 99.9
validation loss is 0.024 and validation accuracy is 1986/2000 99.3
```

The preceding algorithm was able to achieve a maximum training accuracy of 99%, and 99% validation accuracy. Your results may be different since the validation dataset you will have created may have different images.

Some of the advantages of DenseNet are as follows:

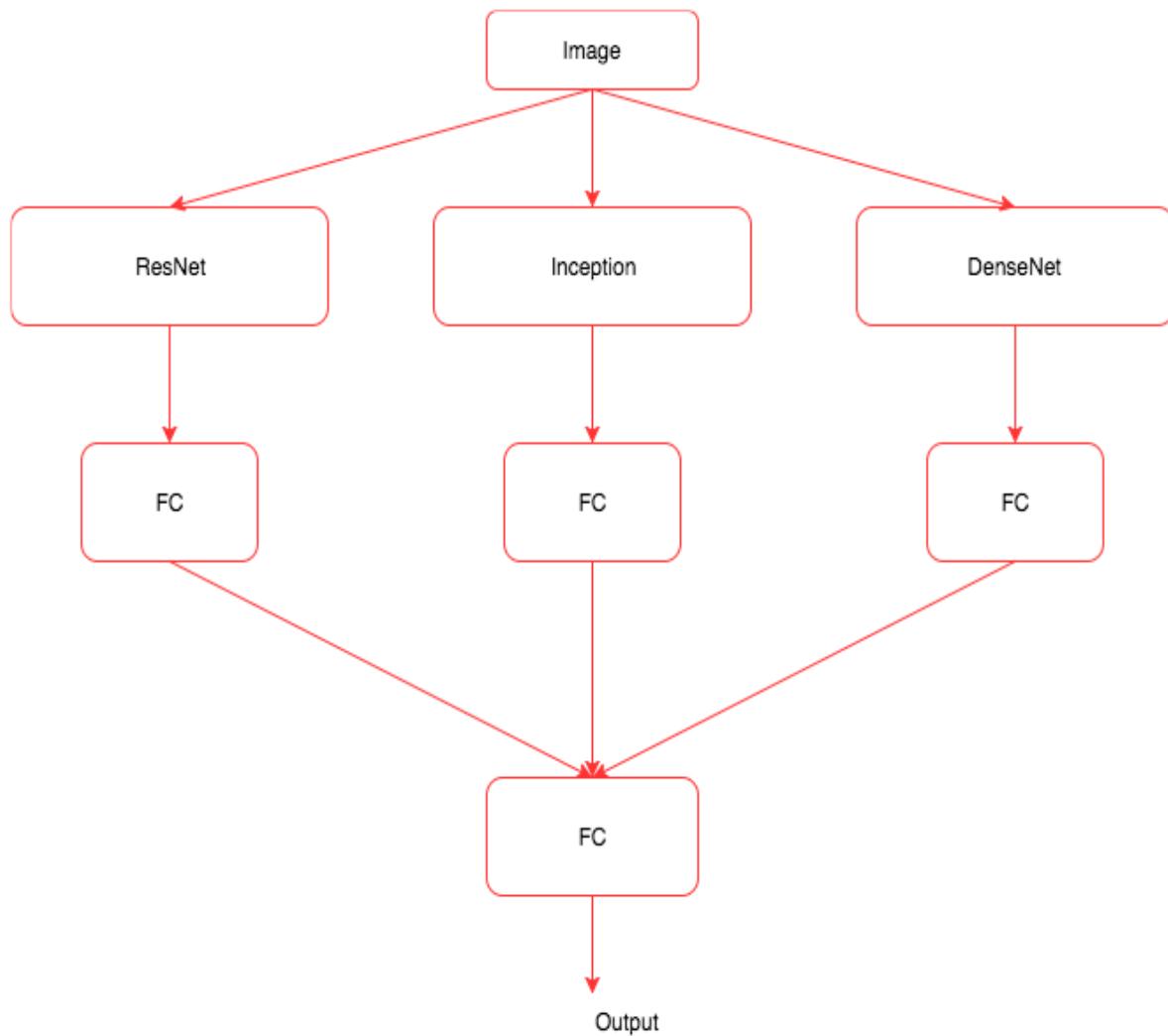
- It substantially reduces the number of parameters required.
- It alleviates the vanishing gradient problem.
- It encourages feature reuse.

In this next section, we will explore how we can build a model that combines the advantages of the convoluted features we've computed using ResNet, Inception, and DenseNet.

Model ensembling

There could be times when we need to try to combine multiple models to build a very powerful model. There are many techniques we can use to build an ensemble model. In this section, we will learn how to combine outputs using the features generated by three different models (ResNet, Inception, and DenseNet) to build a powerful model. We will be using the same dataset that we used for the other examples in this chapter.

The architecture for the ensemble model is as follows:



The preceding diagram shows what we are going to do in the ensemble model, which can be summarized in the following steps:

1. Create three models.
2. Extract the image features using the created models.
3. Create a custom dataset that returns features of all the three models, along with the labels.
4. Create a model that's similar to the architecture shown in the preceding diagram.
5. Train and validate the model.

Let's explore each of these steps in detail.

Creating models

Let's create all the three required models, as shown in the following code blocks.

The code for creating the ResNet model is as follows:

```
resnet_model = resnet34(pretrained=True)

if is_cuda:
    resnet_model = resnet_model.cuda()

resnet_model = nn.Sequential(*list(resnet_model.children())[:-1])

for p in resnet_model.parameters():
    p.requires_grad = False
```

The code for creating the Inception model is as follows:

```
inception_model = inception_v3(pretrained=True)
inception_model.aux_logits = False
if is_cuda:
    inception_model = inception_model.cuda()
for p in inception_model.parameters():
    p.requires_grad = False
```

The code for creating the DenseNet model is as follows:

```
densenet_model = densenet121(pretrained=True).features
if is_cuda:
    densenet_model = densenet_model.cuda()

for p in densenet_model.parameters():
    p.requires_grad = False
```

Now that we have all the models, let's extract the features from the images.

Extracting the image features

Here, we will combine all the logic that we have seen individually for the algorithms in this chapter.

The code for ResNet is as follows:

```
train_labels = []
train_resnet_features = []
for d,la in train_loader:
    o = resnet_model(Variable(d.cuda()))
    o = o.view(o.size(0),-1)
    train_labels.extend(la)
    train_resnet_features.extend(o.cpu().data)
validation_labels = []
validation_resnet_features = []
for d,la in validation_loader:
    o = resnet_model(Variable(d.cuda()))
    o = o.view(o.size(0),-1)
    validation_labels.extend(la)
    validation_resnet_features.extend(o.cpu().data)
```

The code for Inception is as follows:

```
train_inception_features = LayerActivations(inception_model.Mixed_7c)
for da,la in train_loader:
    _ = inception_model(Variable(da.cuda()))

train_inception_features.remove()

validation_inception_features =
LayerActivations(inception_model.Mixed_7c)
for da,la in validation_loader:
    _ = inception_model(Variable(da.cuda()))

validation_inception_features.remove()
```

The code for DenseNet is as follows:

```
train_densenet_features = []
for d,la in train_loader:
    o = densenet_model(Variable(d.cuda()))
    o = o.view(o.size(0),-1)

    train_densenet_features.extend(o.cpu().data)
```

```
validation_densenet_features = []
for d,la in validation_loader:
    o = densnet_model(Variable(d.cuda()))
    o = o.view(o.size(0),-1)
    validation_densenet_features.extend(o.cpu().data)
```

Now, we have created image features using all the models. If you are facing memory issues, then you can either remove one of the models or stop storing features that are slow to train in memory. If you are running this on a CUDA instance, then you can go for a more powerful instance.

Creating a custom dataset, along with data loaders

We won't be able to use the `FeaturesDataset` class as it is since it was developed to pick from the output of only one model. Due to this, the following implementation contains minor changes that have been made to the `FeaturesDataset` class so that we can accommodate all three generated features:

```
class FeaturesDataset(Dataset):
    def __init__(self,featlst1,featlst2,featlst3,labellst):
        self.featlst1 = featlst1
        self.featlst2 = featlst2
        self.featlst3 = featlst3
        self.labellst = labellst

    def __getitem__(self,index):
        return
        (self.featlst1[index],self.featlst2[index],self.featlst3[index],self.labe
        llst[index])

    def __len__(self):
        return len(self.labellst)

train_feat_dset =
FeaturesDataset(train_resnet_features,train_inception_features.features,t
rain_densenet_features,train_labels)
validation_feat_dset =
FeaturesDataset(validation_resnet_features,validation_inception_features.
features,validation_densenet_features,validation_labels)
```

Here, we have made changes to the `__init__` method so that we can store all the features that are generated from the different models. We also changed the `__getitem__` method so that we can retrieve the features and labels of an image. Using the `FeatureDataset` class, we created dataset instances for the training and validation data. Once the dataset has been created, we can use the same data loader for batching data, as shown in the following code:

```
train_feat_loader =
DataLoader(train_feat_dset,batch_size=64,shuffle=True)
validation_feat_loader = DataLoader(validation_feat_dset,batch_size=64)
```

Creating an ensembling model

Now, we need to create a model that works like the architecture diagram we saw previously. The following code implements this:

```
class EnsembleModel(nn.Module):

    def __init__(self, output_size, training=True):
        super().__init__()
        self.fully_connected1 = nn.Linear(8192, 512)
        self.fully_connected2 = nn.Linear(131072, 512)
        self.fully_connected3 = nn.Linear(82944, 512)
        self.fully_connected4 = nn.Linear(512, output_size)

    def forward(self, input1, input2, input3):
        output1 =
        self.fully_connected1(F.dropout(input1, training=self.training))
        output2 =
        self.fully_connected2(F.dropout(input2, training=self.training))
        output3 =
        self.fully_connected3(F.dropout(input3, training=self.training))
        output = output1 + output2 + output3
        output =
        self.fully_connected4(F.dropout(out, training=self.training))
        return output

em = EnsembleModel(2)
if is_cuda:
    em = em.cuda()
```

In the preceding code, we created three linear layers that take the features that will be generated by the different models. We sum up all the outputs from these three linear layers and pass them to another linear layer, which maps them to the required categories. To prevent the model from overfitting, we used dropouts.

Training and validating the model

We need to make some minor changes to the `fit` method to accommodate the three input values we generated from the data loader. The following code implements the new `fit` function:

```
def fit(epoch,model,data_loader,phase='training',volatile=False):
    if phase == 'training':
        model.train()
    if phase == 'validation':
        model.eval()
        volatile=True
    running_loss = 0.0
    running_correct = 0
    for batch_idx , (data1,data2,data3,target) in enumerate(data_loader):
        if is_cuda:
            data1,data2,data3,target =
data1.cuda(),data2.cuda(),data3.cuda(),target.cuda()
            data1,data2,data3,target =
Variable(data1,volatile),Variable(data2,volatile),Variable(data3,volatile),
Variable(target)
        if phase == 'training':
            optimizer.zero_grad()
        output = model(data1,data2,data3)
        loss = F.cross_entropy(output,target)

        running_loss +=
F.cross_entropy(output,target,size_average=False).data[0]
        preds = output.data.max(dim=1,keepdim=True)[1]
        running_correct +=
preds.eq(target.data.view_as(preds)).cpu().sum()
        if phase == 'training':
            loss.backward()
            optimizer.step()

    loss = running_loss/len(data_loader.dataset)
    accuracy = 100. * running_correct/len(data_loader.dataset)

    print(f'{phase} loss is {loss:.2f} and {phase} accuracy is
{running_correct}/{len(data_loader.dataset)}{accuracy:.4f}')
    return loss,accuracy
```

As you can see, most of the code remains the same, except that the loader returns three inputs and one label. Therefore, we have to make changes to the function, which is self-explanatory.

The following is the training code:

```
train_losses , train_accuracy = [], []
validation_losses , validation_accuracy = [], []
for epoch in range(1,10):
    epoch_loss, epoch_accuracy =
    fit(epoch,em,trn_feat_loader,phase='training')
    validation_epoch_loss , validation_epoch_accuracy =
    fit(epoch,em,validation_feat_loader,phase='validation')
    train_losses.append(epoch_loss)
    train_accuracy.append(epoch_accuracy)
    validation_losses.append(validation_epoch_loss)
    validation_accuracy.append(validation_epoch_accuracy)
```

The result of the preceding code is as follows:

```
training loss is 7.2e+01 and training accuracy is 21359/23000 92.87
validation loss is 6.5e+01 and validation accuracy is 1968/2000 98.4
training loss is 9.4e+01 and training accuracy is 22539/23000 98.0
validation loss is 1.1e+02 and validation accuracy is 1980/2000 99.0
training loss is 1e+02 and training accuracy is 22714/23000 98.76
validation loss is 1.4e+02 and validation accuracy is 1976/2000 98.8
training loss is 7.3e+01 and training accuracy is 22825/23000 99.24
validation loss is 1.6e+02 and validation accuracy is 1979/2000 98.95
training loss is 7.2e+01 and training accuracy is 22845/23000 99.33
validation loss is 2e+02 and validation accuracy is 1984/2000 99.2
training loss is 1.1e+02 and training accuracy is 22862/23000 99.4
validation loss is 4.1e+02 and validation accuracy is 1975/2000 98.75
training loss is 1.3e+02 and training accuracy is 22851/23000 99.35
validation loss is 4.2e+02 and validation accuracy is 1981/2000 99.05
training loss is 2e+02 and training accuracy is 22845/23000 99.33
validation loss is 6.1e+02 and validation accuracy is 1982/2000 99.1
training loss is 1e+02 and training accuracy is 22917/23000 99.64
validation loss is 5.3e+02 and validation accuracy is 1986/2000 99.3
```

The ensemble model achieves 99.6% training accuracy and a validation accuracy of 99.3%. Though ensemble models are powerful, they are computationally expensive. They are good techniques to use when you are solving problems in competitions such as Kaggle.

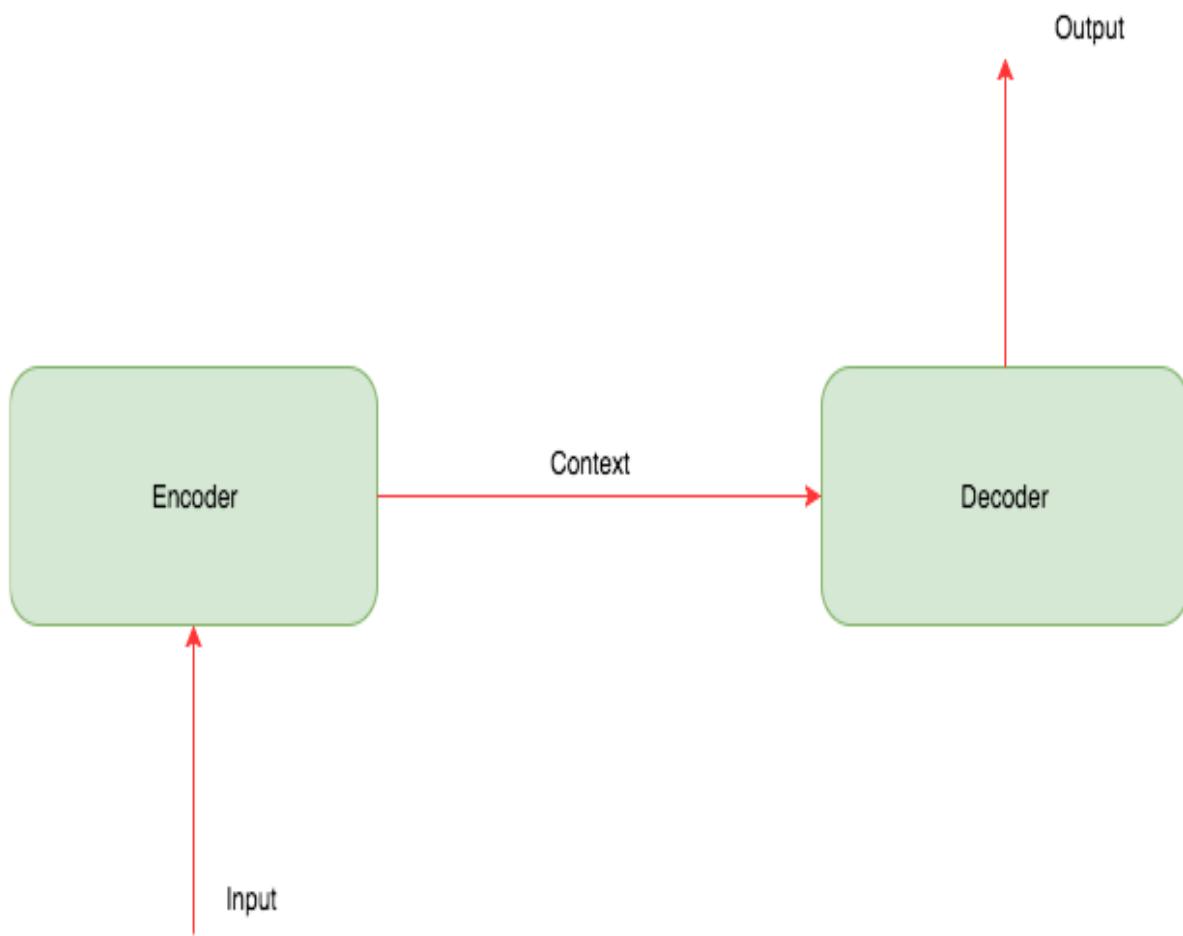
Encoder-decoder architecture

Almost all the deep learning algorithms we have seen in this book are good at learning how to map training data to their corresponding labels. We cannot use them directly for tasks where the model needs to learn from a sequence and generate another sequence or an image. Some example applications are as follows:

- Language translation
- Image captioning
- Image generation (seq2img)
- Speech recognition
- Question answering

Most of these problems can be seen as forms of sequence-to-sequence mapping, and these can be solved using a family of architectures called encoder-decoder architectures. In this section, we will learn about the intuition behind these architectures. We will not be looking at the implementation of these networks as they need to be studied in more detail.

At a high level, the encoder-decoder architecture looks as follows:



An encoder is usually a **recurrent neural network (RNN)** (for sequential data) or a **convolutional neural network (CNN)** (for images) that takes in an image or a sequence and converts it into a fixed-length vector that encodes all the information. The decoder is another RNN or CNN, which learns to decode the vector that was generated by the encoder and generates a new sequence of data. The following diagram shows what the encoder-decoder architecture looks like for an image captioning system:

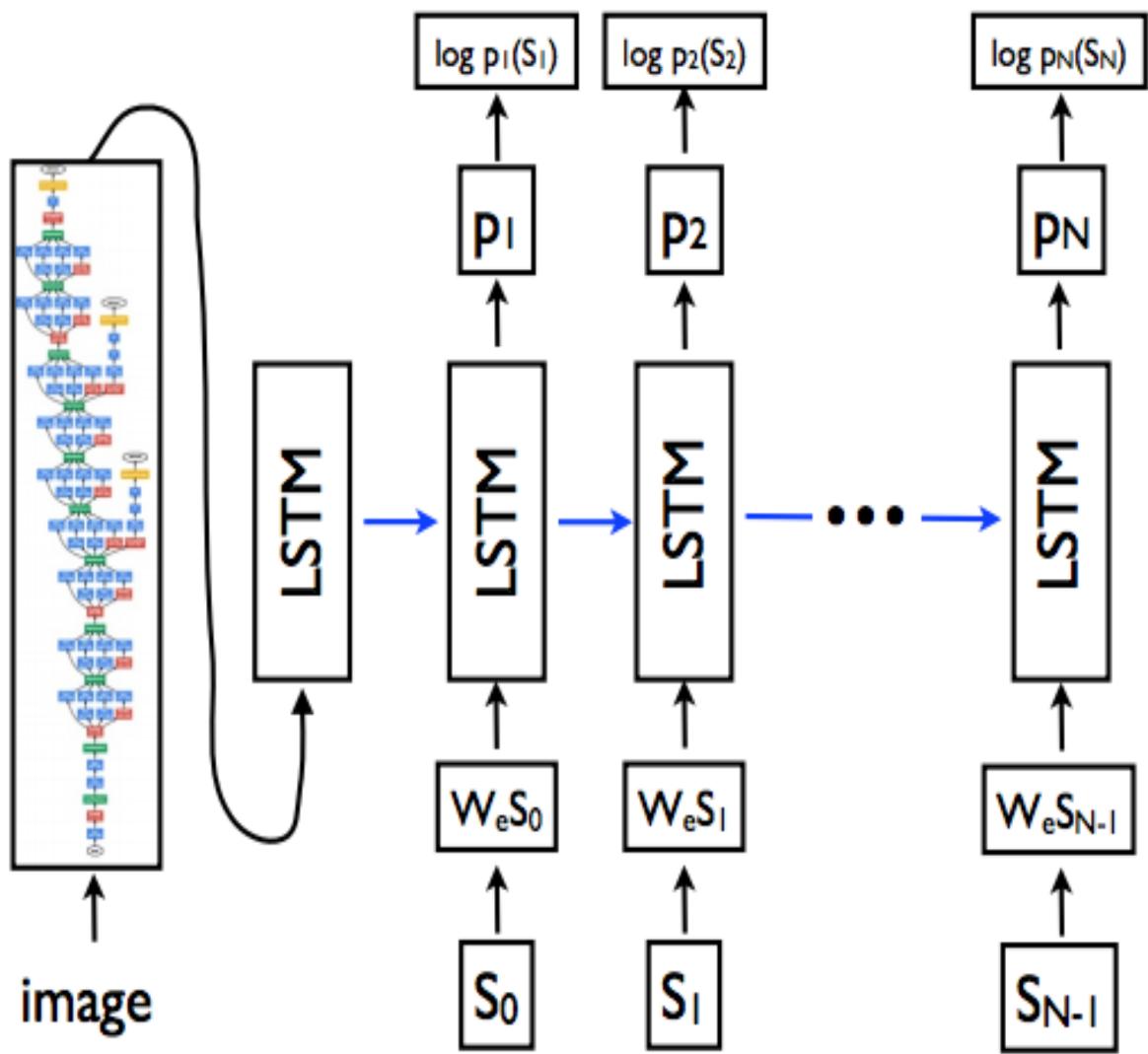


Image source: <https://arxiv.org/pdf/1411.4555.pdf>

Now, let's look at what happens inside an encoder and a decoder architecture for an image captioning system.

Encoder

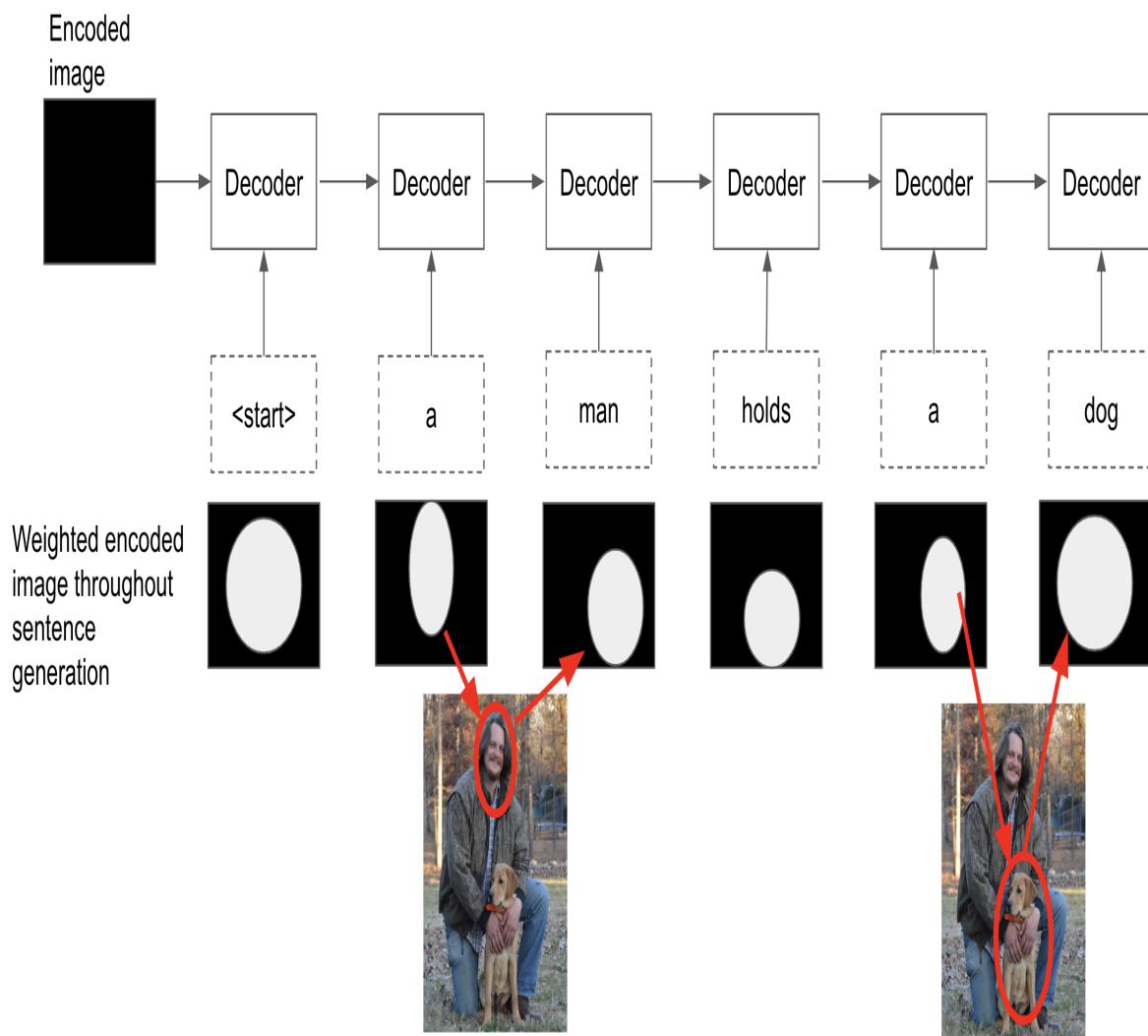
For an image captioning system, we should use a trained architecture, such as ResNet or Inception, to extract features from the image. Like we did for the ensemble model, we can output a fixed vector length by using a linear layer and then make that linear layer trainable.

Decoder

A decoder is a **long short-term memory (LSTM)** layer that will generate a caption for an image. To build a simple model, we can just pass the encoder embedding as input to the LSTM. However, this could be quite challenging for the decoder to learn; instead, it is common practice to provide the encoder embedding at every step of the decoder. Intuitively, a decoder learns to generate a sequence of text that best describes the caption of a given image.

Encoder-decoder with attention

In 2017, a paper titled *Attention Is All You Need*, by Ashish Vaswani and co. (<https://arxiv.org/pdf/1706.03762.pdf>), was published, and it incorporates an attention mechanism. At each timestep, the attention network computes the weights of the pixels. It considers the sequence of words that have been generated so far and outputs what should be described next:



In the preceding example, we can see that it is the LSTM's ability to retain information that can help it to learn that it is logical to write "*is holding a dog*" after "*a man*".

Summary

In this chapter, we explored some modern architectures, such as ResNet, Inception, and DenseNet. We also explored how we can use these models for transfer learning and ensembling, and also introduced the encoder-decoder architecture, which powers a lot of systems, such as language translation systems.

In the next chapter, we will be diving into deep reinforcement learning and learning how models can be applied to solve problems in the real world. We will also look at some PyTorch implementations that can help with this.

Deep Reinforcement Learning

This chapter starts with a basic introduction to **Reinforcement Learning (RL)**, including agents, state, actions, rewards, and policies. It extends to **Deep Learning (DL)**-based architectures for RL problems such as policy gradient methods, Deep-Q networks, and actor-critic models. This chapter will explain how to use these deep learning architectures with hands-on code to solve sequential decision-making problems in the OpenAI Gym environment.

Specifically, the following will be covered:

- Introduction to RL
- Using DL to tackle RL
- Policy gradients and code walk-through in PyTorch
- Deep-Q networks and code walk-through in PyTorch
- Actor-critic networks and code walk-through in PyTorch
- Applications of RL in the real world

Introduction to RL

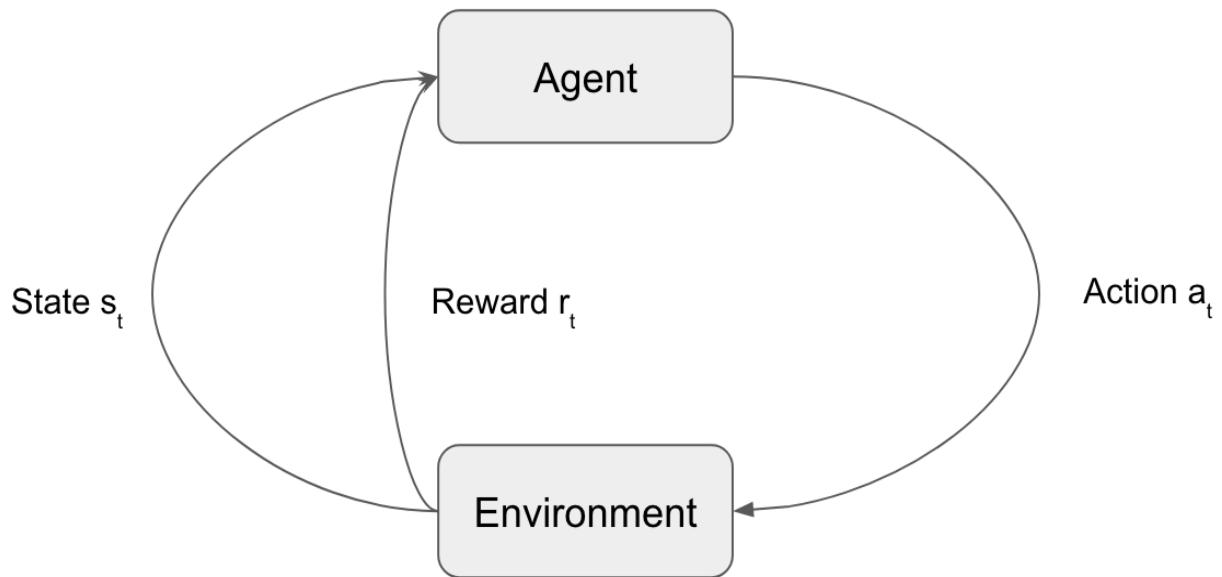
RL is a branch of machine learning where an agent learns to behave optimally in a given environment. The agent performs certain actions and observes the rewards/results. It learns the process of mapping situations to actions to maximize a reward.

The RL process can be modeled as an iterative loop and can be represented as a mathematical framework called the **Markov Decision Process (MDP)**. The following steps outline the process that takes place:

1. RL agent receives state (s_0) from the environment.
2. The RL agent takes an action (a_0) given the current state (s_0). At this stage, the action it takes is random as the agent does not have any previous knowledge about the reward it could receive if it performs the action.
3. After the first action has taken place, the agent can now be considered to be in state s_1 .
4. At this point, the environment gives a reward (r_1) to the agent.

This loop is repeated continuously; it outputs a sequence of state and action and observes the reward.

This process is essentially an algorithm for learning an MDP policy:



The cumulative rewards at each time step with respect to the given action can be represented as follows:

$$\sum_{i=1}^T r_i$$

In some applications, it may be beneficial to give more weight to rewards that are received sooner in time. For example, it would be better for you to receive £100 today rather than in 5 years' time. To incorporate this, it is common to introduce a discount factor, γ .

The cumulative discounted rewards are represented as follows:

$$\sum_{i=1}^T \gamma^{i-1} r_i$$

An important consideration in RL is the fact that rewards may be infrequent and delayed. In cases where there is a long-delayed reward, it can be challenging to trace back which sequence of actions contributed to the award.

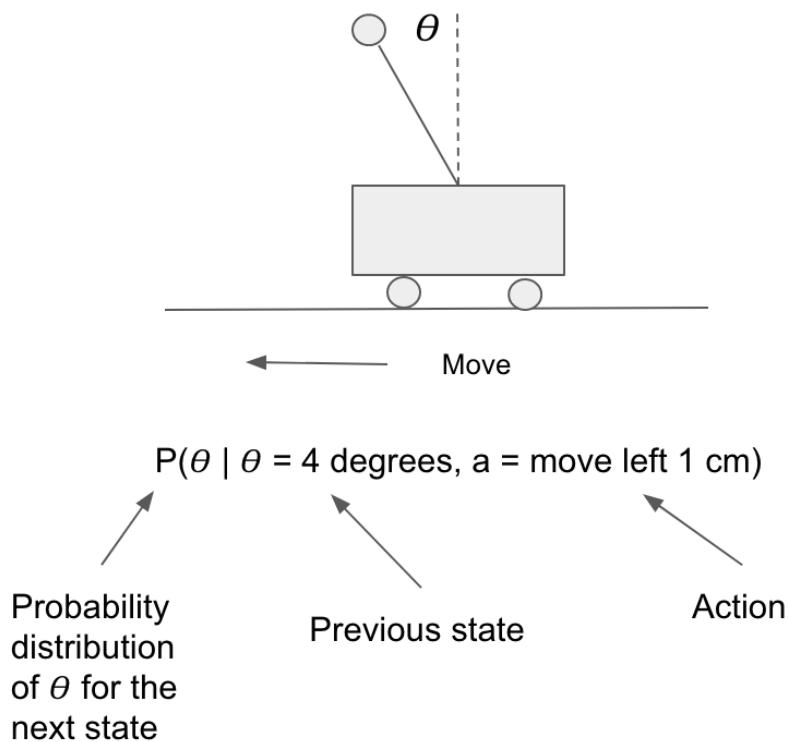
Model-based RL

Model-based RL mimics the behavior of the environment. It predicts the next state after taking an action. It can be represented mathematically as a probability distribution:

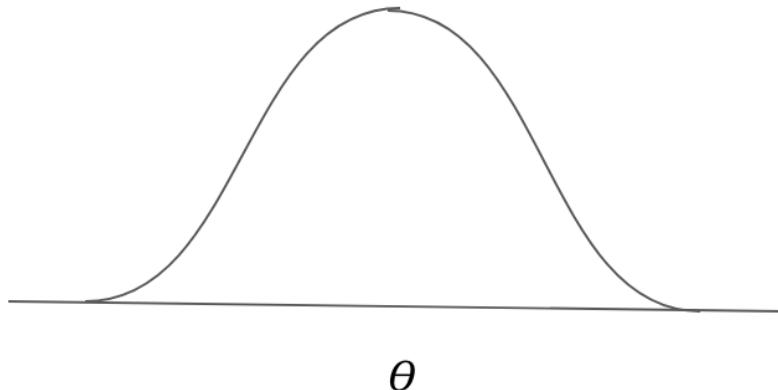
$$x_t = P(x_{t-1}, a_{t-1})$$

Here, p denotes the model, x is the state, and a is the control or action.

This notion can be demonstrated by considering the cart-pole example. The goal is for the pole attached to the cart to stay upright and the agent can decide between two possible actions: to move the cart left or to move the cart right. In the following screenshot, P models the angle of the pole after taking an action:

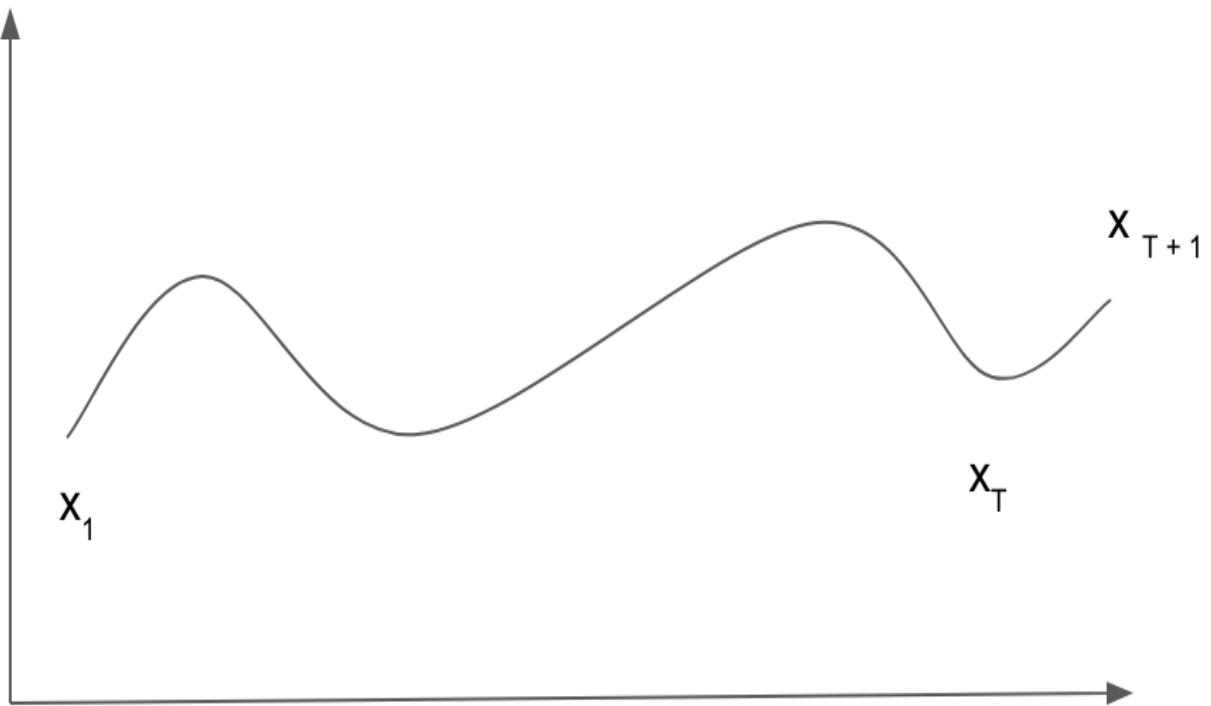


The following diagram depicts the probability distribution output for θ in the subsequent time step:



In this instance, the model describes the law of physics; however, the model could be anything depending on the application. Another example is that the model could be built on the rules of a game of chess.

The core concept of model-based RL is to use the model along with the cost function to locate the optimal path of actions or, in other words, the trajectory of states and actions, τ :



$$\min_{u_1, \dots, u_r} \sum_{t=1}^T c(x + t, u_t) \text{ s.t. } x_t = f(x_{t-1}, u_{t-1})$$

cost model

Trajectory of states and actions

$$\tau = \{s_1, a_1, s_2, a_2, s_3, a_3, \dots, s_T, a_T\}$$

The drawback of model-based algorithms is that they can become impractical as the state space and action space become larger.

Model-free RL

Model-free algorithms rely on trial and error to update their knowledge. As such, they do not require space to store all of the combinations of states and actions. Policy gradients, value learning, or other model-free RL are used to find a policy that takes the best actions for maximum rewards. A key difference between model-free and model-based methods is that model-free methods act in the real environment to learn.

Comparing on-policy and off-policy

The policy defines how the agent behaves; it tells the agent how to act in each state. Every RL algorithm must follow some type of policy to decide how it will act.

The policy function the agent is trying to learn can be represented as follows, where θ is the parameter vector, s is a particular state, and a is an action:

$$\pi_\theta(s, a)$$

An on-policy agent learns the value (the expected discounted rewards) based on its current action and is derived from the current policy. Off-policy learns the value based on the action obtained from another policy such as a greed policy as in Q-learning, which we introduce next.

Q-learning

Q-learning is a model-free RL algorithm whereby a table is created that calculates the maximum expected future reward for each action at each state. It is considered off-policy because the Q-learning function learns from actions that are outside of the current policy.

When Q-learning is performed, a Q-table is created whereby the columns represent the possible actions and the rows represent the states. The value of each cell in the Q-table will be the maximum expected future reward for that given state and action:

Actions			
States			

Each Q-table score will be the maximum expected future reward if taking the action from the best policy. To learn each value in the Q-table, the Q-learning algorithm is used.

The **Q-function** (or **action-value function**) takes two inputs: state and reward. The Q-function returns the expected future reward of that action at that state. It can be represented as shown:

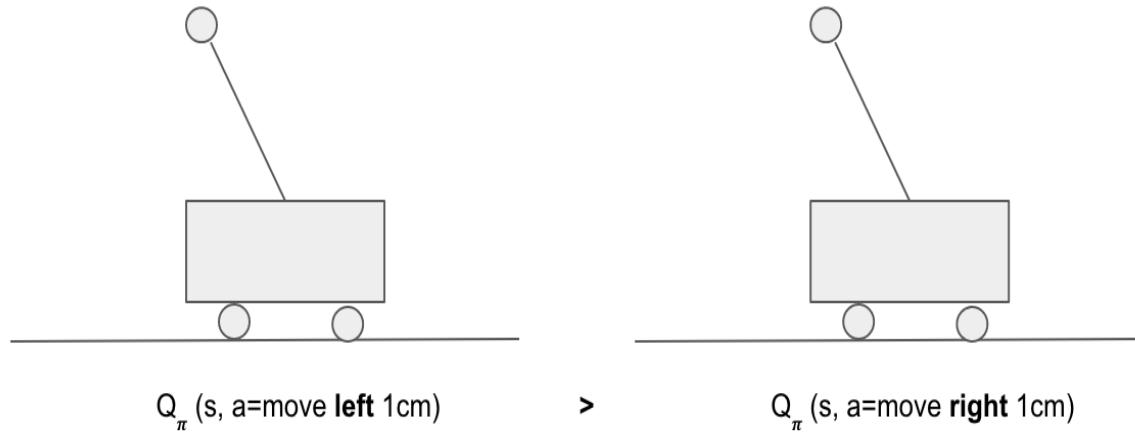
$$Q^\pi(s_t, a_t) = \underline{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | s_t, a_t]$$

Q-Value for the state given
the action

Expected discounted cumulative reward
given the state and action

The Q-function essentially scrolls through the Q-table to find the row associated with the current state and the column associated with the action. From here, it returns the Q value that is the corresponding expected future reward.

Consider this in the cart-pole example shown in the following figure. In its current state, moving left should have a higher Q-value than moving right:



As the environment is explored, the Q-table is updated to give better approximations. The process of the Q-learning algorithm is as follows:

1. The Q-table is initialized.
2. An action in the current state (s) is chosen based on the current Q-value estimates.
3. An action (a) is performed.
4. The reward (r) is measured.
5. Q is updated using the Bellman equation.

The following is the Bellman equation:

$$Q(s, a) = r(s, a) + \gamma \max_a Q(s', a)$$

Steps 2-5 are repeated until the maximum number of episodes is reached or until the training is manually stopped.

The Q-learning algorithm can be expressed as the following equation:

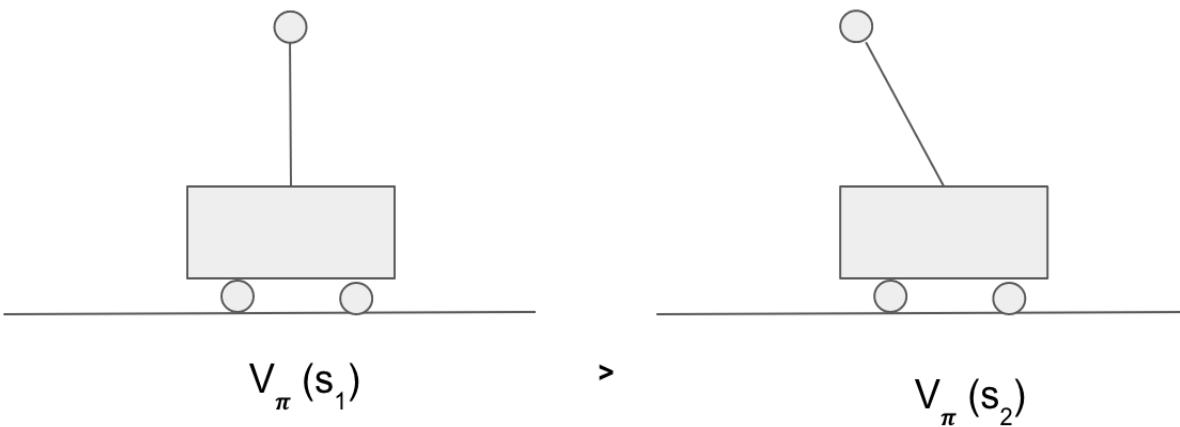
$$Q^{new}(s_t, a_t) \leftarrow \underbrace{(1 - \alpha) \cdot Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\substack{\text{learned value} \\ \text{estimate of optimal future value}}} \right)}^{\text{learned value}}$$

Value methods

Value learning is an approach that can often be a key building block in many RL methods. A value function, $V(s)$, represents how good the state is that the agent is in. It is equal to the expected total reward for an agent starting from the state, s . The total expected reward is dependent on the policy by which the agent chooses actions to perform. If the agent uses a given policy (π) to select its actions, the corresponding value function is given by the following formula:

$$V^\pi(s) = \mathbb{E} \left[\sum_{i=1}^T \gamma^{i-1} r_i \right] \forall s \in \mathbb{S}$$

Considering this in the cart-pole example, we could use the length of time the pole stays up in order to measure the rewards. In the following screenshot, there is a higher probability that the pole will stay upright for state s_1 compared to state s_2 . As such, for most policies, the state s_1 is likely to have a higher value function (a higher expected future reward):



There is an optimal value function that has a higher value than other functions for all states and this can be represented as the following:

$$V^*(s) = \max_{\pi} V^\pi(s) \forall s \in \mathbb{S}$$

There is an optimal policy that corresponds to the optimal value function:

$$\pi^* = \arg \max_{\pi} V^\pi(s) \forall s \in \mathbb{S}$$

There are several different ways in which the optimal policy can be found. This is referred to as policy evaluation.

Value iteration

Value iteration is a process that computes the optimal state-value function by iteratively improving estimates of $V(s)$. Firstly, the algorithm initializes $V(s)$ to be random values. From there, it repeatedly updates the values of $Q(s, a)$ and $V(s)$ until they converge. Value iteration converges to the optimal values.

The following is the value iteration pseudocode:

```
Initialize  $V(s)$  to arbitrary values
Repeat
    For all  $s \in S$ 
        For all  $a \in \mathcal{A}$ 
            
$$Q(s, a) \leftarrow E[r|s, a] + \gamma \sum_{s' \in S} P(s'|s, a)V(s')$$

            
$$V(s) \leftarrow \max_a Q(s, a)$$

    Until  $V(s)$  converge
```

Coded example – value iteration

To consider an example of this, we will use the Frozen Lake environment in OpenAI Gym. In the environment, the player is to imagine they are standing on a frozen lake where not all of it is frozen. The goal is to move from place S to place G without falling into the holes:

S	F	F	F
F	H	F	H
F	F	F	H
H	F	F	G

The letters on the grid represent the following:

- S: Starting point, safe
- F: Frozen surface, safe
- H: Hole, not safe
- G: Goal

There are four possible moves that the agent can take: left, right, down, and up, which are represented as 0, 1, 2, and 3 respectively. As such, there are 16 possible states to be in (4×4). For every H state, the agent receives a reward of -1 and upon reaching the goal, receives a reward of +1.

To implement value-iteration in code, we first import the relevant libraries we wish to use and initialize the `FrozenLake` environment:

```
import gym
import numpy as np
import time, pickle, os
env = gym.make('FrozenLake-v0')
```

Now, we assign the variables:

```
# Epsilon for an epsilon greedy approach
epsilon = 0.95
total_episodes = 1000
# Maximum number of steps to be run for every episode
maximum_steps = 100
learning_rate = 0.75
# The discount factor
gamma = 0.96
```

From here, we initialize the Q-table where `env.observation_space.n` is the number of states and `env.action_space.n` is the number of actions:

```
| Q = np.zeros((env.observation_space.n, env.action_space.n))
```

Define the functions for the agent to choose an action and learn:

```
def select_action(state):
    action=0
    if np.random.uniform(0, 1) < epsilon:

        # If the random number sampled is smaller than epsilon then a
        random action is chosen.

        action = env.action_space.sample()
    else:
        # If the random number sampled is greater than epsilon then we
        choose an action having the maximum value in the Q-table
        action = np.argmax(Q[state, :])
    return action

def agent_learn(state, state_next, reward, action):
    predict = Q[state, action]
    target = reward + gamma * np.max(Q[state_next, :])
    Q[state, action] = Q[state, action] + learning_rate * (target -
predict)
```

From here, we can begin running the episodes and export the Q-table to a pickled file:

```
for episode in range(total_episodes):
    state = env.reset()
    t = 0

    while t < maximum_steps:
        env.render()
        action = select_action(state)
        state_next, reward, done, info = env.step(action)
        agent_learn(state, state_next, reward, action)
        state = state_next
```

```
t += 1
if done:
    break
time.sleep(0.1)

print(Q)
with open("QTable_FrozenLake.pkl", 'wb') as f:
    pickle.dump(Q, f)
```

We can see the process in action by running the preceding code:

```
SFFF
FHFH
FFFF
HFFG
      (Right)
SFFF
FHFH
FFFF
HFFG
      (Left)
SFFF
FHFH
FFFF
HFFG
      (Right)
```

Policy methods

In policy-based RL, the goal is to find the policy that makes the most rewarding decisions and can be represented mathematically as follows:

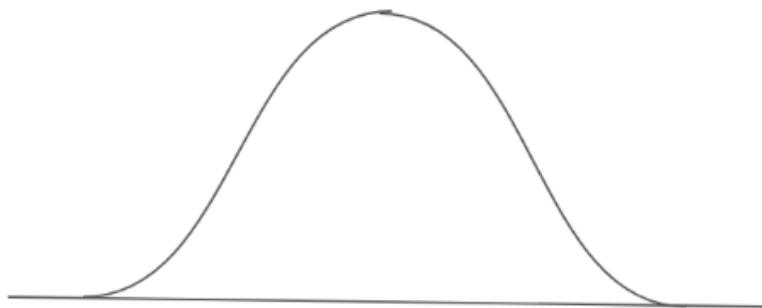
$$\max_{\theta} E_{\tau \sim p_{\theta}(\tau)} \left[\sum_t r(s_t, a_t) \right]$$

↑ ↑
Find policy Expected
with max rewards
rewards

Policies in RL can be either deterministic or stochastic:

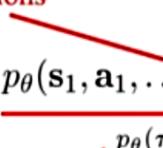
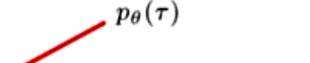
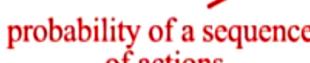
$$p(a_t | s_t) = \pi_{\theta}(a_t | s_t)$$

Stochastic policies output a probability distribution rather than a single discrete value:



We can represent the objective mathematically as shown:

$$p_{\theta}(\mathbf{s}_1, \mathbf{a}_1, \dots, \mathbf{s}_T, \mathbf{a}_T) = p(\mathbf{s}_1) \prod_{t=1}^T \pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t) p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)$$

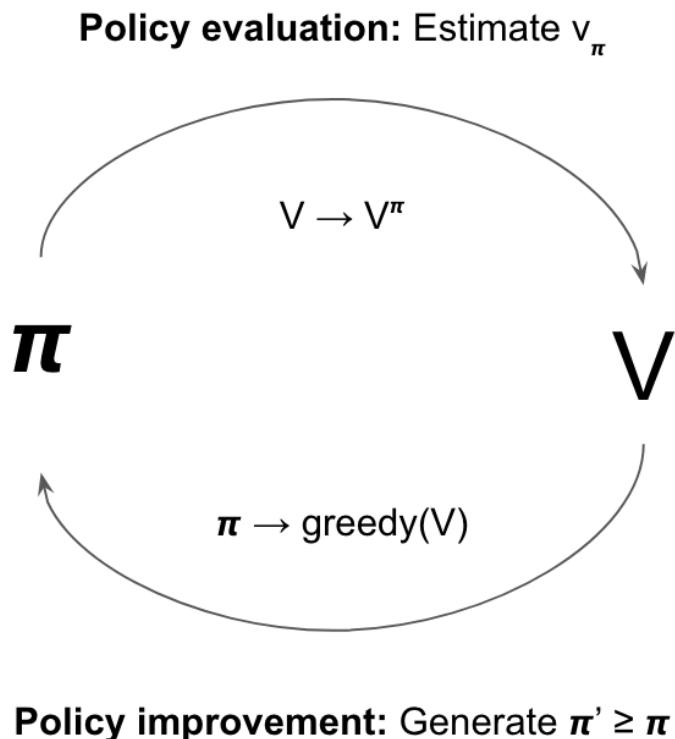
sequence of actions 
 $p_{\theta}(\tau)$ 
probability of a sequence of actions 

model 
policy
rewards 

$$\max_{\theta} E_{\tau \sim p_{\theta}(\tau)} \left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right]$$

Policy iteration

During value iteration, sometimes the optimal policy converges before the value function does since it only cares about finding the optimal policy. Another algorithm called policy iteration can be performed to reach the optimal policy. This is when, after each policy evaluation has taken place, the policy of the next is based on the value function until the policy converges. The following diagram illustrates the policy iteration process:



As such, the policy iteration algorithm is guaranteed to converge to the optimal policy, unlike the value iteration algorithm.

The following is the policy iteration pseudocode:

Initialize a policy π' arbitrarily

Repeat

$$\pi \leftarrow \pi'$$

Compute the values using π by
solving the linear equations

$$V^\pi(s) = E[r|s, \pi(s)] + \gamma \sum_{s' \in S} P(s'|s, \pi(s))V^\pi(s')$$

Improve the policy at each state

$$\pi'(s) \leftarrow \arg \max_a (E[r|s, a] + \gamma \sum_{s' \in S} P(s'|s, a)V^\pi(s'))$$

Until $\pi = \pi'$

Coded example – policy iteration

Here, we consider a coded example of this using the Frozen Lake environment as earlier.

First, import the relevant libraries:

```
import numpy as np
import gym
from gym import wrappers
```

Now, we define the functions to run an episode and return the reward:

```
def run_episode_return_reward(environment, policy, gamma_value = 1.0,
render = False):
    """ Runs an episode and return the total reward """
    obs = environment.reset()
    total_reward = 0
    step_index = 0
    while True:
        if render:
            environment.render()
        obs, reward, done , _ = environment.step(int(policy[obs]))
        total_reward += (gamma_value ** step_index * reward)
        step_index += 1
        if done:
            break
    return total_reward
```

From here, we can define functions to evaluate the policy:

```
def evaluate_policy(environment, policy, gamma_value = 1.0, n = 200):
    model_scores = [run_episode_return_reward(environment, policy,
gamma_value, False) for _ in range(n)]
    return np.mean(model_scores)
```

Then, define the functions to extract the policy:

```
def extract_policy(v, gamma_value = 1.0):
    """ Extract the policy for a given value function """
    policy = np.zeros(environment.nS)
```

```

    for s in range(environment.nS):
        q_sa = np.zeros(environment.nA)
        for a in range(environment.nA):
            q_sa[a] = sum([p * (r + gamma_value * v[s_]) for p, s_, r, _
in environment.P[s][a]])
        policy[s] = np.argmax(q_sa)
    return policy

```

Finally, define the function to compute the policy:

```

def compute_policy_v(environment, policy, gamma_value=1.0):
    """ Iteratively evaluate the value-function under policy.
    Alternatively, we could formulate a set of linear equations in terms
    of v[s]
    and solve them to find the value function.
    """
    v = np.zeros(environment.nS)
    eps = 1e-10
    while True:
        prev_v = np.copy(v)
        for s in range(environment.nS):
            policy_a = policy[s]
            v[s] = sum([p * (r + gamma_value * prev_v[s_]) for p, s_, r,
in environment.P[s][policy_a]])
        if (np.sum((np.fabs(prev_v - v))) <= eps):
            # value converged
            break
    return v

```

From here, we can run policy iteration on the Frozen Lake environment:

```

env_name = 'FrozenLake-v0'
environment = gym.make(env_name)
optimal_policy = policy_iteration(environment, gamma_value = 1.0)
scores = evaluate_policy(environment, optimal_policy, gamma_value = 1.0)
print('Average scores = ', np.mean(scores))

```

We observe that it has converged at step 5:

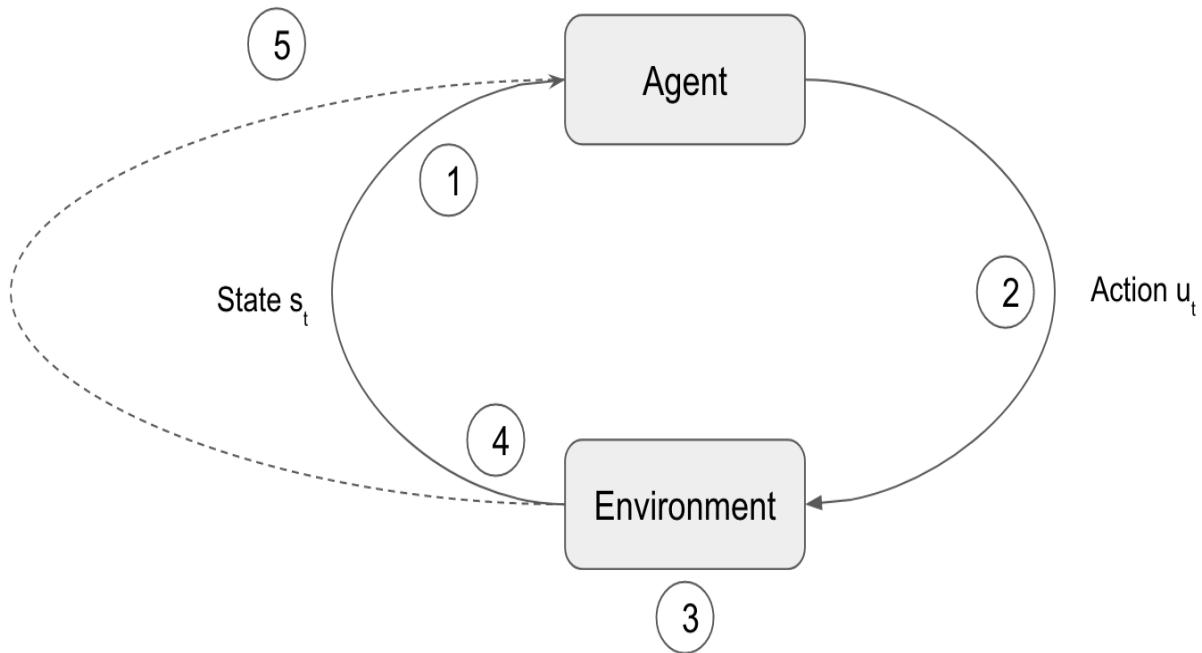
**Policy-Iteration converged at step 5.
Average scores = 0.74**

Value iteration versus policy iteration

Where the agent is assumed to have some prior knowledge about the effects of its actions on the environment, both value and policy iteration algorithms can be used. The algorithms assume that the MDP model is known. Policy iteration, however, is more computationally efficient as it often takes a lower number of iterations to converge.

Policy gradient algorithm

Policy gradient is also an approach to solve reinforcement learning problems and aims to model and optimize the policy directly:



In policy gradients, the following steps are taken:

1. The agent observes the state of the environment (s).
2. The agent takes action u based on their instincts (a policy, π) about the state (s).
3. The agent moves and the environment changes; a new state is formed.
4. The agent takes further actions based on the observed state.
5. After a trajectory (τ) of motions, the agent adjusts its instinct based on the total rewards, $R(\tau)$, received.

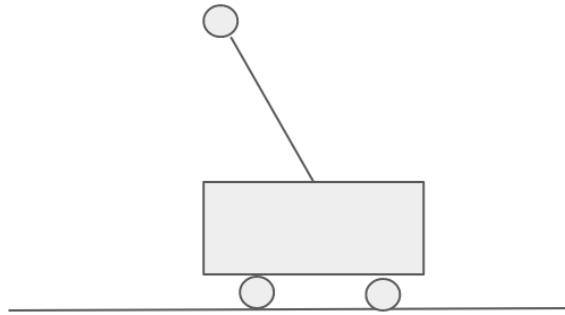
The policy gradient theorem is as follows:

The derivative of the expected reward is the expectation of the product of the reward and gradient of the log of the policy π_θ :

$$\nabla \mathbb{E}_{\pi_\theta} [r(\tau)] = \mathbb{E}_{\pi_\theta} [r(\tau) \nabla \log \pi_\theta(\tau)]$$

Coded example – policy gradient algorithm

In this example, we use the OpenAI environment named CartPole where the goal is for the pole attached to the cart to stay upright for as long as possible. The agent receives a reward for every time step taken in which the pole remains balanced. If the pole falls over, the episode ends:



At any point in time, the cart and pole are in a state, s . This state is represented by a vector of four elements, namely, pole angle, pole velocity, cart position, and cart velocity. The agent can decide between two possible actions: to move the cart left or to move the cart right.

A policy gradient takes small steps and updates the policy based on the reward that is associated with the step. It does this so that it can train the agent without having to map the value for every pair of states and actions in the environment.

In this example, we will apply a technique called the Monte-Carlo policy gradient. Using this approach, the agent will update the policy at the end of each episode based on the rewards obtained.

We will first import the relevant libraries that we plan to use:

```
import gym
import numpy as np
from tqdm import tqdm, trange
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.autograd import Variable
from torch.distributions import Categorical
```

Now, we define a feedforward neural network with one hidden layer of 128 neurons and a dropout of 0.5. We use Adam as the optimizer and a learning rate of 0.02. Using a dropout significantly improves the performance of the policy:

```
class PolicyGradient(nn.Module):
    def __init__(self):
        super(PolicyGradient, self).__init__()

        # Define the action space and state space
        self.action_space = env.action_space.n
        self.state_space = env.observation_space.shape[0]

        self.l1 = nn.Linear(self.state_space, 128, bias=False)
        self.l2 = nn.Linear(128, self.action_space, bias=False)

        self.gamma_value = gamma_value

        # Episode policy and reward history
        self.history_policy = Variable(torch.Tensor())
        self.reward_episode = []

        # Overall reward and loss history
        self.history_reward = []
        self.history_loss = []

    def forward(self, x):
        model = torch.nn.Sequential(
            self.l1,
            nn.Dropout(p=0.5),
            nn.ReLU(),
            self.l2,
            nn.Softmax(dim=-1)
        )
        return model(x)

policy = PolicyGradient()
optimizer = optim.Adam(policy.parameters(), lr=l_rate)
```

Now, we define a `choose_action` function. This function chooses an action based on the policy distribution with the aid of the PyTorch distributions package. The policy returns a probability for each possible action on the action space as an array. In our example, this is to move left or to move right so the output could be [0.1, 0.9]. Based on these probabilities, the action is chosen, the history is recorded, and the action is returned:

```
def choose_action(state):
    # Run the policy model and choose an action based on the
    # probabilities in state
    state = torch.from_numpy(state).type(torch.FloatTensor)
    state = policy.Variable(state)
    c = Categorical(state)
    action = c.sample()
    if policy.history_policy.dim() != 0:
        try:
            policy.history_policy = torch.cat([policy.history_policy,
c.log_prob(action)])
        except:
            policy.history_policy = (c.log_prob(action))
    else:
        policy.history_policy = (c.log_prob(action))
    return action
```

To update the policy, we take a sample of the Q-function (action-value function). Recall that this is the expected return by taking an action in a state by following the policy, π . We can calculate the policy gradient at each time step using the fact that there is a reward of 1 for every step the pole remains vertical. We use the long term reward (v_t) where this is the discounted sum of all future rewards for the length of the episode. As such, the longer the episode, the greater the reward for a state-action pair in the present where gamma is the discount factor.

The discounted reward vector is denoted as follows:

$$v_t = \sum_{k=0}^N \gamma^k r_{t+k}$$

For example, if an episode lasts 4 time steps, the reward for each step would be [4.90, 3.94, 2.97, 1.99]. From here, we can scale the reward vector by subtracting the mean and dividing by the standard deviation.

To improve the policy after each episode, we apply the Monte-Carlo policy gradient, as follows:

$$\Delta\theta_t = \alpha \nabla_{\theta} \log \pi_{\theta}(s_t, a_t) v_t$$

This policy is then multiplied by the rewards and fed into the optimizer and the weights of the neural network are updated using stochastic gradient descent.

The following function defines how we can update the policy in code:

```
def update_policy():
    R = 0
    rewards = []

    # Discount future rewards back to the present using gamma
    for r in policy.reward_episode[::-1]:
        R = r + policy.gamma_value * R
        rewards.insert(0,R)

    # Scale rewards
    rewards = torch.FloatTensor(rewards)
    x = np.finfo(np.float32).eps
    x = np.array(x)
    x = torch.from_numpy(x)
    rewards = (rewards - rewards.mean()) / (rewards.std() + x)
    # Calculate the loss loss
    loss = (torch.sum(torch.mul(policy.history_policy,
Variable(rewards)).mul(-1), -1))

    # Update the weights of the network
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    #Save and intialize episode history counters
    policy.history_loss.append(loss.data[0])
    policy.history_reward.append(np.sum(policy.reward_episode))
    policy.history_policy = Variable(torch.Tensor())
    policy.reward_episode= []
```

From here, we define the main policy training loop. At every step in a training episode, an action is chosen and the new state and reward are recorded. The `update_policy` function is called at the end of each episode to feed the episode history into the neural network:

```
def main_function(episodes):
    running_total_reward = 50
    for e in range(episodes):
        # Reset the environment and record the starting state
        state = env.reset()
        done = False

        for time in range(1000):
            action = choose_action(state)
            # Step through environment using chosen action
            state, reward, done, _ = env.step(action.data.item())

            # Save reward
            policy.reward_episode.append(reward)
            if done:
                break

        # Used to determine when the environment is solved.
        running_total_reward = (running_total_reward * 0.99) + (time *
0.01)

        update_policy()

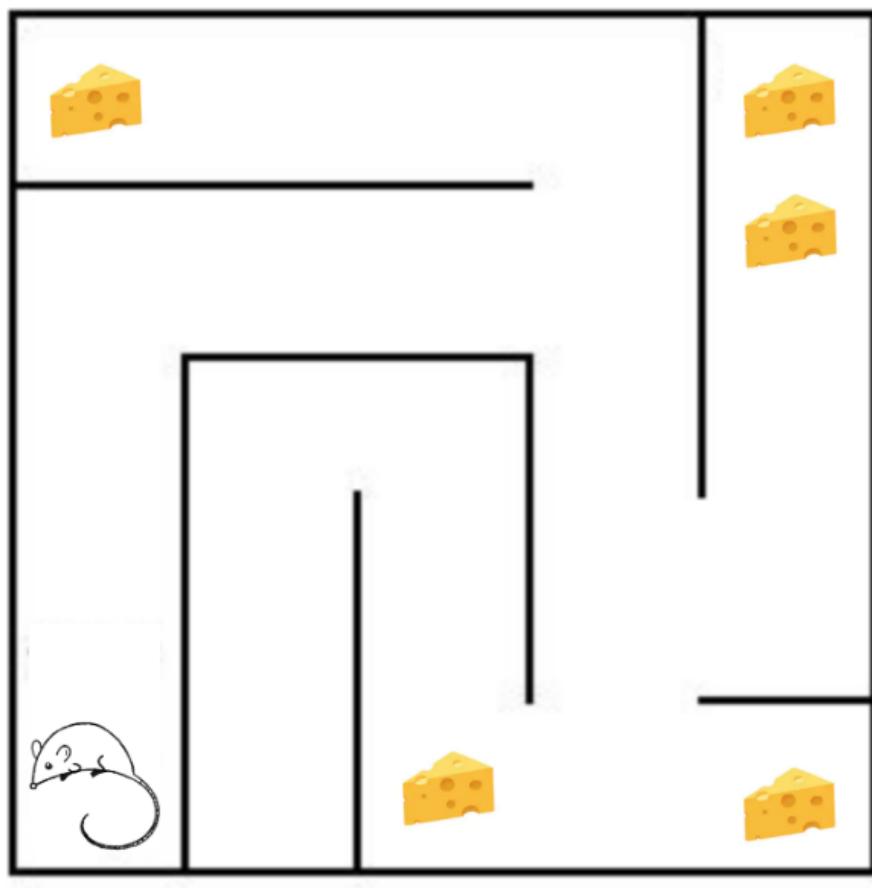
        if e % 50 == 0:
            print('Episode number {}, Last length: {:5d}, Average length:
{:.2f}'.format(e, time, running_total_reward))

            if running_total_reward > env.spec.reward_threshold:
                print("Solved! Running reward is now {} and the last episode
runs to {} time steps!".format(running_total_reward, time))
                break

    episodes = 2000
    main_function(episodes)
```

Deep Q-networks

Deep-Q Networks (DQNs) combine deep learning and RL to learn in several different applications, particularly computer games. Let's consider a simplified example of a game where there is a mouse in a maze and where the goal is for the mouse to eat as much cheese as possible. The more cheese the mouse eats, the more points it gets in the game:

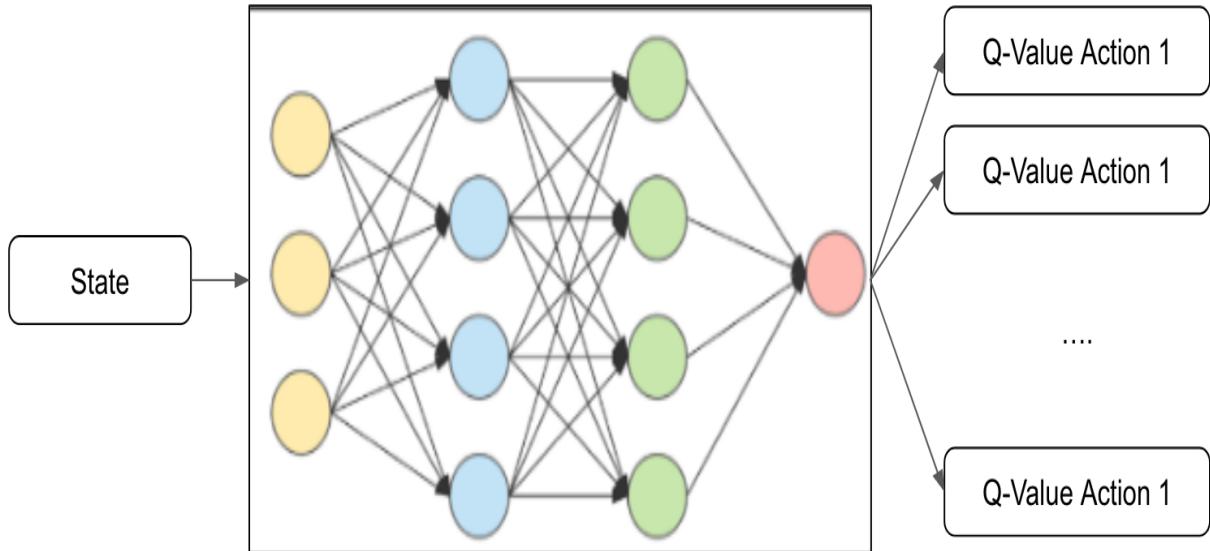


In this example, the RL terms would be as follows:

- Agent: The mouse as this is controlled by the computer
- State: The current moment in the game

- Action: A decision made by the mouse (to move left, right, up, or down)
- Reward: The score in the game/amount of cheese the mouse has eaten—in other words, the value that the agent is trying to maximize

DQNs use Q-learning to learn the best action in a given state. They use **convolutional neural networks (ConvNets)** as a function approximator for the Q-learning function. ConvNets use convolutional layers to find spatial features such as where the mouse currently is in the grid. This means that the agent only has to learn Q-values for a few million rather than billions of different game states:



An example of a DQN architecture when learning the mouse maze game is as follows:

1. The current state (maze screen) is fed as input into the DQN.
2. The input is passed through convolutional layers to find spatial patterns in the image. Note that no pooling is used as it is important to know the spatial position when modeling computer games.
3. The output of the convolutional layers is fed into fully connected linear layers.

4. The output of the linear layers gives the probability that the DQN will take an action given its current state (move up, down, left, or right).

DQN loss function

The DQN requires a loss function for it to improve and get a higher score. This function can be mathematically represented as follows:

$$\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right)$$

Target Network Q-Network

It is the Q-network that chooses which actions to take. The target network is what is used as an approximation for the ground truth. If we consider a scenario where the Q-network predicted that the correct action in a particular state was to move left with 80% certainty and the target network advises to move left, we could tweak the parameters of the Q-network using backpropagation to make it more likely to predict "move left" in that state. In other words, we backpropagate the loss through the DQN and adjust the weights of the Q-network to reduce the overall loss. The loss equation aims to make the probabilities of the move to be nearer 100% certainty of the correct choice.

Experience replay

An experience consists of the current state, the action, the reward, and the next state. Each experience the agents get is recorded in the experience replay memory. An experience from the replay memory is sampled at random to train the network.

Experience replay has some key advantages when compared to traditional Q-learning. One such advantage is that, as each experience has the potential to be used to train the DQN's neural network multiple times, there is greater data efficiency. Another advantage is that, when it is learning from experiences as soon as they are obtained, it is the current parameters that determine the next sample that the parameters are trained on. If we consider this in the maze example, if the next best action is to move left, then the training samples will be dominated by those from the left-hand side of the screen. Such behavior can cause the DQN to get stuck in local minima. With the incorporation of experience replay, the experiences used to train the DQN originate from many different points in time, smoothing out the learning and helping to avoid poor performance.

Coded example – DQN

In this example, we will again consider the `CartPole-v0` environment from OpenAI Gym.

First, we create a class that will permit us to incorporate experience replay when training the DQN. This essentially stores the transitions observed by the agent. The transitions that build up a batch are decorrelated by the sampling process:

```
transition_type = namedtuple('transition_type',
                             ('state', 'action', 'next_state', 'reward'))

class ExperienceReplayMemory(object):
    def __init__(self, model_capacity):
        self.model_capacity = model_capacity
        self.environment_memory = []
        self.pole_position = 0

    def push(self, *args):
        """Saves a transition."""
        if len(self.environment_memory) < self.model_capacity:
            self.environment_memory.append(None)
            self.environment_memory[self.pole_position] =
transition_type(*args)
            self.pole_position = (self.pole_position + 1) %
self.model_capacity

    def sample(self, batch_size):
        return random.sample(self.environment_memory, batch_size)

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

Define the ConvNet model whereby the difference between the current and previous screen patches are fed into it. The model has two outputs— $Q(s, \text{left})$ and $Q(s, \text{right})$. The network is trying to predict the expected reward/return of taking an action given the current input:

```
class DQNAlgorithm(nn.Module):

    def __init__(self, h, w, outputs):
        super(DQNAlgorithm, self).__init__()
        self.conv_layer1 = nn.Conv2d(3, 8, kernel_size=5, stride=2)
        self.batch_norm1 = nn.BatchNorm2d(8)
        self.conv_layer2 = nn.Conv2d(8, 32, kernel_size=5, stride=2)
```

```

        self.batch_norm2 = nn.BatchNorm2d(32)
        self.conv_layer3 = nn.Conv2d(32, 32, kernel_size=5, stride=2)
        self.batch_norm3 = nn.BatchNorm2d(32)

        # The number of linear input connections depends on the number of
conv2d layers
        def conv2d_layer_size_out(size, kernel_size = 5, stride = 2):
            return (size - (kernel_size - 1) - 1) // stride + 1
        convw =
conv2d_layer_size_out(conv2d_layer_size_out(conv2d_layer_size_out(w)))
        convh =
conv2d_layer_size_out(conv2d_layer_size_out(conv2d_layer_size_out(h)))
        linear_input_size = convw * convh * 32
        self.head = nn.Linear(linear_input_size, outputs)

        # Determines next action during optimisation
    def forward(self, x):
        x = F.relu(self.batch_norm1(self.conv_layer1(x)))
        x = F.relu(self.batch_norm2(self.conv_layer2(x)))
        x = F.relu(self.batch_norm3(self.conv_layer3(x)))
        return self.head(x.view(x.size(0), -1))

```

Set the hyperparameters of the model along with some utilities for training:

```

BATCH_SIZE = 128
GAMMA_VALUE = 0.95
EPISODE_START = 0.9
EPISODE_END = 0.05
EPISODE_DECAY = 200
TARGET_UPDATE = 20

init_screen = get_screen()
dummy_1, dummy_2, height_screen, width_screen = init_screen.shape

number_actions = environment.action_space.n

policy_network = DQNAgorithm(height_screen, width_screen,
number_actions).to(device)
target_network = DQNAgorithm(height_screen, width_screen,
number_actions).to(device)
target_network.load_state_dict(policy_network.state_dict())
target_network.eval()

optimizer = optim.RMSprop(policy_network.parameters())
memory = ExperienceReplayMemory(1000)

steps_done = 0

def choose_action(state):
    global steps_done
    sample = random.random()
    episode_threshold = EPISODE_END + (EPISODE_START - EPISODE_END) * \
        math.exp(-1. * steps_done / EPISODE_DECAY)
    steps_done += 1
    if sample > episode_threshold:

```

```

        with torch.no_grad():
            return policy_network(state).max(1)[1].view(1, 1)
    else:
        return torch.tensor([[random.randrange(number_actions)]], device=device, dtype=torch.long)

durations_per_episode = []

def plot_durations():
    plt.figure(2)
    plt.clf()
    durations_timestep = torch.tensor(durations_per_episode, dtype=torch.float)
    plt.title('Training in progress...')
    plt.xlabel('Episode')
    plt.ylabel('Duration')
    plt.plot(durations_timestep.numpy())
    if len(durations_timestep) >= 50:
        mean_values = durations_per_episode.unfold(0, 100, 1).mean(1).view(-1)
        mean_values = torch.cat((torch.zeros(99), mean_values))
        plt.plot(mean_values.numpy())

    plt.pause(0.001)
    plt.show()

```

Finally, we have the code for training our model. This function performs a single step of the optimization. First, it samples a batch and concatenates all of the tensors into a single one. It computes $Q(st,at)$ and $V(st+1)=\max_a Q(st+1,a)$ and combines them into a loss. By definition, we set $V(s)=0$ if s is a terminal state. We also use a target network to compute $V(st+1)$ for added stability:

```

def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions_memory = memory.sample(BATCH_SIZE)
    batch = Transition(*zip(*transitions_memory))

```

Compute a mask of non-final states. After this, we concatenate the batch elements:

```

not_final_mask = torch.tensor(tuple(map(lambda x: x is not None,
                                         batch.next_state)), device=device, dtype=torch.uint8)
not_final_next_states = torch.cat([x for x in batch.next_state if x is not None])

state_b = torch.cat(batch.state)
action_b = torch.cat(batch.action)
reward_b = torch.cat(batch.reward)

```

Compute $Q(s_t, a)$, then select the columns of actions taken:

```
state_action_values = policy_network(state_b).gather(1, action_b)

next_state_values = torch.zeros(BATCH_SIZE, device=device)
next_state_values[not_final_mask] =
target_net(not_final_next_states).max(1)[0].detach()
```

We compute the expected Q values:

```
expected_state_action_values = (next_state_values * GAMMA_VALUE) +
reward_b
```

We then compute the Huber loss function:

```
hb_loss = F.smooth_l1_loss(state_action_values,
expected_state_action_values.unsqueeze(1))
```

Now, we optimize the model:

```
optimizer.zero_grad()
hb_loss.backward()
for param in policy_network.parameters():
    param.grad.data.clamp_(-1, 1)
optimizer.step()

number_episodes = 100
for i in range(number_episodes):
    environment.reset()
    last_screen = get_screen()
    current_screen = get_screen()
    current_state = current_screen - last_screen
    for t in count():
        # Here we both select and perform an action
        action = choose_action(current_state)
        _, reward, done, _ = environment.step(action.item())
        reward = torch.tensor([reward], device=device)
```

Now, we observe the new state:

```
last_screen = current_screen
current_screen = get_screen()
if not done:
    next_state = current_screen - last_screen
else:
    next_state = None
```

We store the transition in memory:

```

    memory.push(current_state, action, next_state, reward)

    # Move to the next state
    current_state = next_state

```

Let's perform one step of the optimization (on the target network):

```

optimize_model()
if done:
    durations_per_episode.append(t + 1)
    plot_durations()
    break

```

Update the target network; copy all weights and biases in the DQN:

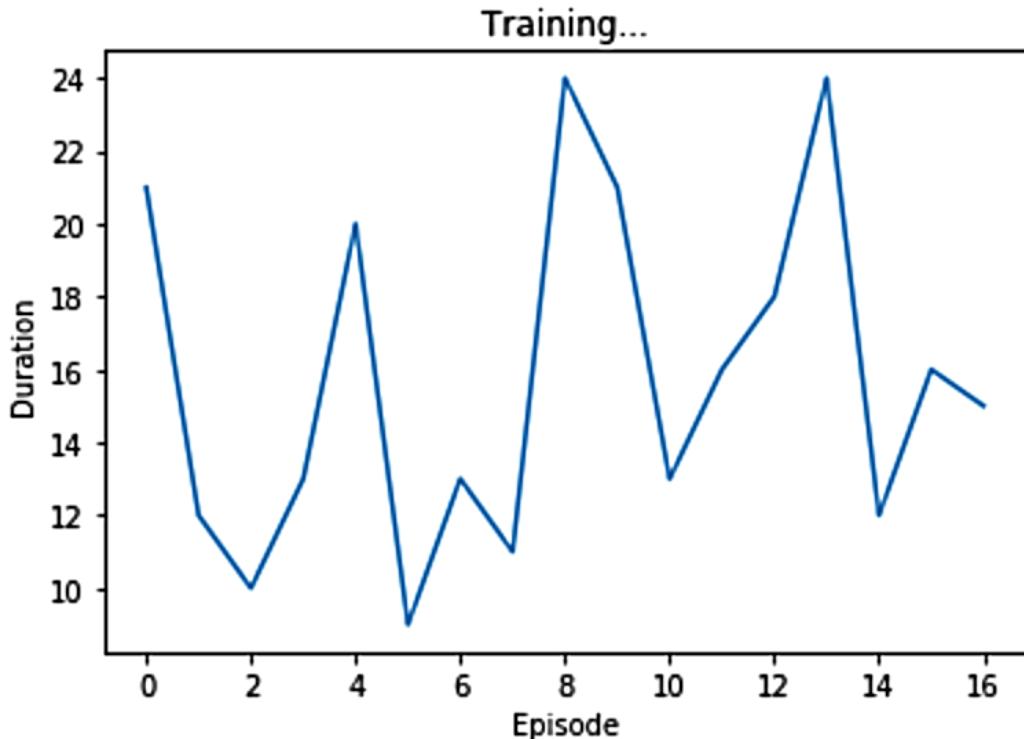
```

if i % TARGET_UPDATE == 0:
    target_network.load_state_dict(policy_network.state_dict())

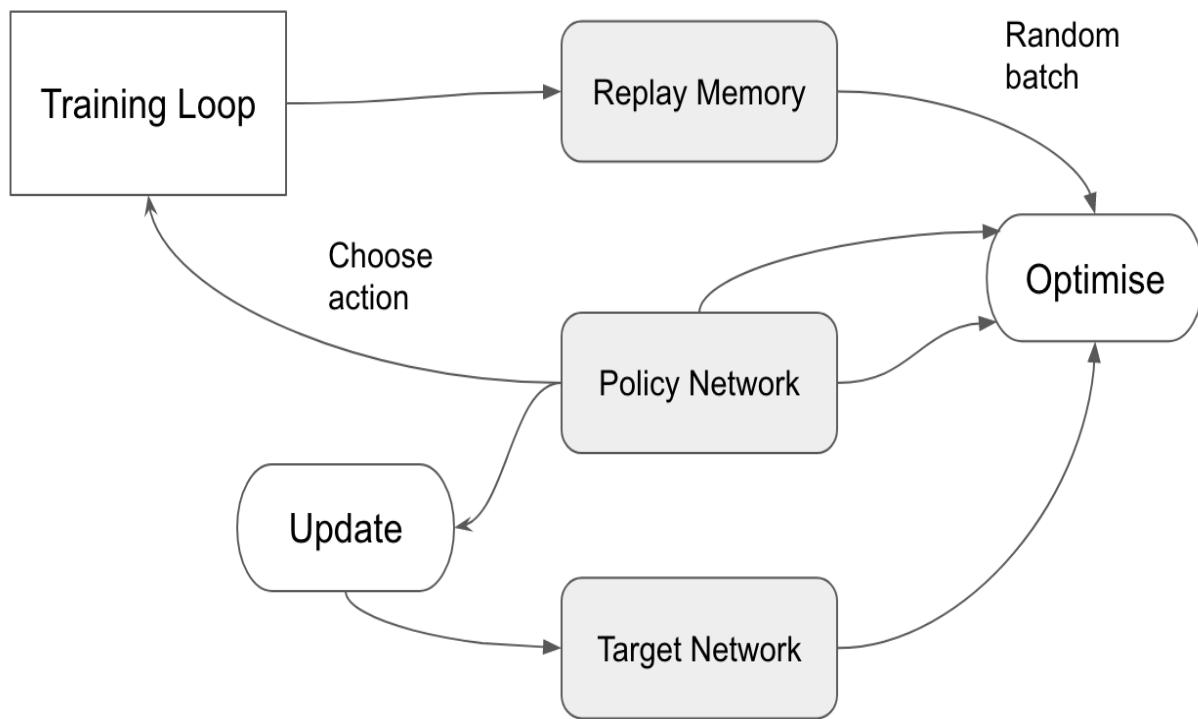
print('Complete')
environment.render()
environment.close()
plt.ioff()
plt.show()

```

This outputs some visualizations to give insight into how the model performs during training:



The following diagram summarizes what the model in this coded example is doing:



Double deep Q-learning

Double deep Q-learning generally leads to better performance of the AI agents when compared to vanilla DQNs. A common problem with deep Q-learning is that, sometimes, the agents can learn unrealistically high action values because it includes a maximization step over estimated action values. This tends to prefer overestimated to underestimated values. If overestimations are not uniform and not concentrated at states about which we wish to learn more, then these can negatively affect the quality of the resulting policy.

The idea of double Q-learning is to reduce these overestimations. It does this by decomposing the max operation in the target into action selection and action evaluation. In the vanilla DQN implementation, the action selection and action evaluation are coupled. It uses the target network to select the action and, at the same time, to estimate the quality of the action.

We are using the target-network to select the action and at the same time to estimate the quality of the action. Double Q-learning essentially tries to decouple both procedures from each other.

In double Q-learning, the temporal difference (TD) target looks as follows:

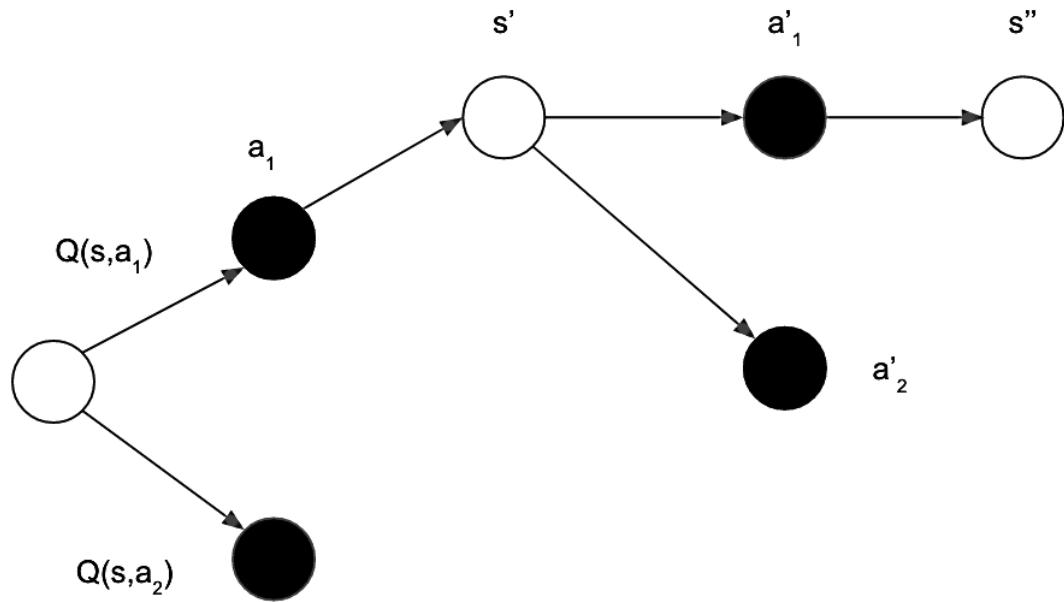
$$y_i^{\text{DoubleQ}} = \mathbb{E}_{a' \sim \mu} \left[r + \gamma \max_a Q(s', a; \theta_i) | S_t = s, A_t = a \right]$$

The calculation of new TD target can be summarized in the following steps:

1. Q-Network uses the next state, s' , to calculate qualities, $Q(s', a)$, for each possible action, a , in the state, s' .
2. The $\arg\max$ operation applied on $Q(s', a)$ chooses the action, a^* , that belongs to the highest quality (action selection).
3. The quality $Q(s', a^*)$, that belongs to the action, a^* , is selected for the calculation of the target.

The process of double Q-learning can be visualized as per the following diagram. An AI agent is at the initial in state s ; it knows, based on some

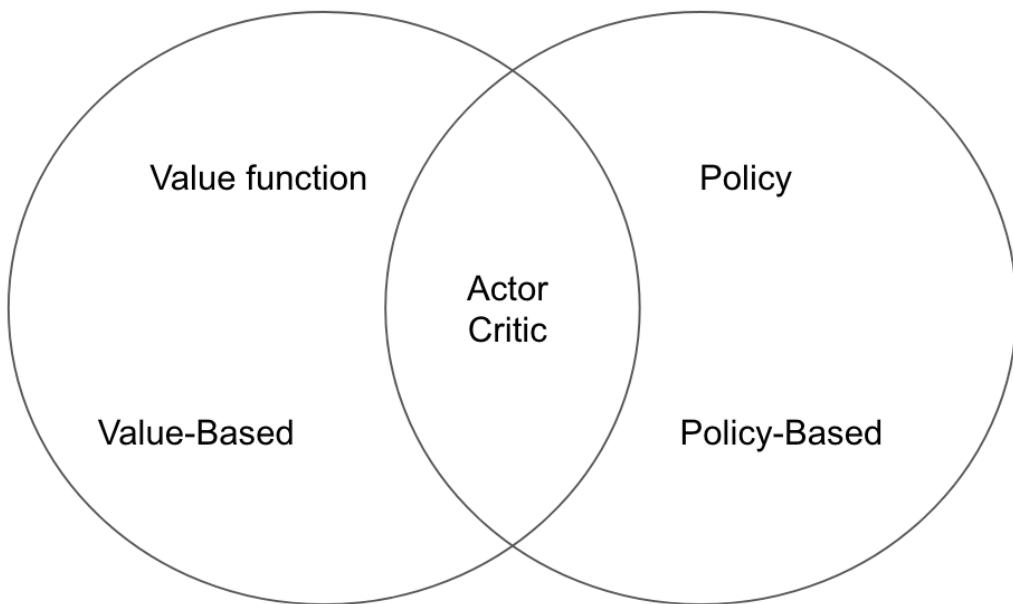
previous calculations, the qualities $Q(s, a_1)$ and $Q(s, a_2)$ for possible two actions in that state. The agent then decides to take action a_1 and ends up in state s' :



$$Q(s, a_1) = r + \gamma Q(s', a'_1; \theta_{i-1})$$

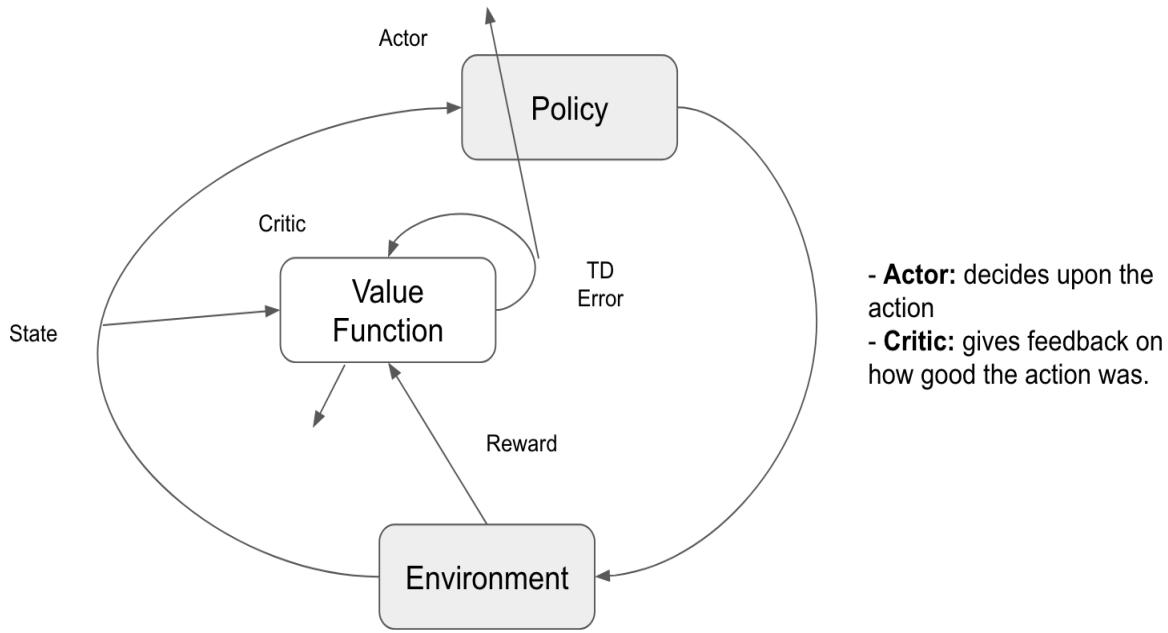
Actor-critic methods

Actor-critic methods aim to incorporate the advantages of both value- and policy-based methods while eliminating their drawbacks:



The fundamental idea behind actor-critics is to split the model into two parts: one for computing an action based on a state and another to produce the Q-values of the action.

The actor is a neural network that takes the state as input and outputs the best action. By learning the optimal policy, it controls how the agent behaves. The critic evaluates the action by computing the value function. In other words, the actor tries to optimize the policy and the critic tries to optimize the value. The two models improve over time at their individual roles and, as such, the overall architecture learns more efficiently than the two methods separately:



The two models are essentially competing with one another. Such an approach is becoming increasingly popular in the field of machine learning; for example, this is also present in generative adversarial networks.

The nature of the actor's role is to be exploratory. It frequently tries new things and explores the environment. The role of the critic is to either criticize or compliment the actions of the actor. The actor takes this feedback on board and adjusts its behavior accordingly. As the actor receives more and more feedback, it becomes better at deciding which actions to take.

Like a neural network, the actor can be a function approximator where its task is to produce the best action for a given state. This could be a fully connected or convolutional neural network, for example. The critic is also a function approximator, which receives the environment as input along with the action by the actor. It concatenates these inputs and outputs the action value (Q-value).

The two networks are trained separately and, to update their weights, they use gradient ascent as opposed to descent as it aims to determine the global maximum rather than minimum. Weights are updated at each step rather than at the end of an episode as opposed to policy gradients.

Actor-critics have been proven to learn complex environments and have been used in many 2D and 3D computer games such as *Super Mario* and *Doom*.

Coded example – actor-critic model

Here, we will consider a coded implementation example in PyTorch. First, we define the `ActorCritic` class:

```
HistoricalAction = namedtuple('HistoricalAction', ['log_prob', 'value'])

class ActorCritic(nn.Module):
    def __init__(self):
        super(ActorCritic, self).__init__()
        self.linear = nn.Linear(4, 128)
        self.head_action = nn.Linear(128, 2)
        self.head_value = nn.Linear(128, 1)

        self.historical_actions = []
        self.rewards = []

    def forward(self, x):
        x = F.relu(self.linear(x))
        scores_actions = self.head_action(x)
        state_values = self.head_value(x)
        return F.softmax(scores_actions, dim=-1), state_values
```

Now, we initialize the model:

```
ac_model = ActorCritic()
optimizer = optim.Adam(ac_model.parameters(), lr=3e-2)
eps = np.finfo(np.float32).eps.item()
```

Define a function that will choose the best action based on the state:

```
def choose_action(current_state):
    current_state = torch.from_numpy(current_state).float()
    probabilities, state_value = ac_model(current_state)
    m = Categorical(probabilities)
    action = m.sample()

    ac_model.historical_actions.append(HistoricalAction(m.log_prob(action),
    state_value))
    return action.item()
```

From here, we need to define the function that calculates the total returns and considers the loss function:

```

def end_episode():
    R = 0
    historical_actions = ac_model.historical_actions
    losses_policy = []
    losses_value = []
    returns = []
    for r in ac_model.rewards[::-1]:
        R = r + gamma * R
        returns.insert(0, R)
    returns = torch.tensor(returns)
    returns = (returns - returns.mean()) / (returns.std() + eps)
    for (log_prob, value), R in zip(historical_actions, returns):
        advantage = R - value.item()
        losses_policy.append(-log_prob * advantage)
        losses_value.append(F.smooth_l1_loss(value, torch.tensor([R])))
    optimizer.zero_grad()
    loss = torch.stack(losses_policy).sum() +
    torch.stack(losses_value).sum()
    loss.backward()
    optimizer.step()
    del ac_model.rewards[:]
    del ac_model.historical_actions[:]

```

Finally, we can train the model and review how it performs:

```

running_reward = 10
for i_episode in count(1):
    current_state, ep_reward = environment.reset(), 0
    for t in range(1, 10000):
        action = choose_action(current_state)
        current_state, reward, done, _ = environment.step(action)
        if render:
            environment.render()
        ac_model.rewards.append(reward)
        ep_reward += reward
        if done:
            break

    running_reward = 0.05 * ep_reward + (1 - 0.05) * running_reward
    end_episode()
    if i_episode % log_interval == 0:
        print('Episode number {} \tLast reward: {:.2f} \tAverage reward: {:.2f}'.format(
            i_episode, ep_reward, running_reward))
        if running_reward > environment.spec.reward_threshold:
            print("Solved! Running reward is {} and "
                  "the last episode runs to {} time"
                  "steps!".format(running_reward, t))
            break

```

This gives the following output:

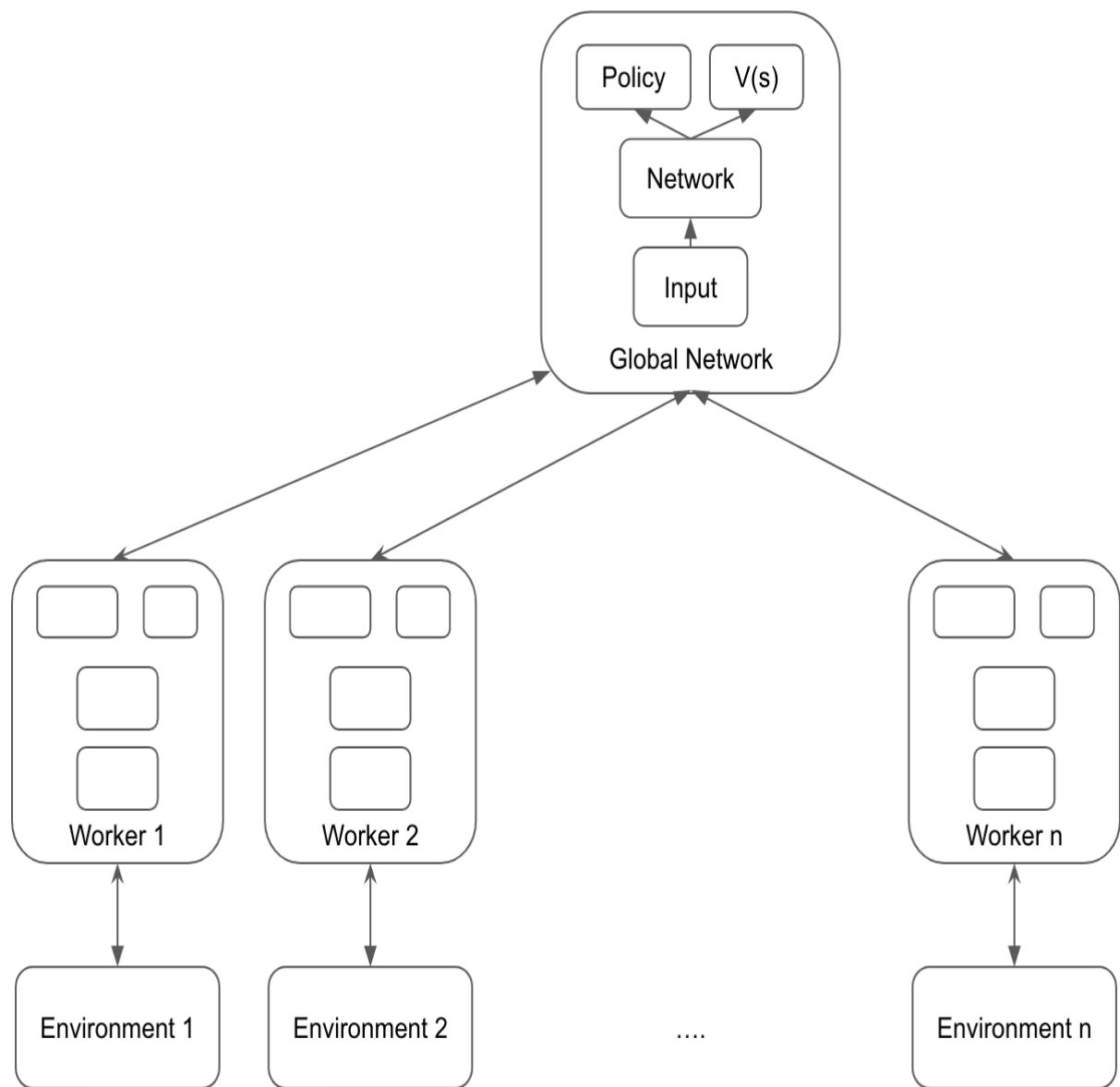
Episode number 10	Last reward: 24.00	Average reward: 21.77
Episode number 20	Last reward: 200.00	Average reward: 88.68
Episode number 30	Last reward: 100.00	Average reward: 101.79
Episode number 40	Last reward: 134.00	Average reward: 115.30
Episode number 50	Last reward: 170.00	Average reward: 131.04
Episode number 60	Last reward: 140.00	Average reward: 130.99
Episode number 70	Last reward: 200.00	Average reward: 145.82
Episode number 80	Last reward: 173.00	Average reward: 162.88
Episode number 90	Last reward: 141.00	Average reward: 169.09
Episode number 100	Last reward: 174.00	Average reward: 164.69
Episode number 110	Last reward: 200.00	Average reward: 176.47
Episode number 120	Last reward: 183.00	Average reward: 185.06
Episode number 130	Last reward: 200.00	Average reward: 182.33
Episode number 140	Last reward: 187.00	Average reward: 187.80
Episode number 150	Last reward: 191.00	Average reward: 186.57
Episode number 160	Last reward: 192.00	Average reward: 181.48
Episode number 170	Last reward: 189.00	Average reward: 178.21
Episode number 180	Last reward: 200.00	Average reward: 185.34
Episode number 190	Last reward: 200.00	Average reward: 191.22
Episode number 200	Last reward: 200.00	Average reward: 194.74

Solved! Running reward is 195.00753109917716 and the last episode runs to 200 time steps!

Asynchronous actor-critic algorithm

Asynchronous Advantage Actor-Critic or A3C is an algorithm proposed by Google's DeepMind. The algorithm has been proven to outperform other algorithms.

In A3C, there are multiple instances of agents, each of which has been initialized differently in their own separate environments. Each individual agent begins to take actions and go through the reinforcement learning process to gather their own unique experiences. These unique experiences are then used to update the global neural network. This global neural network is shared by all of the agents and it influences all of the actions of the agents, and every new experience from each agent improves the overall network speed:



The **Advantage** term in the name is the value that states whether or not there is an improvement in an action compared to the expected average value of that state based on. The advantage formula is as follows:

$$A(s,a) = Q(s,a) - V(s)$$

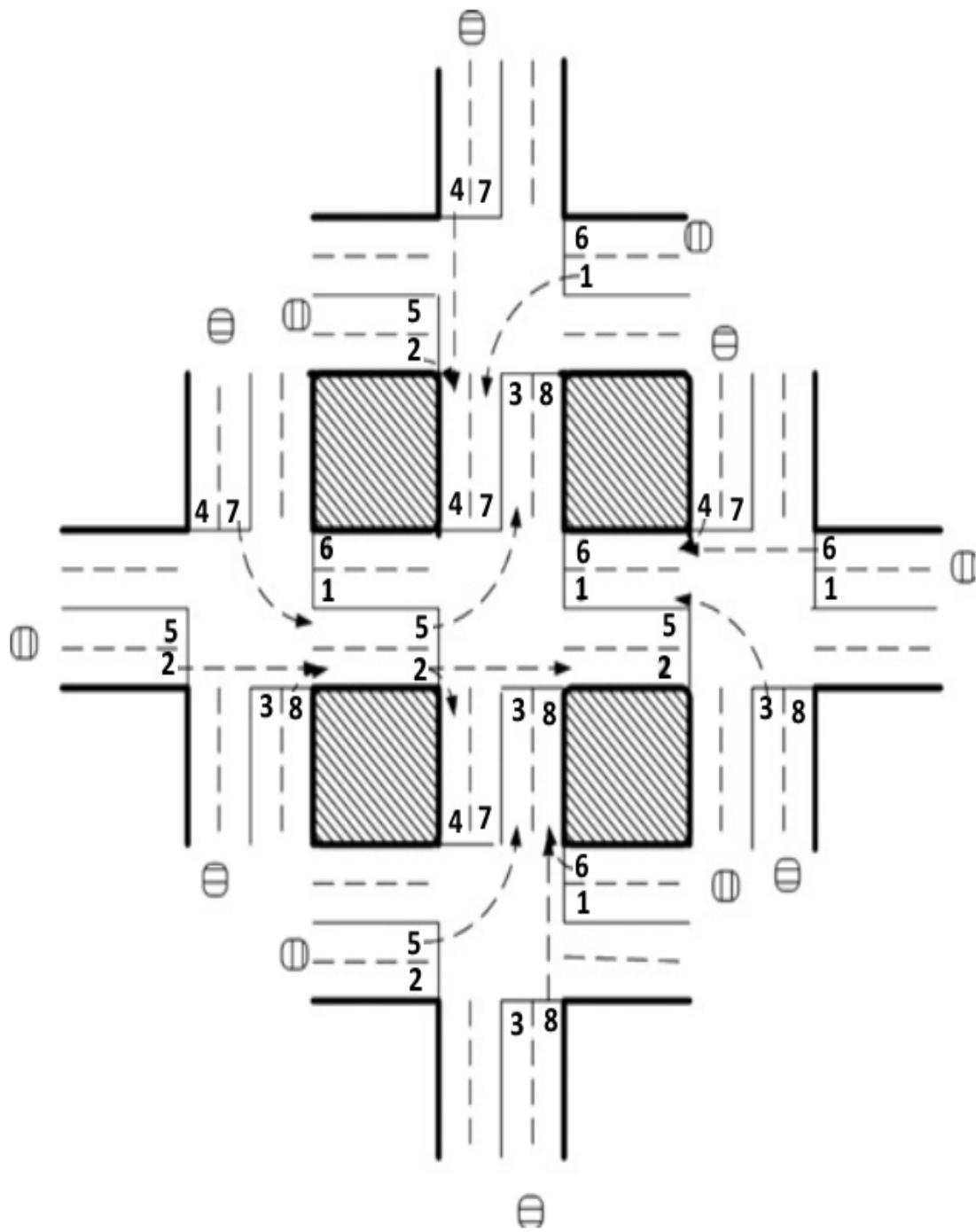
Practical applications

RL methods have been applied to solve problems in a multitude of areas in the real world. Here, we consider some examples of these:

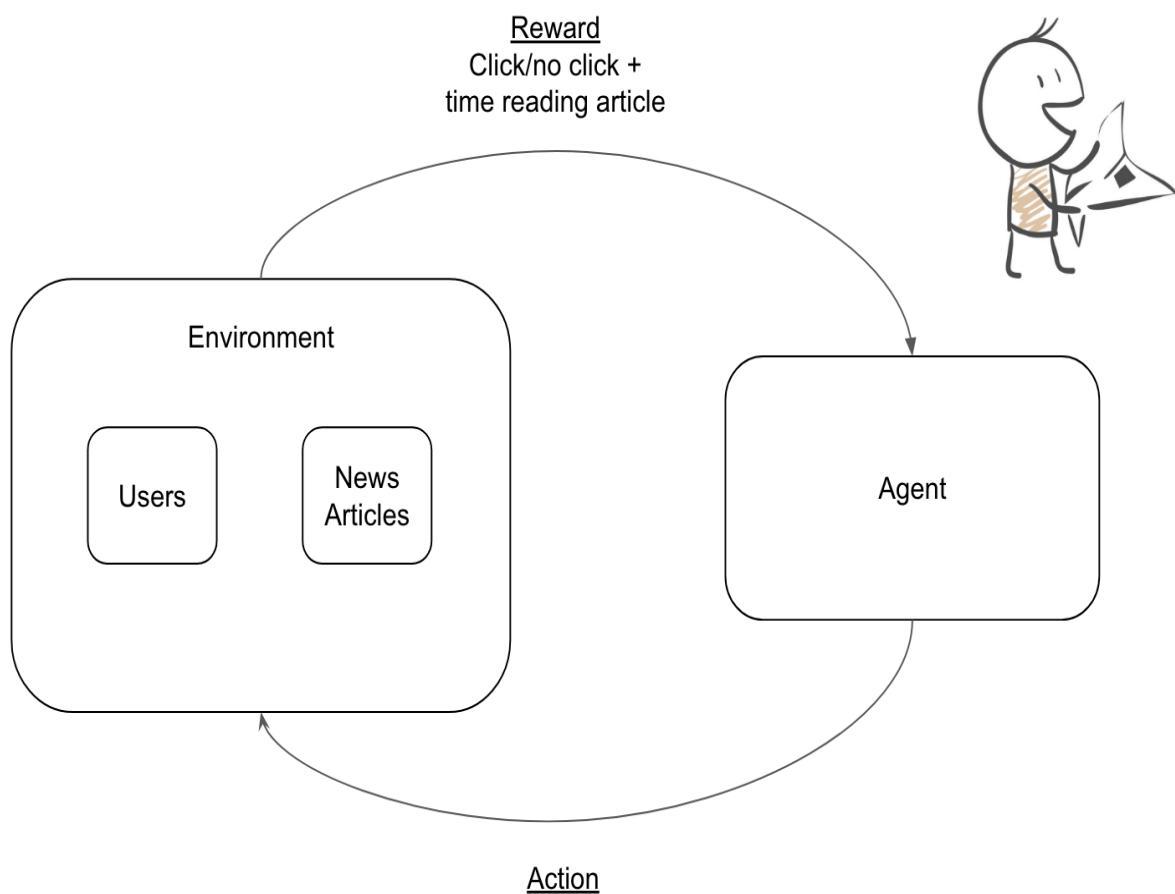
- **Robotics:** There has been a significant amount of work on applying RL in the field of robotics. In the present day, manufacturing facilities are full of robots performing a variety of tasks, the foundations of which are RL methods:



- **Traffic light control:** In the paper *Reinforcement learning-based multi-agent system for network traffic signal control*, researchers designed a traffic light controller to solve congestion problems that demonstrated superior results to other methods:



- **Personalized recommendations:** RL has been applied in news recommendation systems to account for the fact that news changes rapidly and, as users tend to have a short attention span, the click-through rate alone cannot reflect the retention rate of the users:



- **Generating images:** There has been a lot of research into combining RL with other deep learning architectures for a host of different applications. Many of these have shown some impressive results. DeepMind showed that using generative models and RL, they were able to successfully generate images:

INPUT RECONSTRUCTION
 64×64 64×64



Summary

In this chapter, we began by covering the basics of RL and introduced more advanced algorithms that have proven an ability to outperform humans in real-world scenarios. We also gave examples as to how these can be implemented in PyTorch.

In the next and final chapter, there will be an overview of this book, along with some tips on how you can keep yourself up to date with recent advancements in the data science space.

Further reading

Refer to the following links for more information:

- *A Brief Survey of Deep Reinforcement Learning*: <https://arxiv.org/pdf/1708.05866.pdf>
- *Playing Atari with Deep Reinforcement Learning*: <https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf>
- *Deep Reinforcement Learning with Double Q-learning*: <https://arxiv.org/pdf/1509.06461.pdf>
- *Continuous Control with Deep Reinforcement Learning*: <https://arxiv.org/pdf/1509.02971.pdf>
- *Asynchronous Methods for Deep Reinforcement Learning*: <https://arxiv.org/pdf/1602.01783.pdf>
- *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*: <https://arxiv.org/pdf/1801.01290.pdf>
- *Reinforcement learning-based multi-agent system for network traffic signal control*: http://web.eecs.utk.edu/~itamar/Papers/IET_ITS_2010.pdf
- *End-to-End Training of Deep Visuomotor Policies*: <https://arxiv.org/pdf/1504.00702.pdf>
- *DRN: A Deep Reinforcement Learning Framework for News Recommendation*: http://www.personal.psu.edu/~gjz5038/paper/www2018_reinforceRec/www2018_reinforceRec.pdf
- *Synthesizing Programs for Images using Reinforced Adversarial Learning*: <https://arxiv.org/pdf/1804.01118.pdf>

Whats Next?

You've finally finished the book! Give yourself a pat on the back! Thanks for going through the book, and I sincerely hope that it helps you in your path ahead, be it as a data scientist, a machine learning engineer, or one of the many other evolving job titles in the AI space. By now, you should have a firm grasp of the PyTorch API and how to use it to perform several tasks across computer vision, natural language processing, and reinforcement learning. However, this is by no means an end to the journey you've started, but rather a beginning to a fantastic road ahead!

What's next?

In this chapter, we will look at the next sequence of logical steps to take in order to build on the progress you've made with this book. However, let's first take a step back and look at all the tools and techniques we've learned so far and understand how all of what you've learned so far fits into a deeper learning context.

Overview of the book

The following bullet list will help you understand the essence of each chapter from the book and can act as a quick guide to put what we've learned throughout the book into context:

- History of AI, neural networks, and deep learning. The various deep learning frameworks in use. The importance and necessity of PyTorch. Improvements in PyTorch v1.0. GPU versus CPU. Using CUDA to parallelize tensor computations.
- The building blocks of neural networks: How do networks learn representations? We looked at PyTorch `tensors`, `tensor operations`, `nn.module`, `torch optim`, and how the PyTorch `define by run` dynamic DAGs work.
- We learned about the different processes involved in training a neural network, such as the PyTorch dataset for data preparation, data loaders for batching tensors, the `torch.nn` package for creating network architectures, and using PyTorch loss functions and optimizers. We also went through different techniques to handle overfitting, such as dropout, L1 and L2 regularization, and using batch normalization.
- We learned the different building blocks of **convolution neural networks (CNNs)**, and also learned about transfer learning, which helps us to use a pretrained model. We also saw techniques, such as using preconvoluted features, which help in reducing the time taken to train the models.
- We learned about word embedding and how to use it for text classification problems. We also explored how we can use pretrained word embedding. We explored **recurrent neural network (RNN)**, its variants such as **long short-term memory (LSTM)**, and how to use them for text classification problems.
- In this chapter, we examined the idea of semi-supervised learning using autoencoders to denoise data, and variational autoencoders to generate new images.

- We explored generative models and learned how PyTorch can be used for creating an artistic style transfer, and for creating new CIFAR images using a **generative adversarial network (GAN)**. We also explored language modeling techniques that can be used to generate new text or to create domain-specific embedding.
- We explored modern architectures, such as ResNet, Inception, DenseNet and encode-decoder architecture. We also saw how these models can be used for transfer learning. We also built an ensemble model by combining all these models.
- Finally, we looked at how reinforcement learning can be used to train models to make decisions in uncertain sequential environments. We looked at the various deep reinforcement learning strategies, such as deep-Q learning and policy based and actor-critic models. We used an OpenAI gym environment to solve the famous cart-pole problem using deep reinforcement learning.

Reading and implementing research papers

Deep learning is a constantly evolving field and keeping up to date with the latest developments in the field will definitely impact your ability to contribute to the team you are working in and also to the field in general.

Research papers at first might look like jargon-filled convoluted text that is hard to understand, but making a constant effort to read and implement these algorithms will significantly boost your capabilities. One of my favorite repositories of papers and their corresponding implementations in code is [paperswithcode](https://paperswithcode.com/sota) (<https://paperswithcode.com/sota>):



Search for papers, code and tasks



Browse state-of-the-art

Follow

Discuss

About

Log In/Register

Browse state-of-the-art

758 leaderboards • 1065 tasks • 862 datasets • 11134 papers with code

Follow on Twitter for updates

Computer Vision



Semantic
Segmentation

15 leaderboards

389 papers with code



Image
Classification

36 leaderboards

317 papers with code



Object
Detection

25 leaderboards

278 papers with code



Image
Generation

23 leaderboards

136 papers with code



Pose
Estimation

26 leaderboards

132 papers with code

▶ See all 590 tasks

Natural Language Processing



Machine
Translation

26 leaderboards

360 papers with code



Language
Modelling

9 leaderboards

274 papers with code



Question
Answering

31 leaderboards

267 papers with code



Sentiment
Analysis

16 leaderboards

241 papers with code



Text
Classification

25 leaderboards

112 papers with code

▶ See all 204 tasks

The paperswithcode website

You should find the latest research papers and code for various tasks in AI such as CV, NLP, reinforcement learning, speech, and so on. Going through at least one paper per week and implementing the

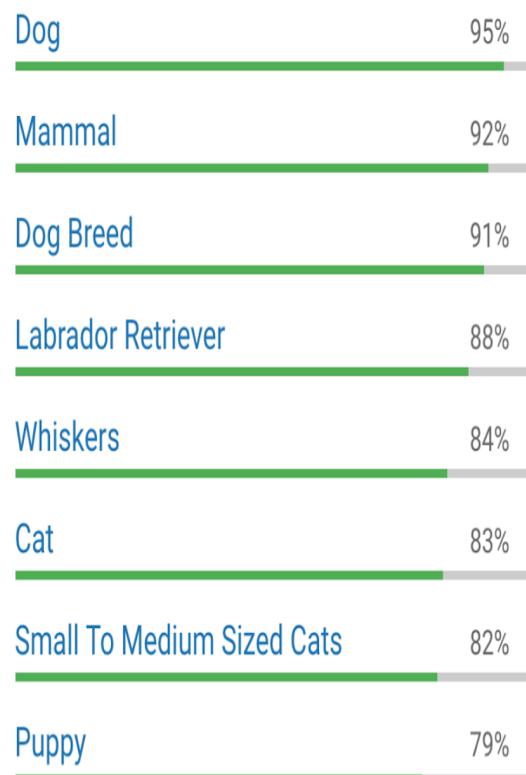
code by downloading the source code will help you stay on track with the latest developments in the field.

Interesting ideas to explore

Most of the concepts that we learned in the book form the foundation of modern applications that are powered by deep learning. In this section, we will look at the different interesting projects that we can do that are related to computer vision and **natural language processing (NLP)**.

Object detection

All the examples we have seen in this book help you in detecting whether a given image is this (cat) or that (dog). But, to solve some of the problems in the real world, you may need to identify different objects in an image, such as the ones shown here:



Output of an object detection algorithm

This image shows the output of an object detection algorithm where the algorithm is detecting objects such as a beautiful dog and cat. Just as there are off-the-shelf algorithms for image classification, there are a bunch of amazing algorithms that can help in building object recognition systems. Here is a list of some of the important algorithms and the papers on object detection:

- **Single shot multibox detector (SSD):** <https://arxiv.org/abs/1512.02325>
- **Faster RCNN:** <https://arxiv.org/abs/1506.01497>
- **YOLO2:** <https://arxiv.org/abs/1612.08242>

Image segmentation

Let's assume you are reading this book from the terrace of a building. What do you see around you? Can you draw an outline of what you see? If you are a good artist, unlike me, then you would have probably drawn a couple of buildings, trees, birds, and a few more interesting things surrounding you. Image segmentation algorithms try to capture something similar. Given an image, they generate a prediction for each pixel, identifying which class each pixel belongs to. The following image shows what image segmentation algorithms identify:



Output of an image segmentation algorithm

Some of the important algorithms that you may want to explore for image segmentation are given here:

- **R-CNN:** <https://arxiv.org/abs/1311.2524>
- **Fast R-CNN:** <https://arxiv.org/abs/1504.08083>
- **Faster R-CNN:** <https://arxiv.org/abs/1506.01497>
- **Mask R-CNN:** <https://arxiv.org/abs/1703.06870>

OpenNMT in PyTorch

The **Open Source Neural Machine Translation (OpenNMT)** ([http://github.com/OpenNMT/OpenNMT-py](https://github.com/OpenNMT/OpenNMT-py)) project helps in building a lot of applications that are powered by the encoder-decoder architecture. Some of the applications that you can build are translation systems, text summarization, and image-to-text.

Allen NLP

Allen NLP is an open source project built on PyTorch that enables us to do many NLP tasks much more easily. There is a demo page (<http://demo.allennlp.org/machinecomprehension>) that you should look at to understand what you can build using Allen NLP.

fast.ai – making neural nets uncool again

One of my favorite places to learn about deep learning, and a great place of inspiration, is a MOOC, with the sole motive of making deep learning accessible to all, organized by two amazing mentors from fast.ai (<http://www.fast.ai/>), Jeremy Howard and Rachel Thomas. For a new version of their course, they built an incredible framework ([http://github.com/fastai/fastai](https://github.com/fastai/fastai)) on top of PyTorch, making it much easier and quicker to build applications. If you have not already started their course, I would strongly recommend you start it. Exploring how the fast.ai framework is built will give you great insight into many powerful techniques.

Open neural network exchange

Open neural network exchange (ONNX) (<http://onnx.ai/>) is the first step towards an open ecosystem that empowers you to choose the right tools as the project evolves. ONNX provides an open source format for deep learning models. It defines an extensible computation graph model as well as definitions of built-in operators and standard data types. Caffe2, PyTorch, Microsoft Cognitive Toolkit, Apache MXNet, and other tools are developing ONNX support. This project can help in product ionizing PyTorch models.

How to keep yourself updated

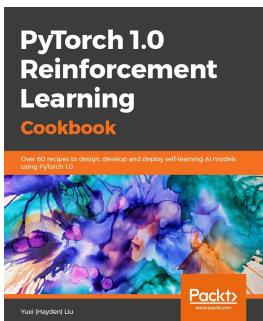
Social media platforms, particularly Twitter, help you to stay updated in the field. There are many people you can follow. If you are unsure of where to start, I would recommend following Jeremy Howard ([http://twitter.com/jeremyphoward](https://twitter.com/jeremyphoward)) and any interesting people he may follow. By doing this, you will be forcing the Twitter recommendation system to work for you. Another important Twitter account you need to follow is PyTorch (<https://twitter.com/PyTorch>). The amazing people behind PyTorch have some great content being shared. If you are looking for research papers, then look at <http://www.arxiv-sanity.com/>, where many smart researchers publish their papers. More great resources for learning about PyTorch are its tutorials (<http://pytorch.org/tutorials/>), its source code (<https://github.com/pytorch/pytorch>), and its documentation (<http://pytorch.org/docs/0.3.0/>).

Summary

There is much more to deep learning and PyTorch. PyTorch is a relatively new framework, which, at the time of writing this chapter, is 3 years old. There is much more to learn and explore, so happy learning. All the best.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

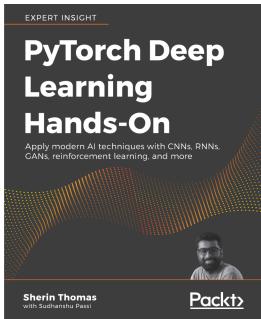


PyTorch 1.x Reinforcement Learning Cookbook

Yuxi (Hayden) Liu

ISBN: 978-1-83855-196-4

- Use Q-learning and the state–action–reward–state–action (SARSA) algorithm to solve various Gridworld problems
- Develop a multi-armed bandit algorithm to optimize display advertising
- Scale up learning and control processes using Deep Q-Networks
- Simulate Markov Decision Processes, OpenAI Gym environments, and other common control problems
- Select and build RL models, evaluate their performance, and optimize and deploy them
- Use policy gradient methods to solve continuous RL problems



PyTorch Deep Learning Hands-On

Sherin Thomas, Sudhanshu Passi

ISBN: 978-1-78883-413-1

- Use PyTorch to build:
- Simple Neural Networks – build neural networks the PyTorch way, with high-level functions, optimizers, and more
- Convolutional Neural Networks – create advanced computer vision systems
- Recurrent Neural Networks – work with sequential data such as natural language and audio
- Generative Adversarial Networks – create new content with models including SimpleGAN and CycleGAN
- Reinforcement Learning – develop systems that can solve complex problems such as driving or game playing
- Deep Learning workflows – move effectively from ideation to production with proper deep learning workflow using PyTorch and its utility packages
- Production-ready models – package your models for high-performance production environments

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!