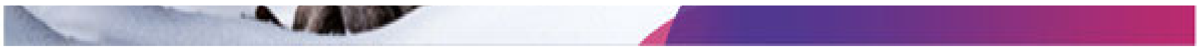
Deep Learning for Data Architects

Unleash the power of Python's deep learning algorithms

Shekhar Khandelwal

bpb



Deep Learning for Data Architects

Unleash the power of Python's deep learning algorithms

Shekhar Khandelwal

www.bpbonline.com

First published: 2024

Published by BPB Online

WeWork

119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55515-391

www.bpbonline.com

Dedicated to

My beloved wife, Niharika
&
My Daughter, Saesha

About the Author

Shekhar Khandelwal is a distinguished Senior AI & Data Scientist, residing in the bustling harbor city of Hamburg, Germany. His academic career shines bright with a Master's degree in Data Science, achieving distinction for his thesis work in the realm of Computer Vision. His name can be spotted in top-tier research papers and publications, predominantly in the area of Deep Learning.

Bringing to the table over 15 years of experience, the author has an extensive professional background in the field of AI and machine learning. His journey ranges from coding and crafting enterprise-level AI products to leading data teams and mentoring future data science professionals. He has successfully developed numerous client solutions utilizing big cloud service platforms such as AWS, Google Cloud, Microsoft Azure, and IBM Cloud.

Despite his deep involvement in the tech industry, our author is also a fitness enthusiast. When he's not making machines smarter, he's likely to be found flexing his muscles at the gym, attending a crossfit class, cycling, or participating in marathon runs. His zeal for fitness is as strong as his passion for AI, making him a well-rounded professional in all respects.

About the Reviewer

Abonia Sojasingarayar is a Machine Learning Scientist, Data Scientist, NLP Engineer, Computer Vision Engineer, AI Analyst, Technical Writer, and Technical Book Reviewer with over 6 years of experience. In my 6+ years of professional experience, I have dealt with ML domains (predictive analysis, computer vision, NLP), data analysis, data mining, data visualization, problem-solving, decision-making, planning, and software development. I am skilled in solving modern problems using AI and deep learning using familiar frameworks in Python such as TensorFlow, scikit-learn, Keras, etc., open-source toolkits such as openCV and NLTK, etc., and familiar with Web Marketing Strategies, front-end and back-end development tools, and languages. (For more info, don't hesitate to visit the project section)

She has experience in management consulting, commercial industry, cyber security, RH, chemical industry, banking and insurance, public transport industry, telecommunication services, e-commerce, and information technology sectors.

Her skills and expertise includes Studying and transforming data science prototypes, designing machine learning systems, Research and implement appropriate ML algorithms and tools, Develop machine learning applications according to requirements, Select appropriate datasets and data representation methods, Run machine learning tests and experiments, Perform statistical analysis and fine-tuning using test results, Train and retrain systems when necessary, Extend existing ML libraries and frameworks, Keep abreast of developments in the field, Execute analytical experiments to help solve various problems, making a true impact across various domains and industries, Identify relevant data sources and sets to mine for client business needs and collect large structured/unstructured datasets and variables, Devise and utilize algorithms and models to mine big data stores, perform data and error analysis to improve models, and clean and validate data for uniformity and accuracy, Analyze data for trends and patterns and interpret data with a clear objective in mind, Implement analytical models into production by collaborating with software developers, Communicate analytic solutions to stakeholders and implement improvements as needed to operational systems.

Acknowledgements

First and foremost, this book is dedicated to my uncle, Dinesh Khandelwal. His unwavering faith in me serves as an enduring source of inspiration. A renowned book-seller, his counter is the most honorable stage for my books, each sale symbolizing a cherished victory.

This work is also a tribute to my parents, the architects of my existence, whose love and nurturing have shaped me into the individual I am today.

Lastly, the dedication extends to the two sparkling gems of my life, my wife and daughter. Their smiles, a profound treasure, are the driving forces that fuel my passion.

Special gratitude is extended to BPB Publications for their mentorship and expertise in bringing this work to life. The journey to finalize this book was an enriching process, involving the valuable engagement and collaboration of reviewers, technical experts, and editors.

Acknowledgement is also due to my colleagues and coworkers in the tech industry. Over the years, their wisdom and feedback have been instrumental in my growth and the refinement of my work.

Lastly, my heartfelt thanks go out to every reader who has taken an interest in this book. Your support has been indispensable in transforming this dream into reality. Your enthusiasm fuels my commitment to share my knowledge.

Preface

The world as we know it is undergoing a profound transformation, a digital metamorphosis that is driven by the immense power of data and artificial intelligence. In the center of this revolutionary tide stands a key player - the data architect. Tasked with the responsibility to make sense of vast data oceans and convert it into meaningful insights, the role of a data architect is evolving at a staggering pace. “Deep Learning for Data Architects” is an embodiment of that evolution.

This book aims to illuminate the path for data architects and enthusiasts, guiding them through the labyrinth of modern data structures and artificial intelligence techniques, from the foundations of Python programming to the advanced landscapes of deep learning models. It is a testament to the compelling narrative of change that the industry is experiencing, catering to professionals who aspire to stay at the forefront of this digital transformation.

“Deep Learning for Data Architects” does not simply regurgitate theoretical concepts. Instead, it creates an engaging dialogue with the reader, providing practical Python implementations for complex AI paradigms, creating a bridge between theory and practice. Each chapter builds upon the last, starting with the basics and gradually delving into the deep end of AI and machine learning.

The journey is as important as the destination, and throughout the course of this book, you will encounter real-world data challenges, explore the depths of neural networks, understand the intricacies of convolutional, recurrent neural networks and unravel the mysteries of Generative Adversarial Networks and Transformers. Each step you take will empower you with new insights and skills, enabling you to tackle any challenge that the data landscape might throw your way.

Whether you are a seasoned data architect aiming to add deep learning to your arsenal, or a budding enthusiast stepping into the exciting intersection of these fields, this book is designed for you. As you turn the pages, you will find yourself not just learning, but evolving with the narrative of deep learning, setting the stage for a future-proof career in this dynamic domain.

Embrace this journey of learning and transformation, and let "Deep Learning for Data Architects" be your guide and companion in the exciting odyssey of AI and data science.

Chapter 1: Python for Data Science - serves as a solid foundation, providing an introduction to Python for data science. You will learn essential programming concepts, data structures, and libraries such as NumPy and Pandas. This chapter ensures that you are equipped with the necessary Python skills for the deep learning journey ahead.

Chapter 2: Real-World Challenges for Data Professionals in Converting Data Into Insights - dives into the challenges faced by data professionals when converting raw data into valuable insights. You will explore data cleaning, handling missing values, outlier detection, and feature engineering techniques. This chapter prepares you for the data preprocessing steps crucial for successful deep learning implementations.

Chapter 3: Build a Neural Network-Based Predictive Model - focuses on building predictive models using neural networks. You will learn about the architecture of a neural network, the role of activation functions, and techniques to handle classification and regression tasks. Through Python code implementations, you will gain hands-on experience in building and training neural networks.

Chapter 4: Convolutional Neural Networks - introduces CNNs, a powerful class of neural networks for image analysis tasks. You will understand the key components of CNNs, such as convolutional layers, pooling layers, and fully connected layers. The chapter provides Python code implementations to build CNN models for image classification tasks.

Chapter 5: Optical Character Recognition - explores the exciting field of OCR using deep learning. You will discover techniques to extract text from images, enabling automated text recognition. Through Python code implementations, you will learn how to build OCR models and apply them to real-world scenarios.

Chapter 6: Object Detection - focuses on object detection, an essential task in computer vision. You will explore popular object detection algorithms and architectures, such as Faster R-CNN and YOLO. Through Python code implementations, you will gain hands-on experience in training and deploying object detection models.

Chapter 7: Image Segmentation - delves into image segmentation, a technique used to partition images into meaningful regions. You will learn about popular segmentation algorithms, including U-Net and Mask R-CNN. Through Python code implementations, you will develop a deeper understanding of image segmentation and its applications.

Chapter 8: Recurrent Neural Networks - introduces RNNs, which are widely used for sequential data analysis. You will understand the architecture of RNNs, including LSTM and GRU units. Through Python code implementations, you will learn how to build RNN models for tasks such as natural language processing and time series forecasting.

Chapter 9: Generative Adversarial Networks - explores the fascinating world of GANs, which can generate new data instances based on training data. You will learn about the adversarial training process and different GAN architectures, including DCGAN and CycleGAN. Through Python code implementations, you will gain hands-on experience in generating realistic images and exploring generative modeling.

Chapter 10: Transformers - introduces Transformers, a revolutionary deep learning architecture that has gained prominence in natural language processing tasks. You will learn about the transformer architecture and its variants, such as BERT and GPT. Through Python code implementations, you will gain practical experience in applying transformers to text-related tasks.

Table of Contents

CHAPTER 1

Python for Data Science

You can't build a great building on a weak foundation. You must have a solid foundation if you are going to have a super-strong structure.

— Gordon B. Hinkley

Data is the most important component of data science. Python libraries for data science are built specifically to solve various peripheral issues that Data Scientists may face, like data sourcing, cleaning, pre-processing, and working with big data. It is important to know what these libraries have to offer and the sneaky tricks that can be implemented in our day-to-day data wrangling. The high-level methods within these libraries save you a lot of time performing humongous tasks, using just a few lines of code.

Additionally, data has to be sourced from all over the place. Hence, as someone who is building predictive models, maybe in academia or at work, should be familiar with different kinds of data that one can deal with and how to source them using Python.

Plus, when you work on your personal system, you have to be really cautious about how you utilize the available resources in terms of computing hardware and so on. Hence, it is of utmost importance to know various tricks that can help you with memory management. The tricks that usually go unnoticed and unexplored within Python packages, if explored, can significantly improve your efficiency while dealing with data.

Structure

In this chapter, we will cover the following topics:

Setting up the development environment

Advance Python libraries for data science

Reading and writing data to and from various file formats

Improving efficiency with the pandas read_csv method

Objectives

After studying this chapter, you should be able to set up your laptop with the required tools and technologies to embark on your journey of implementing various deep learning models that will be discussed in this book. Also, you should be able to install various Python packages that are prerequisites for building any predictive model. Once the development environment is set up, you will learn about various Python libraries that are available for scientific computing, machine learning and deep learning. Additionally, you will learn how to maximize your productivity in a limited hardware environment, since machine learning and deep learning are computationally expensive.

Setting up the development environment

As part of the environment setup, we will look at how to install the Anaconda tool, which gives access to almost all the required underlying tools and technologies for development. Further, we will understand how to get started with Jupyter Notebook, which is the IDE for Data Scientists. Finally, we will look at how to enhance the notebook by installing important plugins for better usability of the tool.

Installing Anaconda

The first and foremost tool that has to be installed on your machine is Anaconda. Anaconda is a distribution of packages for the Python and R programming languages. It includes a package manager called conda that can be used to install, update, and manage packages within the Anaconda environment.

Tools that come along with Anaconda are as follows:

Jupyter Notebook

Orange

Spyder

PyCharm

VS Code

RStudio

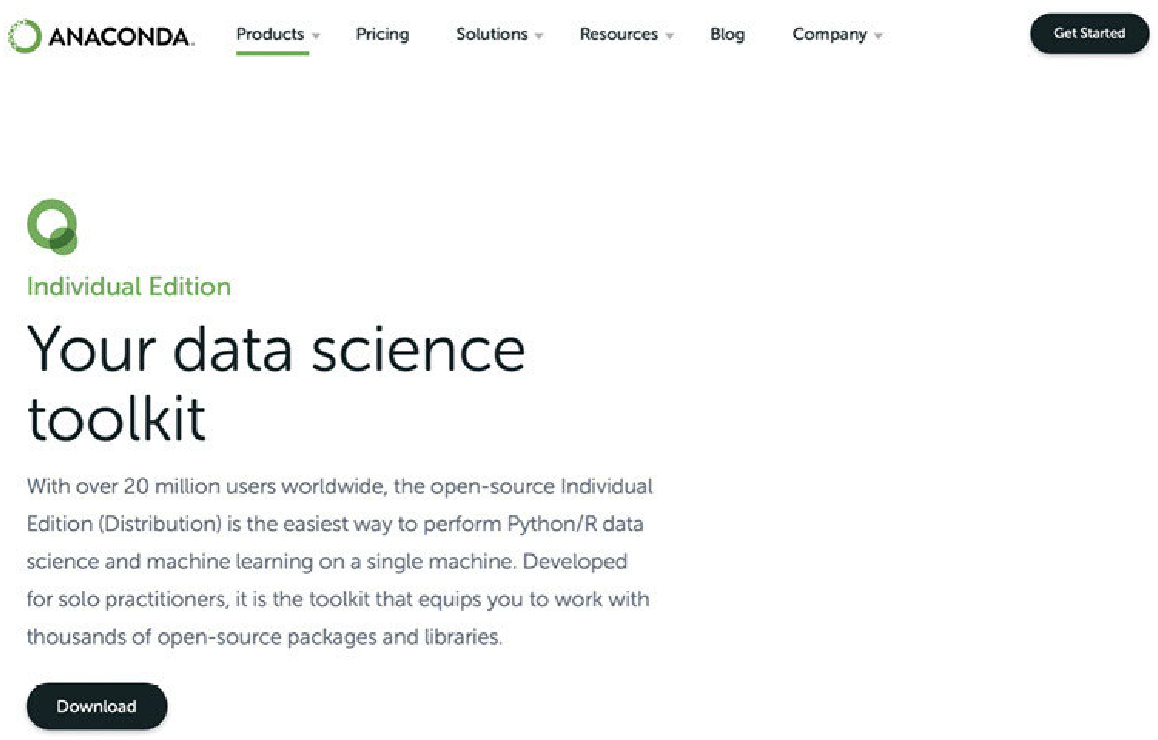
IBM Watson Studio

Each of these software can be installed separately as well. However, with anaconda, all of them can be installed in one shot.

Follow the steps to install Anaconda on your machine:

Go to the home page of Anaconda at

It will look like as shown in Figure

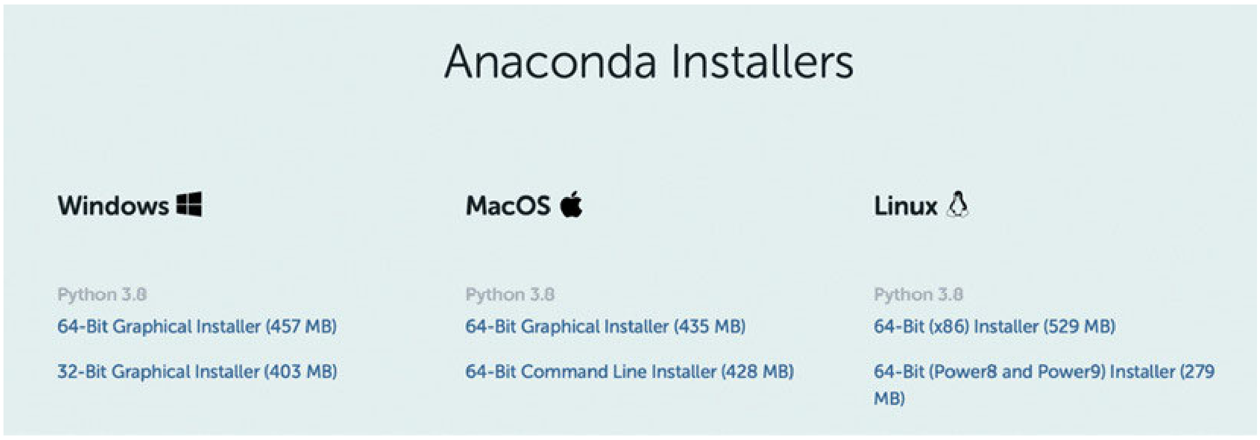


Your data science toolkit

With over 20 million users worldwide, the open-source Individual Edition (Distribution) is the easiest way to perform Python/R data science and machine learning on a single machine. Developed for solo practitioners, it is the toolkit that equips you to work with thousands of open-source packages and libraries.

Figure 1.1: Anaconda home page

Click on Download and download the installer of the choice of your operating system, as shown in the following screenshot:


Anaconda Installers

Windows

Python 3.8
64-Bit Graphical Installer (457 MB)
32-Bit Graphical Installer (403 MB)

MacOS

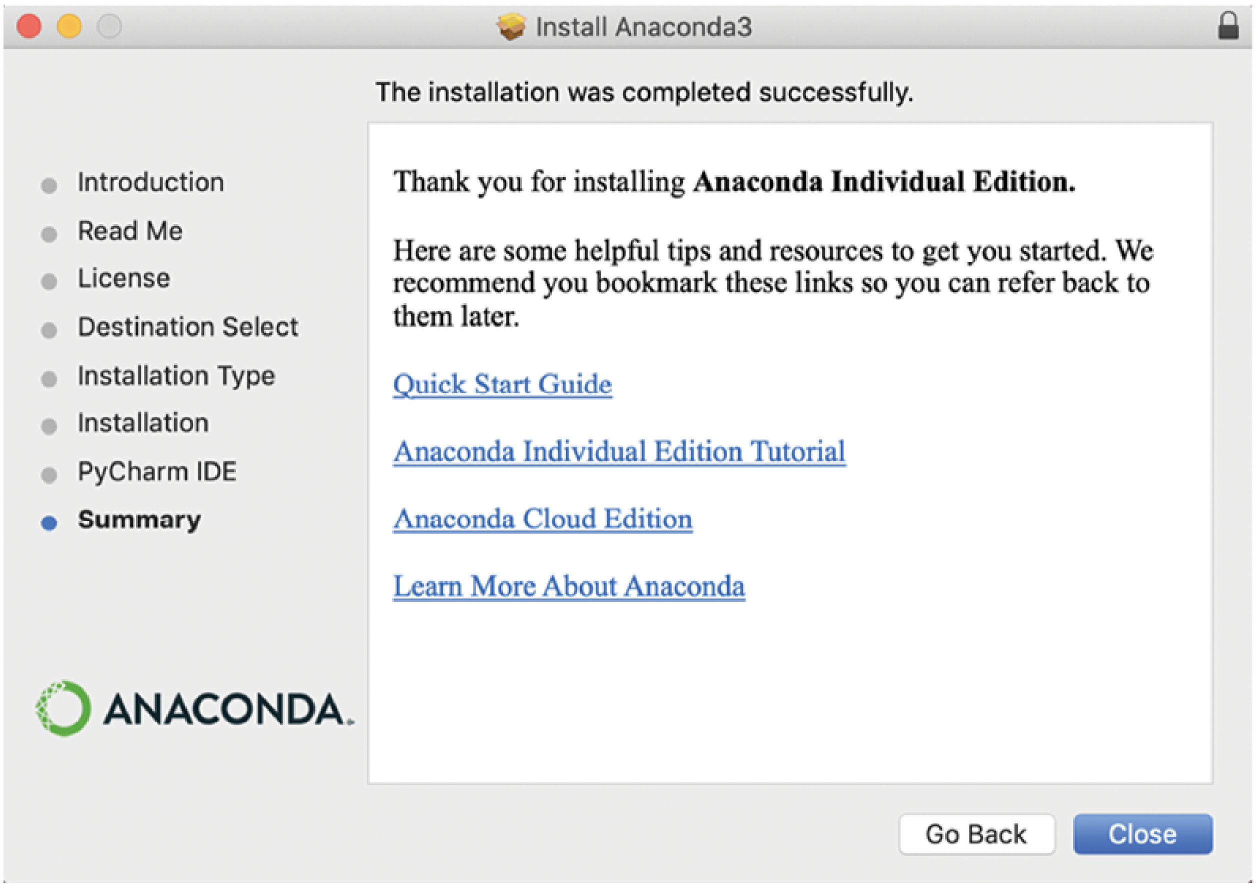
Python 3.8
64-Bit Graphical Installer (435 MB)
64-Bit Command Line Installer (428 MB)

Linux

Python 3.8
64-Bit (x86) Installer (529 MB)
64-Bit (Power8 and Power9) Installer (279 MB)

Figure 1.2: Anaconda installers

Double-click on the downloaded installer and follow the installation wizard, as shown in the following screenshot:



The installation was completed successfully.

Thank you for installing **Anaconda Individual Edition.**

Here are some helpful tips and resources to get you started. We recommend you bookmark these links so you can refer back to them later.

Quick Start Guide

Anaconda Individual Edition Tutorial

Anaconda Cloud Edition

Learn More About Anaconda

Figure 1.3: Anaconda installation wizard

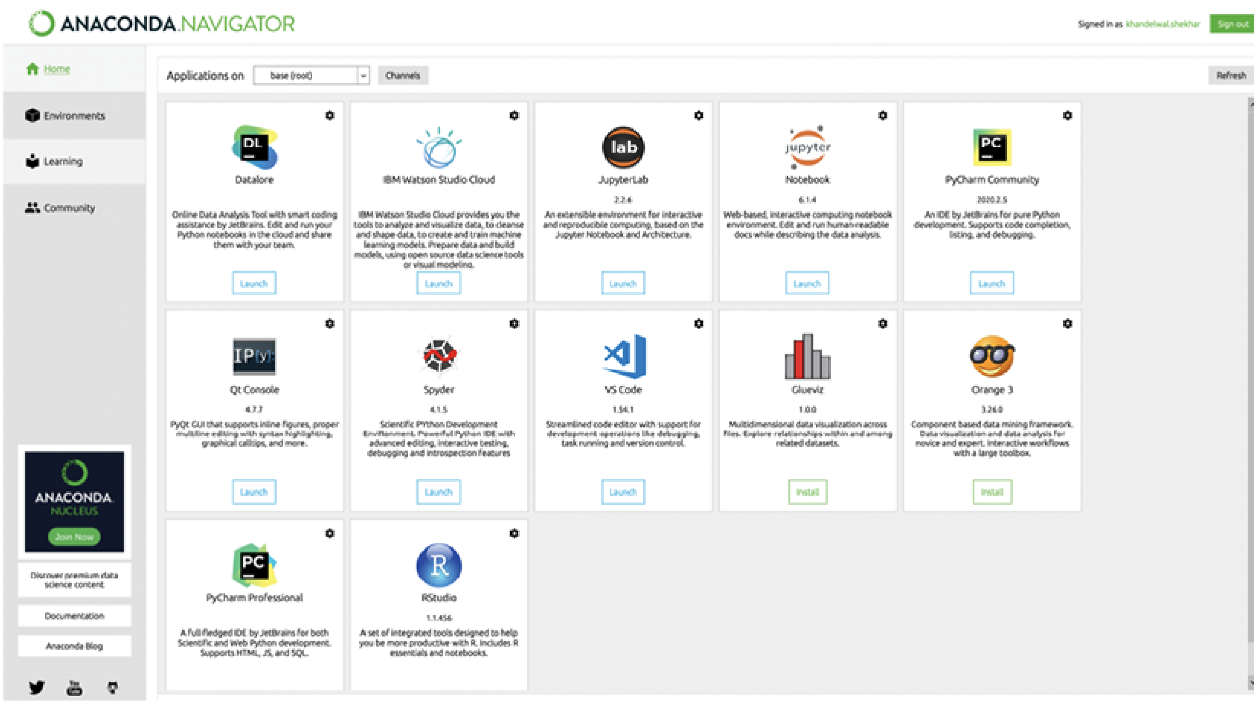
Click the Close button after you complete the installation of Anaconda on your machine.

Now, let's get started with Jupyter Notebook.

Getting Started with Jupyter Notebook

Once Anaconda is installed, you can proceed to launch the Jupyter Notebook on the web browser:

Open the Anaconda Navigator app on your machine. The screen will look as shown in Figure



Datalore
Online Data Analysis Tool with smart coding assistance by JetBrains. Edit and run your Python notebooks in the cloud and share them with your team.

IBM Watson Studio Cloud
IBM Watson Studio Cloud provides you the tools to analyze and visualize data, to cleanse and shape data, to create and train machine learning models. Prepare data and build models, using open source data science tools or visual modeling.

JupyterLab
2.2.6
An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.

Notebook
6.1.4
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

PyCharm Community
2020.2.5
An IDE by JetBrains for pure Python development. Supports code completion, listing, and debugging.

Qt Console
4.7.7
PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.

Spyder
4.1.5
Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features

VS Code
1.54.1
Streamlined code editor with support for development operations like debugging, task running and version control.

Glueviz
1.0.0
Multidimensional data visualization across files. Explore relationships within and among related datasets.

Orange 3
3.26.0
Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox.

PyCharm Professional
A full fledged IDE by JetBrains for both Scientific and Web Python development. Supports HTML, JS, and SQL.

RStudio
1.1.456
A set of integrated tools designed to help you be more productive with R. Includes R essentials and notebooks.

Figure 1.4: Anaconda Navigator

Click on the Launch button under the Jupyter Notebook icon. This will launch the Jupyter homepage on your default browser, as shown in the following screenshot:

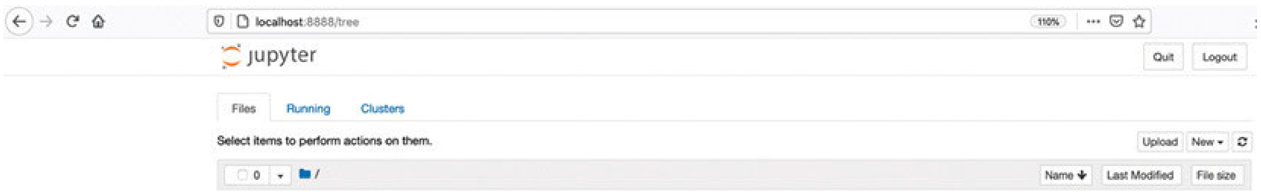


Figure 1.5: Jupyter Notebook

Further, you can navigate to the desired folder in your filesystem and go to the folder where you want to create a notebook to start building your projects.

Click on the New dropdown and click on the Python 3 option, as shown in the following screenshot:

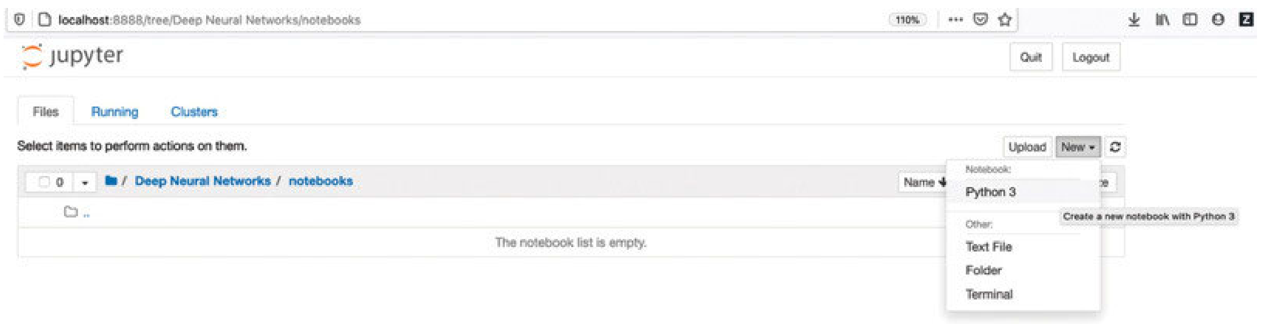

localhost:8888/tree/Deep Neural Networks/notebooks
jupyter
Quit
Logout
Files
Running
Clusters
Select items to perform actions on them.
Upload
New
Deep Neural Networks / notebooks
Name
Notebook:
Python 3
Create a new notebook with Python 3
Other:
Text File
Folder
Terminal
The notebook list is empty.

Figure 1.6: Notebook IDE

This will create a new notebook with the name which can be renamed as shown in the following screenshot:

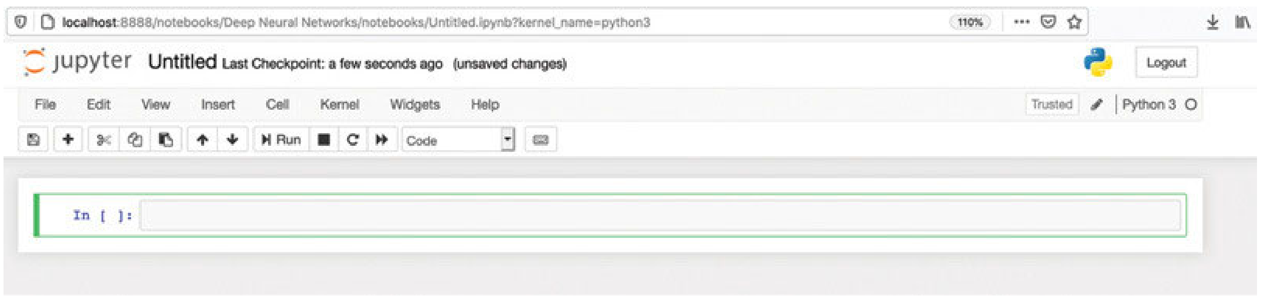
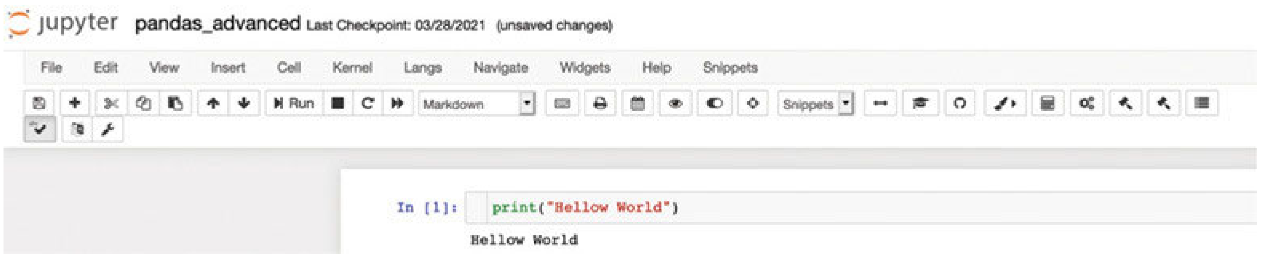

localhost:8888/notebooks/Deep Neural Networks/notebooks/Untitled.ipynb?kernel_name=python3
jupyter
Untitled Last Checkpoint: a few seconds ago (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3
Run
Code
In [ ]:

Figure 1.7: Notebook cell

You can start writing Python code in the cells, as shown in Figure and run them right there using the respective buttons provided at the top of the IDE:


jupyter pandas_advanced Last Checkpoint: 03/28/2021 (unsaved changes)
File Edit View Insert Cell Kernel Langs Navigate Widgets Help Snippets
Run
Markdown
Snippets

```
In [1]: print("Hellow World")
        Hellow World
```

Figure 1.8: Notebook Hello World

Enhancing Jupyter Notebook usability

Many seasoned data scientists usually work on the vanilla setup of Jupyter notebook. However, it is important to know that there are many more amazing features of the Jupyter Notebook that are usually not explored by most users. Once you explore them and start using them, you will realize how important they are to further enhance your usability and productivity with building models and applications using Jupyter Notebook.

These features are the ability to incorporate latex in the markdown cells of the notebook, code prettify for making the code look neater to promote best practices and preserve its integrity, the ability to automatically create a table of contents of your exploratory data analysis steps, code auto-completion with notebook, the ability to save code snippets for reusability, and many more.

Installing the required Python packages

Python, as we know, is a multi-utility programming language. It is also the most preferred programming language for data science due to the amazing scientific computing, visualization and algorithm implementation libraries.

In this section, we will install the most important Python packages that are required to build any machine learning and deep learning model in Python.

Those are as follows:

Pandas

NumPy

Matplotlib

Seaborn

Scikit Learn

TensorFlow

Keras

Scikit-image

OpenCV

Use Code 1.1 snippet and run them on the notebook cell for installing the Python packages on your machine. These are simple pip install commands that can be run on the command line too. In order to run them through the notebook cell, add an “!” before the command. That will make the notebook know that the code has to be interpreted and executed as a shell command:

```
import sys

!{sys.executable} -m pip install -U pandas

!{sys.executable} -m pip install -U numpy

!{sys.executable} -m pip install -U matplotlib

!{sys.executable} -m pip install -U seaborn

!{sys.executable} -m pip install -U scikit-learn

!{sys.executable} -m pip install -U tensorflow

!{sys.executable} -m pip install -U keras

!{sys.executable} -m pip install -U scikit-image

!{sys.executable} -m pip install -U opencv-python
```

Code pip install commands

Note that these are some basic Python libraries required for building any machine learning or deep learning model. However, additional libraries required at specific stages of the model building process will be introduced and discussed in their respective chapters.

Advance Python libraries for data science

Python has become the most preferred language for data science, owing to the amazing and easy-to-use scientific computing and model building libraries. Libraries like Pandas and Numpy provide various high-level methods for data processing. Libraries like matplotlib and seaborn make data visualization a piece of cake. Libraries like scikit-learn provide high-level methods to implement almost all machine learning algorithms. Libraries like Keras and PyTorch provide high-level methods to implement almost all deep learning algorithms.

Also, Python for data science has a very active online community, which constantly contributes to building various high-level Python packages for easy and holistic implementation of almost all stages of the data science life cycle. This makes data sourcing, data cleaning, data pre-processing, data visualization, and model building possible with just a couple of lines of code implementation.

Numpy

Numpy is a Python library that is written for scientific computing and data analysis. It stands for Numerical Python.

When it comes to data science, the data is usually big data. Even when we work on sample data for building and evaluating a predictive model, the data is huge enough to give your machine a hard time to process. Now, Python as we know it may not be that efficient in terms of computational speed and efficiency. Hence, the power of vectorization that is inherently present in the advanced Python libraries, like Numpy, comes to the rescue for faster computations.

In standard Python ways, when you need to deal with data in an iterative fashion, “for” loops are the best premise. In a loop, records are treated one row at a time, which is time-consuming and not very efficient.

Vectorization uses the Single Instruction, Multiple Data architectures. SIMD is a class of parallel computing that enables the hardware of your system to perform a single instruction on multiple data points simultaneously.

So, numpy allows you to vectorize the code for faster computations, which is of utmost importance in the world of machine learning and deep learning.

We’ll conduct a small test to demonstrate the time differences in two methods. First, we’ll use standard Python to perform element-wise multiplication of items in two lists. Then, we'll use numpy arrays for the same data and carry out a vectorized multiplication operation.

Importing the 'time' module to track the time taken for each operation:

```
import time
```

Create two lists with some random data:

```
list_1 = [i for i in range(1000000)]

list_2 = [j**2 for j in range(1000000)]
```

Capture the start time:

```
t0 = time.time()
```

Find the product of the two lists in a conventional pythonic way:

```
list_3 = list(map(lambda x, y: x*y, list_1, list_2))
```

Capture the end time:

```
t1 = time.time()
```

Print the time taken by this process:

```
print("Time taken by a standard python
```

Convert the Python lists into numpy arrays:

array_1 = np.array(list_1)

array_2 = np.array(list_2)

Capture the start time:

t0 = time.time()

Perform numpy multiplication:

array_3 = array_1*array_2

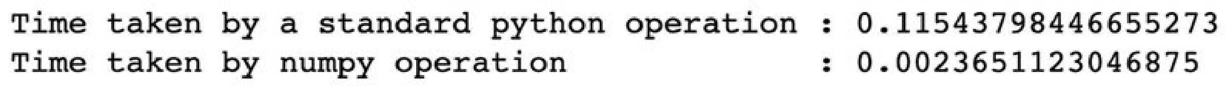
Capture the end time:

t1 = time.time()

Time taken by numpy operation:

print("Time taken by numpy operation   : {}".format(t1-t0))

Output:



```
Time taken by a standard python operation : 0.11543798446655273
Time taken by numpy operation             : 0.0023651123046875
```

Figure 1.9: Numpy time test result

As it can be seen, the time taken by standard Python operation is approximately 50

times more than the time taken by the same operation using numpy.

That clarifies why numpy is the back end of almost all data science libraries like Pandas, which we will look at next.

Pandas

Pandas is a Python library that is used extensively for data manipulation and visualization activities. The most basic data structure in Pandas is a DataFrame. Pandas DataFrame provides a tabular representation of data and many in-built data manipulation methods, which come in handy in the data exploration and preprocessing stages.

Reading and writing data to and from different file formats

Seaborn library contains various datasets that can be utilized for various demonstrations by loading those datasets in a pandas DataFrame.

First, let us import the pandas and seaborn libraries:

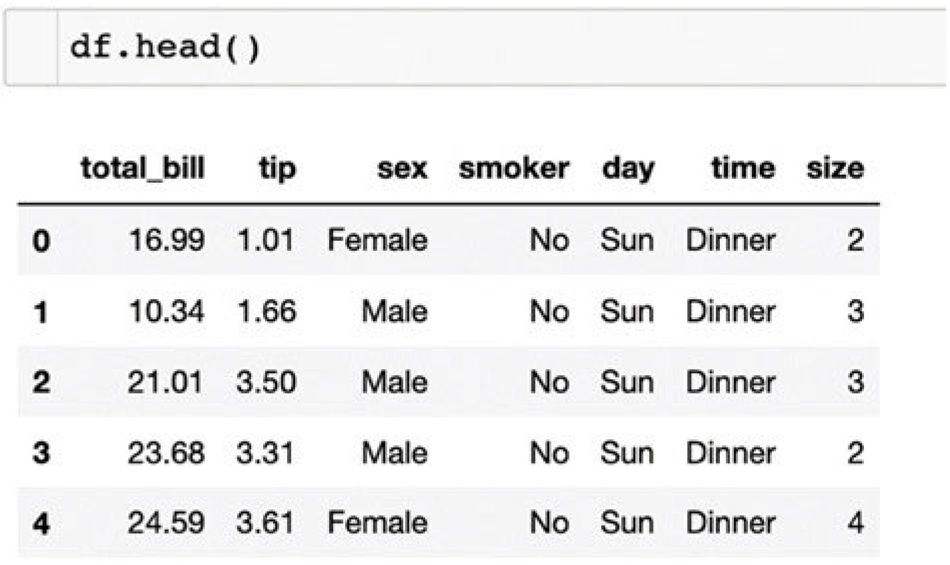
import pandas as pd

import seaborn as sns

Then, let us load one of the sample datasets into a pandas DataFrame:

df = sns.load_dataset("tips")

Output:



```
df.head()
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Figure 1.10: DataFrame head

Format - csv

To unload the Pandas DataFrame into a csv file, use the following:

```
df.to_csv("tips.csv", index=False)
```

total_bill	tip	sex	smoker			
16.99	1.01	Female	No			
10.34	1.66	Male	No			
21.01	3.5	Male	No	Sun	Dinner	
23.68	3.31	Male	No	Sun	Dinner	2
24.59	3.61	Female	No	Sun	Dinner	4
25.29	4.71	Male	No	Sun	Dinner	4
8.77	2.0	Male	No	Sun	Dinner	2
26.88	3.12	Male	No	Sun	Dinner	4
15.04	1.96	Male	No	Sun	Dinner	2
14.78	3.23	Male	No	Sun	Dinner	2
10.27	1.71	Male	No	Sun	Dinner	2
35.26	5.0	Female	No	Sun	Dinner	4
15.42	1.57	Male	No	Sun	Dinner	2
18.43	3.0	Male	No	Sun	Dinner	4
14.83	3.02	[illegible]	[illegible]	Sun	Dinner	2
21.58	3.92	[illegible]	[illegible]	Sun	Dinner	2

Figure 1.11: csv file

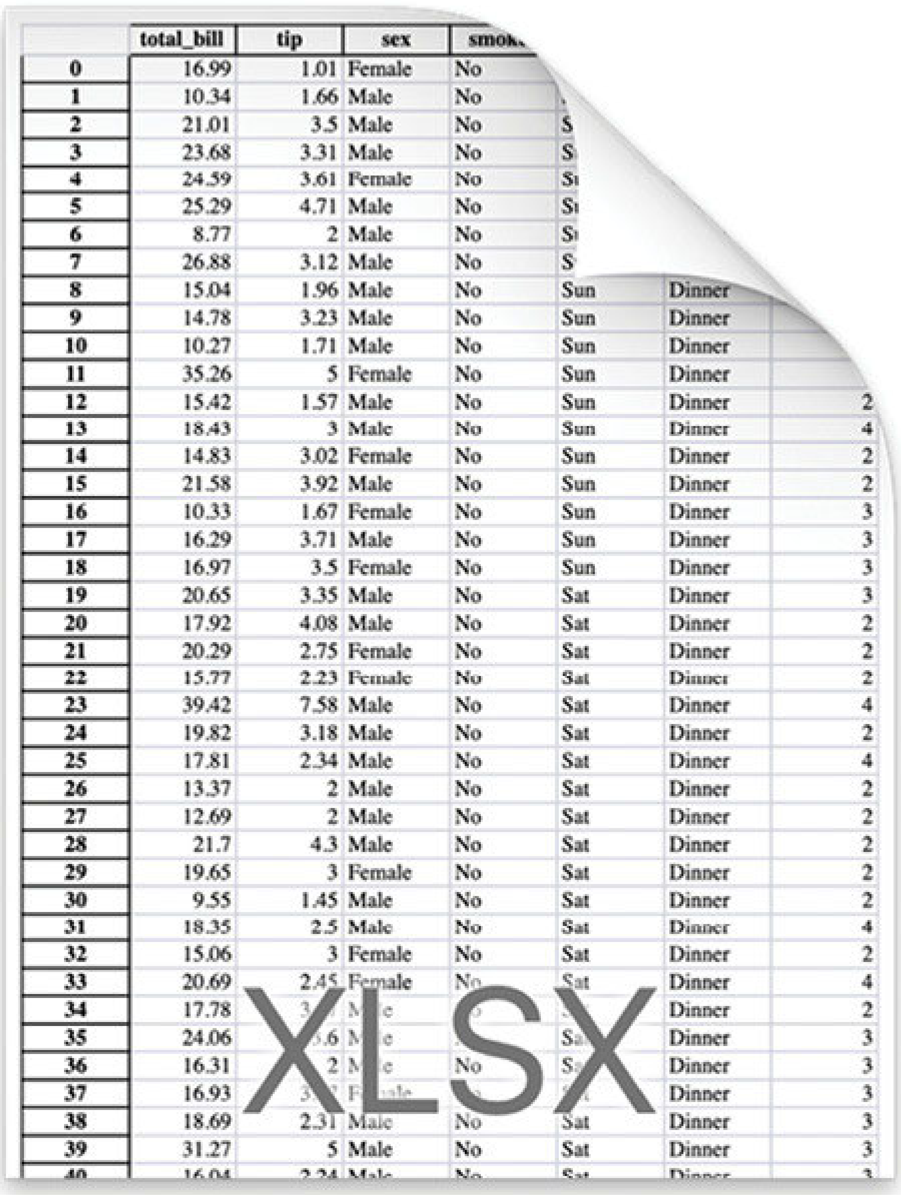
Use the following to load the csv file into Pandas DataFrame:

```
df=pd.read_csv("tips.csv")
```

Format - Excel

Use the following to unload the Pandas DataFrame into an excel file:

```
df.to_excel("tips.xlsx", index=False)
```



	total_bill	tip	sex	smok			
0	16.99	1.01	Female	No			
1	10.34	1.66	Male	No			
2	21.01	3.5	Male	No	S		
3	23.68	3.31	Male	No	S		
4	24.59	3.61	Female	No	S		
5	25.29	4.71	Male	No	S		
6	8.77	2	Male	No	S		
7	26.88	3.12	Male	No	S		
8	15.04	1.96	Male	No	Sun	Dinner	
9	14.78	3.23	Male	No	Sun	Dinner	
10	10.27	1.71	Male	No	Sun	Dinner	
11	35.26	5	Female	No	Sun	Dinner	
12	15.42	1.57	Male	No	Sun	Dinner	2
13	18.43	3	Male	No	Sun	Dinner	4
14	14.83	3.02	Female	No	Sun	Dinner	2
15	21.58	3.92	Male	No	Sun	Dinner	2
16	10.33	1.67	Female	No	Sun	Dinner	3
17	16.29	3.71	Male	No	Sun	Dinner	3
18	16.97	3.5	Female	No	Sun	Dinner	3
19	20.65	3.35	Male	No	Sat	Dinner	3
20	17.92	4.08	Male	No	Sat	Dinner	2
21	20.29	2.75	Female	No	Sat	Dinner	2
22	15.77	2.23	Female	No	Sat	Dinner	2
23	39.42	7.58	Male	No	Sat	Dinner	4
24	19.82	3.18	Male	No	Sat	Dinner	2
25	17.81	2.34	Male	No	Sat	Dinner	4
26	13.37	2	Male	No	Sat	Dinner	2
27	12.69	2	Male	No	Sat	Dinner	2
28	21.7	4.3	Male	No	Sat	Dinner	2
29	19.65	3	Female	No	Sat	Dinner	2
30	9.55	1.45	Male	No	Sat	Dinner	2
31	18.35	2.5	Male	No	Sat	Dinner	4
32	15.06	3	Female	No	Sat	Dinner	2
33	20.69	2.45	Female	No	Sat	Dinner	4
34	17.78	[illegible]	[illegible]	[illegible]	[illegible]	Dinner	2
35	24.06	[illegible]	[illegible]	[illegible]	[illegible]	Dinner	3
36	16.31	2	[illegible]	No	[illegible]	Dinner	3
37	16.93	[illegible]	[illegible]	[illegible]	[illegible]	Dinner	3
38	18.69	2.31	Male	No	Sat	Dinner	3
39	31.27	5	Male	No	Sat	Dinner	3
40	16.04	2.24	Male	No	Sat	Dinner	3

tips.xlsx

Microsoft Excel Workbook (.xlsx) - 14 KB

Figure 1.12: xlsx file

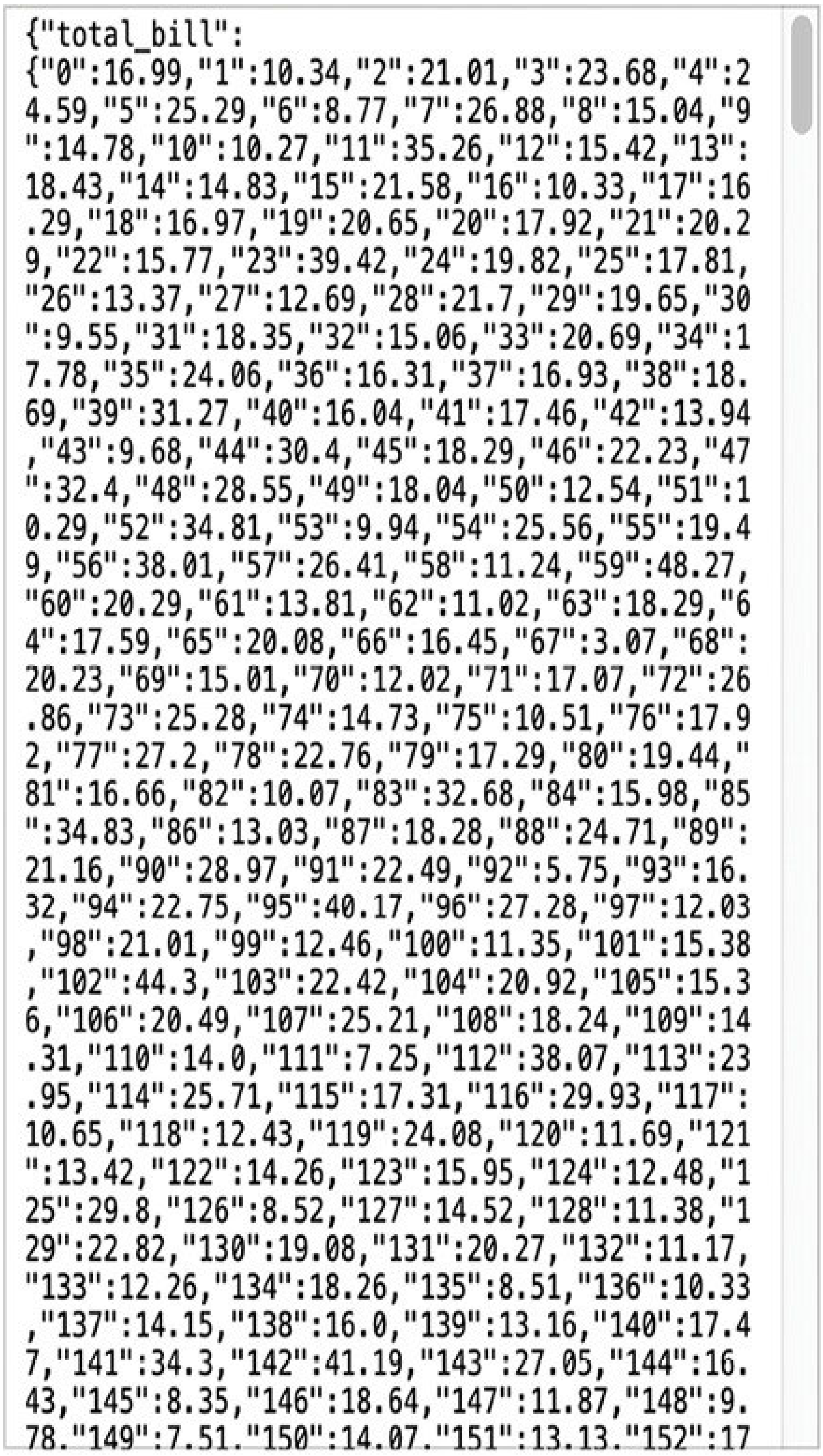
Use the following to load the excel file into Pandas DataFrame:

```
df=pd.read_excel("tips.csv")
```

Format - JSON

Use the following to unload the Pandas DataFrame into a json file:

```
df.to_json("tips.json")
```



```
{"total_bill":
{"0":16.99,"1":10.34,"2":21.01,"3":23.68,"4":2
4.59,"5":25.29,"6":8.77,"7":26.88,"8":15.04,"9
":14.78,"10":10.27,"11":35.26,"12":15.42,"13":
18.43,"14":14.83,"15":21.58,"16":10.33,"17":16
.29,"18":16.97,"19":20.65,"20":17.92,"21":20.2
9,"22":15.77,"23":39.42,"24":19.82,"25":17.81,
"26":13.37,"27":12.69,"28":21.7,"29":19.65,"30
":9.55,"31":18.35,"32":15.06,"33":20.69,"34":1
7.78,"35":24.06,"36":16.31,"37":16.93,"38":18.
69,"39":31.27,"40":16.04,"41":17.46,"42":13.94
,"43":9.68,"44":30.4,"45":18.29,"46":22.23,"47
":32.4,"48":28.55,"49":18.04,"50":12.54,"51":1
0.29,"52":34.81,"53":9.94,"54":25.56,"55":19.4
9,"56":38.01,"57":26.41,"58":11.24,"59":48.27,
"60":20.29,"61":13.81,"62":11.02,"63":18.29,"6
4":17.59,"65":20.08,"66":16.45,"67":3.07,"68":
20.23,"69":15.01,"70":12.02,"71":17.07,"72":26
.86,"73":25.28,"74":14.73,"75":10.51,"76":17.9
2,"77":27.2,"78":22.76,"79":17.29,"80":19.44,"
81":16.66,"82":10.07,"83":32.68,"84":15.98,"85
":34.83,"86":13.03,"87":18.28,"88":24.71,"89":
21.16,"90":28.97,"91":22.49,"92":5.75,"93":16.
32,"94":22.75,"95":40.17,"96":27.28,"97":12.03
,"98":21.01,"99":12.46,"100":11.35,"101":15.38
,"102":44.3,"103":22.42,"104":20.92,"105":15.3
6,"106":20.49,"107":25.21,"108":18.24,"109":14
.31,"110":14.0,"111":7.25,"112":38.07,"113":23
.95,"114":25.71,"115":17.31,"116":29.93,"117":
10.65,"118":12.43,"119":24.08,"120":11.69,"121
":13.42,"122":14.26,"123":15.95,"124":12.48,"1
25":29.8,"126":8.52,"127":14.52,"128":11.38,"1
29":22.82,"130":19.08,"131":20.27,"132":11.17,
"133":12.26,"134":18.26,"135":8.51,"136":10.33
,"137":14.15,"138":16.0,"139":13.16,"140":17.4
7,"141":34.3,"142":41.19,"143":27.05,"144":16.
43,"145":8.35,"146":18.64,"147":11.87,"148":9.
78,"149":7.51,"150":14.07,"151":13.13,"152":17
```

tips.json

JSON file - 19 KB

Figure 1.13: json file

Use the following to load the json file into Pandas DataFrame:

```
df=pd.read_json("tips.csv")
```

Pandas DataFrames can be written and read not only to and from text files like CSV, JSON and so on. but also to binary files like parquet and pickle. For the entire list of Pandas DataFrame supported file formats and their definitions, refer to this blog:

https://khandelwal-shekhar.medium.com/different-file-formats-pandas-dataframe-can-read-and-write-to-38198e48e439

Format - clipboard

Pandas DataFrames can also be written into the system’s clipboard. They can also be read directly from the clipboard, which can be useful for various kinds of applications.

Use the following to unload the Pandas DataFrame in the clipboard:

```
df.to_clipboard()
```

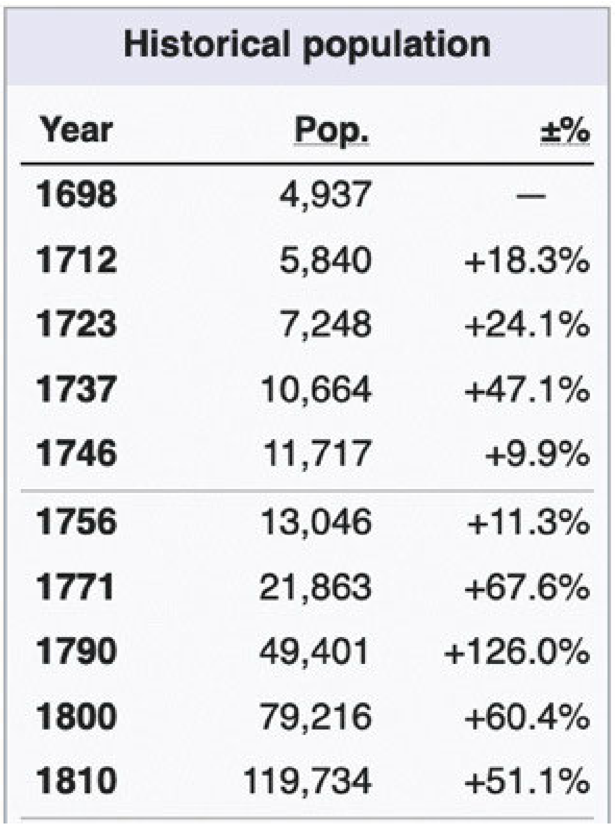
Use the following to load the content from the clipboard into Pandas DataFrame:

```
df=pd.read_clipboard()
```

Format - HTML tables

Tabular data available on various web pages can be directly read into a Pandas DataFrame.

Various tables are present on any web page. One of the table in a web page is shown as follows:



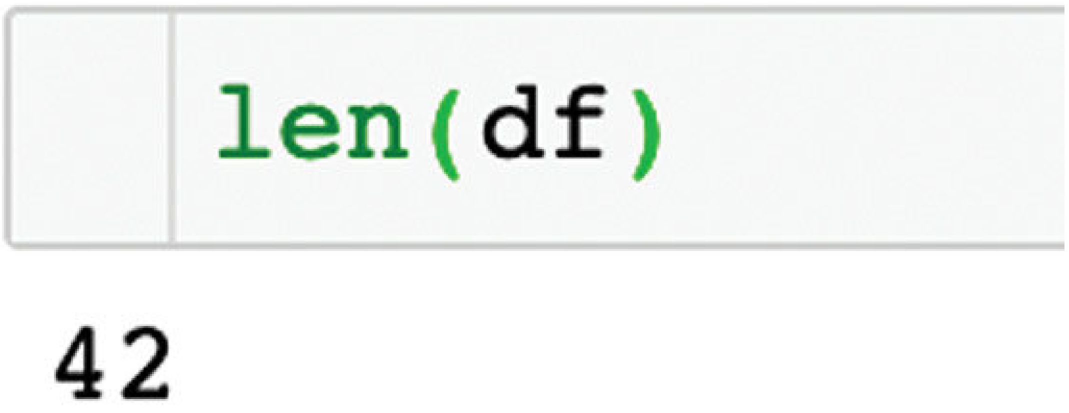
Historical population		
Year	**Pop.**	**±%**
1698	4,937	—
1712	5,840	+18.3%
1723	7,248	+24.1%
1737	10,664	+47.1%
1746	11,717	+9.9%
1756	13,046	+11.3%
1771	21,863	+67.6%
1790	49,401	+126.0%
1800	79,216	+60.4%
1810	119,734	+51.1%

Figure 1.14: HTML table

To read the tabular data from any web page table into a Pandas DataFrame, use the website link that you want to scrape, to read the HTML content in the code:

df=pd.read_html("https://url>")

However, if we look at the size of this DataFrame, it is more than 1.



```
len(df)
```

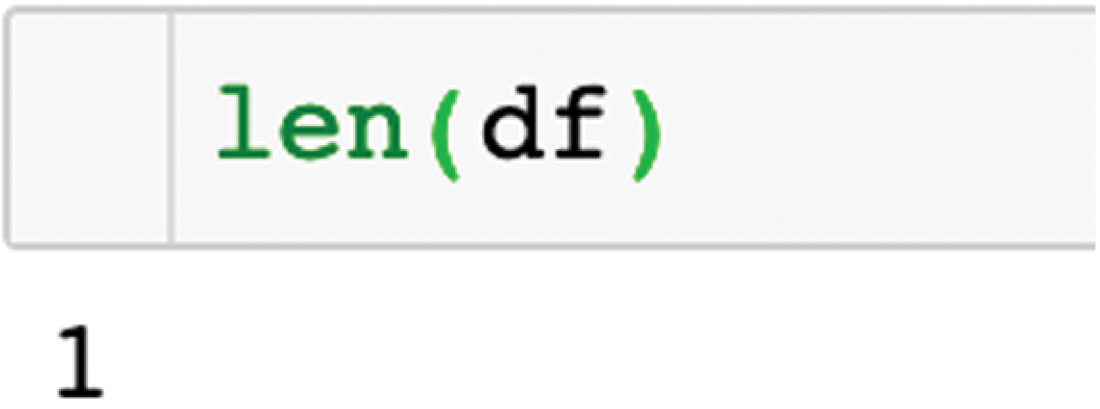
```
42
```

Figure 1.15: DataFrame length

That means this command had read all the tables in the web page. If we want to read any specific table from the web page, we can use the table heading as the identifier.

df=pd.read_html("https://url>", match='Historical population')

Output:

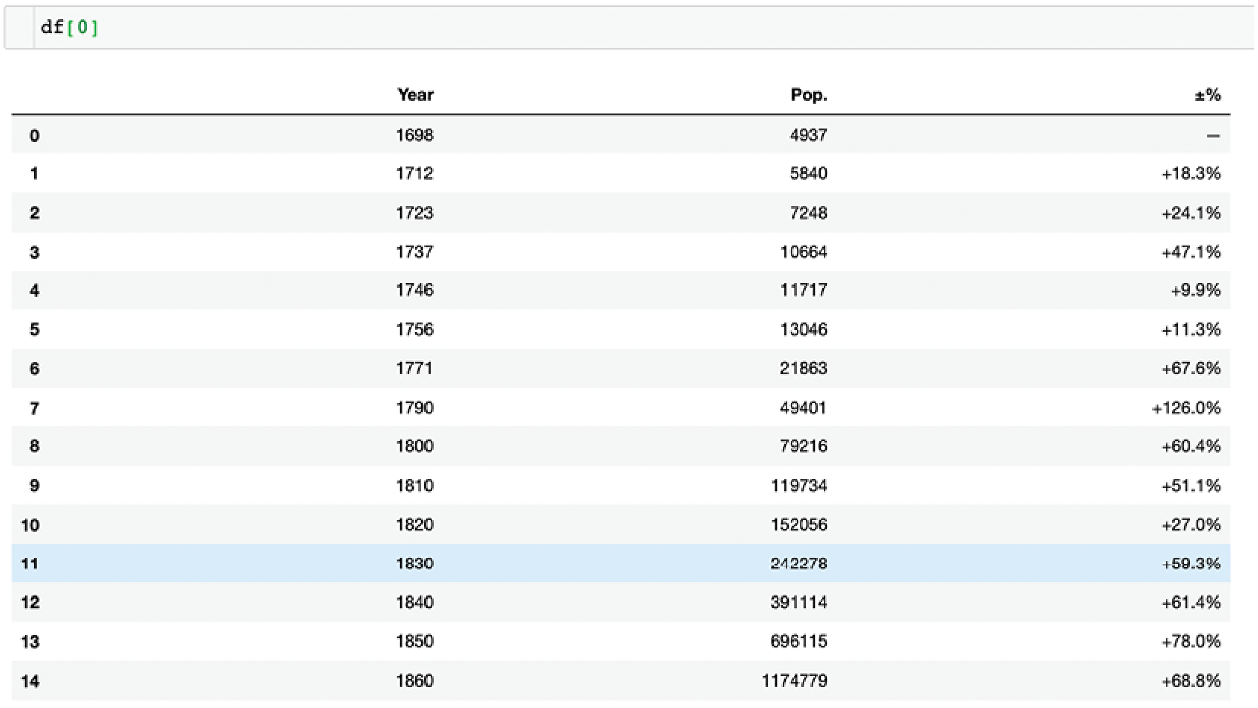


```
len(df)
```

```
1
```

Figure 1.16: DataFrame length

So, now there is only one element in this DataFrame. To see this data, read the first element of the DataFrame.



```
df[0]
```

	Year	Pop.	±%
0	1698	4937	–
1	1712	5840	+18.3%
2	1723	7248	+24.1%
3	1737	10664	+47.1%
4	1746	11717	+9.9%
5	1756	13046	+11.3%
6	1771	21863	+67.6%
7	1790	49401	+126.0%
8	1800	79216	+60.4%
9	1810	119734	+51.1%
10	1820	152056	+27.0%
11	1830	242278	+59.3%
12	1840	391114	+61.4%
13	1850	696115	+78.0%
14	1860	1174779	+68.8%

Figure 1.17: DataFrame elements

However, sometimes data is available in a different data source, which is not inherently supported by Pandas DataFrame to read from, like PDF tables, HTML content of a web page, REST APIs, and so on. To read data from such sources, we can use additional Python libraries like requests, Beautiful Soup, tabula-py, and so on.

Format - PDF

Suppose tabular data is stored in a PDF file that we intend to load in a Pandas DataFrame:

To read tabular data stored in a PDF file, we need to first install the tabula-py library using the following command snippet:

```
import sys
```

```
!{sys.executable} -m pip install -U tabula-py
```

Install Java 8+ using this website based on your operating system:

https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html

Import the following libraries:

```
import tabula
```

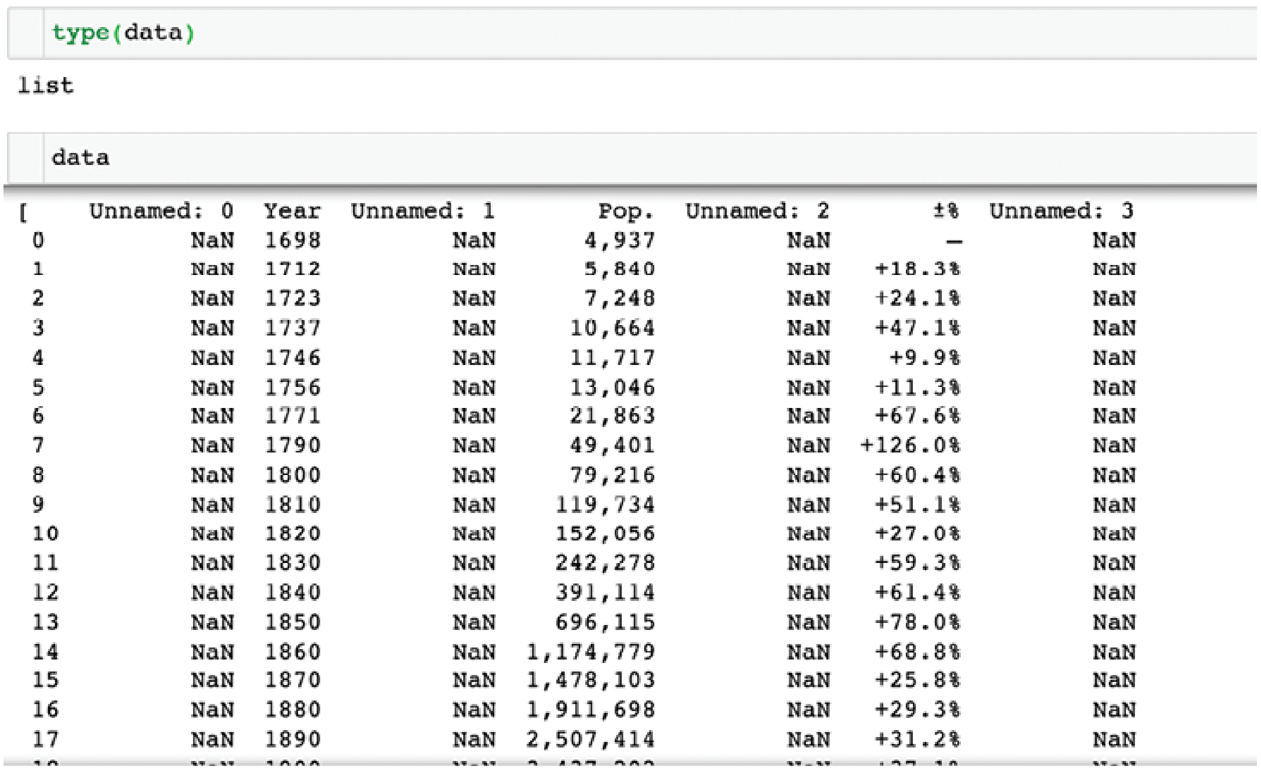
```
import pandas as pd
```

Read the PDF data into a Python object, as follows:

```
data = tabula.read_pdf("data.pdf", pages="all")
```

However, this object is a Python list, and the data is not in a very readable format.

Output:



```
type(data)
```

```
list
```

```
data
```

```
[     Unnamed: 0  Year  Unnamed: 1        Pop.  Unnamed: 2       ±%  Unnamed: 3
 0           NaN  1698         NaN       4,937         NaN        –         NaN
 1           NaN  1712         NaN       5,840         NaN   +18.3%         NaN
 2           NaN  1723         NaN       7,248         NaN   +24.1%         NaN
 3           NaN  1737         NaN      10,664         NaN   +47.1%         NaN
 4           NaN  1746         NaN      11,717         NaN    +9.9%         NaN
 5           NaN  1756         NaN      13,046         NaN   +11.3%         NaN
 6           NaN  1771         NaN      21,863         NaN   +67.6%         NaN
 7           NaN  1790         NaN      49,401         NaN  +126.0%         NaN
 8           NaN  1800         NaN      79,216         NaN   +60.4%         NaN
 9           NaN  1810         NaN     119,734         NaN   +51.1%         NaN
 10          NaN  1820         NaN     152,056         NaN   +27.0%         NaN
 11          NaN  1830         NaN     242,278         NaN   +59.3%         NaN
 12          NaN  1840         NaN     391,114         NaN   +61.4%         NaN
 13          NaN  1850         NaN     696,115         NaN   +78.0%         NaN
 14          NaN  1860         NaN   1,174,779         NaN   +68.8%         NaN
 15          NaN  1870         NaN   1,478,103         NaN   +25.8%         NaN
 16          NaN  1880         NaN   1,911,698         NaN   +29.3%         NaN
 17          NaN  1890         NaN   2,507,414         NaN   +31.2%         NaN
```

Figure 1.18: Data type

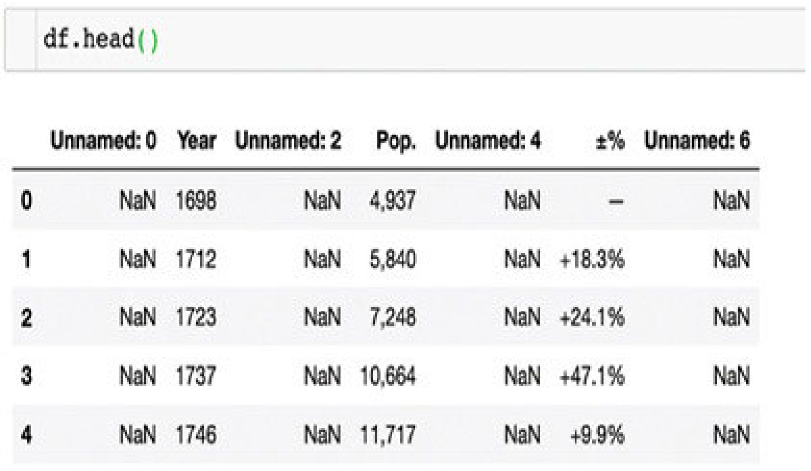
1. Tabula provides an option to read and save the data directly from pdf to csv format:

1.
tabula.convert_into("data.pdf", "data.csv", output_format="csv", pages="all")

2. Now, we can read this csv file and load it in the pandas dataFrame using the pandas read_csv method, as follows:

2. df=pd.read_csv("data.csv")

Output:



```
df.head()
```

	Unnamed: 0	Year	Unnamed: 2	Pop.	Unnamed: 4	±%	Unnamed: 6
0	NaN	1698	NaN	4,937	NaN	–	NaN
1	NaN	1712	NaN	5,840	NaN	+18.3%	NaN
2	NaN	1723	NaN	7,248	NaN	+24.1%	NaN
3	NaN	1737	NaN	10,664	NaN	+47.1%	NaN
4	NaN	1746	NaN	11,717	NaN	+9.9%	NaN

Figure 1.19: DataFrame head

However, we can see that data is not read in a very clean way, and we see that there are a few unwanted columns created during the data transition process; hence, some cleaning will be required before further processing of the DataFrame.

Format - Web scraping

To scrape a web page and extract relevant information from the content, we can use Python libraries like requests and Beautiful Soup:

Install the required Python packages – requests and Beautiful Soup:

```
import sys

!{sys.executable} -m pip install -U requests

!{sys.executable} -m pip install -U bs4
```

Import the required libraries:

```
1. import pandas as pd
```

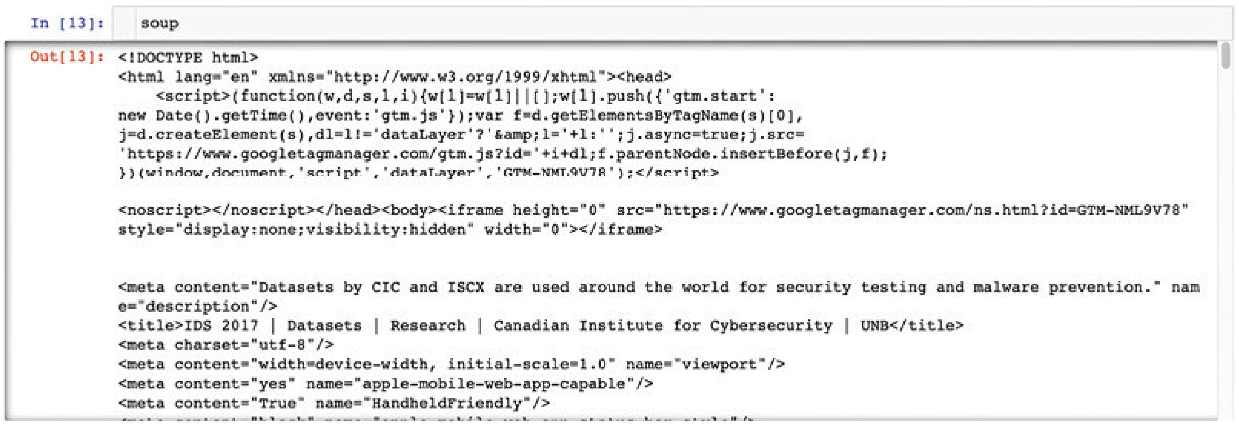
Send the GET request to the desired web page to collect the HTML content of the page:

```
url = "https://www.unb.ca/cic/datasets/ids-2017.html"

req = requests.get(url)
```

Use the Beautiful Soup library to parse the HTML content, as follows:

```
soup = bs4.BeautifulSoup(req.text, "html5lib")
```

Output:



```
In [13]:   soup

Out[13]: <!DOCTYPE html>
         <html lang="en" xmlns="http://www.w3.org/1999/xhtml"><head>
             <script>(function(w,d,s,l,i){w[l]=w[l]||[];w[l].push({'gtm.start':
         new Date().getTime(),event:'gtm.js'});var f=d.getElementsByTagName(s)[0],
         j=d.createElement(s),dl=l!='dataLayer'?'&amp;l='+l:'';j.async=true;j.src=
         'https://www.googletagmanager.com/gtm.js?id='+i+dl;f.parentNode.insertBefore(j,f);
         })(window,document,'script','dataLayer','GTM-NML9V78');</script>

         <noscript></noscript></head><body><iframe height="0" src="https://www.googletagmanager.com/ns.html?id=GTM-NML9V78"
         style="display:none;visibility:hidden" width="0"></iframe>


         <meta content="Datasets by CIC and ISCX are used around the world for security testing and malware prevention." nam
         e="description"/>
         <title>IDS 2017 | Datasets | Research | Canadian Institute for Cybersecurity | UNB</title>
         <meta charset="utf-8"/>
         <meta content="width=device-width, initial-scale=1.0" name="viewport"/>
         <meta content="yes" name="apple-mobile-web-app-capable"/>
         <meta content="True" name="HandheldFriendly"/>
```

Figure 1.20: HTML content

Improving efficiency with the pandas read_csv method

As mentioned earlier, most datasets are made available for predictive modelling in a csv file. Hence, it is important to understand the various options available in Pandas library on how to optimally use system memory while reading the data file.

Dataset used for this section demonstrations is available at

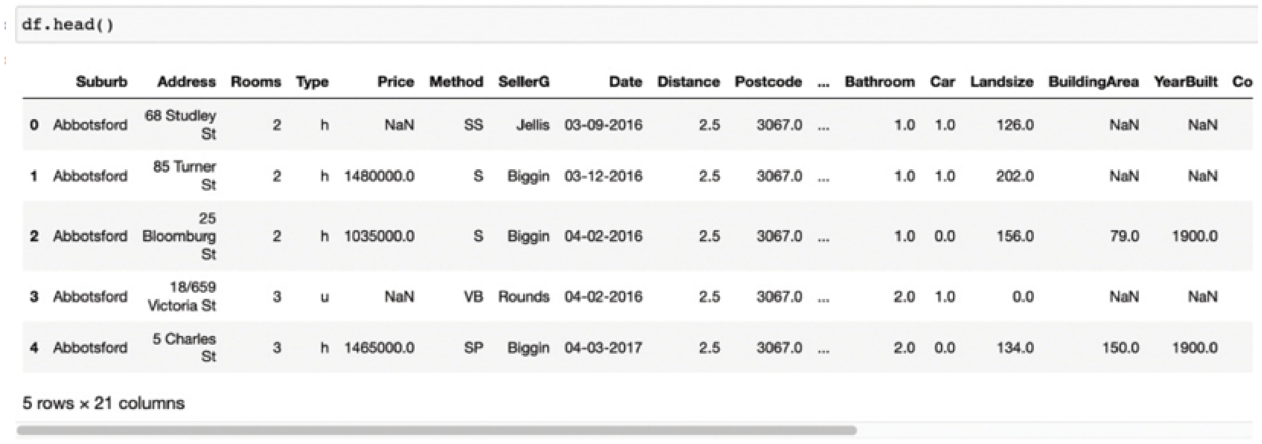
The pandas.read_csv method reads the comma-separated values (csv) file into a DataFrame. In this section, we will look at how we can use various parameters available in the read_csv method to improve the pandas DataFrame efficiency and also improve on memory utilization since that is of utmost importance when we work on huge datasets.

First, read the full CSV data into a pandas DataFrame.

Code:

```
df=pd.read_csv("melbourne.csv")

df.head()
```

Output:

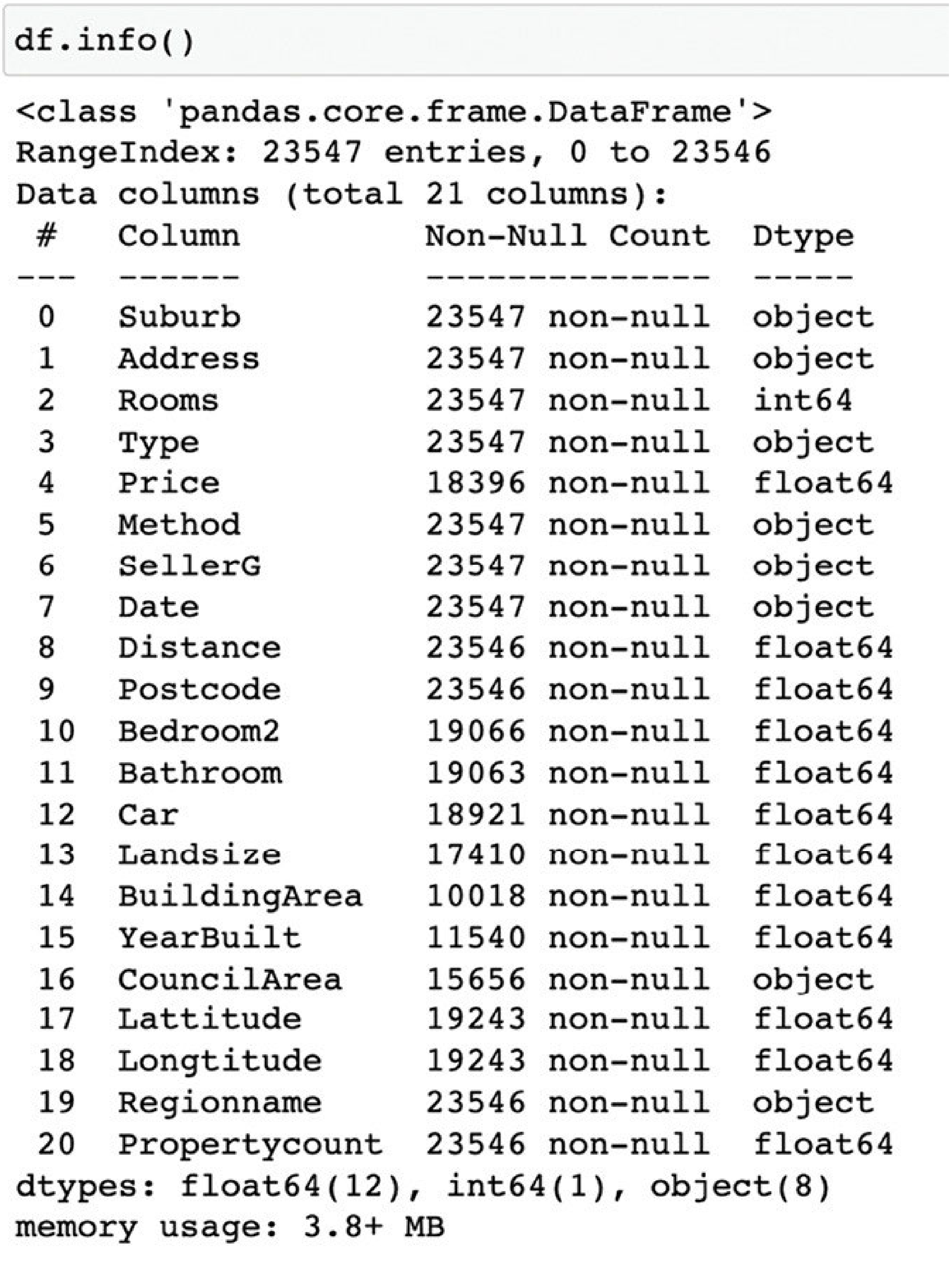


```
df.head()
```

```
      Suburb      Address  Rooms Type      Price Method  SellerG        Date  Distance  Postcode  ...  Bathroom  Car  Landsize  BuildingArea  YearBuilt  Co
0  Abbotsford  68 Studley      2    h        NaN     SS   Jellis  03-09-2016       2.5    3067.0  ...       1.0  1.0     126.0           NaN        NaN
                       St
1  Abbotsford   85 Turner      2    h  1480000.0      S   Biggin  03-12-2016       2.5    3067.0  ...       1.0  1.0     202.0           NaN        NaN
                       St
                       25
2  Abbotsford   Bloomburg      2    h  1035000.0      S   Biggin  04-02-2016       2.5    3067.0  ...       1.0  0.0     156.0          79.0     1900.0
                       St
3  Abbotsford      18/659      3    u        NaN     VB   Rounds  04-02-2016       2.5    3067.0  ...       2.0  1.0       0.0           NaN        NaN
               Victoria St
4  Abbotsford   5 Charles      3    h  1465000.0     SP   Biggin  04-03-2017       2.5    3067.0  ...       2.0  0.0     134.0         150.0     1900.0
                       St

5 rows × 21 columns
```

Figure 1.21: DataFrame head

Suppose we look at each column and the memory usage, as shown in the following screenshot:



```
df.info()
```

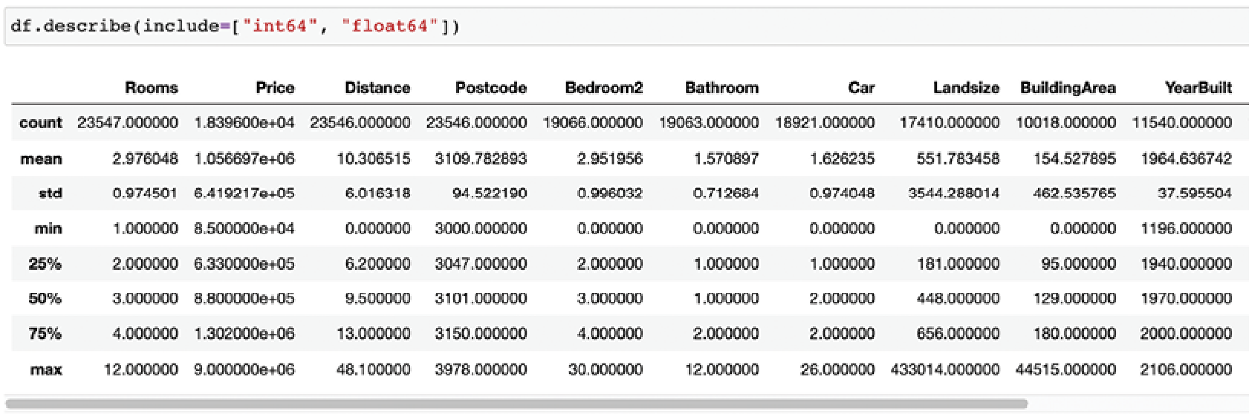
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 23547 entries, 0 to 23546
Data columns (total 21 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   Suburb          23547 non-null  object 
 1   Address         23547 non-null  object 
 2   Rooms           23547 non-null  int64  
 3   Type            23547 non-null  object 
 4   Price           18396 non-null  float64
 5   Method          23547 non-null  object 
 6   SellerG         23547 non-null  object 
 7   Date            23547 non-null  object 
 8   Distance        23546 non-null  float64
 9   Postcode        23546 non-null  float64
 10  Bedroom2        19066 non-null  float64
 11  Bathroom        19063 non-null  float64
 12  Car             18921 non-null  float64
 13  Landsize        17410 non-null  float64
 14  BuildingArea    10018 non-null  float64
 15  YearBuilt       11540 non-null  float64
 16  CouncilArea     15656 non-null  object 
 17  Lattitude       19243 non-null  float64
 18  Longtitude      19243 non-null  float64
 19  Regionname      23546 non-null  object 
 20  Propertycount   23546 non-null  float64
dtypes: float64(12), int64(1), object(8)
memory usage: 3.8+ MB
```

Figure 1.22: DataFrame info

The total memory usage is approximately 3.8 MB.

Parameter - dtype

Let us look at the details of all the numerical columns in the DataFrame:



```
df.describe(include=["int64", "float64"])
```

	Rooms	Price	Distance	Postcode	Bedroom2	Bathroom	Car	Landsize	BuildingArea	YearBuilt
count	23547.000000	1.839600e+04	23546.000000	23546.000000	19066.000000	19063.000000	18921.000000	17410.000000	10018.000000	11540.000000
mean	2.976048	1.056697e+06	10.306515	3109.782893	2.951956	1.570897	1.626235	551.783458	154.527895	1964.636742
std	0.974501	6.419217e+05	6.016318	94.522190	0.996032	0.712684	0.974048	3544.288014	462.535765	37.595504
min	1.000000	8.500000e+04	0.000000	3000.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1196.000000
25%	2.000000	6.330000e+05	6.200000	3047.000000	2.000000	1.000000	1.000000	181.000000	95.000000	1940.000000
50%	3.000000	8.800000e+05	9.500000	3101.000000	3.000000	1.000000	2.000000	448.000000	129.000000	1970.000000
75%	4.000000	1.302000e+06	13.000000	3150.000000	4.000000	2.000000	2.000000	656.000000	180.000000	2000.000000
max	12.000000	9.000000e+06	48.100000	3978.000000	30.000000	12.000000	26.000000	433014.000000	44515.000000	2106.000000

Figure 1.23: DataFrame description

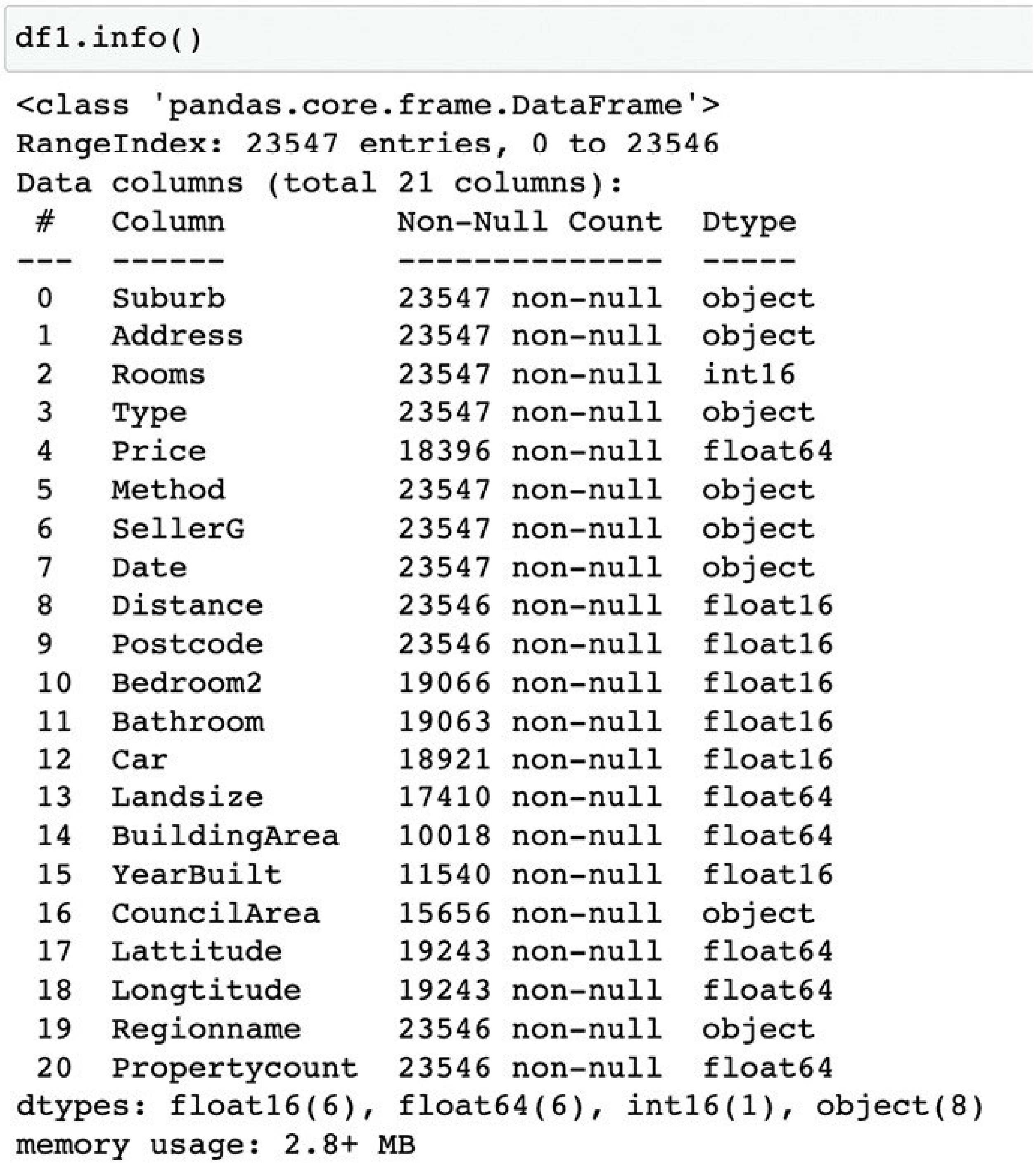
You can see that columns like Car and YearBuilt can be easily accommodated in int16 or float16 data types, but by default, pandas reads them all in int64 or float64 data types, which translates to more memory footprint.

Hence, read_csv provides a parameter called which can be used to specify the desired data type for selected columns.

Code:

```
df1=pd.read_csv(file, dtype={

                    "Rooms":np.int32,
                    "Distance":np.float16,

                    "Postcode":np.float16,

                    "Bedroom2":np.float16,

                    "Bathroom":np.float16,

                    "Car":np.float16,

                   "YearBuilt":np.float16

                    }


             )

df1.info()
```

Output:


```
df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 23547 entries, 0 to 23546
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Suburb         23547 non-null  object
 1   Address        23547 non-null  object
 2   Rooms          23547 non-null  int16
 3   Type           23547 non-null  object
 4   Price          18396 non-null  float64
 5   Method         23547 non-null  object
 6   SellerG        23547 non-null  object
 7   Date           23547 non-null  object
 8   Distance       23546 non-null  float16
 9   Postcode       23546 non-null  float16
 10  Bedroom2       19066 non-null  float16
 11  Bathroom       19063 non-null  float16
 12  Car            18921 non-null  float16
 13  Landsize       17410 non-null  float64
 14  BuildingArea   10018 non-null  float64
 15  YearBuilt      11540 non-null  float16
 16  CouncilArea    15656 non-null  object
 17  Lattitude      19243 non-null  float64
 18  Longtitude     19243 non-null  float64
 19  Regionname     23546 non-null  object
 20  Propertycount  23546 non-null  float64
dtypes: float16(6), float64(6), int16(1), object(8)
memory usage: 2.8+ MB
```

Figure 1.24: DataFrame info

Now, it can be seen that the same data stored in the new pandas DataFrame uses only 2.8 MB of memory space. This can be a huge memory saving technique when working with huge datafiles.

Parameter - usecols

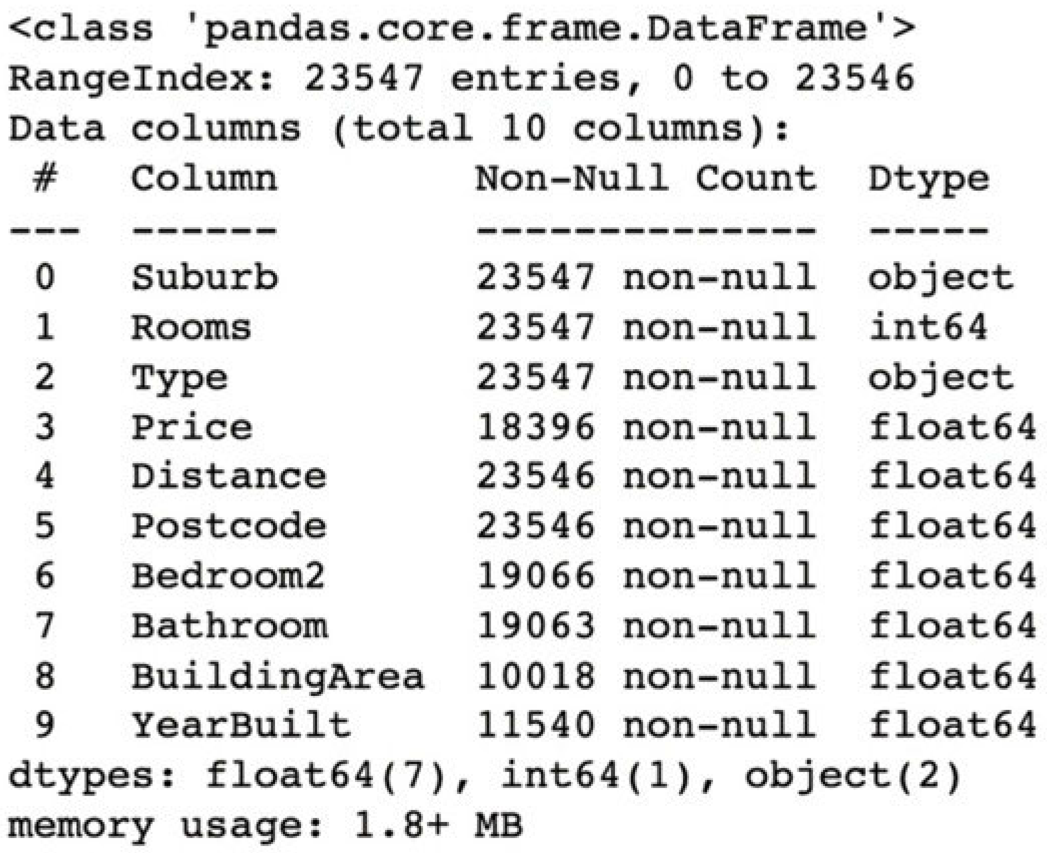
With the preliminary understanding of the features in the dataset, sometimes you may find certain columns that are of no use for the modelling. For example, in the Melbourne housing dataset, assume that columns like Address, latitude, longitude and so on are of no importance. And you are aware of it before even reading the data into a pandas DataFrame. There is always an option to remove them after the DataFrame is created. But when you deal with huge datasets, it would be pretty time consuming and costly to read unnecessary columns, only to remove them later.

The usecols parameter of the read_csv method comes handy at such times. It lets the user define the columns they want to use from the CSV to form the DataFrame.

Code:

```
cols =
["Suburb","Rooms","Type","Price","Distance", "Postcode", "Bathroom", "
Bedroom2", "BuildingArea", "YearBuilt"]

df2 = pd.read_csv(file, usecols=cols)

df2.info()
```

Output:



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 23547 entries, 0 to 23546
Data columns (total 10 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   Suburb        23547 non-null  object
 1   Rooms         23547 non-null  int64
 2   Type          23547 non-null  object
 3   Price         18396 non-null  float64
 4   Distance      23546 non-null  float64
 5   Postcode      23546 non-null  float64
 6   Bedroom2      19066 non-null  float64
 7   Bathroom      19063 non-null  float64
 8   BuildingArea  10018 non-null  float64
 9   YearBuilt     11540 non-null  float64
dtypes: float64(7), int64(1), object(2)
memory usage: 1.8+ MB
```

Figure 1.25: DataFrame info

Memory usage is further reduced to 1.8 MB.

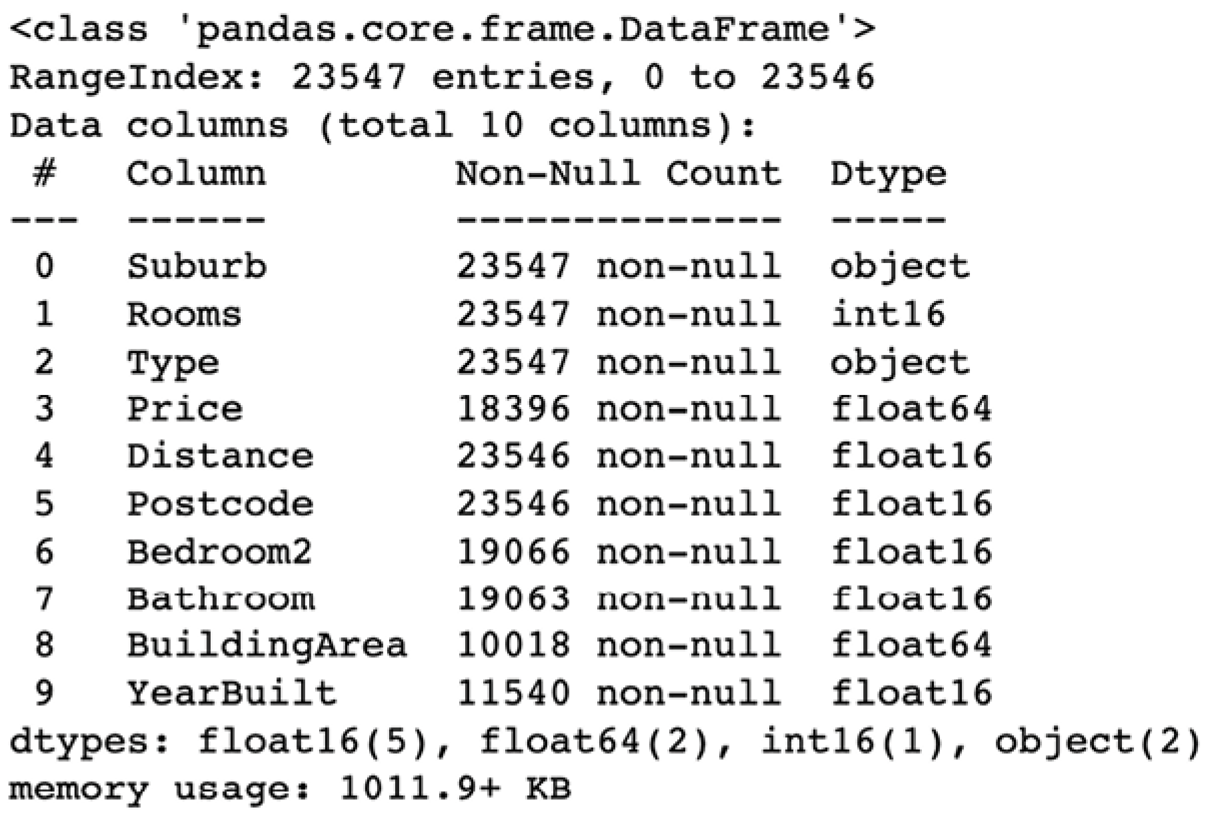
Now, let’s use dtype and usecols together to further optimize the DataFrame.

Code:

```
cols =
["Suburb","Rooms","Type","Price","Distance", "Postcode", "Bathroom", "
Bedroom2", "BuildingArea", "YearBuilt"]
```

```
df3 = pd.read_csv(file,
                  usecols=cols,
                 dtype={
                     "Rooms":np.int16,
                    "Distance":np.float16,
                    "Postcode":np.float16,

                    "Bedroom2":np.float16,
                    "Bathroom":np.float16,
                    "YearBuilt":np.float16
                     })
df3.info()
```

Output:



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 23547 entries, 0 to 23546
Data columns (total 10 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   Suburb        23547 non-null  object
 1   Rooms         23547 non-null  int16
 2   Type          23547 non-null  object
 3   Price         18396 non-null  float64
 4   Distance      23546 non-null  float16
 5   Postcode      23546 non-null  float16
 6   Bedroom2      19066 non-null  float16
 7   Bathroom      19063 non-null  float16
 8   BuildingArea  10018 non-null  float64
 9   YearBuilt     11540 non-null  float16
dtypes: float16(5), float64(2), int16(1), object(2)
memory usage: 1011.9+ KB
```

Figure 1.26: DataFrame info

Memory usage is further reduced to a few thousand KBs.

Parameter - chunksize

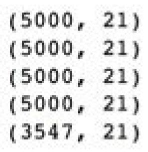
While working with datasets that range from a few GBs or even TBs, even with all the stated hacks, it will not be possible to read the entire dataset in the DataFrame due to memory limitations.

Then, we can use the chunksize parameter to read the data in chunks. The chunksize parameter will give the flexibility to read the dataset in the chunks of the desired number of rows.

Code:

```
for chunk in pd.read_csv(file, chunksize=5000):

    print(chunk.shape)
```

Output:



```
(5000, 21)
(5000, 21)
(5000, 21)
(5000, 21)
(3547, 21)
```

Figure 1.27: File shapes

Conclusion

In this chapter, we looked at setting up the data science development environment. We also looked at various advanced Python packages and concepts that can help in the day-to-day data wrangling process for anyone who is trying to build a predictive model. This chapter also provided a lot of code snippets that can be readily used for various common data sourcing, data loading, data cleaning and preprocessing, and data visualization tasks.

In the next chapter, we will explore real-world challenges faced by data architects to implement data science techniques for data problems. Also, we will look at how such challenges can be resolved by using ready-to-use automated tools and techniques.

Questions

Which of the following Python libraries is/are widely used in data science?

Pandas

Numpy

Matplotlib

All of the above

NumPy stands for which of the following?

Number Python

Numerical Python

Numbers in Python

None of the above

PANDAS stands for which of the following?

Panel data analysis

Panel data analyst

Panel data

Panel dashboard

What are different file types that can be read using the pandas library?

CSV

Excel

JSON

All of the above

Which Python library can be used for API calls?

Seaborn

Beautiful Soup

Requests

All of the above

Answers

(d)

(b)

(d)

(d)

(c)

CHAPTER 2

Real-World Challenges for Data Professionals in Converting Data Into Insights

The miracle isn't that I finished. The miracle is that I had the courage to start.

— John Bingham

Well begun is half done! Most of the time, if you have observed that once you define the problem statement and collect the initial set of observations aka data, you find it difficult to get started with data crunching or data analysis. You kind of get dumbstruck on picking up a variable from your dataset to start performing the data understanding process. The first step is the most important step to finishing the marathon.

At such times, various automated data profiling, data visualization, and model building libraries come in handy. These libraries give you a head start in your predictive model building process. Such libraries come with very easy-to-use, high-level functions that can perform various complicated data analysis and plotting tasks with just a few lines of code.

Further, it has also been observed that once you clean and pre-process your data, you find yourself clueless on which algorithm to apply based on your problem statement and the dataset collected. Hence, the question arises, 'Why not use all the available algorithms?' Well, that does not seem feasible since there are many, and usually, time is limited. There are automated tools that allow you to implement almost all the popular regression or classification algorithms in one shot, with just a few lines of code.

In this chapter, we will discuss various such amazing Python libraries that automate the most cumbersome model building processes and give you the head start that you are looking for to begin your data science problem-solving journey.

Structure

In this chapter, we will cover the following topics:

Pandas-profiling

Sweetviz

AutoViz

Lux

Lazy Predict

PyCaret

Objectives

After studying this chapter, you should be able to automate various important model building processes like descriptive analytics, exploratory data analysis, and model building experimentations. You will also be able to use various advanced Python libraries to automate all these steps and further build upon this knowledge to enhance your predictive model building processes.

Pandas profiling

Pandas-profiling is a Python library that makes descriptive analytics a matter of just a few lines of code. It provides a detailed analysis of the dataset in just seconds. It may not be everything that you need to know about a dataset, but it is a good starting point to understand your data, with all the basic statistics of the features and an optimum level of visualizations, before starting any kind of exploratory data analytics.

Installing and getting started with pandas profiling

Following are the steps to install Pandas profiling and getting started with it:

Step 1: Install the pandas-profiling Python package, as shown here:

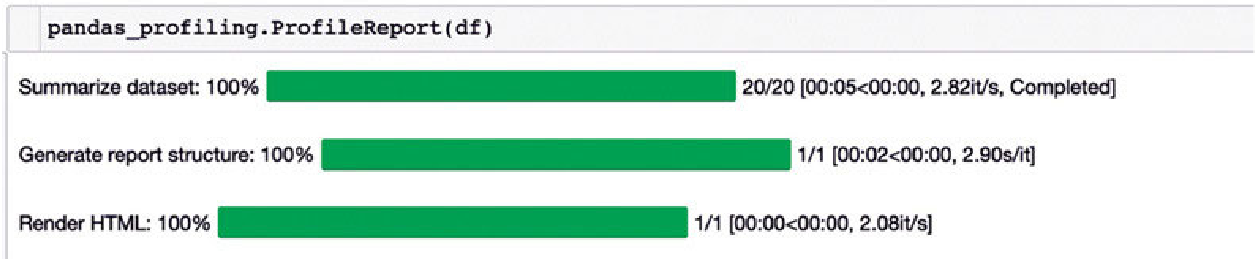
```
import sys

!{sys.executable} -m pip install -U pandas-profiling
```

Step 2: Import pandas_profiling with the following command:

```
import pandas_profiling
```

Step 3: Call the ProfileReport method within pandas_profiling and pass the DataFrame to the method:

```
pandas_profiling.ProfileReport(df)
```



```
pandas_profiling.ProfileReport(df)
```

```
Summarize dataset: 100%          20/20 [00:05<00:00, 2.82it/s, Completed]
Generate report structure: 100%  1/1 [00:02<00:00, 2.90s/it]
Render HTML: 100%                1/1 [00:00<00:00, 2.08it/s]
```

Figure 2.1: Pandas Profile Report

Once completed, a report is generated describing the whole DataFrame. You can navigate through the entire report to completely understand the data they are dealing with. At the top of the report, you will find all the different kind of analysis that has been done on the data by Pandas Profiling, as seen in the following screenshot:

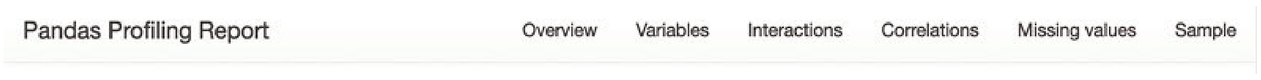
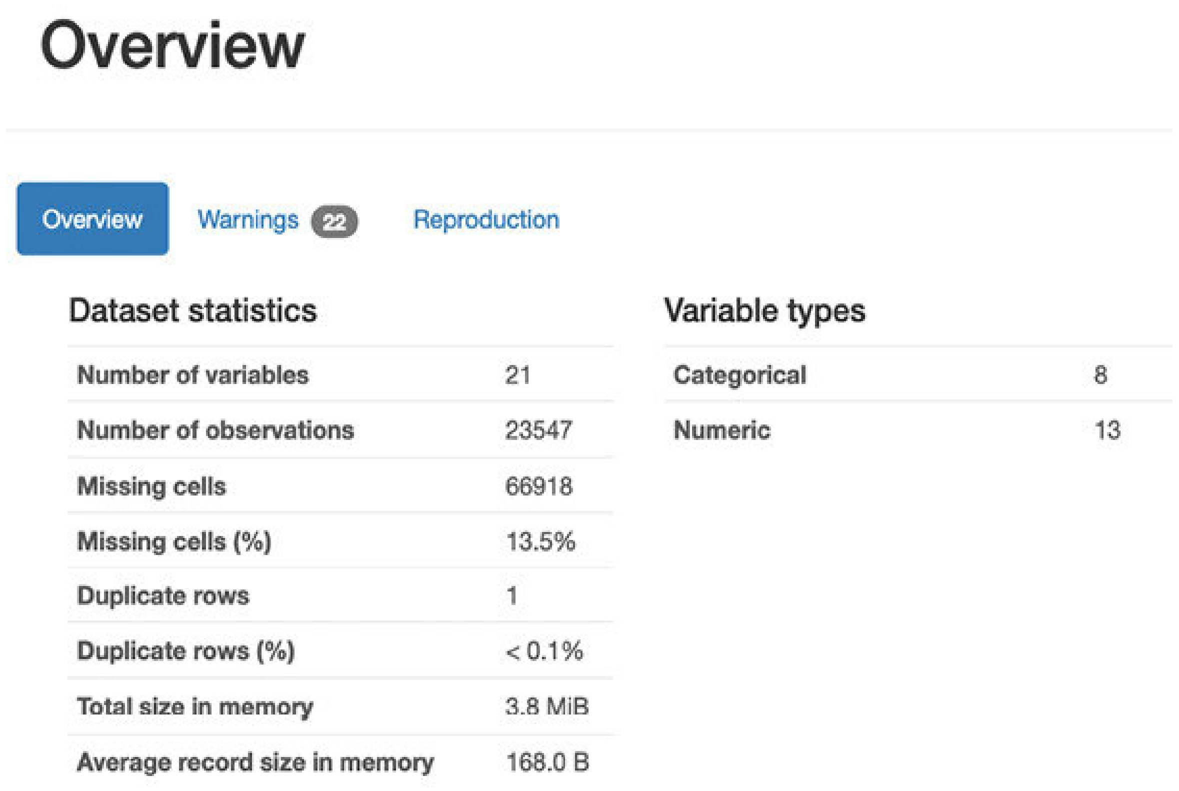


Figure 2.2: Profile Report options

Analyzing Pandas profile report

In the Overview section of the report, you will see details like the number of variables, number of observations, how many of them are categorical, and how many are numerical, and so on. This gives a holistic view of the entire dataset, as shown in the following screenshot:



Dataset statistics

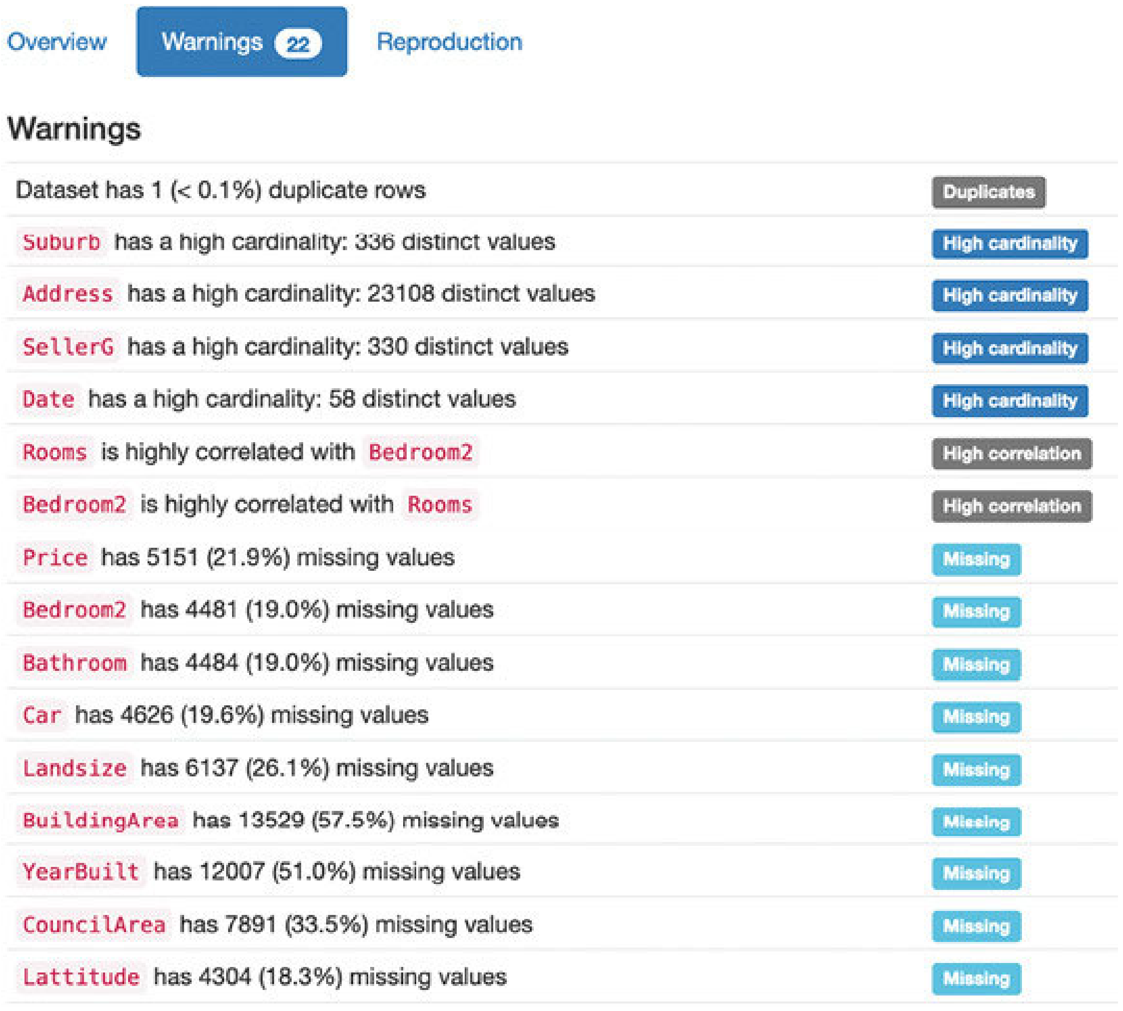
Number of variables	21
Number of observations	23547
Missing cells	66918
Missing cells (%)	13.5%
Duplicate rows	1
Duplicate rows (%)	< 0.1%
Total size in memory	3.8 MiB
Average record size in memory	168.0 B

Variable types

Categorical	8
Numeric	13

Figure 2.3: Pandas Profile overview

The Warnings tab in the overview section will provide more information about dataset variables like cardinality, distinct values, missing values, and correlations among variables in a nutshell for a quick reference. Refer to the following screenshot:

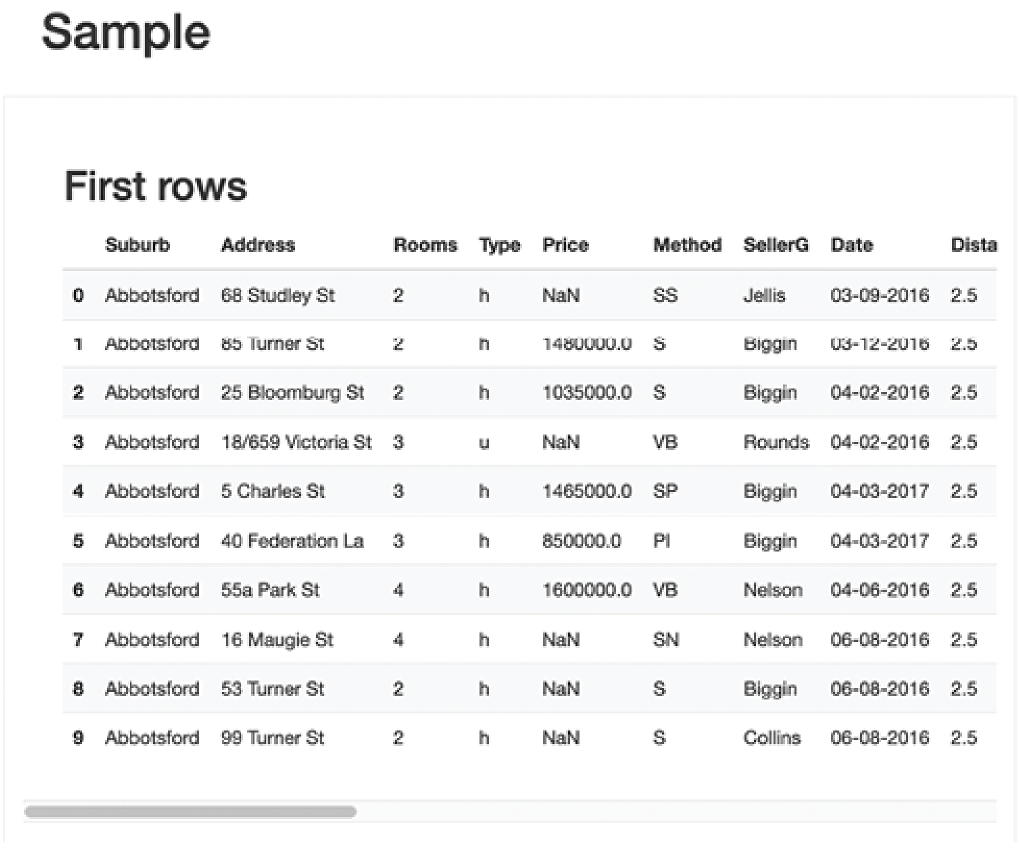


Warnings

Dataset has 1 (< 0.1%) duplicate rows	Duplicates
`Suburb` has a high cardinality: 336 distinct values	High cardinality
`Address` has a high cardinality: 23108 distinct values	High cardinality
`SellerG` has a high cardinality: 330 distinct values	High cardinality
`Date` has a high cardinality: 58 distinct values	High cardinality
`Rooms` is highly correlated with `Bedroom2`	High correlation
`Bedroom2` is highly correlated with `Rooms`	High correlation
`Price` has 5151 (21.9%) missing values	Missing
`Bedroom2` has 4481 (19.0%) missing values	Missing
`Bathroom` has 4484 (19.0%) missing values	Missing
`Car` has 4626 (19.6%) missing values	Missing
`Landsize` has 6137 (26.1%) missing values	Missing
`BuildingArea` has 13529 (57.5%) missing values	Missing
`YearBuilt` has 12007 (51.0%) missing values	Missing
`CouncilArea` has 7891 (33.5%) missing values	Missing
`Lattitude` has 4304 (18.3%) missing values	Missing

Figure 2.4: Pandas Profile warnings

Sample section will show the top and last few records of the dataset, as shown in the following screenshot:

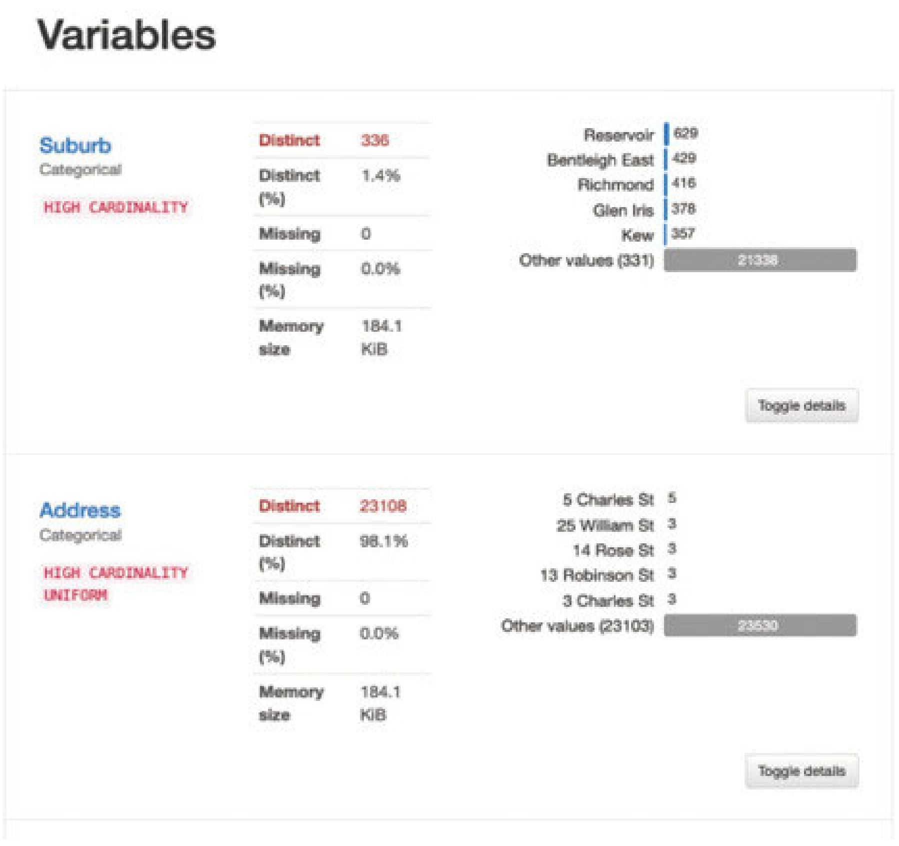

Sample

First rows

	Suburb	Address	Rooms	Type	Price	Method	SellerG	Date	Dista
0	Abbotsford	68 Studley St	2	h	NaN	SS	Jellis	03-09-2016	2.5
1	Abbotsford	85 Turner St	2	h	1480000.0	S	Biggin	03-12-2016	2.5
2	Abbotsford	25 Bloomburg St	2	h	1035000.0	S	Biggin	04-02-2016	2.5
3	Abbotsford	18/659 Victoria St	3	u	NaN	VB	Rounds	04-02-2016	2.5
4	Abbotsford	5 Charles St	3	h	1465000.0	SP	Biggin	04-03-2017	2.5
5	Abbotsford	40 Federation La	3	h	850000.0	PI	Biggin	04-03-2017	2.5
6	Abbotsford	55a Park St	4	h	1600000.0	VB	Nelson	04-06-2016	2.5
7	Abbotsford	16 Maugie St	4	h	NaN	SN	Nelson	06-08-2016	2.5
8	Abbotsford	53 Turner St	2	h	NaN	S	Biggin	06-08-2016	2.5
9	Abbotsford	99 Turner St	2	h	NaN	S	Collins	06-08-2016	2.5

Figure 2.5: Sample rows

The Variables section will provide detailed information about each variable in the dataset. Information like the number of records, distinct values, value distributions and so on are shown in this section; refer to the following figure:



Variables

Suburb
Categorical
HIGH CARDINALITY

Distinct	336
Distinct (%)	1.4%
Missing	0
Missing (%)	0.0%
Memory size	184.1 KiB

Reservoir	629
Bentleigh East	429
Richmond	416
Glen Iris	378
Kew	357
Other values (331)	21338

Toggle details

Address
Categorical
HIGH CARDINALITY
UNIFORM

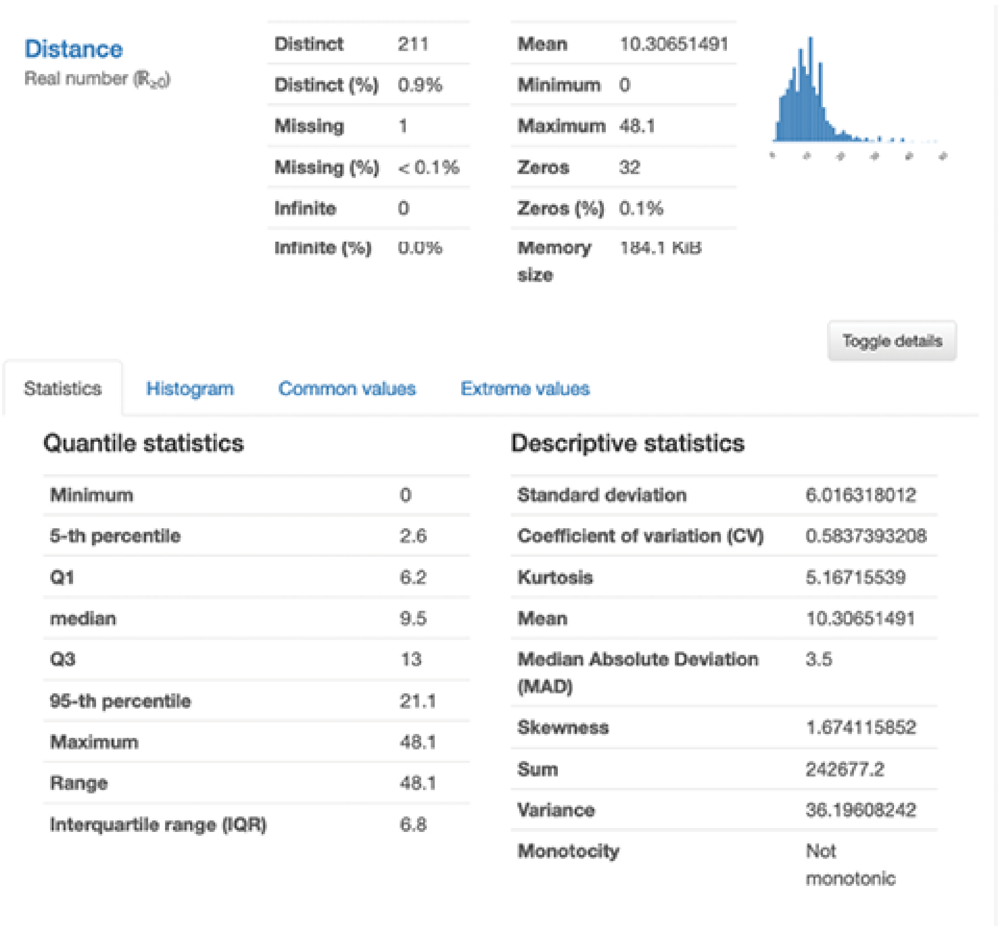
Distinct	23108
Distinct (%)	98.1%
Missing	0
Missing (%)	0.0%
Memory size	184.1 KiB

5 Charles St	5
25 William St	3
14 Rose St	3
13 Robinson St	3
3 Charles St	3
Other values (23103)	23530

Toggle details

Figure 2.6: Feature analysis

With every feature in the variable section comes a Toggle details button. If you click on that button, it will present you with detailed statistical information about the feature, as shown in the following screenshot:



Distance
Real number ($\mathbb{R}_{\geq 0}$)

Distinct	211	Mean	10.30651491
Distinct (%)	0.9%	Minimum	0
Missing	1	Maximum	48.1
Missing (%)	< 0.1%	Zeros	32
Infinite	0	Zeros (%)	0.1%
Infinite (%)	0.0%	Memory size	184.1 KiB

Quantile statistics

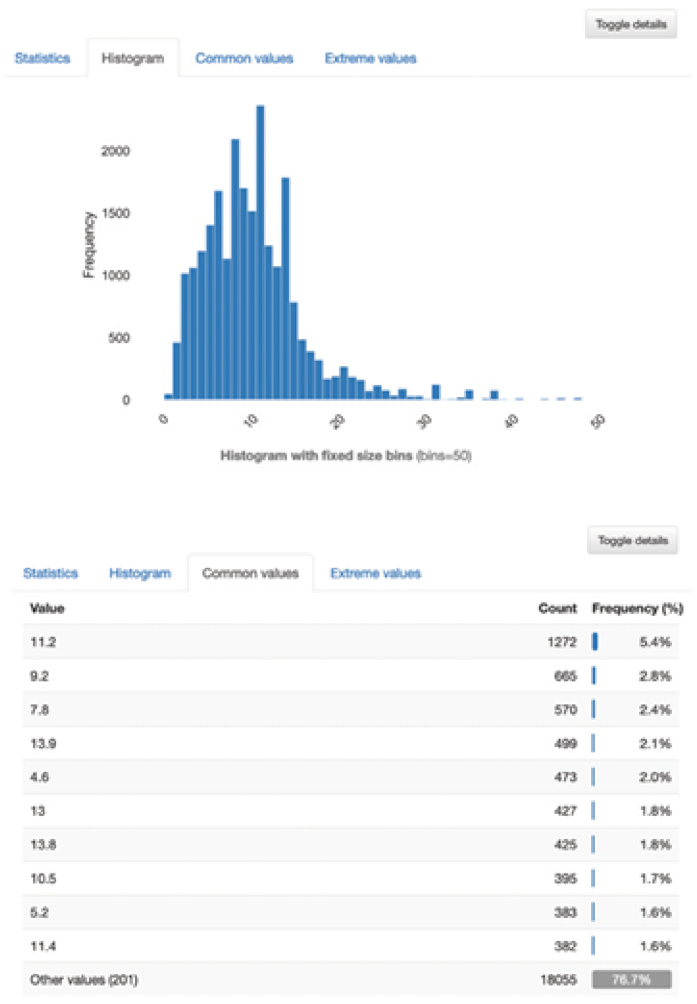
Minimum	0
5-th percentile	2.6
Q1	6.2
median	9.5
Q3	13
95-th percentile	21.1
Maximum	48.1
Range	48.1
Interquartile range (IQR)	6.8

Descriptive statistics

Standard deviation	6.016318012
Coefficient of variation (CV)	0.5837393208
Kurtosis	5.16715539
Mean	10.30651491
Median Absolute Deviation (MAD)	3.5
Skewness	1.674115852
Sum	242677.2
Variance	36.19608242
Monotocity	Not monotonic

Figure 2.7: Toggle details

Within those detailed sections, there are various tabs like Histogram and Common which provide an even more detailed analysis of the feature, as shown in the following screenshot:



Value	Count	Frequency (%)
11.2	1272	5.4%
9.2	665	2.8%
7.8	570	2.4%
13.9	499	2.1%
4.6	473	2.0%
13	427	1.8%
13.8	425	1.8%
10.5	395	1.7%
5.2	383	1.6%
11.4	382	1.6%
Other values (201)	18055	76.7%

Figure 2.8: Feature details

Further, the Interactions section provides a unique capability for users to select various combinations of features to analyze their correlation, as shown here:

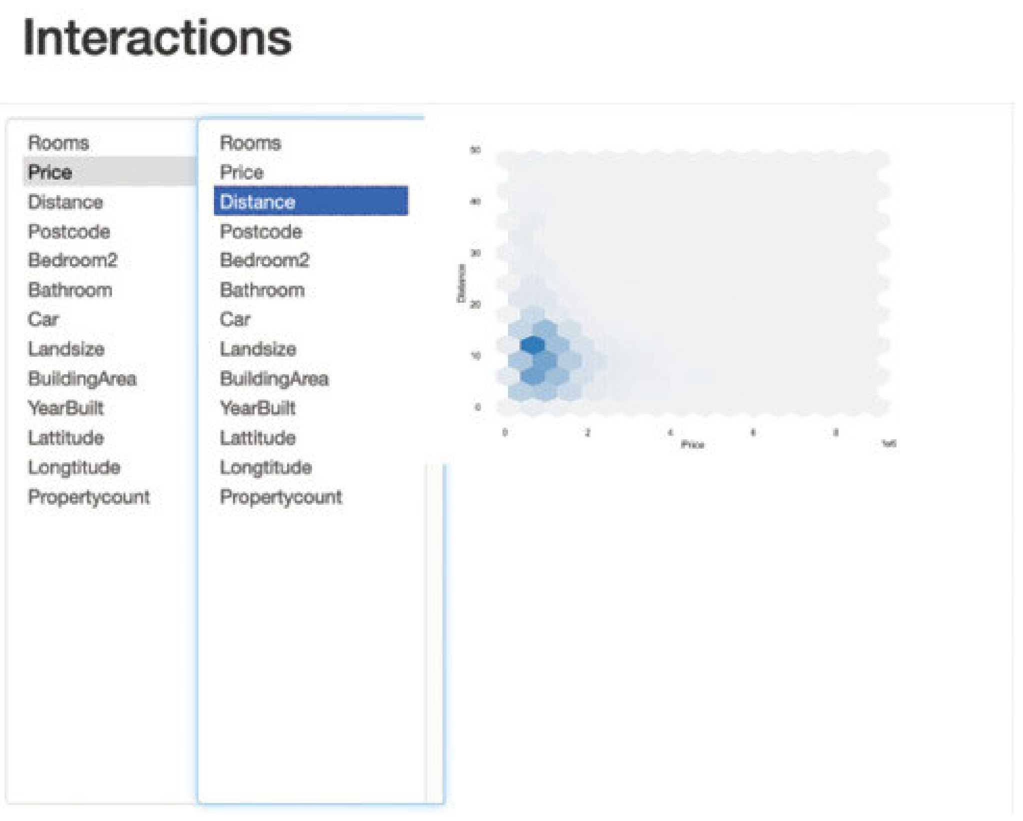


Figure 2.9: Bivariate analysis

However, the correlations section provides the holistic heatmap to show the correlations between every feature combination, as shown in the following screenshot:

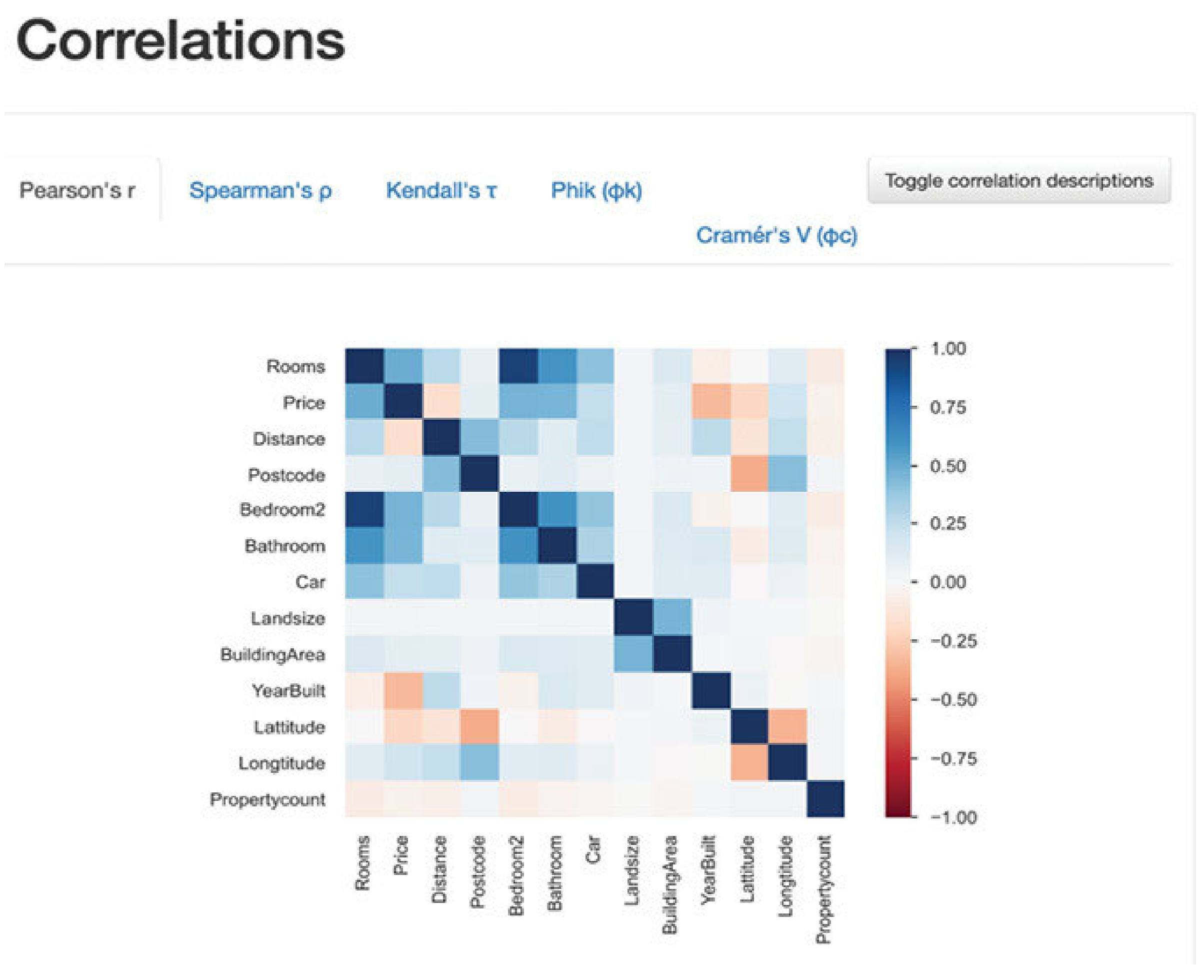


Figure 2.10: Correlations Heatmap

Saving the Pandas profile report to a HTML file

In order to broadcast your data findings, the pandas profile report can be exported into an HTML file, as follows:

```
from pandas_profiling import ProfileReport

df_profile = ProfileReport(df, title="Pandas Profiling Report")

df_profile.to_file("PandasProfiling_report.html")
```

The preceding code will generate and export the Pandas profile report into an HTML file and display the status of the export:

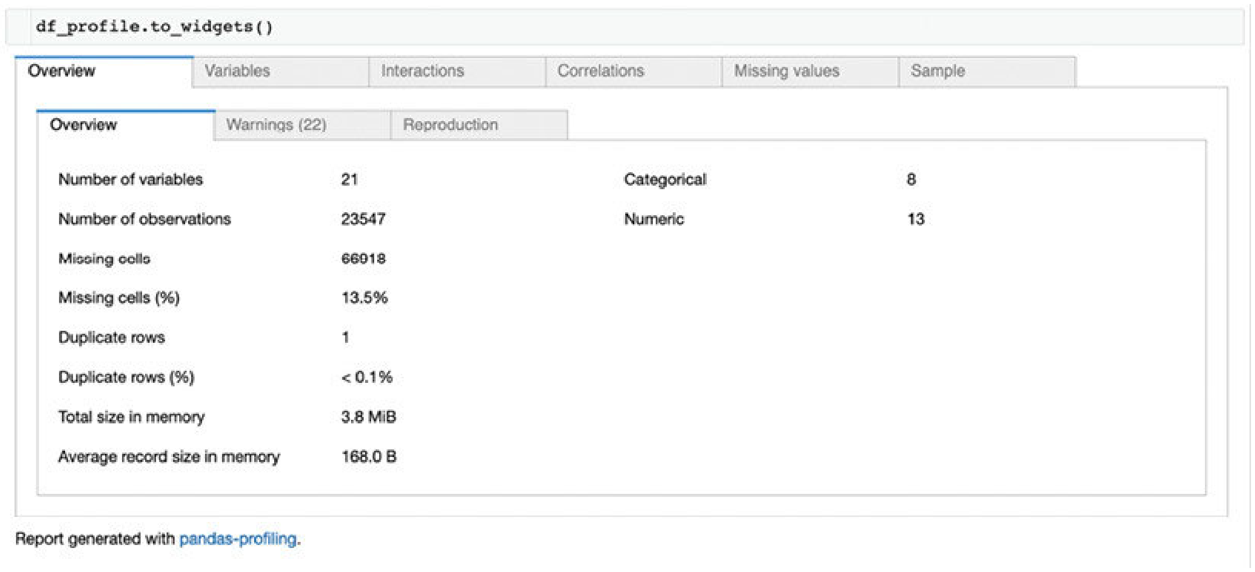
```
Export report to file: 100% 1/1 [00:00<00:00, 22.85it/s]
```

Figure 2.11: Export Report status

Creating a Jupyter Notebook widget

For better usability within Jupyter Notebook, pandas profile report can be converted into a widget within the notebook with this command:

df_profile.to_widgets()



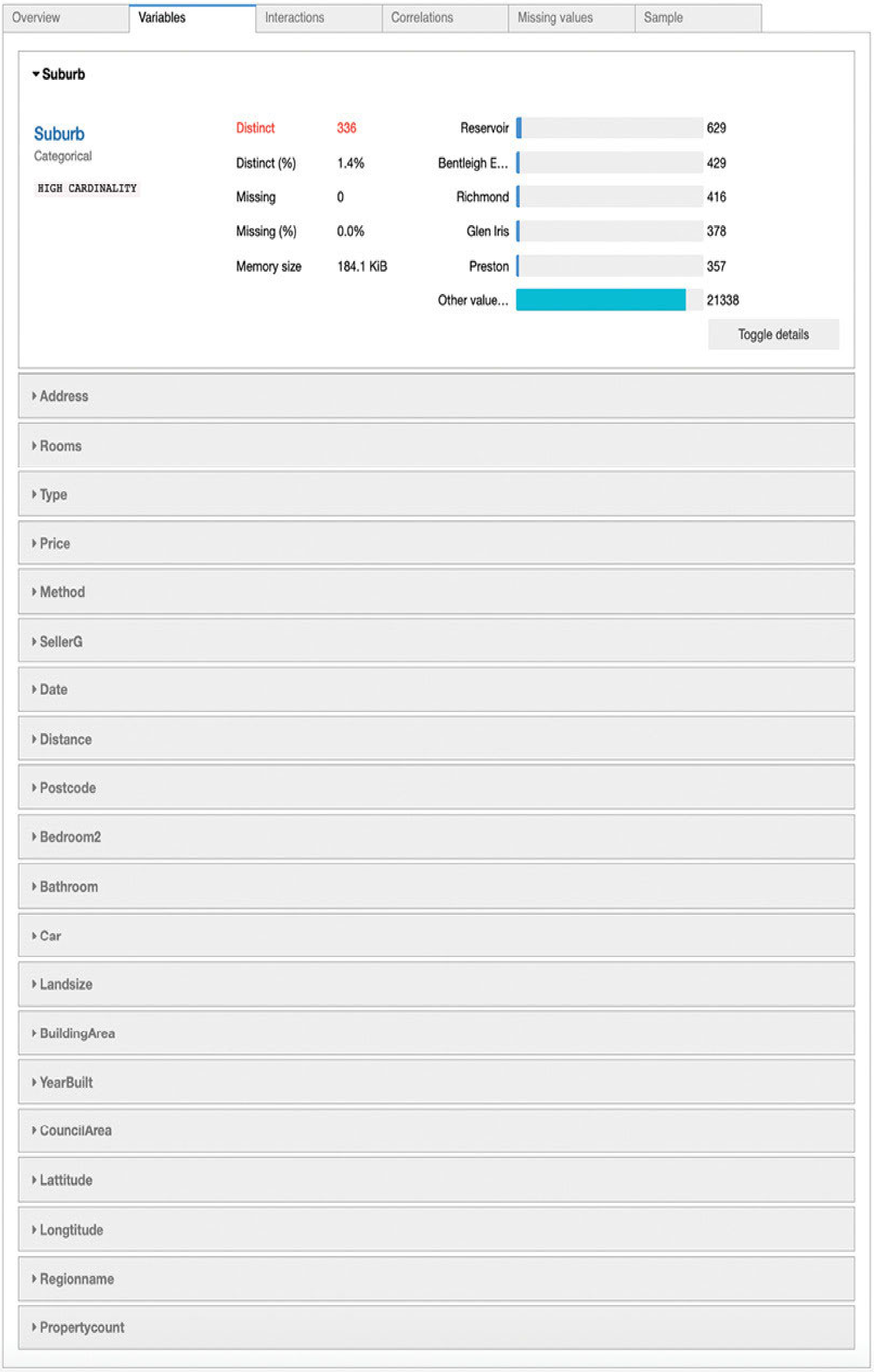
```
df_profile.to_widgets()
```

Number of variables	21	Categorical	8
Number of observations	23547	Numeric	13
Missing cells	66918		
Missing cells (%)	13.5%		
Duplicate rows	1		
Duplicate rows (%)	< 0.1%		
Total size in memory	3.8 MiB		
Average record size in memory	168.0 B		

Report generated with pandas-profiling.

Figure 2.12: Pandas Profile widget

Checking the variables view in the widget :



▾ Suburb

Suburb
Categorical
HIGH CARDINALITY

Distinct	336
Distinct (%)	1.4%
Missing	0
Missing (%)	0.0%
Memory size	184.1 KiB

Reservoir	629
Bentleigh E...	429
Richmond	416
Glen Iris	378
Preston	357
Other value...	21338

Figure 2.13: Feature analysis in widget view

Pandas profile report for big datasets

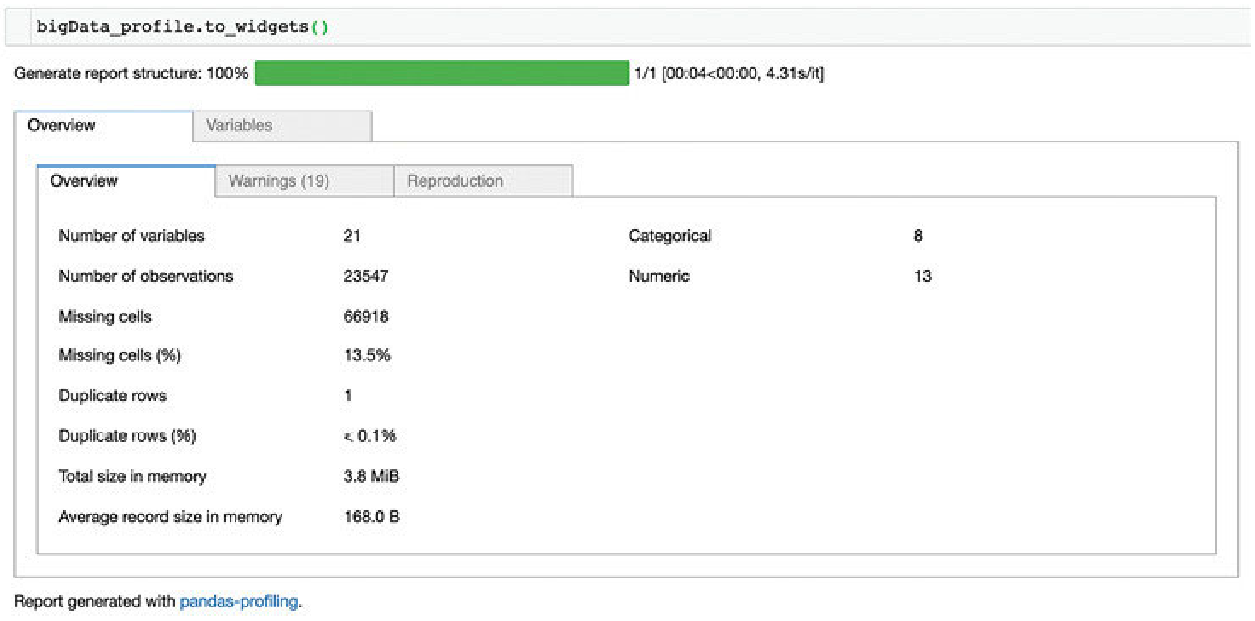
If you are dealing with huge datasets, then generating the full-fledged report would be quite time-consuming and sometimes unnecessary. Hence, the pandas profile provides you with an option to generate only a minimalistic report that gives a decent understanding of the dataset.

The following code will generate the profile report for huge datasets:

1. bigData_profile = ProfileReport(df, minimal=True)

1. bigData_profile.to_widgets()

With minimal option as true, you can see that all the tabs are not present in the report, as with the usual report. However, an overview of the dataset and details about all the features available in the dataset are generated in the report, as shown here:



```
bigData_profile.to_widgets()
```

```
Generate report structure: 100%  1/1 [00:04<00:00, 4.31s/it]
```

```
Number of variables            21          Categorical    8
Number of observations         23547       Numeric        13
Missing cells                  66918
Missing cells (%)              13.5%
Duplicate rows                 1
Duplicate rows (%)             < 0.1%
Total size in memory           3.8 MiB
Average record size in memory  168.0 B
```

Report generated with pandas-profiling.

Figure 2.14: Big data profile minimal report

To keep yourself updated about the latest improvements in the library, visit the official page of Pandas profiling:

Sweetviz

Sweetviz is a Python library that provides the complete data report in an interactive HTML file style-report. It does not only explore one dataset but is also capable of exploring two datasets in one shot and provide a well-defined comparison report between the two datasets, which is a very useful feature for certain use cases, which we will discuss in the next section.

Installing and getting started with Sweetviz

Follow the given steps to get started with Sweetviz:

Step 1: Install the Python package of Sweetviz with the following command:

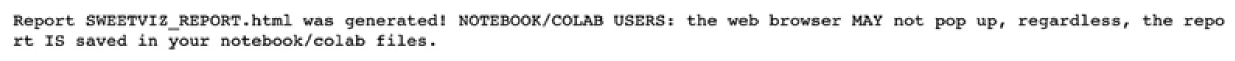
```
import sys

!{sys.executable} -m pip install -U sweetviz
```

Step 2: Import the sweetviz library, as follows:

```
import sweetviz as sv
```

Step 3: Call the analyze method in the Sweetviz library; that's all:

```
sv.analyze(df).show_html()
```



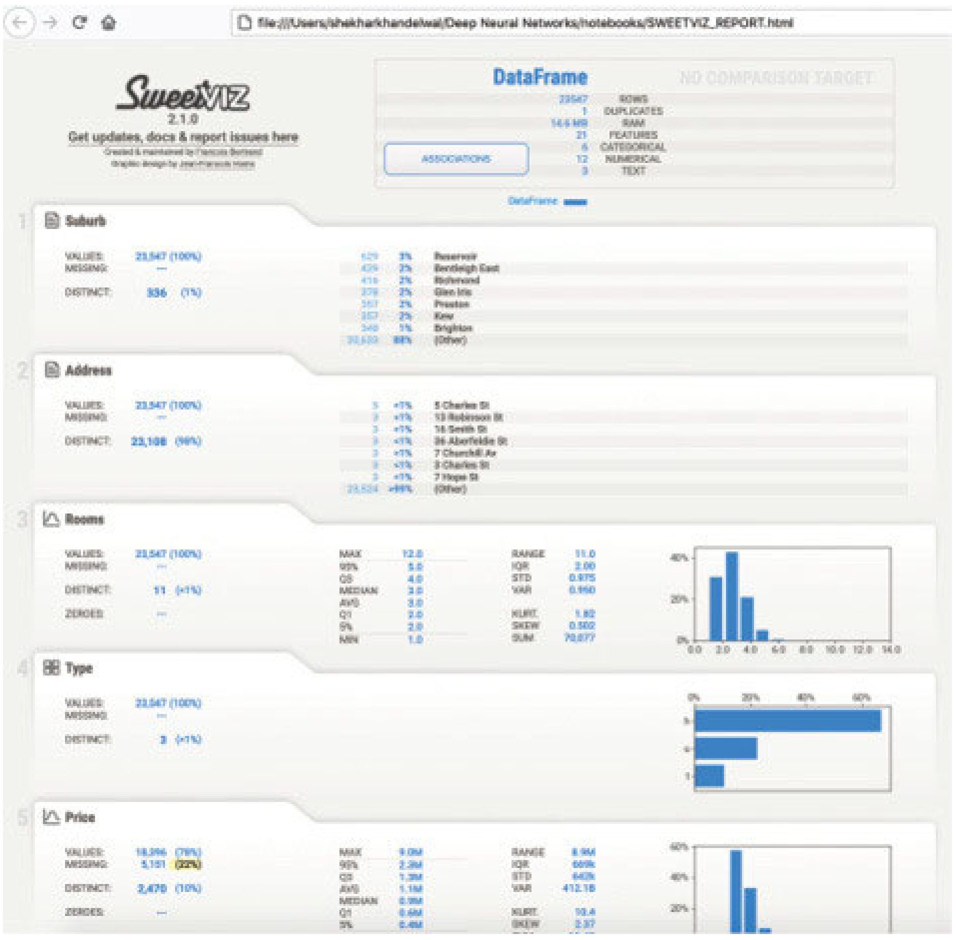
```
Report SWEETVIZ_REPORT.html was generated! NOTEBOOK/COLAB USERS: the web browser MAY not pop up, regardless, the repo
rt IS saved in your notebook/colab files.
```

Figure 2.15: Sweetviz report status

Analyzing Sweetviz report

The Sweetviz report HTML file will be saved automatically on your machine, and the report will also be automatically opened on your browser for you to navigate to.

A sample Sweetviz report is shown here:



file:///Users/shekharkhandelwal/Deep Neural Networks/notebooks/SWEETVIZ_REPORT.html

SweetVIZ 2.1.0
Get updates, docs & report issues here

DataFrame — NO COMPARISON TARGET

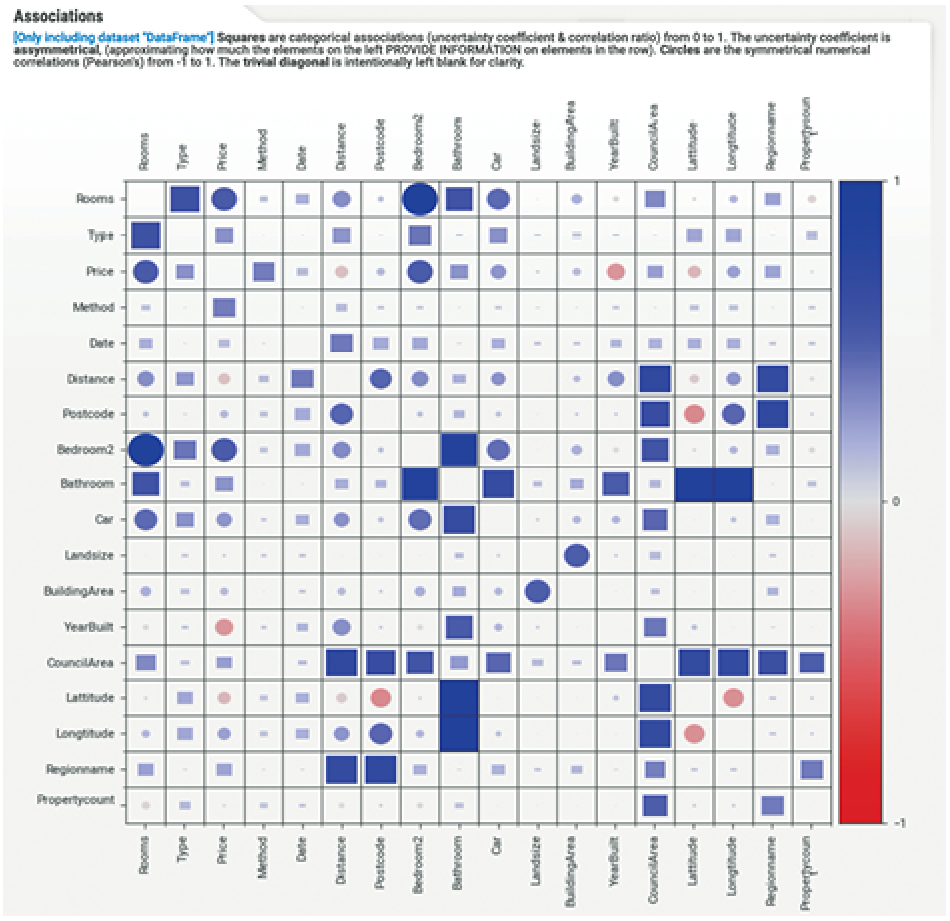
```
23547    ROWS
    1    DUPLICATES
14.6 MB  RAM
   21    FEATURES
    6    CATEGORICAL
   12    NUMERICAL
    3    TEXT
```

ASSOCIATIONS

```
1 Suburb
  VALUES:    23,547 (100%)        629     3%   Reservoir
  MISSING:   ---                  429     2%   Bentleigh East
                                  416     2%   Richmond
  DISTINCT:  336   (1%)           379     2%   Glen Iris
                                  367     2%   Preston
                                  357     2%   Kew
                                  340     1%   Brighton
                               20,633    88%   (Other)

2 Address
  VALUES:    23,547 (100%)          5    <1%   5 Charles St
  MISSING:   ---                    3    <1%   13 Robinson St
                                    3    <1%   16 Smith St
  DISTINCT:  23,108 (98%)           3    <1%   36 Aberfeldie St
                                    3    <1%   7 Churchill Av
                                    3    <1%   3 Charles St
                                    3    <1%   7 Hope St
                               23,524   >99%   (Other)

3 Rooms
  VALUES:    23,547 (100%)    MAX      12.0     RANGE   11.0
  MISSING:   ---              95%       5.0     IQR     2.00
                              Q3        4.0     STD     0.975
  DISTINCT:  11   (<1%)       MEDIAN    3.0     VAR     0.950
                              AVG       3.0
  ZEROES:    ---              Q1        2.0     KURT.   1.82
                              5%        2.0     SKEW    0.502
                              MIN       1.0     SUM     70,077

4 Type
  VALUES:    23,547 (100%)
  MISSING:   ---
  DISTINCT:  3   (<1%)

5 Price
  VALUES:    18,396  (78%)    MAX       9.0M    RANGE   8.9M
  MISSING:    5,151  (22%)    95%       2.3M    IQR     669k
                              Q3        1.3M    STD     642k
  DISTINCT:  2,470  (10%)     AVG       1.1M    VAR     412.1B
                              MEDIAN    0.9M
  ZEROES:    ---              Q1        0.6M    KURT.   10.4
                              5%        0.4M    SKEW    2.37
```

Figure 2.16: Sweeviz report

The report generated by Sweetviz is an interactive one. Clicking on the Associations tab at the top of the report will display the association of every variable with each other in the form of a heatmap in the right pane of the report, as shown in the following screenshot:

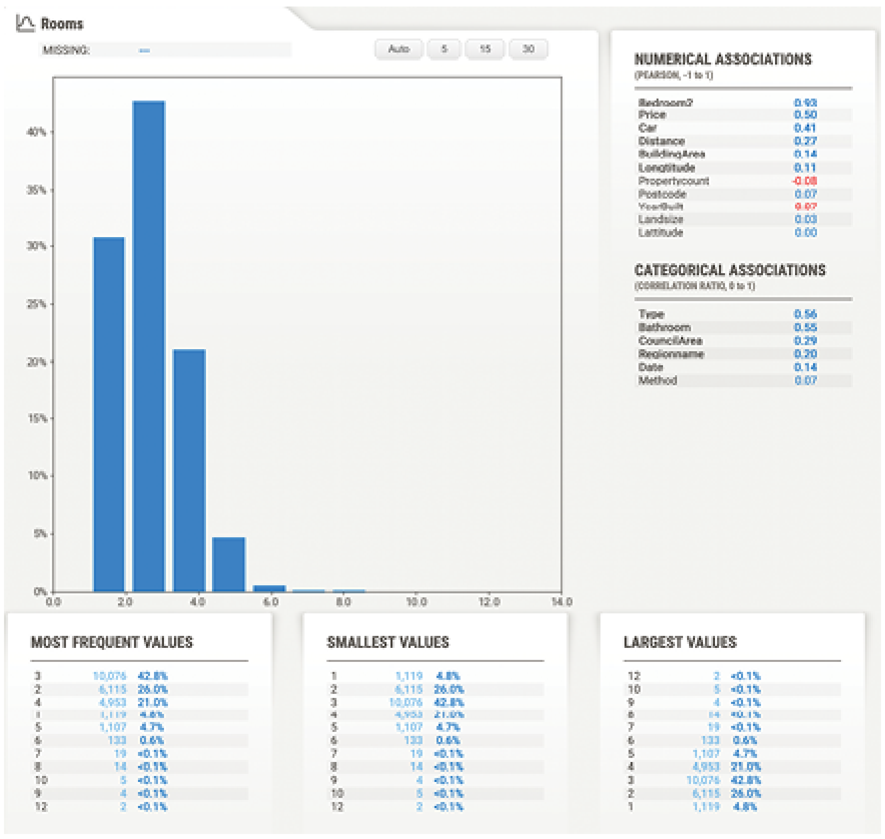


Associations

[Only including dataset "DataFrame"] **Squares** are categorical associations (uncertainty coefficient & correlation ratio) from 0 to 1. The uncertainty coefficient is **assymmetrical**, (approximating how much the elements on the left PROVIDE INFORMATION on elements in the row). **Circles** are the symmetrical numerical correlations (Pearson's) from -1 to 1. The **trivial diagonal** is intentionally left blank for clarity.

Figure 2.17: Sweetviz heatmap visualization

Similarly, clicking on the individual variable in the report will open a more detailed analysis of that variable in the right pane of the report, as you can see here:



NUMERICAL ASSOCIATIONS
(PEARSON, -1 to 1)

Bedroom2	0.93
Price	0.50
Car	0.41
Distance	0.27
BuildingArea	0.14
Longtitude	0.11
Propertycount	-0.08
Postcode	0.07
YearBuilt	-0.07
Landsize	0.03
Lattitude	0.00

CATEGORICAL ASSOCIATIONS
(CORRELATION RATIO, 0 to 1)

Type	0.56
Bathroom	0.55
CouncilArea	0.29
Regionname	0.20
Date	0.14
Method	0.07

MOST FREQUENT VALUES

3	10,076	42.8%
2	6,115	26.0%
4	4,953	21.0%
1	1,119	4.8%
5	1,107	4.7%
6	133	0.6%
7	19	<0.1%
8	14	<0.1%
10	5	<0.1%
9	4	<0.1%
12	2	<0.1%

SMALLEST VALUES

1	1,119	4.8%
2	6,115	26.0%
3	10,076	42.8%
4	4,953	21.0%
5	1,107	4.7%
6	133	0.6%
7	19	<0.1%
8	14	<0.1%
9	4	<0.1%
10	5	<0.1%
12	2	<0.1%

LARGEST VALUES

12	2	<0.1%
10	5	<0.1%
9	4	<0.1%
8	14	<0.1%
7	19	<0.1%
6	133	0.6%
5	1,107	4.7%
4	4,953	21.0%
3	10,076	42.8%
2	6,115	26.0%
1	1,119	4.8%

Figure 2.18: Individual feature analysis

Generating a report to compare two DataFrames using Sweetviz

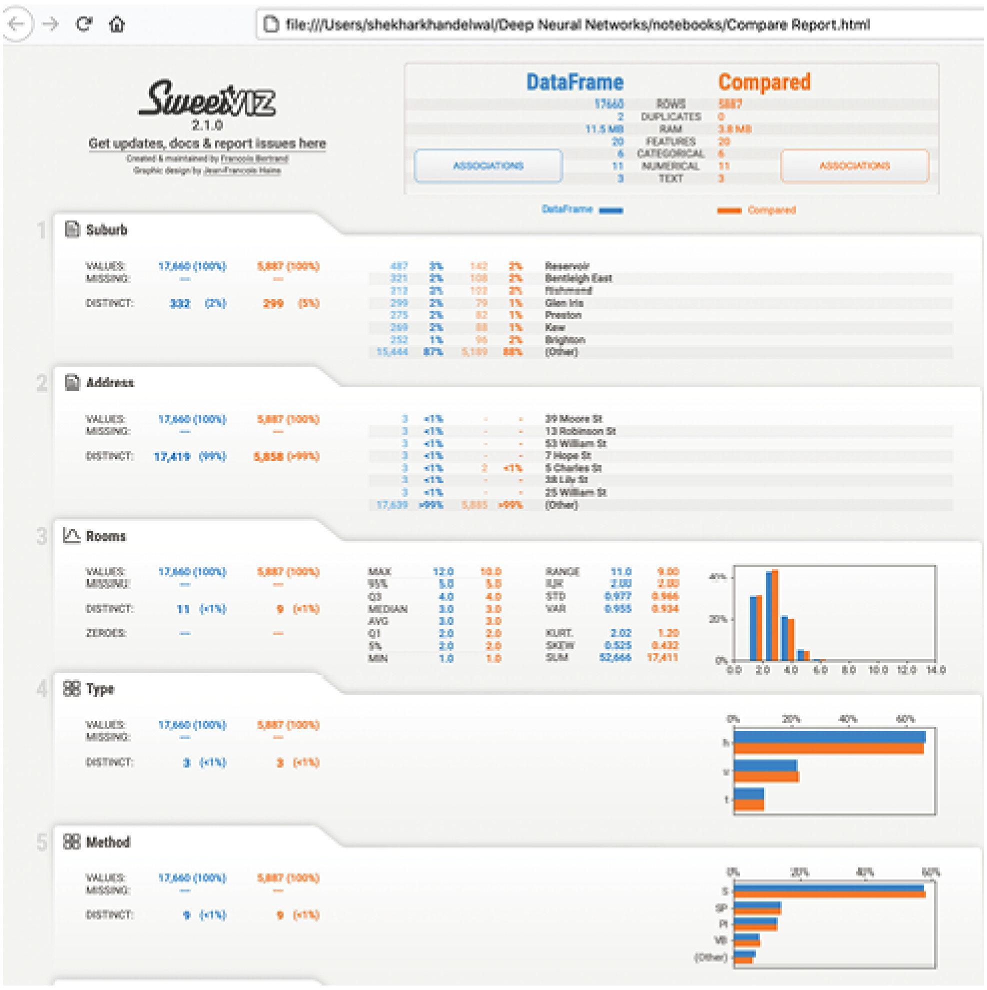
Sweetviz provides an option to compare two DataFrames and generate a detailed comparison report. This feature may come handy in situations when you have different datasets collected to build a predictive model, or when the DataFrame is divided into training and testing sets and you want to compare how the data is distributed.

Create two DataFrames, say df1 and df2, and use the compare method by passing the two DataFrames as an argument to the method. Further, use the show_html method to generate the comparison report:

```
df_compare = sv.compare(df1, df2)

df_compare.show_html('Compare Report.html')
```

This will generate an HTML report that consists of all metrics of comparison in the report.

A sample comparison report is shown in the following screenshot:



```
file:///Users/shekharkhandelwal/Deep Neural Networks/notebooks/Compare Report.html

SweetVIZ
2.1.0
Get updates, docs & report issues here
Created & maintained by Francois Bertrand
Graphic design by Jean-Francois Hains

                 DataFrame              Compared
                     17660  ROWS         5887
                         2  DUPLICATES   0
                   11.5 MB  RAM          3.8 MB
                        20  FEATURES     20
ASSOCIATIONS             6  CATEGORICAL  6          ASSOCIATIONS
                        11  NUMERICAL    11
                         3  TEXT         3

                 DataFrame              Compared

1  Suburb
   VALUES:    17,660 (100%)   5,887 (100%)      487   3%    142   2%   Reservoir
   MISSING:        —               —            321   2%    108   2%   Bentleigh East
                                                312   2%    103   2%   Richmond
   DISTINCT:    332   (2%)      299   (5%)      299   2%     79   1%   Glen Iris
                                                275   2%     82   1%   Preston
                                                269   2%     88   1%   Kew
                                                252   1%     96   2%   Brighton
                                             15,444  87%  5,189  88%   (Other)

2  Address
   VALUES:    17,660 (100%)   5,887 (100%)        3  <1%      -    -   39 Moore St
   MISSING:        —               —              3  <1%      -    -   13 Robinson St
                                                  3  <1%      -    -   53 William St
   DISTINCT:  17,419  (99%)   5,858 (>99%)        3  <1%      -    -   7 Hope St
                                                  3  <1%      2  <1%   5 Charles St
                                                  3  <1%      -    -   38 Lily St
                                                  3  <1%      -    -   25 William St
                                             17,639 >99%  5,885 >99%   (Other)

3  Rooms
   VALUES:    17,660 (100%)   5,887 (100%)   MAX     12.0  10.0   RANGE   11.0    9.00
   MISSING:        —               —         95%      5.0   5.0   IQR     2.00    2.00
                                             Q3       4.0   4.0   STD     0.977   0.966
   DISTINCT:     11  (<1%)        9  (<1%)   MEDIAN   3.0   3.0   VAR     0.955   0.934
                                             AVG      3.0   3.0
   ZEROES:         —               —         Q1       2.0   2.0   KURT.   2.02    1.20
                                             5%       2.0   2.0   SKEW    0.525   0.432
                                             MIN      1.0   1.0   SUM     52,666  17,411

4  Type
   VALUES:    17,660 (100%)   5,887 (100%)
   MISSING:        —               —

   DISTINCT:      3  (<1%)        3  (<1%)

5  Method
   VALUES:    17,660 (100%)   5,887 (100%)
   MISSING:        —               —

   DISTINCT:      9  (<1%)        9  (<1%)
```

Figure 2.19: AutoViz report

To keep yourself updated on the latest improvements in the library, visit the official page of Sweetviz:

AutoViz

AutoViz is a Python library that enables users to create different kinds of visualizations that can be generated for the variables in the dataset. This can be considered as a starting point of data analysis through visualization. With just one line of code, a huge number of plots are generated for careful analysis of the variables in the dataset, visually.

Installing and getting started with AutoViz

Following are the steps to install and get started with AutoViz:

Step 1: Install the autoviz Python package with the following command:

```
pip install autoviz
```

Step 2: Import the AutoViz class from the autoviz library, as shown here:

```
from autoviz.AutoViz_Class import AutoViz_Class
```

Step 3: Create an object for the AutoViz class with the given command:

```
AV = AutoViz_Class()
```

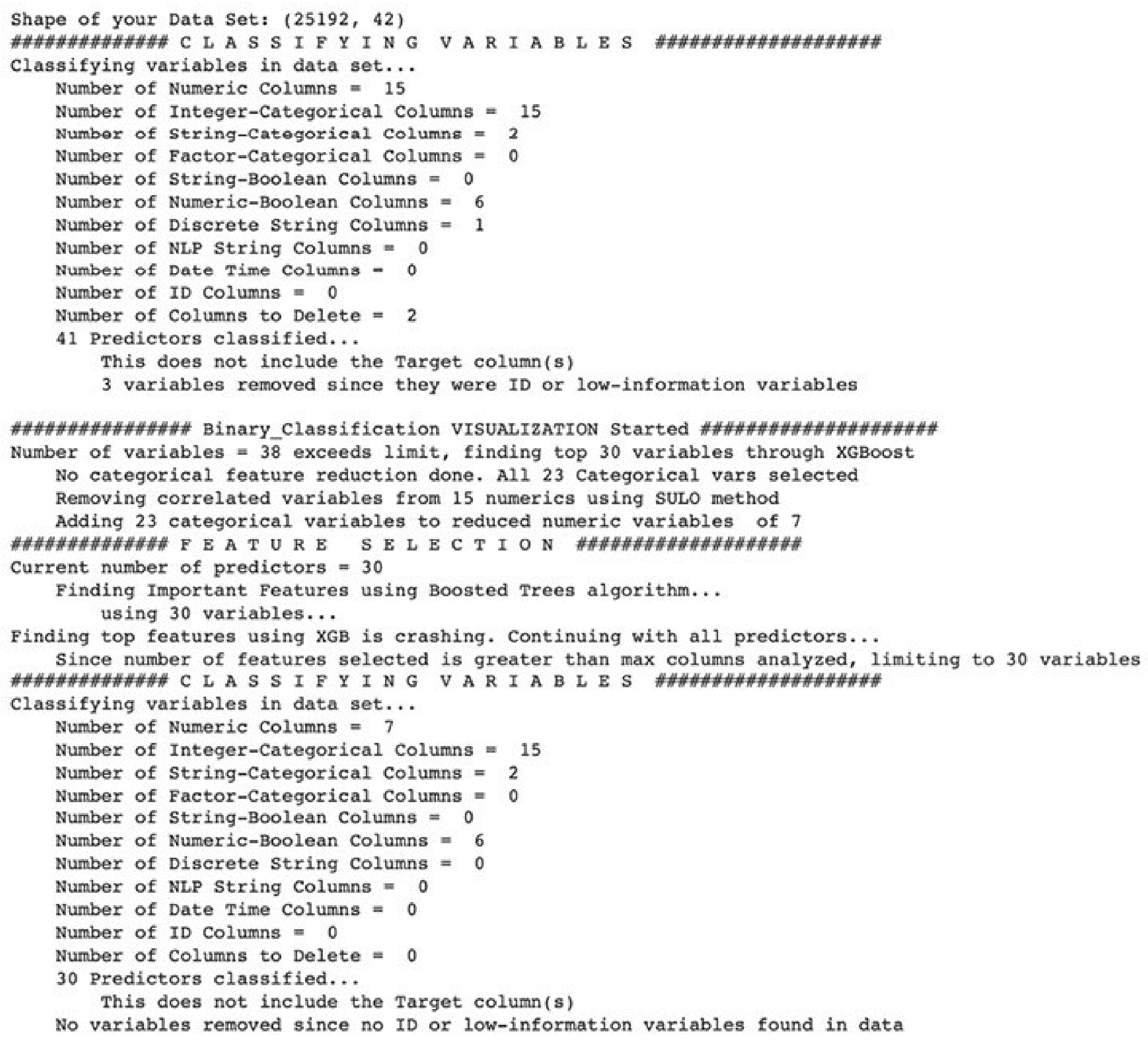
Step 4: Pass the datafile and the dependent variable name in the AutoViz method in the AutoViz class object, as shown here:

```
df = AV.AutoViz('Train_data.csv', depVar="class")
```

This will generate the AutoViz report with all the visualization possible, depending upon the features within the dataset.

Analyzing AutoViz report

First, the analysis of the dataset will be provided as the output, as follows:



```
Shape of your Data Set: (25192, 42)
############## C L A S S I F Y I N G  V A R I A B L E S  ####################
Classifying variables in data set...
    Number of Numeric Columns =  15
    Number of Integer-Categorical Columns =  15
    Number of String-Categorical Columns =  2
    Number of Factor-Categorical Columns =  0
    Number of String-Boolean Columns =  0
    Number of Numeric-Boolean Columns =  6
    Number of Discrete String Columns =  1
    Number of NLP String Columns =  0
    Number of Date Time Columns =  0
    Number of ID Columns =  0
    Number of Columns to Delete =  2
    41 Predictors classified...
        This does not include the Target column(s)
        3 variables removed since they were ID or low-information variables

################ Binary_Classification VISUALIZATION Started #####################
Number of variables = 38 exceeds limit, finding top 30 variables through XGBoost
    No categorical feature reduction done. All 23 Categorical vars selected
    Removing correlated variables from 15 numerics using SULO method
    Adding 23 categorical variables to reduced numeric variables  of 7
############## F E A T U R E   S E L E C T I O N  ####################
Current number of predictors = 30
    Finding Important Features using Boosted Trees algorithm...
        using 30 variables...
Finding top features using XGB is crashing. Continuing with all predictors...
    Since number of features selected is greater than max columns analyzed, limiting to 30 variables
############## C L A S S I F Y I N G  V A R I A B L E S  ####################
Classifying variables in data set...
    Number of Numeric Columns =  7
    Number of Integer-Categorical Columns =  15
    Number of String-Categorical Columns =  2
    Number of Factor-Categorical Columns =  0
    Number of String-Boolean Columns =  0
    Number of Numeric-Boolean Columns =  6
    Number of Discrete String Columns =  0
    Number of NLP String Columns =  0
    Number of Date Time Columns =  0
    Number of ID Columns =  0
    Number of Columns to Delete =  0
    30 Predictors classified...
        This does not include the Target column(s)
    No variables removed since no ID or low-information variables found in data
```

Figure 2.20: Autoviz report

Further, the visualizations will be presented in the report.

Since the dependent variable is the 'class' column, all the variables are plotted on a scatter plot against class feature, as seen in the following screenshot:

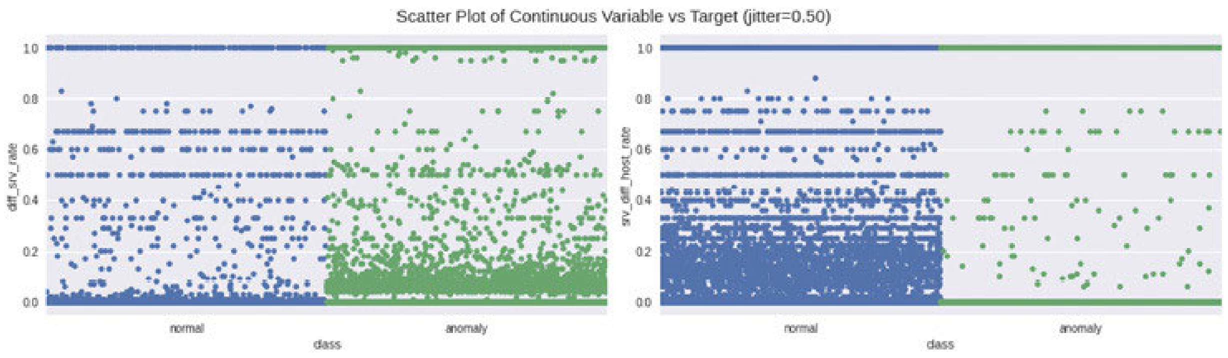


Figure 2.21: Continuous variable scatter plot

The report further contains all the different kinds of plots that can be created based on the feature data types, like scatter plot, box plot, violin plot, heatmaps and so on, as shown in the following screenshot:

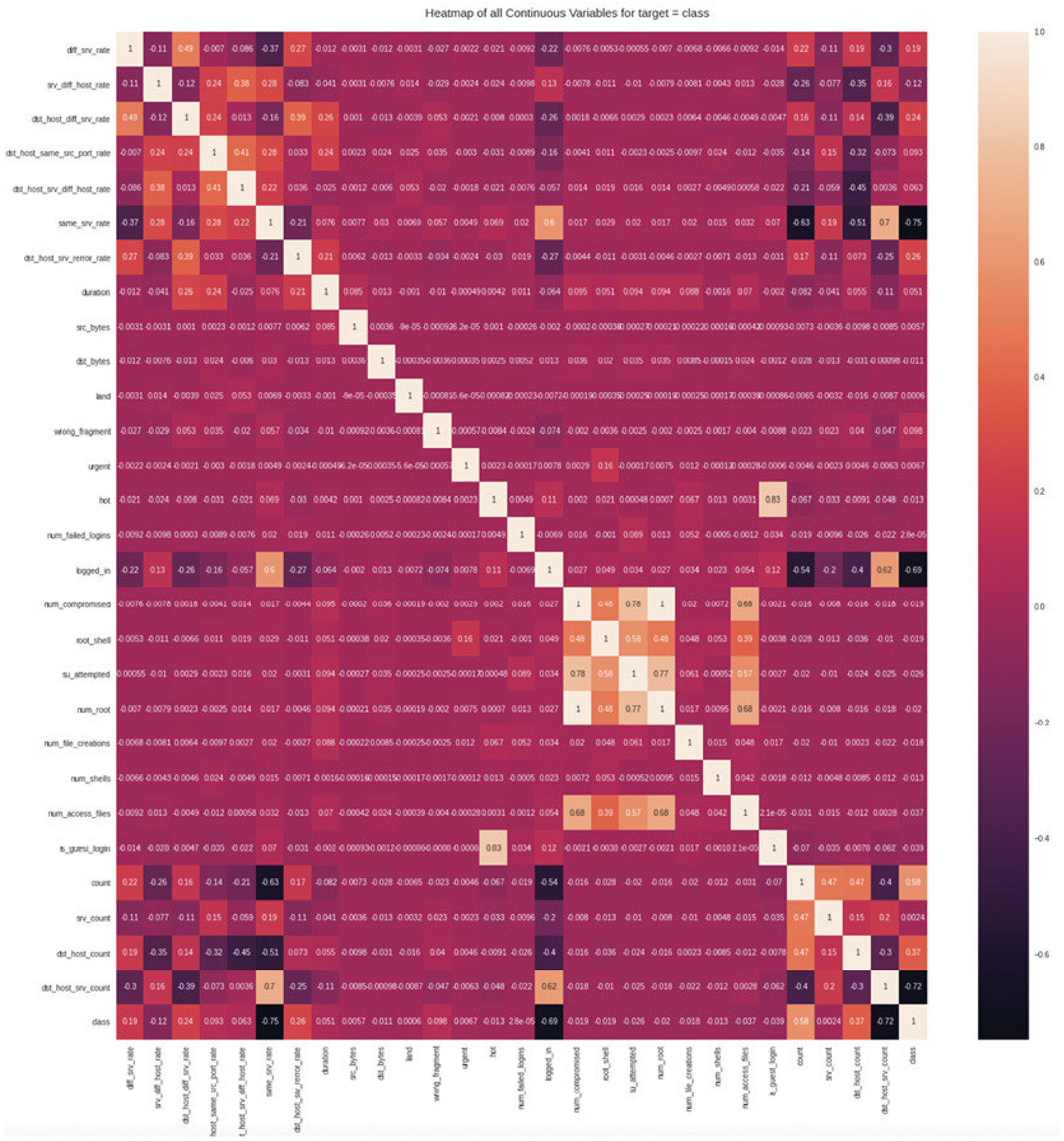


Figure Autoviz heatmap

As can be seen here, with just one line of code, a huge number of plots can be generated using the Autoviz library to analyze data visually, before proceeding with more detailed exploratory data analysis.

To keep yourself updated on the latest improvements in the library, visit the official page of Autoviz:

Lux

Lux is an advanced data visualization library that can be integrated with a Jupyter Notebook as a widget. It enables the notebook cells to display toggle buttons and to display on-demand plots for a given DataFrame if the user chooses to click on the displayed toggle buttons on the notebook output cell.

Installing and getting started with Lux

Here are the steps to install and get started with Lux:

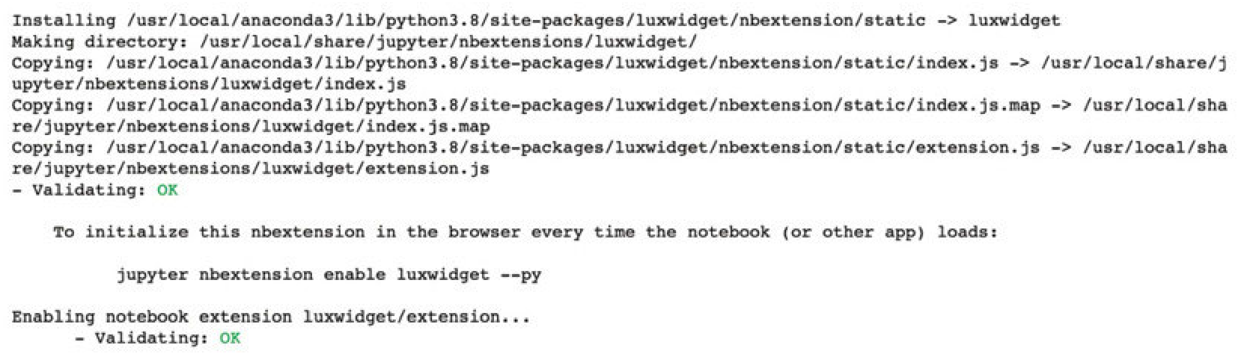
Step 1: Install lux package using pip

import sys

!{sys.executable} -m pip install -U lux-api

Step 2: In order to enable Lux widget in Jupyter Notebook, luxwidget extensions need to be installed and enabled, as follows:

The output obtained is as shown here:



```
Installing /usr/local/anaconda3/lib/python3.8/site-packages/luxwidget/nbextension/static -> luxwidget
Making directory: /usr/local/share/jupyter/nbextensions/luxwidget/
Copying: /usr/local/anaconda3/lib/python3.8/site-packages/luxwidget/nbextension/static/index.js -> /usr/local/share/j
upyter/nbextensions/luxwidget/index.js
Copying: /usr/local/anaconda3/lib/python3.8/site-packages/luxwidget/nbextension/static/index.js.map -> /usr/local/sha
re/jupyter/nbextensions/luxwidget/index.js.map
Copying: /usr/local/anaconda3/lib/python3.8/site-packages/luxwidget/nbextension/static/extension.js -> /usr/local/sha
re/jupyter/nbextensions/luxwidget/extension.js
- Validating: OK

    To initialize this nbextension in the browser every time the notebook (or other app) loads:

          jupyter nbextension enable luxwidget --py

Enabling notebook extension luxwidget/extension...
      - Validating: OK
```

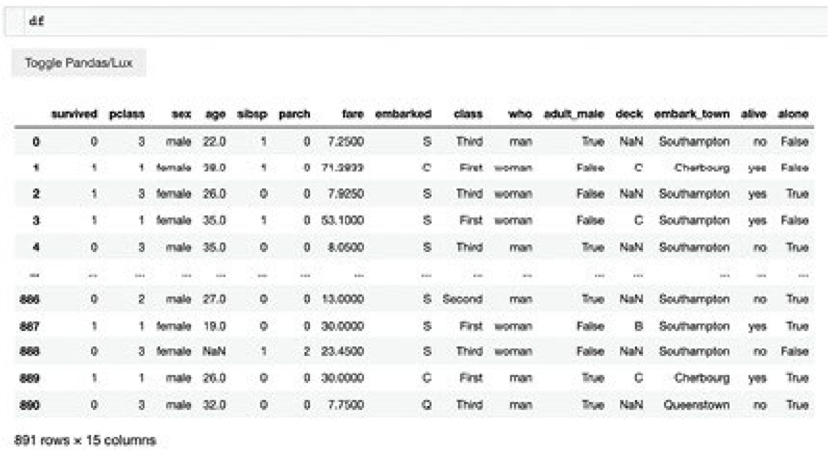
Figure 2.23: Extension install status

Step 3: For demonstration purposes, let’s create a DataFrame using one of the dataset available in seaborn sample data, with the following code:

import seaborn as sns

df=sns.load_dataset('titanic')

Step 4: Upon the display of the DataFrame on the notebook cell, in addition to the content of the DataFrame, a Toggle Pandas/Lux button will also be displayed now:



```
df
```

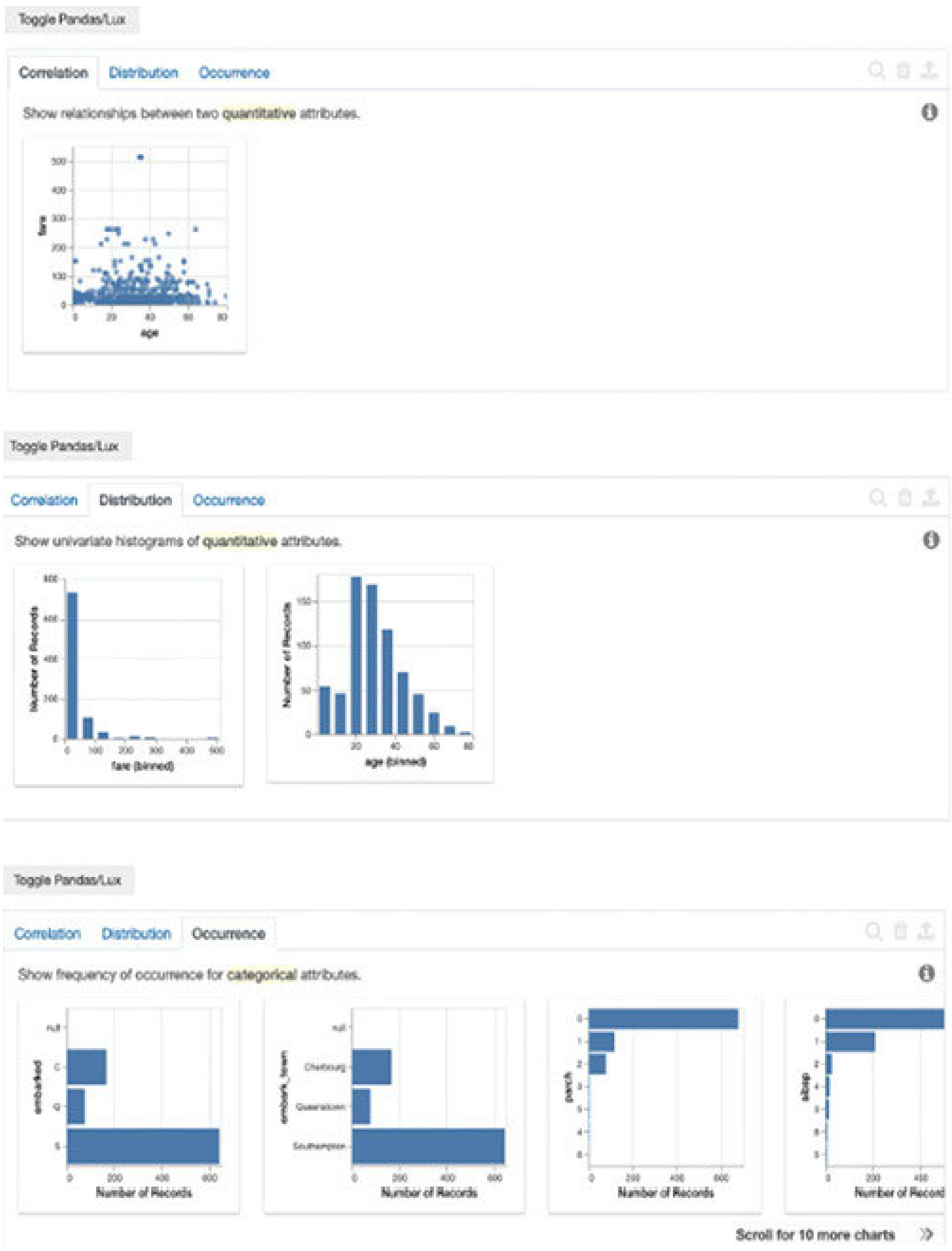
	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
886	0	2	male	27.0	0	0	13.0000	S	Second	man	True	NaN	Southampton	no	True
887	1	1	female	19.0	0	0	30.0000	S	First	woman	False	B	Southampton	yes	True
888	0	3	female	NaN	1	2	23.4500	S	Third	woman	False	NaN	Southampton	no	False
889	1	1	male	26.0	0	0	30.0000	C	First	man	True	C	Cherbourg	yes	True
890	0	3	male	32.0	0	0	7.7500	Q	Third	man	True	NaN	Queenstown	no	True

891 rows × 15 columns

Figure 2.24: DataFrame in Lux analysis

Analyzing Lux report

Upon clicking on the toggle button, Lux visualization will be displayed within the notebook cell. The report contains various tabs like and as shown in the following screenshot:



Show relationships between two quantitative attributes.

Show univariate histograms of quantitative attributes.

Show frequency of occurrence for categorical attributes.

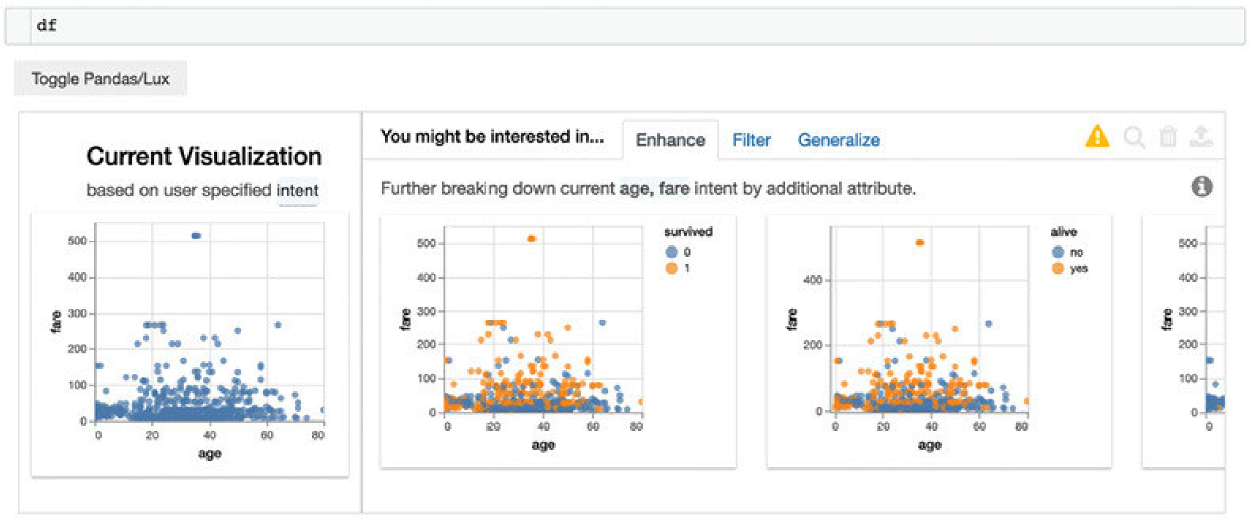
Figure 2.25: LUX visualizations

Generating Lux visualizations based on intent

Lux also provides the flexibility to generate intent-based visualizations. For example, if you want to generate plots against selected features within the dataset, then you can create an intent with the following code:

df.intent=['age’, 'fare']

Once the intent is created, when the DataFrame is displayed on the notebook cell, the visualizations are automatically created based on the intent defined against the DataFrame, as shown in the following screenshot:



```
df
```

Toggle Pandas/Lux
Current Visualization
based on user specified intent
You might be interested in...
Enhance
Filter
Generalize
Further breaking down current age, fare intent by additional attribute.
fare
age
survived
0
1
alive
no
yes

Figure 2.26: LUX intent visualization

Another distinguishable feature of Lux is that in addition to the intended plots, it recommends enhancement of the plots by adding another variable against the intended variables to analyze the correlation of all of these variables, as seen here:

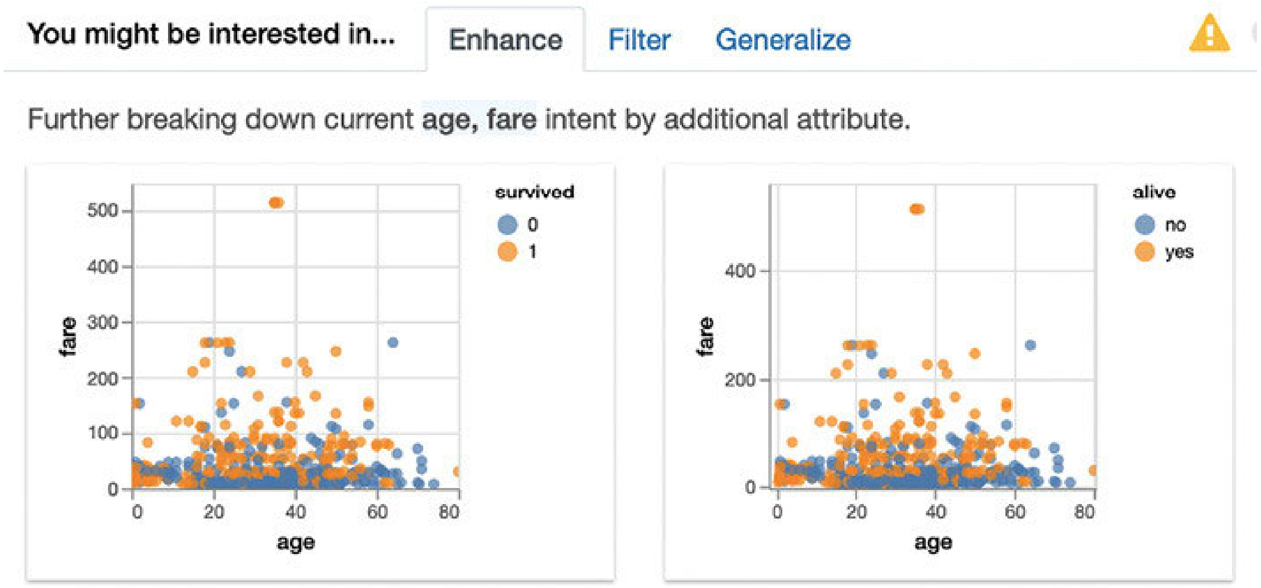

You might be interested in...
Enhance
Filter
Generalize
Further breaking down current age, fare intent by additional attribute.
survived
0
1
fare
age
alive
no
yes
fare
age

Figure 2.27: Lux advance intent visualization

Also, note that all these plots are interactive, and the cursor can be used to hover on various data points to get the details of those data points right there on the plot, as shown in the following screenshot:

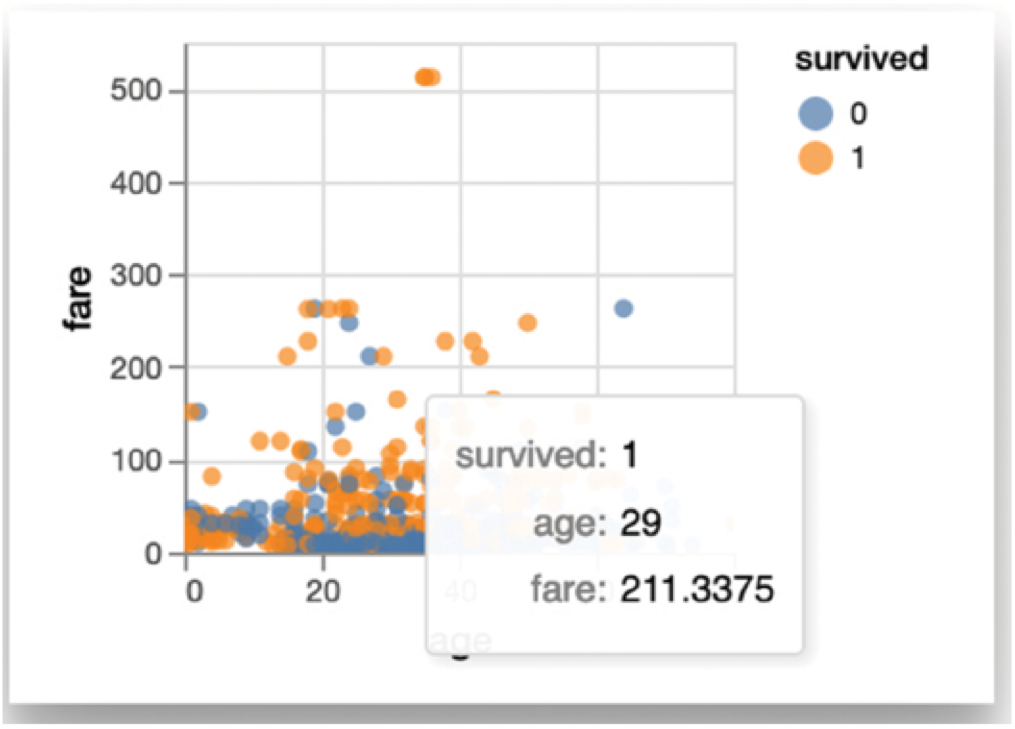


Figure 2.28: LUX visual features

Additionally, all the plots displayed can be downloaded using the Export feature, as shown in the following screenshot:

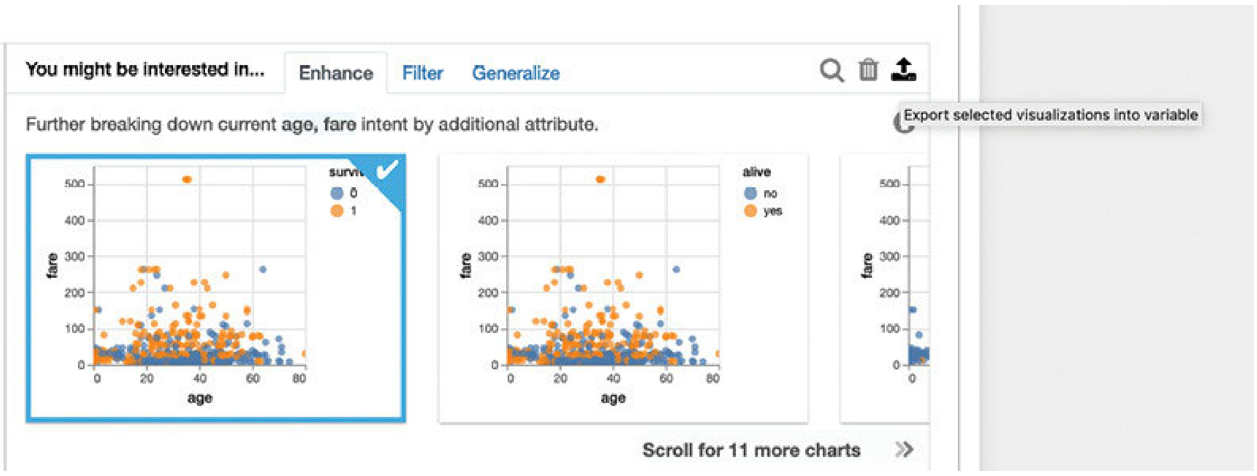

You might be interested in...
Enhance
Filter
Generalize
Further breaking down current age, fare intent by additional attribute.
Export selected visualizations into variable
survived
0
1
alive
no
yes
fare
age
Scroll for 11 more charts

Figure 2.29: LUX report export

Saving Lux report to a HTML file

LUX reports can be saved as an HTML file for sharing purposes, with the following code:

```
df.save_as_html('lux_report.html')
```

The output obtained is as follows:

```
Saved HTML to lux_report.html
```

Figure 2.30: Save output message

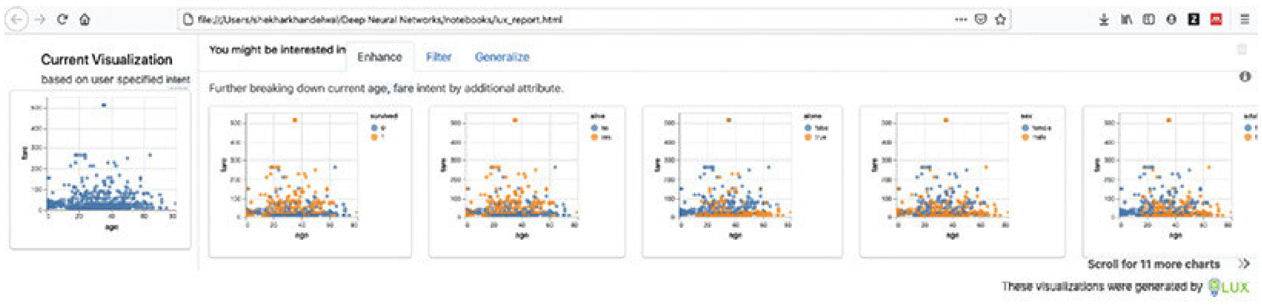

file:///Users/shekharkhandelwal/Deep Neural Networks/notebooks/lux_report.html
Current Visualization
based on user specified intent
You might be interested in
Enhance
Filter
Generalize
Further breaking down current age, fare intent by additional attribute.
Scroll for 11 more charts
These visualizations were generated by LUX

Figure 2.31: LUX export

To clear the intent from the DataFrame, use the following command:

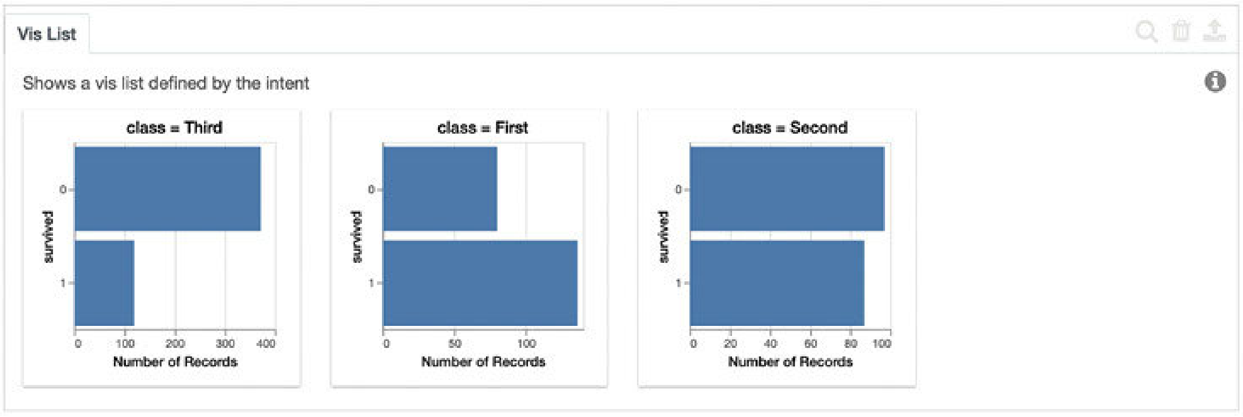
```
df.clear_intent()
```

Advanced features in Lux reports

Lux plots can be further optimized by using classes like VisList. For example, if user wants to see distribution of data based on one variable against all distinct values of another variable, we will write the following code:

```
from lux.vis.VisList import VisList

VisList(['class=?', 'survived'], df)
```

This will display plots of records distribution based on the survived feature for all the different values of the class feature, as shown here:



```
[<Vis  (x: COUNT(Record), y: survived  -- [class=Third]  ) mark: bar, score: 0.00 >,
 <Vis  (x: COUNT(Record), y: survived  -- [class=First]  ) mark: bar, score: 0.00 >,
 <Vis  (x: COUNT(Record), y: survived  -- [class=Second] ) mark: bar, score: 0.00 >]
```

Figure 2.32: LUX advance

To keep yourself updated on the latest improvements in the library, visit the official page of LUX:

Lazy Predict

While working on a data science problem statement, wherein you are supposed to build a predictive model, whether regression or classification, there is always a limitation on the number of experiments that can be conducted in terms of algorithms implementations and comparing the results.

It is usually based on your experience and instinct; you pick a certain algorithm, let us say random forest classifier for a classification use case, and implement it on your dataset. Then, you consider the performance metrics of the model created using random forest as a baseline and move on to tune your model for better performance.

Lazy Predict is a library that enables users to experiment their dataset on almost every popular algorithm available for regression and classification problems, with just a few lines of code.

This expands the horizon of the experiment that a user conducts in terms of algorithms implementations and gives a confidence on setting the baseline performance metrics, which the user can use to build upon for further improvements.

Installing and getting started with Lazy Predict

Follow the given steps to install and get started with Lazy Predict:

Step 1: Install the lazypredict Python library using pip

Step 2: Import the LazyClassifier and LazyRegressor classes, as shown here:

```
from lazypredict.Supervised import LazyClassifier, LazyRegressor
```

Step 3: Import and load the dataset within the sklearn library to create a DataFrame for demonstration purposes, with the following code:

```
from sklearn import datasets

from sklearn.model_selection import train_test_split

data = datasets.load_breast_cancer()
```

Step 4: Distribute DataFrame into train and test sets using the train_test_split method from the sklearn library, as shown here:

```
X, y = data.data, data.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.2, rando
m_state=42)
```

Step 5: Create an object for the LazyClassifier class:

```
clf = LazyClassifier(predictions=True)
```

Step 6: Use the fit method within the LazyClassifier class, and pass train and test DataFrames and as well as the associated labeled DataFrames and to the fit method. That will return two objects: models and predictions:

```
models, predictions = clf.fit(X_train, X_test, y_train, y_test)
```

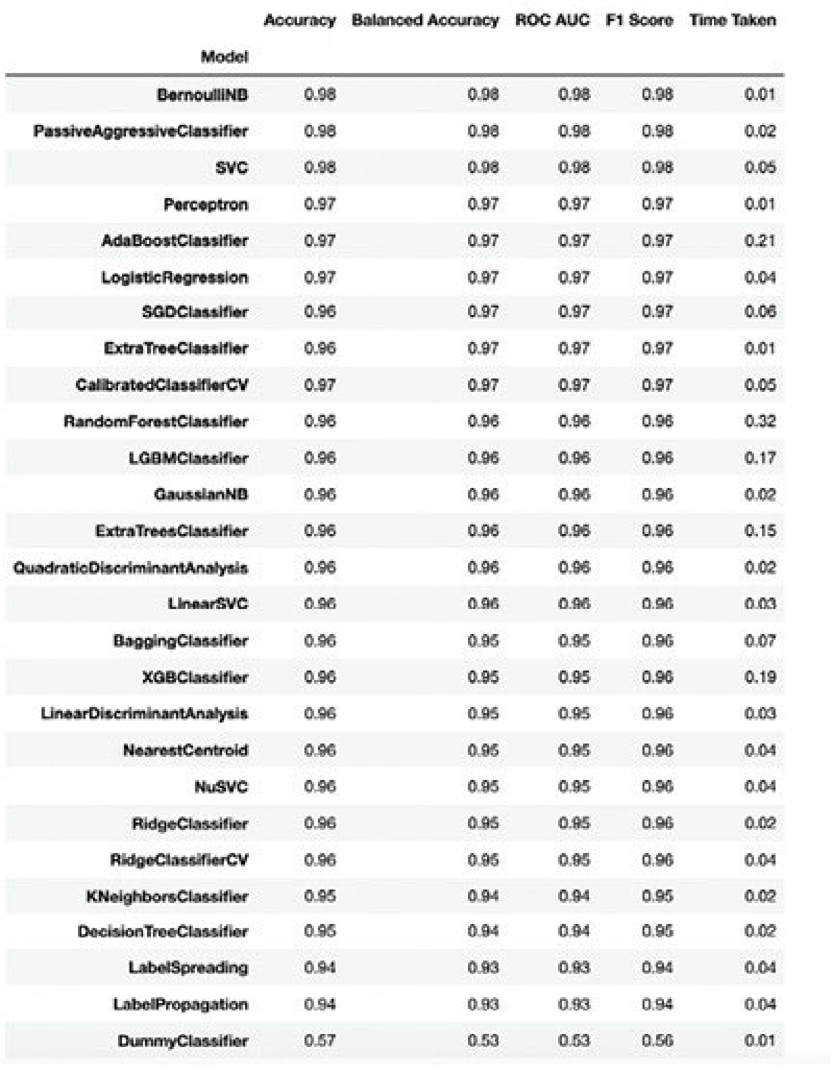
Analyzing Lazy Predict experimentation results

Models will contain the comparison of all the algorithms implemented on the DataFrame passed to the fit method for classification.

Let us print the models' content and see for ourselves:

```
models
```

The output obtained is as follows:



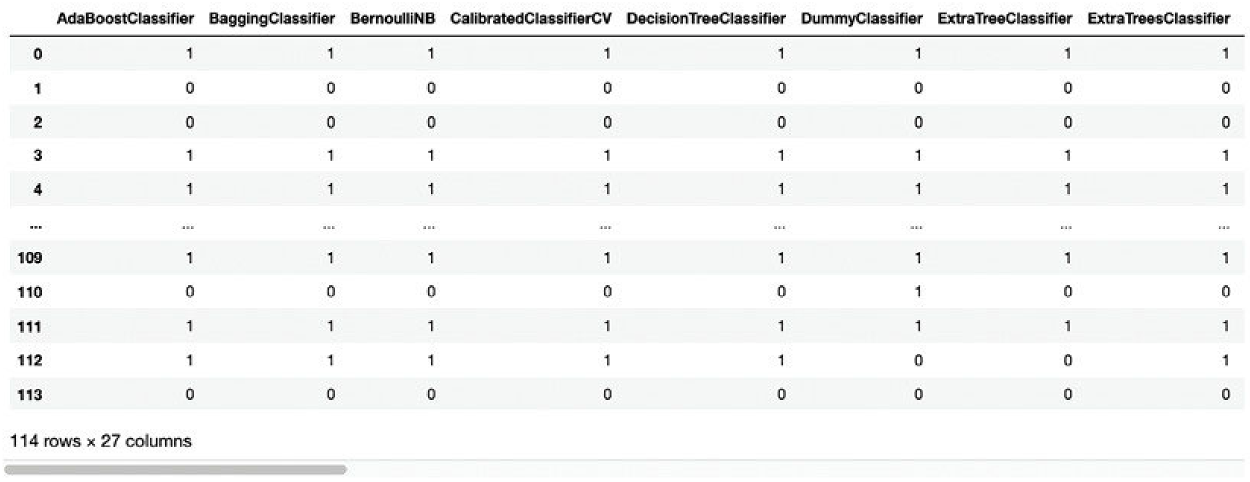
Model	Accuracy	Balanced Accuracy	ROC AUC	F1 Score	Time Taken
BernoulliNB	0.98	0.98	0.98	0.98	0.01
PassiveAggressiveClassifier	0.98	0.98	0.98	0.98	0.02
SVC	0.98	0.98	0.98	0.98	0.05
Perceptron	0.97	0.97	0.97	0.97	0.01
AdaBoostClassifier	0.97	0.97	0.97	0.97	0.21
LogisticRegression	0.97	0.97	0.97	0.97	0.04
SGDClassifier	0.96	0.97	0.97	0.97	0.06
ExtraTreeClassifier	0.96	0.97	0.97	0.97	0.01
CalibratedClassifierCV	0.97	0.97	0.97	0.97	0.05
RandomForestClassifier	0.96	0.96	0.96	0.96	0.32
LGBMClassifier	0.96	0.96	0.96	0.96	0.17
GaussianNB	0.96	0.96	0.96	0.96	0.02
ExtraTreesClassifier	0.96	0.96	0.96	0.96	0.15
QuadraticDiscriminantAnalysis	0.96	0.96	0.96	0.96	0.02
LinearSVC	0.96	0.96	0.96	0.96	0.03
BaggingClassifier	0.96	0.95	0.95	0.96	0.07
XGBClassifier	0.96	0.95	0.95	0.96	0.19
LinearDiscriminantAnalysis	0.96	0.95	0.95	0.96	0.03
NearestCentroid	0.96	0.95	0.95	0.96	0.04
NuSVC	0.96	0.95	0.95	0.96	0.04
RidgeClassifier	0.96	0.95	0.95	0.96	0.02
RidgeClassifierCV	0.96	0.95	0.95	0.96	0.04
KNeighborsClassifier	0.95	0.94	0.94	0.95	0.02
DecisionTreeClassifier	0.95	0.94	0.94	0.95	0.02
LabelSpreading	0.94	0.93	0.93	0.94	0.04
LabelPropagation	0.94	0.93	0.93	0.94	0.04
DummyClassifier	0.57	0.53	0.53	0.56	0.01

Figure 2.33: Lazy Predict report

You can see that various performance metrics are captured for various classification algorithms for the dataset used for the classification problem.

Predictions will contain all the predictions made by each algorithm for every record in the test set, as shown here:

predictions



	AdaBoostClassifier	BaggingClassifier	BernoulliNB	CalibratedClassifierCV	DecisionTreeClassifier	DummyClassifier	ExtraTreeClassifier	ExtraTreesClassifier
0	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1
...	...	...	...	...	...	...	...	...
109	1	1	1	1	1	1	1	1
110	0	0	0	0	0	1	0	0
111	1	1	1	1	1	1	1	1
112	1	1	1	1	1	0	0	1
113	0	0	0	0	0	0	0	0

114 rows × 27 columns

Figure 2.34: Lazy Predict predictions

Official page of Lazy Predict

Welcome to Lazy Predict’s documentation! — Lazy Predict 0.2.9 documentation

PyCaret

PyCaret is another amazing Python library that automates the model building process and experiments your dataset with almost every regression and classification algorithm out there. It also provides the detailed analysis of the model performances built using PyCaret.

The official page of PyCaret is

Installing and getting started with PyCaret

Follow the given steps to install and get started with PyCaret:

Step 1: Install the pycaret library using pip

Step 2: Import sample datasets available within the pycaret library and use a sample dataset for demonstration purposes, as shown here:

```
from pycaret.datasets import get_data
```

```
diabetes=get_data('diabetes')
```

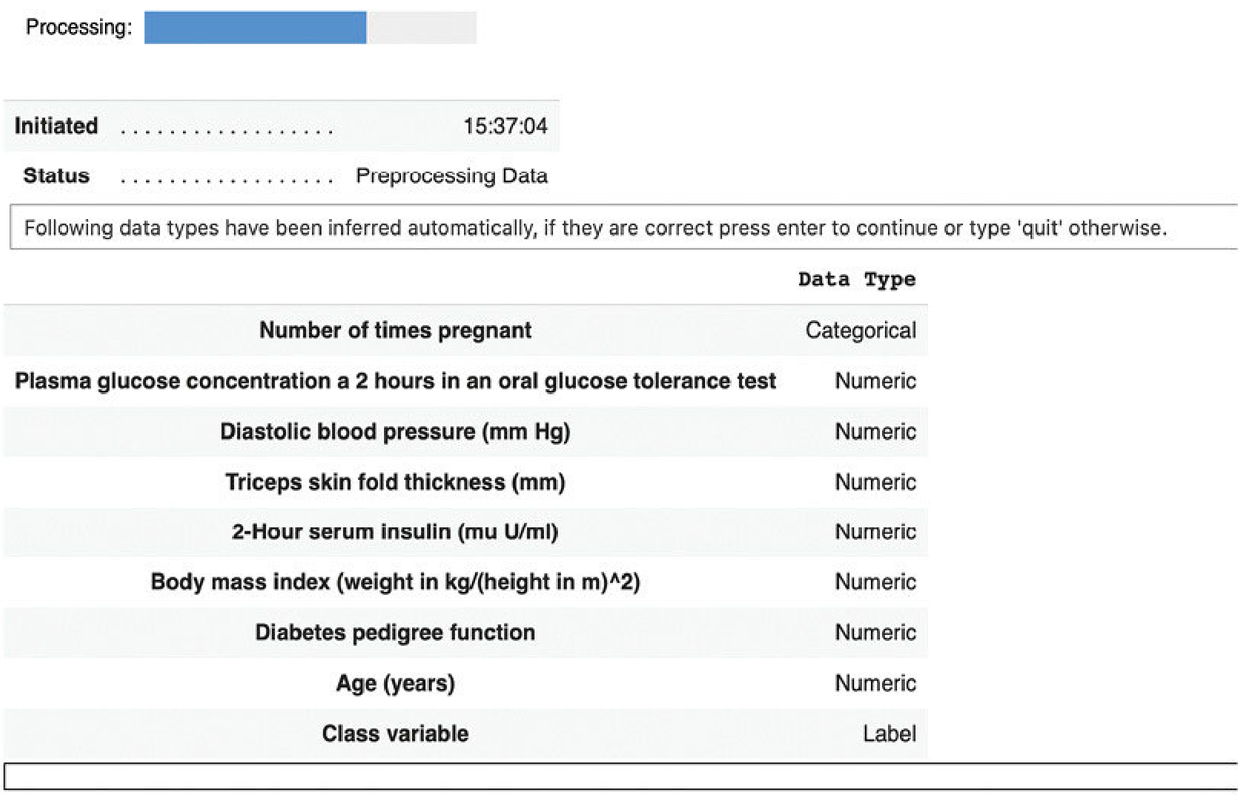
Step 3: Import the classification class within the pycaret library:

```
from pycaret.classification import *
```

Step 4: Use the setup method to initiate the classification experiment on the dataset by passing the DataFrame and the target variable to the method:

```
clf=setup(diabetes, target='Class variable')
```

It will perform some data analysis and ask for confirmation on whether the data types assumed for the provided columns in the DataFrame are correct. If they are correct, you can press and if you want to change the datatypes before the features are fed into the classifier, then type quit and press



Processing:

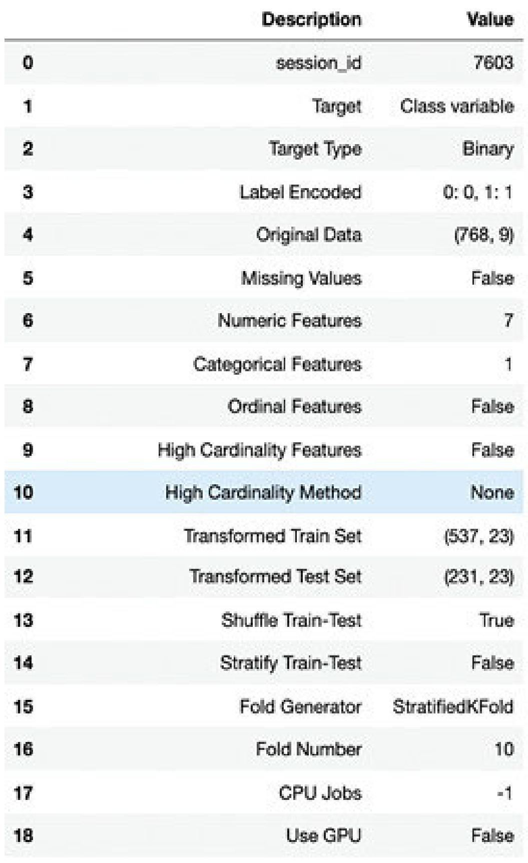
Initiated	....	15:37:04
Status	....	Preprocessing Data

Following data types have been inferred automatically, if they are correct press enter to continue or type 'quit' otherwise.

	Data Type
Number of times pregnant	Categorical
Plasma glucose concentration a 2 hours in an oral glucose tolerance test	Numeric
Diastolic blood pressure (mm Hg)	Numeric
Triceps skin fold thickness (mm)	Numeric
2-Hour serum insulin (mu U/ml)	Numeric
Body mass index (weight in kg/(height in m)^2)	Numeric
Diabetes pedigree function	Numeric
Age (years)	Numeric
Class variable	Label

Figure 2.35: PyCaret data analysis

Step 5: Assuming that the data types are correctly identified, you press and then the classification process upon various supported algorithms within the PyCaret library begins:



	Description	Value
0	session_id	7603
1	Target	Class variable
2	Target Type	Binary
3	Label Encoded	0: 0, 1: 1
4	Original Data	(768, 9)
5	Missing Values	False
6	Numeric Features	7
7	Categorical Features	1
8	Ordinal Features	False
9	High Cardinality Features	False
10	High Cardinality Method	None
11	Transformed Train Set	(537, 23)
12	Transformed Test Set	(231, 23)
13	Shuffle Train-Test	True
14	Stratify Train-Test	False
15	Fold Generator	StratifiedKFold
16	Fold Number	10
17	CPU Jobs	-1
18	Use GPU	False

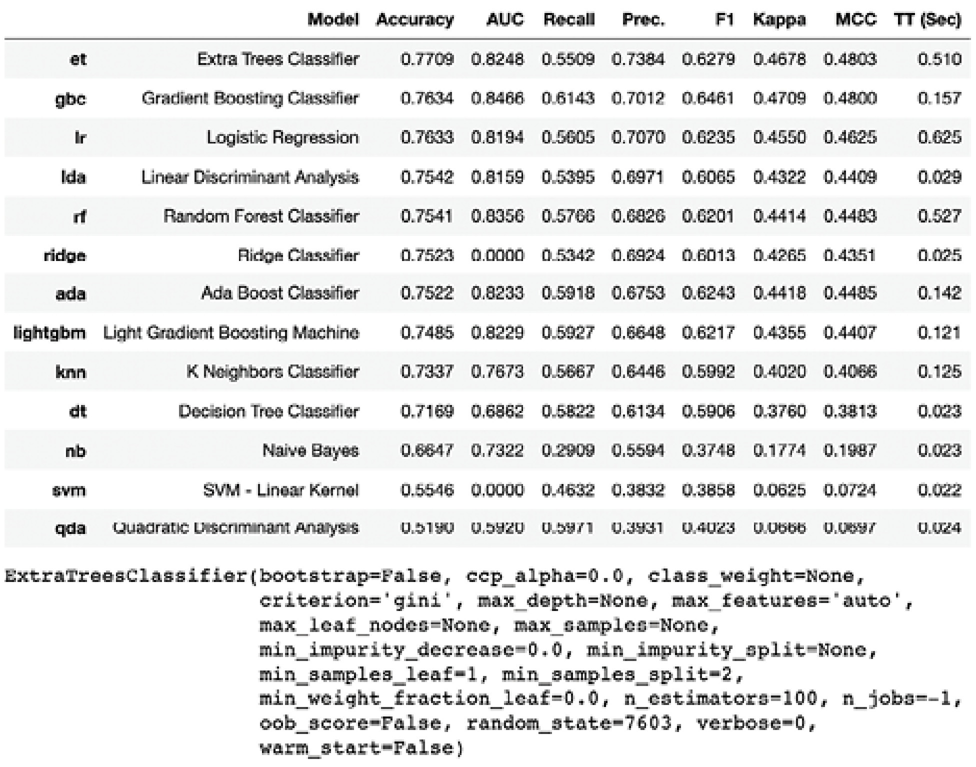
Figure 2.36: PyCaret experiment description

Analyzing PyCaret experimentation results

Once the processing is completed, use the compare_models method to see the full analysis of all the experimentations:

compare_models()

The output obtained is as follows:



	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	TT (Sec)
et	Extra Trees Classifier	0.7709	0.8248	0.5509	0.7384	0.6279	0.4678	0.4803	0.510
gbc	Gradient Boosting Classifier	0.7634	0.8466	0.6143	0.7012	0.6461	0.4709	0.4800	0.157
lr	Logistic Regression	0.7633	0.8194	0.5605	0.7070	0.6235	0.4550	0.4625	0.625
lda	Linear Discriminant Analysis	0.7542	0.8159	0.5395	0.6971	0.6065	0.4322	0.4409	0.029
rf	Random Forest Classifier	0.7541	0.8356	0.5766	0.6826	0.6201	0.4414	0.4483	0.527
ridge	Ridge Classifier	0.7523	0.0000	0.5342	0.6924	0.6013	0.4265	0.4351	0.025
ada	Ada Boost Classifier	0.7522	0.8233	0.5918	0.6753	0.6243	0.4418	0.4485	0.142
lightgbm	Light Gradient Boosting Machine	0.7485	0.8229	0.5927	0.6648	0.6217	0.4355	0.4407	0.121
knn	K Neighbors Classifier	0.7337	0.7673	0.5667	0.6446	0.5992	0.4020	0.4066	0.125
dt	Decision Tree Classifier	0.7169	0.6862	0.5822	0.6134	0.5906	0.3760	0.3813	0.023
nb	Naive Bayes	0.6647	0.7322	0.2909	0.5594	0.3748	0.1774	0.1987	0.023
svm	SVM - Linear Kernel	0.5546	0.0000	0.4632	0.3832	0.3858	0.0625	0.0724	0.022
qda	Quadratic Discriminant Analysis	0.5190	0.5920	0.5971	0.3931	0.4023	0.0666	0.0697	0.024

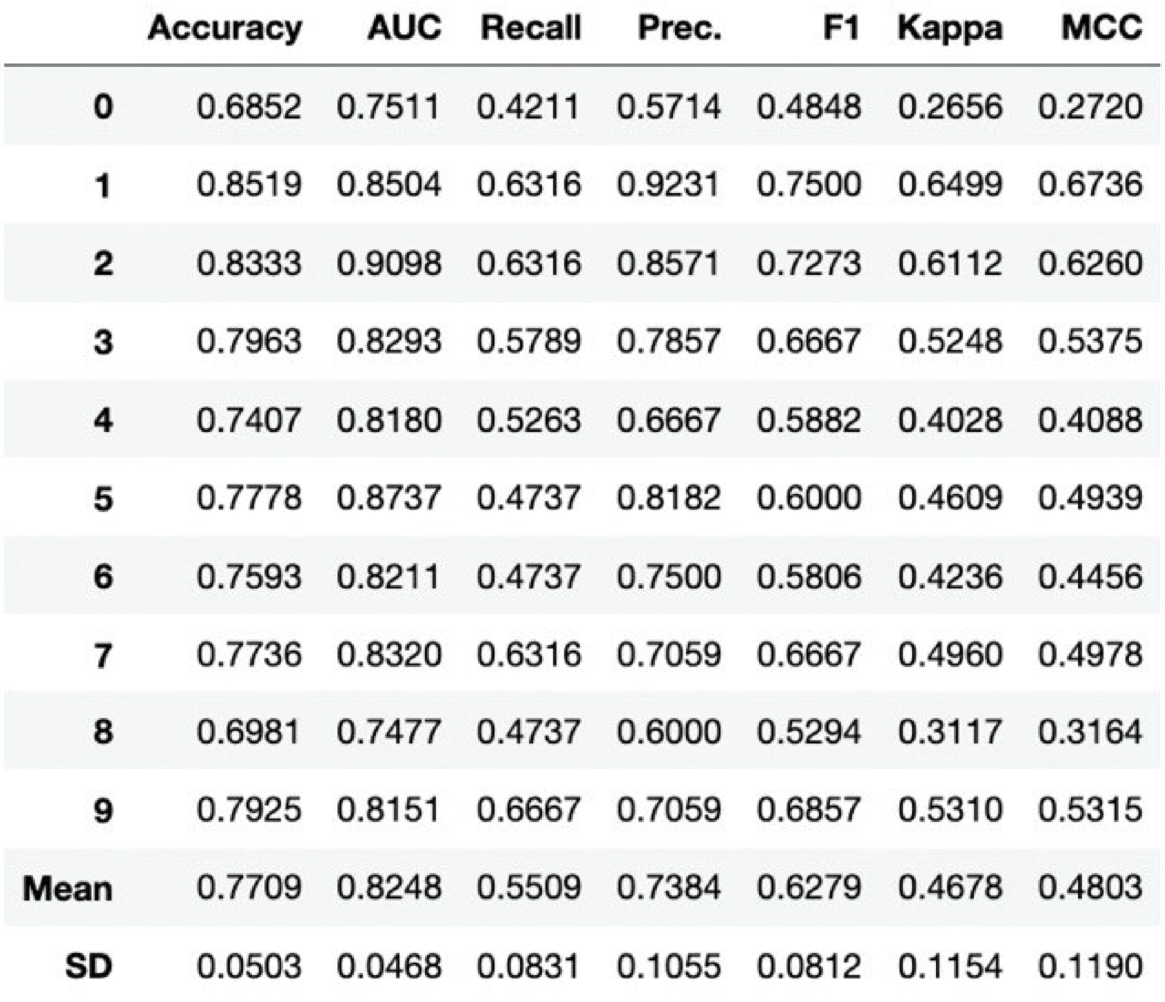
```
ExtraTreesClassifier(bootstrap=False, ccp_alpha=0.0, class_weight=None,
                     criterion='gini', max_depth=None, max_features='auto',
                     max_leaf_nodes=None, max_samples=None,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=1, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=-1,
                     oob_score=False, random_state=7603, verbose=0,
                     warm_start=False)
```

Figure 2.37: PyCaret report

The results are, by default, sorted based on 'Accuracy' metrics; you can see that Extra Trees Classifier worked well on this dataset. But based on the performance metrics that you are looking for your problem statement, you can choose the appropriate classifier from the list and create the model using the create_model method:

```
et = create_model('et')
```

This will build the model of your choice and perform the cross-validation for 10 iterations by default and present the final performance metrics, as shown in the following screenshot:



	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.6852	0.7511	0.4211	0.5714	0.4848	0.2656	0.2720
1	0.8519	0.8504	0.6316	0.9231	0.7500	0.6499	0.6736
2	0.8333	0.9098	0.6316	0.8571	0.7273	0.6112	0.6260
3	0.7963	0.8293	0.5789	0.7857	0.6667	0.5248	0.5375
4	0.7407	0.8180	0.5263	0.6667	0.5882	0.4028	0.4088
5	0.7778	0.8737	0.4737	0.8182	0.6000	0.4609	0.4939
6	0.7593	0.8211	0.4737	0.7500	0.5806	0.4236	0.4456
7	0.7736	0.8320	0.6316	0.7059	0.6667	0.4960	0.4978
8	0.6981	0.7477	0.4737	0.6000	0.5294	0.3117	0.3164
9	0.7925	0.8151	0.6667	0.7059	0.6857	0.5310	0.5315
Mean	0.7709	0.8248	0.5509	0.7384	0.6279	0.4678	0.4803
SD	0.0503	0.0468	0.0831	0.1055	0.0812	0.1154	0.1190

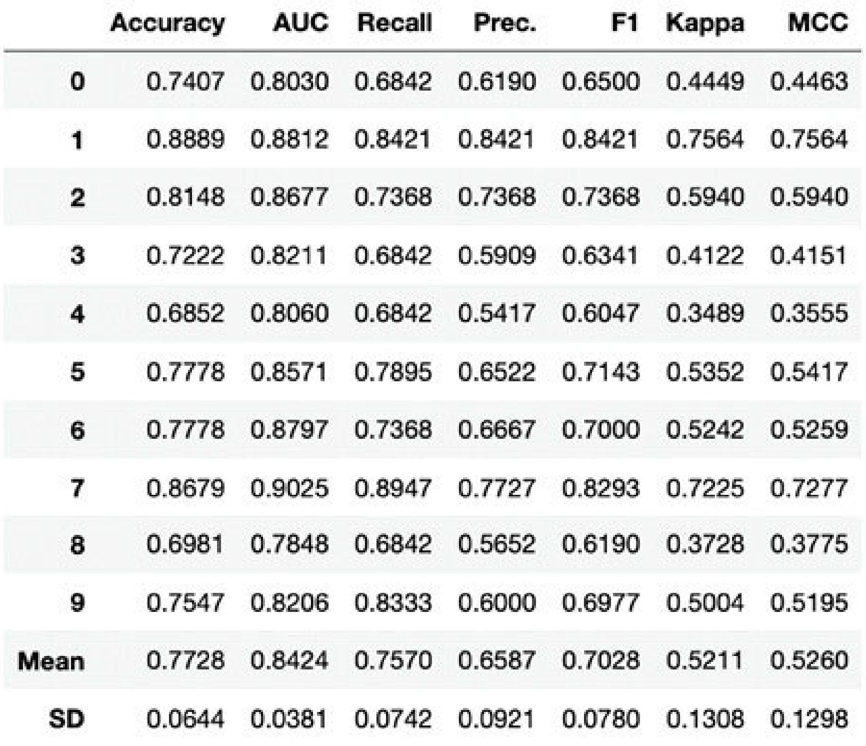
Figure 2.38: PyCaret performance metrics

Advanced features of PyCaret

However, this is not the end of the story. PyCaret comes with many more advanced capabilities post the experimentation and model building process.

Once basic experimentation is completed, the pycaret library also provides the flexibility to tune the model with just a single line of code:

```
tuned_et = tune_model(et)
```



	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.7407	0.8030	0.6842	0.6190	0.6500	0.4449	0.4463
1	0.8889	0.8812	0.8421	0.8421	0.8421	0.7564	0.7564
2	0.8148	0.8677	0.7368	0.7368	0.7368	0.5940	0.5940
3	0.7222	0.8211	0.6842	0.5909	0.6341	0.4122	0.4151
4	0.6852	0.8060	0.6842	0.5417	0.6047	0.3489	0.3555
5	0.7778	0.8571	0.7895	0.6522	0.7143	0.5352	0.5417
6	0.7778	0.8797	0.7368	0.6667	0.7000	0.5242	0.5259
7	0.8679	0.9025	0.8947	0.7727	0.8293	0.7225	0.7277
8	0.6981	0.7848	0.6842	0.5652	0.6190	0.3728	0.3775
9	0.7547	0.8206	0.8333	0.6000	0.6977	0.5004	0.5195
Mean	0.7728	0.8424	0.7570	0.6587	0.7028	0.5211	0.5260
SD	0.0644	0.0381	0.0742	0.0921	0.0780	0.1308	0.1298

Figure 2.39: PyCaret tuned model performance metrics

Once the model is created, you can also plot the model ROC curves with just a single line of code:

```
plot_model(et)
```

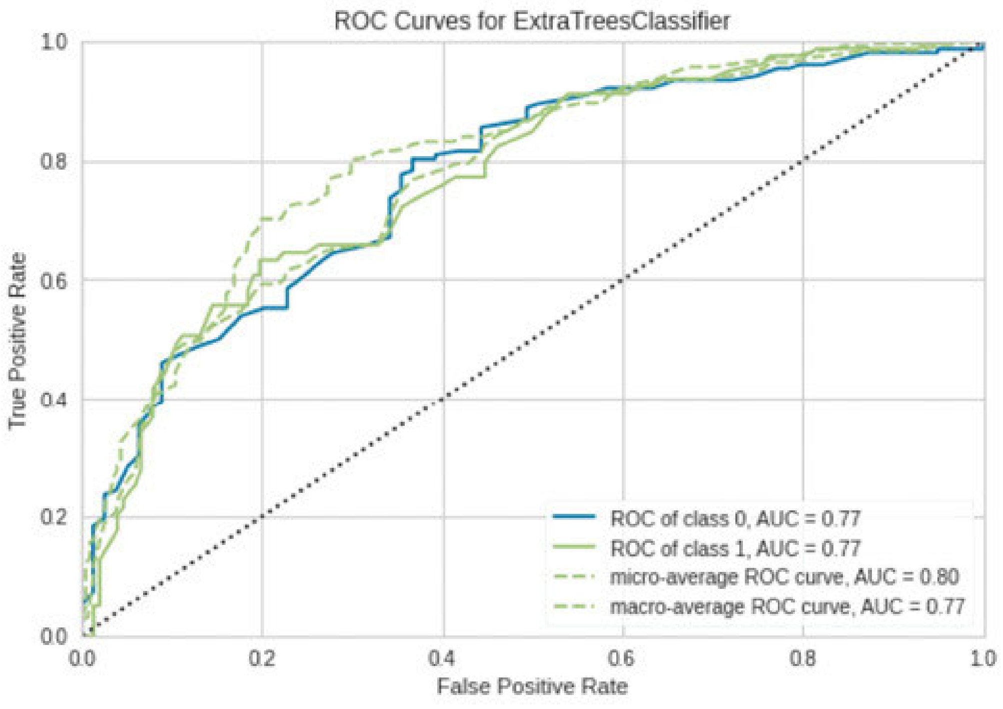


Figure 2.40: Plot model performance

Plotting confusion matrix of the model

Once the model is created, you can also plot the model confusion matrix with just a single line of code:

```
plot_model(et, plot = 'confusion_matrix')
```

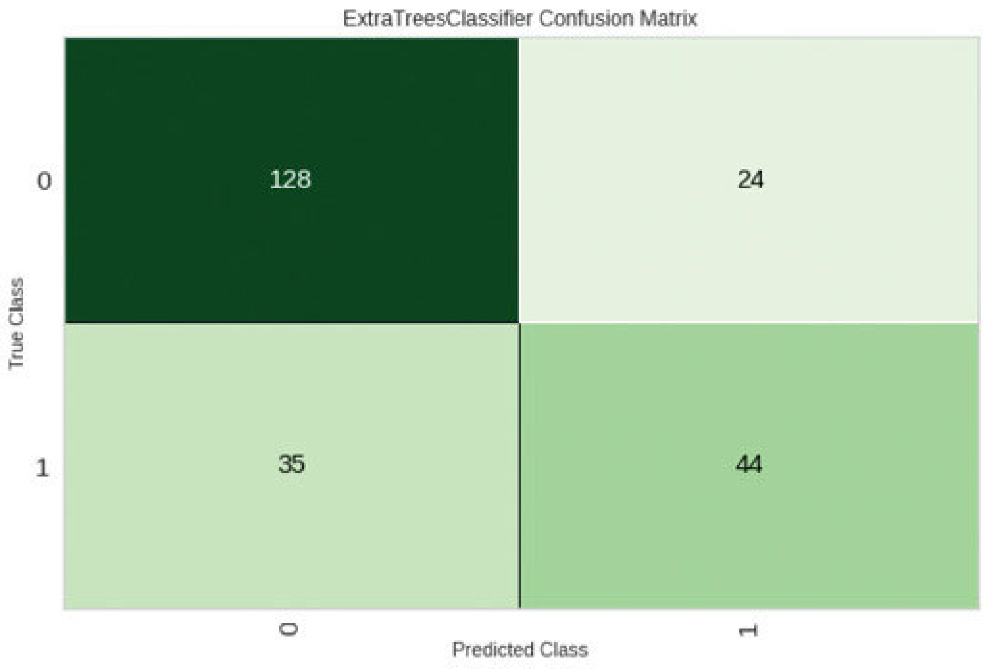


Figure 2.41: Confusion matrix

Plotting feature importance

Once the model is created, you can also plot the feature importance with just a single line of code:

plot_model(et, plot = 'feature')

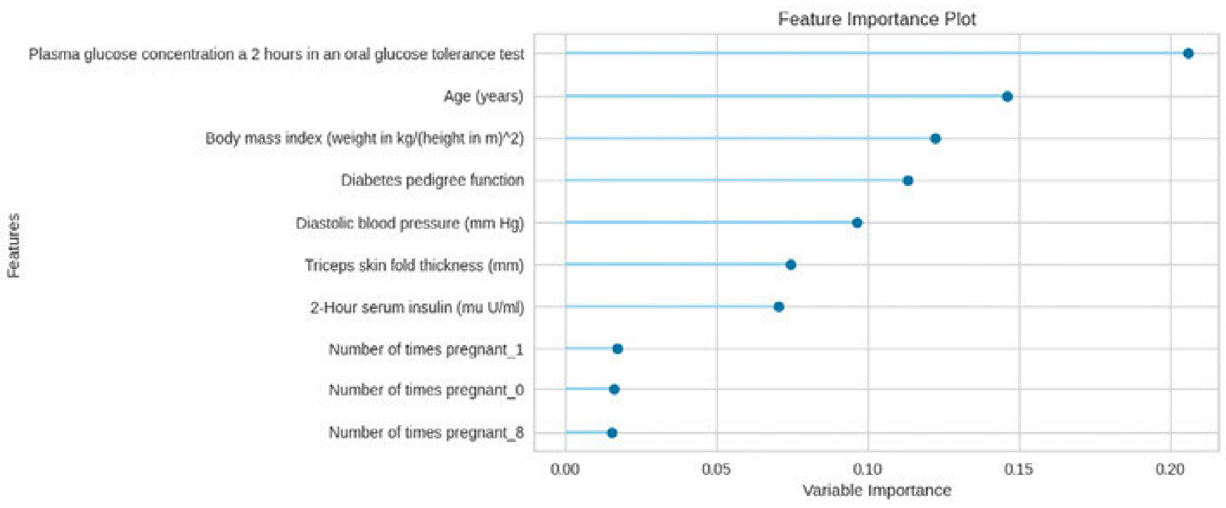


Figure 2.42: Feature importance

We have only scratched the surface of the features available in the PyCaret library. Refer to the official documentation to know about many more things that can be done using this library, which can speed up your model building, optimization, and evaluation processes.

Conclusion

In this chapter, we looked at various advanced Python libraries that help in automating various exploratory data analysis steps. With only a few lines of code, profiling of the dataset, visualizations, and detailed statistical reports can be generated in no time. We also looked at various Python libraries that enable experimenting with implementation of all kinds of classification and regression algorithms, without having to code for each of those individual algorithm implementations. Additionally, we discussed how these automated tools give a head start to the data modeling process and save a lot of time as well.

In the next chapter, we will cover the basics of neural networks and look at how to build the basic neural network-based predictive model from scratch.

Questions

Which of the following Python libraries can be used for automated data exploration?

Pandas profiling

AutoViz

Sweetviz

All of the above

Which of the following Python libraries can be used for automated ML modeling?

PyCaret

Lazy Predict

Both of the above

None of the above

What is the confusion matrix?

Visualization algorithm

Machine learning algorithm

Performance metrics

None of the above

What are the different types of machine learning?

Supervised learning

Unsupervised learning

Reinforcement learning

All of the above

What are the different types of problems that machine learning can solve?

Classification

Regression

Clustering

Forecasting

All of the above

None of the above

Answers

(d)

(c)

(c)

(d)

(e)

CHAPTER 3

Build a Neural Network-Based Predictive Model

We are all now connected by the internet, like neurons in a giant brain.
— Stephen Hawking

Have you ever wondered how our brain functions? How does it identify things, understand patterns, and make decisions? This is an age-old mystery, and all one can do is to presume the neurological functioning of the brain, which makes it capable of performing all the incredible cognitive and mathematical analysis. The answer is ‘Biological Neural Networks’. The human brain is a humongous network of neurons, and these neurons are connected in a systematic way that helps interpret data absorbed by the sense organs like the eyes, ears, and nose from the physical world. Artificial Neural Networks is structurally inspired by the human brain, wherein connected networks of neurons are artificially developed, which are then trained using training data to make the network understand the pattern of the problems we are making computers to solve.

Structure

In this chapter, we will cover the following topics:

ANN and its components

Building a classification model using neural network

Building a regression model using neural network

Objectives

After studying this chapter, you should be aware of the various components of an artificial neural network. Further, you will build neural network-based classification and regression models using the Python keras library. Once you build a predictive model, it is important to understand the performance of the model. To assess the performance of the model, there are various performance metrics for both classification and regression problems. In this chapter, you will understand what performance metrics one should derive in order to optimally assess the model. Finally, you will understand how to tune the model by tuning various hyperparameters.

Artificial neural network and its components

Artificial Neural Network as the name suggests, is an artificially created neural network composed of artificial neurons. These neurons are nothing but interceptors of the incoming data signals, and they fire up based on the underlying mathematical inference.

First, what is a neural network? The word ‘neural’ refers to ‘neurons’, and the word ‘network’ refers to a graph-like structure. Hence, neural networks are just a systematic network of neurons, which can interpret the physical world around us.

These neurons are systematically arranged in various layers, as shown in Figure

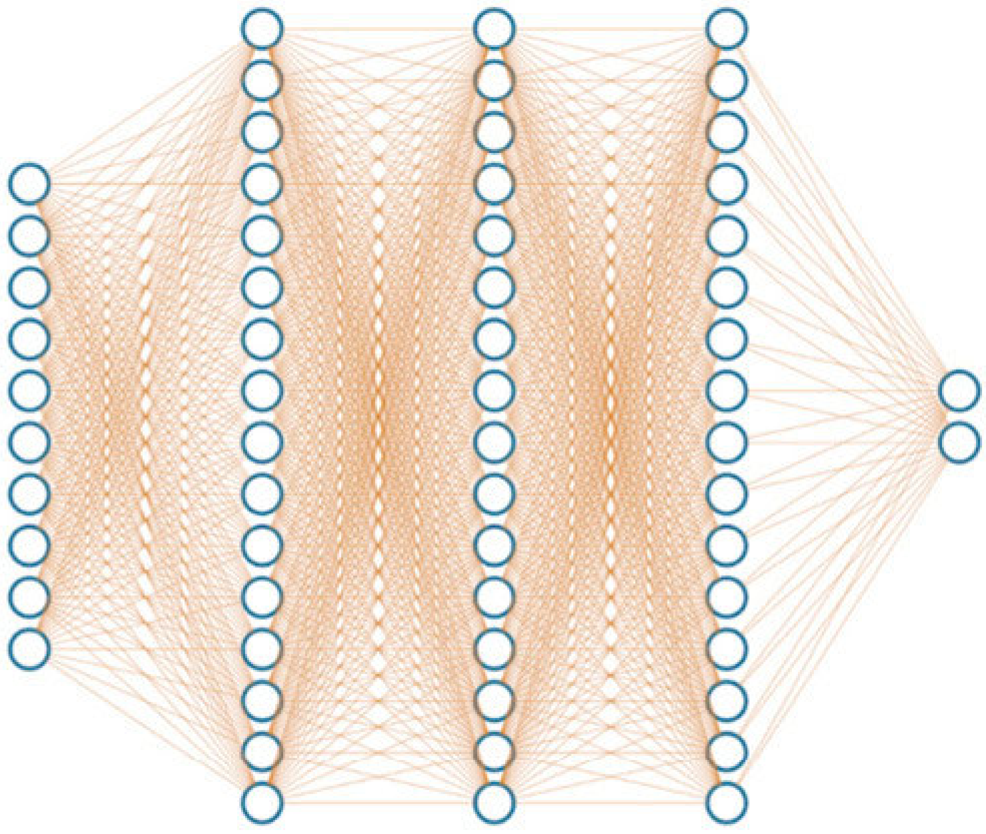


Figure 3.1: Hypothetical depiction of neural network

The first layer is called the input layer, and the last layer is called the output layer. All the layers in between are called hidden layers. Now, if many layers are defined in the hidden layer, the network is called a deep neural network.

Figure 3.2 gives a simple and clear picture of how Artificial Intelligence Machine Learning and neural networks are related:

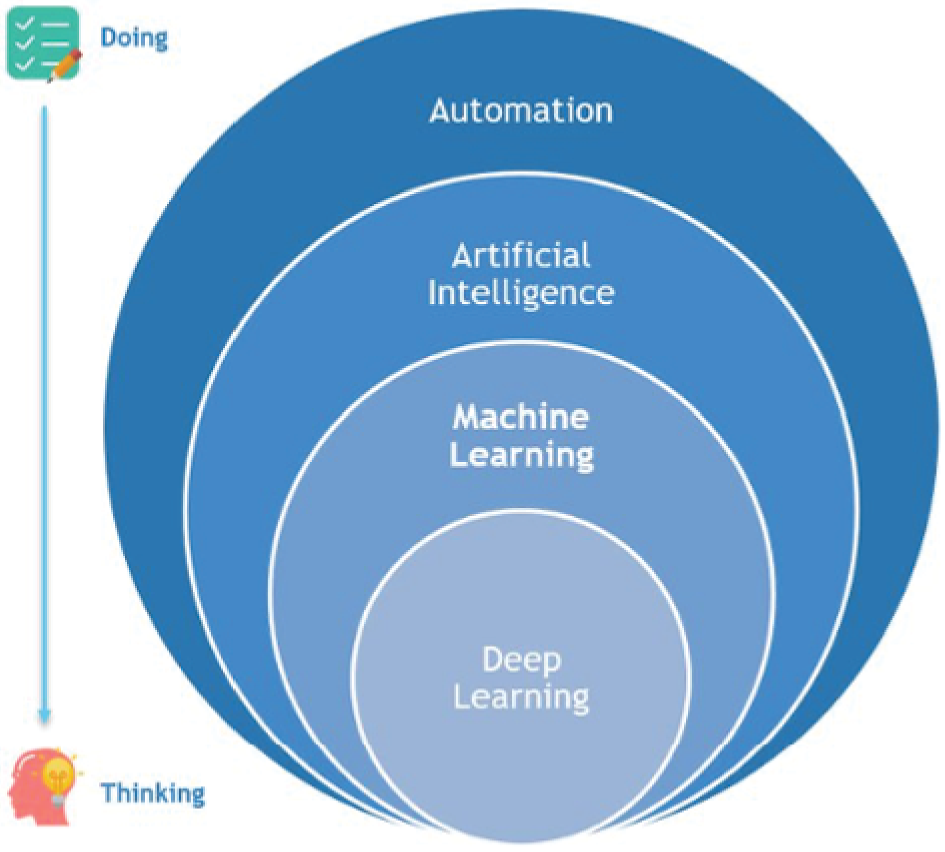


Figure 3.2: Subfields of AI

Neurons

Neurons are the most basic components of a neural network. The best way to understand an artificial neuron is by drawing parallels with its biological counterpart, the human brain neuron.

In a human brain, information is intercepted via dendrites to the neuron cell body, and based on some rule within the cell body, the relevant information is passed further via axon terminals, as shown in Figure

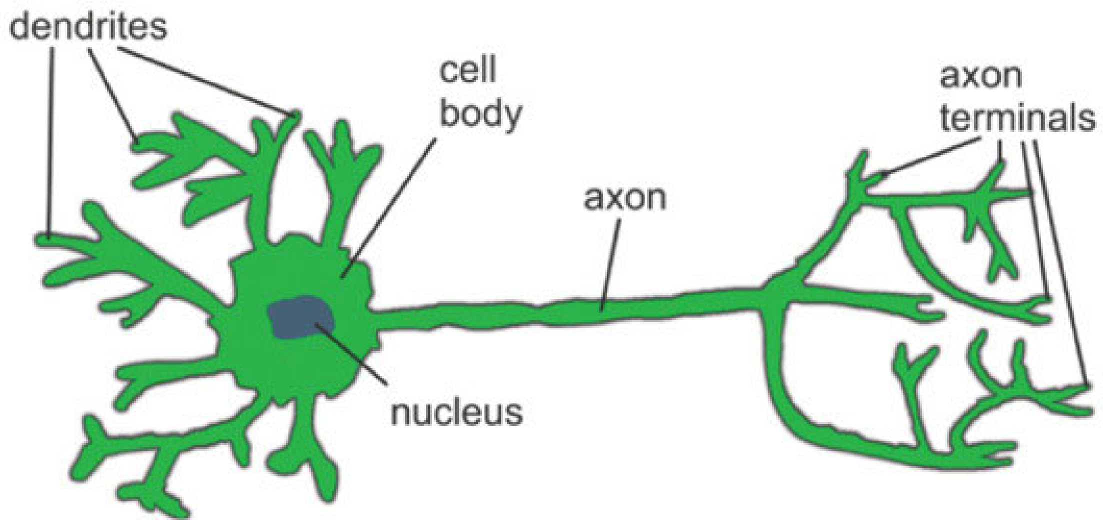


Figure 3.3: Biological neuron and its components

Similarly, in an artificial neuron, as shown in Figure the inputs are injected into a cell body (neuron) through one or multiple input neurons. Then, certain mathematical calculations are processed within the cell body, and an output is generated. Now, that output can be passed for further processing. Take a look at the following figure:

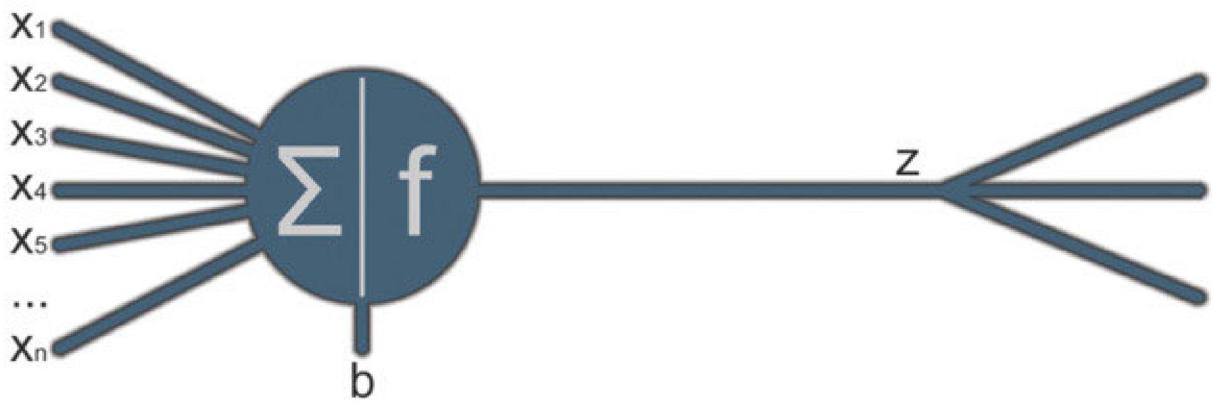


Figure 3.4: Artificial neuron and its components

The cell body in the artificial neuron, as shown in Figure comprises two components: a summation and a function. Now, every input in a neural network is associated with a weight. And every neuron in the network has a bias term associated with it. The summation in the neuron is the sum of the input and weights and the bias term, as shown in Figure

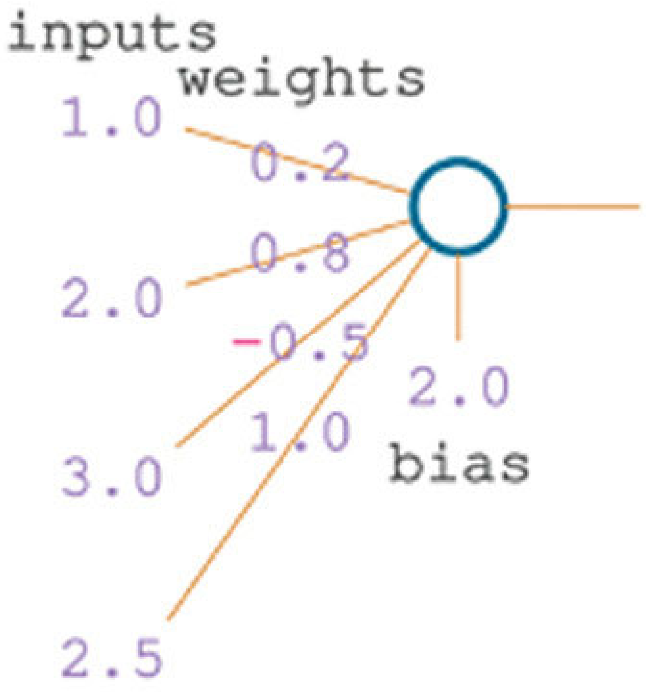


Figure 3.5: Mathematical depiction of a neuron

Each input with its associated weights is mathematically computed and added to a bias term that is specific to each neuron. Hence, the output from the calculation will be as follows:

```
1.0*0.2 + 2.0*0.8 + 3.0*-0.5 + 2.0 = 2.3
```

Figure 3.6: Neuron formula

Consider this happening with each neuron in the network, as shown in Figure

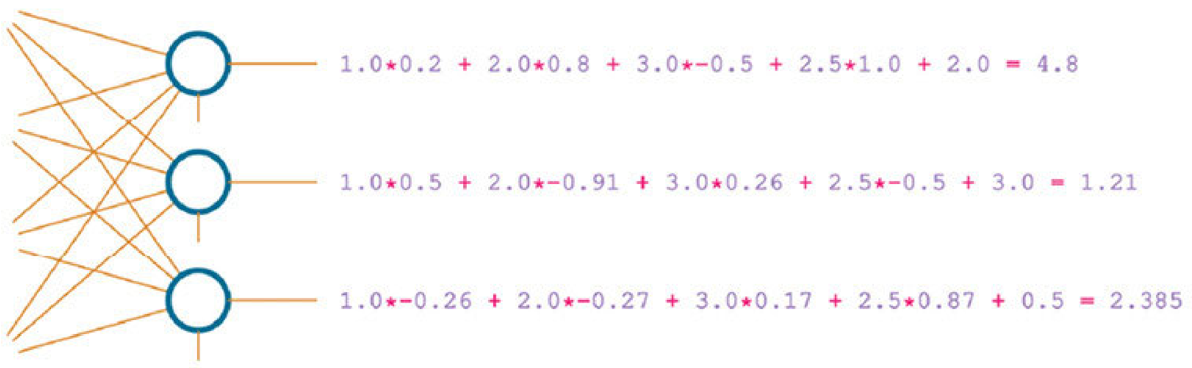


Figure 3.7: Neurons and associated maths

Now, this output value is treated by a function, as shown in Figure called the activation function. Finally, based on the activation function output, the final output from the neuron is generated and passed for further processing in the network.

The full picture of what happens within a neuron is depicted in Figure

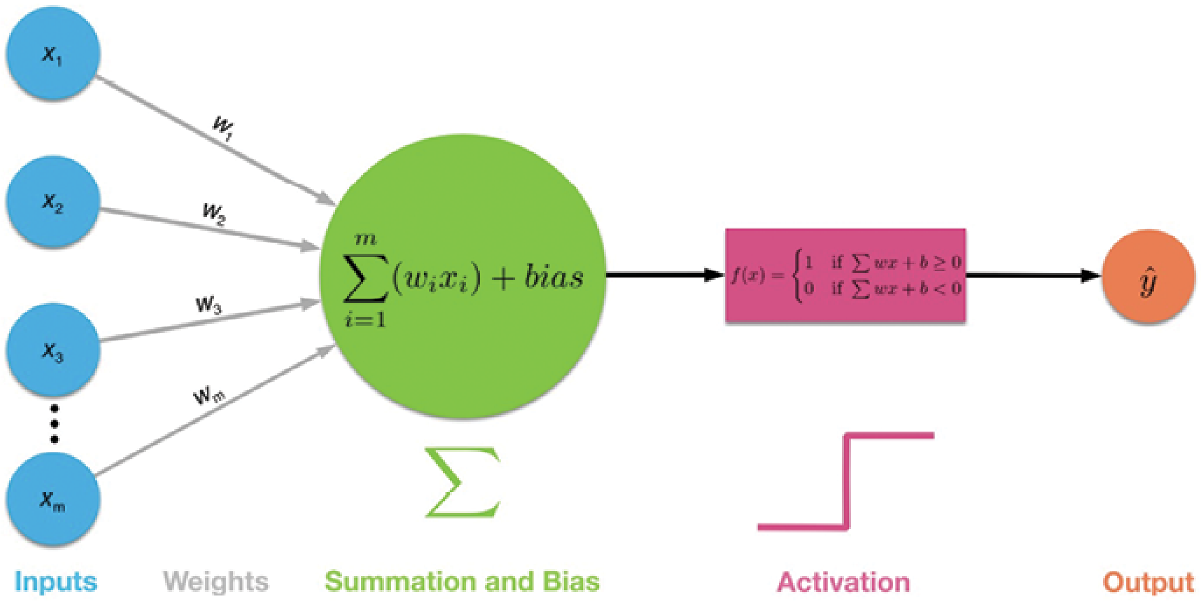

x_1
x_2
x_3
x_m
w_1
w_2
w_3
w_m
$\sum_{i=1}^{m}(w_i x_i) + bias$
$f(x) = \begin{cases} 1 & \text{if } \sum wx + b \geq 0 \\ 0 & \text{if } \sum wx + b < 0 \end{cases}$
$\hat{y}$
$\sum$
Inputs
Weights
Summation and Bias
Activation
Output

Figure 3.8: End-to-end mathematical depiction of a single neuron pass

This underlying process of inputs interacting with weights and biases, and then the process of activation function treatment within each neuron happens within every artificial neuron in the artificial neural network, as shown in Figure

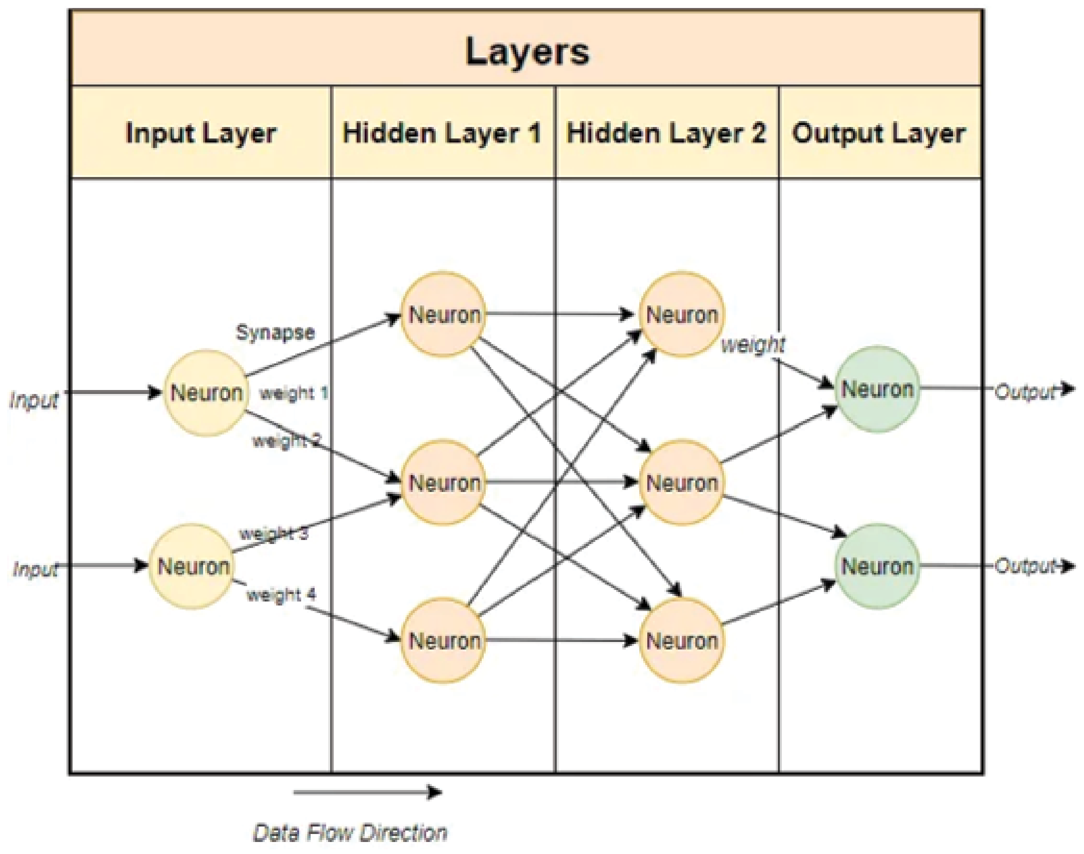


Figure 3.9: Neural network

Before we jump into building a deep learning model using the keras library, it is important to understand a few other components of a deep neural network and its training process, that is, the model building process.

Other than neurons, weights, biases, activation functions, input layer, output layer, and hidden layers, as discussed earlier, there are various other important components in an artificial neural network that are important to understand before we delve into building a deep neural network using the keras library.

The first thing to understand here is that this process of training a network or building a model is all about finding the optimum values of weights and biases of each neuron in the network.

Feed forward

When a network is built and before the input data is fed into the network for starting the training process, we must know that the weights and biases of each neuron in the network are randomly generated. Now, with those random weights and biases, the input data is traversed through the network end to end, and the output is generated in the output layer for each input. This process of traversing the input data from input layer till the output layer is called feed forward.

Activation functions

In a neural network, activation functions are used to introduce non-linearity to the output of a neuron. Without activation functions, a neural network would be limited to linear transformations of the input data, which would severely limit its ability to model complex relationships between inputs and outputs.

Activation functions transform the input signal to a neuron into an output signal, which is then passed on to the next layer of the neural network. They can make the output signal more or less sensitive to changes in the input signal, and they can also be used to squash the output signal to a specific range, such as between 0 and 1 or between -1 and 1.

By introducing non-linearity to the output of a neuron, activation functions enable the neural network to model complex patterns and relationships in the input data, allowing it to learn and make decisions based on those patterns.

Here are some commonly used activation functions in neural networks and their descriptions:

A function that maps any input to a value between 0 and 1, and is used for binary classification tasks

Rectified Linear Unit A function that outputs the input value if it is positive, and 0 if it is negative; it is commonly used in deep neural networks due to its simplicity and effectiveness

Tanh (Hyperbolic A function that maps any input to a value between -1 and 1; it is similar to sigmoid but with a range that is symmetric around 0

A function that maps a vector of inputs to a probability distribution over the classes in a multi-class classification task

Leaky A modified version of ReLU that allows for a small positive output when the input is negative, to avoid the “dead neuron” problem

Exponential Linear Unit A function that is similar to ReLU but with a smooth curve that allows for negative inputs.

Each of these activation functions has its own strengths and weaknesses and is appropriate for different types of neural networks and tasks.

Loss function

Once the output is generated from the feed forward process, the predicted output is compared with the actual output, considering we are performing supervised learning. Upon this comparison, an error for each record is captured, and subsequently, the overall error or loss of the network is calculated using the loss function. There are various functions used to calculate the loss, depending on whether it is a classification or a regression problem.

Some of the widely used loss functions are as follows:

You can read more about these loss functions here:

https://keras.io/api/losses/

Backward propagation

The overall error or loss generated by the network in a certain forward pass has been captured so that the process can go back into the network and tune the weights and biases of each neuron to produce better results the next time. This process of going back into the network after a forward pass and determining the loss in quest of a better performing model is called backward propagation.

Epoch

This process of forward pass, loss calculation, and backward pass can be performed as many times as required for building the most accurate and optimum model. This end-to-end movement of the entire dataset from the input layer to the output layer and then back to the input layer is called epoch.

Batch

The data used for training a model can be a lot and may not be feasible to be processed all at once. Hence, data must be divided into batches. Let us say there are 1000 records in the input dataset, and it seems to be computationally feasible to compute only 250 records in one pass. Hence, 4 batches will be created with a batch size of 250 records for each pass.

Iteration

As we know, an epoch is considered complete when the entire dataset is passed through the network to and fro once. Now, if the training sample is divided into a certain number of batches, then the number of passes it will require for all the batches to make one pass in the network is called iteration. Hence, if there are 1000 training samples divided into a batch size of 250, it will take 4 iterations to complete 1 Epoch.

Optimizer

As we know, during the backward propagation, the weights and biases are tuned in the network to find their optimum values. The optimizers are the ones that work on the concept of gradient descent in the quest of finding the global minima, which tune these parameters. Without delving deep into this process, understand that it is the optimizer's task to determine whether the values of weights and biases are to be increased or decreased during the backward propagation.

Some of the widely used optimizers are as follows:

The Stochastic Gradient Descent optimizer is an iterative optimization algorithm used in machine learning that updates the model parameters in small steps by computing the gradients of the objective function with respect to the parameters, using a random subset of the training data at each iteration.

The RMSprop optimizer is a variant of the Stochastic Gradient Descent algorithm that uses an exponentially weighted moving average of past squared gradients to adjust the learning rate and accelerate convergence for non-convex optimization problems.

The Adam optimizer is an adaptive optimization algorithm used in machine learning that combines the advantages of the RMSprop and momentum methods by incorporating first- and second-order moments of the gradients to update the learning rate and improve convergence speed for stochastic optimization problems.

You can read more about these optimizers here:

https://keras.io/api/optimizers/

Learning rate

As we know, the optimizers increase or decrease the weights and biases of the neurons in the network to tune those parameters for the model to produce the best results. The learning rate determines the rate at which these parameters should be increased or decreased by the optimizer in each epoch to test the model performance.

Since we theoretically know the building blocks of neural networks, let us experiment with building a classification model using neural network.

Building a classification model using neural network

Classification models can be binary, multi-class, or multi-label.

When you need to classify the data points into only two categories, like either spam or ham for email classification, it is said to be a binary classification. An email can be classified into any one of the two classes only in a binary classification.

When you need to classify an email into more than two categories, like classifying emails into either ‘primary’, ‘promotional’, ‘social’, or any other class, it is said to be a multi-class classification. An email can be classified into any of the defined classes in multi-class classification.

When you need to tag multiple categories to an email, like an email can be both ‘primary’ and ‘promotional’, multiple labels will be tagged to an email through a classification model. This is multi-label classification.

Let us look at a neural network-based model implementation for a binary classification.

Problem statement

The problem is about classifying a tumor into either ‘malignant’ or ‘benign’ using the characteristics of the breast mass provided in the dataset.

Dataset

We are using the breast cancer dataset from the sklearn.datasets library. The data contains records of the characteristics of breast mass like radius, texture, smoothness, fractal dimensions, and so on. It is a labeled dataset, and each record is labeled as either ‘malignant’ or ‘benign’. This is clearly a problem of binary classification.

Implementation

We will implement a classification model using the Python keras library.

Load Python libraries

Let us start by loading all the required libraries:

```
import keras

import tensorflow as tf

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn import datasets
```

Load data

Use the load_breast_cancer method in the datasets module within the sklearn library to load the data into a bunch object, as follows:

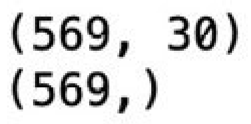
```
breast_cancer_data = datasets.load_breast_cancer()
```

The bunch object created is used to separate predictors and target variables into X and y objects separately, as shown:

```
X, y = breast_cancer_data.data, breast_cancer_data.target
```

Let us check the size of the dataset:

```
print(X.shape)

print(y.shape)
```

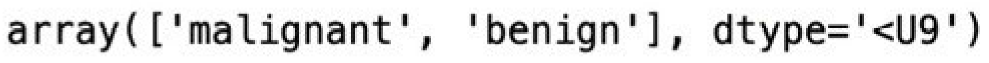


```
(569, 30)
(569,)
```

Figure 3.10: DataFrame shape

So, there are a total of 569 records and 30 features in the dataset. Using those 30 features, a classification model needs to be trained to predict the class of the breast mass as either ‘benign’ or ‘malignant’, as shown in the following code snippet:

breast_cancer_data.target_names

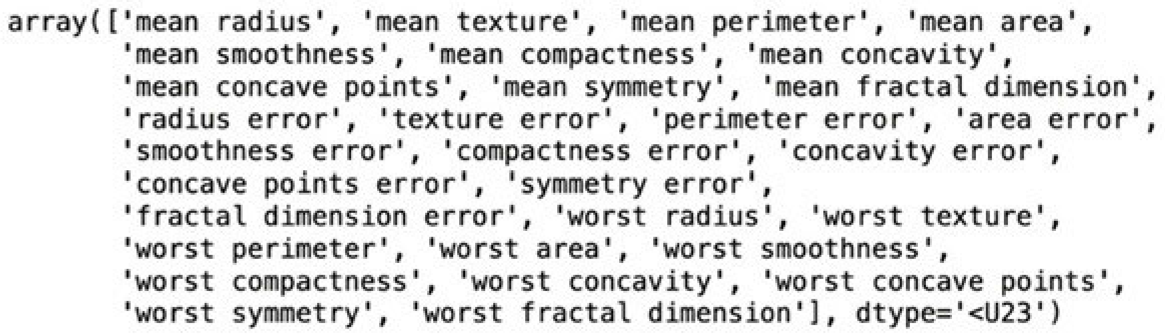


```
array(['malignant', 'benign'], dtype='<U9')
```

Figure 3.11: Target names

The set of 30 features in the dataset is as follows:

breast_cancer_data.feature_names



```
array(['mean radius', 'mean texture', 'mean perimeter', 'mean area',
       'mean smoothness', 'mean compactness', 'mean concavity',
       'mean concave points', 'mean symmetry', 'mean fractal dimension',
       'radius error', 'texture error', 'perimeter error', 'area error',
       'smoothness error', 'compactness error', 'concavity error',
       'concave points error', 'symmetry error',
       'fractal dimension error', 'worst radius', 'worst texture',
       'worst perimeter', 'worst area', 'worst smoothness',
       'worst compactness', 'worst concavity', 'worst concave points',
       'worst symmetry', 'worst fractal dimension'], dtype='<U23')
```

Figure 3.12: Feature names

Descriptive analytics

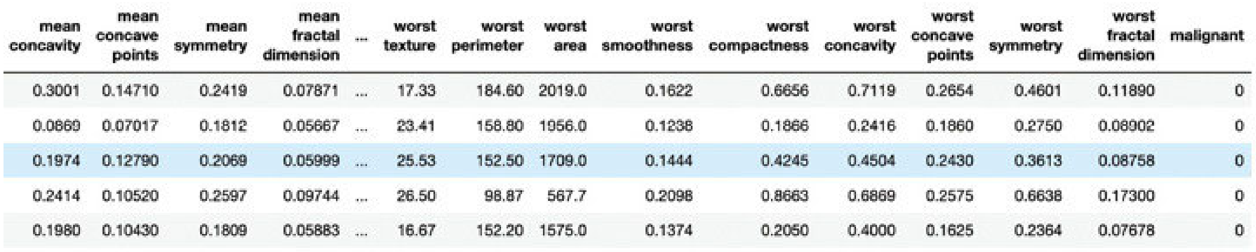
Let us create a pandas DataFrame using X and y for descriptive analytics, as shown:

```
df=pd.DataFrame(X,     columns=list(breast_cancer_data.feature_names))
```

```
df['malignant']=y
```

```
df.head()
```

The preceding code prints the first 5 rows of the DataFrame:

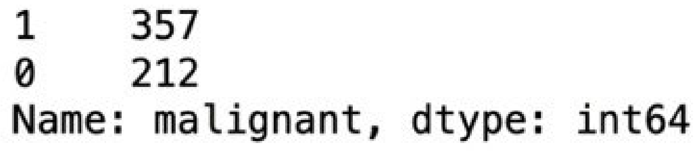


mean concavity	mean concave points	mean symmetry	mean fractal dimension	...	worst texture	worst perimeter	worst area	worst smoothness	worst compactness	worst concavity	worst concave points	worst symmetry	worst fractal dimension	malignant
0.3001	0.14710	0.2419	0.07871	...	17.33	184.60	2019.0	0.1622	0.6656	0.7119	0.2654	0.4601	0.11890	0
0.0869	0.07017	0.1812	0.05667	...	23.41	158.80	1956.0	0.1238	0.1866	0.2416	0.1860	0.2750	0.08902	0
0.1974	0.12790	0.2069	0.05999	...	25.53	152.50	1709.0	0.1444	0.4245	0.4504	0.2430	0.3613	0.08758	0
0.2414	0.10520	0.2597	0.09744	...	26.50	98.87	567.7	0.2098	0.8663	0.6869	0.2575	0.6638	0.17300	0
0.1980	0.10430	0.1809	0.05883	...	16.67	152.20	1575.0	0.1374	0.2050	0.4000	0.1625	0.2364	0.07678	0

Figure 3.13: Head of the DataFrame

Let us look at the class distribution:

```
df.malignant.value_counts()
```



```
1    357
0    212
Name: malignant, dtype: int64
```

Figure 3.14: Value counts of dependent variable

So, there are 357 records out of 569 that are of class ‘malignant’ and 212 records out of 569 which are of class ‘benign’.

To set the baseline, if the model predicts all the records as malignant, even then the accuracy of the model will be as follows:

So, the dataset is slightly class imbalanced, and even if the model predicts all the records as one class, the model will be 62.74% accurate.

For now, we will continue modeling without treating the class imbalance.

Data pre-processing

Store the features data in X and the target in y, as shown:

Perform the train-test split; reserve 80% data for training and 20% data for testing:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.2, rando
m_state=42)
```

Scale the data using

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X_train=scaler.fit_transform(X_train)

X_test=scaler.transform(X_test)
```

Scaler is fit using the training dataset and used to transform the training dataset using the fit_transform method. Later, the same scale is used to transform the testing dataset.

Now, both the training and testing datasets are ready to train the model.

Modeling

Import the Sequential and Dense functions from the keras library:

```
from keras.models import Sequential
```

```
from keras.layers import Dense
```

Create a new Sequential object to build the model upon:

```
model = Sequential()
```

Experiment 1 - Hidden layer -1, epoch-100 – shallow neural network

In this experiment, we will build a shallow neural network with just one hidden layer, as shown in Figure

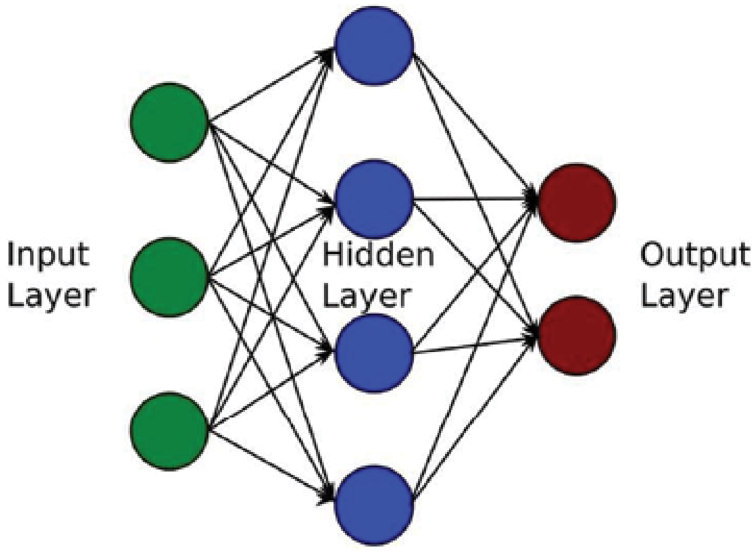


Figure 3.15: Neural network with one hidden layer

Add the first hidden layer with 16 neurons. The activation function used will be and the name of the layer is The input layer will contain the number of neurons equal to the number of features in the dataset, that is, 30, as shown:

```
model.add(Dense(16, input_shape=(30, ), activation='relu', name='dense_1'))
```

Finally, create the output layer with the sigmoid activation function. Since we are going to use sparse_categorical_crossentropy as the loss function, the model will spit out the probability of the input belonging to a certain class.

Since it is a binary classification problem, we need two neurons in the last layer, as follows:

model.add(Dense(2, activation='sigmoid', name='dense_output'))

If you plan to use the binary_crossentropy loss function for binary classification problems, then it would be a mandatory step to convert the target data into 0 and 1 format before the training. You can use just one neuron in the output layer, since in that case, the model is just going to spit the probability of the input data belonging to class 1.

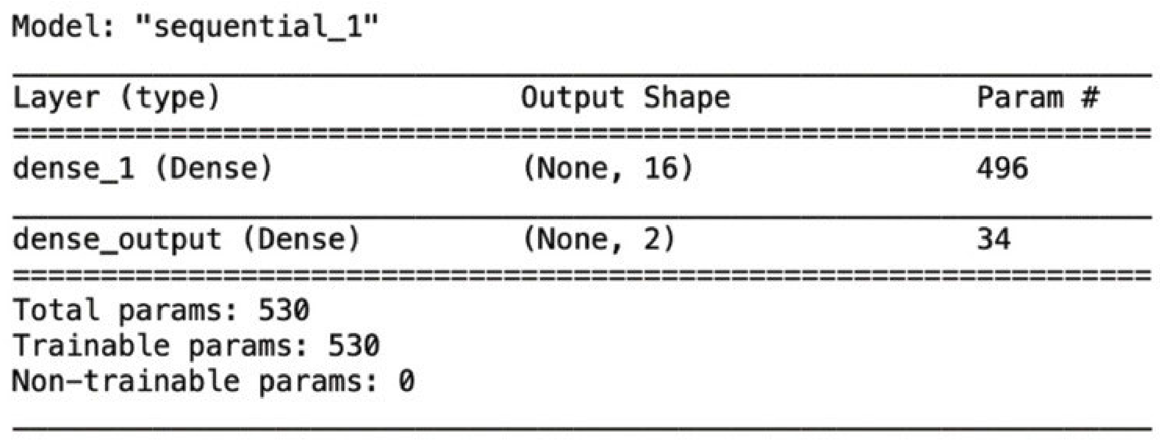
Once the whole network is built, we will compile the network using the adam optimizer and sparse categorical cross entropy loss function:

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', met rics=['accuracy'])

After compilation, we will see the summary of the model:

model.summary()

Model Summary is depicted in Figure 3.16.



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 16)                496
_________________________________________________________________
dense_output (Dense)         (None, 2)                 34
=================================================================
Total params: 530
Trainable params: 530
Non-trainable params: 0
_________________________________________________________________
```

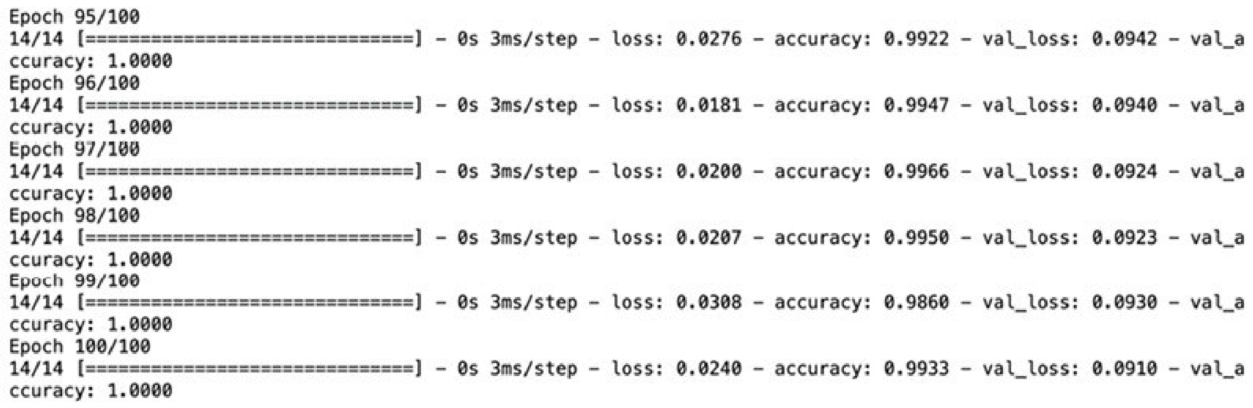
Figure 3.16: Model summary

The summary shows exactly what we have built. The artificial neural network we built consists of 30 neurons in the input layer, since there are 30 features in the dataset. Then, there is a hidden layer comprising 16 neurons. And finally, there is an output layer that has two neurons since it is a binary classification. Hence, each output neuron will hold the probability of the input belonging to that class, and the class of the input will be predicted based on the highest probability in both the output neurons.

First, let us fit the model on the training dataset. In this step, the model is being trained using the training dataset X_train and We will train the model for 100 epochs:

result = model.fit(X_train, y_train, epochs=100, validation_split=0.05)

Model run summary is depicted in Figure



```
Epoch 95/100
14/14 [==============================] - 0s 3ms/step - loss: 0.0276 - accuracy: 0.9922 - val_loss: 0.0942 - val_a
ccuracy: 1.0000
Epoch 96/100
14/14 [==============================] - 0s 3ms/step - loss: 0.0181 - accuracy: 0.9947 - val_loss: 0.0940 - val_a
ccuracy: 1.0000
Epoch 97/100
14/14 [==============================] - 0s 3ms/step - loss: 0.0200 - accuracy: 0.9966 - val_loss: 0.0924 - val_a
ccuracy: 1.0000
Epoch 98/100
14/14 [==============================] - 0s 3ms/step - loss: 0.0207 - accuracy: 0.9950 - val_loss: 0.0923 - val_a
ccuracy: 1.0000
Epoch 99/100
14/14 [==============================] - 0s 3ms/step - loss: 0.0308 - accuracy: 0.9860 - val_loss: 0.0930 - val_a
ccuracy: 1.0000
Epoch 100/100
14/14 [==============================] - 0s 3ms/step - loss: 0.0240 - accuracy: 0.9933 - val_loss: 0.0910 - val_a
ccuracy: 1.0000
```

Figure 3.17: Model run summary

Model evaluation

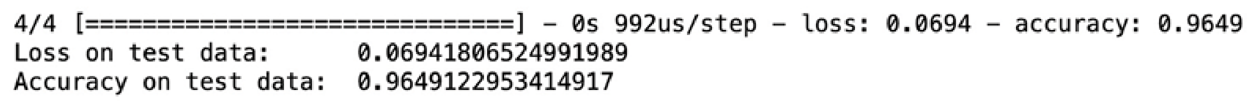
Once the model is trained using the training dataset, it is time to evaluate the model performance using the test dataset:

loss, acc = model.evaluate(X_test, y_test)

print('Loss on test data: ', loss)

print('Accuracy on test data: ', acc)

Model evaluation summary is depicted in Figure



```
4/4 [==============================] - 0s 992us/step - loss: 0.0694 - accuracy: 0.9649
Loss on test data:      0.06941806524991989
Accuracy on test data:  0.9649122953414917
```

Figure 3.18: Model evaluation summary

We can see that the accuracy of the model is approximately 96% and the loss is 0.069.

Performance metrics

Let us first create an array of the predicted classes, as shown:

y_pred=model.predict(X_test)

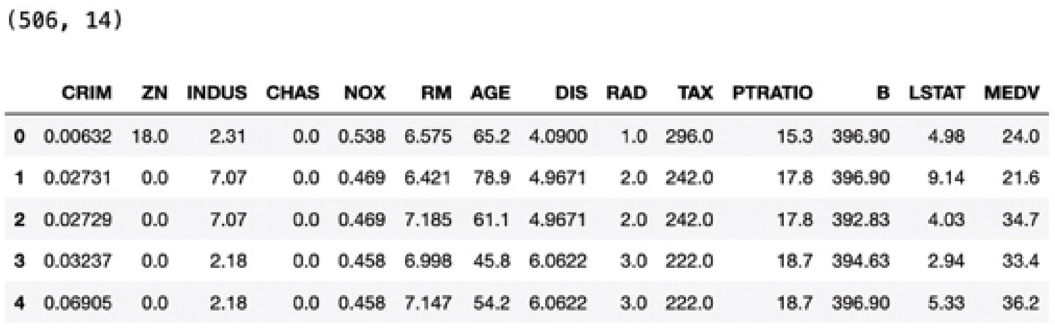
y_pred=np.argmax(y_pred,axis=1)

Now, since we have both the list of the actual classes and that of the predicted classes for the test data, let us generate the classification report that shows all the important metrics with respect to the model:

```
from sklearn.metrics import classification_report, confusion_matrix
```

```
print(classification_report(y_pred, y_test))
```

This displays the classification report as shown Figure



```
(506, 14)
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

Figure 3.19: Classification report

Confusion matrix

Further, let us build and visualize the confusion matrix for the binary classification. We have a function called plot_confusion_matrix() that has been taken directly from scikit-learn’s website. This is the code they provide to plot the confusion matrix. Hence, it can be used as follows:

```
def plot_confusion_matrix(cm, classes,
                  normalize=False,

                  title='Confusion matrix',

                  cmap=plt.cm.Blues):

"""

This function prints and plots the confusion matrix.

Normalization can be applied by setting `normalize=True`.

«»»

plt.imshow(cm, interpolation='nearest', cmap=cmap)

plt.title(title)

plt.colorbar()

tick_marks = np.arange(len(classes))

plt.xticks(tick_marks, classes, rotation=45)

plt.yticks(tick_marks, classes)

if normalize:

    cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    print("Normalized confusion matrix")
else:
    print('Confusion matrix, without normalization')
print(cm)
import itertools

thresh = cm.max() / 2.
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, cm[i, j],
        horizontalalignment="center",
        color="white" if cm[i, j] > thresh else "black")
plt.tight_layout()
plt.ylabel('True label')
plt.xlabel('Predicted label')
```

Now, let us create a function that will build and visualize the confusion matrix as shown in the following code snippet:

```
def create_confusion_matrix(y_test, y_pred):

    from sklearn.metrics import confusion_matrix

    cnf_matrix = confusion_matrix(y_test, y_pred)

    plot_confusion_matrix(cnf_matrix, [0,1])
```

Create the confusion matrix for the model built in Experiment 1, as follows:

```
create_confusion_matrix(y_test, y_pred)
```

This generates the confusion matrix shown in Figure

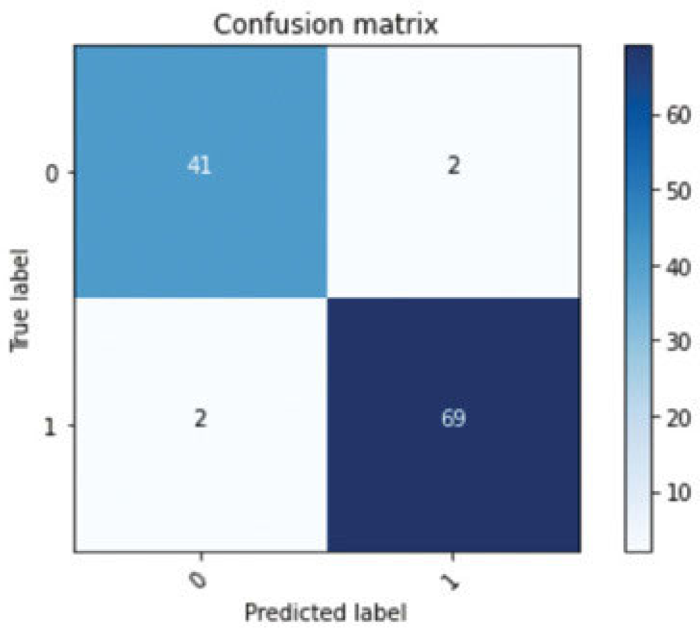


Figure 3.20: Confusion matrix

Based on the confusion matrix, the following result was obtained:

Out of 43 benign records, 41 were correctly predicted as benign and 2 were incorrectly predicted as malignant.

Out of 71 malignant records, 69 were correctly predicted as malignant and 2 were incorrectly predicted as benign.

Experiment 2 - hidden layer -2, epoch-100 – deep neural network

In this experiment, we will build a deep neural network with more than one hidden layer, as shown in Figure

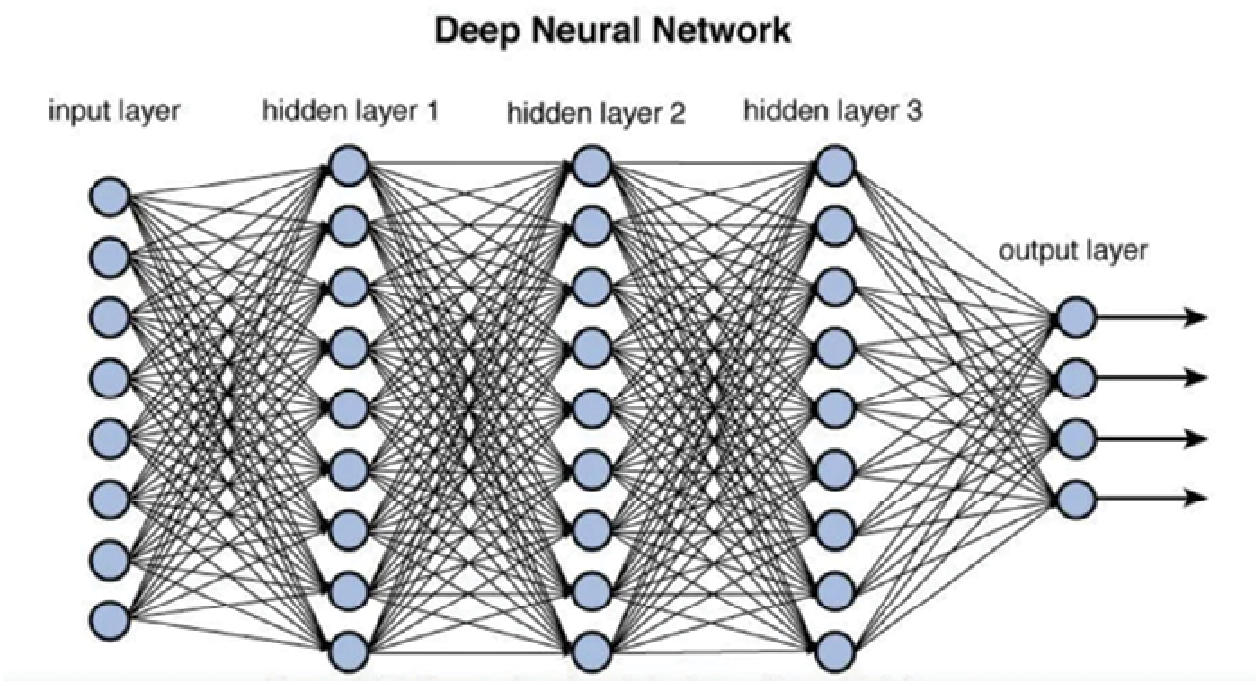


Figure 3.21: Deep neural network

Since the steps involved in building a neural network are already explained in Experiment 1, we will use the following code to build a deep neural network with two hidden layers, each consisting of 16 neurons. All the other details will remain the same as in Experiment 1:

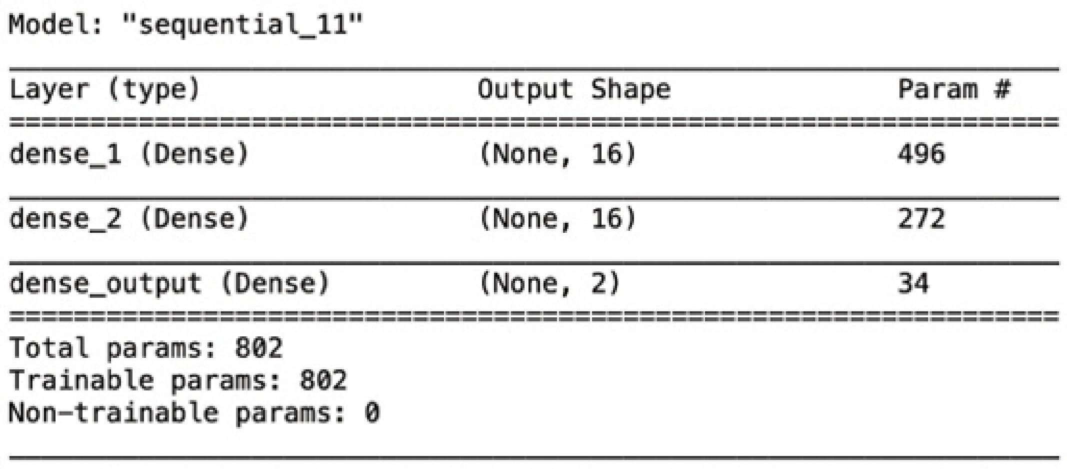
```
model.add(Dense(16, input_shape=(30, ), activation='relu', name='dense_1'))

model.add(Dense(16, activation='relu', name='dense_2'))
```

model.add(Dense(2, activation='sigmoid', name='dense_output'))

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

model.summary()

This displays the model summary as per Figure



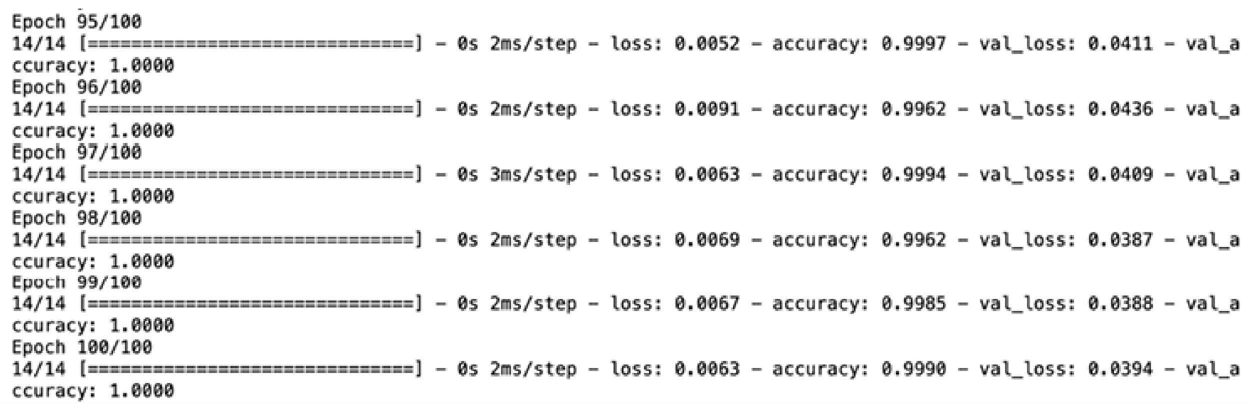
```
Model: "sequential_11"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 16)                496
_________________________________________________________________
dense_2 (Dense)              (None, 16)                272
_________________________________________________________________
dense_output (Dense)         (None, 2)                 34
=================================================================
Total params: 802
Trainable params: 802
Non-trainable params: 0
_________________________________________________________________
```

Figure 3.22: Model summary

Let us train the model with 100 epochs:

result = model.fit(X_train, y_train, epochs=100, validation_split=0.05)

This displays the model run summary as per Figure



```
Epoch 95/100
14/14 [==============================] - 0s 2ms/step - loss: 0.0052 - accuracy: 0.9997 - val_loss: 0.0411 - val_a
ccuracy: 1.0000
Epoch 96/100
14/14 [==============================] - 0s 2ms/step - loss: 0.0091 - accuracy: 0.9962 - val_loss: 0.0436 - val_a
ccuracy: 1.0000
Epoch 97/100
14/14 [==============================] - 0s 3ms/step - loss: 0.0063 - accuracy: 0.9994 - val_loss: 0.0409 - val_a
ccuracy: 1.0000
Epoch 98/100
14/14 [==============================] - 0s 2ms/step - loss: 0.0069 - accuracy: 0.9962 - val_loss: 0.0387 - val_a
ccuracy: 1.0000
Epoch 99/100
14/14 [==============================] - 0s 2ms/step - loss: 0.0067 - accuracy: 0.9985 - val_loss: 0.0388 - val_a
ccuracy: 1.0000
Epoch 100/100
14/14 [==============================] - 0s 2ms/step - loss: 0.0063 - accuracy: 0.9990 - val_loss: 0.0394 - val_a
ccuracy: 1.0000
```

Figure 3.23: Model run summary

Model evaluation

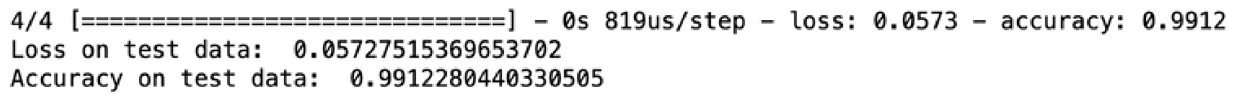
Once the model is trained using the training dataset, it is time to evaluate the model performance using the test dataset:

loss, acc = model.evaluate(X_test, y_test)

print('Loss on test data: ', loss)

print('Accuracy on test data: ', acc)

This displays model evaluation summary as per Figure



```
4/4 [==============================] - 0s 819us/step - loss: 0.0573 - accuracy: 0.9912
Loss on test data:  0.05727515369653702
Accuracy on test data:  0.9912280440330505
```

Figure 3.24: Model evaluation summary

Here, the accuracy of the model is approximately 99% and the loss is 0.057, which is better than the shallow neural network.

Performance metrics

Let us first create an array of the predicted classes:

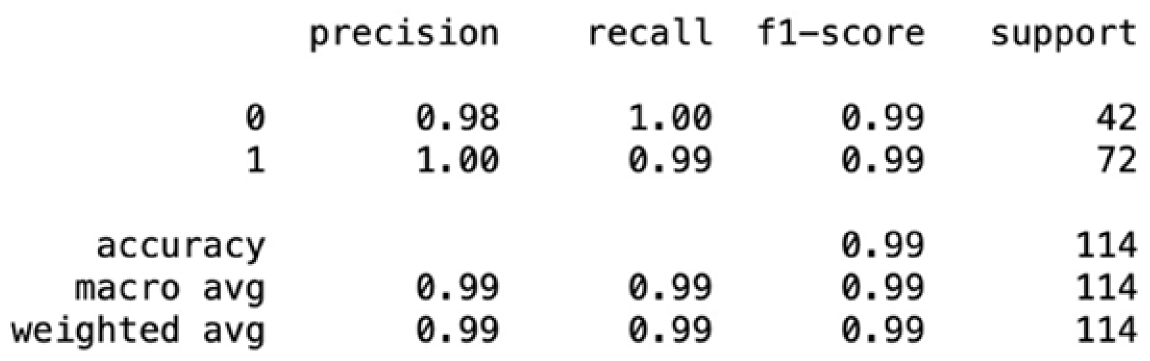
y_pred=model.predict_classes(X_test)

Since we have both the list of the actual and the predicted classes for the test data, let us generate the classification report that shows all the important metrics with respect to the model:

from sklearn.metrics import classification_report, confusion_matrix

print(classification_report(y_pred, y_test))

This displays the classification report as shown in Figure



```
              precision    recall  f1-score   support

           0       0.98      1.00      0.99        42
           1       1.00      0.99      0.99        72

    accuracy                           0.99       114
   macro avg       0.99      0.99      0.99       114
weighted avg       0.99      0.99      0.99       114
```

Figure 3.25: Classification report

Confusion matrix

Since we have already created the create_confusion_matrix function in Experiment 1, we will just pass y_test and y_pred from the model in Experiment 2 to build and visualize the confusion matrix for the deep neural network:

```
create_confusion_matrix(y_test, y_pred)
```

This displays the classification report as per Figure

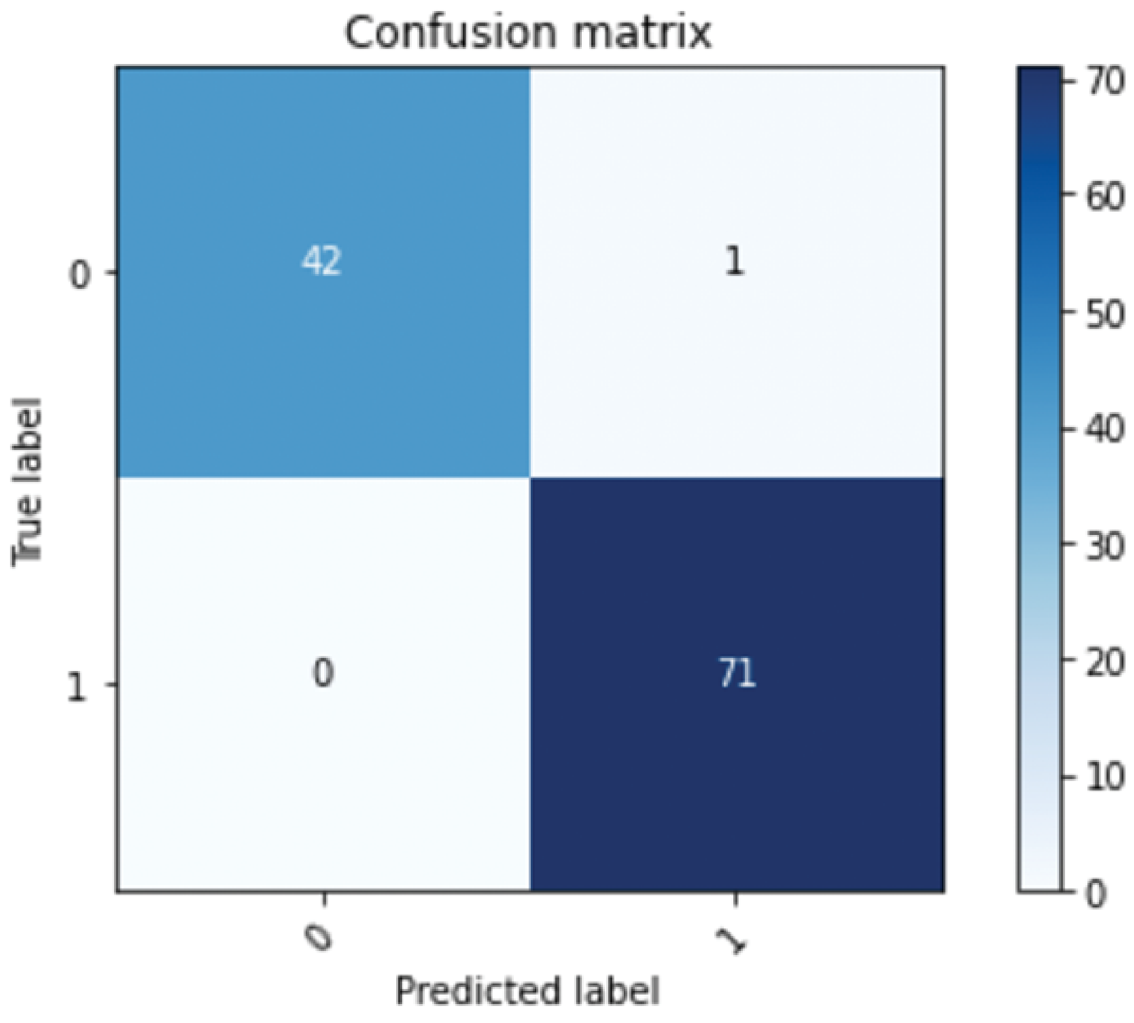


Figure 3.26: Confusion Matrix

Based on the confusion matrix, the following result was obtained:

Out of 43 benign records, 42 were correctly predicted as benign and 1 was incorrectly predicted as malignant.

Out of 71 malignant records, all 71 were correctly predicted as malignant.

Hence, looking at the performance metrics of Experiment 1 and Experiment 2, it is clear that a deep neural network performs better than a shallow neural network with only 1 hidden layer.

Building a regression model using neural network

As we know, regression models are used to predict continuous variables. We can use linear regression-based traditional machine learning to do the same.

Problem statement

The problem that we are trying to solve here is predicting the price of the house based on various features of the house, like the number of rooms, number of bathrooms, locality, and so on. Now, we know these features do determine the price of the house. With the use of a neural network-based predictive model, we will build an optimum application that can predict the most accurate price of the house based on the various features of the house.

Dataset

We will use the Boston housing dataset within the sklearn.datasets library to build a neural network-based regression model.

Firstly, import the required libraries:

```
from sklearn.datasets import load_boston
```

```
import pandas as pd
```

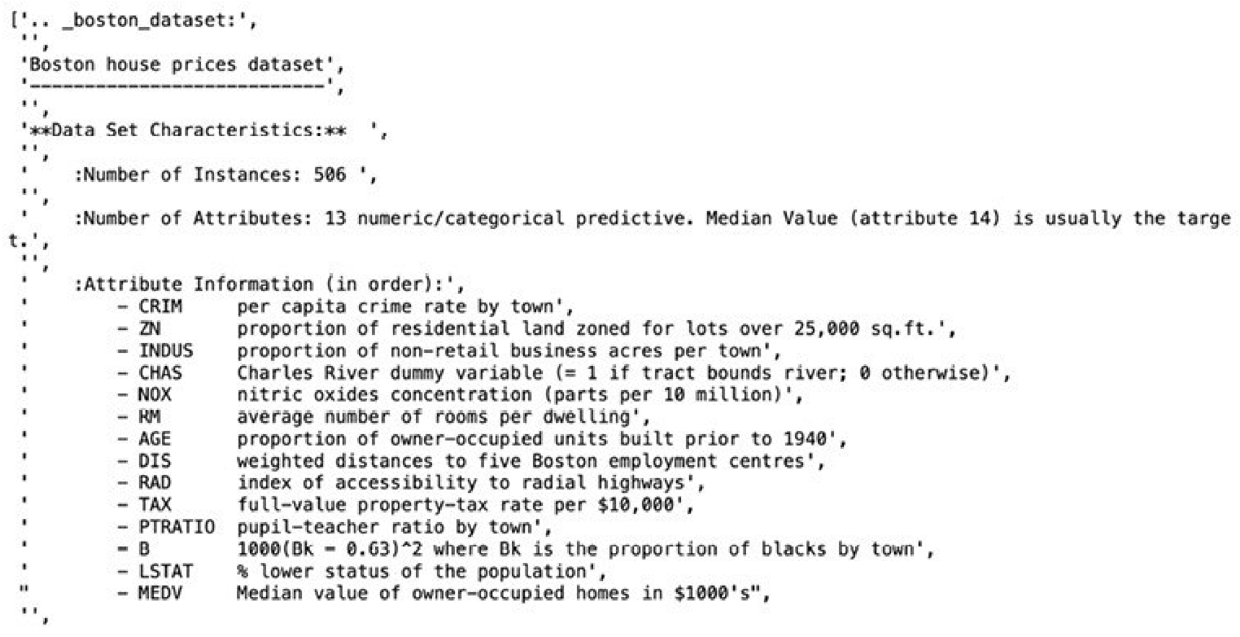
Then, load the Boston housing dataset:

```
boston_housing_dataset = load_boston()
```

First, let us understand what is there in the boston_housing_dataset object:

```
boston_housing_dataset.DESCR.splitlines()
```

Dataset desription is depicted in Figure



```
['.. _boston_dataset:',
 '',
 'Boston house prices dataset',
 '---------------------------',
 '',
 '**Data Set Characteristics:**  ',
 '',
 '    :Number of Instances: 506 ',
 '',
 '    :Number of Attributes: 13 numeric/categorical predictive. Median Value (attribute 14) is usually the targe
t.',
 '',
 '    :Attribute Information (in order):',
 '        - CRIM     per capita crime rate by town',
 '        - ZN       proportion of residential land zoned for lots over 25,000 sq.ft.',
 '        - INDUS    proportion of non-retail business acres per town',
 '        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)',
 '        - NOX      nitric oxides concentration (parts per 10 million)',
 '        - RM       average number of rooms per dwelling',
 '        - AGE      proportion of owner-occupied units built prior to 1940',
 '        - DIS      weighted distances to five Boston employment centres',
 '        - RAD      index of accessibility to radial highways',
 '        - TAX      full-value property-tax rate per $10,000',
 '        - PTRATIO  pupil-teacher ratio by town',
 '        - B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town',
 '        - LSTAT    % lower status of the population',
 "        - MEDV     Median value of owner-occupied homes in $1000's",
 '',
```

Figure 3.27: boston_housing_dataset features

In this dataset, there are 506 observations. There are 14 attributes, out of which 13 are predictors and MEDV is the target variable. MEDV is the price of the house, and the other 13 attributes are the features of the house.

We intend to create a neural network-based regression model, which uses the 13 independent variables to predict the dependent variable, that is, the price of the house.

Data pre-processing

First, let us create a pandas DataFrame using

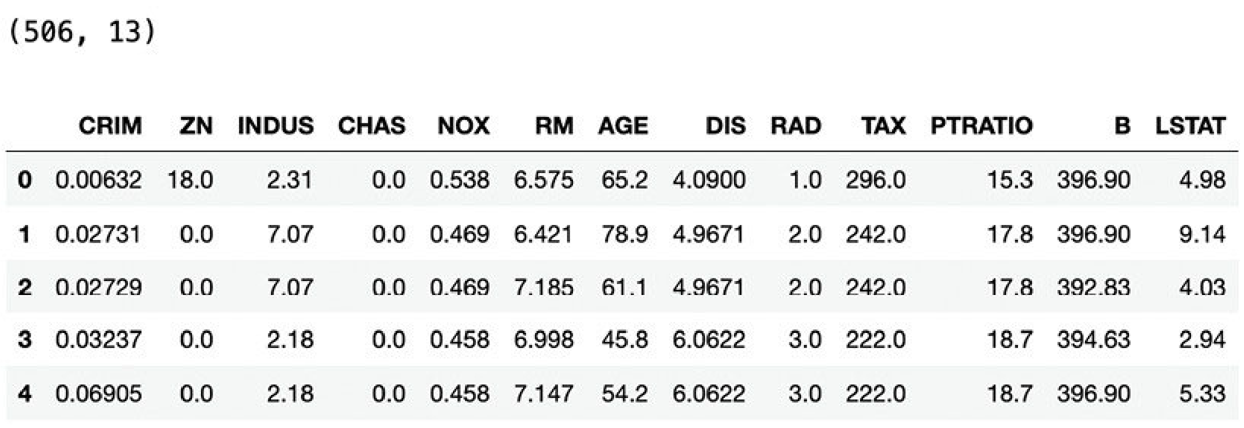
df = pd.DataFrame(boston_housing_dataset.data, columns=boston_housing_dataset.feature_names)

Let us see the shape of the newly created DataFrame and print the top 5 records:

print(df.shape)

df.head()

First few records of the dataset is depicted in Figure



```
(506, 13)
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33

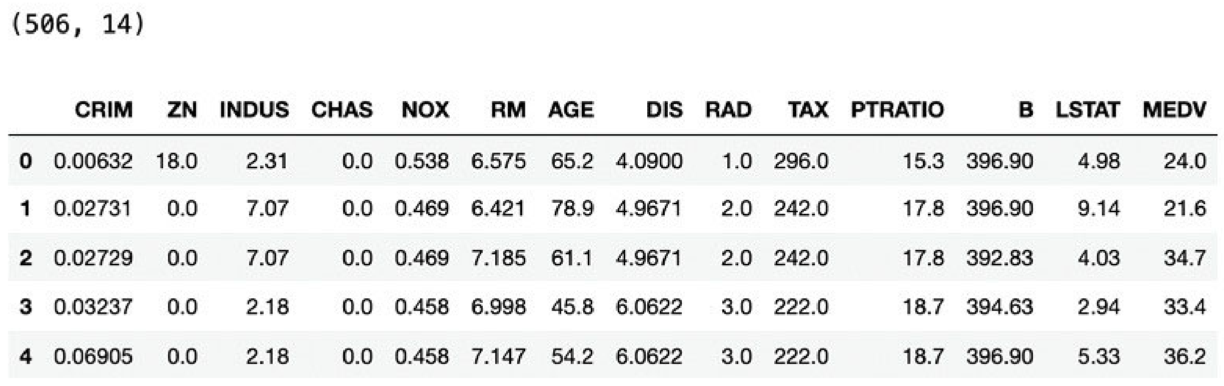
Figure 3.28: DataFrame head

So, the newly created DataFrame contains all the 506 records and 13 independent variables. We need to add the target variable MEDV in the DataFrame, as shown here:

```
df['MEDV']=boston_housing_dataset.target
```

Let us again see the shape of the DataFrame and print the top 5 records:

```
print(df.shape)

df.head()
```



```
(506, 14)
```

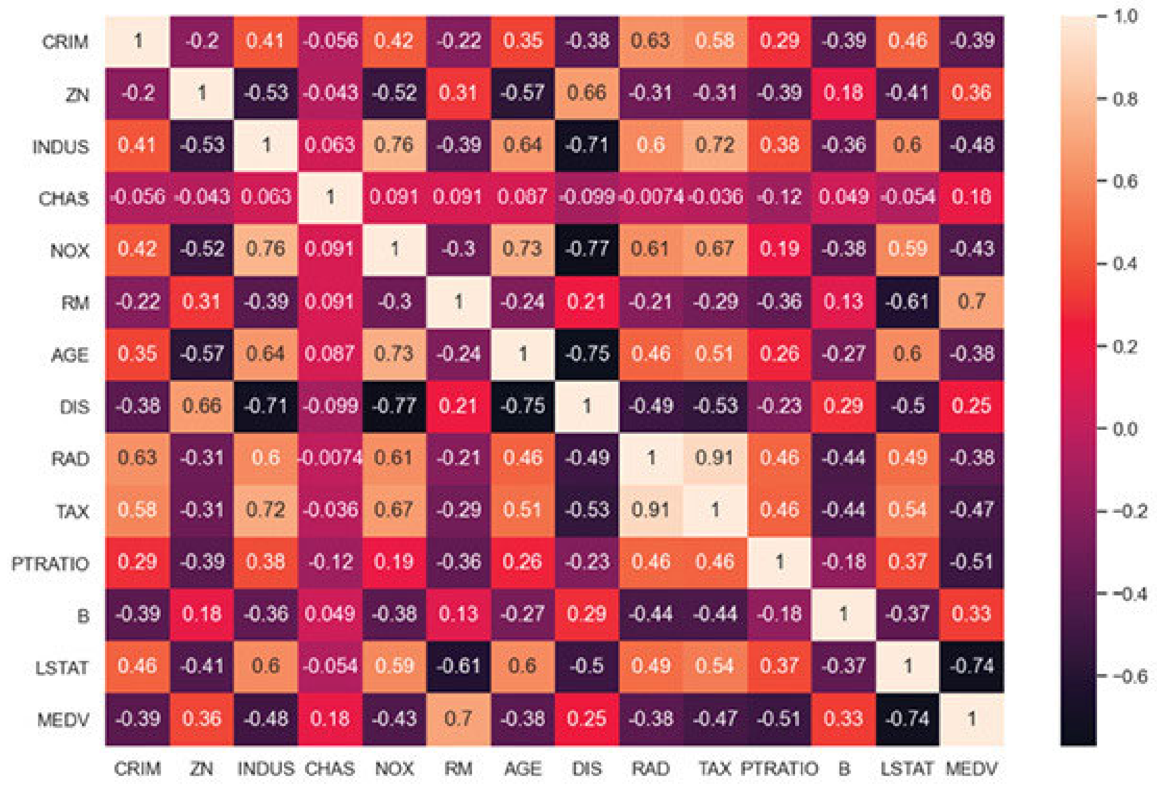
	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

Figure 3.29: DataFrame head

Let us see which variable is highly correlated with the target variable

```
import seaborn as sns

sns.set(rc={'figure.figsize':(12,8)})

sns.heatmap(df.corr(), annot=True)
```

This will display a heatmap for the correlation of all the features, as shown in the following screenshot:



	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
CRIM	1	-0.2	0.41	-0.056	0.42	-0.22	0.35	-0.38	0.63	0.58	0.29	-0.39	0.46	-0.39
ZN	-0.2	1	-0.53	-0.043	-0.52	0.31	-0.57	0.66	-0.31	-0.31	-0.39	0.18	-0.41	0.36
INDUS	0.41	-0.53	1	0.063	0.76	-0.39	0.64	-0.71	0.6	0.72	0.38	-0.36	0.6	-0.48
CHAS	-0.056	-0.043	0.063	1	0.091	0.091	0.087	-0.099	-0.0074	-0.036	-0.12	0.049	-0.054	0.18
NOX	0.42	-0.52	0.76	0.091	1	-0.3	0.73	-0.77	0.61	0.67	0.19	-0.38	0.59	-0.43
RM	-0.22	0.31	-0.39	0.091	-0.3	1	-0.24	0.21	-0.21	-0.29	-0.36	0.13	-0.61	0.7
AGE	0.35	-0.57	0.64	0.087	0.73	-0.24	1	-0.75	0.46	0.51	0.26	-0.27	0.6	-0.38
DIS	-0.38	0.66	-0.71	-0.099	-0.77	0.21	-0.75	1	-0.49	-0.53	-0.23	0.29	-0.5	0.25
RAD	0.63	-0.31	0.6	-0.0074	0.61	-0.21	0.46	-0.49	1	0.91	0.46	-0.44	0.49	-0.38
TAX	0.58	-0.31	0.72	-0.036	0.67	-0.29	0.51	-0.53	0.91	1	0.46	-0.44	0.54	-0.47
PTRATIO	0.29	-0.39	0.38	-0.12	0.19	-0.36	0.26	-0.23	0.46	0.46	1	-0.18	0.37	-0.51
B	-0.39	0.18	-0.36	0.049	-0.38	0.13	-0.27	0.29	-0.44	-0.44	-0.18	1	-0.37	0.33
LSTAT	0.46	-0.41	0.6	-0.054	0.59	-0.61	0.6	-0.5	0.49	0.54	0.37	-0.37	1	-0.74
MEDV	-0.39	0.36	-0.48	0.18	-0.43	0.7	-0.38	0.25	-0.38	-0.47	-0.51	0.33	-0.74	1

Figure 3.30: Correlation heatmap

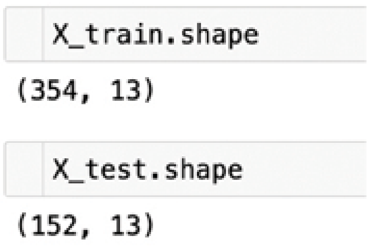
It is clearly visible from the heatmap that the RM attribute, which is the average number of rooms per dwelling, and which is the % lower status of the population, are highly correlated with the target variable. However, RM is positively correlated and LSTAT is negatively correlated. This is good information to retain for later evaluation purposes.

For now, let us continue with the modeling process.

The next step in the data pre-processing stage will be to distribute the data into train and test datasets, as shown in the following code snippet:

```
from sklearn.model_selection import train_test_split

X = df.loc[:, df.columns != 'MEDV']

y = df.loc[:, df.columns == 'MEDV']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_
state=123)
```

Let us see the shape of both training and testing datasets, as shown in the following screenshot:



```
X_train.shape
(354, 13)
X_test.shape
(152, 13)
```

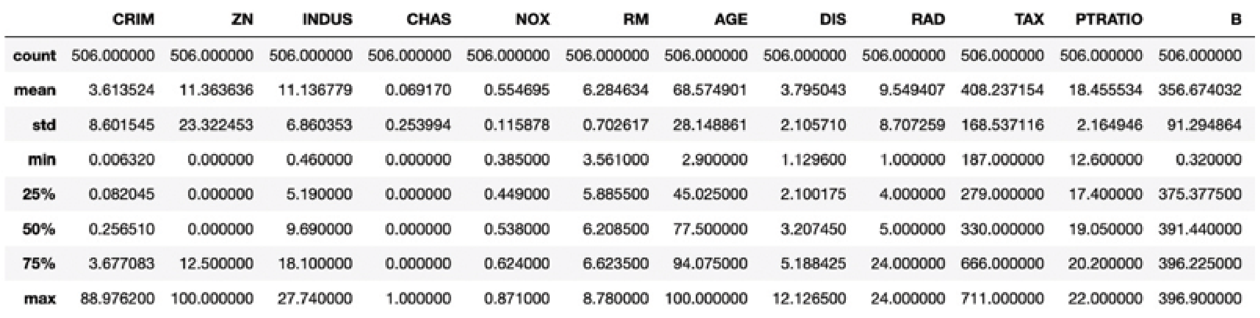
Figure 3.31: Train and test dataset shape

As we can see, 354 records fell into the training dataset and 152 records fell into the testing dataset.

Now, let us see how the data is distributed statistically to figure out if scaling is required:

df.describe()

This displays the statistical description of the dataset:



	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795043	9.549407	408.237154	18.455534	356.674032
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105710	8.707259	168.537116	2.164946	91.294864
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000	12.600000	0.320000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175	4.000000	279.000000	17.400000	375.377500
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207450	5.000000	330.000000	19.050000	391.440000
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188425	24.000000	666.000000	20.200000	396.225000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000	22.000000	396.900000

Figure 3.32: Statistical description of the dataset

As shown, the attribute values are not in the same range and vary significantly. Attributes with significant difference in scales may result in biases in the model toward the attribute that is higher in range as compared to other attributes. Also, the significant difference in the scale of the values of various attributes takes more time for the model to converge.

Hence, we will use the StandardScaler module with the sklearn library to scale the data.

First, let us import the library:

from sklearn.preprocessing import StandardScaler

Then, we will create an object of the StandardScaler module:

scaler = StandardScaler()

Further, use the training data to fit the scalar object and transform the training data using the trained scaler:

```
X_train_scaled=scaler.fit_transform(X_train)
```

Since the scaler object has never seen the test data and has been trained only on the training data, we will use only the transform method for the test data, as shown:

```
X_test_scaled=scaler.transform(X_test)
```

Modeling

To build the model, we will first import the Sequential and Dense modules within the keras library, as shown:

```
from keras.models import Sequential
```

```
from keras.layers import Dense
```

We will create a model instance using the Sequential module:

```
model = Sequential()
```

First, create the input layer for all the 13 input variables in the dataset. Further, create the first hidden layer with 128 neurons. Activate the neurons using the relu activation function, and name the layer

```
model.add(Dense(128, input_shape=
(13, ), activation='relu', name='dense_1'))
```

Next, create another hidden layer with 128 neurons and relu activation:

```
model.add(Dense(128, activation='relu', name='dense_2'))
```

Finally, create the output layer with one neuron, which will hold the predicted price of the house. Since we need a regression model, we will use the linear activation function:

```
model.add(Dense(1, activation='linear', name='dense_output'))
```

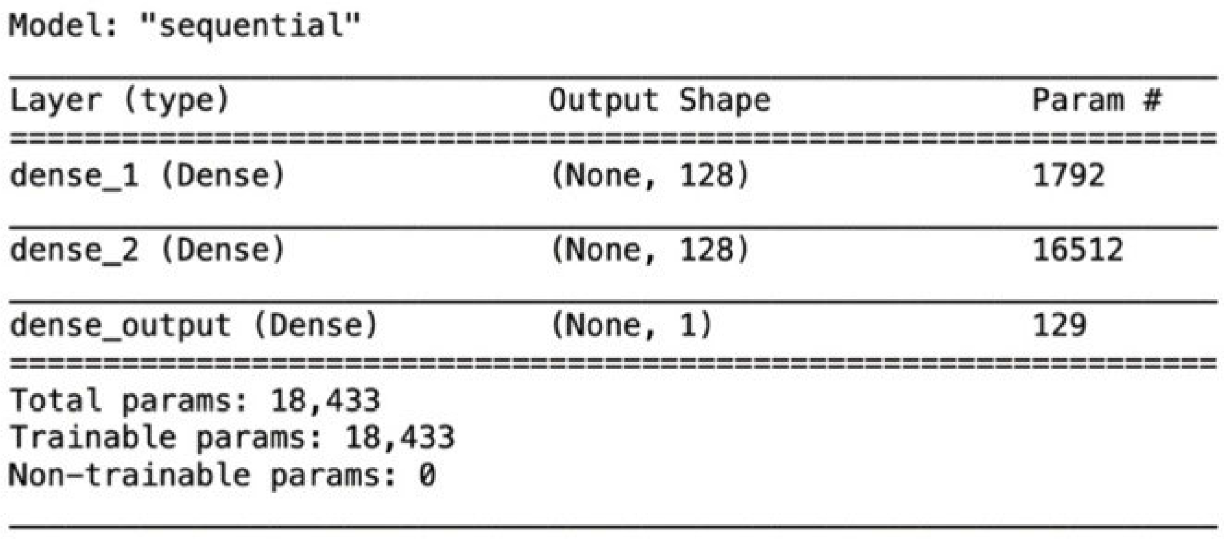
We will use the adam optimizer to optimize the learnable parameters during the training process. Since this is a regression problem, we will use mean squared error or mse as the loss function. And finally, we will use mean absolute error or mae as the performance metrics to evaluate the performance of the model:

```
model.compile(optimizer='adam', loss='mse', metrics=['mae'])
```

Let us look at the model summary:

```
model.summary()
```

This displays the summary of the model:



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 128)               1792
_________________________________________________________________
dense_2 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_output (Dense)         (None, 1)                 129
=================================================================
Total params: 18,433
Trainable params: 18,433
Non-trainable params: 0
_________________________________________________________________
```

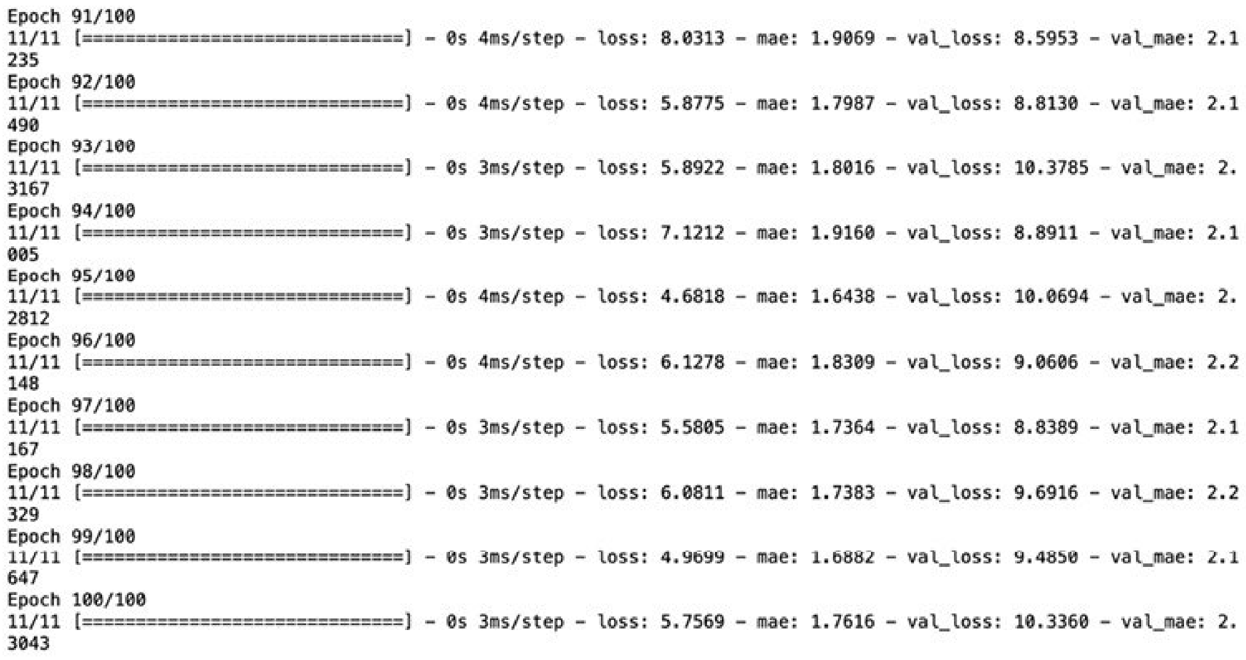
Figure 3.33: Model summary

As we can see, there are 18,433 trainable parameters, which are the weights of every input to the neurons in the network and biases for each neuron in the network.

Let us start the training process. We will train the model for 100 epochs and validate the model during the training process using the validation split of 5% from the training dataset:

history = model.fit(X_train, y_train, epochs=100, validation_split=0.05)

This displays the model run summary:



```
Epoch 91/100
11/11 [==============================] - 0s 4ms/step - loss: 8.0313 - mae: 1.9069 - val_loss: 8.5953 - val_mae: 2.1
235
Epoch 92/100
11/11 [==============================] - 0s 4ms/step - loss: 5.8775 - mae: 1.7987 - val_loss: 8.8130 - val_mae: 2.1
490
Epoch 93/100
11/11 [==============================] - 0s 3ms/step - loss: 5.8922 - mae: 1.8016 - val_loss: 10.3785 - val_mae: 2.
3167
Epoch 94/100
11/11 [==============================] - 0s 3ms/step - loss: 7.1212 - mae: 1.9160 - val_loss: 8.8911 - val_mae: 2.1
005
Epoch 95/100
11/11 [==============================] - 0s 4ms/step - loss: 4.6818 - mae: 1.6438 - val_loss: 10.0694 - val_mae: 2.
2812
Epoch 96/100
11/11 [==============================] - 0s 4ms/step - loss: 6.1278 - mae: 1.8309 - val_loss: 9.0606 - val_mae: 2.2
148
Epoch 97/100
11/11 [==============================] - 0s 3ms/step - loss: 5.5805 - mae: 1.7364 - val_loss: 8.8389 - val_mae: 2.1
167
Epoch 98/100
11/11 [==============================] - 0s 3ms/step - loss: 6.0811 - mae: 1.7383 - val_loss: 9.6916 - val_mae: 2.2
329
Epoch 99/100
11/11 [==============================] - 0s 3ms/step - loss: 4.9699 - mae: 1.6882 - val_loss: 9.4850 - val_mae: 2.1
647
Epoch 100/100
11/11 [==============================] - 0s 3ms/step - loss: 5.7569 - mae: 1.7616 - val_loss: 10.3360 - val_mae: 2.
3043
```

Figure 3.34: Model run summary

Model evaluation

Once the model is trained, let us use the testing dataset to evaluate the model performance. Let us use the following code snippet using the plotly Python library to plot the loss and mean absolute error of the model.

First, import the required Python libraries:

```
from plotly.subplots import make_subplots

import plotly.graph_objects as go

import math

import numpy as np
```

Further, create the figure using the plotly library:

```
fig = go.Figure()

fig.add_trace(go.Scattergl(y=history.history['loss'],

                    name='Train'))

fig.add_trace(go.Scattergl(y=history.history['val_loss'],

                    name='Valid'))
```

```
fig.update_layout(height=500, width=700,

xaxis_title='Epoch',

                  yaxis_title='Loss')

fig.show()
```

This displays a model loss plot, as shown in the following screenshot:

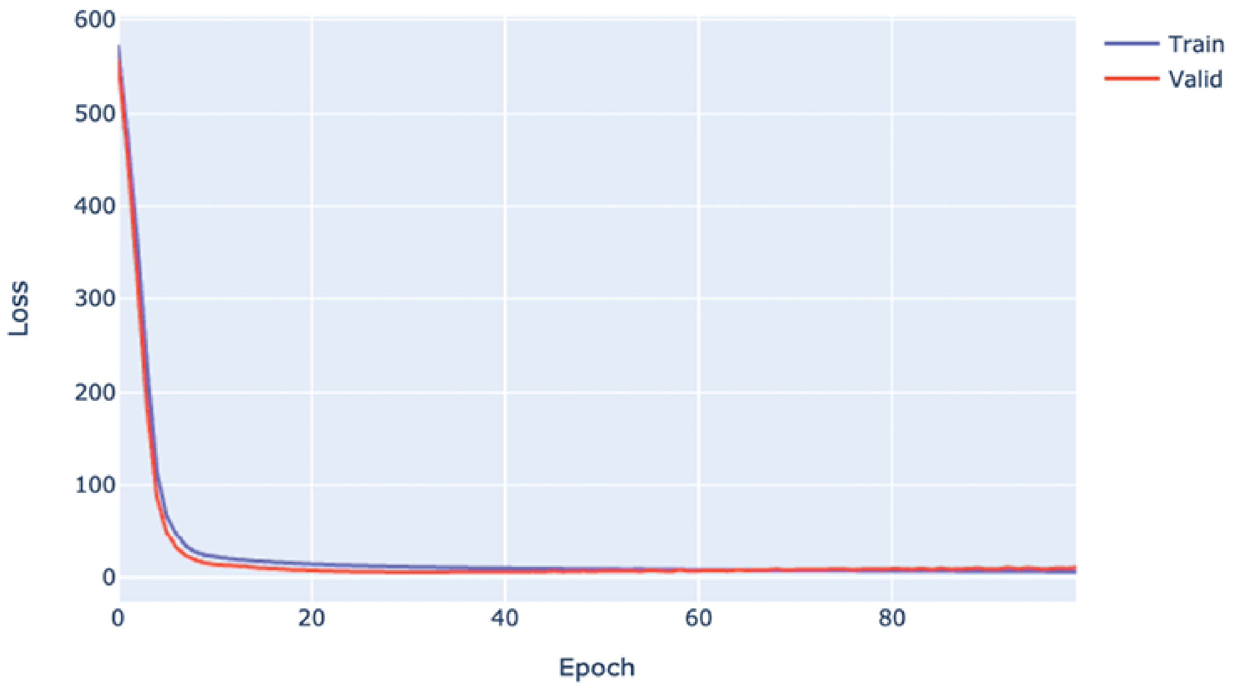


Figure 3.35: Model loss

Let us build the same plot for mean absolute error:

```
fig = go.Figure()
```

```
fig.add_trace(go.Scattergl(y=history.history['mae'],

                    name='Train'))

fig.add_trace(go.Scattergl(y=history.history['val_mae'],

                    name='Valid'))

fig.update_layout(height=500, width=700,

                  xaxis_title='Epoch',

                  yaxis_title='Mean Absolute Error')

fig.show()
```

This displays a mean absolute error plot, as shown in the following screenshot:

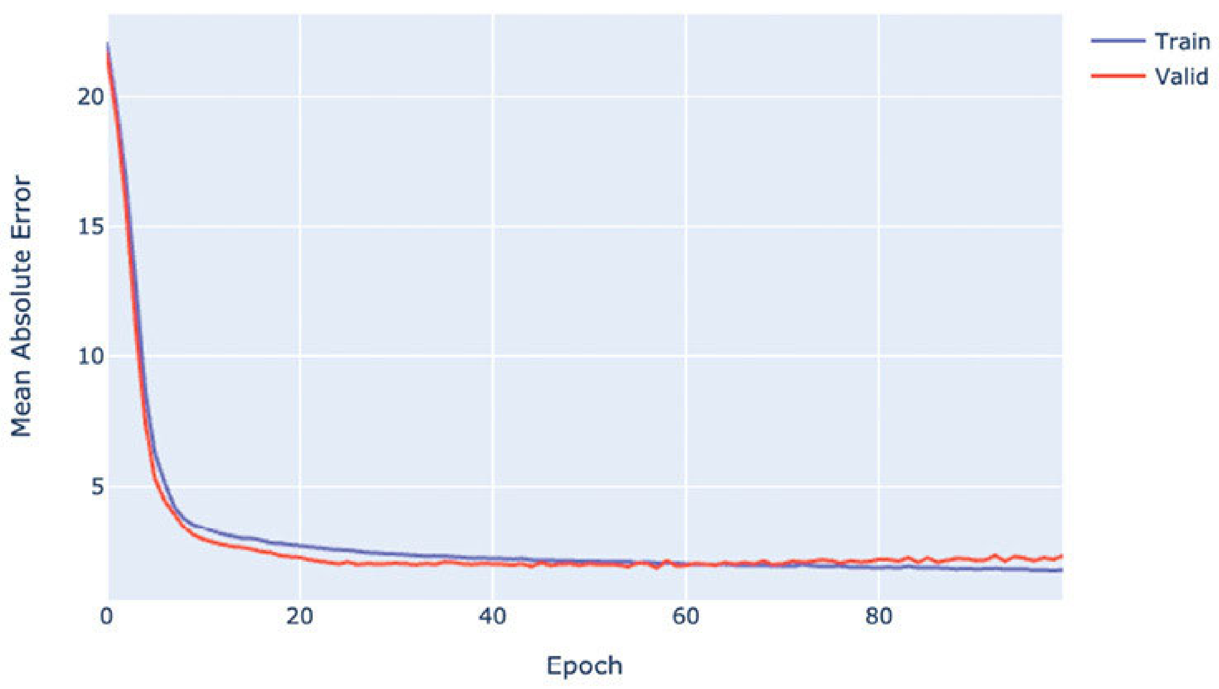


Figure 3.36: Model error plot

Finally, let us use the test dataset to evaluate the model:

mse_nn, mae_nn = model.evaluate(X_test, y_test)

print('Mean squared error on test data: ', mse_nn)

print('Mean absolute error on test data: ', mae_nn)

This displays the model evaluation summary:

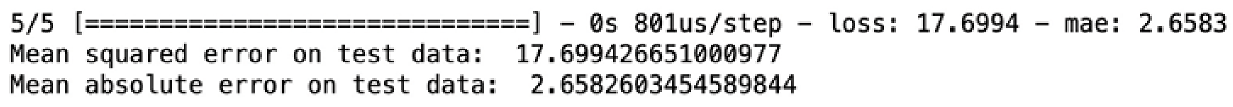

```
5/5 [==============================] - 0s 801us/step - loss: 17.6994 - mae: 2.6583
Mean squared error on test data:  17.699426651000977
Mean absolute error on test data:  2.6582603454589844
```

Figure 3.37: Model evaluation summary

Conclusion

In this chapter, we delved deep into the world of Artificial Neural Networks (ANN), breaking down its essential components and understanding its foundational mechanisms. Through practical application, we demonstrated how ANNs can be used to construct both classification and regression models. By comparing and contrasting the two types of models, readers should now have a clearer grasp of the versatility of neural networks in handling various types of data problems. Whether you're predicting categories or continuous outcomes, neural networks offer a powerful toolset for data-driven insights. As we move forward, harnessing the knowledge gained here will be invaluable in our exploration of even more advanced machine learning topics.

In the next chapter, we will look at Convolutional Neural Networks and their building blocks. We will conceptually and programmatically understand how to build a CNN model from scratch using Python.

Questions

What does ANN stand for?

Augmented Neural Networks

Auto Neural Networks

Artificial Neural Networks

None of the above

What are the different types of neural networks?

Artificial Neural Networks (ANN)

Convolutional Neural Networks (CNN)

Recurrent Neural Networks (RNN)

All of the above

Modeling dataset is usually divided into?

Train and Test set

Train and Validation set

Train, Validation and Test set

None of the above

What are different types of activation functions?

ReLU

Sigmoid

Softmax

Tanh

All of the above

What are different types of optimizers in deep learning?

adam

adagrad

rmsprop

All of the above

Answers

(c)

(d)

(c)

(e)

(d)

CHAPTER 4

Convolutional Neural Networks

One picture is worth a thousand words

— Albert Einstein

We give less credit to the intelligence of the human brain than it deserves, simply because this intelligence is just given to us. We are born with it, and it was not explicitly coded. But now, when we try to mimic the intelligence of the human body through artificial neural networks, we realize how easily our brain does certain things that are quite difficult to mimic. For instance, looking at a cat or a dog and identifying them is so natural for us. But if you think about it, once the light rays fall upon these objects, they reflect and are absorbed by our eyes. From the eyes, this information is transferred to the brain to process and conclude that what we are seeing is a cat or a dog. And all this happens naturally and in a fraction of milliseconds. A whole lot of code must be written to mimic this intelligence. The code breaks down an image into data that can be fed into an Artificial Neural Network to further process and classify the respective class of the object. The process of converting the details in an image into manageable data is termed “convolution.” The entire network that applies these convolutions to an image and then classifies it using artificial neural networks is known as a Convolutional Neural Network

Structure

In this chapter, we will cover the following topics:

Convolutional neural networks components

Image classification using CNN

Hyperparameters tuning using automated tools

Objectives

After completing this chapter, you should be aware of the various components of convolutional neural networks. You should be familiar with the technology behind digital image processing. You should also be able to solve a problem of image classification by building your own convolutional neural network using the python keras library. Additionally, you understand how to tune the hyperparameters of the deep learning model using an automated hyperparameter tuner, such as KerasTuner.

Convolutional neural networks components

A convolutional neural network comprises various underlying components. Some components are used to build the basic network, and others are used to optimize the network.

Load required libraries

Let us start with loading all the required libraries:

```
import keras

from keras.datasets import mnist

import numpy as np

import cv2

import matplotlib.pyplot as plt

%matplotlib inline
```

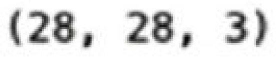
Digital image as a numpy array

Read the image from the disk:

```
image=cv2.imread("mnist_zero.png")
```

```
image.shape
```

The image is read as a RGB image with 3 channels. Hence, the image is of size (28,28) and 3 channels:



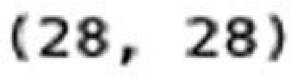
```
(28, 28, 3)
```

Figure 4.1: image array size

First, let us convert the image from a 3 channel (RGB) image to a 1 channel (grayscale) image:

```
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

```
gray_image.shape
```



```
(28, 28)
```

Figure 4.2: Image array size

To display the image as a full matrix that can display the width of the image in 1 line, we will reduce the image size from (28,28) to (18,18):

```
gray_image = cv2.resize(gray_image, (18,18))
```

```
gray_image.shape
```

```
(18, 18)
```

Figure 4.3: Image array size

Print the image object as an image:

```
plt.imshow(gray_image, cmap='gray')
```

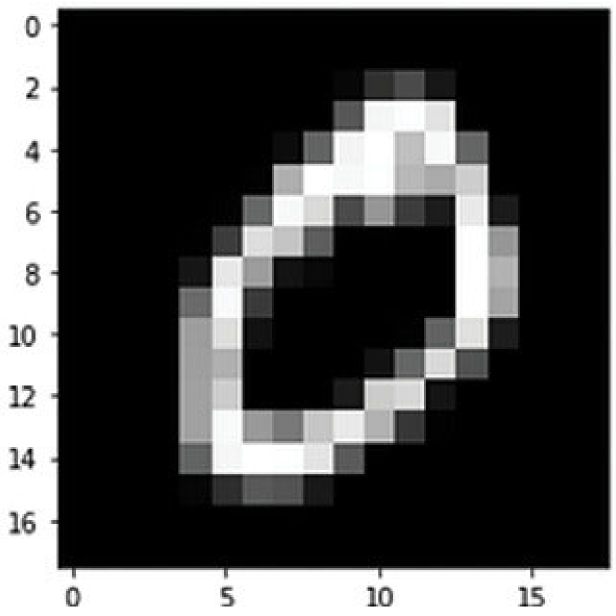
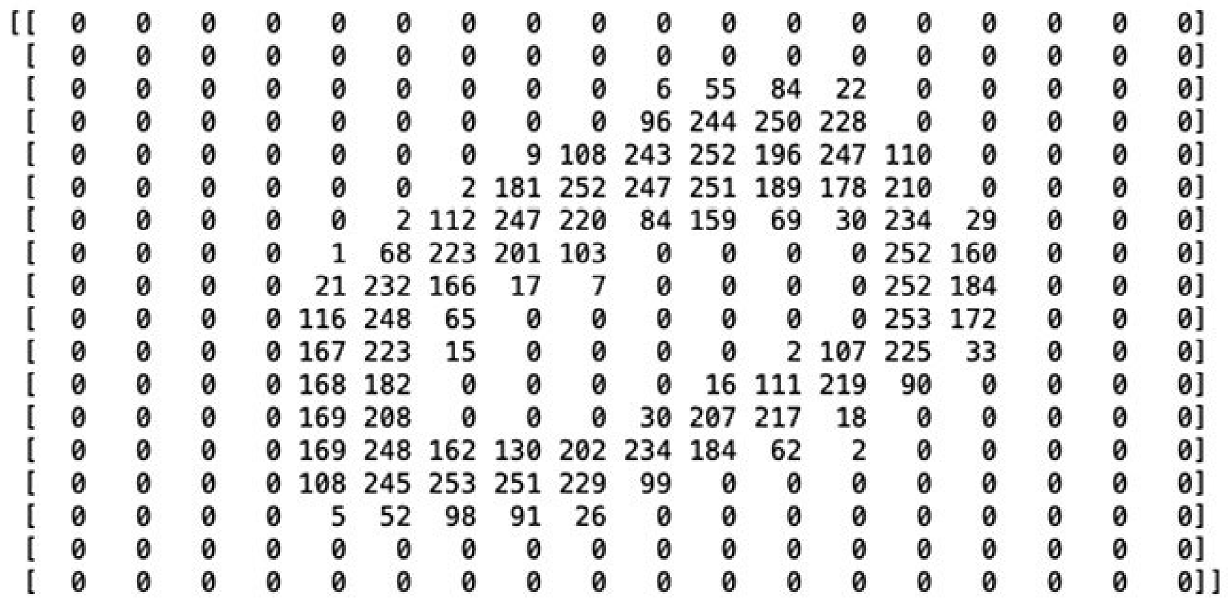


Figure 4.4: Image display

Now, display the image object as a numpy array:

print(gray_image)



```
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   6  55  84  22   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0  96 244 250 228   0   0   0   0   0]
 [  0   0   0   0   0   0   0   9 108 243 252 196 247 110   0   0   0   0]
 [  0   0   0   0   0   0   2 181 252 247 251 189 178 210   0   0   0   0]
 [  0   0   0   0   0   2 112 247 220  84 159  69  30 234  29   0   0   0]
 [  0   0   0   0   1  68 223 201 103   0   0   0   0 252 160   0   0   0]
 [  0   0   0   0  21 232 166  17   7   0   0   0   0 252 184   0   0   0]
 [  0   0   0   0 116 248  65   0   0   0   0   0   0 253 172   0   0   0]
 [  0   0   0   0 167 223  15   0   0   0   0   2 107 225  33   0   0   0]
 [  0   0   0   0 168 182   0   0   0   0  16 111 219  90   0   0   0   0]
 [  0   0   0   0 169 208   0   0   0  30 207 217  18   0   0   0   0   0]
 [  0   0   0   0 169 248 162 130 202 234 184  62   2   0   0   0   0   0]
 [  0   0   0   0 108 245 253 251 229  99   0   0   0   0   0   0   0   0]
 [  0   0   0   0   5  52  98  91  26   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]]
```

Figure 4.5: Pixel display

You can see that each pixel of the image represents the intensity of white in that pixel. Hence, the pixel that is black in color is represented as 0. The more the number is nearer to 255, the higher the intensity of white in that pixel.

Kernels/filters and convolution process

As we know, an image is nothing but a numpy array, and the dimension of the array is its pixel arrangement in width and height. Additionally, if it is a color image, the same dimension of array will be present for all the 3 channels: Red, Green, and Blue.

Now, we can convert this array into a one-dimensional vector and ingest this vector into a neural network for training a classification model.

But the problem with that is the model will be trained based on the pixel values at a given cell in the array. Hence, the model may fit well for the training data, but as soon as the pixel value changes in the testing data, the model will fail miserably to identify the class of the image in the test data.

For instance, if the model is trained only on the raw pixel values and then if model is trained on the image in Figure which is number 4, the model will not be able to identify number 4 in the image in Figure just because the pixel values are differently placed in both the images, even though they both represent the same number:

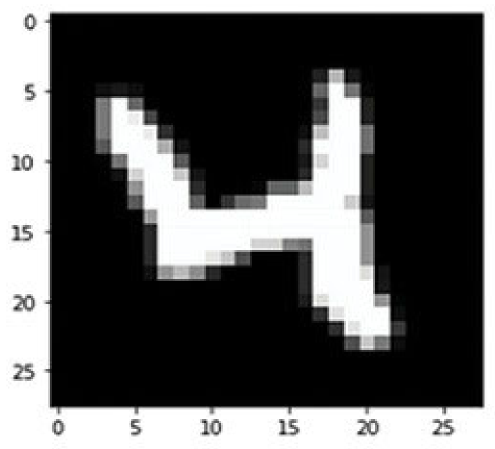


Figure 4.6: Training image

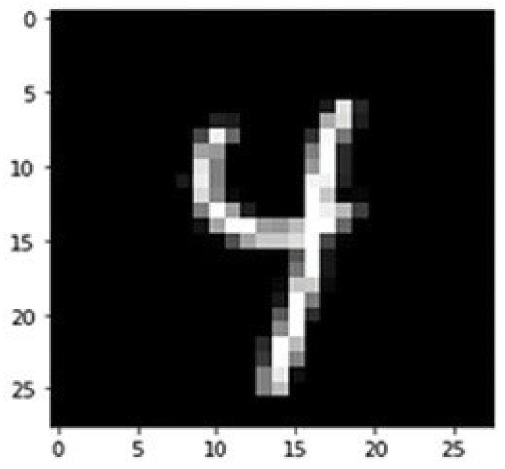


Figure 4.7: Testing image

Hence, we cannot rely on just the raw pixel values of an image to build a predictive model. This is where the concept of convolutions comes into the picture.

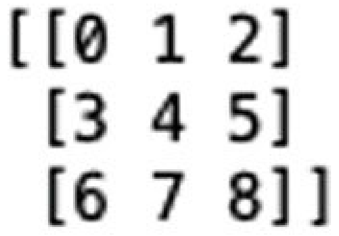
First, let us look at the process of convolution, mathematically.

Let us import the numpy library:

```
import numpy as np
```

Create a matrix

```
matrix_a=np.arange(0,9).reshape(3,3)

print(matrix_a)
```

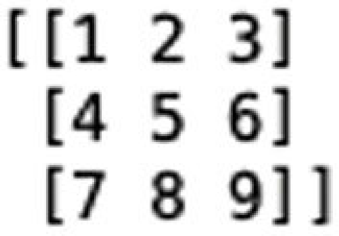


```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

Figure 4.8: matrix_a

Create another matrix

```
matrix_b=np.arange(1,10).reshape(3,3)

print(matrix_b)
```



```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Figure 4.9: matrix_b

The element-wise multiplication of both matrices means the number at a particular index in matrix_a should be multiplied by the number at the same index of

For instance,

$$matrix\_a[0][0] = 0$$

$$matrix\_b[0][0] = 1$$

Hence, element-wise matrix multiplication will give the resultant matrix value at the same index, as shown here:

$$matrix\_c[0][0] = matrix\_a[0][0] \ast matrix\_b[0][0]$$

$$matrix\_c[0][0] = 0 \ast 1 = 0$$

Similarly:

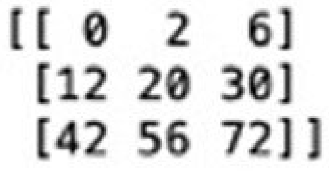
$$matrix\_c[0][1] = matrix\_a[0][1] \ast matrix\_b[0][1]$$

$$matrix\_c[0][1] = 1 \ast 2 = 2$$

and so on:

matrix_c=matrix_a*matrix_b

print(matrix_c)



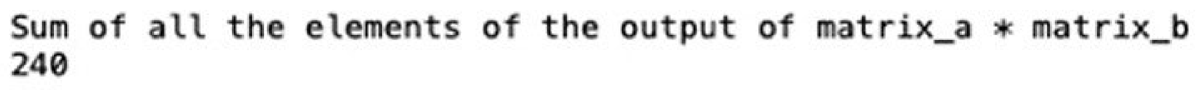
```
[[ 0  2  6]
 [12 20 30]
 [42 56 72]]
```

Figure 4.10: matrix_c

Now, as per definition, a convolution is nothing but an element-wise matrix multiplication of two matrices, and then the addition of all the elements of the output matrix:

print("Sum of all the elements of the output of matrix_a * matrix_b")

print((matrix_a * matrix_b).sum())



```
Sum of all the elements of the output of matrix_a * matrix_b
240
```

Figure 4.11: Matrix multiplication

Hence, the convolution value of matrix_a and matrix_b is 240.

The process of convolution is to extract features from the images using kernels or filters, which can also be called a convolution matrix.

Now, like an image is a matrix of numbers, a kernel is also a matrix of numbers, just that an image is a bigger matrix and a kernel is a smaller matrix.

Then how do kernels convolve an image, since to convolve, both the matrices should have the same dimensions to perform the element-wise multiplication. For that, the kernels convolve one smaller chunk of the image matrix at a time. The smaller chunk of the image matrix will be of the same dimension as of the kernel matrix.

In this process, the kernel will slide from one chunk to another chunk of the image until the whole image is convolved.

Hence, the kernel slides over the image from left to right and top to bottom for the convolution process, as shown here:

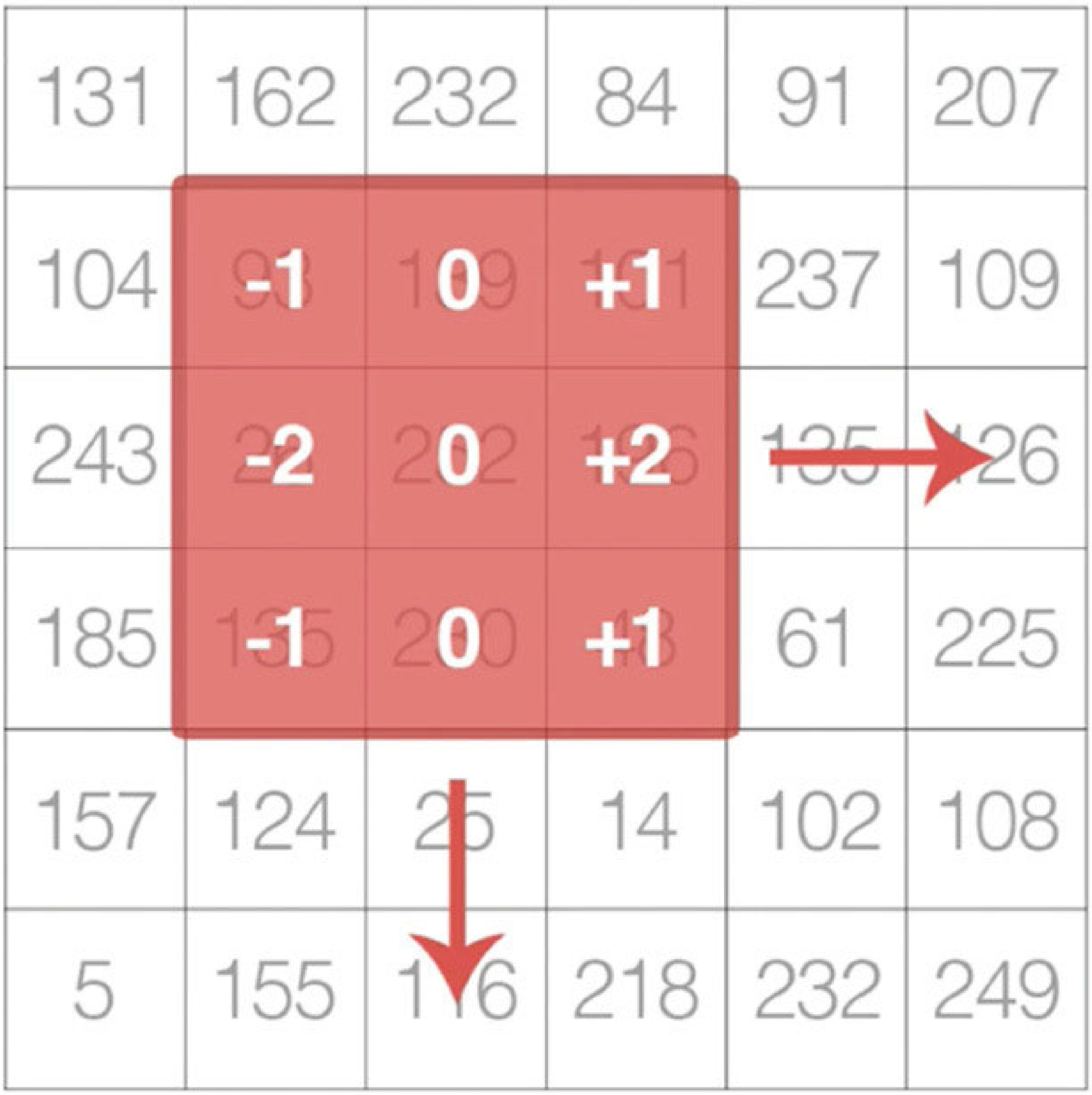


Figure 4.12: Kernel display

Every time the kernel convolves a chunk of the image, the convolution value is inscribed on the resultant matrix, as shown here:

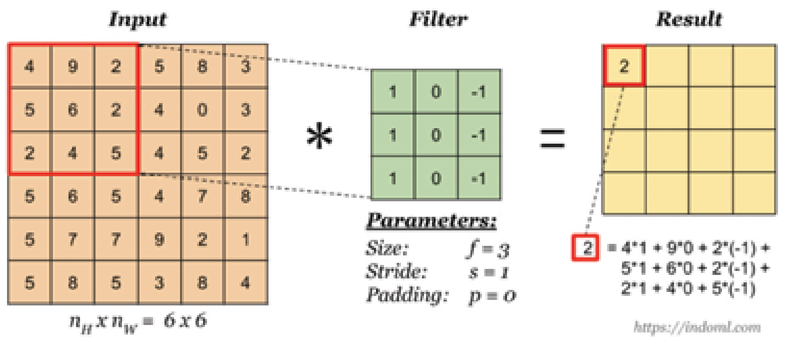


Figure 4.13: Convolution process

Stride

The number of steps the kernel will move while sliding over the image is based on the stride value.

If the stride value is 1, the kernel will move 1 element at a time while sliding to the right and also while sliding to the bottom, as shown here:

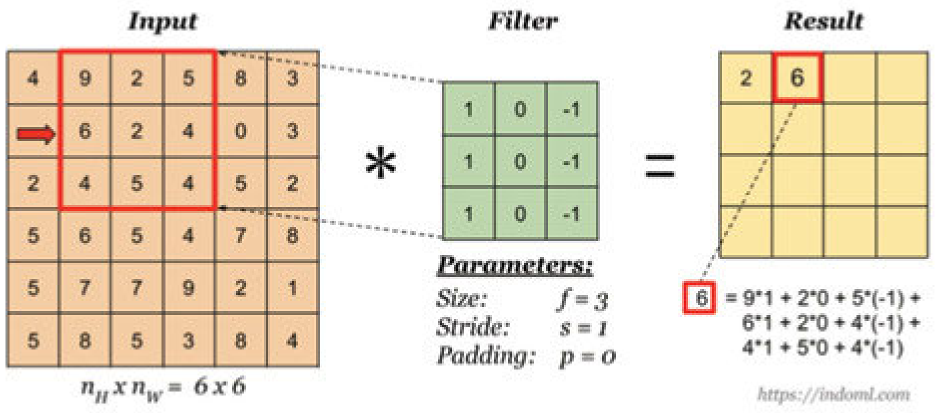


Figure 4.14: Stride process

If the stride value is 2, the kernel will move 2 elements at a time while sliding to the right and also while sliding to the bottom, as shown here:

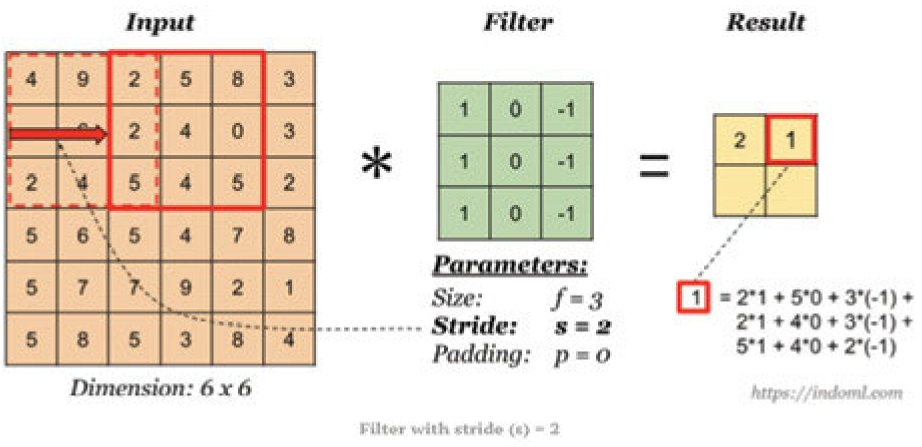


Figure 4.15: Stride process for two-step slide

Padding

As we can see, the dimension of the resultant matrix is always smaller than the dimension of the original image. That means some information is lost in the process of convolution. That also means that the deeper the network, the smaller the output matrix will become, which may result in significant loss of information in deeper networks.

Additionally, sometimes it becomes mathematically infeasible to convolve a certain dimension of image using a certain dimension of filter or kernel.

For example, if the image is of the dimension (5,5), the kernel is of the dimension (3,3), and the stride length is 2, the kernel will not be able to slide over the image after first convolution, as there will not be enough pixels left in the remaining steps.

Here comes the need for padding in the convolution process.

Padding is simply adding an additional layer around the original image. The value of the elements in the padding layer can be anything; the most commonly used values in the elements of the padded layer is either 0 or the pixel value of the adjacent cell, as shown in the following screenshot:

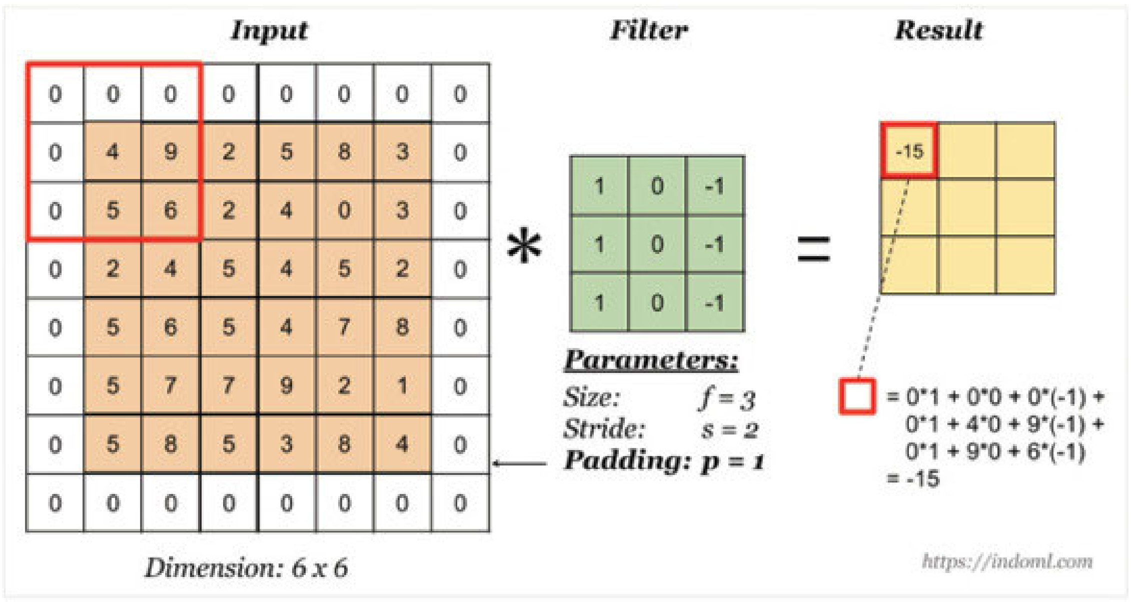


Figure 4.16: Padding

Now, the dimension of the output matrix is determined by the following formula:

$$Output\ size = \frac{((input\ size) + 2 * padding - (kernel\ size)) + 1}{stride}$$

Suppose input size = 5, padding = 1, kernel size = 3 and stride = 2.

Then

$$output\ size = \frac{5 + 2*1 - 3 + 1}{2} = 3$$

Hence, the resultant matrix will be of the dimension (3,3).

Convolution on RGB image

For a color (RGB) image, there are 3 image matrices, one for each channel. Hence, to convolve three image matrices, 3 kernel matrices are required. Hence, as per the definition, the number of channels of the filter is always equal to the number of channels in the image.

Mathematically, the kernel associated with Red channel will convolve the matrix of Red channel of the image, the kernel associated with Green channel will convolve the matrix of Green channel of the image, and the kernel associated with Blue channel will convolve the matrix of Blue channel of the image.

Hence, there will be 3 output matrices. The final output matrix of the convolution process will be the element-wise addition of the 3 output matrices, as shown in the following screenshot:

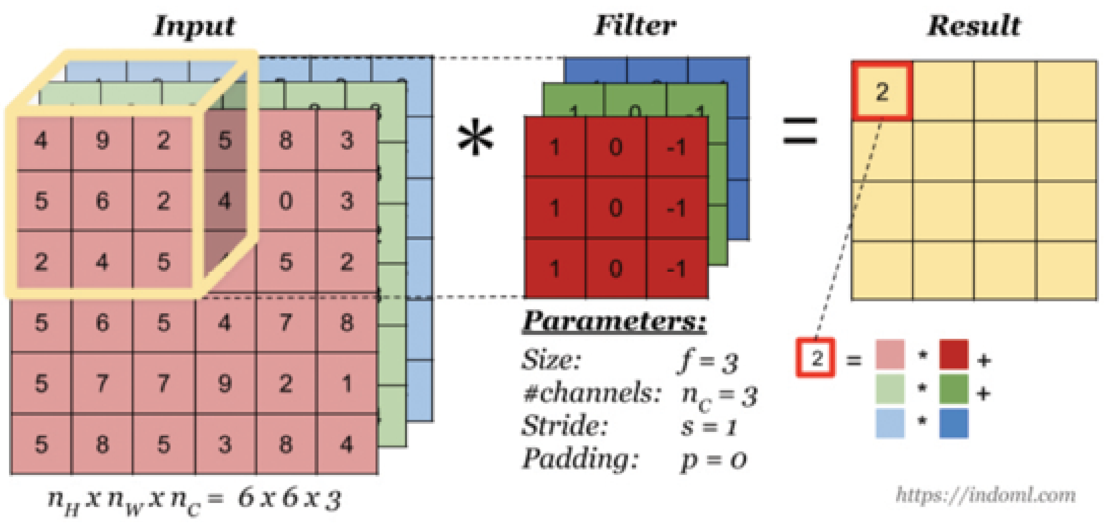


Figure 4.17: Convolution process

Hence, irrespective of whether the input image is a grayscale image (1 channel) or an RGB image (3 channels), the output will always be a single-dimensional matrix.

Convolution operation with multiple filters

The whole purpose of convolving an image is to extract the features from the image. And since we need to extract multiple features from the image, the same image can be convolved by multiple filters. As we know, each filter, irrespective of the number of channels, will produce an output matrix of one dimension. Hence, the number of channels in the final output matrix depends on the number of filters used to convolve an image, as shown in the following screenshot:

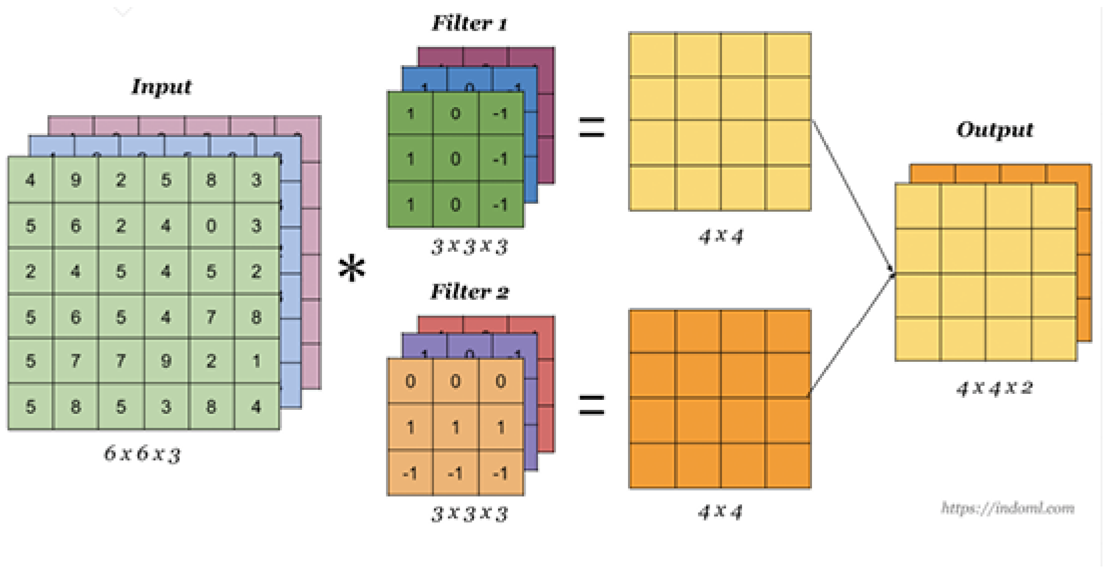


Figure 4.18: Multiple Filters

One convolution layer

To stitch the entire process of a single convolution layer in a CNN, each filter produces one output matrix from an image. Hence, the number of output matrices from one convolution layer depends on the number of filters or kernels applied.

Once the n-layered output matrix is derived, the activation function is applied on each and every element of the output matrix in all the layers.

Further, the bias term is added on each and every element of the output matrix.

Finally, after activation function is implemented and bias term is added on every element of the n-layered output matrix, the final n-dimensional output is generated from a single convolutional layer of an n-layered CNN, as shown in the following screenshot:

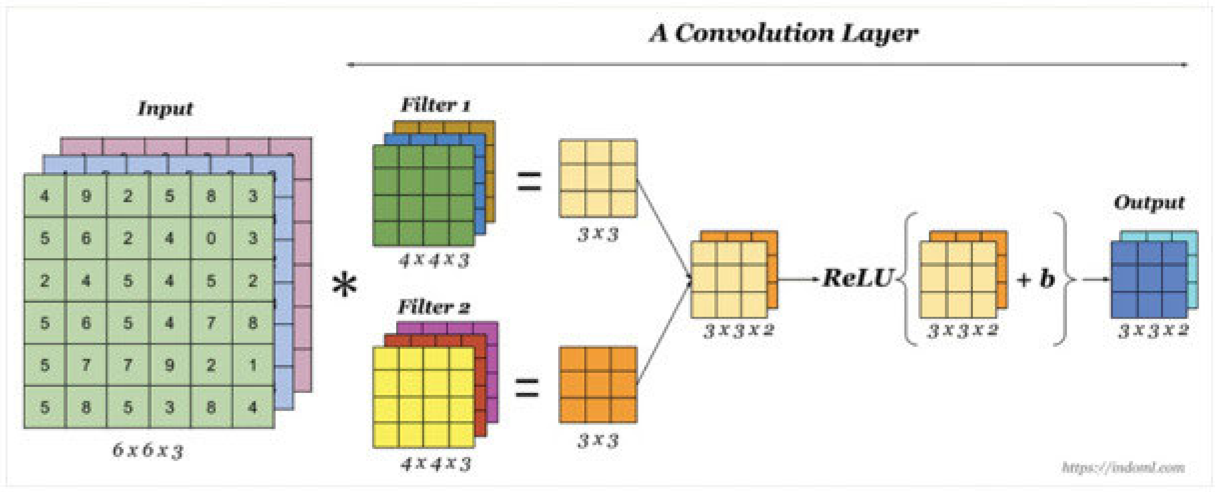


Figure 4.19: One convolution layer

Further, the final output matrix from this layer will be the input matrix for the next layer, and the same process continues.

Pooling

In order to make the model more generalised, the pooling operation is performed in subsequent layers.

The pooling process reduces the dimensions of the input matrix, which depends on the pool size and the stride with which the pooling window slides.

If max pooling is applied, then the maximum number in the pooling window of the size (2,2) will be the result of the output matrix. And if average pooling is applied, then the average of the numbers in those 2*2 dimensional matrix will be the result of the output matrix, as shown in the following screenshot:

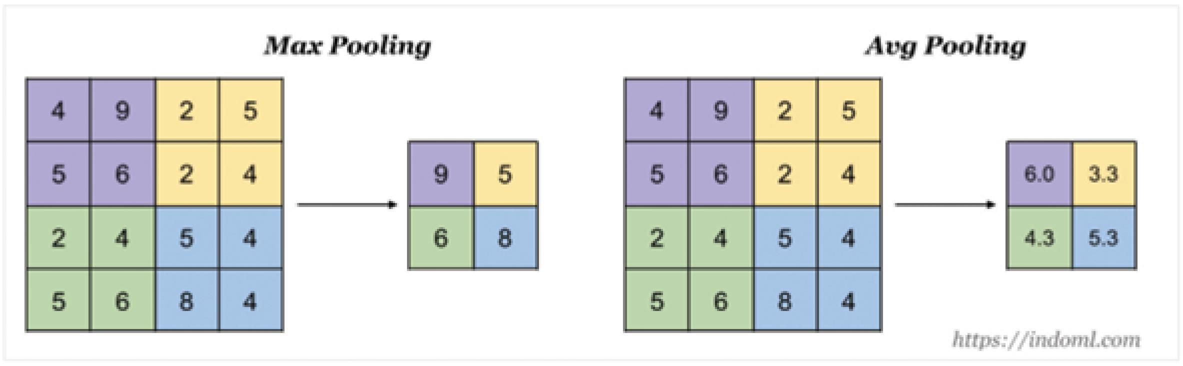


Figure 4.20: Pooling

The pooling layer reduces the size of the image representations, which speeds up calculations. In addition, the pooling process makes some of the features it detects a bit more robust.

For an RGB image, the pooling concept remains the same, just that it is applied on all the channels of the image and produces an equal number of channels in the output, as shown in the following screenshot:

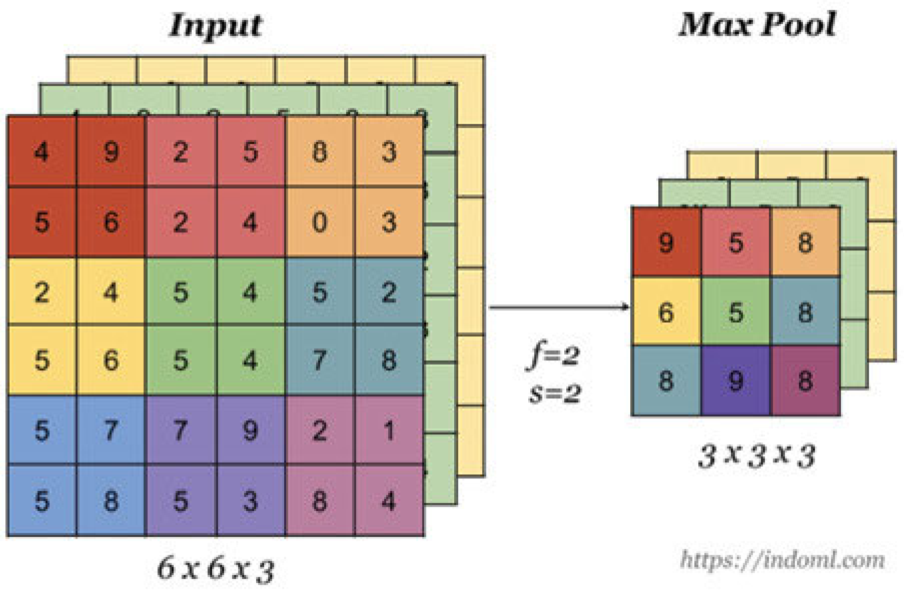


Figure 4.21: Max Pool

Flattening

The flattening process refers to converting the n-dimensional matrix into a 1-d vector, as shown in the following screenshot:

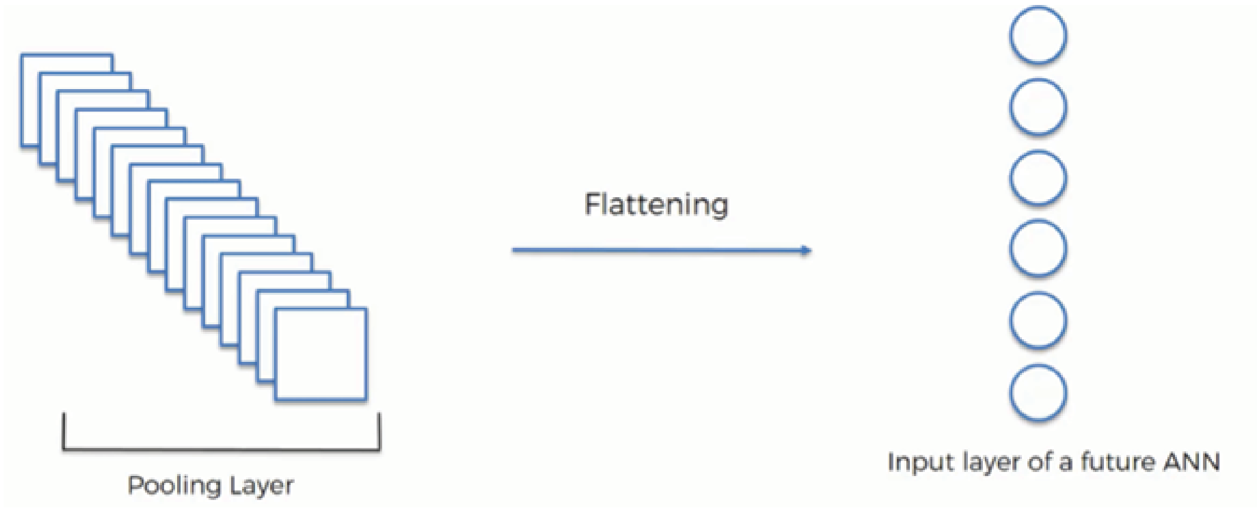


Figure 4.22: Flattening

The whole purpose of convolving an image is to extract features from the image. The initial layers of the network extract more low-level features like edges, while the layers toward the end extract more high-level features of the image.

Dense layers

After the extracted features from the image are flattened and converted into a 1-d vector, it becomes feasible to ingest this vector into an artificial neural network, which is also known as dense layer, since all the neurons in a layer are connected to all the neurons to the subsequent layers, as shown in the following screenshot:

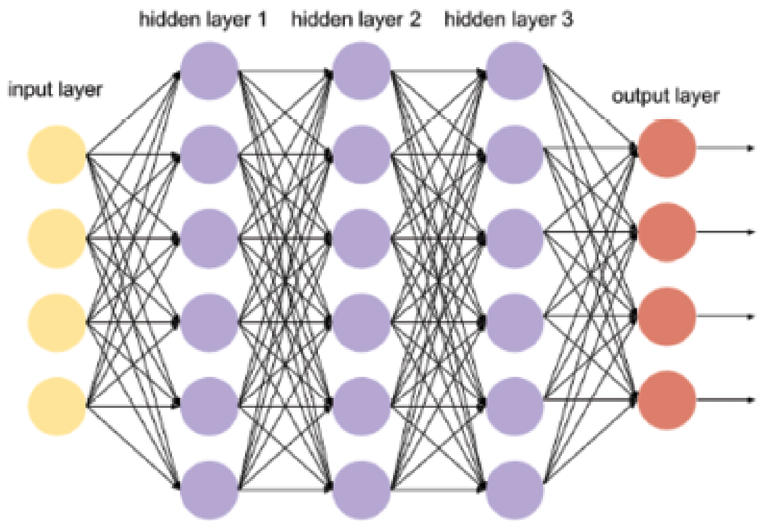


Figure 4.23: Dense layers

It is the dense layer that is responsible to finally perform the task of classification of the images by training the hidden layers of the network and coming up with the optimum values of the weights and biases for each neuron in the neural network.

Now, we could have simply flattened the pixel values in the matrix derived by the input image and fed the pixel values into this dense layer for the model training for the classification task. But, as stated earlier, with even a single pixel value change in the input image, the model would start giving incorrect results. In simple words, the model will not be robust.

Hence, using convolution layers, first the features from the image are extracted using various convolutional layers, and then those features are flattened and ingested into dense layers for classification training, as shown in the following screenshot:

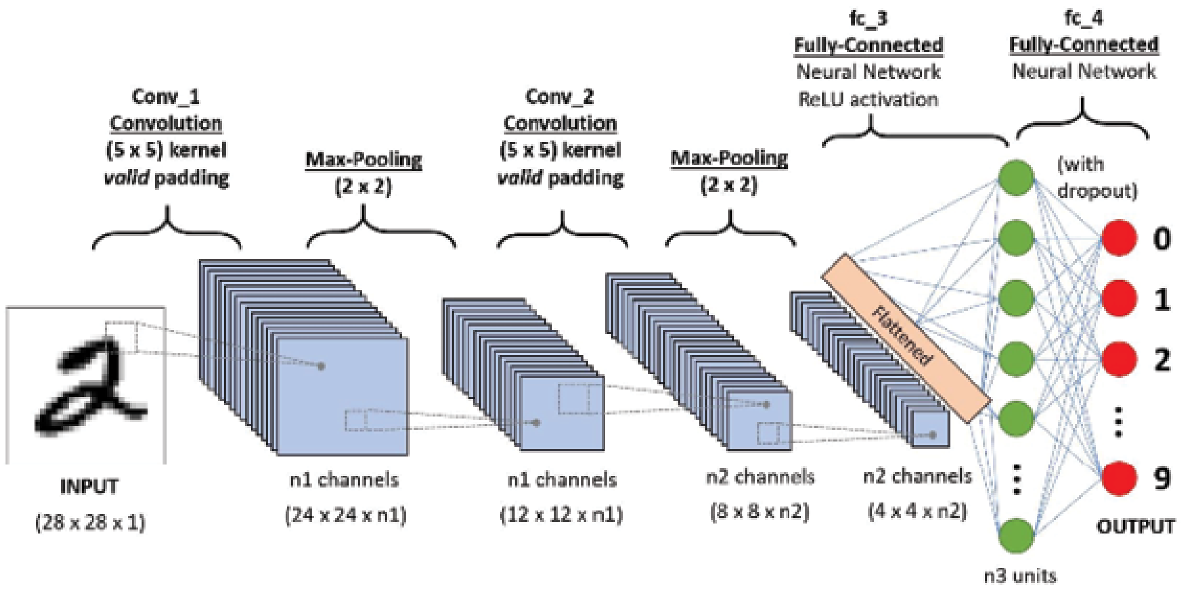


Figure 4.24: End-to-end CNN architecture

Image classification using CNN

Classification of images is the most important use case that can be solved using CNN. In this section, we will look at the CNN implementation for a classification problem.

Problem statement

The problem is about classifying the numbers between 0 to 1 in a set of images containing handwritten numbers.

Dataset - MNIST

MNIST dataset is considered to be the “Hello World” program of Data Science. Any tutorial on Computer vision and image classification starts with the MNIST dataset implementation using convolutional neural network.

The MNIST dataset is embedded into the keras.dataset library and can be loaded directly into memory, without the need to load images from the disk.

Implementation

Let us look at how to implement CNN in Python.

Load Python libraries

Let us first import all the required Python libraries:

import tensorflow as tf

from keras.datasets import mnist

import pandas as pd

import matplotlib.pyplot as plt

import random

import numpy as np

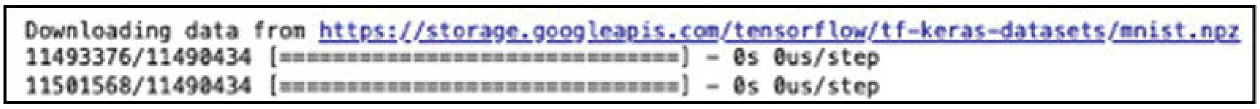
from keras.models import Sequential

from keras.layers import Conv2D, MaxPooling2D, Dense, Flatten, Dropout, BatchNormalization, Activation

Load data

Load the MNIST data into train and test tuples directly from the keras.dataset library:

(trainX, trainY), (testX, testY) = mnist.load_data()



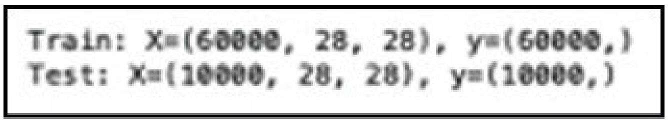
```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
```

Figure 4.25: Data download status

Let us look at the shape of both train and test datasets:

print('Train: X=%s, y=%s' % (trainX.shape, trainY.shape))

print('Test: X=%s, y=%s' % (testX.shape, testY.shape))



```
Train: X=(60000, 28, 28), y=(60000,)
Test: X=(10000, 28, 28), y=(10000,)
```

Figure 4.26: Test and train data shapes

Hence, the training set contains 60,000 images and the testing set contains 10,000 images.

Using the matplotlib library, we can plot the first few images of the dataset shown here:

```
for i in range(9):

# define subplot

 plt.subplot(330 + 1 + i)

# plot raw pixel data

 plt.imshow(trainX[i], cmap=plt.get_cmap('gray'))

# show the figure

plt.show()
```

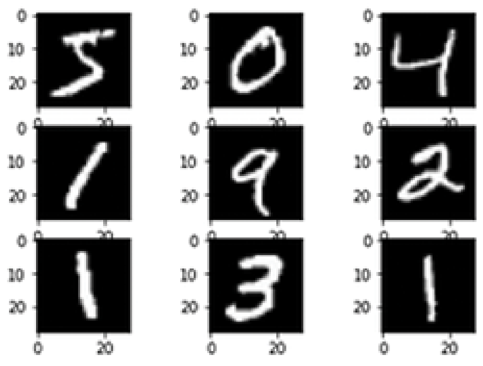


Figure 4.27: MNIST numbers random display

Data pre-processing

Reshape the dataset to have a single channel:

trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))

testX = testX.reshape((testX.shape[0], 28, 28, 1))

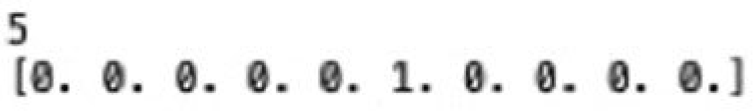
One-hot encode the target values:

trainY_cat = tf.keras.utils.to_categorical(trainY)

testY_cat = tf.keras.utils.to_categorical(testY)

print(trainY[0])

print(trainY_cat[0])



```
5
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

Figure 4.28: One-hot encoding

Normalization by scaling pixels:

```
# convert from integers to floats

trainX = trainX.astype('float32')

testX = testX.astype('float32')

# normalize to range 0-1

trainX = trainX / 255.0

testX = testX / 255.0
```

Modeling

We will use the Sequential module with the keras library to create the neural network. The model object is created for defining the CNN architecture:

```
model = Sequential()
```

Add the first convolutional layer in the model object; 32 kernels of the size (3,3) are used in this layer. The shape of the input data needs to be mentioned in the input_data argument:

```
model.add(Conv2D(32, (3, 3), input_shape = (28,28,1)))
```

Batch normalization is used as a regularization technique.

This is a technique used in deep neural networks to improve training speed and stability by normalizing the input to each layer to have zero mean and unit variance, and then scaling and shifting the normalized values using learnable parameters. This helps mitigate the problem of internal covariate shift, where the distribution of the inputs to a layer changes during training and can slow down convergence. By normalizing the inputs to each layer, batch normalization enables more stable and efficient training and can also regularize the model and reduce overfitting.

```
model.add(BatchNormalization())
```

The activation function used here is

```
model.add(Activation("relu"))
```

Further, add a max pooling layer with a pool size of (2,2):

```
model.add(MaxPooling2D(pool_size = (2, 2)))
```

Create further convolutional layers with similar configurations and different number of kernels:

```
model.add(Conv2D(64, (3, 3)))

model.add(BatchNormalization())

model.add(Activation("relu"))

model.add(MaxPooling2D(pool_size = (2, 2)))

model.add(Conv2D(128, (3, 3)))

model.add(BatchNormalization())

model.add(Activation("relu"))

model.add(MaxPooling2D(pool_size = (2, 2)))
```

Flatten the final feature matrix to create a 1-d vector that can be fed into the dense layers:

```
model.add(Flatten())
```

Create the first dense layer with 128 neurons and relu activation function:

```
model.add(Dense(128, activation = 'relu'))
```

Finally, create the output layer; the number of neurons in this layer will be the number of classes in the dataset, which is 10 in case of the MNIST dataset:

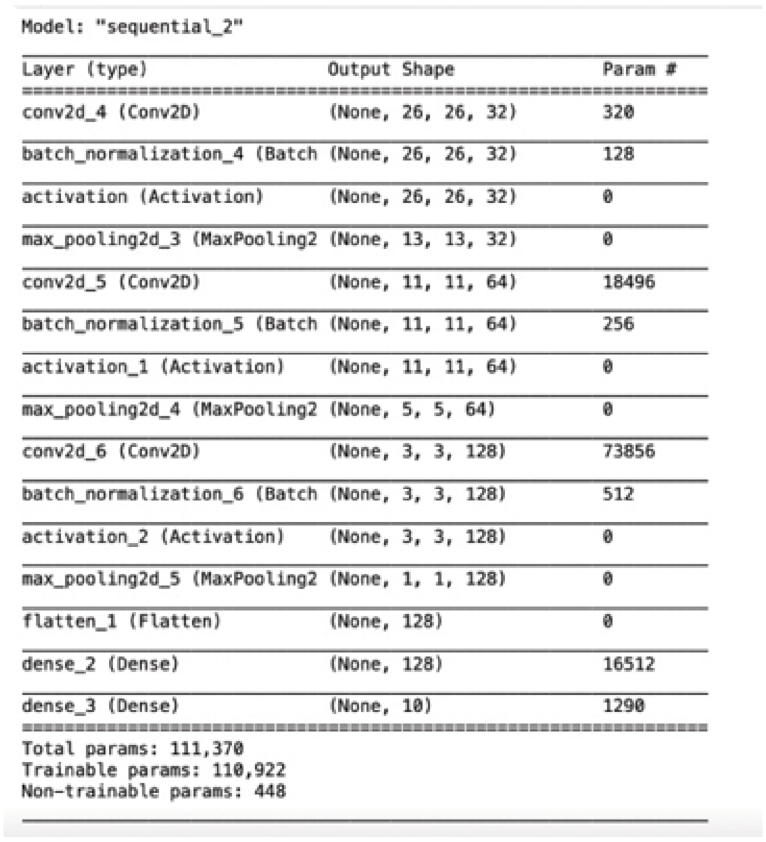
```
model.add(Dense(10, activation = 'softmax'))
```

Further, compile the model. The hyperparameters used are random based on experience:

```
model.compile(loss ='categorical_crossentropy', optimizer='adam', metrics = ['acc'])
```

Once the model is compiled, print the model summary to see the overall architecture of the convolutional neural network that we just designed:

```
model.summary()
```

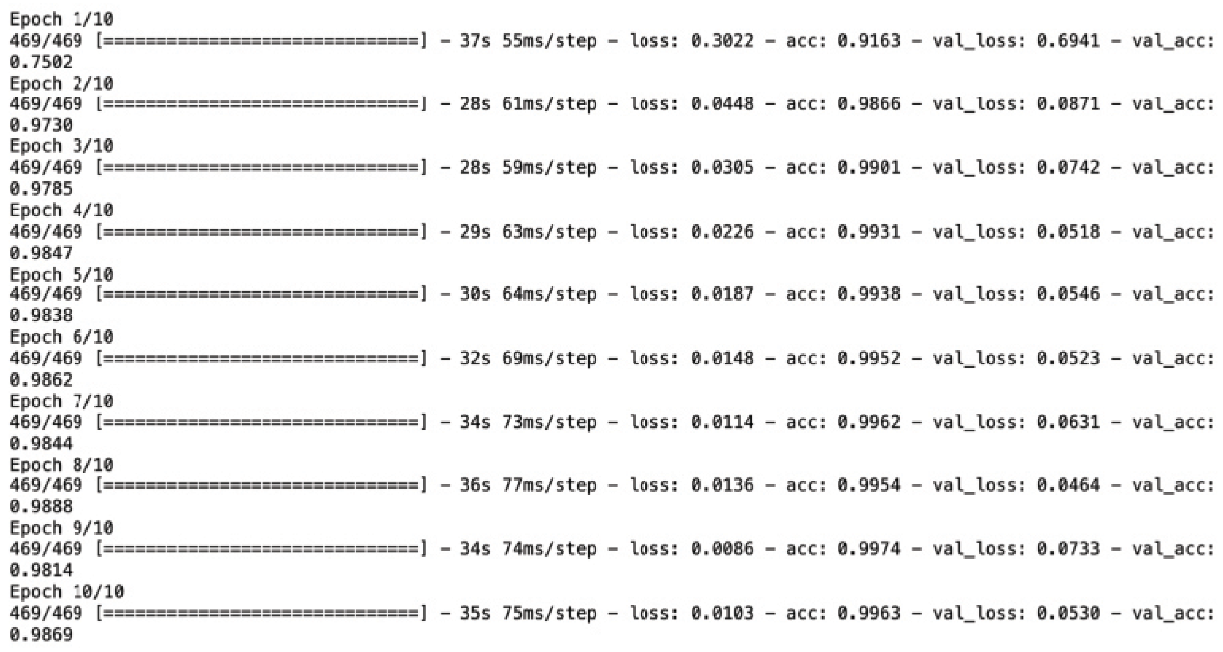


```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
batch_normalization_4 (Batch (None, 26, 26, 32)        128
_________________________________________________________________
activation (Activation)      (None, 26, 26, 32)        0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
batch_normalization_5 (Batch (None, 11, 11, 64)        256
_________________________________________________________________
activation_1 (Activation)    (None, 11, 11, 64)        0
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 3, 3, 128)         73856
_________________________________________________________________
batch_normalization_6 (Batch (None, 3, 3, 128)         512
_________________________________________________________________
activation_2 (Activation)    (None, 3, 3, 128)         0
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 1, 1, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1290
=================================================================
Total params: 111,370
Trainable params: 110,922
Non-trainable params: 448
_________________________________________________________________
```

Figure 4.29: Model summary

Train the model using the fit function. Validation data used during the training process is the test dataset:

```
history = model.fit(trainX, trainY_cat, batch_size = 128, epochs = 10, verbos
e = 1, validation_data = (testX, testY_cat))
```



```
Epoch 1/10
469/469 [==============================] - 37s 55ms/step - loss: 0.3022 - acc: 0.9163 - val_loss: 0.6941 - val_acc:
0.7502
Epoch 2/10
469/469 [==============================] - 28s 61ms/step - loss: 0.0448 - acc: 0.9866 - val_loss: 0.0871 - val_acc:
0.9730
Epoch 3/10
469/469 [==============================] - 28s 59ms/step - loss: 0.0305 - acc: 0.9901 - val_loss: 0.0742 - val_acc:
0.9785
Epoch 4/10
469/469 [==============================] - 29s 63ms/step - loss: 0.0226 - acc: 0.9931 - val_loss: 0.0518 - val_acc:
0.9847
Epoch 5/10
469/469 [==============================] - 30s 64ms/step - loss: 0.0187 - acc: 0.9938 - val_loss: 0.0546 - val_acc:
0.9838
Epoch 6/10
469/469 [==============================] - 32s 69ms/step - loss: 0.0148 - acc: 0.9952 - val_loss: 0.0523 - val_acc:
0.9862
Epoch 7/10
469/469 [==============================] - 34s 73ms/step - loss: 0.0114 - acc: 0.9962 - val_loss: 0.0631 - val_acc:
0.9844
Epoch 8/10
469/469 [==============================] - 36s 77ms/step - loss: 0.0136 - acc: 0.9954 - val_loss: 0.0464 - val_acc:
0.9888
Epoch 9/10
469/469 [==============================] - 34s 74ms/step - loss: 0.0086 - acc: 0.9974 - val_loss: 0.0733 - val_acc:
0.9814
Epoch 10/10
469/469 [==============================] - 35s 75ms/step - loss: 0.0103 - acc: 0.9963 - val_loss: 0.0530 - val_acc:
0.9869
```

Figure 4.30: Model run summary

It can be seen that the CNN we just designed is able to classify the test dataset with more than 98% accuracy with just 10 epochs of training.

Using the matplotlib library, we plot the training and validation accuracy and loss at each epoch, as shown here:

loss = history.history['loss']

val_loss = history.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'y', label='Training loss')

plt.plot(epochs, val_loss, 'r', label='Validation loss')

```
plt.title('Training and validation loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()
```

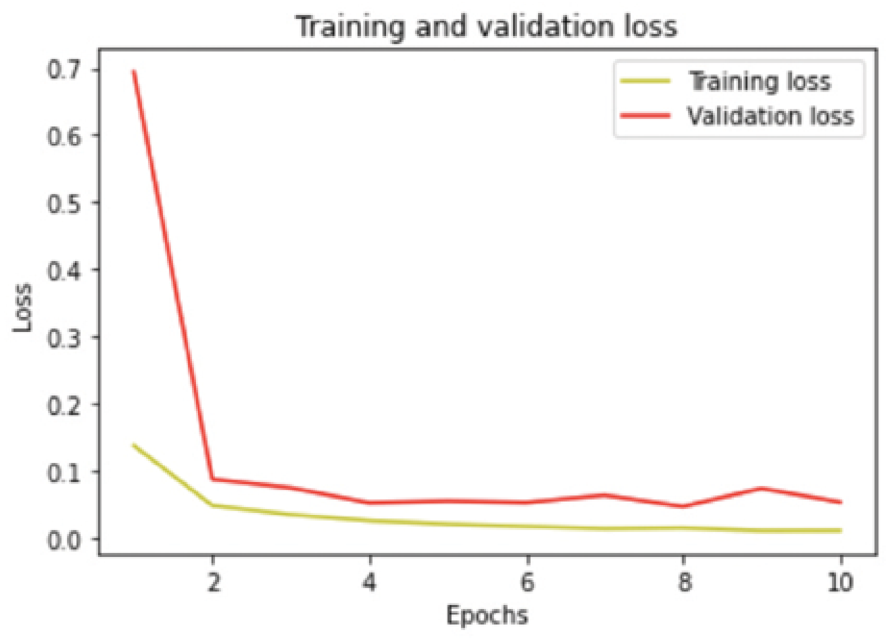


Figure 4.31: Training and validation loss

```
acc = history.history['acc']

val_acc = history.history['val_acc']

plt.plot(epochs, acc, 'y', label='Training acc')
plt.plot(epochs, val_acc, 'r', label='Validation acc')

plt.title('Training and validation accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.show()
```

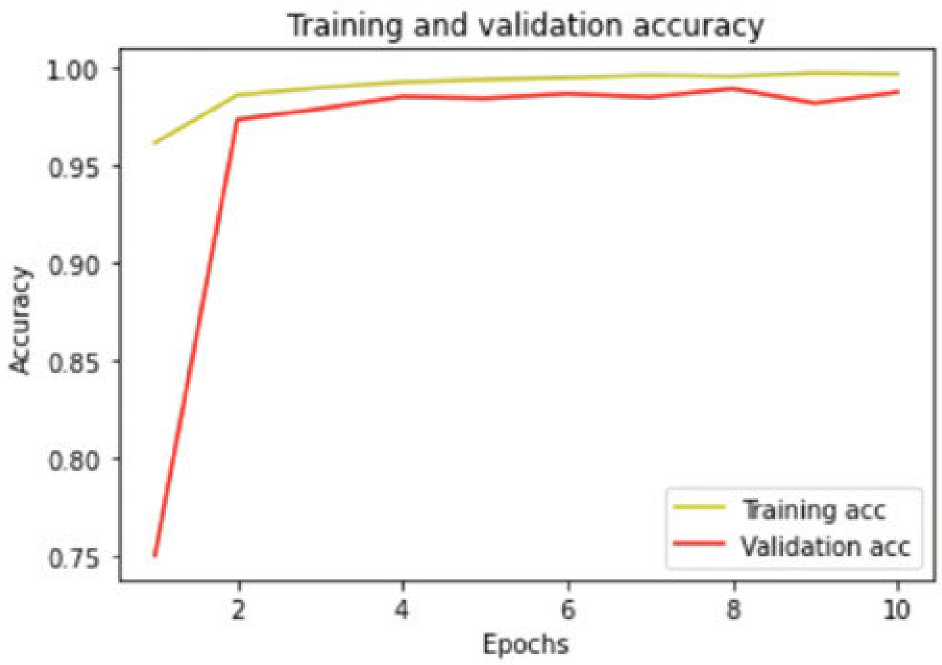
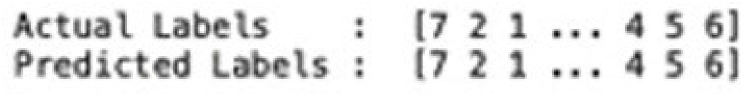


Figure 4.32: Training and validation accuracy

Use the predict_classes method to predict the classes of the test dataset to manually compare the predicted values with the ground truth:

```
prediction = model.predict(testX)

prediction = np.argmax(prediction,axis=1)

print("Actual Labels    : ", testY)

print("Predicted Labels : ", prediction)
```



```
Actual Labels    :  [7 2 1 ... 4 5 6]
Predicted Labels :  [7 2 1 ... 4 5 6]
```

Figure 4.33: Actual and predicted Labels

Plot a random image and its associated actual and predicted labels:

```
i = random.randint(1,len(prediction))

plt.imshow(testX[i,:,:,0])

print("Predicted Label: ", int(prediction[i]))

print("True Label: ", int(testY[i]))
```

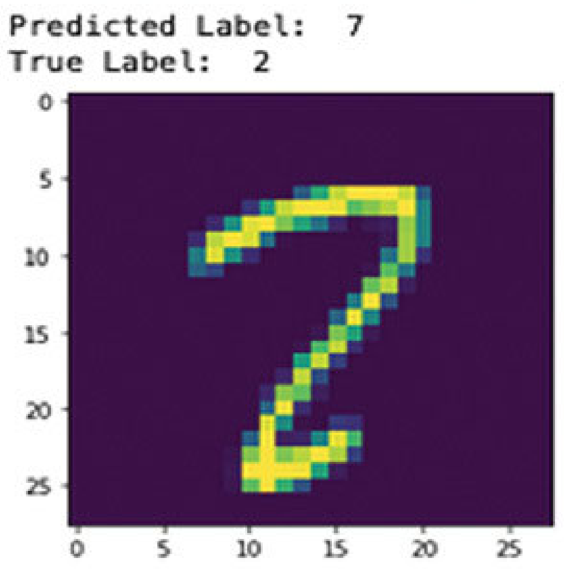

```
Predicted Label:   7
True Label:   2
```

Figure 4.34: Actual and predicted label for an image

As you can see, the model predicted a wrong label for this image. The handwritten number is 2, but it's predicted as 7. However, there is no denying that the number is written incorrectly and actually looks more like 7 instead of 2, even to human eyes.

Plot confusion matrix

To plot the confusion matrix, the sklearn library provides this generic function that can be copied from their official documentation at

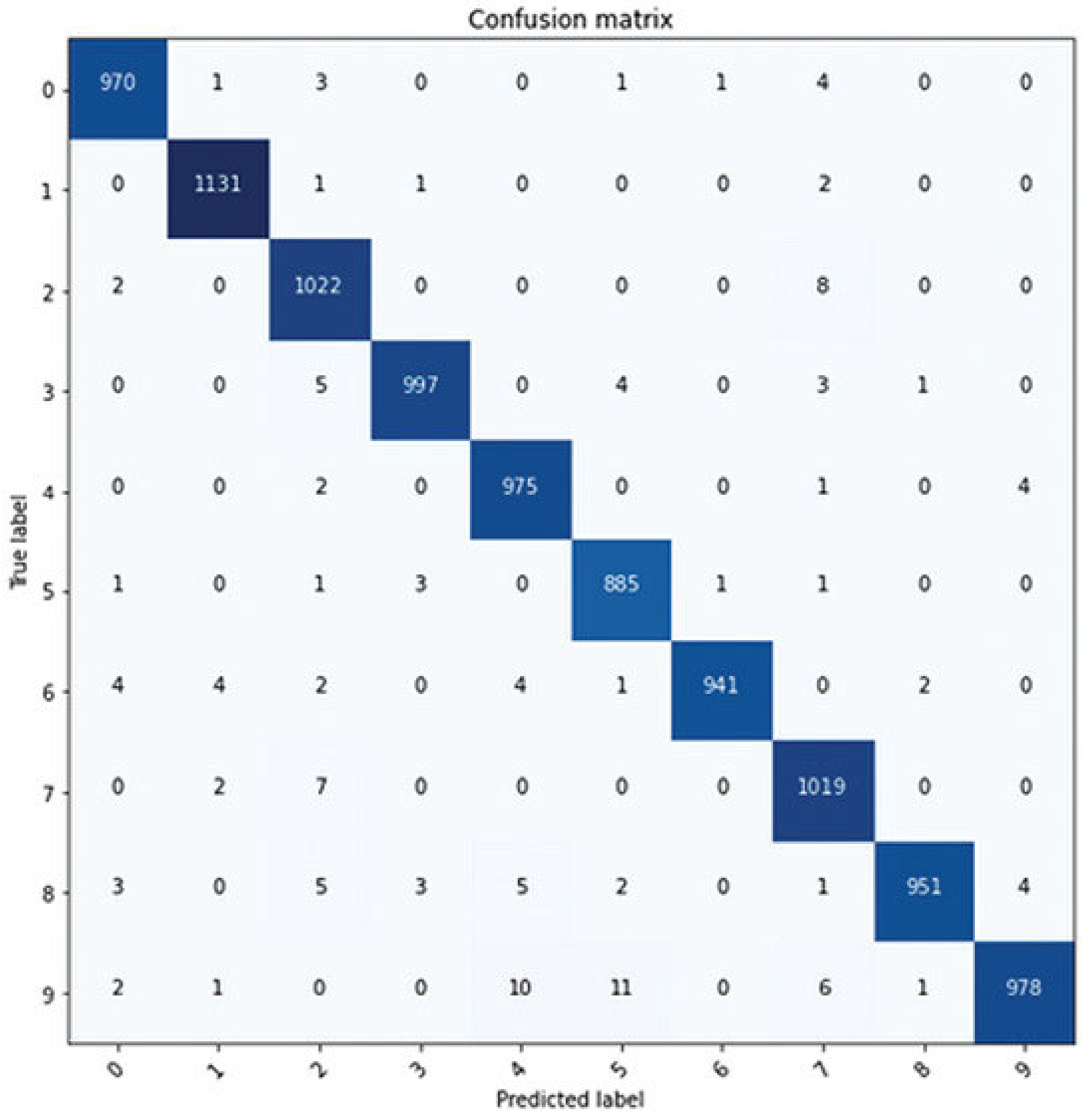
```
def plot_confusion_matrix(cm, classes,

                          normalize=False,

                          title='Confusion matrix',

                          cmap=plt.cm.Blues):

    """

    This function prints and plots the confusion matrix.

    Normalization can be applied by setting `normalize=True`.

    «»»

    import itertools

    if normalize:
```

```
    cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    print("Normalized confusion matrix")

else:

    print('Confusion matrix, without normalization')

plt.imshow(cm, interpolation='nearest', cmap=cmap)

plt.title(title)

tick_marks = np.arange(len(classes))


plt.xticks(tick_marks, classes, rotation=45)

plt.yticks(tick_marks, classes)

fmt = '.2f' if normalize else 'd'

thresh = cm.max() / 2.

for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):

    plt.text(j, i, format(cm[i, j], fmt),

             horizontalalignment="center",
color="white" if cm[i, j] > thresh else "black")

  plt.ylabel('True label')

  plt.xlabel('Predicted label')

  plt.tight_layout()

  plt.show()
```

Create a confusion matrix and plot

To create a confusion matrix, use the following code snippet:

```
from sklearn.metrics import confusion_matrix

test_label_list=list(np.unique(testY))

cnf_matrix = confusion_matrix(testY, prediction,labels=test_label_list)

np.set_printoptions(precision=2)

# Plot non-normalized confusion matrix

plt.figure(figsize=(8,8))

plot_confusion_matrix(cnf_matrix, classes=test_label_list,

                      title='Confusion matrix')
```


Confusion matrix

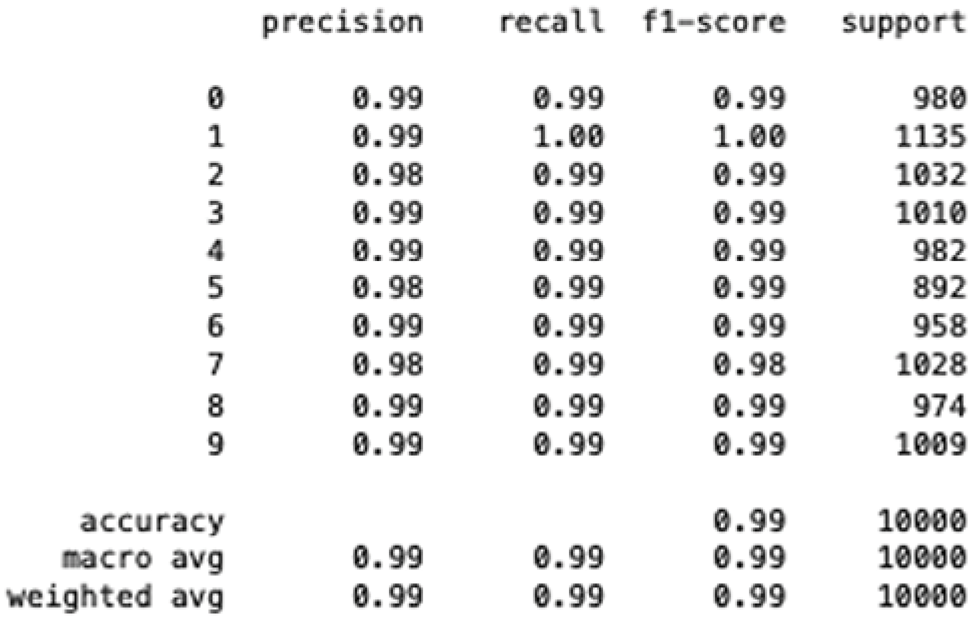
True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	970	1	3	0	0	1	1	4	0	0
1	0	1131	1	1	0	0	0	2	0	0
2	2	0	1022	0	0	0	0	8	0	0
3	0	0	5	997	0	4	0	3	1	0
4	0	0	2	0	975	0	0	1	0	4
5	1	0	1	3	0	885	1	1	0	0
6	4	4	2	0	4	1	941	0	2	0
7	0	2	7	0	0	0	0	1019	0	0
8	3	0	5	3	5	2	0	1	951	4
9	2	1	0	0	10	11	0	6	1	978

Figure 4.35: Confusion matrix

Confusion matrix gives a complete picture of the actual and predicted classes of all the images in the dataset. Additionally, let us plot the classification report to check the various other performance metrics like precision, recall, and f1-score:

```
from sklearn.metrics import classification_report, confusion_matrix
```

```
print(classification_report(testY, prediction))
```



```
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       980
           1       0.99      1.00      1.00      1135
           2       0.98      0.99      0.99      1032
           3       0.99      0.99      0.99      1010
           4       0.99      0.99      0.99       982
           5       0.98      0.99      0.99       892
           6       0.99      0.99      0.99       958
           7       0.98      0.99      0.98      1028
           8       0.99      0.99      0.99       974
           9       0.99      0.99      0.99      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000
```

Figure 4.36: Classification report

Hyperparameters tuning using KerasTuner

As we saw in the preceding tutorial where we used a MNIST dataset and trained a CNN model to classify the handwritten digits, we were able to achieve more than 99% accuracy with a very simple neural network. However, that is not the case with most real-world problems. The same network with the specific arrangement of convolutional layers clubbed with other layers like pooling layer, batch normalization layer, or even the selection of activation functions may not work for other problems at hand. The depth of the network plays a significant role in making a robust and generalized model.

Adding more hidden layers in the network may or may not improve the performance of the model. There are various other parameters that can be tuned to improve a model's performance. Such parameters are called hyperparameters.

Tuning the parameters of a neural network, like weights and biases, can be done by training the model. But tuning the hyperparameters like the number of hidden layers, learning rate, epochs, kernels, and so on is a matter of trial and experimentation. We can create as many permutations and combinations of these hyperparameters and build a neural network for each combination to come up with the most optimum model, but this is not feasible manually, as there can be hundreds or even thousands of such combinations.

Hence, KerasTuner comes to the rescue. KerasTuner is a Python library that automates the process of finding the best hyperparameters for a model.

The official website of the KerasTuner library is given here for further reading:

https://keras-team.github.io/keras-tuner/

For demonstrating the capabilities of KerasTuner, let us work on the Fashion MNIST datasets, which is very similar to MNIST datasets, but instead of handwritten numbers, the images belong to various wearable items.

Dataset – Fashion MNIST

Fashion MNIST dataset is very similar to MNIST dataset in terms of the number of classes and training and testing images available.

Fashion MNIST dataset are as follows:

60,000 training examples

10,000 testing examples

10 classes

28×28 grayscale/single channel images

The ten fashion class labels are as follows:

T-shirt/top

Trouser/pants

Pullover shirt

Dress

Coat

Sandal

Shirt

Sneaker

Bag

Ankle boot

Implementation

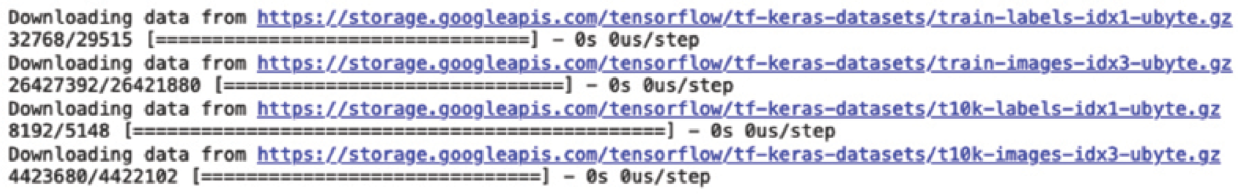
Since we have already implemented MNIST dataset in the previous tutorial, and fashion-mnist dataset can be implemented using the same set of code by just loading the fashion-mnist dataset from the keras dataset library instead of loading the mnist dataset. Hence, in this KerasTuner implementation, we will not follow the same approach to avoid redundancy.

Instead, we will first load the dataset and plot the images for the basic understanding of the dataset. Further, we will use KerasTuner to find the best model for this problem statement.

Loading the fashion-mnist dataset:

from tensorflow import keras

(X_train, y_train), (X_test, y_test) = keras.datasets.fashion_mnist.load_data()

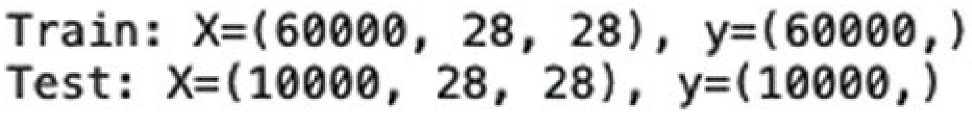


```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
32768/29515 [=================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26427392/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
8192/5148 [===============================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [==============================] - 0s 0us/step
```

Figure 4.37: Fashion-mnist dataset

Let us check the shape of the dataset.

print('Train: X=%s, y=%s' % (X_train.shape, y_train.shape)) print('Test: X=%s, y=%s' % (X_test.shape, y_test.shape))



```
Train: X=(60000, 28, 28), y=(60000,)
Test: X=(10000, 28, 28), y=(10000,)
```

Figure 4.38: Train and test data frame shapes

Let us plot the first few images of the dataset:

```
import matplotlib.pyplot as plt

for i in range(9):

 plt.subplot(330 + 1 + i)

 plt.imshow(X_train[i], cmap=plt.get_cmap('gray'))

plt.show()
```

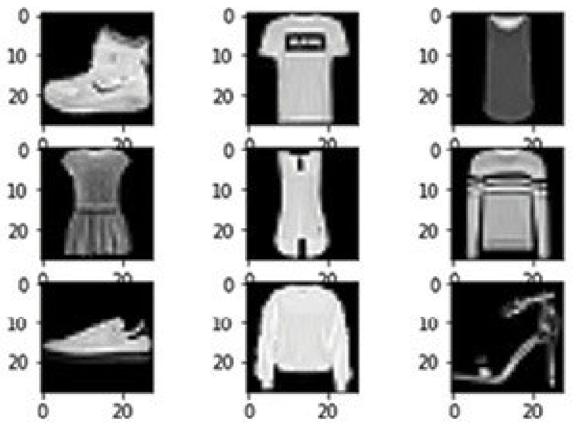


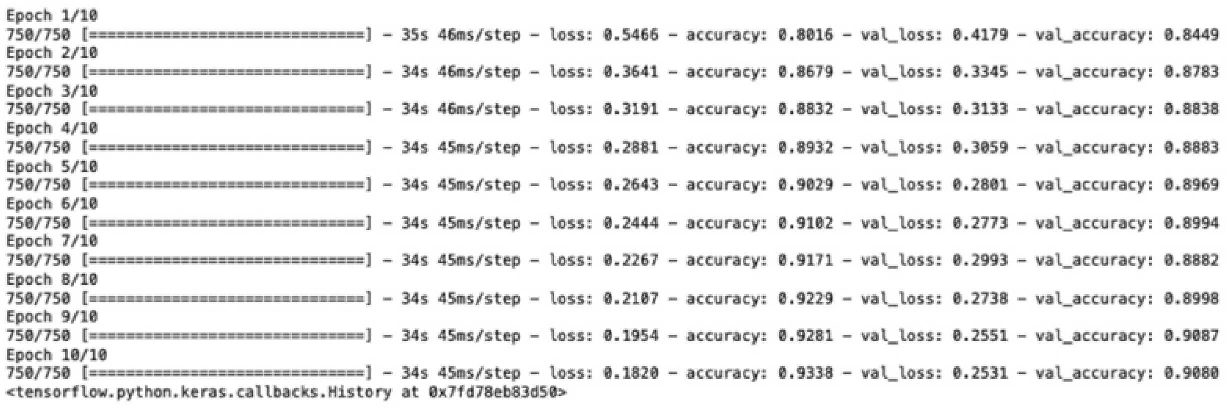
Figure 4.39: Fashion MNIST

Let us build a custom CNN model. Since the basic CNN architecture building process was explained in the previous MNIST tutorial, it has been skipped here, but we are building this baseline model to further compare it with the model we will be creating using KerasTuner:

```
from tensorflow import keras



from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dense, Flatten,
 Activation


(X_train, y_train), (X_test, y_test) = keras.datasets.fashion_mnist.load_data()



X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))

X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))

# Normalize pixel values between 0 and 1

X_train = X_train.astype('float32') / 255.0

X_test = X_test.astype('float32') / 255.0

model = keras.models.Sequential()
```

```
model.add(Conv2D(32, (3, 3), input_shape=X_train.shape[1:]))

model.add(Activation('relu'))

model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, (3, 3)))

model.add(Activation('relu'))

model.add(MaxPooling2D(pool_size=(2, 2)))



model.add(Flatten())

model.add(Dense(128))

model.add(Activation("relu"))

model.add(Dense(10))

model.add(Activation("softmax"))

model.compile(optimizer="adam",

            loss="sparse_categorical_crossentropy",


            metrics=["accuracy"])
```

model.fit(X_train, y_train, batch_size=64, epochs=10, validation_split=0.2)



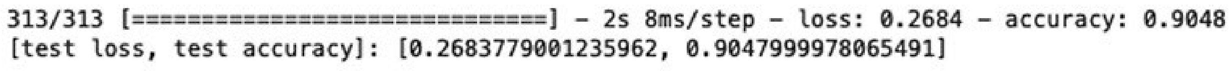
```
Epoch 1/10
750/750 [==============================] - 35s 46ms/step - loss: 0.5466 - accuracy: 0.8016 - val_loss: 0.4179 - val_accuracy: 0.8449
Epoch 2/10
750/750 [==============================] - 34s 46ms/step - loss: 0.3641 - accuracy: 0.8679 - val_loss: 0.3345 - val_accuracy: 0.8783
Epoch 3/10
750/750 [==============================] - 34s 46ms/step - loss: 0.3191 - accuracy: 0.8832 - val_loss: 0.3133 - val_accuracy: 0.8838
Epoch 4/10
750/750 [==============================] - 34s 45ms/step - loss: 0.2881 - accuracy: 0.8932 - val_loss: 0.3059 - val_accuracy: 0.8883
Epoch 5/10
750/750 [==============================] - 34s 45ms/step - loss: 0.2643 - accuracy: 0.9029 - val_loss: 0.2801 - val_accuracy: 0.8969
Epoch 6/10
750/750 [==============================] - 34s 45ms/step - loss: 0.2444 - accuracy: 0.9102 - val_loss: 0.2773 - val_accuracy: 0.8994
Epoch 7/10
750/750 [==============================] - 34s 45ms/step - loss: 0.2267 - accuracy: 0.9171 - val_loss: 0.2993 - val_accuracy: 0.8882
Epoch 8/10
750/750 [==============================] - 34s 45ms/step - loss: 0.2107 - accuracy: 0.9229 - val_loss: 0.2738 - val_accuracy: 0.8998
Epoch 9/10
750/750 [==============================] - 34s 45ms/step - loss: 0.1954 - accuracy: 0.9281 - val_loss: 0.2551 - val_accuracy: 0.9087
Epoch 10/10
750/750 [==============================] - 34s 45ms/step - loss: 0.1820 - accuracy: 0.9338 - val_loss: 0.2531 - val_accuracy: 0.9080
<tensorflow.python.keras.callbacks.History at 0x7fd78eb83d50>
```

Figure 4.40: Model run summary

Let us evaluate the model on the testing dataset and check the accuracy, as shown here:

eval_result = model.evaluate(X_test, y_test)

print("[test loss, test accuracy]:", eval_result)



```
313/313 [==============================] - 2s 8ms/step - loss: 0.2684 - accuracy: 0.9048
[test loss, test accuracy]: [0.2683779001235962, 0.9047999978065491]
```

Figure 4.41: Model evaluation summary

Hence, the basic CNN model built here has yielded an accuracy of 90.4%. Now, this model can be improved by tuning the hyperparameters like the number of layers in the network, number of neurons in the dense layers, choice of learning rate, choice of optimizer, choice of loss function, and so on.

But the answer for how to find the optimum hyperparameters is by ‘trial and error’. We need to experiment by building as many models as we can using different permutations and combinations of all the hyperparameters to derive the most desirable ones.

Automated tools for deriving the most optimum hyperparameters perform the same process, but with less coding and less effort. Let us demonstrate the usage of one such tool: KerasTuner.

Install KerasTuner

First, let us install keras-tuner using a pip installer on the notebook cell:

```
! pip install keras-tuner
```

Next, import the libraries required to implement KerasTuner:

```
import tensorflow as tf

from tensorflow import keras

from keras.datasets import fashion_mnist

from keras.layers import Dense, Conv2D, MaxPooling2D, Activation, Dense,
 Flatten

from keras_tuner import RandomSearch

from keras_tuner.engine.hyperparameters import HyperParameters
```

Now, load the data:

```
(X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()

print('Train: X=%s, y=%s' % (X_train.shape, y_train.shape))
```

print('Test: X=%s, y=%s' % (X_test.shape, y_test.shape))

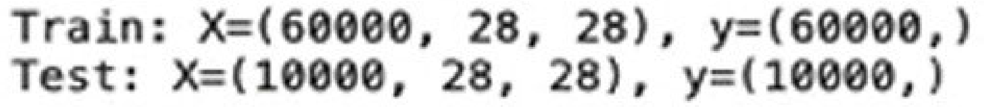

```
Train: X=(60000, 28, 28), y=(60000,)
Test: X=(10000, 28, 28), y=(10000,)
```

Figure 4.42: Train and test data shapes

Then, reshape and normalize the data as follows:

```
# reshape dataset to have a single channel


X_train = X_train.reshape((X_train.shape[0], 28, 28, 1))

X_test = X_test.reshape((X_test.shape[0], 28, 28, 1))

# Normalize pixel values between 0 and 1

X_train = X_train.astype('float32') / 255.0

X_test = X_test.astype('float32') / 255.0
```

Next, define the function that will build the model using various combinations of hyperparameters for experimentations, as shown here:

```
def build_model(hp):
```

```
model = keras.Sequential()

model.add(Conv2D(hp.Int('conv_0_units',

                    min_value=32,

                    max_value=256,

                    step=32),

                (3, 3),

                input_shape=X_train.shape[1:]))

model.add(Activation(hp.Choice(f"conv_0_activation",

                         ['relu']

                         )))

model.add(MaxPooling2D(pool_size=(2, 2)))

for i in range(hp.Int('conv_layers', 2, 5)):

  model.add(Conv2D(hp.Int(f'conv_{i+1}_units',

                      min_value=32,

                      max_value=256,
                    step=32),
                (3, 3)))

  model.add(Activation(hp.Choice(f"conv_{i+1}_activation",
                        ['relu']
                        )))
# this converts our 3D feature maps to 1D feature vectors
model.add(Flatten())
for i in range(hp.Int('dense_layers', 2, 5)):
  model.add(Dense(units=hp.Int(f'dense_{i}_units',
                min_value=32,
                max_value=512,
                step=32)))
  model.add(Activation(hp.Choice(f"dense_{i}_activation",
                        ['relu', 'sigmoid']
                    )))

  model.add(Dense(10))

  model.add(Activation('softmax'))

  model.compile(optimizer=keras.optimizers.Adam(

 # Choose an optimal value from 0.01, 0.001, or 0.0001

              learning_rate=hp.Choice('learning_rate',

                             values=[1e-2, 1e-3, 1e-4])),


              loss=hp.Choice('loss_function',

                             values=['sparse_categorical_crossentropy']),

              metrics=['accuracy'])

  return model
```

We create the tuner object and pass the required arguments:

```
tuner = RandomSearch(

    build_model,
    objective='val_accuracy',

    max_trials=3,

    executions_per_trial=3,

    directory='.',

    project_name='keras_tuner'

)
```

Further, we create an object for early stopping of the network to avoid overfitting:

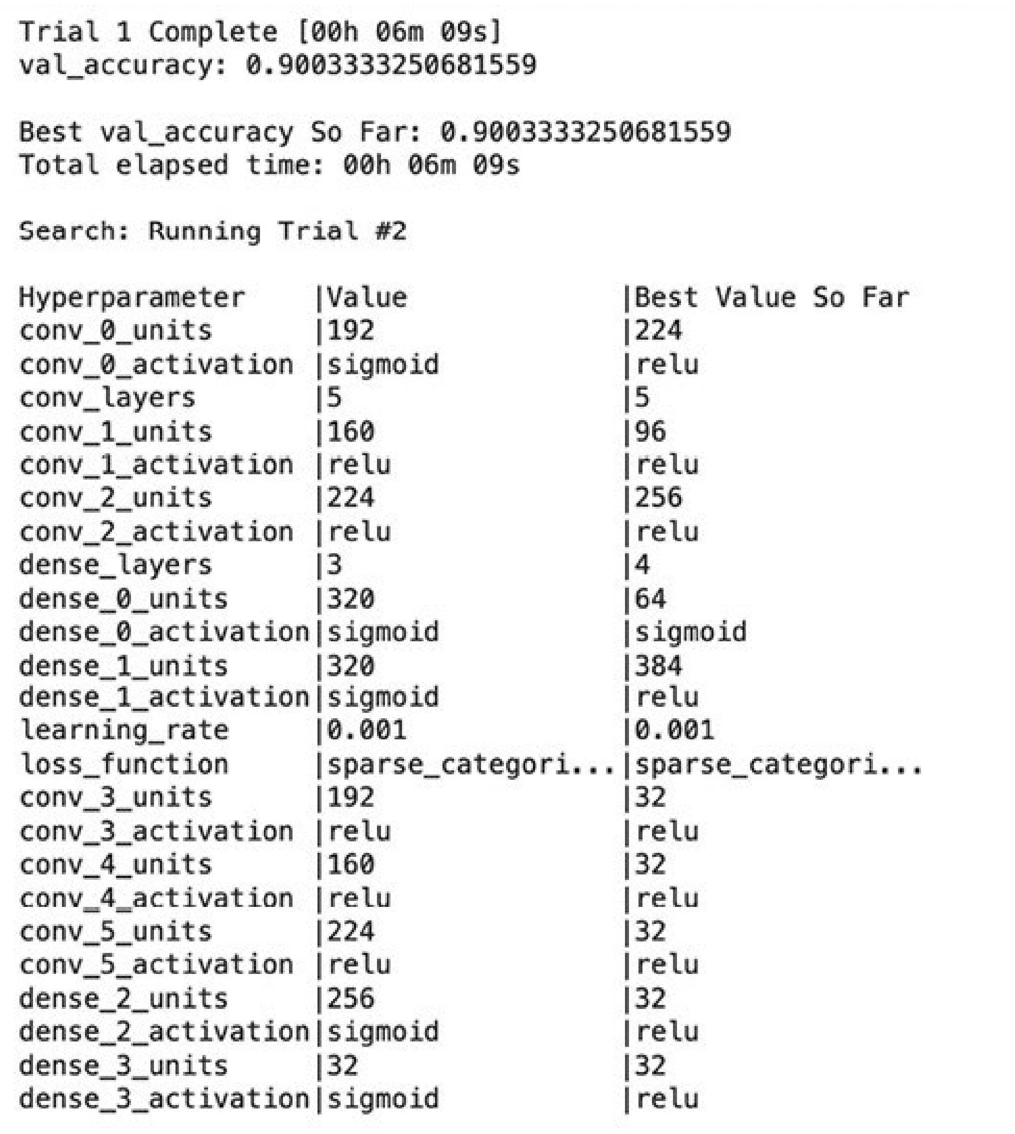
```
stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
```

Further, we search the best model hyperparameters using the tuner search module:

```
tuner.search(x=X_train,

             y=y_train,

             epochs=10,

             validation_split=0.2,
```

callbacks=[stop_early])

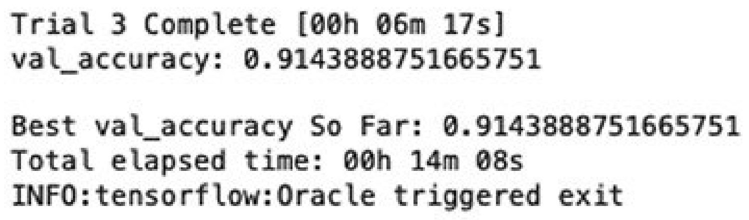
While the tuner search is in progress, you can see the architecture of the current model in progress and the best architecture found so far, with their respective accuracy, as depicted in Figure



```
Trial 1 Complete [00h 06m 09s]
val_accuracy: 0.9003333250681559

Best val_accuracy So Far: 0.9003333250681559
Total elapsed time: 00h 06m 09s

Search: Running Trial #2

Hyperparameter     |Value              |Best Value So Far
conv_0_units       |192                |224
conv_0_activation  |sigmoid            |relu
conv_layers        |5                  |5
conv_1_units       |160                |96
conv_1_activation  |relu               |relu
conv_2_units       |224                |256
conv_2_activation  |relu               |relu
dense_layers       |3                  |4
dense_0_units      |320                |64
dense_0_activation|sigmoid             |sigmoid
dense_1_units      |320                |384
dense_1_activation|sigmoid             |relu
learning_rate      |0.001              |0.001
loss_function      |sparse_categori...|sparse_categori...
conv_3_units       |192                |32
conv_3_activation  |relu               |relu
conv_4_units       |160                |32
conv_4_activation  |relu               |relu
conv_5_units       |224                |32
conv_5_activation  |relu               |relu
dense_2_units      |256                |32
dense_2_activation|sigmoid             |relu
dense_3_units      |32                 |32
dense_3_activation|sigmoid             |relu
```

Figure 4.43: KerasTuner run summary

After the tuner search completes all the defined trials, the output will show the best accuracy achieved during the entire experimentation process, as shown here:

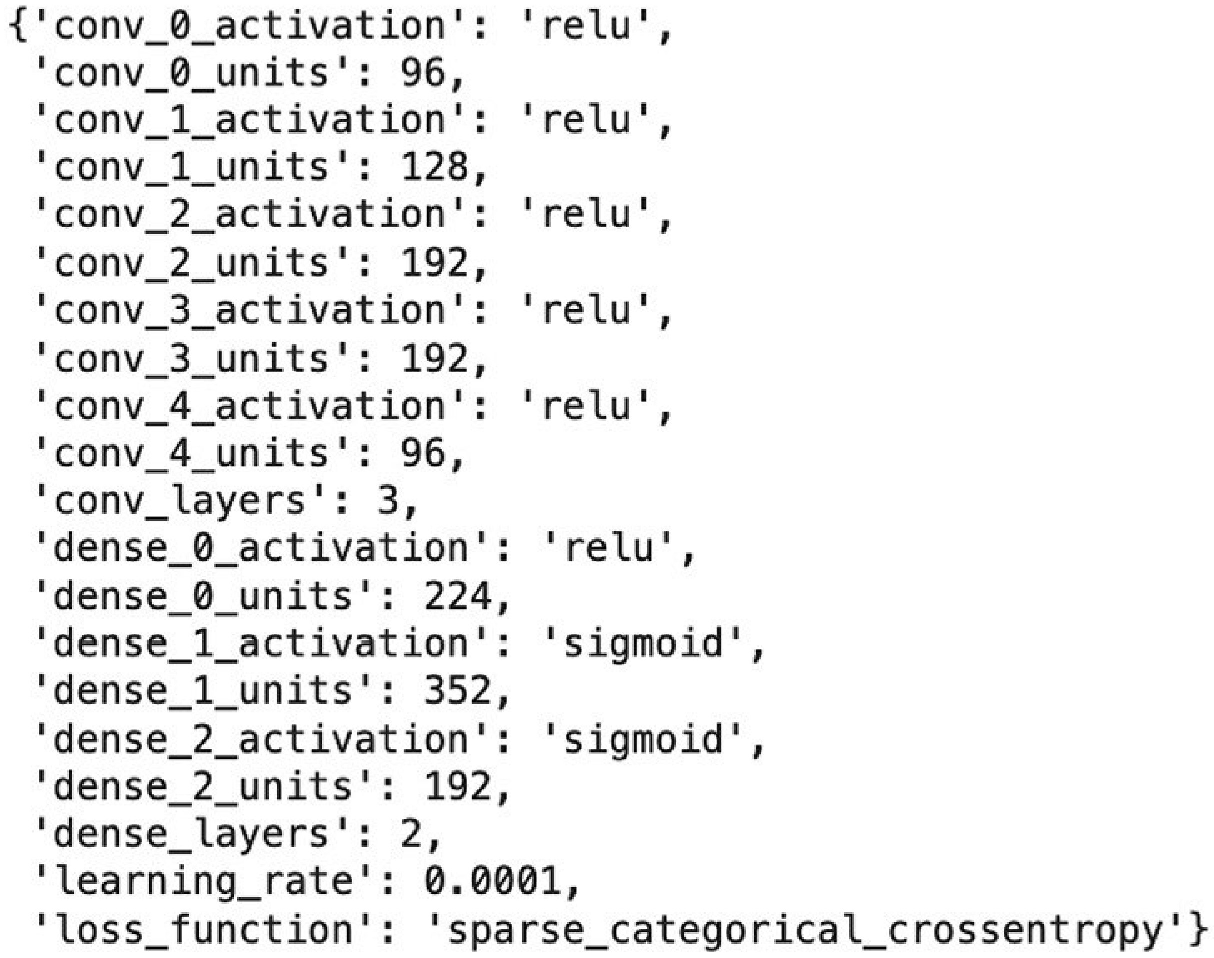


```
Trial 3 Complete [00h 06m 17s]
val_accuracy: 0.9143888751665751

Best val_accuracy So Far: 0.9143888751665751
Total elapsed time: 00h 14m 08s
INFO:tensorflow:Oracle triggered exit
```

Figure 4.44: Best accuracy

To check the best hyperparameter values for the best model identified by the tuner search, use the get_best_hyperparameters method:

```
tuner.get_best_hyperparameters()[0].values
```

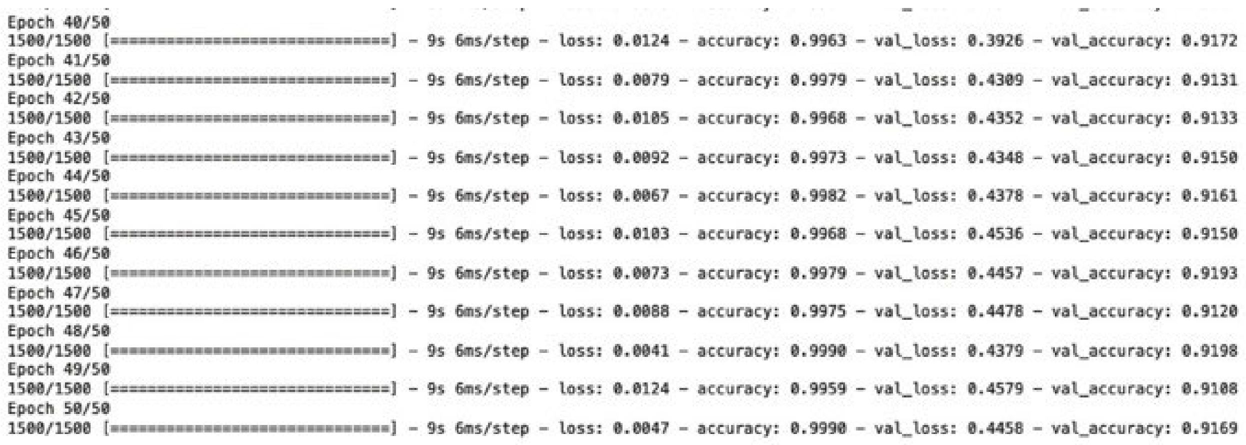


```
{'conv_0_activation': 'relu',
 'conv_0_units': 96,
 'conv_1_activation': 'relu',
 'conv_1_units': 128,
 'conv_2_activation': 'relu',
 'conv_2_units': 192,
 'conv_3_activation': 'relu',
 'conv_3_units': 192,
 'conv_4_activation': 'relu',
 'conv_4_units': 96,
 'conv_layers': 3,
 'dense_0_activation': 'relu',
 'dense_0_units': 224,
 'dense_1_activation': 'sigmoid',
 'dense_1_units': 352,
 'dense_2_activation': 'sigmoid',
 'dense_2_units': 192,
 'dense_layers': 2,
 'learning_rate': 0.0001,
 'loss_function': 'sparse_categorical_crossentropy'}
```

Figure 4.45: Best hyperparameters

Now, to train the model, we use the best hyperparameters found by the tuner:

```
best_hps=tuner.get_best_hyperparameters(num_trials=1)[0]

model = tuner.hypermodel.build(best_hps)

history = model.fit(X_train, y_train, epochs=50, validation_split=0.2)
```

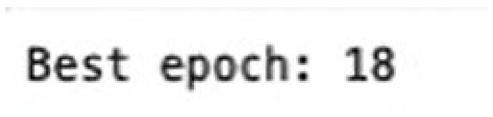


```
Epoch 40/50
1500/1500 [==============================] - 9s 6ms/step - loss: 0.0124 - accuracy: 0.9963 - val_loss: 0.3926 - val_accuracy: 0.9172
Epoch 41/50
1500/1500 [==============================] - 9s 6ms/step - loss: 0.0079 - accuracy: 0.9979 - val_loss: 0.4309 - val_accuracy: 0.9131
Epoch 42/50
1500/1500 [==============================] - 9s 6ms/step - loss: 0.0105 - accuracy: 0.9968 - val_loss: 0.4352 - val_accuracy: 0.9133
Epoch 43/50
1500/1500 [==============================] - 9s 6ms/step - loss: 0.0092 - accuracy: 0.9973 - val_loss: 0.4348 - val_accuracy: 0.9150
Epoch 44/50
1500/1500 [==============================] - 9s 6ms/step - loss: 0.0067 - accuracy: 0.9982 - val_loss: 0.4378 - val_accuracy: 0.9161
Epoch 45/50
1500/1500 [==============================] - 9s 6ms/step - loss: 0.0103 - accuracy: 0.9968 - val_loss: 0.4536 - val_accuracy: 0.9150
Epoch 46/50
1500/1500 [==============================] - 9s 6ms/step - loss: 0.0073 - accuracy: 0.9979 - val_loss: 0.4457 - val_accuracy: 0.9193
Epoch 47/50
1500/1500 [==============================] - 9s 6ms/step - loss: 0.0088 - accuracy: 0.9975 - val_loss: 0.4478 - val_accuracy: 0.9120
Epoch 48/50
1500/1500 [==============================] - 9s 6ms/step - loss: 0.0041 - accuracy: 0.9990 - val_loss: 0.4379 - val_accuracy: 0.9198
Epoch 49/50
1500/1500 [==============================] - 9s 6ms/step - loss: 0.0124 - accuracy: 0.9959 - val_loss: 0.4579 - val_accuracy: 0.9108
Epoch 50/50
1500/1500 [==============================] - 9s 6ms/step - loss: 0.0047 - accuracy: 0.9990 - val_loss: 0.4458 - val_accuracy: 0.9169
```

Figure 4.46: Model run summary

We check the optimum epoch that gives the highest validation accuracy during the training process. This is to avoid overtraining of the model and train only for the number of epochs required:

```
val_acc_per_epoch = history.history['val_accuracy']

best_epoch = val_acc_per_epoch.index(max(val_acc_per_epoch)) + 1

print('Best epoch: %d' % (best_epoch,))
```



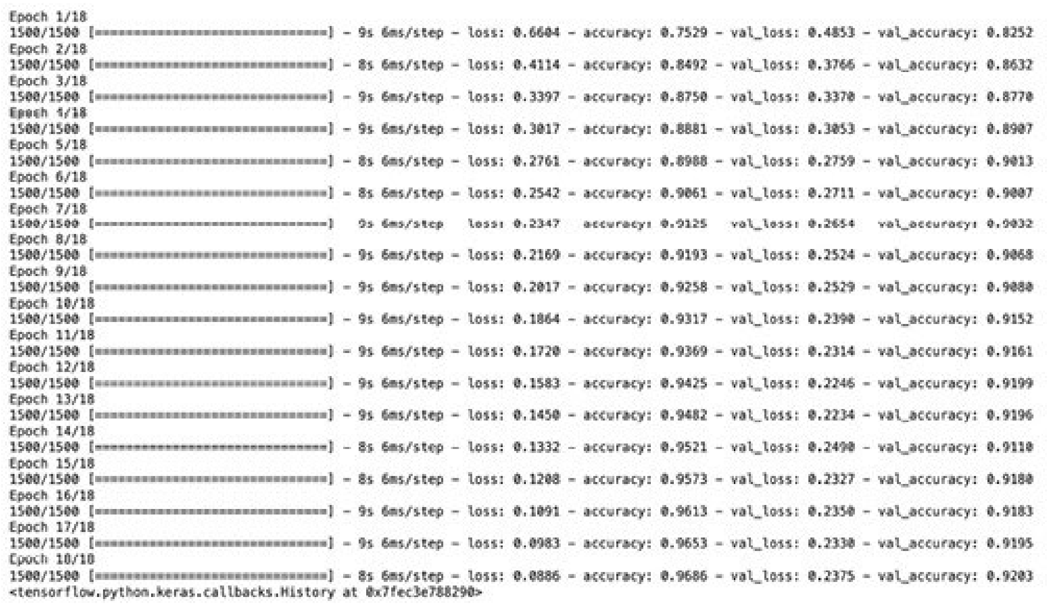
```
Best epoch: 18
```

Figure 4.47: Best epoch

We then create a hypermodel by training against the best hyperparameters and best epoch found:

```
hypermodel = tuner.hypermodel.build(best_hps)
```

```
hypermodel.fit(X_train, y_train, epochs=best_epoch, validation_split=0.2)
```

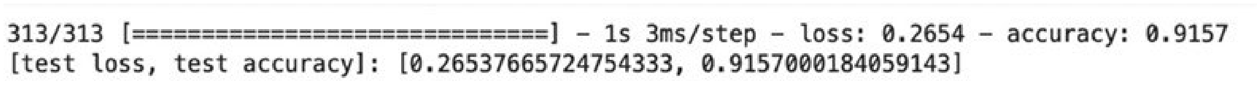


```
Epoch 1/18
1500/1500 [==============================] - 9s 6ms/step - loss: 0.6604 - accuracy: 0.7529 - val_loss: 0.4853 - val_accuracy: 0.8252
Epoch 2/18
1500/1500 [==============================] - 8s 6ms/step - loss: 0.4114 - accuracy: 0.8492 - val_loss: 0.3766 - val_accuracy: 0.8632
Epoch 3/18
1500/1500 [==============================] - 9s 6ms/step - loss: 0.3397 - accuracy: 0.8750 - val_loss: 0.3370 - val_accuracy: 0.8770
Epoch 4/18
1500/1500 [==============================] - 9s 6ms/step - loss: 0.3017 - accuracy: 0.8881 - val_loss: 0.3053 - val_accuracy: 0.8907
Epoch 5/18
1500/1500 [==============================] - 8s 6ms/step - loss: 0.2761 - accuracy: 0.8988 - val_loss: 0.2759 - val_accuracy: 0.9013
Epoch 6/18
1500/1500 [==============================] - 8s 6ms/step - loss: 0.2542 - accuracy: 0.9061 - val_loss: 0.2711 - val_accuracy: 0.9007
Epoch 7/18
1500/1500 [==============================]   9s 6ms/step   loss: 0.2347   accuracy: 0.9125   val_loss: 0.2654   val_accuracy: 0.9032
Epoch 8/18
1500/1500 [==============================] - 9s 6ms/step - loss: 0.2169 - accuracy: 0.9193 - val_loss: 0.2524 - val_accuracy: 0.9068
Epoch 9/18
1500/1500 [==============================] - 9s 6ms/step - loss: 0.2017 - accuracy: 0.9258 - val_loss: 0.2529 - val_accuracy: 0.9080
Epoch 10/18
1500/1500 [==============================] - 9s 6ms/step - loss: 0.1864 - accuracy: 0.9317 - val_loss: 0.2390 - val_accuracy: 0.9152
Epoch 11/18
1500/1500 [==============================] - 9s 6ms/step - loss: 0.1720 - accuracy: 0.9369 - val_loss: 0.2314 - val_accuracy: 0.9161
Epoch 12/18
1500/1500 [==============================] - 9s 6ms/step - loss: 0.1583 - accuracy: 0.9425 - val_loss: 0.2246 - val_accuracy: 0.9199
Epoch 13/18
1500/1500 [==============================] - 9s 6ms/step - loss: 0.1450 - accuracy: 0.9482 - val_loss: 0.2234 - val_accuracy: 0.9196
Epoch 14/18
1500/1500 [==============================] - 8s 6ms/step - loss: 0.1332 - accuracy: 0.9521 - val_loss: 0.2490 - val_accuracy: 0.9110
Epoch 15/18
1500/1500 [==============================] - 8s 6ms/step - loss: 0.1208 - accuracy: 0.9573 - val_loss: 0.2327 - val_accuracy: 0.9180
Epoch 16/18
1500/1500 [==============================] - 9s 6ms/step - loss: 0.1091 - accuracy: 0.9613 - val_loss: 0.2350 - val_accuracy: 0.9183
Epoch 17/18
1500/1500 [==============================] - 9s 6ms/step - loss: 0.0983 - accuracy: 0.9653 - val_loss: 0.2330 - val_accuracy: 0.9195
Epoch 18/18
1500/1500 [==============================] - 8s 6ms/step - loss: 0.0886 - accuracy: 0.9686 - val_loss: 0.2375 - val_accuracy: 0.9203
<tensorflow.python.keras.callbacks.History at 0x7fec3e788290>
```

Figure 4.48: Model run summary

Finally, we check the accuracy for the best model created using the test dataset:

```
eval_result = hypermodel.evaluate(X_test, y_test)

print("[test loss, test accuracy]:", eval_result)
```



```
313/313 [==============================] - 1s 3ms/step - loss: 0.2654 - accuracy: 0.9157
[test loss, test accuracy]: [0.26537665724754333, 0.9157000184059143]
```

Figure 4.49: Model evaluation summary

So, we can see that we were able to improve the accuracy by 1% using the KerasTuner tool to find the best hyperparameters. However, this can be improved all the more by further tuning and by increasing the number of trials and experiments that should be performed by the KerasTuner, which was kept as minimum for this demonstration.

This is how KerasTuner automates the process of hyperparameter tuning and facilitates the process of coming up with the best model suited for the problem statement and the dataset available for the model training.

Conclusion

In this chapter, we looked at various components of convolutional neural networks. We understood the process of convolutions and how images are convolved using kernels or filters to generate various feature maps of the image, which can be used to train an artificial neural network for image classification problems. We also saw how tuning the hyperparameters of the network can improve the accuracy of the model. This can be achieved by either manually performing various experimentations or by utilizing the automated tools like KerasTuner to find the optimum hyperparameters for the network you are trying to build.

In the next chapter, we will explore different CNN architectures developed by researchers. The parameters of these models have been made publicly available, allowing others to utilize them in their work. The process of utilizing the learned parameters of other networks in your own problem statement is called transfer learning. We will cover this topic in detail in the next chapter and use Python to solve image classification problems using transfer learning.

Questions

CNN is mostly used for which of the following?

Structured data

Unstructured data

Both

None

What are the different problems that CNN can solve?

Image classification

Object detection

Image segmentation

All of the above

A convolutional neural network broadly consists of which of these?

Convolutional layers

Fully connected dense layers

Convolutional and fully connected dense layers

Convolutional, pooling, and fully connected dense layers

Which is transforming a two-dimensional feature array into a one-dimensional vector called?

Pooling

Padding

Flattening

None of the above

What is adding a layer of pixels in an image array for convolution purposes called?

Pooling

Padding

Flattening

None of the above

Answers

(c)

(d)

(d)

(c)

(b)

CHAPTER 5
Optical Character Recognition

OCR technology liberates information trapped on paper.

— anonymous

Introduction

Optical Character Recognition is a technology used in computer vision that enables machines to recognize text from images or scanned documents. OCR works by analyzing the pixel patterns of an image to identify the shapes of individual letters and words, and then converting them into machine-readable text.

OCR is used in various fields, such as healthcare, finance, legal, and government, to digitize important documents and make them easier to access and process. OCR involves several steps, including image preprocessing, segmentation, feature extraction, and classification, which are used to improve the accuracy of the recognition process.

The OCR technology has greatly improved the speed and efficiency of document processing and has revolutionized the way we work with large amounts of text-based data. As the technology continues to improve, it is expected to play an even more significant role in the digital transformation of various industries, making it easier to access and analyze data, and improving the speed and accuracy of document processing.

Structure

In this chapter, we will cover the following topics:

Optical character recognition

OCR Python libraries and their implementation

Tesseract OCR

keras-ocr

EasyOCR

TrOCR

Objectives

After studying this chapter, you should be able to understand the concept of Optical character recognition, and you should be familiar with the various libraries available to solve reading text from images and other data sources, like noneditable PDF files.

Optical character recognition

Optical Character Recognition is a technique that transforms text in images into machine-readable text. This method is commonly applied to items such as invoices, bank statements, restaurant receipts, signboards, traffic symbols, and handwritten texts. Converting these visual representations to text is beneficial for tasks like extracting information, digitizing books or documents into PDFs, and online processing, such as text-to-speech. This last feature is especially valuable for the visually impaired and is often employed in autonomous vehicles for interpretation. The field of OCR is continuously evolving, with advancements aimed at enhancing accuracy and performance.

Refer to the following figure:

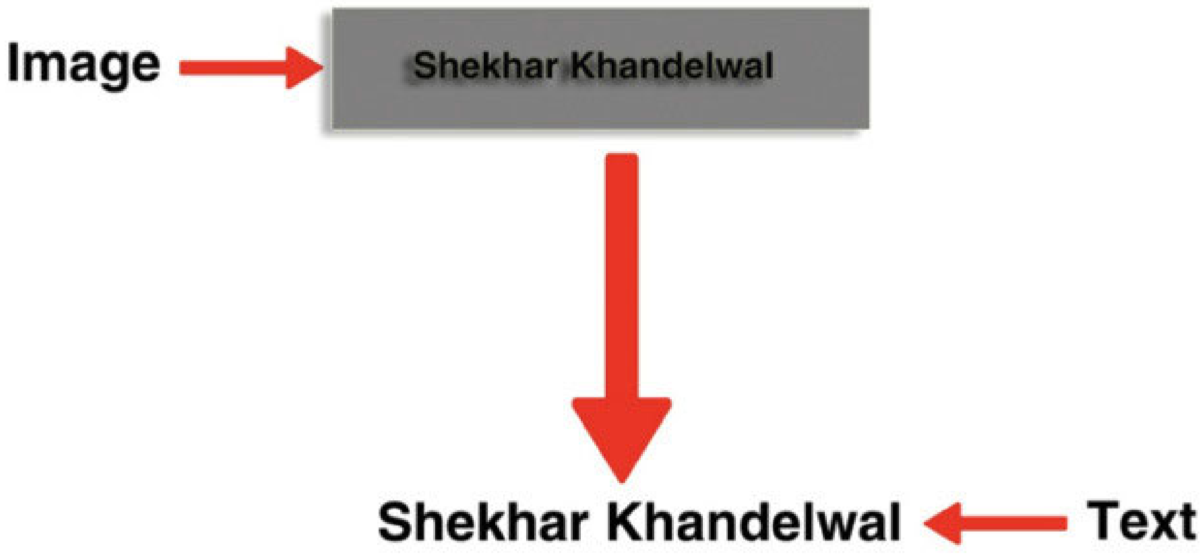


Figure 5.1: OCR working principle

OCR Python libraries and their implementation

In this section, we will learn about the OCR Python libraries and their implementation.

Tesseract OCR

Between 1985 and 1994, Hewlett-Packard Laboratories in Bristol, United Kingdom, and Hewlett-Packard Co. in Greeley, Colorado, USA, developed Tesseract. Some additional improvements were made in 1996 to port it to Windows, and some C++ization was done in 1998. HP released the source code for Tesseract in 2005. It was created by Google between 2006 and November 2018.

The most recent stable version, major version 5, was introduced on November 30, 2021, with the release of 5.0.0. Refer to the following figure:

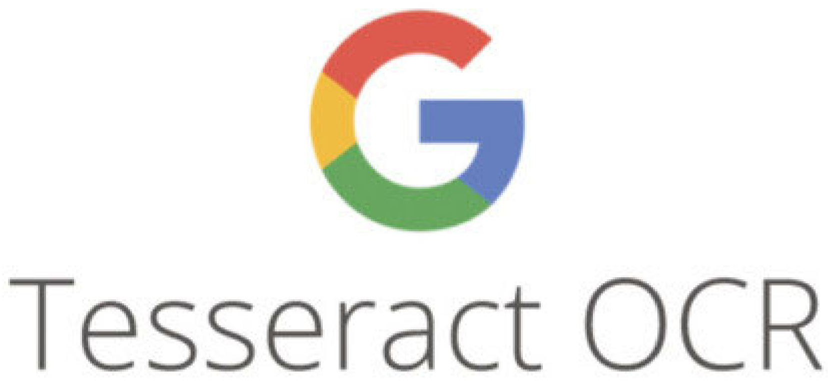


Figure 5.2: Tesseract OCR from Google

The classic Tesseract OCR engine, which recognizes character patterns, is still supported, in addition to the new neural net Long Short Term Memory OCR engine that is focused on line identification.

Tesseract OCR demo

Let us start with the demo:

Importing important libraries:

```
import cv2

import pytesseract
```

Giving the tesseract.exe file path:
Windows:

```
pytesseract.pytesseract.tesseract_cmd=r'C:\Program Files\Tesseract-OCR\tesseract.exe'
```

Mac:

```
pytesseract.pytesseract.tesseract_cmd=r'\usr\local\bin\tesseract'
```

Importing image in cv2 and displaying in Windows:

```
img = cv2.imread("image.jpg")

img = cv2.resize(img, (400, 450))

cv2.imshow("Image", img)
```

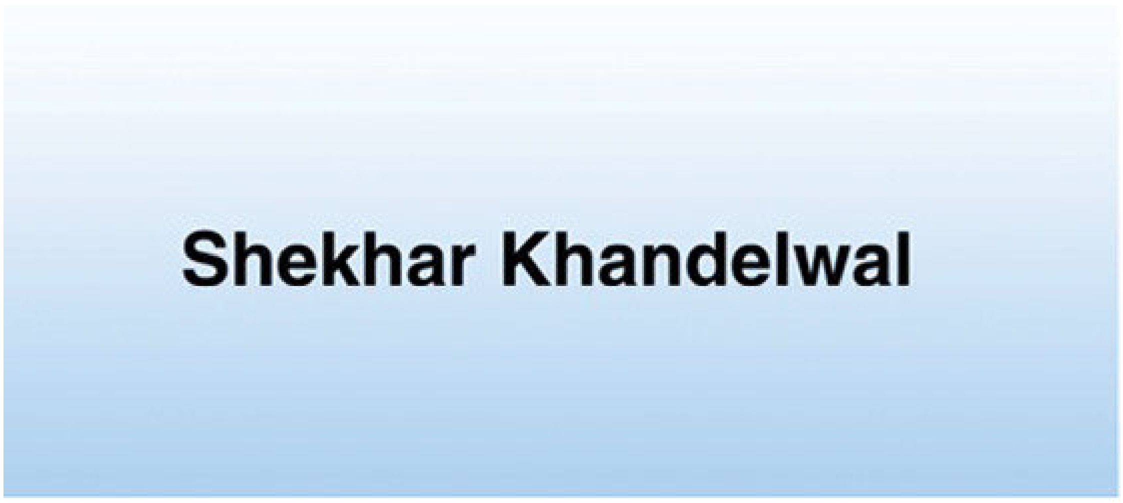


Figure 5.3: Sample image with imprinted text

Passing image to pytesseract and getting image text as output:

```
text = pytesseract.image_to_string(img)



print(text)
```

Output:

```
Shekhar Khandelwal
```

Destroying all the windows:

```
cv2.waitKey(0)

cv2.destroyAllWindows()
```

keras-ocr

The keras-ocr library provides a high-level API and end-to-end training pipeline to build new OCR models. In the next section, we will see a step-by-step tutorial using keras-ocr to extract text from multiple images, as shown in the following figure:

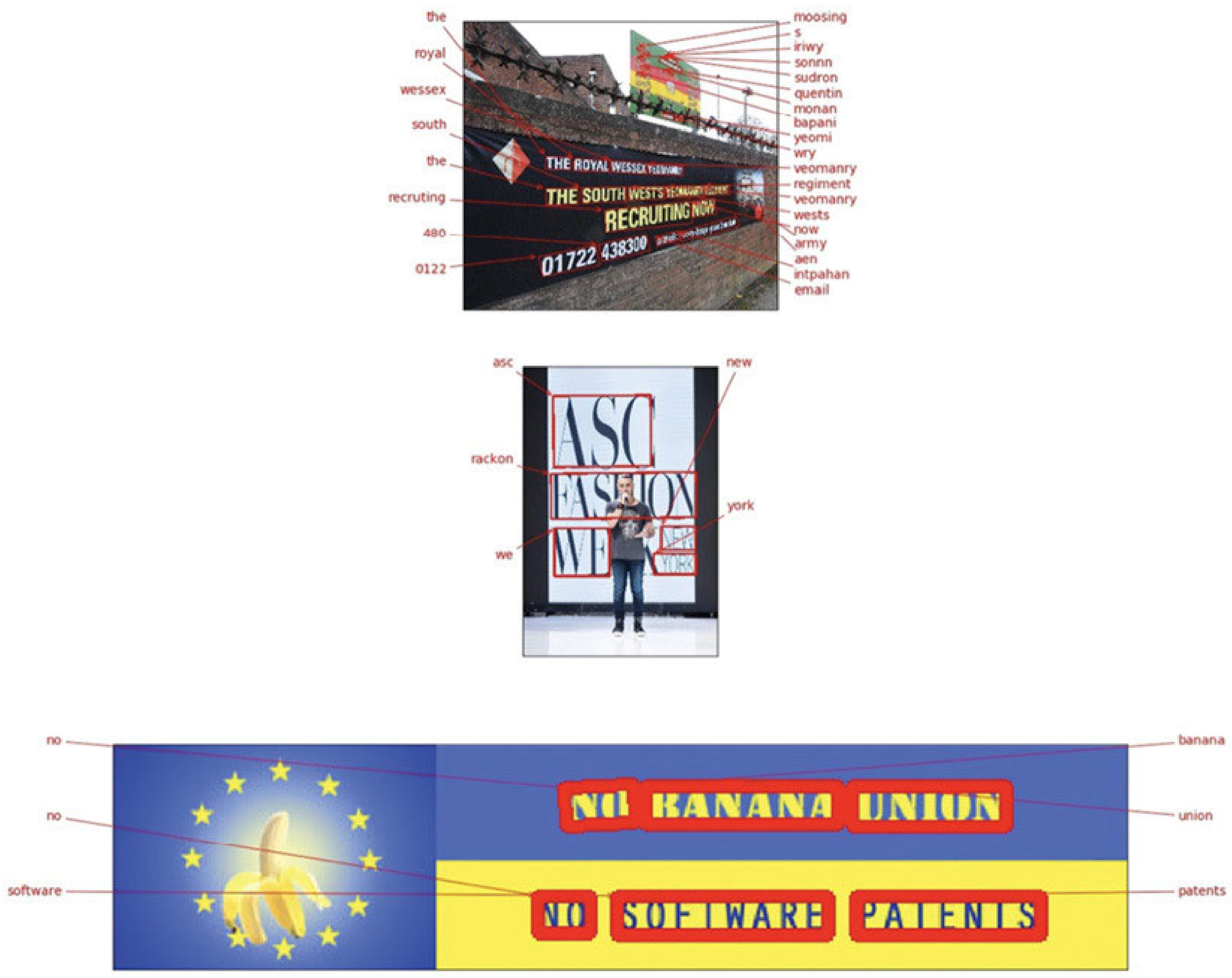


Figure 5.4: Sample bounding box against imprinted text on images

keras-ocr demo

We will construct a keras-ocr pipeline in this part to extract text from a few test images. For this tutorial, we will use Google Colab.

Using the following code, let us first install the keras-ocr library (supports Python >= 3.6 and TensorFlow >= 2.0.0):

Installing important libraries:

```
!pip install -q keras-ocr
```

```
!pip install matplotlib
```

Importing keras_ocr and matplotlib for visualizing images in a notebook only:

```
import keras_ocr
```

```
import matplotlib.pyplot as plt
```

Importing the keras pipeline:

```
pipeline = keras_ocr.pipeline.Pipeline()
```

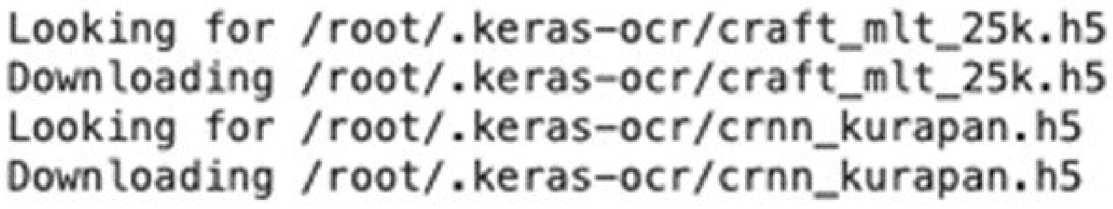

```
Looking for /root/.keras-ocr/craft_mlt_25k.h5
Downloading /root/.keras-ocr/craft_mlt_25k.h5
Looking for /root/.keras-ocr/crnn_kurapan.h5
Downloading /root/.keras-ocr/crnn_kurapan.h5
```

Figure 5.5: Console output

Importing images for processing:

```
images = [

        keras_ocr.tools.read(img) for img in ['keras-ocr-sample1.png',

                                              'keras-ocr-sample2.png'

                                              ]

]
```

Plotting image and checking in notebook:

```
plt.figure(figsize=(10,20))

plt.imshow(images[0])
```

```
<matplotlib.image.AxesImage at 0x7fc00e0b5970>
```

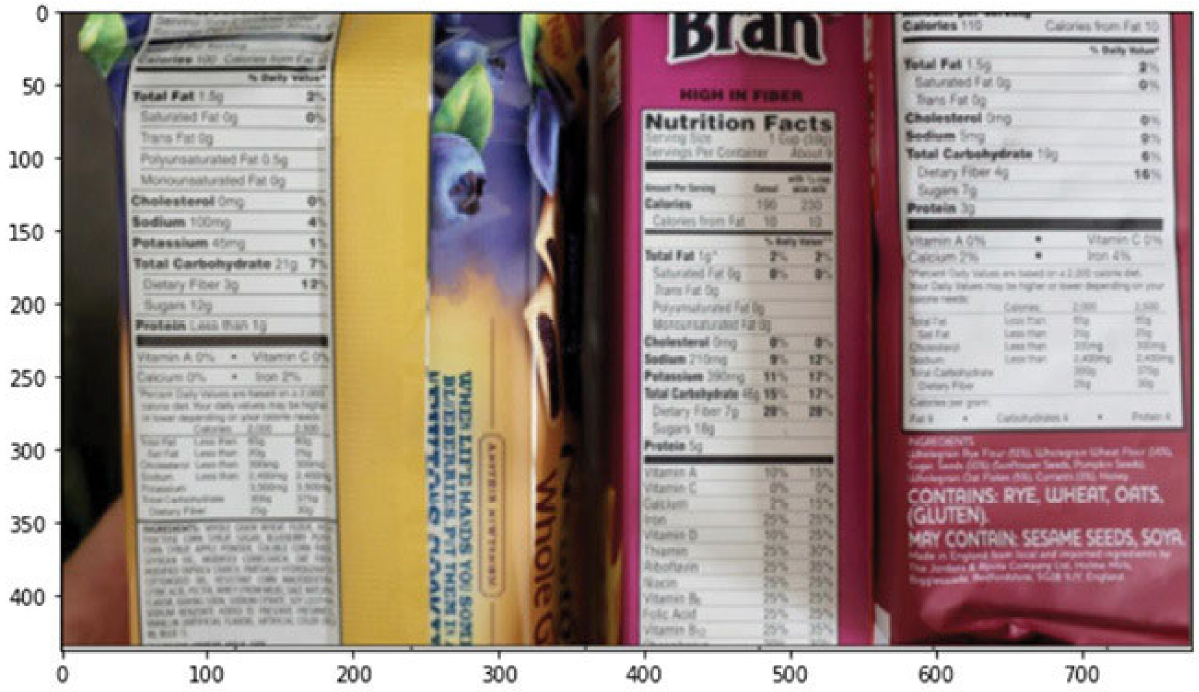
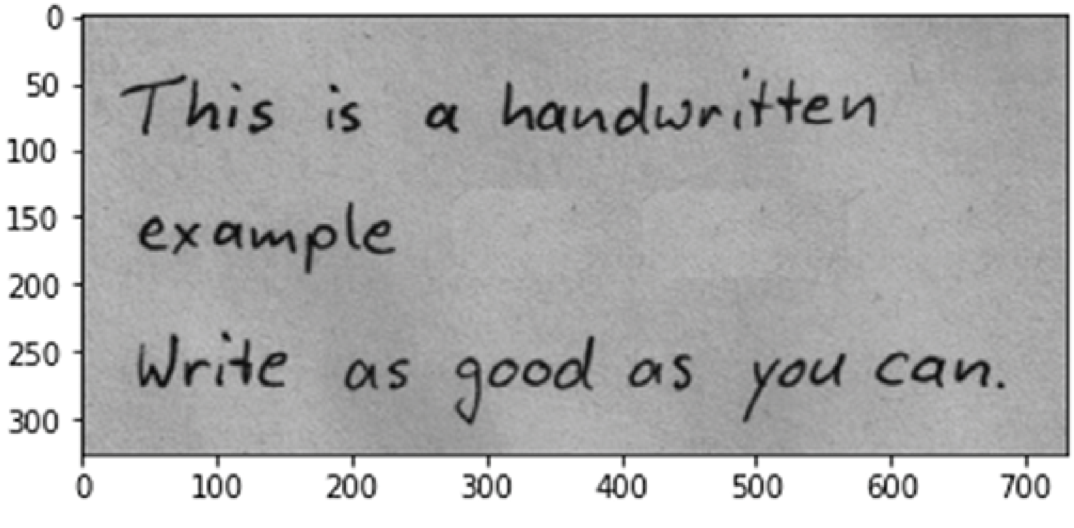

Bran
HIGH IN FIBER
Nutrition Facts
CONTAINS: RYE, WHEAT, OATS, (GLUTEN).
MAY CONTAIN: SESAME SEEDS, SOYA.

Figure 5.6: Text imprinted on images of cartons

plt.imshow(images[1])

```
<matplotlib.image.AxesImage at 0x7fc0077e4940>
```



This is a handwritten
example
Write as good as you can.

Figure 5.7: Text imprinted on the image of letter

Sending image to the pipeline for detection; you can choose n number of images and put the image objects in an array and pass it:

prediction_groups = pipeline.recognize(image)

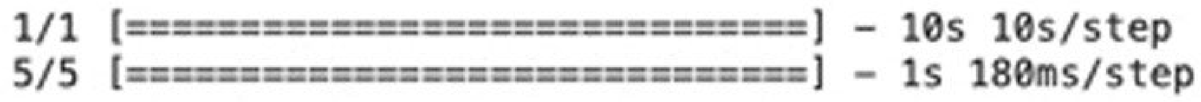

```
1/1 [==============================] - 10s 10s/step
5/5 [==============================] - 1s 180ms/step
```

Figure 5.8: Console Output

Passing all the images in keras_ocr and getting prediction and output in the notebook:

```
fig, axs = plt.subplots(nrows=len(image), figsize=(10, 20))

for ax, image, predictions in zip(axs, image, prediction_groups):

  keras_ocr.tools.drawAnnotations(image=image,

                                  predictions=predictions,

                                  ax=ax

                                  )
```

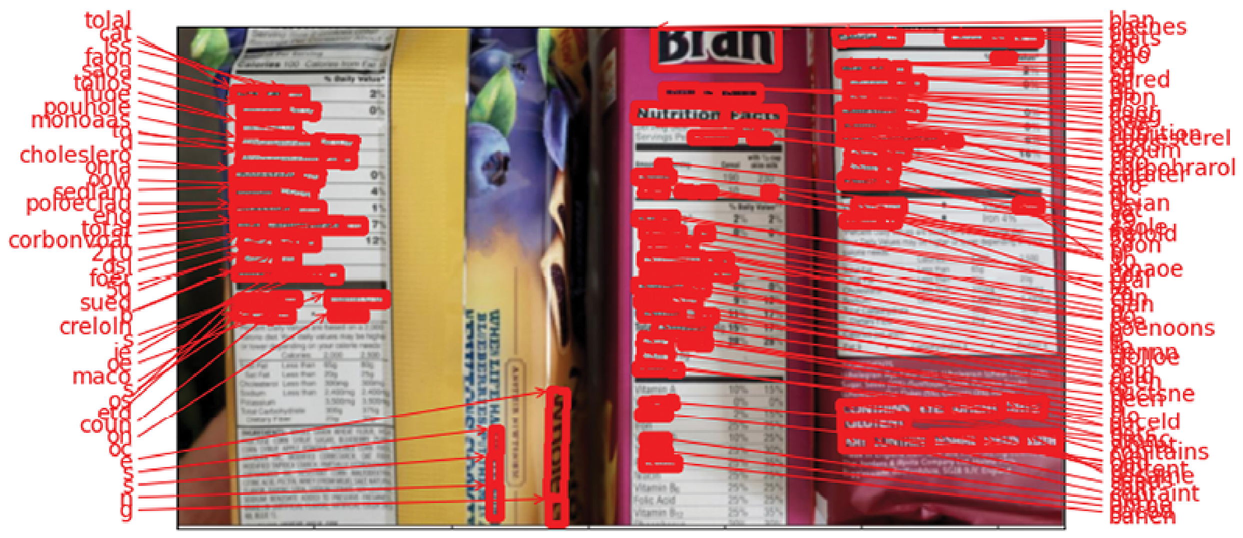


Figure 5.9 Bounding box against all detected text for image 1

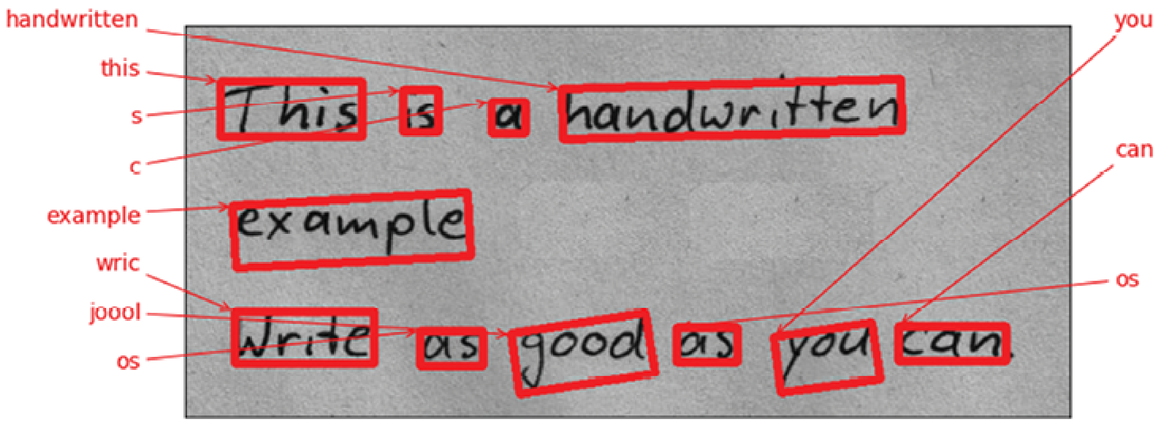


Figure 5.9 (b): Bounding box against all detected text for image 1

The prediction_groups object can be further processed to access the extracted text from the images.

EasyOCR

EasyOCR Python package makes it possible to convert images into text. It has access to over 70 languages, including English, Chinese, Japanese, Korean, and Hindi, and many more are being added. It is, by far, the simplest approach to implementing OCR. Jaided AI is the firm that developed EasyOCR.

EasyOCR demo

Text detection in images with EasyOCR can be done using the following steps:

Installing important libraries:

```
!apt-get install poppler-utils

!pip install pdf2image

!pip install easyocr
```

Importing libraries:

```
from pdf2image import convert_from_path

import easyocr

import numpy as np

import PIL

from PIL import ImageDraw

import spacy
```

```
from IPython.display import display, Image
```

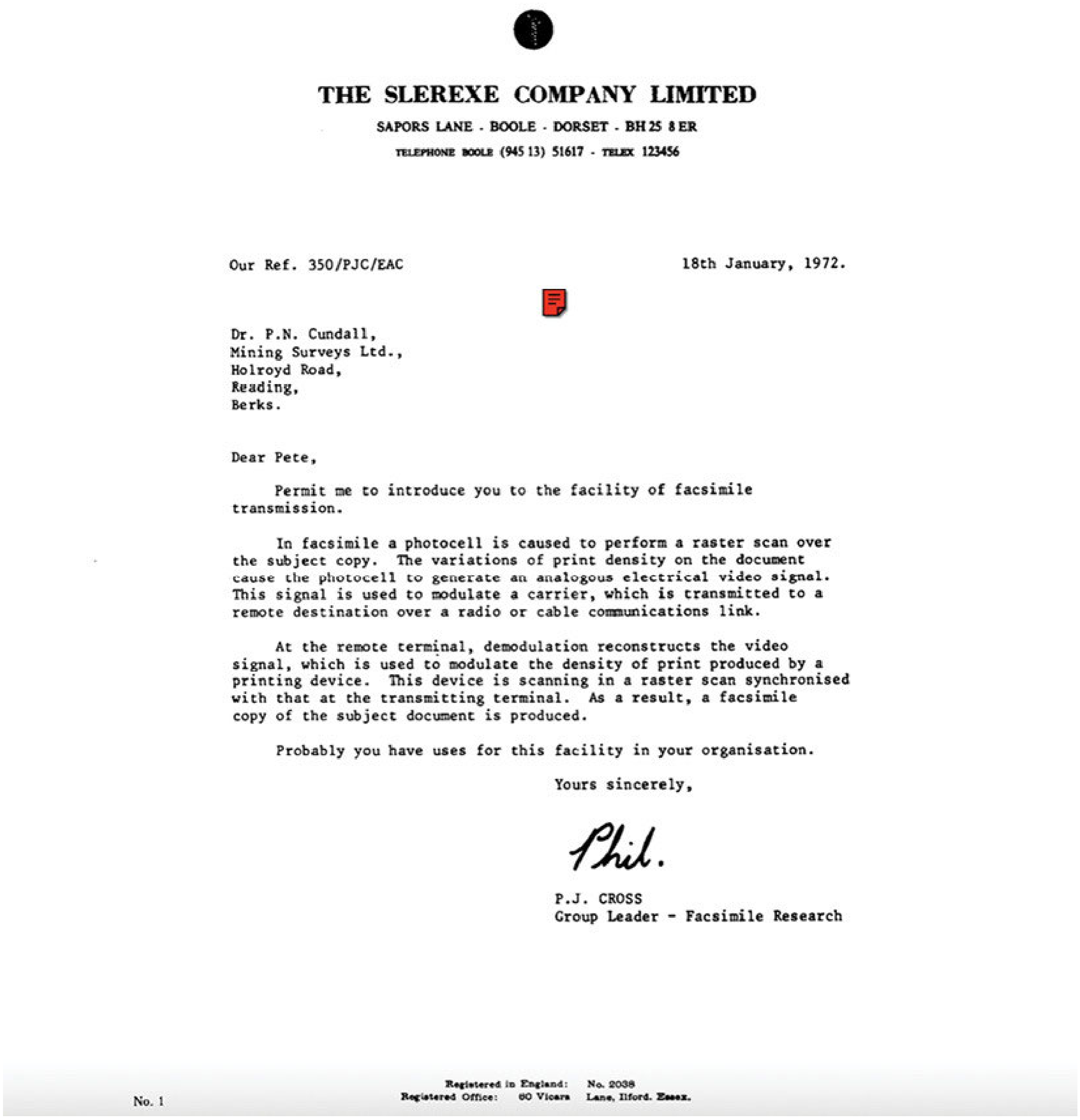
Setting our language to which we want to detect in our images:

```
reader=easyocr.Reader(['en'])
```

Importing the PDF which we want to detect text in.
Download a sample PDF from this URL:
http://solutions.weblite.ca/pdfocrx/scansmpl.pdf

```
images=convert_from_path("scansmpl.pdf")
```

```
display(images[0])
```



THE SLEREXE COMPANY LIMITED

SAPORS LANE - BOOLE - DORSET - BH 25 8 ER

TELEPHONE BOOLE (945 13) 51617 - TELEX 123456

Our Ref. 350/PJC/EAC 18th January, 1972.

Dr. P.N. Cundall,
Mining Surveys Ltd.,
Holroyd Road,
Reading,
Berks.

Dear Pete,

Permit me to introduce you to the facility of facsimile transmission.

In facsimile a photocell is caused to perform a raster scan over the subject copy. The variations of print density on the document cause the photocell to generate an analogous electrical video signal. This signal is used to modulate a carrier, which is transmitted to a remote destination over a radio or cable communications link.

At the remote terminal, demodulation reconstructs the video signal, which is used to modulate the density of print produced by a printing device. This device is scanning in a raster scan synchronised with that at the transmitting terminal. As a result, a facsimile copy of the subject document is produced.

Probably you have uses for this facility in your organisation.

Yours sincerely,

Phil.

P.J. CROSS
Group Leader - Facsimile Research

No. 1

Registered in England: No. 2038
Registered Office: 60 Vicars Lane, Ilford. Essex.

Figure 5.10: Image of letter with imprinted text

Reading text from images using readtext from the reader:

```
bounds = reader.readtext(np.array(images[0]))

print(bounds)
```

This will generate an output like this:

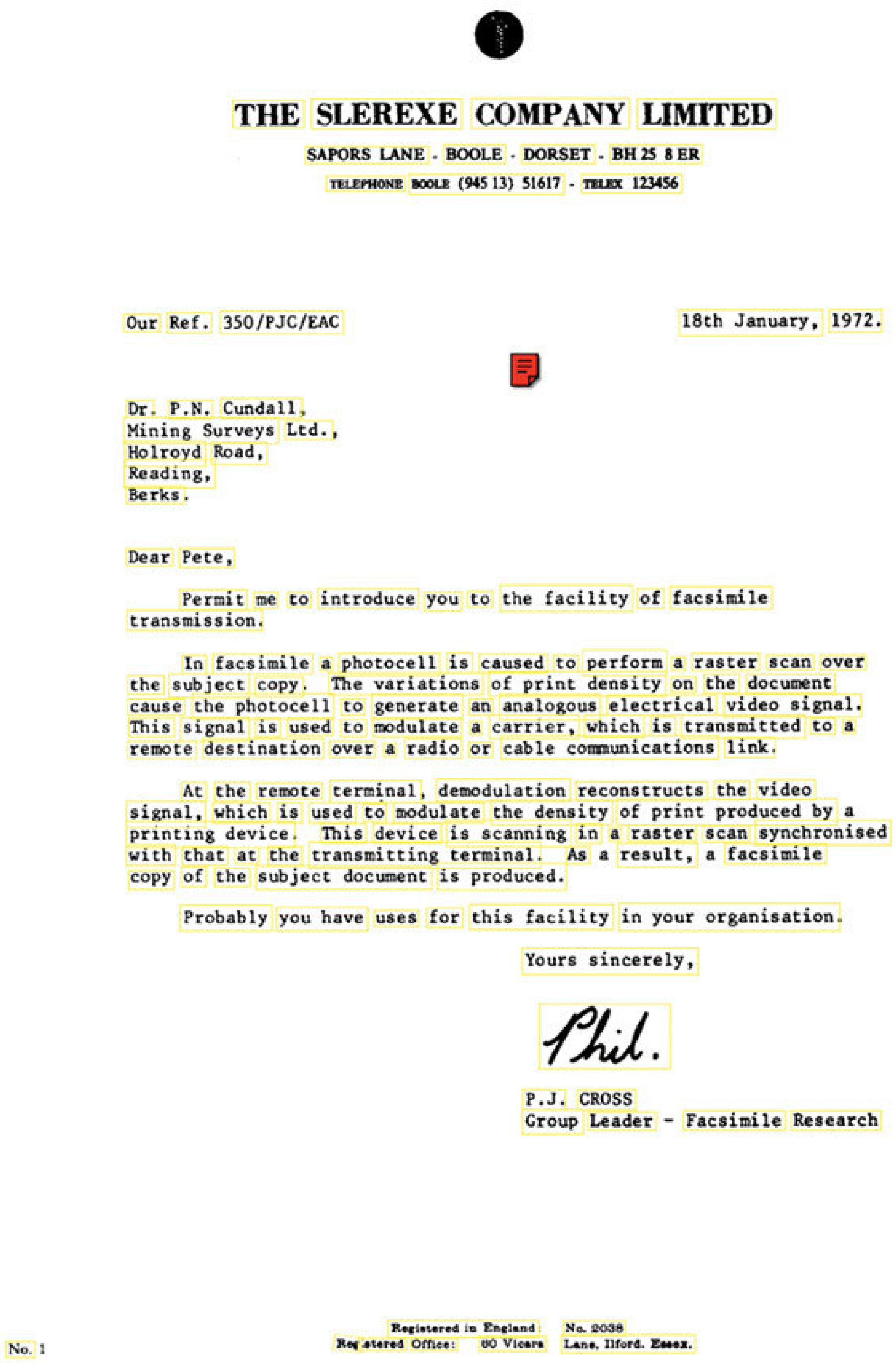
```
[([[447, 182], [557, 182], [557, 226], [447, 226]], 'THE',
0.999958842455258),

([[570, 180], [804, 180], [804, 228], [570, 228]],

'SLEREXE',



0.7436111647485029),

([[820, 179], [1066, 179], [1066, 227], [820, 227]],

'COMPANY',

0.9999328194688163),

([[1080, 180], [1296, 180], [1296, 228], [1080, 228]],

'LIMITED',

0.9986524611037886),

([[560, 254], [752, 254], [752, 282], [560, 282]],

'SAPORS LANE',

0.7222110685501141),
([[776, 252], [872, 252], [872, 282], [776, 282]],

‘BOOLE’,

0.9236137994214395),

([[898, 252], [1012, 252], [1012, 282], [898, 282]],

‘DORSET’,

0.9998741940557054),
```

Truncate the output for documentation limitations.

Creating a bounding box in the image using the preceding coordinates:

```
def draw_boxes(image, bounds, color='yellow', width=2):

  draw = ImageDraw.Draw(image)

  for bound in bounds:

    p0, p1, p2, p3 = bound[0]

    draw.line([*p0, *p1, *p2, *p3, *p0], fill=color, width=width)

  return image
```

```
draw_boxes(images[0], bounds)
```



THE SLEREXE COMPANY LIMITED

SAPORS LANE - BOOLE - DORSET - BH 25 8 ER

TELEPHONE BOOLE (945 13) 51617 - TELEX 123456

Our Ref. 350/PJC/EAC

18th January, 1972.

Dr. P.N. Cundall,
Mining Surveys Ltd.,
Holroyd Road,
Reading,
Berks.

Dear Pete,

Permit me to introduce you to the facility of facsimile transmission.

In facsimile a photocell is caused to perform a raster scan over the subject copy. The variations of print density on the document cause the photocell to generate an analogous electrical video signal. This signal is used to modulate a carrier, which is transmitted to a remote destination over a radio or cable communications link.

At the remote terminal, demodulation reconstructs the video signal, which is used to modulate the density of print produced by a printing device. This device is scanning in a raster scan synchronised with that at the transmitting terminal. As a result, a facsimile copy of the subject document is produced.

Probably you have uses for this facility in your organisation.

Yours sincerely,

Phil.

P.J. CROSS
Group Leader - Facsimile Research

Registered in England: No. 2038
Registered Office: 60 Vicars Lane, Ilford. Essex.

No. 1

Figure Bounding box against detected text

Extracting text from the tensor:

```
text=''

for i in range(len(bounds)):

  text = text + bounds[i][1] + '\n'

print(text)
```

Output:

```
1  THE

SLEREXE

COMPANY

LIMITED

SAPORS LANE

BOOLE

DORSET
```

BH 25 8 ER

TELEPHONE

BOOLE (945 13) 51617

TELEX 123456

Our

Ref .

350 /PJC /EAC

18th January,

1972 .

Dr _

P.N,

Cundall .

Mining Surveys

Ltd.

Holroyd

Road ,

Reading,

Berks

Dear

Pete,

Permit

me

to

introduce

you

to

the facility

of

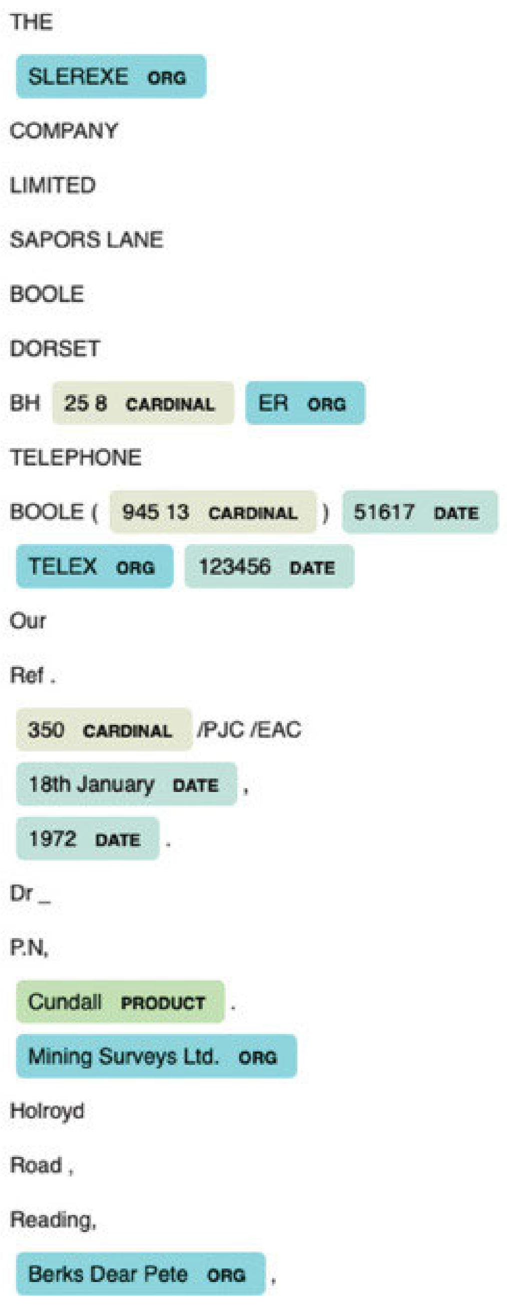
```
facsimi le

transmission _
```

Truncate the output for documentation limitations.

As you can see in the output, all the words are there, but they are not in proper paragraphs, so we are going to use spacy to fix paragraphs and all the issues:

```
nlp = spacy.load('en_core_web_sm')

doc = nlp(text)

from spacy import displacy

displacy.render(nlp(doc.text), style='ent', jupyter=True)
```

Output:



```
THE
SLEREXE ORG
COMPANY
LIMITED
SAPORS LANE
BOOLE
DORSET
BH 25 8 CARDINAL ER ORG
TELEPHONE
BOOLE ( 945 13 CARDINAL ) 51617 DATE
TELEX ORG 123456 DATE
Our
Ref .
350 CARDINAL /PJC /EAC
18th January DATE ,
1972 DATE .
Dr _
P.N,
Cundall PRODUCT .
Mining Surveys Ltd. ORG
Holroyd
Road ,
Reading,
Berks Dear Pete ORG ,
```

Figure Console output

TrOCR

TrOCR is a cutting-edge research project that aims to develop a robust OCR system that can recognize text from natural images captured by smartphones without relying on additional sensors or specialized hardware. Traditional OCR systems have limitations in recognizing text from low-quality images with uneven illumination, perspective distortion, and complex backgrounds. TrOCR addresses these challenges by incorporating advanced computer vision and machine learning techniques to improve the accuracy and speed of text recognition. The project has the potential to revolutionize the way we interact with text, making it easier to extract and analyze text from images in various applications, such as augmented reality, image search, and mobile translation.

[TrOCR demo]

Text detection in images with TrOCR can be done using the following steps:

Installing important libraries:

```
!pip install -q transformers
```

Loading image:

```
import requests

from PIL import Image

url = "https://fki.tic.heia-fr.ch/static/img/a01-122-02.jpg"

image = Image.open(requests.get(url, stream=True).raw).convert("RGB")

image
```

The output will look as follows:

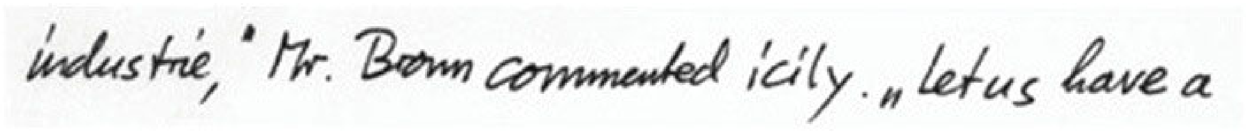

industrie, " Mr. Brown commented icily. " let us have a

Figure Console output

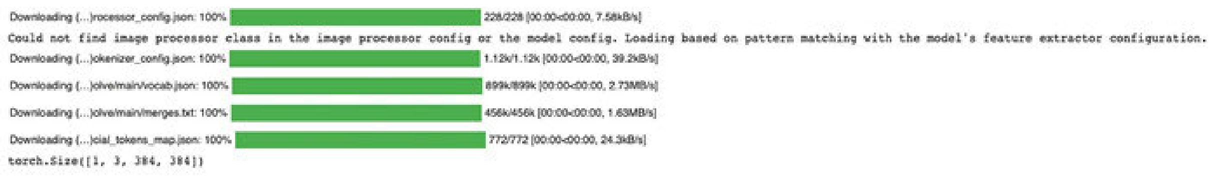
Now, the image is prepared for the model using TrOCR processor. Calling the processor is equivalent to calling the feature extractor:

from transformers import TrOCRProcessor

processor = TrOCRProcessor.from_pretrained("microsoft/trocr-base-handwritten")

pixel_values = processor(image, return_tensors="pt").pixel_values

print(pixel_values.shape)



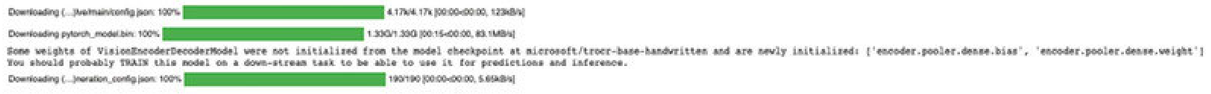
```
Downloading (…)rocessor_config.json: 100%  228/228 [00:00<00:00, 7.58kB/s]
Could not find image processor class in the image processor config or the model config. Loading based on pattern matching with the model's feature extractor configuration.
Downloading (…)okenizer_config.json: 100%  1.12k/1.12k [00:00<00:00, 39.2kB/s]
Downloading (…)olve/main/vocab.json: 100%  899k/899k [00:00<00:00, 2.73MB/s]
Downloading (…)olve/main/merges.txt: 100%  456k/456k [00:00<00:00, 1.63MB/s]
Downloading (…)cial_tokens_map.json: 100%  772/772 [00:00<00:00, 24.3kB/s]
torch.Size([1, 3, 384, 384])
```

Figure Console output

Load the model from the hub: https://huggingface.co/models?other=trocr

from transformers import VisionEncoderDecoderModel

model = VisionEncoderDecoderModel.from_pretrained("microsoft/trocr-base-handwritten")



```
Downloading (…)ve/main/config.json: 100%  4.17k/4.17k [00:00<00:00, 123kB/s]
Downloading pytorch_model.bin: 100%  1.33G/1.33G [00:15<00:00, 83.1MB/s]
Some weights of VisionEncoderDecoderModel were not initialized from the model checkpoint at microsoft/trocr-base-handwritten and are newly initialized: ['encoder.pooler.dense.bias', 'encoder.pooler.dense.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Downloading (…)neration_config.json: 100%  190/190 [00:00<00:00, 5.65kB/s]
```

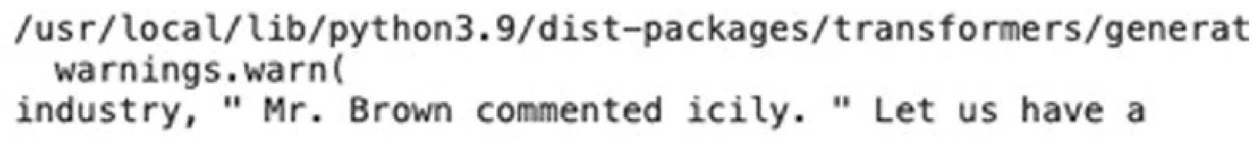
Figure Console Output

Finally, the text is generated:

generated_ids = model.generate(pixel_values)

generated_text = processor.batch_decode(generated_ids, skip_special_tokens=True)[0]

print(generated_text)



```
/usr/local/lib/python3.9/dist-packages/transformers/generat
  warnings.warn(
industry, " Mr. Brown commented icily. " Let us have a
```

Figure Console output

Conclusion

In conclusion, OCR has come a long way in recent years with the development of various open-source libraries and frameworks that make text recognition from images more accessible and accurate. In this chapter, we explored four popular OCR implementations in Python: Tesseract OCR, keras-ocr, EasyOCR, and TrOCR.

Tesseract OCR, the most widely used OCR library, is based on Google's OCR engine and provides a reliable solution for recognizing text in scanned documents. Keras-ocr, built on top of TensorFlow and keras, offers a deep learning-based approach for recognizing text in images and can handle more complex scenarios.

EasyOCR is another popular OCR library that supports more than 70 languages and provides a simple interface for recognizing text in natural images. Finally, TrOCR is a promising research project that aims to develop a robust OCR system for recognizing text in images captured by smartphones.

Regardless of the OCR library or framework chosen, it is important to carefully consider the use case and requirements to select the most appropriate solution. Furthermore, it is important to fine-tune the OCR system for optimal performance by preprocessing the images, selecting appropriate recognition algorithms, and training the OCR model with sufficient and relevant data.

In the next chapter, we will cover the topic of object detection. Classifying an image is about telling what object is in the image. Taking this one step further, object detection is about telling where exactly the object is located in the image.

Questions

What does OCR stand for?

Optical Character Recognition

Object Character Recognition

Operating Character Recognition

Order Character Recognition

All of the above

Which of the following is not an application of OCR technology?

Scanning and digitizing documents

Translating text from one language to another

Recognizing handwritten text

Extracting data from images

None of the above

What is the primary function of OCR technology?

To identify and recognize text in images and documents

To edit and modify text in digital documents

To compress and optimize digital images

To create 3D models from 2D images

Which of the following types of documents can be processed by OCR technology?

PDF files

Scanned images

Handwritten documents

All of the above

Which of the following factors can affect the accuracy of OCR results?

Image resolution

Font type and size

Language of the text

All of the above

Answers

a

b

a

d

d

CHAPTER 6

Object Detection

“You can’t defend. You can’t prevent. The only thing you can do is detect and respond.”

— Bruce Schneier

By now, we have understood that machines are now capable of classifying images using intelligence infused through convolutional neural networks. However, just classifying images does not solve any real-world problems. An image can contain both a dog and a cat, and it would be expected from a machine to tell exactly where in the frame these two objects lie. Also, detecting multiple objects in a single image is of utmost importance. These challenges can be tackled by object detection algorithms. In this chapter, we will discuss various such algorithms in detail. Also, we will see how these trained and ready-to-use algorithms can be applied in projects, without the need to retrain the models from scratch.

Structure

In this chapter, we will cover the following topics:

Object localization and object detection

Object detection algorithms and their comparison

Single Shot Detector Python implementation

You Only Look Once Python implementation

Objectives

After studying this chapter, you should be able to understand the concept of object localization and object detection in an image, and how they add more value to simple image classification tasks. You should also be able to implement various image detection algorithms in Python.

Object localization and detection

The most common use case of computer vision is image classification, which is simply about classifying an image with a class. It only tells whether there is a cat or dog in an image. It has nothing to do with the location of the cat or dog in the image.

Object localization is about finding a relevant object in the image and creating a bounding box against the object.

Object detection is a cumulative result of image classification and object localization. This means not only an object location is detected, but the class of the object is also identified.

Figure 6.1 depicts that in the first image a cat is identified, and the image is classified as ´CAT´. But it does not locate the exact location of the cat. In the second image, a bounding box is created around the cat and labelled as This feature where an image is not only classified but also localized is the object detection mechanism. Refer to the following figure:

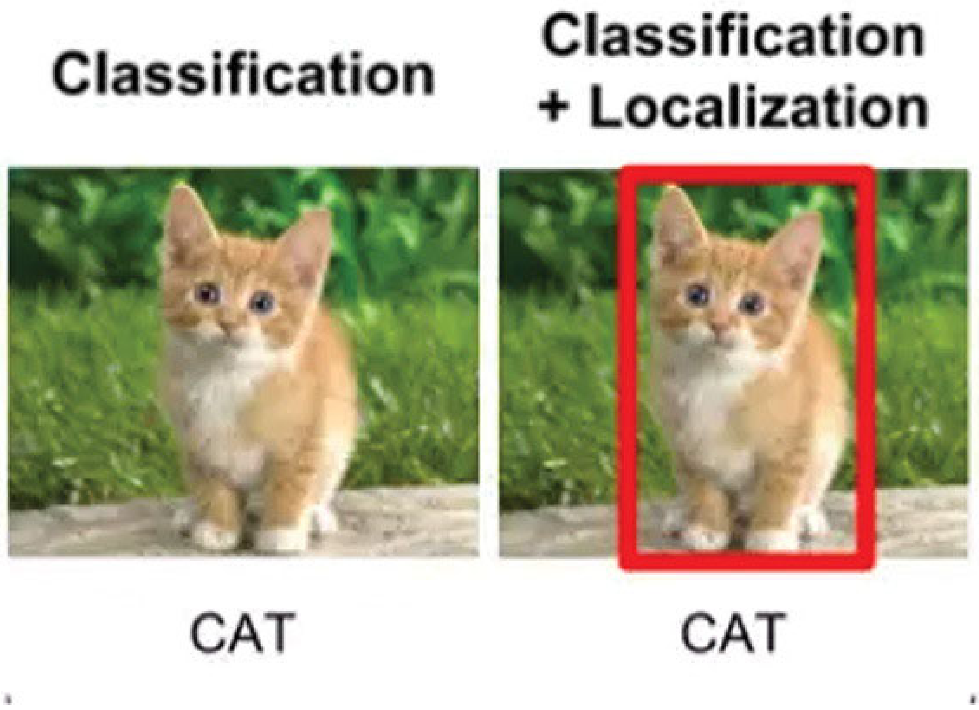


Figure 6.1: Various computer vision tasks

Object detection algorithms and their comparison

There are numerous algorithms that can perform object detection and are readily available to be used in real-time projects. Some of them are as follows:

Region-based Convolutional Neural Network

Fast RCNN

Faster RCNN

Single Shot Detector

You Only Look Once

Figure 6.2 depicts a comparison of all the noted object detection algorithms in terms of accuracy and speed. It can be seen that there is a clear tradeoff between accuracy and speed, which can be a deciding factor when choosing an algorithm for your project based on the specific business needs. Refer to the following figure:

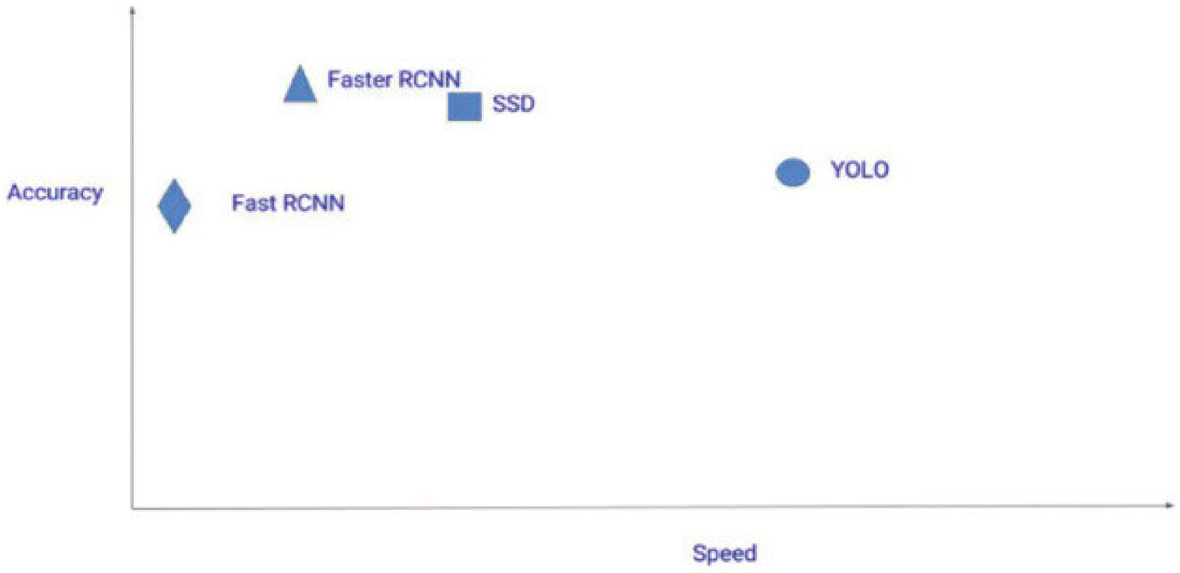


Figure 6.2: Comparison of object detection algorithms

Single shot detector Python implementation

Follow these steps for successful Single Shot Detector Python implementation:

This step installs two Python packages:

scikit-image version 0.19.1, which is a collection of algorithms for image processing

which is used for creating interactive Graphical User Interfaces for Jupyter notebooks

```
!pip install scikit-image==0.19.1
```

```
!pip install ipywidgets --trusted-host pypi.org --trusted-host pypi.python.org --trusted-host=files.pythonhosted.org
```

This step imports the torch package, which is the base package for PyTorch. It also disables a check in the torch.hub module that usually prevents the use of repositories that have been forked. This is done by overriding an internal function called _validate_not_a_forked_repo to always return

```
import torch
```

```
torch.hub._validate_not_a_forked_repo=lambda a,b,c: True
```

This step lists all the available models in the PyTorch Hub provided by NVIDIA under the DeepLearningExamples repository. PyTorch Hub is a pre-trained model repository.

# List of available models in PyTorch Hub from Nvidia/DeepLearningExamples

torch.hub.list('NVIDIA/DeepLearningExamples:torchhub')

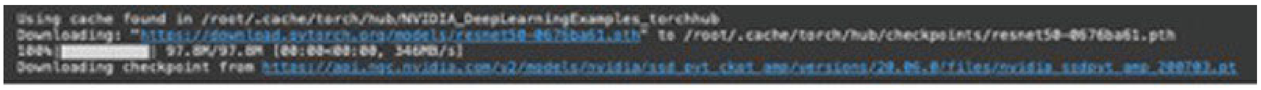
This step loads a pretrained SSD model that is capable of object detection. It specifies the precision of the model to be fp32 (32-bit floating point). The model is loaded from NVIDIA's DeepLearningExamples repository in the PyTorch Hub:

# load SSD model pretrained on COCO from Torch Hub

precision = 'fp32'

ssd300 = torch.hub.load('NVIDIA/DeepLearningExamples:torchhub', 'nvidia_ssd', model_math=precision);

Figure 6.3 depicts successful model loading:



```
Using cache found in /root/.cache/torch/hub/NVIDIA_DeepLearningExamples_torchhub
Downloading: "https://download.pytorch.org/models/resnet50-0676ba61.pth" to /root/.cache/torch/hub/checkpoints/resnet50-0676ba61.pth
100%|██████████| 97.8M/97.8M [00:00<00:00, 346MB/s]
Downloading checkpoint from https://api.ngc.nvidia.com/v2/models/nvidia/ssd_pyt_ckpt_amp/versions/20.06.0/files/nvidia_ssdpyt_amp_200703.pt
```

Figure 6.3: Model loading

This step specifies a list of sample image URLs from the COCO validation dataset that will be used for inference by the model:

```
# Inference

# Sample images from the COCO validation set

uris = [

]
```

For conveniently formatting the input and output of the model, a set of utility methods are loaded from the NVIDIA's DeepLearningExamples repository. These utilities will help in processing the images and the output:

```
# For convenient and comprehensive formatting of input and output of the model, load a set of utility methods.

utils = torch.hub.load('NVIDIA/DeepLearningExamples:torchhub', 'nvidia_ssd_processing_utils')
```

This step processes the images from the specified URLs to make them compatible with the network input requirements. The images are formatted and then converted into tensors that are multi-dimensional arrays containing elements of a single data type:

```
# Format images to comply with the network input

inputs = [utils.prepare_input(uri) for uri in uris]

tensor = utils.prepare_tensor(inputs, False)
```

As the SSD model was trained on the COCO dataset, this step loads a dictionary that maps class IDs to their respective object names. This is essential for understanding the output of the model:

```
# The model was trained on COCO dataset, which we need to access in order
to

# translate class IDs into object names.



classes_to_labels = utils.get_coco_object_dictionary()
```

This step sets the model to evaluation mode and specifies that the computation should be performed on a GPU It then runs object detection on the pre-processed images:

```
# Next, we run object detection

model = ssd300.eval().to("cuda")

detections_batch = model(tensor)
```

The raw output from the SSD model for each input image contains a large number of bounding boxes. This step filters the output to only include reasonable detections (with confidence over 40%) and formats the results:

```
# By default, raw output from SSD network per input image contains 8732
boxes with

# localization and class probability distribution.

# Let's filter this output to only get reasonable detections (confidence>40%)
in a more comprehensive format.

results_per_input = utils.decode_results(detections_batch)

best_results_per_input = [utils.pick_best(results, 0.40) for results in
results_per_input]
```

This part of the code is a function definition called plot_results that takes in the best results per input and plots the images along with the predicted bounding boxes and their confidence scores. The function uses a plotting library for the Python programming language:

```
from matplotlib import pyplot as plt

import matplotlib.patches as patches

# The utility plots the images and predicted bounding boxes (with confidence
scores).
```

```
def plot_results(best_results):

    for image_idx in range(len(best_results)):

        fig, ax = plt.subplots(1)

# Show original, denormalized image...

        image = inputs[image_idx] / 2 + 0.5

        ax.imshow(image)

# ...with detections

        bboxes, classes, confidences = best_results[image_idx]

        for idx in range(len(bboxes)):

            left, bot, right, top = bboxes[idx]

            x, y, w, h = [val * 300 for val in [left, bot, right - left, top - bot]]

            rect = patches.Rectangle((x, y), w, h, linewidth=1, edgecolor='r',
facecolor='none')

            ax.add_patch(rect)
            ax.text(x, y, "{} {:.0f}%".format(classes_to_labels[classes[idx] - 1],
confidences[idx]*100), bbox=dict(facecolor='white', alpha=0.5))

    plt.show()
```

Finally, this function is called with the best results per input to visualize the object detection results. The images are displayed with bounding boxes around detected objects and labels that indicate what object has been detected, along with the confidence in percentage:

```
# Visualize results without Torch-TensorRT

plot_results(best_results_per_input)
```

Figures 6.4 , 6.5 and 6.6 depict sample object detection results using SSD model:

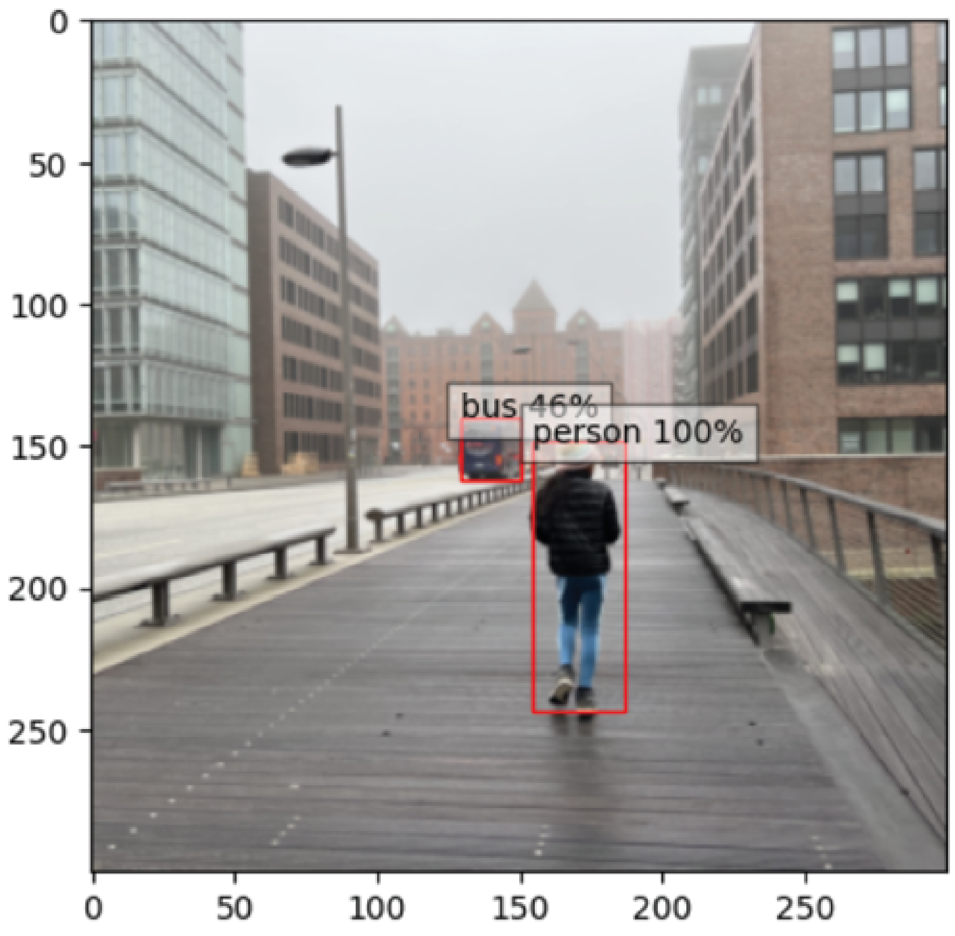


Figure Sample result

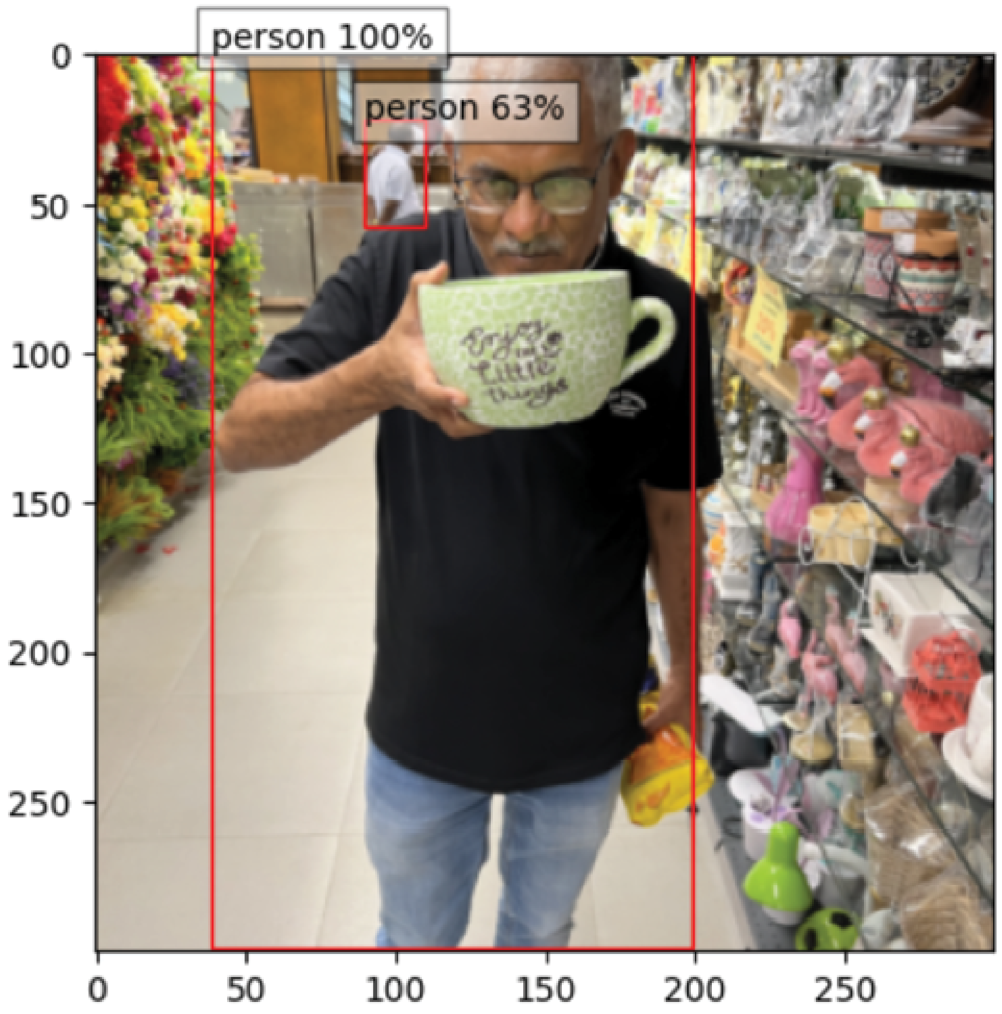


Figure 6.5: Sample result

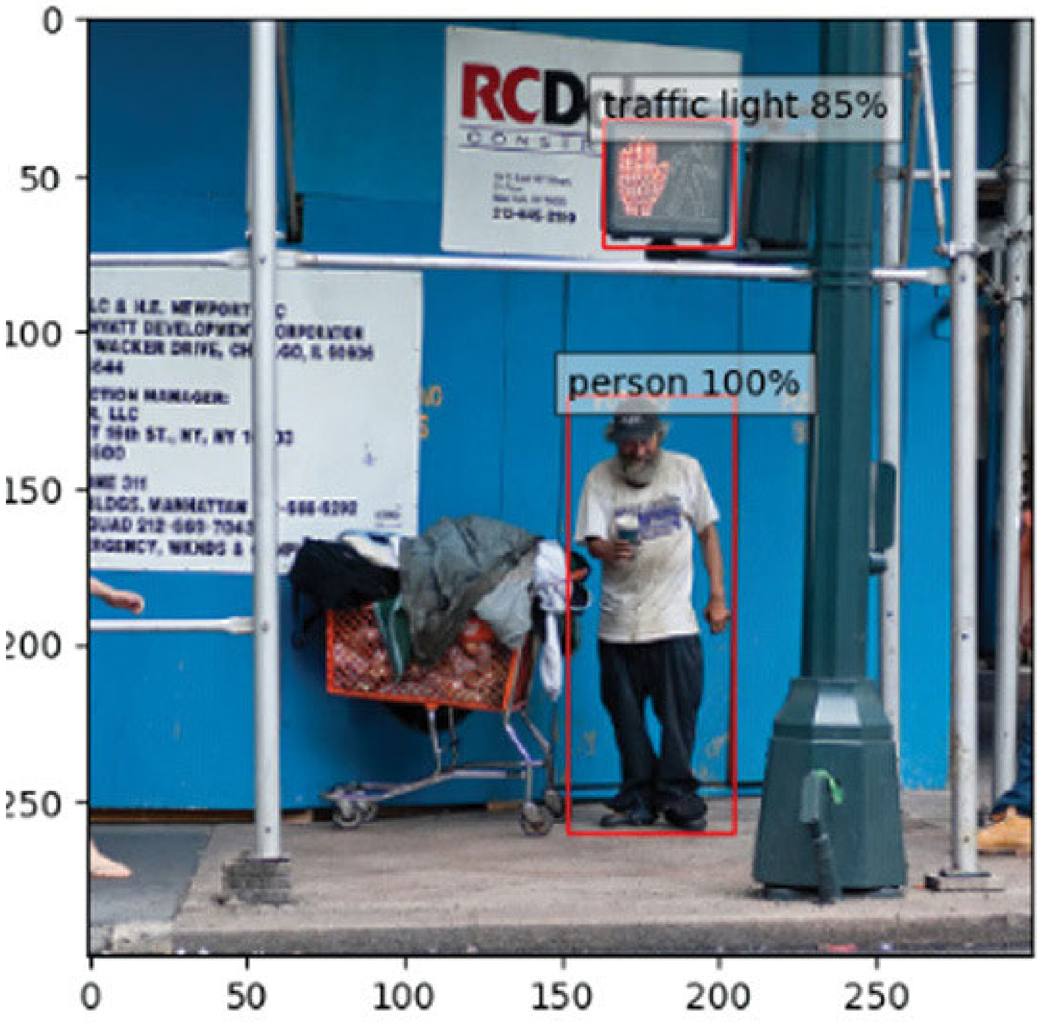


Figure 6.6: Sample result

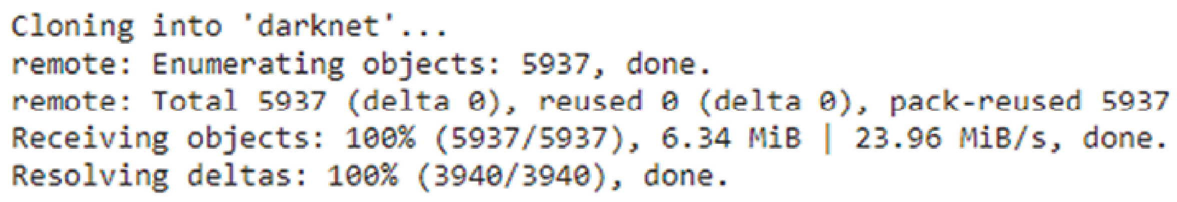
YOLO v3 Python implementation

Follow these steps for a successful YOLO v3 Python implementation:

Git clone the project codes. This will copy all the required files and folders from GitHub repo to the local machine:

!git clone https://github.com/pjreddie/darknet

This will clone the repository on your local machine.



```
Cloning into 'darknet'...
remote: Enumerating objects: 5937, done.
remote: Total 5937 (delta 0), reused 0 (delta 0), pack-reused 5937
Receiving objects: 100% (5937/5937), 6.34 MiB | 23.96 MiB/s, done.
Resolving deltas: 100% (3940/3940), done.
```

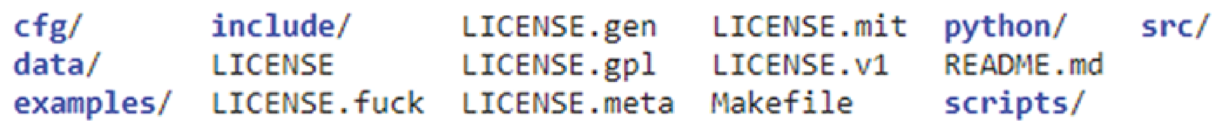
Figure 6.7: Git clone command output

cd into the darknet folder and look at all the downloaded files and folders:

cd darknet

ls

Folder contents depicted in Figure

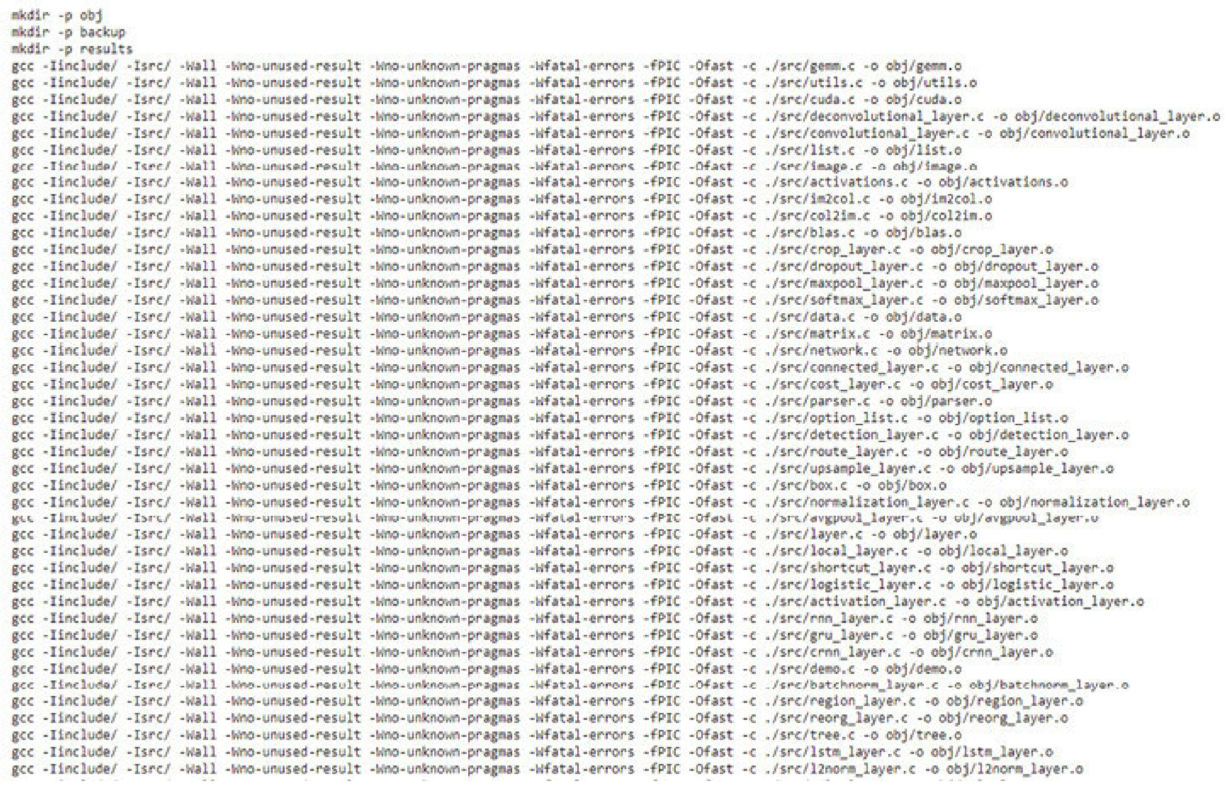


```
cfg/        include/      LICENSE.gen   LICENSE.mit  python/    src/
data/       LICENSE       LICENSE.gpl   LICENSE.v1   README.md
examples/   LICENSE.fuck  LICENSE.meta  Makefile     scripts/
```

Figure 6.8: Darknet folder structure

Run the make command to build the project locally:

!make



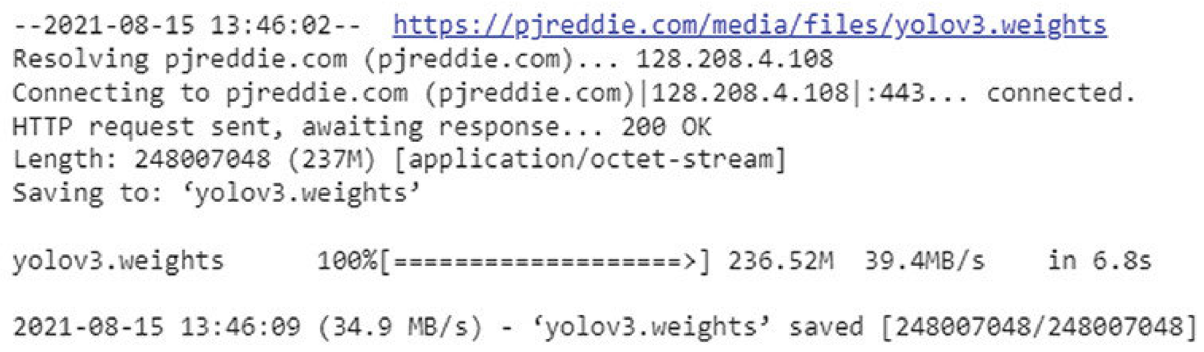
```
mkdir -p obj
mkdir -p backup
mkdir -p results
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/gemm.c -o obj/gemm.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/utils.c -o obj/utils.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/cuda.c -o obj/cuda.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/deconvolutional_layer.c -o obj/deconvolutional_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/convolutional_layer.c -o obj/convolutional_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/list.c -o obj/list.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/image.c -o obj/image.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/activations.c -o obj/activations.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/im2col.c -o obj/im2col.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/col2im.c -o obj/col2im.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/blas.c -o obj/blas.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/crop_layer.c -o obj/crop_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/dropout_layer.c -o obj/dropout_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/maxpool_layer.c -o obj/maxpool_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/softmax_layer.c -o obj/softmax_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/data.c -o obj/data.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/matrix.c -o obj/matrix.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/network.c -o obj/network.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/connected_layer.c -o obj/connected_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/cost_layer.c -o obj/cost_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/parser.c -o obj/parser.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/option_list.c -o obj/option_list.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/detection_layer.c -o obj/detection_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/route_layer.c -o obj/route_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/upsample_layer.c -o obj/upsample_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/box.c -o obj/box.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/normalization_layer.c -o obj/normalization_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/avgpool_layer.c -o obj/avgpool_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/layer.c -o obj/layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/local_layer.c -o obj/local_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/shortcut_layer.c -o obj/shortcut_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/logistic_layer.c -o obj/logistic_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/activation_layer.c -o obj/activation_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/rnn_layer.c -o obj/rnn_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/gru_layer.c -o obj/gru_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/crnn_layer.c -o obj/crnn_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/demo.c -o obj/demo.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/batchnorm_layer.c -o obj/batchnorm_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/region_layer.c -o obj/region_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/reorg_layer.c -o obj/reorg_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/tree.c -o obj/tree.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/lstm_layer.c -o obj/lstm_layer.o
gcc -Iinclude/ -Isrc/ -Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC -Ofast -c ./src/l2norm_layer.c -o obj/l2norm_layer.o
```

Figure 6.9: Make command output

Download yolov3.weights using the wget command:

!wget https://pjreddie.com/media/files/yolov3.weights

Download yolo weights as depicted in Figure



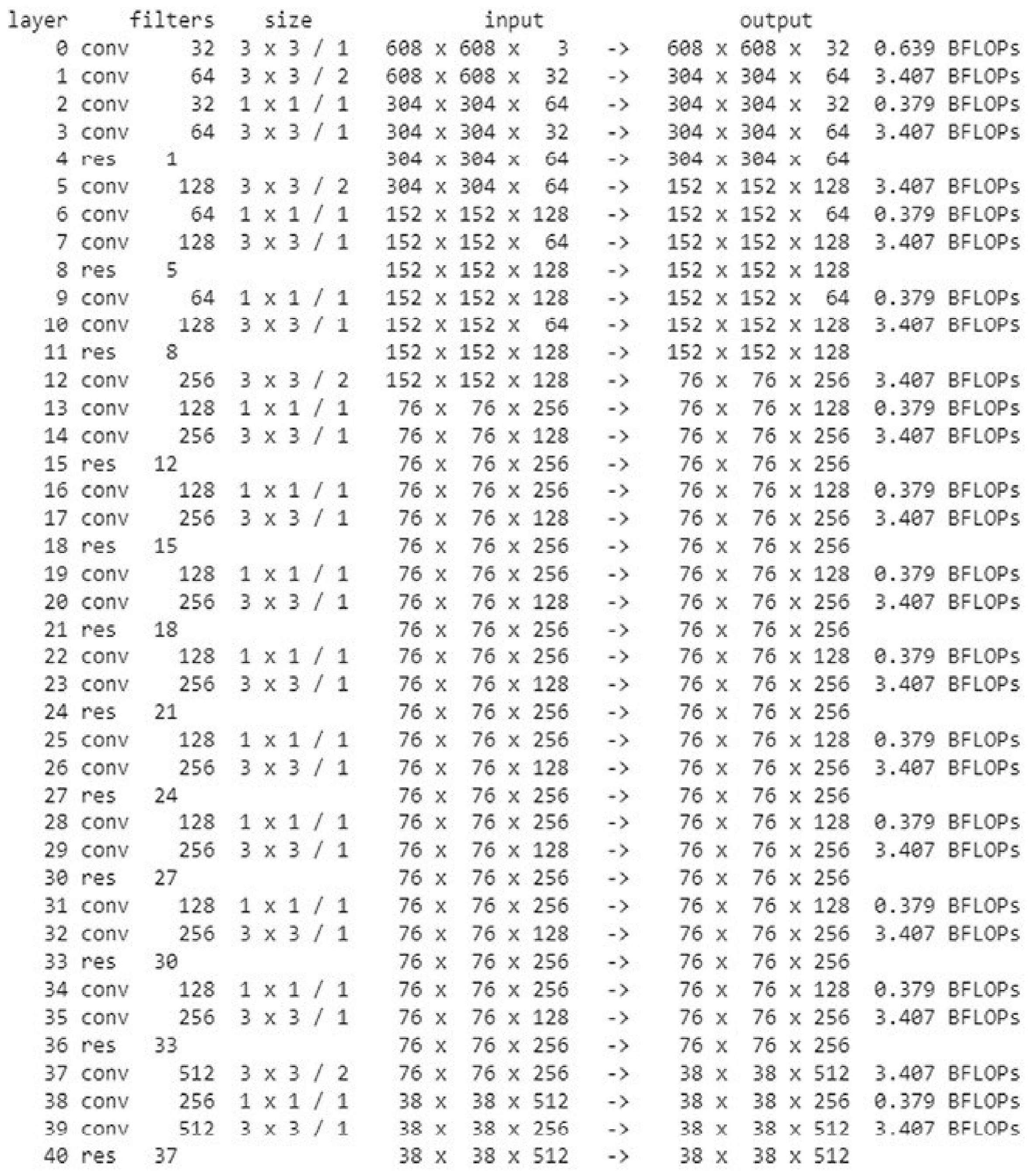
```
--2021-08-15 13:46:02--  https://pjreddie.com/media/files/yolov3.weights
Resolving pjreddie.com (pjreddie.com)... 128.208.4.108
Connecting to pjreddie.com (pjreddie.com)|128.208.4.108|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 248007048 (237M) [application/octet-stream]
Saving to: ‘yolov3.weights’

yolov3.weights      100%[===================>] 236.52M  39.4MB/s    in 6.8s

2021-08-15 13:46:09 (34.9 MB/s) - ‘yolov3.weights’ saved [248007048/248007048]
```

Figure 6.10: wget yolo weights command output

Run the detect command to perform the predictions:

```
!./darknet detect cfg/yolov3.cfg yolov3.weights data/dog.jpg
```

Output of the detect command is depicted in Figure



```
layer     filters    size              input                output
    0 conv     32  3 x 3 / 1   608 x 608 x   3   ->   608 x 608 x  32  0.639 BFLOPs
    1 conv     64  3 x 3 / 2   608 x 608 x  32   ->   304 x 304 x  64  3.407 BFLOPs
    2 conv     32  1 x 1 / 1   304 x 304 x  64   ->   304 x 304 x  32  0.379 BFLOPs
    3 conv     64  3 x 3 / 1   304 x 304 x  32   ->   304 x 304 x  64  3.407 BFLOPs
    4 res    1                 304 x 304 x  64   ->   304 x 304 x  64
    5 conv    128  3 x 3 / 2   304 x 304 x  64   ->   152 x 152 x 128  3.407 BFLOPs
    6 conv     64  1 x 1 / 1   152 x 152 x 128   ->   152 x 152 x  64  0.379 BFLOPs
    7 conv    128  3 x 3 / 1   152 x 152 x  64   ->   152 x 152 x 128  3.407 BFLOPs
    8 res    5                 152 x 152 x 128   ->   152 x 152 x 128
    9 conv     64  1 x 1 / 1   152 x 152 x 128   ->   152 x 152 x  64  0.379 BFLOPs
   10 conv    128  3 x 3 / 1   152 x 152 x  64   ->   152 x 152 x 128  3.407 BFLOPs
   11 res    8                 152 x 152 x 128   ->   152 x 152 x 128
   12 conv    256  3 x 3 / 2   152 x 152 x 128   ->    76 x  76 x 256  3.407 BFLOPs
   13 conv    128  1 x 1 / 1    76 x  76 x 256   ->    76 x  76 x 128  0.379 BFLOPs
   14 conv    256  3 x 3 / 1    76 x  76 x 128   ->    76 x  76 x 256  3.407 BFLOPs
   15 res   12                  76 x  76 x 256   ->    76 x  76 x 256
   16 conv    128  1 x 1 / 1    76 x  76 x 256   ->    76 x  76 x 128  0.379 BFLOPs
   17 conv    256  3 x 3 / 1    76 x  76 x 128   ->    76 x  76 x 256  3.407 BFLOPs
   18 res   15                  76 x  76 x 256   ->    76 x  76 x 256
   19 conv    128  1 x 1 / 1    76 x  76 x 256   ->    76 x  76 x 128  0.379 BFLOPs
   20 conv    256  3 x 3 / 1    76 x  76 x 128   ->    76 x  76 x 256  3.407 BFLOPs
   21 res   18                  76 x  76 x 256   ->    76 x  76 x 256
   22 conv    128  1 x 1 / 1    76 x  76 x 256   ->    76 x  76 x 128  0.379 BFLOPs
   23 conv    256  3 x 3 / 1    76 x  76 x 128   ->    76 x  76 x 256  3.407 BFLOPs
   24 res   21                  76 x  76 x 256   ->    76 x  76 x 256
   25 conv    128  1 x 1 / 1    76 x  76 x 256   ->    76 x  76 x 128  0.379 BFLOPs
   26 conv    256  3 x 3 / 1    76 x  76 x 128   ->    76 x  76 x 256  3.407 BFLOPs
   27 res   24                  76 x  76 x 256   ->    76 x  76 x 256
   28 conv    128  1 x 1 / 1    76 x  76 x 256   ->    76 x  76 x 128  0.379 BFLOPs
   29 conv    256  3 x 3 / 1    76 x  76 x 128   ->    76 x  76 x 256  3.407 BFLOPs
   30 res   27                  76 x  76 x 256   ->    76 x  76 x 256
   31 conv    128  1 x 1 / 1    76 x  76 x 256   ->    76 x  76 x 128  0.379 BFLOPs
   32 conv    256  3 x 3 / 1    76 x  76 x 128   ->    76 x  76 x 256  3.407 BFLOPs
   33 res   30                  76 x  76 x 256   ->    76 x  76 x 256
   34 conv    128  1 x 1 / 1    76 x  76 x 256   ->    76 x  76 x 128  0.379 BFLOPs
   35 conv    256  3 x 3 / 1    76 x  76 x 128   ->    76 x  76 x 256  3.407 BFLOPs
   36 res   33                  76 x  76 x 256   ->    76 x  76 x 256
   37 conv    512  3 x 3 / 2    76 x  76 x 256   ->    38 x  38 x 512  3.407 BFLOPs
   38 conv    256  1 x 1 / 1    38 x  38 x 512   ->    38 x  38 x 256  0.379 BFLOPs
   39 conv    512  3 x 3 / 1    38 x  38 x 256   ->    38 x  38 x 512  3.407 BFLOPs
   40 res   37                  38 x  38 x 512   ->    38 x  38 x 512
```

Figure 6.11: Predictions output

To view the objects detected within the sample image, use the Image module in Python. Image and the objects detected are displayed with bounding boxes, as shown in Figure

from PIL import Image

```
img=Image.open("predictions.jpg")

img
```

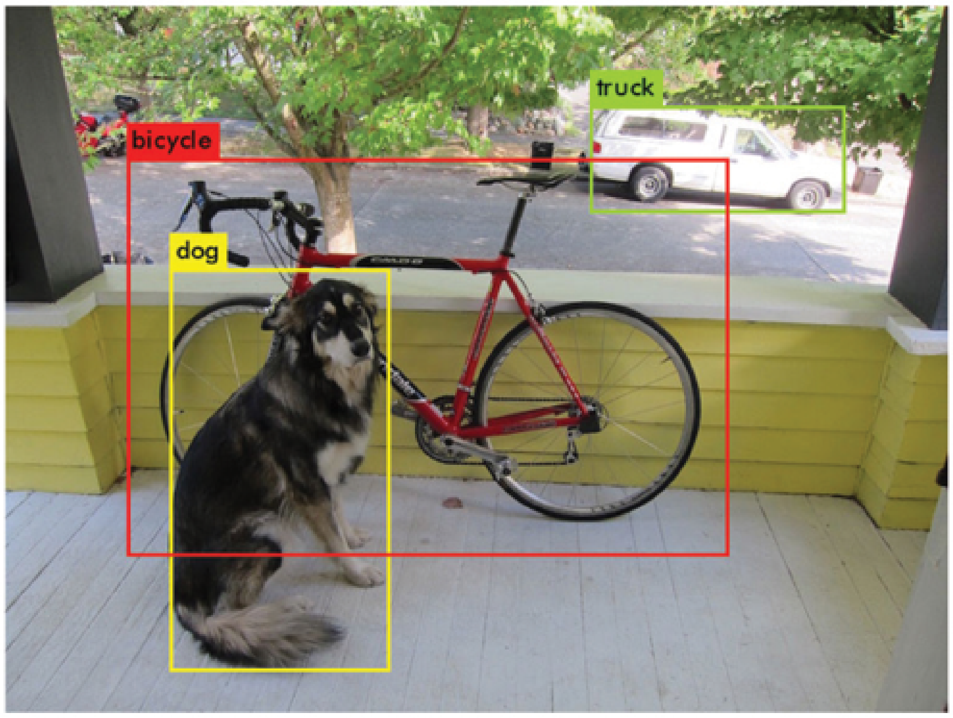


Figure Predictions with bounding boxes

Experiment 2

Animals of different sizes and shapes are also captured by the algorithm. Predictions are depicted in Figure

```
!./darknet detect cfg/yolov3.cfg yolov3.weights data/giraffe.jpg
```

```
from PIL import Image
```

```
img=Image.open("predictions.jpg")
```

```
img
```

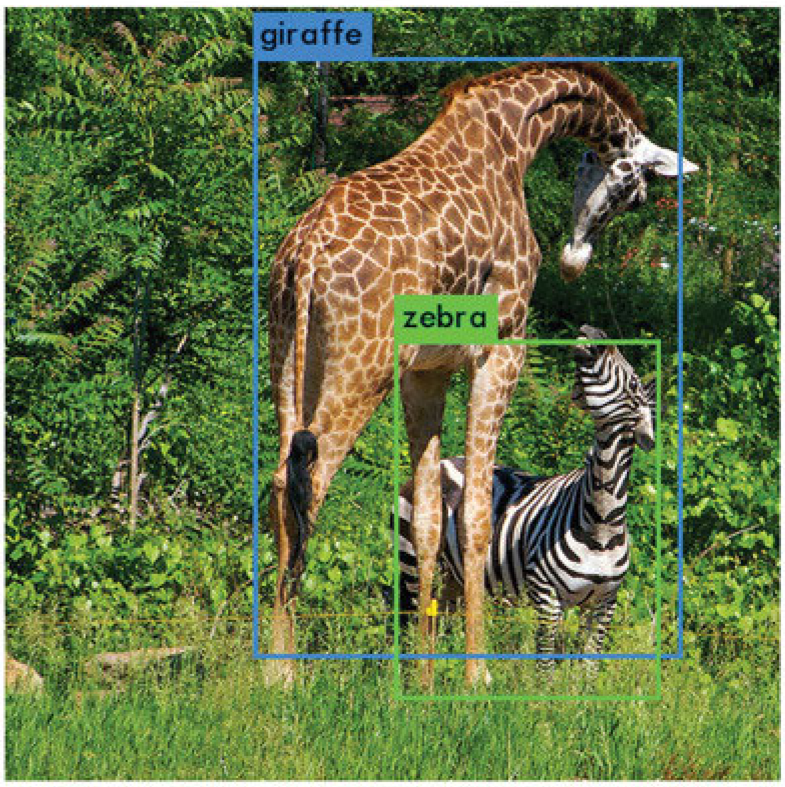


Figure 6.13: Predictions with bounding boxes

Experiment 3

People of different ages and sizes with various background objects are also captured by the algorithm. Predictions are depicted in Figure

```
!./darknet detect cfg/yolov3.cfg yolov3.weights
/content/gdrive/MyDrive/coffee.JPG
```

```
from PIL import Image
```

```
img=Image.open("predictions.jpg")
```

```
img
```

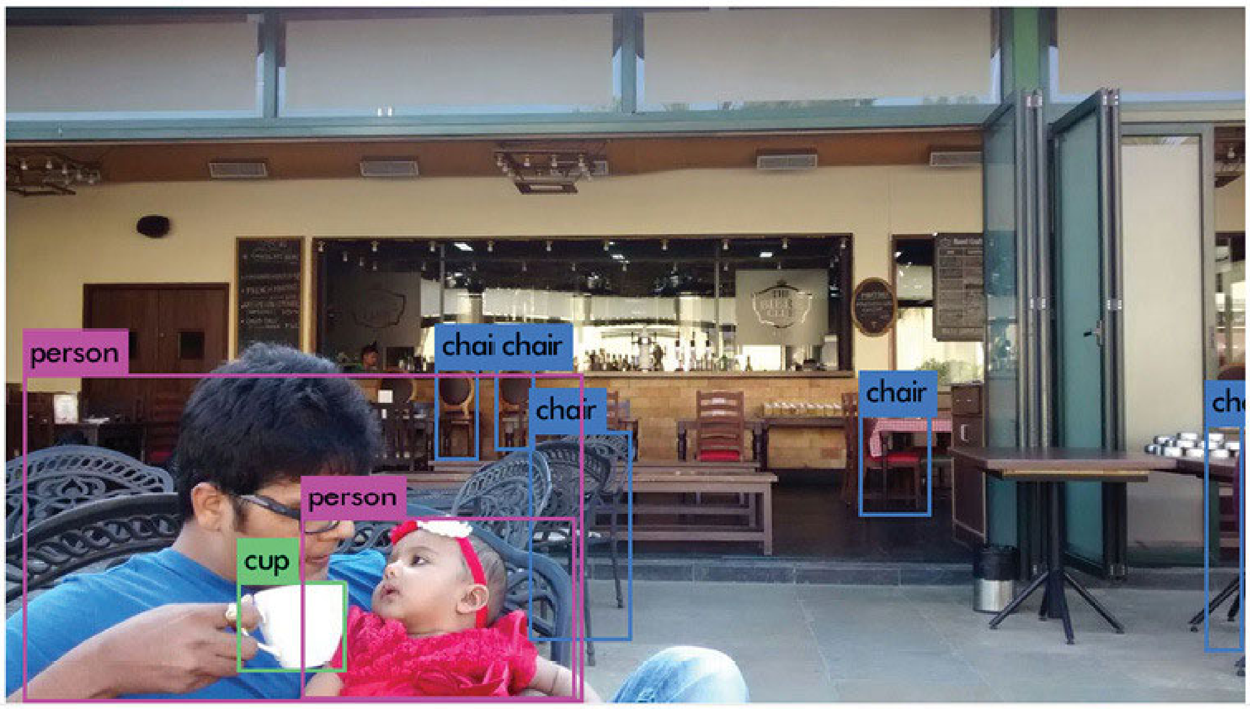


Figure 6.14: Predictions with bounding boxes

Experiment 4

People with different poses, one horizontal and another in vertical positions are also captured by the algorithm. Predictions are depicted in Figure

```
!./darknet detect cfg/yolov3.cfg yolov3.weights
/content/gdrive/MyDrive/yoga.JPG

from PIL import Image

img=Image.open("predictions.jpg")

img
```

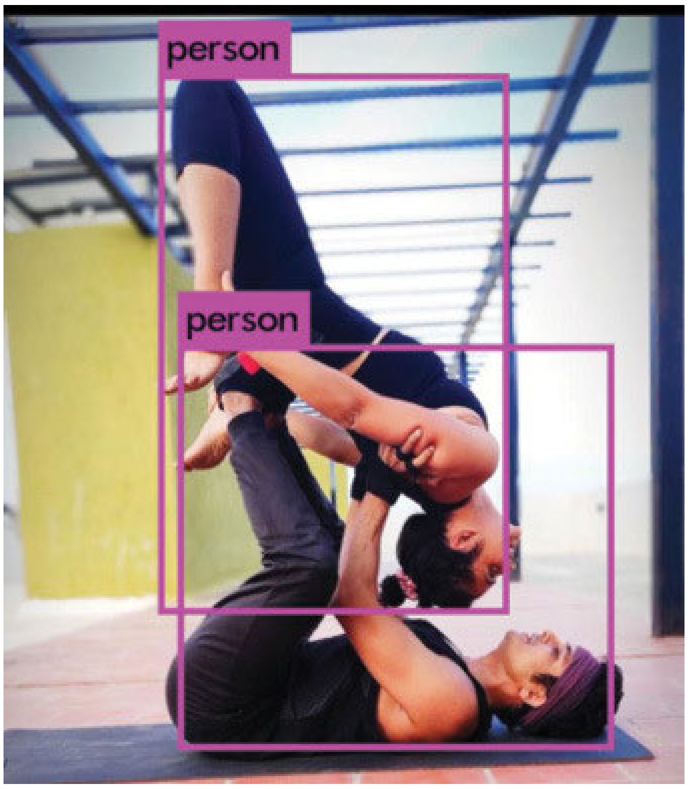


Figure 6.15: Predictions with bounding boxes

Experiment 5

Very small objects are also captured by the algorithm. Predictions are depicted in Figure

```
!./darknet detect cfg/yolov3.cfg yolov3.weights data/kite.jpg
```

```
from PIL import Image
```

```
img=Image.open("predictions.jpg")
```

```
img
```

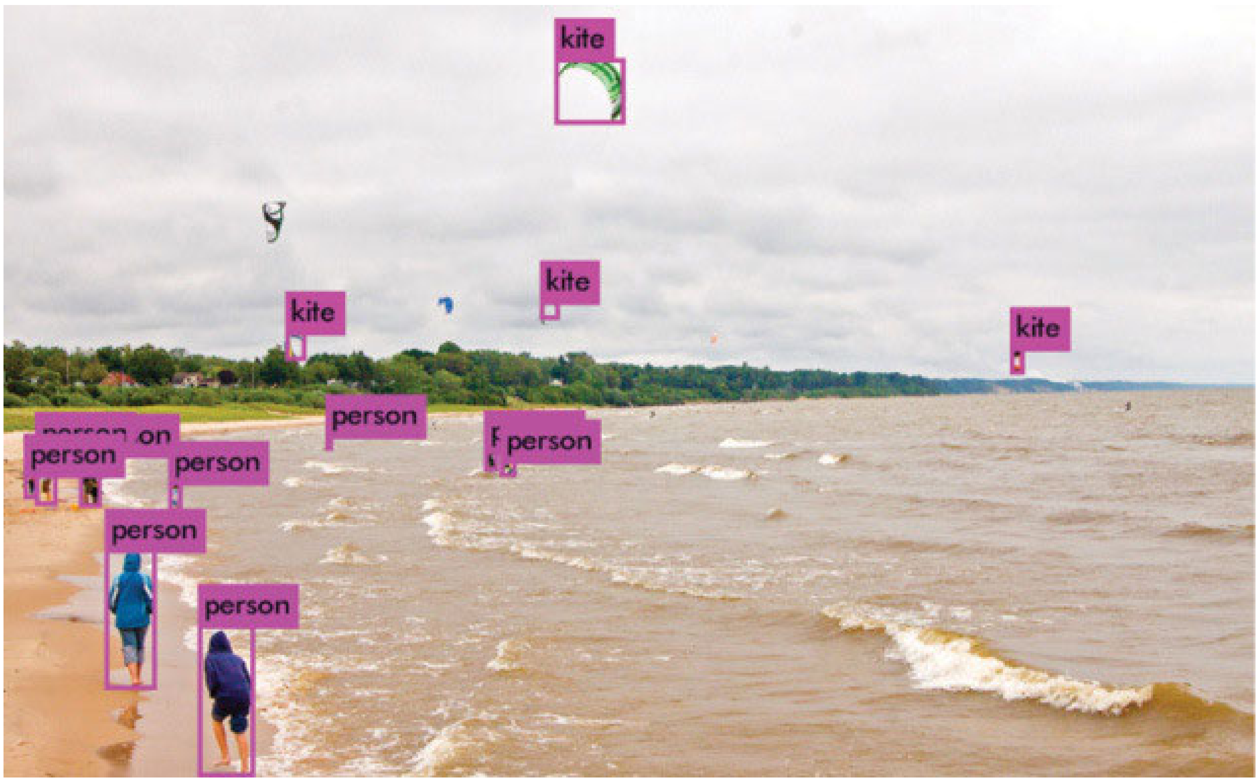


Figure Predictions with bounding boxes

Conclusion

In this chapter, we first understood the difference between image classification, object localization, and object detection. Further, we looked at various object detections algorithms and their prediction efficiencies. Then, we looked at various ways of implementing them using Python.

In the next chapter, we will learn about object segmentation and understand the importance of segmenting an object in an image. We will also go through the advancement in this field and various ready-to-use algorithm implementations that can be used in projects.

Questions

Image classification and image detection are the same thing. Is this true or false?

True

False

What are the different algorithms for object detection?

R-CNN

SSD

YOLO

All of the above

Object localization and object detection are the same thing. Is this true or false?

True

False

TFOD stands for which of the following?

Tensor Fast Object Detection

Tensor Flow Object Detection

Test Flow Object Detection

None of the above

Which algorithms can be implemented using TFOD?

SSD

Faster R-CNN

Mask R-CNN

All of the above

Answers

b

d

b

b

d

CHAPTER 7

Image Segmentation

The human eye has the power to find a needle in a haystack, to localize an object in an image, to see beyond the surface and uncover the hidden details.

— anonymous

Image segmentation is a process in computer vision where we divide an image into smaller, more meaningful parts. It is like cutting a big puzzle into smaller pieces, with each piece representing a different object or part of an object in the image. For example, suppose you have a picture of a group of animals, like a lion, a zebra, and a giraffe all standing together. With image segmentation, a computer can be trained to recognize and separate each animal, so we can see the lion, zebra, and giraffe as individual images. This makes it easier to understand the image and analyze different objects present in it. It is a useful technique that helps us extract meaningful information from images and make sense of them.

Structure

In this chapter, we will cover the following topics:

Difference between image classification, detection, and segmentation

Image segmentation architectures

U-Net python implementation

FCN-8 python implementation

Mask R-CNN python implementation

Objectives

After studying this chapter, you should be able to understand the concept of image segmentation and how it is different from image classification and object detection techniques. You will also know the process of and the underlying code for implementing image segmentation architectures using Python.

Difference between image classification, detection and segmentation

Image classification is the process of identifying what is in an image. For example, a classifier might be trained to recognize that an image contains a dog.

Object detection is similar to image classification, but it also tells you where in the image the object is located. So, in addition to identifying that an image contains a dog, an object detector would draw a bounding box around the dog in the image.

Image segmentation is a process of dividing an image into multiple segments or regions, each of which corresponds to a different object or part of an object. So, in addition to identifying that an image contains a dog and where the dog is located, image segmentation might identify the dog's nose, tail, and paws as separate segments.

Figure 7.1 depicts the difference between image classification, detection, and segmentation:

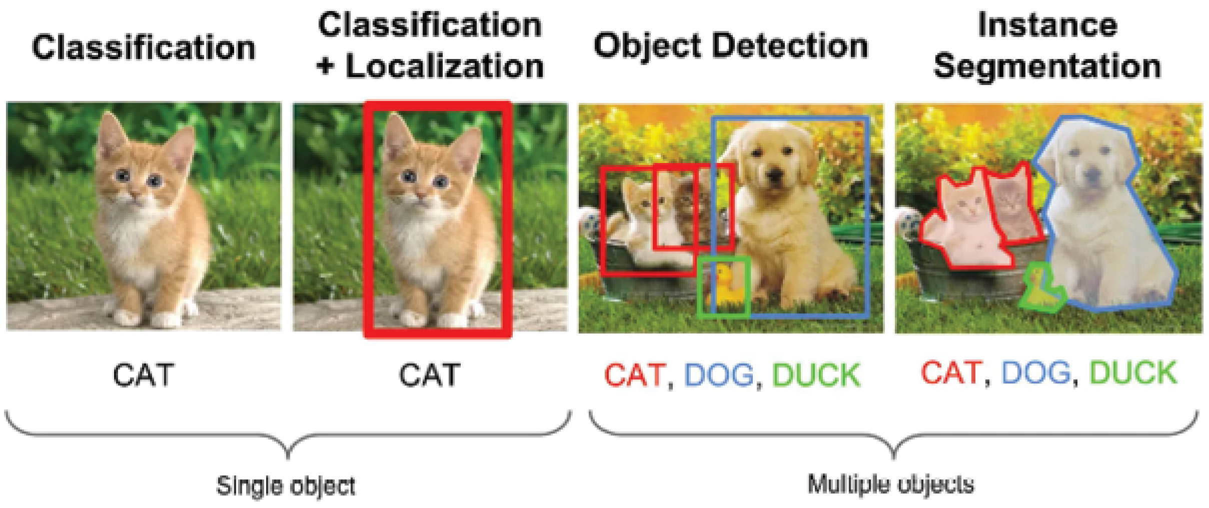


Figure 7.1: Image classification vs detection vs segmentation

Image segmentation architectures

There are several popular image segmentation architectures that can be implemented using the Keras Python library, including the following:

This architecture is a Fully Convolutional Neural Network that is often used for biomedical image segmentation. It consists of an encoder and a decoder, where the encoder is used to extract features from the image, and the decoder is used to generate the segmentation mask.

This is an FCN architecture that uses a VGG-16 encoder and a decoder made of transpose convolutional layers.

This is an FCN architecture that uses a pre-trained VGG-16 model as the encoder and a series of upsampling and convolutional layers as the decoder.

This is an FCN architecture that uses an atrous convolutional layer to increase the resolution of the output segmentation mask.

Mask This architecture uses a Region-Based Convolutional Neural Network to predict object bounding boxes and an FCN to generate the segmentation mask.

This is an extension of the FCN architecture that combines the advantages of global and local context and uses the pyramid pooling module to adaptively reweigh the spatial information at multiple scales.

Encoder-decoder with Atrous separable convolution (DeepLab This is an extension of the DeepLab V3 architecture. It uses Atrous separable convolution in the decoder and is considered state-of-the-art for semantic image segmentation tasks.

It is a lightweight version of UNet architecture. It uses pre-trained encoder as a backbone, and segmentation network is built using the decoder block of UNet.

These architectures can be implemented using Keras, along with other libraries like TensorFlow, which helps implement the neural networks, along with other image processing libraries like OpenCV, PIL, and so on.

U-Net Python implementation

The U-Net architecture is a popular image segmentation architecture that was developed for biomedical image segmentation. It is a fully convolutional neural network that is built upon the autoencoder architecture.

Figure 7.2 depicts the U-Net architecture:

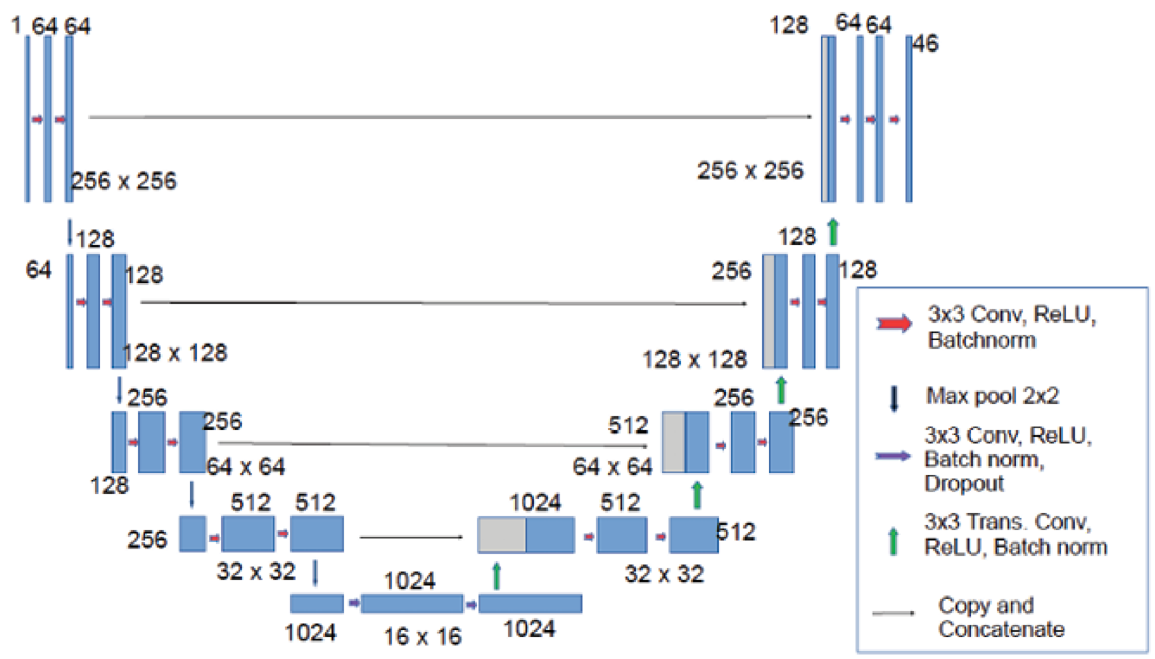


Figure 7.2: Architecture of U-Net for producing k 256-by-256 image masks for a 256-by-256 RGB image

In this section, we will walk through the implementation of the U-Net architecture, which is widely used for image segmentation tasks. We will be using TensorFlow’s Keras API and the Oxford-IIIT Pet Dataset for demonstration.

Prerequisites

Python 3

TensorFlow 2.x

Knowledge of convolutional neural networks

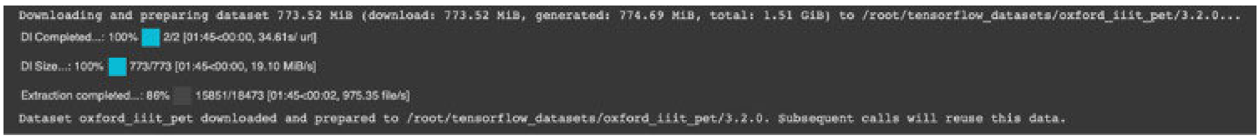
```
pip install tensorflow
```

```
import os

import numpy as np

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers

from tensorflow.keras.preprocessing.image import load_img, img_to_array

import matplotlib.pyplot as plt
```

For this implementation, we will use the Oxford-IIIT Pet dataset, which is available through TensorFlow datasets. This dataset contains images of cats and dogs, with corresponding segmentation masks:

```
import tensorflow_datasets as tfds

# Download the dataset

dataset, info = tfds.load('oxford_iiit_pet:3.2.0', with_info=True)
```

Figure 7.3 depicts the output of the data download step:



```
Downloading and preparing dataset 773.52 MiB (download: 773.52 MiB, generated: 774.69 MiB, total: 1.51 GiB) to /root/tensorflow_datasets/oxford_iiit_pet/3.2.0...
DI Completed...: 100% 2/2 [01:45<00:00, 34.61s/ url]
DI Size...: 100% 773/773 [01:45<00:00, 19.10 MiB/s]
Extraction completed...: 86% 15851/18473 [01:45<00:02, 975.35 file/s]
Dataset oxford_iiit_pet downloaded and prepared to /root/tensorflow_datasets/oxford_iiit_pet/3.2.0. Subsequent calls will reuse this data.
```

Figure 7.3: Data download

We will define functions for loading and preprocessing the images and the masks:

```
def normalize(input_image, input_mask):


    input_image = tf.cast(input_image, tf.float32) / 255.0

    input_mask -= 1

    return input_image, input_mask

@tf.function
```

```
def load_image_train(datapoint):

    input_image = tf.image.resize(datapoint['image'], (128, 128))

    input_mask = tf.image.resize(datapoint['segmentation_mask'], (128, 128))

    if tf.random.uniform(()) > 0.5:

        input_image = tf.image.flip_left_right(input_image)

        input_mask = tf.image.flip_left_right(input_mask)

    input_image, input_mask = normalize(input_image, input_mask)

    return input_image, input_mask

def load_image_test(datapoint):

    input_image = tf.image.resize(datapoint['image'], (128, 128))

    input_mask = tf.image.resize(datapoint['segmentation_mask'], (128, 128))

    input_image, input_mask = normalize(input_image, input_mask)

    return input_image, input_mask

TRAIN_LENGTH = info.splits['train'].num_examples

BATCH_SIZE = 64
```

```
BUFFER_SIZE = 1000



STEPS_PER_EPOCH = TRAIN_LENGTH // BATCH_SIZE

train = dataset['train'].map(load_image_train,
num_parallel_calls=tf.data.experimental.AUTOTUNE)

test = dataset['test'].map(load_image_test)

train_dataset =
train.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).repeat()

train_dataset =
train_dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)

test_dataset = test.batch(BATCH_SIZE)
```

U-Net is an architecture for semantic segmentation. It’s an encoder-decoder type network, where the encoder downsamples the input image and the decoder upsamples and recovers the segmentation mask:

```
def build_unet(input_shape):

    inputs = keras.Input(shape=input_shape)

# Encoder
```

```
   conv1 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')
(inputs)

   conv1 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')
(conv1)

   pool1 = layers.MaxPooling2D(pool_size=(2, 2))(conv1)

   conv2 = layers.Conv2D(128, (3, 3), activation='relu', padding='same')
(pool1)

   conv2 = layers.Conv2D(128, (3, 3), activation=’relu', padding='same')
(conv2)

   pool2 = layers.MaxPooling2D(pool_size=(2, 2))(conv2)

# Decoder



   conv3 = layers.Conv2D(256, (3, 3), activation='relu', padding='same')
(pool2)

   conv3 = layers.Conv2D(256, (3, 3), activation='relu', padding='same')
(conv3)

   up1 = layers.concatenate([layers.UpSampling2D(size=(2, 2))(conv3),
conv2], axis=-1)

   conv4 = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(up1)
```

```
    conv4 = layers.Conv2D(128, (3, 3), activation='relu', padding='same')
(conv4)

    up2 = layers.concatenate([layers.UpSampling2D(size=(2, 2))(conv4),
conv1], axis=-1)

    conv5 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(up2)

    conv5 = layers.Conv2D(64, (3, 3), activation='relu', padding='same')
(conv5)

# Output Layer

    outputs = layers.Conv2D(3, (1, 1), activation='softmax')(conv5)

    return keras.Model(inputs=inputs, outputs=outputs)

# Build U-Net model

model = build_unet((128, 128, 3))

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

EPOCHS = 10


VAL_SUBSPLITS = 5
```

VALIDATION_STEPS =
info.splits['test'].num_examples//BATCH_SIZE//VAL_SUBSPLITS

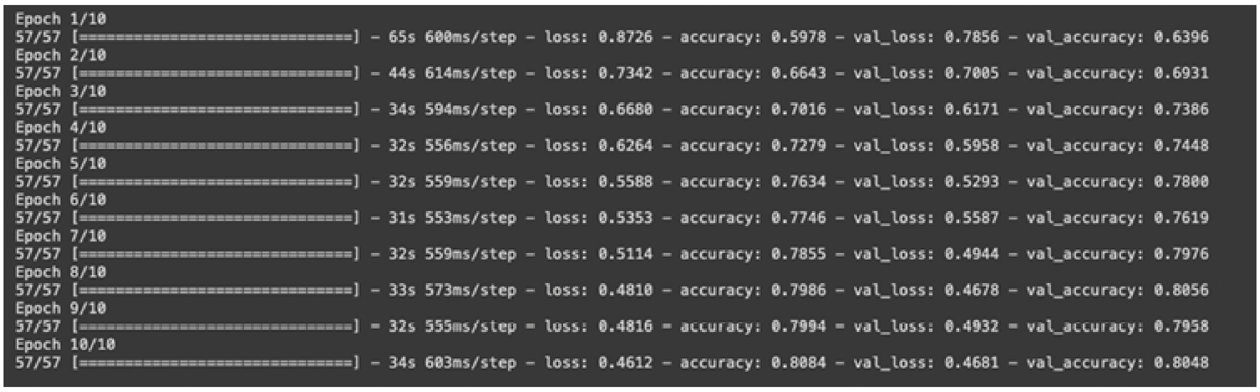
model_history = model.fit(train_dataset, epochs=EPOCHS,

steps_per_epoch=STEPS_PER_EPOCH,

validation_steps=VALIDATION_STEPS,

validation_data=test_dataset)

Figure 7.4 depicts model fitting epochs:



```
Epoch 1/10
57/57 [==============================] - 65s 600ms/step - loss: 0.8726 - accuracy: 0.5978 - val_loss: 0.7856 - val_accuracy: 0.6396
Epoch 2/10
57/57 [==============================] - 44s 614ms/step - loss: 0.7342 - accuracy: 0.6643 - val_loss: 0.7005 - val_accuracy: 0.6931
Epoch 3/10
57/57 [==============================] - 34s 594ms/step - loss: 0.6680 - accuracy: 0.7016 - val_loss: 0.6171 - val_accuracy: 0.7386
Epoch 4/10
57/57 [==============================] - 32s 556ms/step - loss: 0.6264 - accuracy: 0.7279 - val_loss: 0.5958 - val_accuracy: 0.7448
Epoch 5/10
57/57 [==============================] - 32s 559ms/step - loss: 0.5588 - accuracy: 0.7634 - val_loss: 0.5293 - val_accuracy: 0.7800
Epoch 6/10
57/57 [==============================] - 31s 553ms/step - loss: 0.5353 - accuracy: 0.7746 - val_loss: 0.5587 - val_accuracy: 0.7619
Epoch 7/10
57/57 [==============================] - 32s 559ms/step - loss: 0.5114 - accuracy: 0.7855 - val_loss: 0.4944 - val_accuracy: 0.7976
Epoch 8/10
57/57 [==============================] - 33s 573ms/step - loss: 0.4810 - accuracy: 0.7986 - val_loss: 0.4678 - val_accuracy: 0.8056
Epoch 9/10
57/57 [==============================] - 32s 555ms/step - loss: 0.4816 - accuracy: 0.7994 - val_loss: 0.4932 - val_accuracy: 0.7958
Epoch 10/10
57/57 [==============================] - 34s 603ms/step - loss: 0.4612 - accuracy: 0.8084 - val_loss: 0.4681 - val_accuracy: 0.8048
```

Figure 7.4: Model fitting

Let's plot the training history and visualize the segmentation results on test images:

# Plot training history

plt.figure(figsize=(12, 4))

```
plt.subplot(1, 2, 1)

plt.plot(model_history.history["loss"], label="Training Loss")

plt.plot(model_history.history["val_loss"], label="Validation Loss")

plt.legend()


plt.subplot(1, 2, 2)

plt.plot(model_history.history["accuracy"], label="Training Accuracy")

plt.plot(model_history.history["val_accuracy"], label="Validation Accuracy")

plt.legend()

plt.show()

# Visualize predictions

for image, mask in test_dataset.take(1):

    pred_mask = model.predict(image)

    plt.figure(figsize=(10, 10))

    for i in range(9):
        plt.subplot(3, 3, i + 1)

        plt.imshow(image[i])

        plt.imshow(np.argmax(pred_mask[i], axis=-1), alpha=0.5)

        plt.axis("off")

    plt.show()
```

Figure 7.5 depicts model train and validation accuracies and loss:

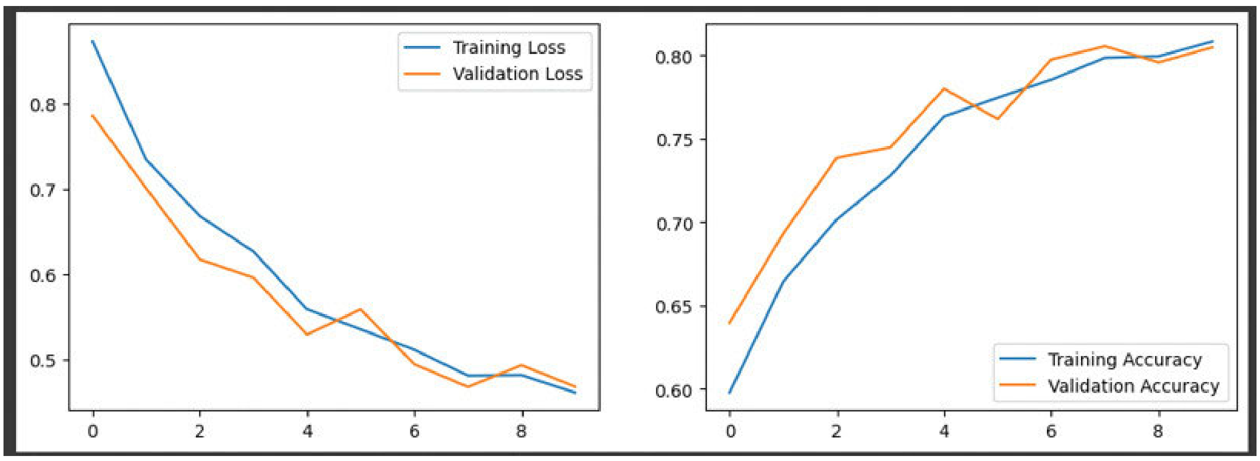


Figure 7.5: Model results

Figure 7.6 depicts image segmentation on a sample images:

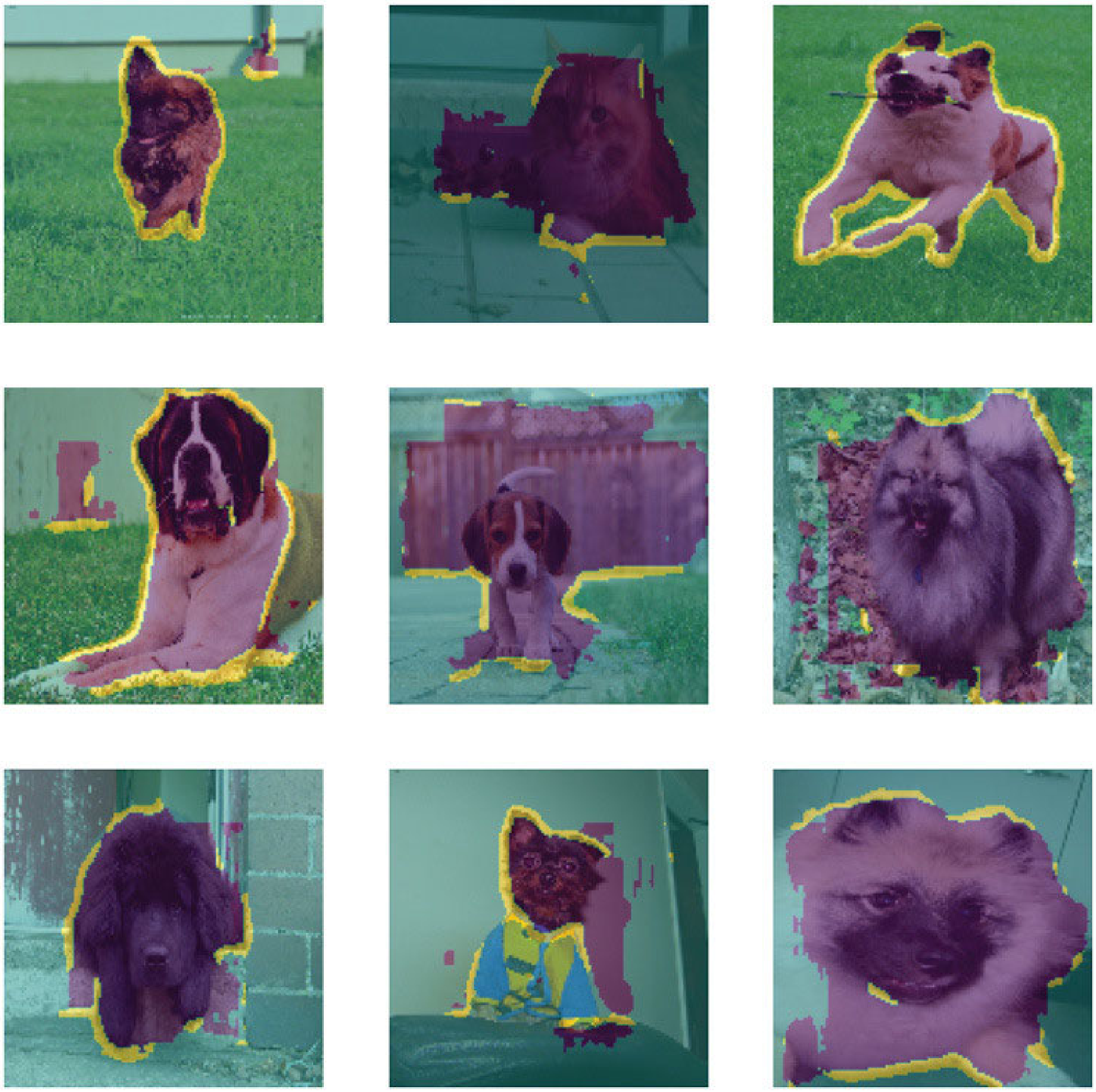


Figure 7.6: Image segmentation on a sample image

In this section, we learned how to implement the U-Net architecture for image segmentation using TensorFlow's Keras API. We built the model, trained it on the Oxford-IIIT Pet dataset, and visualized the segmentation results. The U-Net model is powerful and versatile, making it a popular choice for various image segmentation tasks.

FCN-8 Python implementation

FCN8 is a fully convolutional neural network architecture for image segmentation. It uses a series of convolutional layers to extract features from the input image, and then it uses transposed convolutional layers to upsample the feature maps to the same size as the input image. The final layer produces a segmentation map with a per-pixel prediction of the class label.

In this section, we will walk through the implementation of Fully Convolutional Networks (FCN-8 variant) for semantic image segmentation.

Prerequisites

Python 3

TensorFlow 2.x

Knowledge of convolutional neural networks

pip install tensorflow

Let’s begin by importing the necessary libraries and modules:

import numpy as np

```
import tensorflow as tf

from tensorflow import keras

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D,
Dropout, Conv2DTranspose, concatenate



def fcn8(input_shape, num_classes):

    inputs = Input(input_shape)
```

This line initializes the input tensor of the model with the specified shape.

It consists of five blocks with Conv2D layers, followed by MaxPooling. Each block is responsible for extracting features from the input images.

```
# Block 1

    conv1 = Conv2D(64, 3, activation='relu', padding='same')(inputs)

    conv1 = Conv2D(64, 3, activation='relu', padding='same')(conv1)

    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)

# Block 2
```

```
    conv2 = Conv2D(128, 3, activation='relu', padding='same')(pool1)

    conv2 = Conv2D(128, 3, activation='relu', padding='same')(conv2)

    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

# Block 3

    conv3 = Conv2D(256, 3, activation='relu', padding='same')(pool2)

    conv3 = Conv2D(256, 3, activation='relu', padding='same')(conv3)

    pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)

# Block 4

    conv4 = Conv2D(512, 3, activation='relu', padding='same')(pool3)


    conv4 = Conv2D(512, 3, activation='relu', padding='same')(conv4)

    pool4 = MaxPooling2D(pool_size=(2, 2))(conv4)

# Block 5

    conv5 = Conv2D(512, 3, activation='relu', padding='same')(pool4)
    conv5 = Conv2D(512, 3, activation='relu', padding='same')(conv5)

    pool5 = MaxPooling2D(pool_size=(2, 2))(conv5)
```

Convert the fully connected layers to convolutional layers to make the network fully convolutional:

```
# Fully Convolutionalization

    fc6 = Conv2D(4096, 7, activation='relu', padding='same')(pool5)

    fc6 = Dropout(0.5)(fc6)

    fc7 = Conv2D(4096, 1, activation='relu', padding='same')(fc6)

    fc7 = Dropout(0.5)(fc7)
```

This section merges high-level and low-level features to refine the segmentation outcomes. The Conv2DTranspose layers are employed to enlarge or upscale the feature maps.

```
# Score Pooling 4

    score_pool4 = Conv2D(num_classes, 1, activation='relu',
padding='same')(pool4)

# Score Pooling 7
```

```
    score_fc7 = Conv2D(num_classes, 1, activation='relu', padding='same')
(fc7)

# Deconvolution 2x

    upsample_2x = Conv2DTranspose(num_classes, 4, strides=(2, 2),
padding='same')(score_fc7)

# Score Sum 2

    score_sum2 = concatenate([score_pool4, upsample_2x], axis=3)

# Deconvolution 2x

    upsample_4x = Conv2DTranspose(num_classes, 4, strides=(2, 2),
padding='same')(score_sum2)

# Final Prediction

    upsample_final = Conv2DTranspose(num_classes, 16, strides=(8, 8),
padding='same')(upsample_4x)

    outputs = keras.activations.softmax(upsample_final, axis=-1)
```

The generate_sample_data function creates random input data and labels for training and evaluation:

```
def generate_sample_data(num_samples, image_shape, num_classes):
    X = np.random.rand(num_samples, *image_shape)

    y = np.random.randint(0, num_classes, (num_samples,
*image_shape[:-1], 1))

    return X, y

# Define input shape and number of classes

input_shape = (256, 256, 3)

num_classes = 10



# Generate sample training data

X_train, y_train = generate_sample_data(100, input_shape, num_classes)

# Generate sample test data

X_test, y_test = generate_sample_data(20, input_shape, num_classes)
```

Compile the model with the architecture defined earlier:

```
    model = Model(inputs=inputs, outputs=outputs)

    return model
```

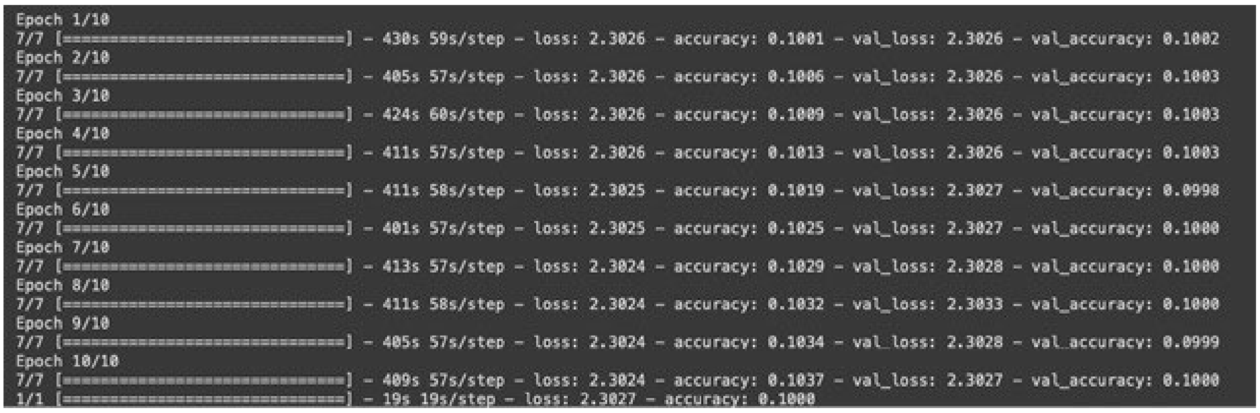
Compile the model with the optimizer, loss function, and metric:

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

Train the model on the sample dataset:

model.fit(X_train, y_train, batch_size=16, epochs=10, validation_data=(X_test, y_test))

Figure 7.7 depicts model fitting epochs:



```
Epoch 1/10
7/7 [==============================] - 430s 59s/step - loss: 2.3026 - accuracy: 0.1001 - val_loss: 2.3026 - val_accuracy: 0.1002
Epoch 2/10
7/7 [==============================] - 405s 57s/step - loss: 2.3026 - accuracy: 0.1006 - val_loss: 2.3026 - val_accuracy: 0.1003
Epoch 3/10
7/7 [==============================] - 424s 60s/step - loss: 2.3026 - accuracy: 0.1009 - val_loss: 2.3026 - val_accuracy: 0.1003
Epoch 4/10
7/7 [==============================] - 411s 57s/step - loss: 2.3026 - accuracy: 0.1013 - val_loss: 2.3026 - val_accuracy: 0.1003
Epoch 5/10
7/7 [==============================] - 411s 58s/step - loss: 2.3025 - accuracy: 0.1019 - val_loss: 2.3027 - val_accuracy: 0.0998
Epoch 6/10
7/7 [==============================] - 401s 57s/step - loss: 2.3025 - accuracy: 0.1025 - val_loss: 2.3027 - val_accuracy: 0.1000
Epoch 7/10
7/7 [==============================] - 413s 57s/step - loss: 2.3024 - accuracy: 0.1029 - val_loss: 2.3028 - val_accuracy: 0.1000
Epoch 8/10
7/7 [==============================] - 411s 58s/step - loss: 2.3024 - accuracy: 0.1032 - val_loss: 2.3033 - val_accuracy: 0.1000
Epoch 9/10
7/7 [==============================] - 405s 57s/step - loss: 2.3024 - accuracy: 0.1034 - val_loss: 2.3028 - val_accuracy: 0.0999
Epoch 10/10
7/7 [==============================] - 409s 57s/step - loss: 2.3024 - accuracy: 0.1037 - val_loss: 2.3027 - val_accuracy: 0.1000
1/1 [==============================] - 19s 19s/step - loss: 2.3027 - accuracy: 0.1000
```

Figure 7.7: Model fitting epochs

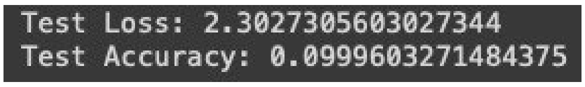
Evaluate the model's performance on the test data:

loss, accuracy = model.evaluate(X_test, y_test)

print("Test Loss:", loss)

```
print("Test Accuracy:", accuracy)
```

Figure 7.8 depicts model accuracy and loss values:



```
Test Loss: 2.3027305603027344
Test Accuracy: 0.0999603271484375
```

Figure 7.8: Model accuracy and loss

In this section, we implemented the FCN-8 architecture for image segmentation using TensorFlow’s Keras API. FCN-8 is an effective architecture for semantic segmentation that combines deep features with upsampling and skip connections.

Mask R-CNN Python implementation

Mask R-CNN is a popular architecture for instance segmentation tasks, which involves predicting both object bounding boxes and per-pixel mask segmentations for each object in an image.

Here’s an example of how to implement Mask R-CNN in Python using Tensorflow:

This step clones the TensorFlow Tensor Processing Unit (TPU) repository from GitHub. Cloning refers to downloading the code and files of this repository to your local machine or environment. This is done by executing the git clone command, followed by the repository URL. The repository contains models and code designed to run on TPUs, which are hardware accelerators specialized in deep learning tasks:

```
!git clone https://github.com/tensorflow/tpu/
```

In this step, you are importing various libraries and modules required for the subsequent code. These libraries include the following:

IPython.display for displaying images

PIL.Image for handling image data

numpy for handling arrays

tensorflow for building and executing machine learning models

sys to manipulate the Python runtime environment

You are also modifying the system path to include directories within the cloned repository and This allows you to import modules that are inside these directories. You are importing coco_metric and visualization_utils from these directories. Lastly, you are disabling TensorFlow v2 behavior to make sure the code is compatible with TensorFlow version 1:

```
from IPython import display

from PIL import Image

import numpy as np

import tensorflow as tf

import sys

sys.path.insert(0, 'tpu/models/official')

sys.path.insert(0, 'tpu/models/official/mask_rcnn')

import coco_metric

from mask_rcnn.object_detection import visualization_utils
```

```
import tensorflow.compat.v1 as tf

tf.disable_v2_behavior()
```

This step initializes a dictionary named which maps integers to names of objects. These integers represent unique identifiers for each object category (for example, ‘person’, ‘bicycle’, ‘car’, and so on.). The category_index is created from ID_MAPPING to store information in a slightly different format, which is likely to be used later for visualization or analysis:

```
ID_MAPPING = {

    1: 'person',



    2: 'bicycle',

    3: 'car',

    4: 'motorcycle',

    5: 'airplane',

    6: 'bus',

    7: 'train',

    8: 'truck',
    9: 'boat',

    10: 'traffic light',

    11: 'fire hydrant',

    13: 'stop sign',

    14: 'parking meter',

    15: 'bench',

    16: 'bird',

    17: 'cat',

    18: 'dog',

    19: 'horse',

    20: 'sheep',

    21: 'cow',

    22: 'elephant',

    23: 'bear',
```

24: 'zebra',

25: 'giraffe',

27: 'backpack',

28: 'umbrella',

31: 'handbag',

32: 'tie',

33: 'suitcase',

34: 'frisbee',

35: 'skis',

36: 'snowboard',

37: 'sports ball',

38: 'kite',

39: 'baseball bat',

```
40: 'baseball glove',

41: 'skateboard',

42: 'surfboard',

43: 'tennis racket',

44: 'bottle',

46: 'wine glass',

47: 'cup',

48: 'fork',

49: 'knife',

50: 'spoon',

51: 'bowl',

52: 'banana',

53: 'apple',

54: 'sandwich',
```

55: 'orange',

56: 'broccoli',

57: 'carrot',

58: 'hot dog',

59: 'pizza',

60: 'donut',

61: 'cake',

62: 'chair',

63: 'couch',

64: 'potted plant',

65: 'bed',

67: 'dining table',

70: 'toilet',

72: 'tv',

73: 'laptop',

74: 'mouse',

75: 'remote',

76: 'keyboard',

77: 'cell phone',

78: 'microwave',

79: 'oven',

80: 'toaster',

81: 'sink',

82: 'refrigerator',

84: 'book',

85: 'clock',

86: 'vase',

87: 'scissors',

```
    88: 'teddy bear',

    89: 'hair drier',

    90: 'toothbrush',

}

category_index = {k: {'id': k, 'name': ID_MAPPING[k]} for k in
ID_MAPPING}
```

In this step, the required libraries are imported, and an image is downloaded from a specified URL. The image is saved as The code then reads the image in binary mode, and the image data is converted into a numpy array. The image dimensions (width and height) are extracted and stored. The image is displayed in the IPython environment with a width of 1024 pixels:

```
# the required libraries

import numpy as np

from PIL import Image

from IPython import display

# Download an image from a specific URL and save it as "test1.jpg"
```

```
!wget https://img.theculturetrip.com/768x/smart/wp-
content/uploads/2019/11/r9j01k.jpg -O test1.jpg

# Set the path of the downloaded image

image_path = 'test1.jpg'

# Open the image file in binary mode

with open(image_path, 'rb') as file:

# Convert the image data into a numpy array

    np_image_string = np.array([file.read()])

# Open the image using the PIL library

image = Image.open(image_path)

# Get the width and height of the image

width, height = image.size

# Convert the image data into a numpy array with the specified dimensions
and data type

np_image = np.array(image.getdata()).reshape(height, width,
3).astype(np.uint8)
```

```
# Display the image with a width of 1024 pixels

display.display(display.Image(image_path, width=1024))
```

Figure 7.9 depicts a sample image for image segmentation demo:

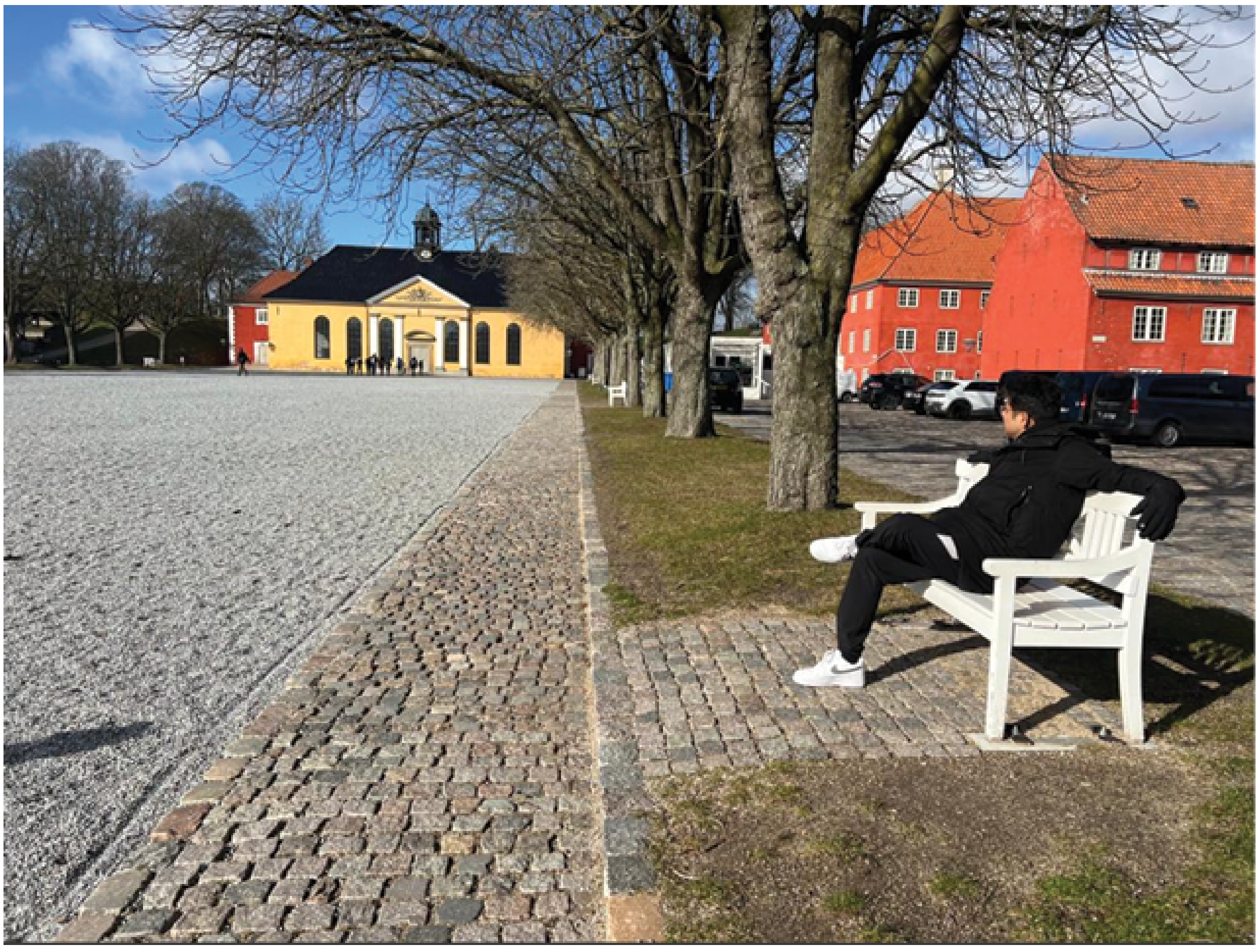


Figure 7.9: Sample image for image segmentation

A TensorFlow session is created and assigned to the variable session. This session will allow you to run a computation graph. It is part of TensorFlow's v1.x way of executing models:

```
session = tf.Session(graph=tf.Graph())
```

In this step, a pre-trained model is loaded from a specified directory in Google Cloud Storage. This directory contains the saved model. The tf.saved_model.loader.load() function is used to load the model into the previously created TensorFlow session:

```
# Loading pretarined model

saved_model_dir = 'gs://cloud-tpu-checkpoints/mask-rcnn/1555659850'
#@param {type:"string"}

_ = tf.saved_model.loader.load(session, ['serve'], saved_model_dir)
```

This step involves running the instance segmentation on the input image. Instance segmentation is a process where each object instance in an image is detected and delineated. The model’s outputs include the number of detections, bounding boxes, class labels, scores, and masks for the objects detected. The step also processes the raw outputs to make them ready for visualization.

```
# Instance Segmentation

num_detections, detection_boxes, detection_classes, detection_scores,
detection_masks, image_info = session.run(

    ['NumDetections:0', 'DetectionBoxes:0', 'DetectionClasses:0',
'DetectionScores:0', 'DetectionMasks:0', 'ImageInfo:0'],
    feed_dict={'Placeholder:0': np_image_string})

num_detections = np.squeeze(num_detections.astype(np.int32), axis=(0,))

detection_boxes = np.squeeze(detection_boxes * image_info[0, 2], axis=
(0,))[0:num_detections]

detection_scores = np.squeeze(detection_scores, axis=(0,))
[0:num_detections]



detection_classes = np.squeeze(detection_classes.astype(np.int32), axis=
(0,))[0:num_detections]

instance_masks = np.squeeze(detection_masks, axis=(0,))
[0:num_detections]

ymin, xmin, ymax, xmax = np.split(detection_boxes, 4, axis=-1)

processed_boxes = np.concatenate([xmin, ymin, xmax - xmin, ymax -
ymin], axis=-1)

segmentations =
coco_metric.generate_segmentation_from_masks(instance_masks,
processed_boxes, height, width)
```

In this final step, the results are visualized. The code draws the detection boxes and labels on the original image using the processed outputs from the previous step. The number of boxes to draw and the minimum score threshold can be specified. The resulting image with detections is saved as test_results.jpg and is also displayed in the IPython environment:

```
# Results

max_boxes_to_draw = 50  #@param

min_score_thresh = 0.1  #@param {type:"slider", min:0, max:1, step:0.01}

image_with_detections =
visualization_utils.visualize_boxes_and_labels_on_image_array(

    np_image,

    detection_boxes,

    detection_classes,

    detection_scores,



    category_index,

    instance_masks=segmentations,

    use_normalized_coordinates=False,

    max_boxes_to_draw=max_boxes_to_draw,
    min_score_thresh=min_score_thresh)

output_image_path = 'test_results.jpg'

Image.fromarray(image_with_detections.astype(np.uint8)).save(output_ima
ge_path)

display.display(display.Image(output_image_path, width=1024))
```

Figure 7.10 depicts the sample image with bounding boxes after image segmentation:

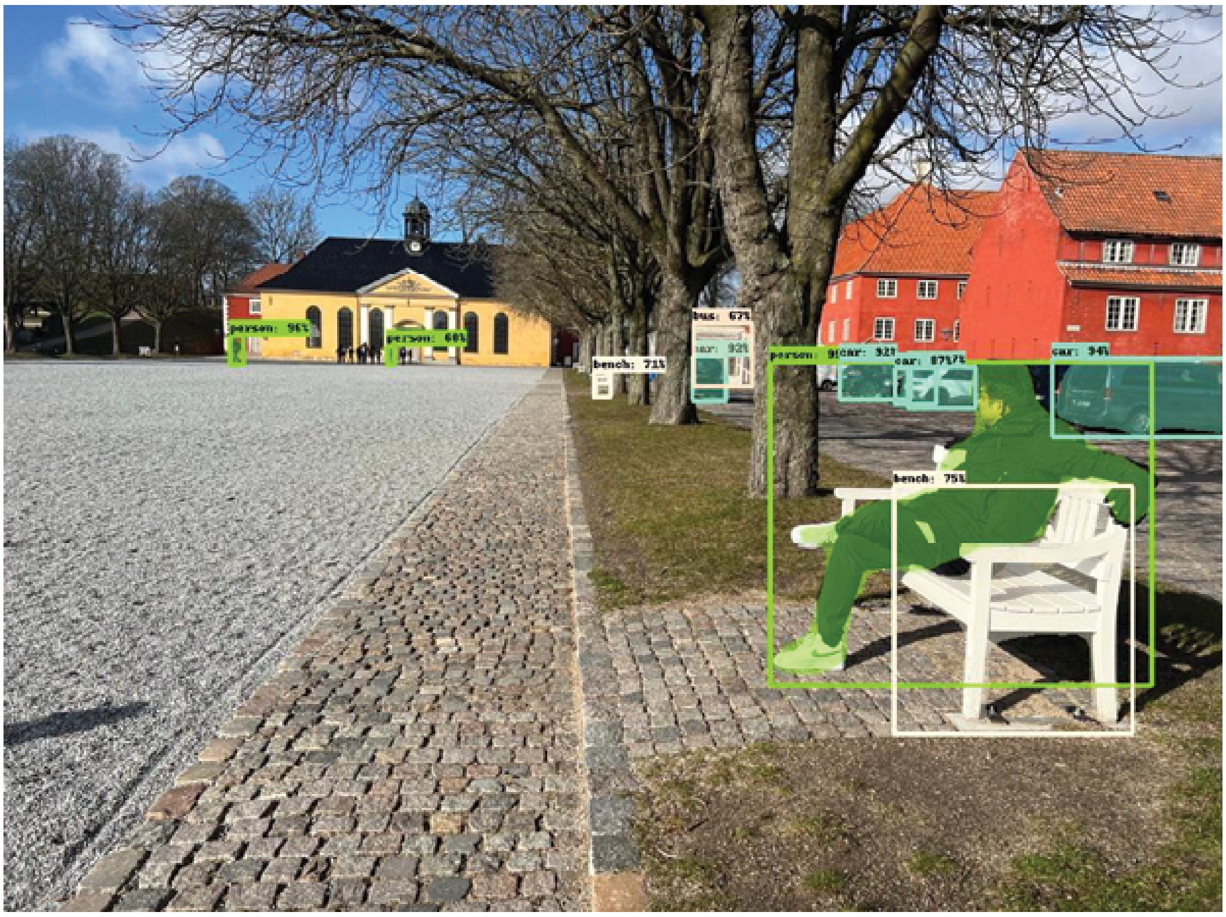


Figure 7.10: Sample image with bounding boxes after image segmentation

Conclusion

Image segmentation is a powerful technique that allows for the identification of specific objects or regions of interest within an image. There are several approaches to image segmentation, including thresholding, region-based methods, and deep learning-based methods. Each approach has its own strengths and weaknesses and is suited for different types of images and tasks.

FCN and Mask R-CNN are two of the most popular deep learning-based image segmentation architectures. FCN utilizes a series of convolutional and transposed convolutional layers to extract features from an image and then sample the feature maps to produce a segmentation map. Mask R-CNN is an extension of the Faster R-CNN object detector; it includes a branch for predicting an object mask in parallel with the existing branch for bounding box recognition.

Implementing image segmentation models in Keras can be relatively easy with the help of pre-built libraries like and However, it’s important to keep in mind that the architecture and training parameters may need to be adjusted to suit the specific task and dataset.

In the next chapter, we will learn about RNN, which is an advanced computational model extensively used in the field of data science and architecture. It is specifically designed to process and analyze sequential data, making it an invaluable tool for tasks like natural language processing, time series analysis, and speech recognition.

Questions

What is the primary goal of image segmentation?

To classify pixels in an image

To detect objects in an image

To enhance the quality of an image

To separate an image into multiple segments

Which of the following is not a type of image segmentation?

Semantic segmentation

Instance segmentation

Object detection

Color-based segmentation

Which of the following is a popular technique for image segmentation?

K-means clustering

Random Forest

Convolutional Neural Networks (CNN)

Support Vector Machines

What is the primary advantage of using Fully Convolutional Networks (FCNs) for image segmentation?

They are less computationally expensive

They are better at handling variable-size inputs

They are better at handling large datasets

They are better at handling high-dimensional inputs

What is the main advantage of using deep learning-based methods for image segmentation?

They can handle large amounts of data

They can handle high-dimensional inputs

They can handle complex and non-linear relationships between inputs and outputs

They can handle all of the above

Answers

d

c

c

b

d

CHAPTER 8

Recurrent Neural Networks

One picture is worth a thousand words

— Albert Einstein

A Recurrent Neural Network is a type of artificial neural network that is well suited to processing sequential data, such as natural language, time series data, and audio. RNNs have an internal memory that allows them to process information from previous time steps in addition to the current input. This allows them to recognize patterns and dependencies across the input sequence and make predictions about future events based on this information.

One of the key features of RNNs is their ability to process sequences of variable length, which makes them useful for a wide range of applications, such as language translation, language modeling, and speech recognition.

RNNs can be trained using various optimization algorithms, such as gradient descent and stochastic gradient descent, to learn the weights of the network and make accurate predictions.

Overall, RNNs are a powerful tool for analyzing and modeling sequential data and have been widely used in several fields to perform various tasks.

Structure

In this chapter, we will cover the following topics:

Algorithms for RNN implementation

RNN implementation

Long Short-Term Memory implementation

Gated Recurrent Unit implementation

Objectives

By the end of this chapter, you should understand the different algorithms suitable for RNN implementation. Additionally, you will be well-acquainted with key algorithms such as LSTM and GRU. Furthermore, you will be equipped to address various challenges using RNN, LSTM, and GRU with the help of Python libraries.

Algorithms for RNN implementation

Backpropagation Through Time This is a widely used algorithm that involves unrolling the RNN and treating it as a deep feedforward neural network, allowing the weights of the network to be updated using gradient descent.

Truncated Backpropagation Through Time This is a variant of BPTT that involves only unrolling the RNN for a limited number of time steps, which can reduce the computational complexity of training.

Stochastic Gradient Descent This is an optimization algorithm that involves updating the weights of the network using small, random batches of data rather than the entire dataset.

Adaptive Moment Estimation This popular optimization algorithm combines the ideas of SGD and momentum to improve the convergence rate of the training process.

Long Short-Term Memory This is a type of RNN that includes special units called "memory cells" that can store information for long periods, allowing the network to better capture long-term dependencies in the data.

Gated Recurrent Unit This is another type of RNN that uses "gates" to control the flow of information within the network, allowing it to better capture long-term dependencies in the data.

Overall, the choice of algorithm will depend on the specific requirements of the task and the available computational resources.

RNN implementation

We will start with a very simple RNN implementation for the most famous MNIST dataset.

Following are the steps for an end-to-end RNN implementation using Python Keras library.

Firstly, we need to import the necessary libraries. We will need numpy for numerical calculations, keras for building the model, and the string library for working with text data:

```
import numpy as np

from keras.models import Sequential

from keras.layers import Dense, SimpleRNN

import string
```

Next, we need to prepare the dataset. For simplicity, we will use a short sequence of characters from the English alphabet:

```
# define the raw dataset

alphabet = string.ascii_lowercase
```

```
# create mapping of characters to integers and reverse

char_to_int = dict((c, i) for i, c in enumerate(alphabet))

int_to_char = dict((i, c) for i, c in enumerate(alphabet))

# prepare the dataset of input to output pairs encoded as integers

seq_length = 3

dataX = []

dataY = []


for i in range(0, len(alphabet) - seq_length, 1):

    seq_in = alphabet[i:i + seq_length]

    seq_out = alphabet[i + seq_length]

    dataX.append([char_to_int[char] for char in seq_in])

    dataY.append(char_to_int[seq_out])
```

In this step, we are creating a mapping of characters to integers and vice versa. We are then creating our input and output sequences. Each input sequence will be a sequence of three characters from the alphabet, and the output will be the next character in the alphabet.

Next, we need to reshape our input sequences into the form [samples, time steps, features] expected by an RNN:

```
X = np.reshape(dataX, (len(dataX), seq_length, 1))
```

We normalize the input values to the range 0-to-1. This is a common practice when working with neural networks:

```
X = X / float(len(alphabet))
```

We are going to predict the next character in the alphabet, which is a multi-class classification problem. Therefore, we need to one hot encode our output variable:

```
from keras.utils import np_utils

y = np_utils.to_categorical(dataY)
```

We can now define our RNN model. We will use a single hidden layer with 32 units. The output layer is a Dense layer using the softmax activation function to output a probability prediction for each of the 26 characters between 0 and 1:

```
model = Sequential()


model.add(SimpleRNN(32, input_shape=(X.shape[1], X.shape[2])))
```

model.add(Dense(y.shape[1], activation='softmax'))

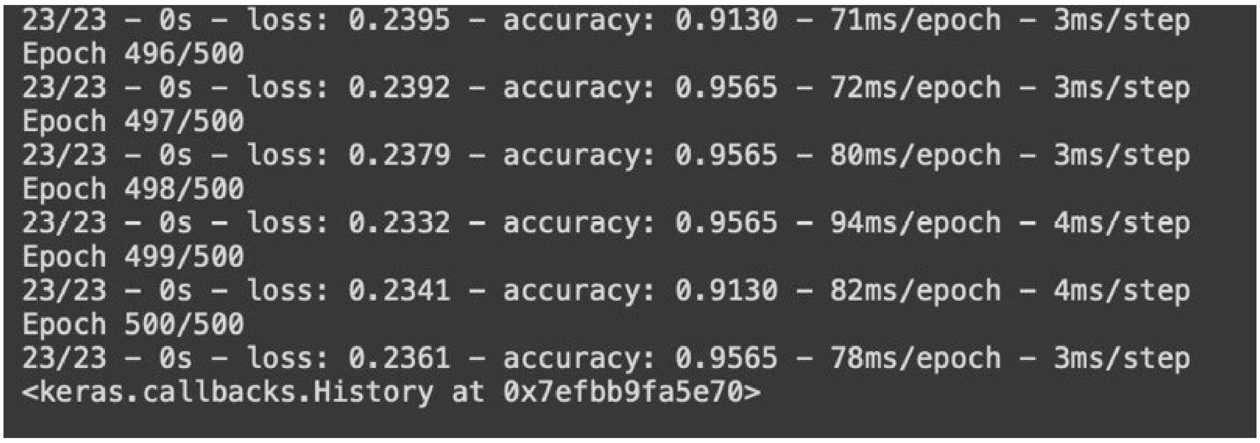
We compile our model using the log loss function in Keras) and use the efficient ADAM optimization algorithm to find the weights.

We then fit the model using batch learning and 500 epochs:

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

model.fit(X, y, epochs=500, batch_size=1, verbose=2)

Figure 8.1 depicts model training epochs:



```
23/23 - 0s - loss: 0.2395 - accuracy: 0.9130 - 71ms/epoch - 3ms/step
Epoch 496/500
23/23 - 0s - loss: 0.2392 - accuracy: 0.9565 - 72ms/epoch - 3ms/step
Epoch 497/500
23/23 - 0s - loss: 0.2379 - accuracy: 0.9565 - 80ms/epoch - 3ms/step
Epoch 498/500
23/23 - 0s - loss: 0.2332 - accuracy: 0.9565 - 94ms/epoch - 4ms/step
Epoch 499/500
23/23 - 0s - loss: 0.2341 - accuracy: 0.9130 - 82ms/epoch - 4ms/step
Epoch 500/500
23/23 - 0s - loss: 0.2361 - accuracy: 0.9565 - 78ms/epoch - 3ms/step
<keras.callbacks.History at 0x7efbb9fa5e70>
```

Figure 8.1: Model training epochs

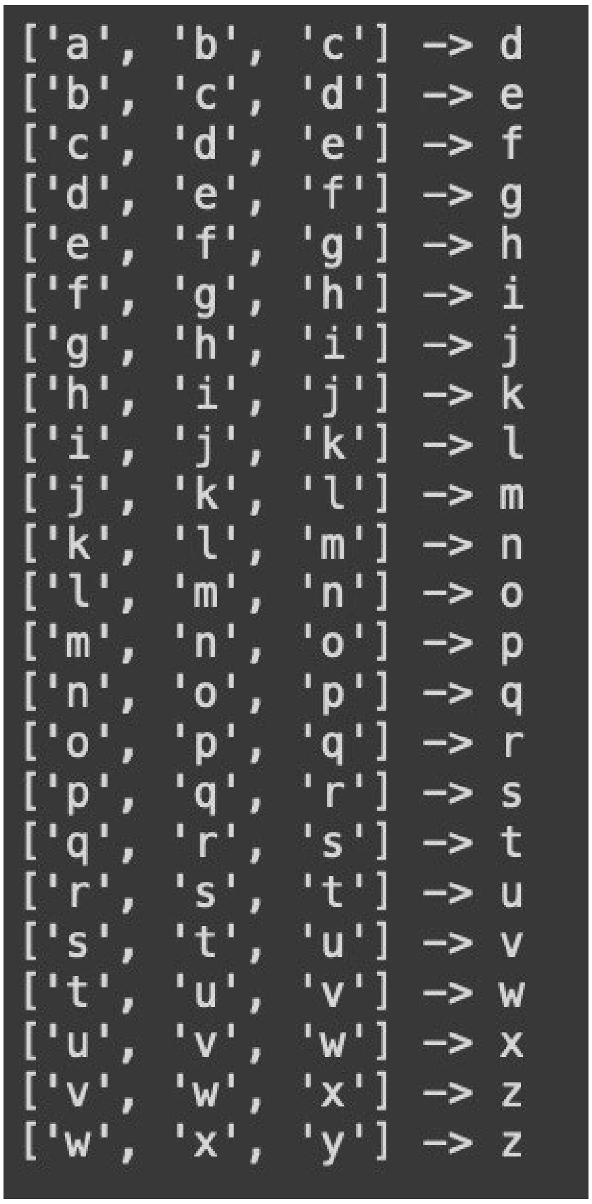
After training, let’s use the model to make predictions:

for pattern in dataX:

```
    x = np.reshape(pattern, (1, len(pattern), 1))

    x = x / float(len(alphabet))

    prediction = model.predict(x, verbose=0)

    index = np.argmax(prediction)

    result = int_to_char[index]

    seq_in = [int_to_char[value] for value in pattern]



    print(seq_in, "->", result)
```

This code snippet will output the input sequence and the predicted output character.

Figure 8.2 depicts model prediction results:



```
['a', 'b', 'c'] -> d
['b', 'c', 'd'] -> e
['c', 'd', 'e'] -> f
['d', 'e', 'f'] -> g
['e', 'f', 'g'] -> h
['f', 'g', 'h'] -> i
['g', 'h', 'i'] -> j
['h', 'i', 'j'] -> k
['i', 'j', 'k'] -> l
['j', 'k', 'l'] -> m
['k', 'l', 'm'] -> n
['l', 'm', 'n'] -> o
['m', 'n', 'o'] -> p
['n', 'o', 'p'] -> q
['o', 'p', 'q'] -> r
['p', 'q', 'r'] -> s
['q', 'r', 's'] -> t
['r', 's', 't'] -> u
['s', 't', 'u'] -> v
['t', 'u', 'v'] -> w
['u', 'v', 'w'] -> x
['v', 'w', 'x'] -> z
['w', 'x', 'y'] -> z
```

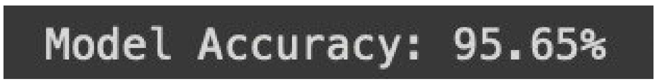
Figure 8.2: Model prediction results

Finally, let us evaluate the performance of our model:

```
scores = model.evaluate(X, y, verbose=0)

print("Model Accuracy: %.2f%%" % (scores[1]*100))
```

This will give you the accuracy of the model on the dataset.

Figure 8.3 depicts model accuracy:



```
Model Accuracy: 95.65%
```

Figure 8.3: Model accuracy

This is a basic example of how to implement a simple RNN in Keras for a sequence prediction problem. Note that for more complex problems and datasets, you may need to tweak the architecture, use different types of RNNs like LSTM or GRU, and spend more time tuning the hyperparameters.

Long short-term memory implementation

Long short-term memory is a type of RNN that is particularly well-suited for modeling long-term dependencies in time series data. Unlike traditional RNNs, which use a single hidden state to capture the entire history of the input sequence, LSTMs use a series of hidden states, known as memory cells, to selectively remember and forget information from the past. This allows LSTMs to capture long-term dependencies without suffering from the vanishing gradient problem, which occurs when the gradients of the error signal become very small over many time steps. LSTMs have been widely used in natural language processing tasks, such as language translation and language modeling, and also in other areas like speech recognition and financial forecasting.

Let's look at an example of how you could implement a LSTM network using the Keras library in Python. In this task, we will train an LSTM on a simple and intuitive dataset: the airline passengers' dataset. This dataset shows the total number of airline passengers in a month, from 1949 to 1960. This problem is common for demonstrating time series prediction because it is a good example where understanding the long-term trend is beneficial for the model.

Start by importing the necessary libraries:

```
import numpy as np

import matplotlib.pyplot as plt
import pandas as pd

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import mean_squared_error


from keras.models import Sequential

from keras.layers import LSTM, Dense

import urllib.request
```

We can download the dataset and read it:

```
url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv"

file = urllib.request.urlopen(url)

data = pd.read_csv(file, usecols=[1], engine='python')
```

We need to normalize the dataset to make the scale of the input features similar:

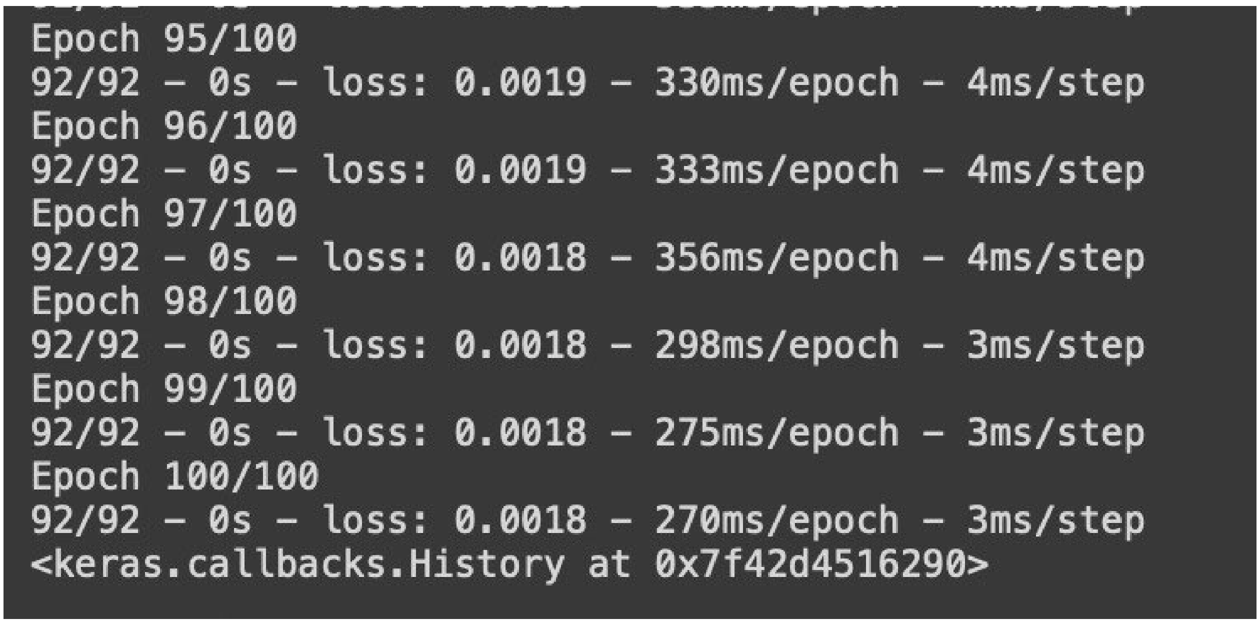
```
scaler = MinMaxScaler(feature_range=(0, 1))

data = scaler.fit_transform(data)
```

We split the dataset into training and testing sets:

```
train_size = int(len(data) * 0.67)

test_size = len(data) - train_size

train, test = data[0:train_size, :], data[train_size:len(data), :]

def create_dataset(dataset, look_back=1):

    dataX, dataY = [], []

    for i in range(len(dataset) - look_back - 1):

        dataX.append(dataset[i:(i + look_back), 0])

        dataY.append(dataset[i + look_back, 0])

    return np.array(dataX), np.array(dataY)

look_back = 3

trainX, trainY = create_dataset(train, look_back)

testX, testY = create_dataset(test, look_back)



# reshape input to be [samples, time steps, features]
```

```
trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))

testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))

model = Sequential()

model.add(LSTM(4, input_shape=(1, look_back)))

model.add(Dense(1))

model.compile(loss='mean_squared_error', optimizer='adam')

model.fit(trainX, trainY, epochs=100, batch_size=1, verbose=2)
```

Figure 8.4 depicts model training epochs:

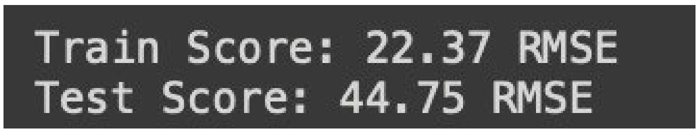


```
Epoch 95/100
92/92 - 0s - loss: 0.0019 - 330ms/epoch - 4ms/step
Epoch 96/100
92/92 - 0s - loss: 0.0019 - 333ms/epoch - 4ms/step
Epoch 97/100
92/92 - 0s - loss: 0.0018 - 356ms/epoch - 4ms/step
Epoch 98/100
92/92 - 0s - loss: 0.0018 - 298ms/epoch - 3ms/step
Epoch 99/100
92/92 - 0s - loss: 0.0018 - 275ms/epoch - 3ms/step
Epoch 100/100
92/92 - 0s - loss: 0.0018 - 270ms/epoch - 3ms/step
<keras.callbacks.History at 0x7f42d4516290>
```

Figure 8.4: Model training epochs

```
trainPredict = model.predict(trainX)

testPredict = model.predict(testX)

# Invert predictions back to original scale



trainPredict = scaler.inverse_transform(trainPredict)

trainY = scaler.inverse_transform([trainY])

testPredict = scaler.inverse_transform(testPredict)

testY = scaler.inverse_transform([testY])

trainScore = np.sqrt(mean_squared_error(trainY[0], trainPredict[:, 0]))

print(f'Train Score: {trainScore:.2f} RMSE')

testScore = np.sqrt(mean_squared_error(testY[0], testPredict[:, 0]))

print(f'Test Score: {testScore:.2f} RMSE')
```

Figure 8.5 depicts model scores:



```
Train Score: 22.37 RMSE
Test Score: 44.75 RMSE
```

Figure 8.5: Model scores

```
plt.plot(scaler.inverse_transform(data))

trainPredictPlot = np.empty_like(data)

trainPredictPlot[:, :] = np.nan

trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict

testPredictPlot = np.empty_like(data)

testPredictPlot[:, :] = np.nan

testPredictPlot[len(trainPredict)+(look_back*2)+1:len(data)-1, :] = testPredict

plt.plot(trainPredictPlot)

plt.plot(testPredictPlot)

plt.show()
```

This will plot the original dataset in blue, the predictions for the training dataset in orange, and the predictions on the test dataset in green.

Figure 8.6 depicts model predictions:

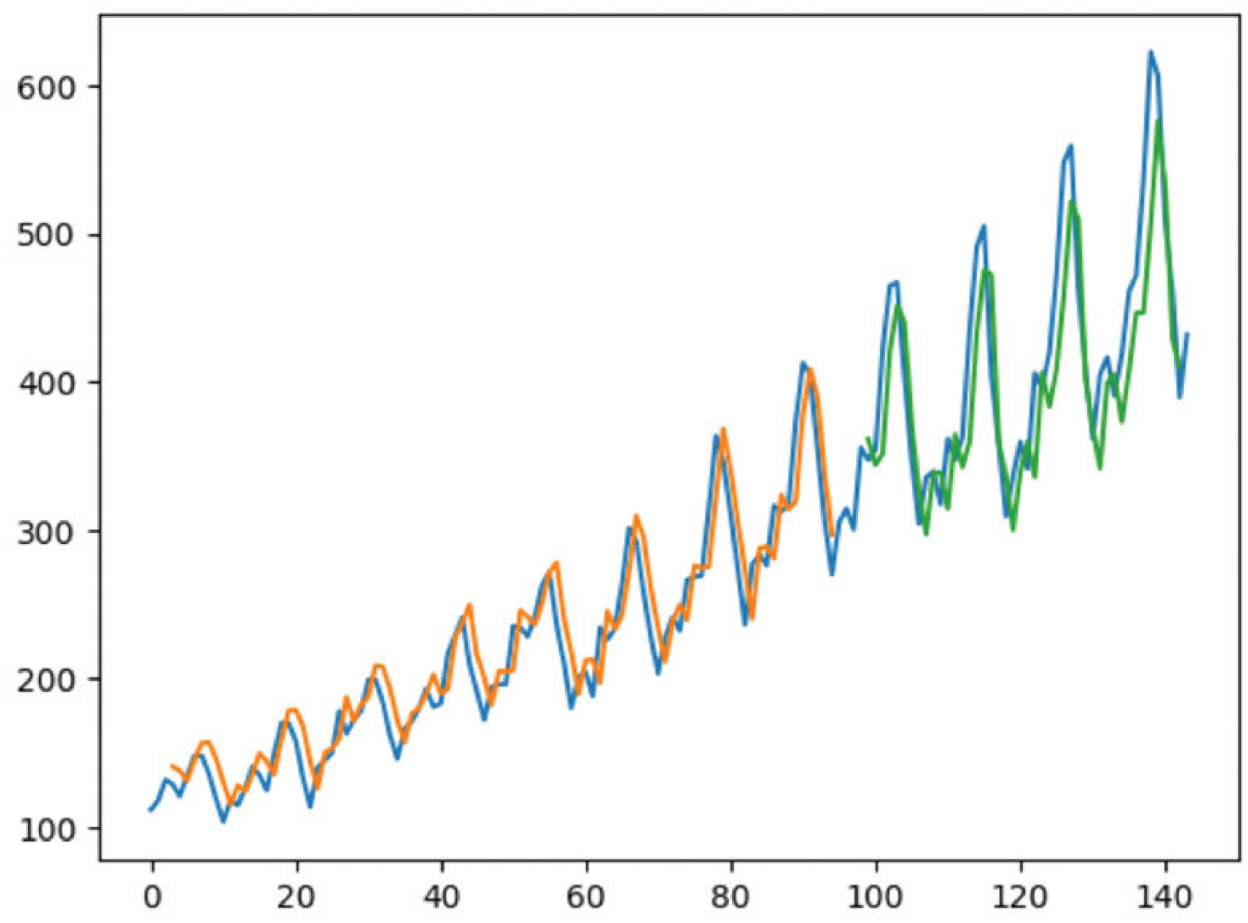


Figure 8.6: Model predictions

Gated Recurrent Unit implementation

Gated recurrent unit (GRU) is another type of RNN that is designed to address the vanishing gradient problem that can occur when training traditional RNNs on long sequences. Like LSTM networks, GRUs use gates to control the flow of information through the network, but they use a simpler gating mechanism that requires fewer parameters and is easier to train. GRUs have been shown to be competitive with LSTMs on a variety of natural language processing tasks, such as language translation and language modeling, and they are often preferred for their simplicity and efficiency. GRUs have also been used in other areas, such as speech recognition and time series forecasting, where they have demonstrated good performance.

In this section, we will walk you through the steps to implement a GRU with some advanced techniques like dropout and recurrent dropout.

Let's begin by importing the necessary libraries:

```
import numpy as np

import tensorflow as tf

from tensorflow.keras.datasets import imdb

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, GRU, Embedding
from tensorflow.keras.preprocessing.sequence import pad_sequences

from tensorflow.keras.utils import to_categorical
```

We load the IMDB dataset, which is available in TensorFlow Keras. We will restrict the dataset to the top 10,000 words:

```
top_words = 10000

(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=top_words)
```

We need to make sure all sequences have the same length. For this, we will use padding:

```
max_review_length = 500

X_train = pad_sequences(X_train, maxlen=max_review_length)

X_test = pad_sequences(X_test, maxlen=max_review_length)
```

We will build a simple model with an Embedding layer, followed by a GRU layer with dropout, and finally, a Dense layer for the output:

```
embedding_vector_length = 32

model = Sequential()
```

model.add(Embedding(top_words, embedding_vector_length, input_length=max_review_length))

model.add(GRU(units=100, dropout=0.2, recurrent_dropout=0.2))

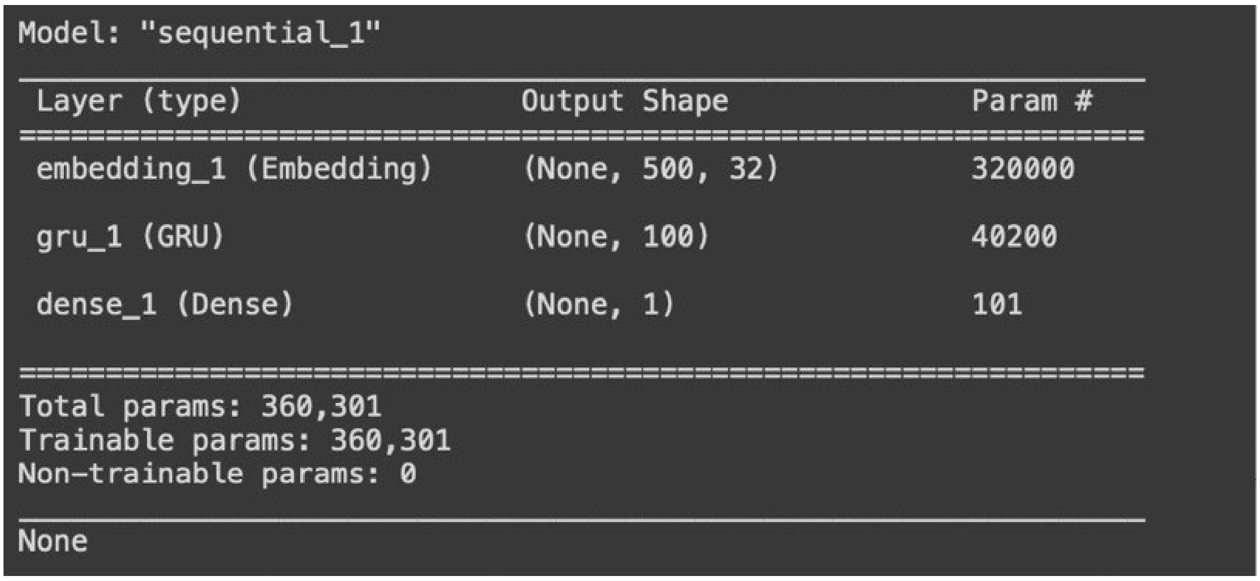
model.add(Dense(1, activation='sigmoid'))

We will compile the model using binary cross-entropy as the loss function (since it is a binary classification problem) and optimizer:

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

print(model.summary())

Figure 8.7 depicts model summary:



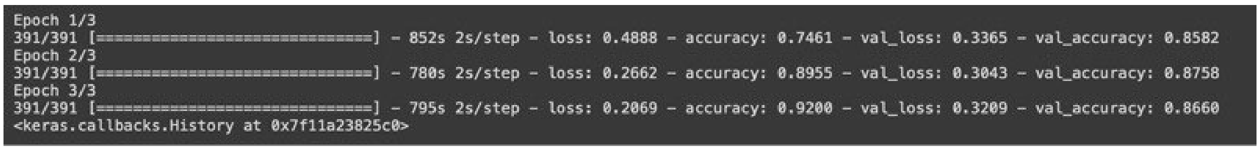
```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, 500, 32)           320000

 gru_1 (GRU)                 (None, 100)               40200

 dense_1 (Dense)             (None, 1)                 101

=================================================================
Total params: 360,301
Trainable params: 360,301
Non-trainable params: 0
_________________________________________________________________
None
```

Figure 8.7: Model summary

Now, let us train the model using the training dataset. We will also monitor its performance on the test dataset:

model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=3, batch_size=64)

Figure 8.8 depicts model training epochs:



```
Epoch 1/3
391/391 [==============================] - 852s 2s/step - loss: 0.4888 - accuracy: 0.7461 - val_loss: 0.3365 - val_accuracy: 0.8582
Epoch 2/3
391/391 [==============================] - 780s 2s/step - loss: 0.2662 - accuracy: 0.8955 - val_loss: 0.3043 - val_accuracy: 0.8758
Epoch 3/3
391/391 [==============================] - 795s 2s/step - loss: 0.2069 - accuracy: 0.9200 - val_loss: 0.3209 - val_accuracy: 0.8660
<keras.callbacks.History at 0x7f11a23825c0>
```

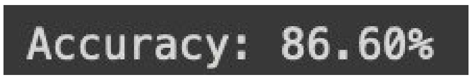
Figure 8.8: Model training epochs

Let us evaluate the model's performance on the test dataset:

scores = model.evaluate(X_test, y_test, verbose=0)

print("Accuracy: %.2f%%" % (scores[1] * 100))

Figure 8.9 depicts model accuracy:



```
Accuracy: 86.60%
```

Figure 8.9: Model accuracy

Conclusion

In conclusion, RNNs are a type of neural network that are well-suited for processing sequential data, such as time series data or natural language text. RNNs have the ability to maintain a hidden state that captures information from the past, which allows them to capture dependencies between elements in the sequence. LSTM networks and GRU networks are variants of RNNs that use additional gates to control the flow of information through the network, which allows them to better capture long-term dependencies in the data. LSTMs and GRUs have been widely used in natural language processing tasks, such as language translation and language modeling, and in other areas like speech recognition and financial forecasting. In the Keras library, the LSTM and GRU layers can be easily added to a model, making it easy to experiment with these powerful models.

Multiple choice Questions

What is a characteristic of RNNs that makes them well-suited for processing sequential data?

They use convolutional layers to extract features from the data.

They use a single hidden state to capture the entire history of the input sequence.

They use a fixed-length input window to process the data.

They use a series of hidden states to selectively remember and forget information from the past.

What is a disadvantage of using traditional RNNs to model long-term dependencies in time series data?

They require a large number of parameters.

They are computationally inefficient.

They suffer from the vanishing gradient problem.

They are prone to overfitting.

How do LSTM networks address the vanishing gradient problem in traditional RNNs?

They use a series of hidden states, known as memory cells, to selectively remember and forget information from the past.

They use a gating mechanism to control the flow of information through the network.

They use a combination of convolutional and fully connected layers to extract features from the data.

They use a fixed-length input window to process the data.

How do GRU networks differ from LSTM networks?

GRU networks use a simpler gating mechanism that requires fewer parameters.

GRU networks are more computationally efficient than LSTM networks.

GRU networks are more powerful than LSTM networks.

GRU networks are less prone to overfitting than LSTM networks.

Which of the following is a common use case for RNNs, LSTMs, and GRUs?

Object detection in images

Speech recognition

Language translation

Financial forecasting

Answers

b

c

a

a

b

CHAPTER 9

Generative Adversarial Networks

The true test of artificial intelligence is not the ability to imitate humans but the ability to outdo them.

— Tim Urban

A Generative Adversarial Network is a type of neural network used to generate synthetic data that is similar to a given training dataset. A GAN is a type of computer program that is used to create synthetic data that looks like it could be real. For example, a GAN could be used to generate synthetic images of people that look like they could be real photographs.

The way a GAN works is by using two different parts: a generator and a discriminator. The generator is like an artist who creates synthetic images, and the discriminator is like a critic who judges whether the images are real or fake.

The generator and discriminator work together to improve the quality of the synthetic images. The generator creates synthetic images and tries to fool the discriminator into thinking they are real. The discriminator looks at the images and tries to tell if they are real or fake. As the generator gets better at creating synthetic images that the discriminator cannot tell are fake, the discriminator gets better at telling real images from synthetic ones.

Through this process, the GAN is able to learn what real images look like, and it can use this knowledge to create synthetic images that look more and more like real ones. GANs have been used to create a wide range of synthetic data, including images, audio, and text.

Structure

In this chapter, we will cover the following topics:

Types of GAN

Vanilla GAN Python implementation

Key difference between Vanilla GAN and DCGAN

DCGAN Python implementation

StyleGAN Python implementation

Objectives

After studying this chapter, you will be aware of the various types of GAN. Additionally, you will be able to implement important GAN architectures in Python.

Types of GAN

There are several types of Generative Adversarial Networks including the following:

Vanilla This is the original GAN architecture, proposed by Ian Goodfellow et al. in 2014. This type of GAN consists of a generator network and a discriminator network, trained using the adversarial process..

Conditional GAN This is a variant of the GAN architecture that allows the generation of synthetic data to be controlled by a set of external conditions.

Deep Convolutional GAN This is a variant of the GAN architecture that uses deep convolutional neural networks as the generator and discriminator networks. This type of GAN is particularly well suited for generating images.

Wasserstein GAN This is a variant of the GAN architecture that uses the Wasserstein distance as the loss function for training the generator and discriminator networks.

Style This is a variant of the GAN architecture that uses an intermediate latent space to control the style of the generated data. This type of GAN is particularly well suited for generating images with high levels of variability.

These are just a few examples of the many types of GANs that have been developed. GANs are a rapidly evolving field, and new variations and improvements are being proposed and developed all the time.

Vanilla GAN Python implementation

In this section, we will walk you through a step-by-step implementation of a Vanilla GAN using Python and PyTorch. GANs consist of two neural networks, i.e., a generator and a discriminator, that are trained simultaneously. The generator creates new data instances, while the discriminator evaluates them. The goal is to train the generator to create data that is indistinguishable from real data for the discriminator.

Make sure you have PyTorch installed in your environment. Use the below pip command for the installation.

```
pip install torch torchvision
```

Let us begin by importing the necessary libraries.

```
import torch

import torch.nn as nn

import torch.optim as optim

import torchvision

import torchvision.transforms as transforms

from torch.utils.data import DataLoader
```

```
import matplotlib.pyplot as plt

import numpy as np
```

For this tutorial, we will use the MNIST dataset. The dataset consists of handwritten digits and is a common dataset used for training simple models in machine learning.

```
transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.5,), (0.5,))])



train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
download=True, transform=transform)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
```

The generator takes a random noise vector as input and produces an output image. The discriminator takes an image as input and outputs a scalar representing the probability of the input image being real.

```
# Generator

class Generator(nn.Module):

    def __init__(self):

        super(Generator, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(100, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 1024),
            nn.ReLU(),
            nn.Linear(1024, 28*28),
            nn.Tanh()
        )
    def forward(self, input):
        return self.main(input).view(-1, 1, 28, 28)
# Discriminator
class Discriminator(nn.Module):
```

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

    self.main = nn.Sequential(
        nn.Linear(28*28, 1024),
        nn.LeakyReLU(0.2),
        nn.Linear(1024, 512),
        nn.LeakyReLU(0.2),
        nn.Linear(512, 256),
        nn.LeakyReLU(0.2),
        nn.Linear(256, 1),
        nn.Sigmoid()
    )
def forward(self, input):
    return self.main(input.view(-1, 28*28))
```

We will use the binary cross entropy loss and the Adam optimizer.

```
# Create the Generator and Discriminator

generator = Generator()

discriminator = Discriminator()

# Loss function

criterion = nn.BCELoss()

# Optimizers

lr = 0.0002

optimizer_G = optim.Adam(generator.parameters(), lr=lr)

optimizer_D = optim.Adam(discriminator.parameters(), lr=lr)
```

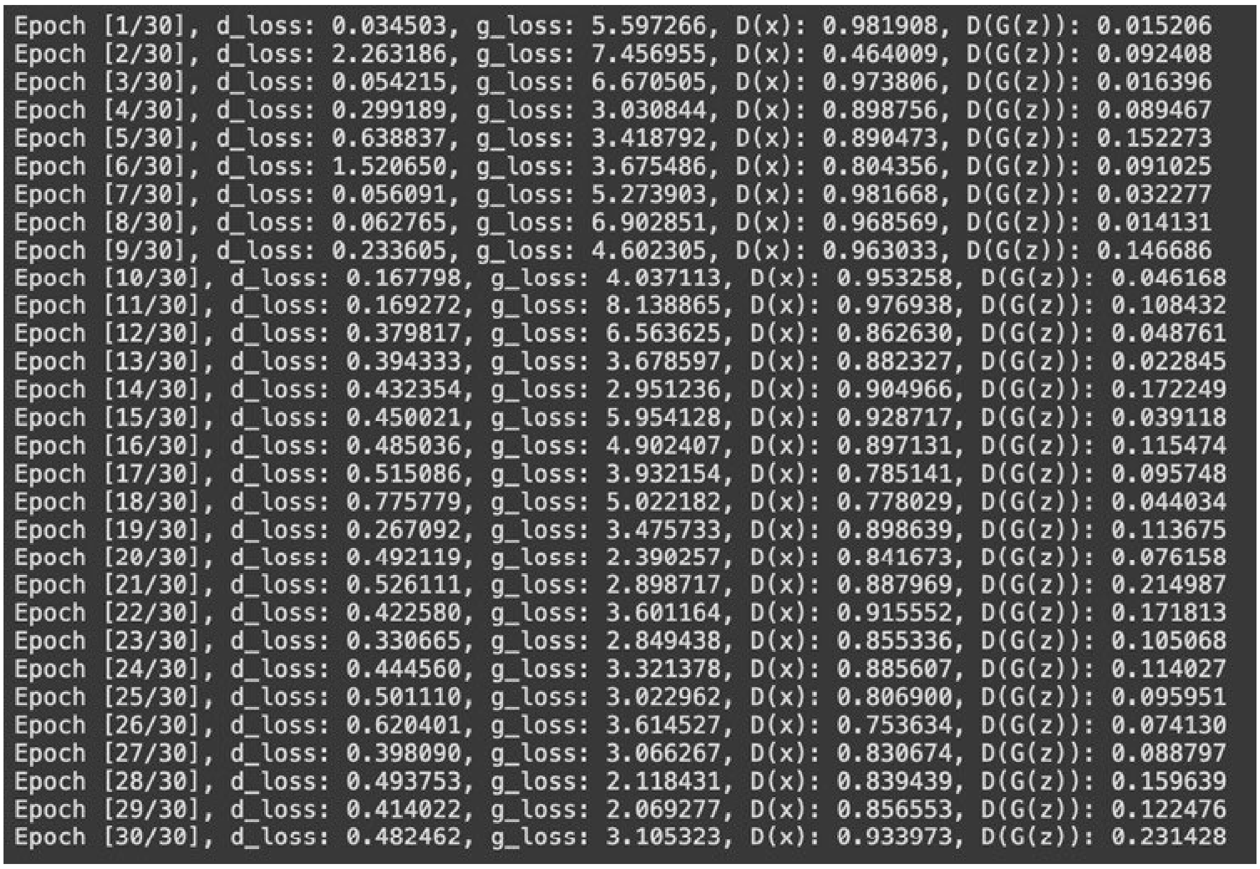
Now, we will train the GAN. For each batch of real data, we train the discriminator with a batch of fake data generated by the generator. We then train the generator to produce more realistic data.

```
num_epochs = 30

latent_vector_size = 100
```

```
for epoch in range(num_epochs):
    for i, (images, _) in enumerate(train_loader):
        batch_size = images.size(0)
        # Labels
        real_labels = torch.ones(batch_size, 1)
        fake_labels = torch.zeros(batch_size, 1)
        # Train Discriminator
        outputs = discriminator(images)
        d_loss_real = criterion(outputs, real_labels)
        real_score = outputs
        z = torch.randn(batch_size, latent_vector_size)
        fake_images = generator(z)
        outputs = discriminator(fake_images.detach())
        d_loss_fake = criterion(outputs, fake_labels)
        fake_score = outputs
        d_loss = d_loss_real + d_loss_fake

        optimizer_D.zero_grad()

        d_loss.backward()

        optimizer_D.step()

        # Train Generator

        z = torch.randn(batch_size, latent_vector_size)

        fake_images = generator(z)



        outputs = discriminator(fake_images)

        g_loss = criterion(outputs, real_labels)

        optimizer_G.zero_grad()

        g_loss.backward()

        optimizer_G.step()

    print(f'Epoch [{epoch+1}/{num_epochs}], d_loss: {d_loss.item():.6f},
g_loss: {g_loss.item():.6f}, '
```

f'D(x): {real_score.mean().item():.6f}, D(G(z)): {fake_score.mean().item():.6f}')

Figure 9.1 depicts training epoch status:



```
Epoch [1/30], d_loss: 0.034503, g_loss: 5.597266, D(x): 0.981908, D(G(z)): 0.015206
Epoch [2/30], d_loss: 2.263186, g_loss: 7.456955, D(x): 0.464009, D(G(z)): 0.092408
Epoch [3/30], d_loss: 0.054215, g_loss: 6.670505, D(x): 0.973806, D(G(z)): 0.016396
Epoch [4/30], d_loss: 0.299189, g_loss: 3.030844, D(x): 0.898756, D(G(z)): 0.089467
Epoch [5/30], d_loss: 0.638837, g_loss: 3.418792, D(x): 0.890473, D(G(z)): 0.152273
Epoch [6/30], d_loss: 1.520650, g_loss: 3.675486, D(x): 0.804356, D(G(z)): 0.091025
Epoch [7/30], d_loss: 0.056091, g_loss: 5.273903, D(x): 0.981668, D(G(z)): 0.032277
Epoch [8/30], d_loss: 0.062765, g_loss: 6.902851, D(x): 0.968569, D(G(z)): 0.014131
Epoch [9/30], d_loss: 0.233605, g_loss: 4.602305, D(x): 0.963033, D(G(z)): 0.146686
Epoch [10/30], d_loss: 0.167798, g_loss: 4.037113, D(x): 0.953258, D(G(z)): 0.046168
Epoch [11/30], d_loss: 0.169272, g_loss: 8.138865, D(x): 0.976938, D(G(z)): 0.108432
Epoch [12/30], d_loss: 0.379817, g_loss: 6.563625, D(x): 0.862630, D(G(z)): 0.048761
Epoch [13/30], d_loss: 0.394333, g_loss: 3.678597, D(x): 0.882327, D(G(z)): 0.022845
Epoch [14/30], d_loss: 0.432354, g_loss: 2.951236, D(x): 0.904966, D(G(z)): 0.172249
Epoch [15/30], d_loss: 0.450021, g_loss: 5.954128, D(x): 0.928717, D(G(z)): 0.039118
Epoch [16/30], d_loss: 0.485036, g_loss: 4.902407, D(x): 0.897131, D(G(z)): 0.115474
Epoch [17/30], d_loss: 0.515086, g_loss: 3.932154, D(x): 0.785141, D(G(z)): 0.095748
Epoch [18/30], d_loss: 0.775779, g_loss: 5.022182, D(x): 0.778029, D(G(z)): 0.044034
Epoch [19/30], d_loss: 0.267092, g_loss: 3.475733, D(x): 0.898639, D(G(z)): 0.113675
Epoch [20/30], d_loss: 0.492119, g_loss: 2.390257, D(x): 0.841673, D(G(z)): 0.076158
Epoch [21/30], d_loss: 0.526111, g_loss: 2.898717, D(x): 0.887969, D(G(z)): 0.214987
Epoch [22/30], d_loss: 0.422580, g_loss: 3.601164, D(x): 0.915552, D(G(z)): 0.171813
Epoch [23/30], d_loss: 0.330665, g_loss: 2.849438, D(x): 0.855336, D(G(z)): 0.105068
Epoch [24/30], d_loss: 0.444560, g_loss: 3.321378, D(x): 0.885607, D(G(z)): 0.114027
Epoch [25/30], d_loss: 0.501110, g_loss: 3.022962, D(x): 0.806900, D(G(z)): 0.095951
Epoch [26/30], d_loss: 0.620401, g_loss: 3.614527, D(x): 0.753634, D(G(z)): 0.074130
Epoch [27/30], d_loss: 0.398090, g_loss: 3.066267, D(x): 0.830674, D(G(z)): 0.088797
Epoch [28/30], d_loss: 0.493753, g_loss: 2.118431, D(x): 0.839439, D(G(z)): 0.159639
Epoch [29/30], d_loss: 0.414022, g_loss: 2.069277, D(x): 0.856553, D(G(z)): 0.122476
Epoch [30/30], d_loss: 0.482462, g_loss: 3.105323, D(x): 0.933973, D(G(z)): 0.231428
```

Figure 9.1: Training epoch status

Finally, let us visualize some images generated by the trained generator:

z = torch.randn(1, latent_vector_size)

sample_images = generator(z).detach().numpy().reshape(-1, 28, 28)

plt.figure(figsize=(6, 6))

```
for i in range(1):
    plt.subplot(1, 1, i+1)
    plt.imshow(sample_images[i], cmap='gray')
    plt.axis('off')
plt.show()
```

Figure 9.2 depicts a GAN generated sample image:

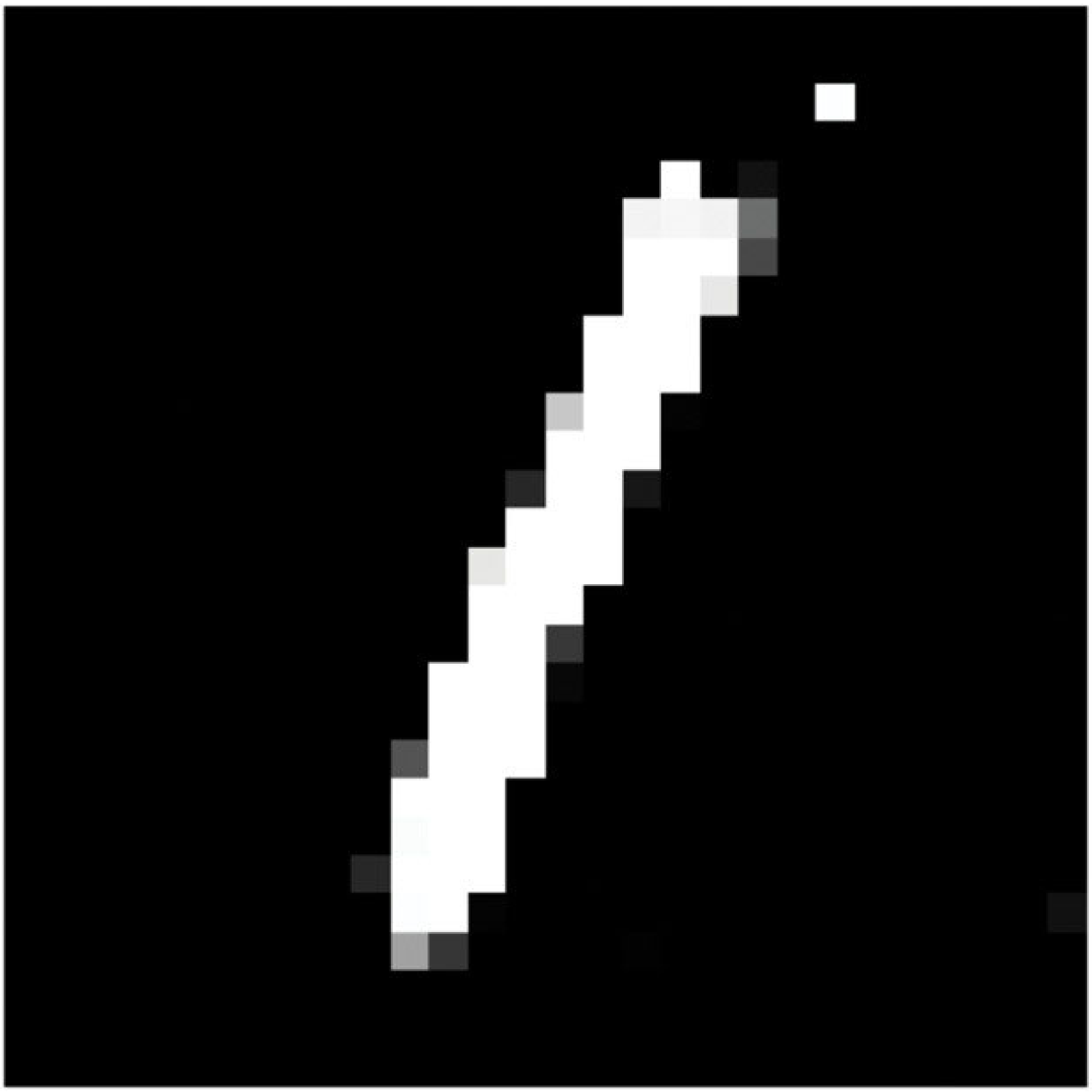


Figure 9.2 : GAN generated sample image

You have just trained your first GAN in PyTorch. You can further experiment with different architectures, loss functions, and datasets to improve the performance and quality of generated images.

Key difference between Vanilla GAN and DCGAN

A Deep Convolutional Generative Adversarial Network is a type of GAN that is composed of convolutional layers. GANs are a type of neural network that is designed to generate new, previously unseen examples from a given dataset, such as images or audio. A GAN consists of two parts: a generator network that creates new examples, and a discriminator network that tries to distinguish the generated examples from the real examples in the dataset.

In a DCGAN, the generator network is composed of transposed convolutional layers, which increase the spatial resolution of the input, while the discriminator network is composed of convolutional layers, which decrease the spatial resolution of the input. The goal of the generator is to create examples that are similar to the real examples in the dataset, while the goal of the discriminator is to distinguish the generated examples from the real ones.

The generator and discriminator are trained together in an adversarial fashion, with the generator trying to create examples that can fool the discriminator and the discriminator trying to correctly identify the generated examples. The training process continues until the generator is able to create examples that are indistinguishable from the real examples to the discriminator.

DCGANs have been used to generate a wide range of data, including images, audio, and text. They have been used for various tasks, such as image generation, style transfer, and super-resolution.

DCGAN are generally considered as one of the first GANs architecture that has been successful on image generation tasks and hence, a benchmark for other image generation techniques.

The key difference between a Vanilla GAN and a DCGAN is the architecture of the generator and discriminator networks.

The generator network in a Vanilla GAN typically consists of a series of fully connected layers, which are not ideal for working with image data. This is because fully connected layers do not take into account the 2D spatial structure of an image and can lead to checkerboard artifacts in the generated images.

On the other hand, the discriminator network in a Vanilla GAN typically consists of a series of fully connected layers, followed by a sigmoid activation function, which outputs a probability of the input being a real image.

In contrast, the generator network in a DCGAN consists of a series of transposed convolutional layers, which increase the spatial resolution of the input and take into account the 2D spatial structure of the images. This leads to generated images with a more natural and smooth appearance.

The discriminator network in a DCGAN typically consists of a series of convolutional layers, followed by batch normalization and LeakyReLU, which is more suited for image data.

In summary, the main difference between a Vanilla GAN and a DCGAN is that the generator and discriminator in a DCGAN use convolutional layers, which are well suited for working with image data and producing visually realistic results, while the generator and discriminator in a Vanilla GAN use fully connected layers, which are not well suited for working with image data and can lead to poor results.

DCGAN Python implementation

In this section, we will walk you through the implementation of a simplified Deep Convolutional Generative Adversarial Network in Python using TensorFlow. The aim is to create an end-to-end example that you can easily follow and execute. DCGAN is an architecture for training GANs to generate high-quality images.

Prerequisites:

Python 3

TensorFlow 2.x

Make sure you have the prerequisites installed:

```
pip install tensorflow
```

Let us start by importing the necessary libraries:

```
import tensorflow as tf

from tensorflow.keras.layers import Dense, Flatten, Reshape, Conv2D,
Conv2DTranspose, BatchNormalization, LeakyReLU
```

```
from tensorflow.keras.models import Sequential

from tensorflow.keras.optimizers import Adam

import numpy as np

import matplotlib.pyplot as plt
```

For this example, we will use the MNIST dataset, a classic dataset of handwritten digits:

```
(train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()



train_images = (train_images - 127.5) / 127.5  # Normalize the images to
[-1, 1]
```

The generator takes a random noise vector as input and generates an image:

```
def make_generator_model():

    model = Sequential()

    # Dense layer

    model.add(Dense(256 * 7 * 7, input_shape=(100,), use_bias=False))
    model.add(BatchNormalization())

    model.add(LeakyReLU())

    # Reshape layer

    model.add(Reshape((7, 7, 256)))

    # Convolutional Transpose layers

    model.add(Conv2DTranspose(128, (5, 5), strides=(1, 1),
padding='same', use_bias=False))

    model.add(BatchNormalization())

    model.add(LeakyReLU())

    model.add(Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same',
use_bias=False))

    model.add(BatchNormalization())

    model.add(LeakyReLU())

    # Output layer

    model.add(Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same',
use_bias=False, activation='tanh'))
```

```
    return model

generator = make_generator_model()
```

The discriminator takes an image as input and outputs the probability of the image being real:

```
def make_discriminator_model():

    model = Sequential()

    # Convolutional layers

    model.add(Conv2D(64, (5, 5), strides=(2, 2), padding='same',
input_shape=[28, 28, 1]))

    model.add(LeakyReLU())

    model.add(BatchNormalization())

    model.add(Conv2D(128, (5, 5), strides=(2, 2), padding='same'))

    model.add(LeakyReLU())

    model.add(BatchNormalization())
    # Flatten layer

    model.add(Flatten())

    # Output layer

    model.add(Dense(1))

    return model

discriminator = make_discriminator_model()
```

We will use the binary cross-entropy loss for both the generator and the discriminator. The generator tries to minimize this loss, while the discriminator tries to maximize it:

```
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):

    real_loss = cross_entropy(tf.ones_like(real_output), real_output)

    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)


    total_loss = real_loss + fake_loss

    return total_loss
```

```
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
generator_optimizer = Adam(1e-4)
discriminator_optimizer = Adam(1e-4)
```

This function defines the operations performed in a single training step:

```
@tf.function
def train_step(images):
    noise = tf.random.normal([batch_size, 100])
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)
        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)
        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)
    gradients_of_generator = gen_tape.gradient(gen_loss,
generator.trainable_variables)

    gradients_of_discriminator = disc_tape.gradient(disc_loss,
discriminator.trainable_variables)


    generator_optimizer.apply_gradients(zip(gradients_of_generator,
generator.trainable_variables))


discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
discriminator.trainable_variables))
```

We will train the DCGAN and periodically save generated images to visually inspect the progress:

```
def generate_and_save_images(model, epoch, test_input):

    predictions = model(test_input, training=False)

    predictions = (predictions + 1) * 127.5

    plt.figure(figsize=(4, 4))

    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i+1)

        plt.imshow(predictions[i, :, :, 0], cmap='gray')

        plt.axis('off')

    plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))

    plt.show()

epochs = 50

noise_dim = 100

num_examples_to_generate = 16

random_vector_for_generation =
tf.random.normal([num_examples_to_generate, noise_dim])

batch_size = 256

# Batch and shuffle the data

train_dataset =
tf.data.Dataset.from_tensor_slices(train_images.reshape(train_images.shap
e[0], 28, 28, 1)).shuffle(60000).batch(batch_size)

# Training Loop
```

```
for epoch in range(epochs):

    for images in train_dataset:

        train_step(images)

# Generate and save images

    if (epoch + 1) % 5 == 0:

        generate_and_save_images(generator, epoch + 1,
random_vector_for_generation)
```

Once the training is complete, you can use the saved images to analyze the results:

```
# Display the final epoch's image

generate_and_save_images(generator, epochs,
random_vector_for_generation)
```

Figure 9.3 depicts DCGAN generated sample images:

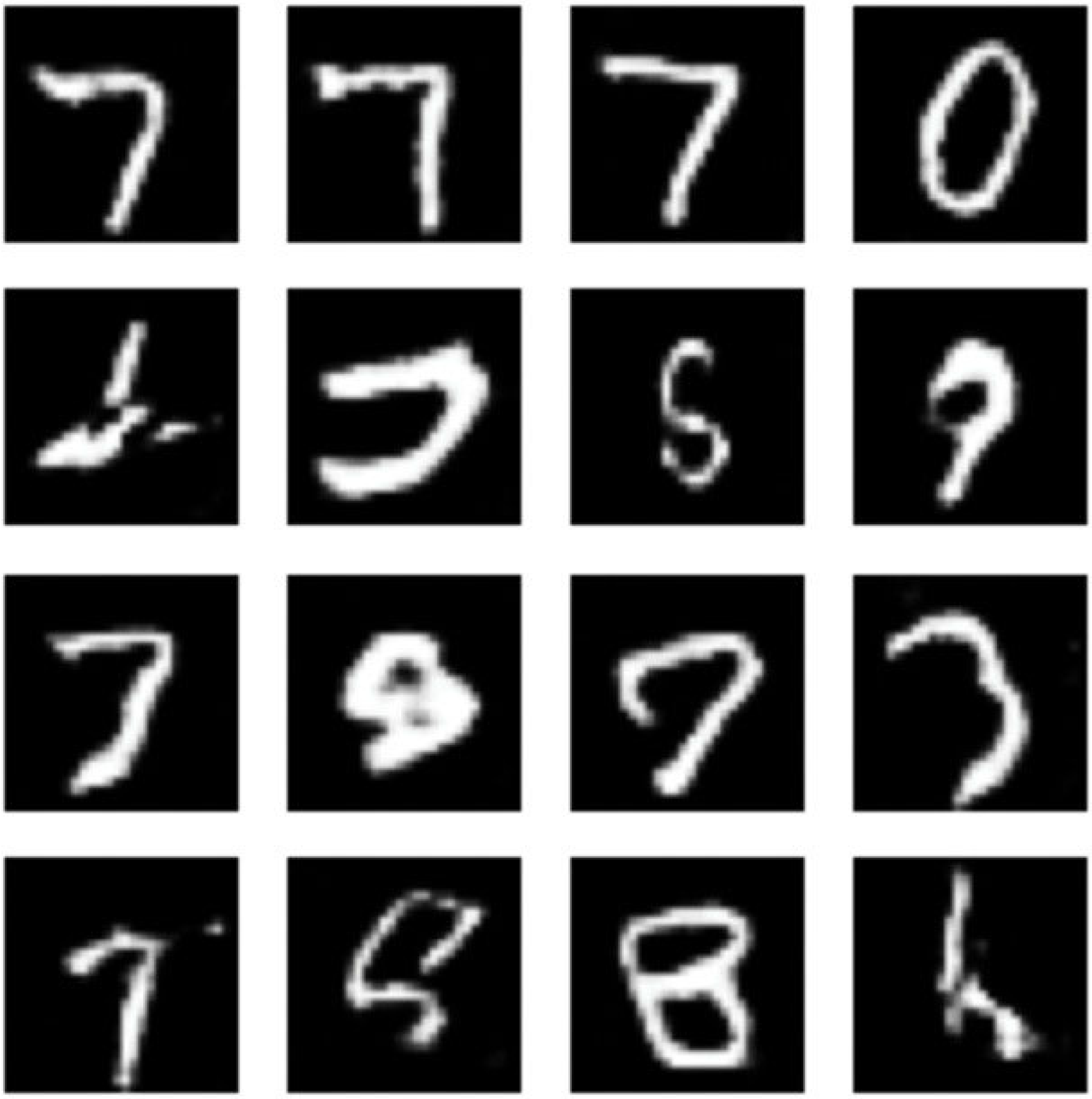


Figure 9.3: DCGAN generated sample images

This completes the DCGAN example. Note that training a GAN can be tricky and may require tweaking hyperparameters or using different architecture choices. However, this simplified example should give you a foundation to build upon. Happy coding!

StyleGAN Python implementation

StyleGAN is a type of GAN architecture that was introduced by NVIDIA in 2018. It is specifically designed for generating high-resolution images, such as photographs of faces. The key innovation of StyleGAN is the use of style-based generator architecture, which separates the high-level structure of the image, such as the pose and identity of a face, from the low-level details, such as the texture of the skin and the color of the eyes. This separation allows the generator to generate highly realistic images while still maintaining the ability to control high-level attributes, such as the pose or the expression of a face. Additionally, StyleGAN uses an Adaptive Instance Normalization technique that enables transferring the style from a reference image to the generated image. This allows for high-quality control over the fine details of the image, such as the wrinkles and the freckles on a face, while maintaining the overall structure of the image.

Here is an example of how you can implement a StyleGAN in Python.

Setup environment

Before you begin, make sure you have installed the required libraries. You can install them using

```
pip install torch torchvision matplotlib
```

Start by importing the necessary libraries:

```
import torch

import torch.nn as nn

import torch.optim as optim

import torchvision

import torchvision.transforms as transforms

from torch.utils.data import DataLoader

import matplotlib.pyplot as plt

import numpy as np
```

Use GPU for faster computation if available:

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

For this tutorial, we will use the CIFAR-10 dataset:

```
batch_size = 64

transform = transforms.Compose([

    transforms.Resize(64),

    transforms.ToTensor(),

    transforms.Normalize((0.5,), (0.5,))

])

dataset = torchvision.datasets.CIFAR10(root='./data', train=True,
download=True, transform=transform)



dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

Define a class for Generator:

```
class Generator(nn.Module):
    def __init__(self, z_dim=128, channels=3):
        super().__init__()
        self.z_dim = z_dim
        self.gen = nn.Sequential(
# Input: N x z_dim x 1 x 1
            self._block(z_dim, 512, 4, 1, 0), # 4x4
            self._block(512, 256, 4, 2, 1), # 8x8
            self._block(256, 128, 4, 2, 1), # 16x16
            self._block(128, 64, 4, 2, 1), # 32x32
# Output: N x channels x 64 x 64
            nn.ConvTranspose2d(64, channels, 4, 2, 1),
            nn.Tanh()
        )
    def _block(self, in_channels, out_channels, kernel_size, stride,
padding):
        return nn.Sequential(
            nn.ConvTranspose2d(in_channels, out_channels, kernel_size,
stride, padding, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
        )
    def forward(self, x):
        return self.gen(x.view(len(x), self.z_dim, 1, 1))
```

Define a class for Discriminator

```
class Discriminator(nn.Module):
    def __init__(self, channels=3):
        super().__init__()
        self.disc = nn.Sequential(
# Input: N x channels x 64 x 64
```

```
            self._block(channels, 64, 4, 2, 1), # 32x32

            self._block(64, 128, 4, 2, 1), # 16x16

            self._block(128, 256, 4, 2, 1), # 8x8

            self._block(256, 512, 4, 2, 1), # 4x4

# Output: N x 1

            nn.Conv2d(512, 1, 4, 1, 0),

            nn.Sigmoid()

        )

    def _block(self, in_channels, out_channels, kernel_size, stride,
padding):

        return nn.Sequential(

            nn.Conv2d(in_channels, out_channels, kernel_size, stride,
padding, bias=False),

            nn.BatchNorm2d(out_channels),

            nn.LeakyReLU(0.2, inplace=True),
        )

    def forward(self, x):

        return self.disc(x).view(-1)

z_dim = 128

lr = 0.0002

gen = Generator(z_dim).to(device)

gen_opt = optim.Adam(gen.parameters(), lr=lr)

disc = Discriminator().to(device)



disc_opt = optim.Adam(disc.parameters(), lr=lr)

criterion = nn.BCELoss()

n_epochs = 5

sample_size = 64

fixed_noise = torch.randn(sample_size, z_dim, device=device)
```

```
for epoch in range(n_epochs):

    for real, _ in dataloader:

        real = real.to(device)

        noise = torch.randn(real.shape[0], z_dim, device=device)

        fake = gen(noise)

        # Train Discriminator

        disc_opt.zero_grad()

        disc_loss = criterion(disc(real), torch.ones(real.size(0),
device=device)) + criterion(disc(fake.detach()), torch.zeros(real.size(0),
device=device))

        disc_loss.backward()

        disc_opt.step()

        # Train Generator

        gen_opt.zero_grad()
        gen_loss = criterion(disc(fake), torch.ones(real.size(0),
device=device))

        gen_loss.backward()

        gen_opt.step()

    # Output Images

    with torch.no_grad():

        samples = gen(fixed_noise)



        samples = (samples + 1) / 2

        samples = samples.cpu().permute(0, 2, 3, 1).numpy()

        plt.figure(figsize=(6, 6))

        for i in range(16):

            plt.subplot(4, 4, i + 1)

            plt.imshow(samples[i])

            plt.axis('off')
    plt.show()
```

This implementation of StyleGAN is a simplified version of the original architecture, which uses a more complex generator architecture consisting of multiple levels, where each level is composed of multiple blocks, wherein each block is composed of several layers. Note that the architecture and hyperparameters are simplified and may not generate good results; it is important to experiment with different architectures and hyperparameters to obtain better results.

Conclusion

In this chapter, we explored several important GAN architectures, including Vanilla GANs, DCGANs, and StyleGANs. We also provided example implementations of these architectures in Python using the Python Keras library.

Vanilla GANs are the simplest GAN architecture, consisting of a generator and a discriminator that are both simple fully connected neural networks. DCGANs are an extension of Vanilla GANs, which use deep convolutional neural networks for both the generator and discriminator. And style GANs, which is one of the most advanced architectures that allows the generation of high-resolution images, such as photographs of faces. With Style GANs, the generator learns to produce new data samples that are similar to the training data, while the discriminator learns to distinguish the generated samples from the real samples.

GANs can be used for a wide variety of tasks, such as image synthesis, image-to-image translation, and image super-resolution. With continued advancements in deep learning techniques and increasing computational resources, GANs will likely play an even more important role in the field of artificial intelligence in the future.

In the next chapter, we will discuss the ground-breaking algorithm in the deep learning sphere: Transformers. We will discuss the various Transformers architectures, the difference between contextual and non-contextual embeddings, and finally, Python code implementation on two major NLP libraries: GPT and BERT.

Questions

What is the main goal of the generator network in a GAN?

To produce new data samples that are similar to the training data

To distinguish the generated samples from the real samples

To optimize the discriminator network

To classify the input data

What type of neural networks are typically used for the generator and discriminator in a DCGAN?

Simple fully-connected neural networks

Deep convolutional neural networks

Recurrent neural networks

Transformer networks

What is the main difference between a Vanilla GAN and a DCGAN?

DCGANs use deep convolutional neural networks, while Vanilla GANs use simple fully-connected neural networks

DCGANs are designed for image generation, while Vanilla GANs are designed for text generation

DCGANs use an adversarial loss function, while Vanilla GANs use a reconstruction loss function

DCGANs are designed for image super-resolution, while Vanilla GANs are designed for image-to-image translation

How does StyleGAN differ from other GAN architectures?

StyleGAN separates the high-level structure of the image from the low-level details

StyleGAN uses an adaptive instance normalization technique

StyleGAN uses progressive growing of GANs

All of the above

What is the main disadvantage of GANs?

GANs are difficult to train and often suffer from stability issues

GANs are very computationally expensive

GANs can only be used for a limited set of tasks

GANs are not interpretable

Answers

a

b

a

d

a

CHAPTER 10

Transformers

Transformers are the backbone of data science and natural language processing, they are the key to unlocking the true potential of data.

— Unknown

Transformers are a type of deep learning model that help computers understand and generate language. Think about all the different ways you can say the same thing, like "I'm going to the store" or "I am off to the store", or "I will go to the store"; they all mean the same thing but are written differently. Transformers help a computer understand all the different ways of saying the same thing. They also help a computer generate new sentences that make sense, like a computer writing a story or a poem. It is like having a good teacher to help you with your writing, but instead of just helping you, it helps the computer understand and write language too!

Transformers are a type of neural network architecture that have become increasingly popular in the field of Natural Language Processing and other domains like computer vision, speech recognition and data science. The architecture, introduced in the 2017 paper Attention Is All You Need by Google researchers, utilizes self-attention mechanisms to process input sequences in a parallel manner, allowing for faster and more efficient training and inference. These models have been shown to achieve state-of-the-art performance on a variety of NLP tasks and have also been applied to other domains with promising results.

Structure

In this chapter, we will cover the following topics:

Introduction to Transformers in deep learning

Various Transformers architectures

Difference between contextual and non-contextual embeddings

BERT Python implementation

GPT Python implementation

Objectives

After studying this chapter, you should be able to understand the concept of Transformers architectures and how it is transforming NLP domain within data science. You should also have understood the process and the underlying code to implement various Transformer-based architectures using Python.

Introduction to Transformers in deep learning

In recent years, deep learning has made significant progress in the field of NLP. One of the most important contributions to this progress has been the Transformer model. In this chapter, we will explore the Transformer model, its architecture, and how it has revolutionized the field of NLP.

The Transformer model was first introduced in the paper Attention is All You Need by Google researchers in 2017. The Transformer model is a neural network architecture that is designed to process sequences of data, such as sentences or paragraphs of text. The key innovation of the Transformer model is the use of self-attention mechanisms, which allows the model to weigh the importance of different parts of the input sequence when making predictions.

The Transformer model comprises an encoder and decoder. The encoder takes the input sequence and produces a set of hidden representations, called the keys and The decoder then takes these hidden representations and uses them to make predictions about the output sequence. The self-attention mechanism is applied in the encoder, where it allows the model to focus on specific parts of the input sequence when producing the hidden representations.

One of the most important applications of the Transformer model is in pre-training large language models. The Bidirectional Encoder Representations from Transformers model, developed by Google researchers in 2018, is a transformer-based model that is pre-trained on a massive amount of unlabeled text data. BERT is trained to understand the meaning of words in context and has shown to be very effective in a wide range of NLP tasks.

Generative Pre-training Transformer model, developed by OpenAI in 2018, is another large-scale language model that is trained on a massive amount of text data. GPT is trained to generate text and has been used in a wide range of natural language generation tasks, such as language translation and text summarization.

The Transformer architecture has also been used to improve the performance of other deep learning models for NLP tasks. The Robustly Optimized BERT model, developed by Facebook AI in 2019, is an improved version of BERT that is trained on more data with more advanced techniques. The Text-to-Text Transfer Transformer model, developed by Google AI in 2019, is designed for a wide range of natural language processing tasks.

In conclusion, the Transformer model has been a major breakthrough in the field of natural language processing. Its self-attention mechanism allows the model to weigh the importance of different parts of the input sequence, and its architecture is particularly well suited to processing sequences of data. The Transformer model has been used to pre-train large language models and improve the performance of other deep learning models for NLP tasks. As the demand for more sophisticated natural language processing continues to grow, the Transformer model will likely continue to be an important part of the deep learning landscape.

Various transformers architectures

There are several variations of the Transformer architecture that have been developed and used in various NLP and other tasks. Some of the most popular Transformer architectures are as follows:

This model, developed by Google, is trained on a large corpus of text data and can be fine-tuned for various NLP tasks, such as question answering and sentiment analysis.

This model, developed by OpenAI, is trained on a massive amount of text data and can be fine-tuned for various language tasks, such as language translation, text generation and summarization.

This model is an optimized version of BERT that uses a larger dataset and other techniques to improve performance.

This model is also developed by Google. It uses a permutation-based architecture that allows it to consider all the context in a given sentence, unlike BERT, which only considers the context to the left of a given word.

This model, also developed by Google, is trained on a diverse set of data sources and tasks and can be fine-tuned for a wide range of language tasks using a simple text-to-text transfer learning framework.

This model, developed by Google, is a lite version of BERT that is faster and requires less memory resources but still maintains good performance.

This model, developed by Microsoft, is an extension of BERT that uses dynamic masked self-attention, which can improve the performance of the model.

These are some most popular Transformer architectures, but new models and variations are being developed and proposed regularly.

Difference between contextual and non-contextual embeddings

Embeddings are a way to represent words, phrases, or other linguistic units in a numerical format that can be used as input to machine learning models. Embeddings can be either contextual or non-contextual:

Contextual embeddings take into account the context in which a word or phrase appears. They assign different embeddings to the same word, depending on the context. For example, the embedding for the word “cat” in the sentence “My cat is sleeping” would be different from the embedding for the same word in the sentence “I saw a cat in the park.”

Non-contextual embeddings, also known as static embeddings, represent a word or phrase with a fixed, pre-trained vector, regardless of the context in which it appears. For example, the embedding for the word “cat” would be the same regardless of whether it appears in the sentence “My cat is sleeping” or in “I saw a cat in the park.”

BERT is a transformer-based model that generates contextual embeddings for words and phrases. It uses a technique called self-attention to analyze the context in which a word appears in a sentence. This allows BERT to generate embeddings that take into account not only that word but also the words that appear before and after it in the sentence. BERT also uses a technique called pre-training, which trains the model on a large corpus of text data before fine-tuning it for a specific task. This allows BERT to generate high-quality embeddings that can be used for various NLP tasks, such as sentiment analysis, question answering, and language translation.

In summary, BERT generates contextual embeddings, which are more informative and accurate than non-contextual embeddings, by using a self-attention mechanism and pre-training on a large corpus of text data. These embeddings are useful for various natural language processing tasks.

BERT Python implementation

In this section, we will discuss an example of how to produce word embeddings using regular methods and BERT.

Using regular methods:

```
from gensim.models import Word2Vec

# Train a Word2Vec model on a dataset

sentences = [['this', 'is', 'the', 'first', 'sentence'], ['this', 'is', 'the', 'second', 'sentence']]

model = Word2Vec(sentences, size=100, window=5, min_count=1, workers=4)

# Get the embedding for the word 'sentence'

word_embedding = model.wv['sentence']
```

Using BERT:

```
from transformers import BertTokenizer, BertModel
```

```
# Load the BERT tokenizer and model

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

model = BertModel.from_pretrained('bert-base-uncased')

# Tokenize the input text

text = "This is an example sentence."

tokenized_text = tokenizer.tokenize(text)

# Convert the tokens to their corresponding IDs

input_ids = tokenizer.convert_tokens_to_ids(tokenized_text)

# Add the necessary special tokens

input_ids = [101] + input_ids + [102]

# Create the attention masks



attention_masks = [1] * len(input_ids)

# Convert the input IDs and attention masks to tensors

input_ids = torch.tensor(input_ids).unsqueeze(0)
```

```
attention_masks = torch.tensor(attention_masks).unsqueeze(0)

# Pass the input through the model to get the embeddings

with torch.no_grad():

    last_hidden_states = model(input_ids, attention_masks)[0]

# Get the embedding for the word 'sentence'

word_embedding = last_hidden_states[0]
[tokenized_text.index('sentence')]
```

BERT can produce more informative and accurate embeddings than regular methods like Word2Vec because of its self-attention mechanism, which allows it to take into account the context in which a word appears in a sentence. BERT also uses pre-training, which trains the model on a large corpus of text data before fine-tuning it for a specific task; this makes its embeddings more generalizable. Additionally, BERT uses a transformer-based architecture that allows it to process input sequences in a parallel manner, which makes the training and inference faster.

In summary, BERT generates embeddings that consider the context in which a word appears that leads to better performance on a wide range of natural language processing tasks than regular methods like Word2Vec.

GPT Python implementation

Install the Hugging Face’s transformers library:

```
pip install transformers
```

Load a pre-trained GPT model:

```
from transformers import GPT2Tokenizer, GPT2LMHeadModel

# Load the GPT-2 tokenizer

tokenizer = GPT2Tokenizer.from_pretrained("gpt2")

# Load the GPT-2 model

model = GPT2LMHeadModel.from_pretrained("gpt2")
```

Prepare the input data:

```
# Tokenize the input text

text = "This is an example sentence."

tokenized_text = tokenizer.tokenize(text)
```

# Convert the tokens to their corresponding IDs

input_ids = tokenizer.convert_tokens_to_ids(tokenized_text)

# Add the necessary special tokens

input_ids = [tokenizer.cls_token_id] + input_ids +
[tokenizer.sep_token_id]

# Create the attention masks

attention_masks = [1] * len(input_ids)

Perform the NLP task:

# Perform the language modeling task

outputs = attention_mask=attention_masks)

# Get the predicted next token

= :]).item())

Fine-tune the model on a specific task:

from transformers import AdamW

```
# Define the loss function and the optimizer

loss_fn = nn.CrossEntropyLoss()

optimizer = AdamW(model.parameters(), lr=2e-5, eps=1e-8)

# Fine-tune the model on the task-specific dataset

for _ in trange(epochs, desc="Epoch"):

# Set the model to training mode

 # Training loop

 for step, batch in enumerate(train_dataloader):

# Add batch to GPU

    batch = tuple(t.to(device) for t in batch)
```

```
# Unpack the inputs from our dataloader

    b_input_ids, b_attention_masks, b_labels = batch

# Clear out the gradients (by default they accumulate)

    optimizer.zero_grad()

  # Forward pass

    outputs = model(b_input_ids, attention_mask=b_attention_masks, labels=b_labels)

    loss = outputs[0]

# Backward pass



    loss.backward()

# Update parameters and take a step using the computed gradient

    optimizer.step()
```

It is important to note that in this example, you should replace train_dataloader and epochs with the appropriate values for your dataset. Also, it is recommended to use the DistributedDataParallel wrapper for multiple gpu support or DataParallel for single gpu support to train the model.

Conclusion

In conclusion, Transformer architectures have revolutionized the field of NLP and other domains like computer vision, speech recognition, and data science. Transformer-based models like BERT and GPT have shown to achieve state-of-the-art performance on a wide range of NLP tasks, such as sentiment analysis, language translation, text generation, summarization, and question answering.

Overall, Transformer-based models have proven to be highly effective for a wide range of NLP tasks. BERT and GPT are some of the most popular models among them; they are pre-trained models that can be fine-tuned on specific NLP tasks. These models' ability to consider the context of a word in a sentence, thanks to their self-attention mechanism, makes them highly powerful. The Hugging Face's transformers library provides an easy-to-use interface to fine-tune and use these models. With the growing amount of textual data, the use of Transformer models will continue to increase in the industry, and they will play a significant role in the development of advanced NLP applications. In addition, with the ongoing research on transformer-based models, we can expect to see even more powerful models in the future.

These models are widely adopted in the industry for various NLP applications. With the advancements in language models and Transformer architectures, we can expect to see more exciting NLP applications in the future.

Questions

What is the main advantage of transformer-based models over traditional models?

They take into account the context of a word in a sentence

They are pre-trained on a large corpus of data

They use self-attention mechanisms

All of the above

BERT is a transformer-based model that generates which of the following?

Non-contextual embeddings

Contextual embeddings

Static embeddings

None of the above

What is the main difference between BERT and GPT?

BERT generates contextual embeddings, while GPT generates non-contextual embeddings

BERT is fine-tuned on specific NLP tasks, while GPT is pre-trained

BERT is pre-trained on a large corpus of text data, while GPT is fine-tuned on specific NLP tasks

None of the above

How can we fine-tune a pre-trained transformer-based model on a specific task?

By using a pre-defined loss function and optimizer

By using pre-trained weights as a starting point and training on the specific task

By using a pre-defined dataset

All of the above

What is the purpose of the self-attention mechanism in transformer-based models?

To take into account the context in which a word appears in a sentence

To pre-train the model on a large corpus of text data

To fine-tune the model on a specific task

None of the above

Answers

d

b

c

b

a

Index