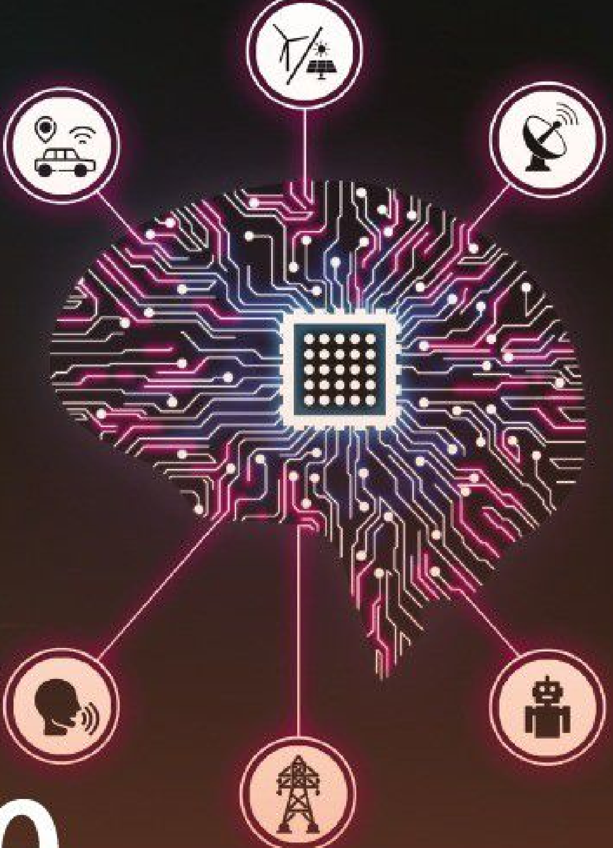
Eklas Hossain



Machine Learning Crash Course for Engineers

Springer

Machine Learning Crash Course for Engineers

Eklas Hossain

Machine Learning Crash Course for Engineers

Springer

Eklas Hossain
Department of Electrical and Computer
Engineering
Boise State University
Boise, ID, USA

ISBN 978-3-031-46989-3   ISBN 978-3-031-46990-9 (eBook)
https://doi.org/10.1007/978-3-031-46990-9

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

*To my beloved wife, **Jenny**, for her unconditional love and support*

Preface

I am sure there is not a single soul who has not heard of machine learning. Thanks to its versatility and widespread applicability, machine learning is a highly demanding topic today. It is basically the ability of machines to learn things and adapt. Since machines do not have the inherent ability to do that, humans have to devise ways or algorithms for them. Here, humans enable machines to learn patterns independently and perform our tasks with little to zero human intervention.

Engineers have to design and build prototypes, forecast future demands and prices, simulate the performance of the designs, and perform many other tasks. Machine learning can significantly help in these works by making them much easier, faster, and free from human errors. There is no field where machine learning has not penetrated yet; it has pervaded well beyond scientific and engineering applications in STEM fields. From entertainment to space research, machine learning and artificial intelligence are ubiquitous today, directly or indirectly, in almost every sector of our daily life. However, as machine learning is a very recent technological upgrade, there is a dearth of suitable resources that can help people use it in practical applications. In fact, when supervising multidisciplinary students as a faculty member, I also find it challenging to guide them in their work due to the lack of a resource that delivers exactly what is needed. Realizing the importance of a basic understanding of machine learning for engineers and the difficulties faced by many students in terms of finding suitable resources, I felt the necessity to compile my experience with machine learning into a book.

Why This Book?

There are numerous books on machine learning out there. So why do we need another one in this field? What does this book offer that other similar books do not have? A reader has the right to ask such questions. Furthermore, as an author, I am obliged to make my stand clear about writing a crash course book on machine learning.

This book is a crash course, delivering only what engineers need to know about machine learning to apply in their specialized domains. The book focuses more on applications and slightly less on the theory to cater to the needs of people in non-programming majors. The first three chapters are concept-building chapters, where the basics are discussed with relevant programming examples. In my opinion, the true beauty of this book lies in the following three chapters, where the applications of machine learning in *signal processing, energy systems, and robotics* are elaborately discussed with some interesting examples. The final chapter talks about state-of-the-art technologies in machine learning and artificial intelligence and their endless future possibilities.

Overall, this book delivers a complete machine learning package for engineers, covering common concepts such as convolutional neural networks (CNN), long short-term memory (LSTM), natural language processing (NLP), load forecasting, generative adversarial networks (GAN), transfer learning, and many more. Students are welcome to reuse, reproduce, and utilize the programming examples in this book for their projects, theses, and any other work related to academic and research activities. I believe this book can help students develop ideas, formulate projects, and draw conclusions from their research. If students find this book helpful, only then can I consider this book a successful endeavor.

If anyone wants to learn machine learning from scratch, this book can only be a supporting tool for them. However, to learn basic programming and algorithmic concepts, one should primarily refer to a more fundamental book that covers detailed theories. Nevertheless, this book is for students, researchers, academicians, and industry professionals who would like to study the various practical aspects of machine learning and keep themselves updated on the advances in this field. Beginners might require a significant amount of time to understand the book. If the readers have a clear concept of the basics of mathematics, logic, engineering, and programming, they will enjoy the book more and understand it better. The intermediate-level readers will require less time, and advanced readers can complete the book in a week or so. A long time of about three years has been invested into the making of this book. As such, some old datasets have been used, but the information on the latest technologies, such as ChatGPT, EfficientNet, YOLO, etc., is included in the book.

I have added references wherever applicable and guided the readers to more valuable references for further reading. This book will be useful material for engineers and researchers starting out their machine learning endeavors. The link to the GitHub repository with the codes used in this book is: https://github.com/eklashossain/Machine-Learning-Crash-Course-for-Engineers.

Key features of this book:

- Development of an overall concept of machine learning sufficient to implement the algorithms in practical applications
- A simplistic approach to the machine learning models and algorithms designed to pique the beginners
- Application of machine learning in signal processing, energy systems, and robotics
- Worked out mathematical examples, programming examples, illustrations, and relevant exercises to build the concepts step-by-step

- Examples of machine learning applications in image processing, object detection, solar energy, wind energy forecast, reactive power control and power factor correction, line follower robot, and lots more

Boise, ID, USA Eklas Hossain
2023

Acknowledgments

I am indebted to the works of all the amazing authors of wonderful books, journals, and online resources that were studied to ideate this book, for providing invaluable information to educate me.

I express my deepest appreciation for Michael McCabe and everyone else at Springer who are involved with the production of this book. Had they not been there with their kindness and constant support, it would have been quite difficult to publish this book.

I am thankful to all my colleagues at Boise State University for their fraternity and friendship and their support for me in all academic and professional affairs, believing in me, and cheering for me in all my achievements.

This book has been written over a long period of three years and by the collective effort of many people that I must acknowledge and heartily thank. I want to especially mention *Dr. Adnan Siraj Rakin*, Assistant Professor in the Department of Computer Science at Binghamton University (SUNY), USA, for extensively planning the contents with me and helping me write this book. I would also like to mention *S M Sakeeb Alam, Ashraf Ul Islam Shihab, Nafiu Nawar, Shetu Mohanto, and Noushin Gauhar*—alumni from my alma mater Khulna University of Engineering and Technology, Bangladesh—for dedicatedly, patiently, and perseverantly contributing to the making of this book from its initiation to its publication. I would also like to heartily thank and appreciate *Nurur Raiyana*, an architect from American International University-Bangladesh, for creatively and beautifully designing the cover illustration of the book. I wish all of them the very best in their individual careers, and I believe that they will all shine bright in life.

I am also indebted to my loving family for their constant support, understanding, and tolerance toward my busy life in academia, and for being the inspiration for me to work harder and keep moving ahead in life.

Lastly, I am supremely grateful to the Almighty Creator for boundlessly blessing me with everything I have in my life.

Contents

About the Author

Eklas Hossain received his PhD in 2016 from the College of Engineering and Applied Science at the University of Wisconsin Milwaukee (UWM). He received his MS in Mechatronics and Robotics Engineering from the International Islamic University Malaysia, Malaysia, in 2010, and a BS in Electrical and Electronic Engineering from Khulna University of Engineering and Technology, Bangladesh, in 2006. He has been an IEEE Member since 2009, and an IEEE Senior Member since 2017.

He is an Associate Professor in the Department of Electrical and Computer Engineering at Boise State University, Idaho, USA, and a registered Professional Engineer (PE) (license number 93558PE) in the state of Oregon, USA. As the director of the iPower research laboratory, he has been actively working in the area of electrical power systems and power electronics and has published many research papers and posters. In addition, he has served as an Associate Editor for multiple reputed journals over time. He is the recipient of an NSF MRI award (Award Abstract # 2320619) in 2023 for the *"Acquisition of a Digital Real-Time Simulator to Enhance Research and Student Research Training in Next-Generation Engineering and Computer Science"*, and a CAES Collaboration fund with Idaho National Laboratory (INL) on *"Overcoming Barriers to Marine Hydrokinetic (MHK) Energy Harvesting in Offshore Environments"*.

His research interests include power system studies, encompassing the utility grid, microgrid, smart grid, renewable energy, energy storage systems, etc., and power electronics, which spans various converter and inverter topologies and control systems. He has worked on several research projects related to machine learning, big data, and deep learning applications in the field of power systems, which include load forecasting, renewable energy systems, and smart grids. With his dedicated research team and a group of PhD students, Dr. Hossain looks forward to exploring methods to make electric power systems more sustainable, cost-effective, and secure through extensive research and analysis on grid resilience, renewable energy systems, second-life batteries, marine and hydrokinetic systems, and machine learning applications in renewable energy systems, power electronics, and climate change effect mitigation.

A list of books by Eklas Hossain:

1. *The Sun, Energy, and Climate Change*, 2023
2. *MATLAB and Simulink Crash Course for Engineers*, 2022
3. *Excel Crash Course for Engineers*, 2021
4. *Photovoltaic Systems: Fundamentals and Applications* (co-author), 2021
5. *Renewable Energy Crash Course: A Concise Introduction* (co-author), 2021

1 Introduction to Machine Learning

1.1 Introduction

Machine learning may seem intimidating to those who are new to this field. This ice-breaker chapter is intended to acquaint the reader with the fundamentals of machine learning and make them realize what a wonderful invention this subject is. The chapter draws out the preliminary concepts of machine learning and lays out the fundamentals of learning advanced concepts. First, the basic idea of machine learning and some insights into artificial intelligence and deep learning are covered. Second, the gradual evolution of machine learning throughout history is explored in chronological order from 1940 to the present. Then, the motivation, purpose, and importance of machine learning are described in light of some practical applications of machine learning in real life. After that, the prerequisite knowledge to master machine learning is introduced to ensure that the readers know what they need to know before their course on machine learning. The next section discusses the programming languages and associated tools required to use machine learning. *This book uses Python as the programming language, Visual Studio Code as the code editor or compiler, and Anaconda as the platform for machine learning applications.* This chapter presents the reasons for choosing these three among the other alternatives. Before the conclusion, the chapter finally demonstrates some real-life examples of machine learning that all engineering readers will be able to relate to, thus instigating their curiosity to enter the world of machine learning.

1.2 What Is Machine Learning?

Modern technology is getting smarter and faster with extensive and continuous research, experimentation, and developments. For instance, machines are becoming intelligent and performing tasks much more efficiently. Artificial intelligence has been evolving at an unprecedented rate, whose results are apparent as our phones and computers are becoming more multi-functional each day, automation systems are becoming ubiquitous, intelligent robots are being built, and whatnot.

In 1959, the computer scientist and machine learning pioneer Arthur Samuel defined machine learning as the "*field of study that gives computers the ability to learn without being explicitly programmed*" [1]. Tom Mitchell's 1997 book on machine learning defined machine learning as "*the study of computer algorithms that allows computer programs to automatically improve through experience*" [2]. He defines *learning* as follows: "*A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*"

Machine learning (ML) is a branch of artificial intelligence (AI) that allows computers and machines to learn from existing information and apply that learning to perform other similar tasks. Without explicit programming, the machine learns from the data it feeds. The machine picks up or learns patterns, trends, or essential features from previous data and makes a prediction on new data. A real-life machine learning application is in recommendation systems. For instance, a movie streaming site will recommend movies to the user based on their previously watched movie list.

ML algorithms are broadly classified as *supervised learning* and *unsupervised learning*, with some other types as well, such as *reinforcement learning* and *semi-supervised learning*. These will be discussed in detail in Chap. 3.

1.2.1 Machine Learning Workflow

Before going into the details, a beginner must have a holistic view of the entire workflow of ML. The bigger picture of the process reveals that there are four main steps in a typical ML workflow: dataset collection, data preprocessing, training model, and finally, model evaluation. Figure 1.1 portrays the block diagram of the four steps of the ML workflow. These steps are generally followed in all ML applications:

1. **Dataset Collection:** The initial step of ML is collecting the dataset. This step depends on the type of experiments or projects one wants to conduct. Different experiments or projects demand different data. One also has to decide the type of data to be required. Should it be numerical or categorical data? For instance, if

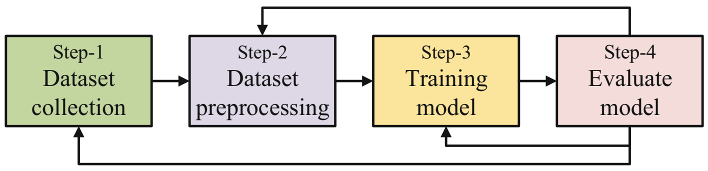
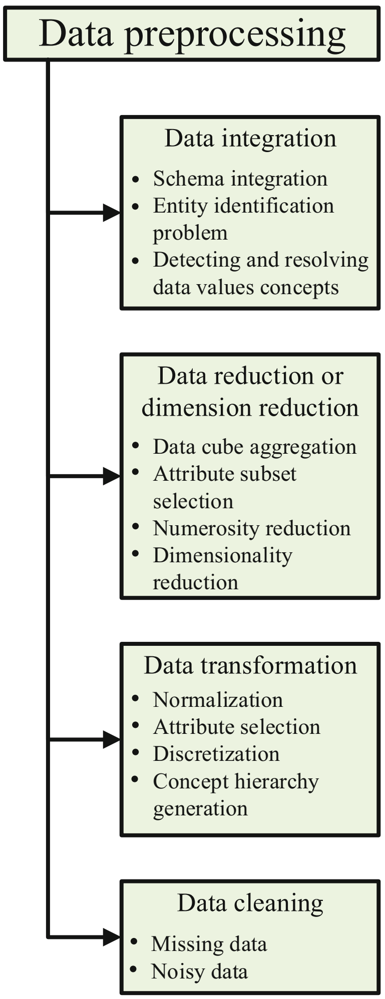


Fig. 1.1 The block diagram of the machine learning workflow

we want to perform a prediction on house prices, we would require the following information: the price of houses, the address of the houses, room numbers, the condition of the house, house size, etc. Then the question also arises—what should the price unit be? Should it be in dollars or pounds or some other currency?

2. **Data Preprocessing:** The data we gather are often unorganized and cannot be directly used to train models. Before proceeding to the next step, the data need to be preprocessed. First, the dataset may contain missing or noisy data. This problem needs to be resolved before passing data to the model. Different data can be in different ranges, which might be an issue for the models, so the data need to be standardized so all data can be in the same range. Again, not all data would be equally crucial for predicting the target variable. We need to find and select the data that contribute more to finding target variables. Finally, the dataset must be split into two sets—the training and test sets. The split is usually done in an 80:20 ratio, where 80% of the dataset is the *train set* and the 20% set is the *test set*. This ratio may vary based on the dataset size and problem nature. Here, the train set will be used to train the models, and the test set will be used to evaluate the models. Oftentimes the dataset is split into the train, validation, and test sets. This validation set is used for hyperparameter tuning, which is discussed in Chap. 2 of this book. Figure 1.2 shows the different data preprocessing steps. These steps will be explained in Chap. 3.
3. **Train Model:** Based on the problem, the type of model required should be selected first. While selecting a model, the information available on the dataset should be considered. For instance, supervised classification can be approached if the dataset contains information on both the input and output values. Sometimes more than one model must be used to train and do the job. Performing numerical analysis, the model fits or learns the data. This step is very crucial because the performance of the model highly relies on how well the data have been fitted or learned by the model. While training the model, one should be mindful of not underfitting or overfitting the model. Underfitting and overfitting have been explained in Chap. 2.
4. **Model Evaluation:** Once the model is built and trained, it is essential to understand how well the model has been trained, how well it will perform, and if the model will be useful for the experiment. It will be useless if the model does not work well or serve its purpose. Therefore, the test dataset is used to test the model, and different evaluation metrics are used to evaluate and understand the model. The evaluation metrics include accuracy, precision, recall, and a few others, which are used to get an insight into how well the model will perform. The evaluation metrics have been discussed in Chap. 2. Based on the evaluation of the model, it might require going back to previous steps and redoing them again accordingly.

Fig. 1.2 The block diagram of data preprocessing



1.2.2 What Is Not Machine Learning?

Machine learning is a buzzword in today's world. Almost all fields of science and technology involve one or more aspects of ML. However, it is necessary to distinguish between what is ML and what is not. For example, programming in the general sense is not ML, since a program explicitly tells or instructs a machine what to do and when to do it without enabling the machine to learn by itself and implement the learning in a new but similar environment. A recommendation system that is explicitly designed to give recommendations is not an application of ML.

If the system is designed in a way where a specific set of movies are given as conditions and suggestion is also explicitly given, such as:

> *If the person has watched Harry Potter or Pirates of the Caribbean or The Lord of the Rings, then recommend Fantastic Beasts to the user. Also, if the person has watched Divergent or Maze Runner, recommend Hunger Games.*

This recommendation system is not an ML application. Here, the machine does not explore or learn trends, features, or characteristics from previously watched movies. Instead, it simply relies on the given conditions and suggests the given movie.

For an ML-based recommendation system, the program does not explicitly tell the system which movie to recommend based on the previously watched list. Instead, it is programmed so that the system explores the previously watched movie list. It looks for significant features or characteristics such as genres, actors, directors, producers, etc. It also checks which movies were watched by other users so the system can form a type of group. Based on this learning and observation, the system concludes and gives a recommendation. For example, a watchlist of a person goes like this: Sully, Catch Me If You Can, Captain Philips, Inception, and Interstellar. One can gather the following insights from this list:

- Three of the movies are of the biographical genre; the other two are science fiction.
- Three of the movies have Tom Hanks in them.
- The science fiction movies on the list are directed by Christopher Nolan.

Based on this pattern, the system may recommend biographical movies that do not include Tom Hanks. The system will also recommend more Tom Hanks movies that can be biographical or other genres. It may also recommend science fiction movies that Tom Hanks has starred in. The system will also recommend more movies directed by Christopher Nolan. As this recommendation system decides by learning the patterns from the watchlist, it will be regarded as an ML application.

1.2.3 Machine Learning Jargon

While going through the book, we will get many terminologies related to machine learning. Therefore, it is essential that we understand this jargon. The terminologies that we need to understand are discussed in this section.

1.2.3.1 Features

Features, also known as attributes, predictor variables, or independent variables, are simply the characteristics or labels of the dataset. Suppose we have information on the height and the mass of sixty students in a class. The height and the mass are known as the features within the dataset. These features are extracted from the raw dataset and are fed to models as inputs.

1.2.3.2 Target Variable

Simply put, target variables are the outputs that the models should give. For instance, a movie review must be classified as positive or negative. Here, the variable positive/negative is the target variable in this case. First, this target variable needs to be determined by the user. Then, after the target variable is determined, the relationship between the features and the target variable must be understood to perform further operations.

1.2.3.3 Optimization Problem

Optimization problems are defined as a class of problems that seek the optimal solution under a set of given conditions. Such problems usually involve a trade-off between different conditions. For example, a battery has to be purchased for power backup at a residence, but we are unsure about the right battery size, which comes in 6.4 and 13.5 kWh. If we buy the larger size, we can store more energy and also enjoy a myriad of additional features of the battery, but we also need to pay more. If we buy the smaller size, we can store less energy and get little to no extra features, but we save more money. We need to optimize our needs in this scenario. If we only require power backup without any special requirement for the added features, the smaller size will suffice the need. This would be the optimal solution to the battery dilemma.

1.2.3.4 Objective Function

Usually, more than one solution exists for a problem. Among all the solutions, finding the optimal solution is required, which is usually done by measuring a quantity and requiring it to meet a standard. The objective function is the standard that the optimal solution needs to meet. The objective function is designed to take parameters and evaluate the performance of the solution. The goal of the objective function may vary with the problem under consideration. Maximizing or minimizing a particular parameter might be needed to qualify the solution as optimal. For example, many ML algorithms use a distance measure as an objective function. A problem may require minimizing the Euclidean distance between each data point. Different metrics are used to calculate distance, such as Euclidean, Manhattan, or Minkowski distance, discussed later in Chap. 2.

1.2.3.5 Cost Function

The cost function is used to understand how well the model performs on a given dataset. The cost function also calculates the difference between the predicted and ground output values. Therefore, the cost and loss functions may seem to be the same. However, the loss function is calculated for a single data point after a single training, and the cost function is calculated for a given dataset after the model training is done. Hence, it can be implied that *the cost function is the average loss function for the entire dataset after the model training*. The terms loss function and cost function are often used interchangeably in ML. Similar to loss functions, different kinds of cost functions are used in different contexts in different ML algorithms.

Suppose J is a cost function used to evaluate a model's performance. It is generally defined with the loss function L. The generalized form of the cost function J is given below:

$$J(\theta) = \sum_{i=1}^{m} L(h_\theta(x^i), y^i), \tag{1.1}$$

where θ is a parameter being optimized, m is the number of training samples, i is the number of examples and outputs, h is a hypothesis function of the model, x is the estimated predicted value, and y is the ground truth value.

1.2.3.6 Loss Function

Suppose a function $L : (z, y) \in \mathbb{R} \times Y \longrightarrow L(z, y) \in \mathbb{R}$ is given. Function L takes z as inputs, where z is the predicted value given by an ML model. The function then compares z with respect to its corresponding real value y and outputs a value that signifies the difference between the predicted value and the real value. This function is known as a *loss function*.

The loss function is significant because it clearly explains how the models perform in modeling the data they are being fed. The loss function calculates the error, which is the difference between the predicted output and ground output values. Therefore, it is intuitive that a lower value of the loss function indicates a lower error value, implying that the model has learned or fitted the data well. While learning the data, the goal of model training is always to lower the value of the loss function. After every iteration of training, the model keeps making necessary changes based on the value of the current loss function to minimize it. Different kinds of loss functions are used for different ML algorithms, which will be discussed later in detail in Chap. 2.

1.2.3.7 Comparison Between Loss Function, Cost Function, and Objective Function

Both the loss function and cost function represent the error value, i.e., the difference between the output value and the real value, to determine how well an ML model performs to fit the data. However, the difference between the loss and cost functions is that the loss function measures the error for a single data point only, while the cost function measures the error for the whole dataset. The cost function is usually the sum of the loss function and some penalty. On the other hand, the objective function is a function that needs to be optimized, i.e., either maximized or minimized, to obtain the desired objective. The loss function is a part of the cost function; concurrently, the cost function can be used as a part of the objective function.

1.2.3.8 Algorithm, Model/Hypothesis, and Technique

As a beginner, it is essential to be able to differentiate between ML models and ML algorithms. An algorithm in ML is the step-by-step instruction provided in the form of code and run on a particular dataset. This algorithm is analogous to a general programming code. For example, finding the arithmetic average of a set of numbers. Similarly, in ML, an algorithm can be applied to learn the statistics of a given data or apply current statistics to predict any future data.

On the other hand, an ML model can be depicted as a set of parameters that is learned based on given data. For example, assume a function $f(x) = x\theta$, where θ is the parameter of the given function, and x is input. Thus, for a given input x, the output depends on the function parameter θ. Similarly, in ML, the input x is labeled as the input feature, and θ is defined as an ML model parameter. The goal of any ML algorithm is to learn the set of parameters of a given model. In some cases, the model is also referred to as a *hypothesis*. Suppose the hypothesis or model is denoted by h_θ. If we input data $x(i)$ to the model, the predicted output will be $h_\theta(x(i))$.

In contrast, an ML technique can be viewed as a general approach in attempting to solve a problem at hand. In many cases, one may require combining a wide range of algorithms to come up with a technique to solve an ML problem.

1.2.4 Difference Between Data Science, Machine Learning, Artificial Intelligence, Deep Learning

Data science, artificial intelligence, machine learning, and deep learning are closely related terminologies, and people often tend to confuse them or use these terms alternatively. However, these are distinctly separate fields of technology. Machine learning falls within the subset of artificial intelligence, while deep learning is considered to fall within the subset of machine learning, as is demonstrated by Fig. 1.3.

The difference between ML and deep learning is in the fact that deep learning requires more computing resources and very large datasets. Thanks to the advancement of hardware computing resources, people are shifting toward deep learning approaches to solve similar problems which ML can solve. Deep learning is especially helpful in handling large volumes of text or images.

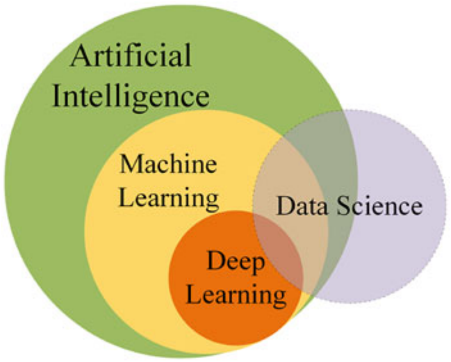


Fig. 1.3 Deep learning falls within the subset of machine learning, and machine learning falls within the subset of artificial intelligence. Data science involves a part of all these three fields

Table 1.1 Sample data of house prices

Year	Price
2001	$200
2002	$400
2003	$800

Data science is an interdisciplinary field that involves identifying data patterns and making inferences, predictions, or insights from the data. Data science is closely related to deep learning, data mining, and big data. Here, data mining is the field that deals with identifying patterns and extracting information from big datasets using techniques that combine ML, statistics, and database systems, and by definition, big data refers to vast and complex data that are too huge to be processed by traditional systems using traditional algorithms. ML is one of the primary tools used to aid the data analysis process in data science, particularly for making extrapolations or predictions on future data trends.

For example, predicting the market price of houses in the next year is an ML application. Consider a sample dataset as given in Table 1.1.

The observation of the data in Table 1.1 allows us to form the intuition that the next price in 2004 will be $1600. This intuition is formed based on the previous years' house prices, which shows a clear trend of doubling the price each year.

However, for large and complex datasets, this prediction may not be so straightforward. Then we require an ML prediction model to predict the prices of houses. Given enough computing resources, these problems can be solved using deep learning models categorized under deep learning. In general, machine learning and deep learning fall within artificial intelligence, but they all require processing, preparing, and cleaning available data; thus, data science is an integral part of all these three branches.

1.3 Historical Development of Machine Learning

Machine learning has been under development since the 1940s. It is not the brainchild of one ingenious human, nor is it the result of a particular event. The multifaceted science of ML has been shaped by years of studies and research and by the dedicated efforts of numerous scientists, engineers, mathematicians, programmers, researchers, and students. ML is a continuously progressing field of science, and it is still under development. Table 1.2 enlists the most significant milestones marked in the history of the development of ML. Do not be frightened if you do not yet know some of the terms mentioned in the table. We will explore them in later chapters.

Table 1.2 Historical development of machine learning

Year	Development
1940s	The paper "A logical calculus of the ideas immanent in nervous activity," co-authored by Walter Pitts and Warren McCulloch in 1943, first discusses the mathematical model of neural networks [3]
1950s	• The term "Machine learning" is first used by Arthur Samuel. He designed a computer checker program that was on par with top-level games • In 1951, Marvin Minsky and Dean Edmonds developed the first artificial neural network consisting of 40 neurons. The neural network had both short-term and long-term memory capabilities • The two-month-long Dartmouth workshop in 1956 first introduces AI and ML research. Many recognize this workshop as the "birthplace of AI"
1960s	• In 1960, Alexey (Oleksii) Ivakhnenko and Valentin Lapa present the hierarchical representation of a neural network. Alexey Ivakhnenko is regarded as the father of deep learning • Thomas Cover and Peter E. Hart published an article on the nearest neighbor algorithms in 1967. These algorithms are now used for regression and classification tasks in machine learning • A project related to intelligent robot, Stanford Cart, was began in this decade. The task was to navigate through 3D space autonomously
1970s	• Kunihiko Fukushima, a Japanese computer scientist, published a work on pattern recognition using hierarchical multilayered neural networks. This is known as neocognition. This work later paved the way for convolutional neural networks • The Stanford Cart project finally became able to traverse through a room full of chairs for five hours without human intervention in 1979
1980s	• In 1985, the artificial neural network named NETtalk is invented by Terrence Sejnowski. NETtalk can simplify models of human cognitive tasks so that machines can potentially learn how to do them • The restricted Boltzmann machine (RBM), initially called Harmonium, invented by Paul Smolensky, was introduced in 1986. It could analyze an input set and learn probability distribution from it. At present, the RBM modified by Geoffrey Hinton is used for topic modeling, AI-powered recommendations, classification, regression, dimensionality reduction, collaborative filtering, etc.
1990s	• Boosting for machine learning is introduced in the paper "The Strength of Weak Learnability," co-authored by Robert Schapire and Yoav Freund in 1990. Boosting algorithm increases the prediction capability of AI models. The algorithm generates and combines many weak models using techniques, such as averaging or voting on the predictions • In 1995, Tin Kam Ho introduced random decision forests in his paper. The algorithm creates multiple decision trees randomly and merges them to create one "forest." The use of multiple decision trees significantly improves the accuracy of the models • In 1997, Christoph Bregler, Michele Covell, and Malcolm Slaney develop world's first "deepfake" software • The year 1997 will be a major milestone in AI. The AI-based chess program, Deep Blue, beat one of the best chess players of human history, Garry Kasparov. This incident shed a new light on AI technology
2000	Igor Aizenberg, a neural networks researcher, first introduces the term "deep learning." He used this term to describe the larger networks consisting of Boolean threshold neurons

(continued)

Table 1.2 (continued)

Year	Development
2009	Fei-Fei Li launched the most extensive dataset of labeled images, ImageNet. It was intended to contribute to providing versatile and real-life training data for AI and ML models. The Economist has commented on ImageNet as an exceptional event for popularizing AI throughout the tech community and stepping toward a new era of deep learning history
2011	Google's X Lab team develops an AI algorithm Google Brain for image processing, which is able to identify cats from images
2014	1. Ian Goodfellow and his colleagues developed generative adversarial networks (GANs). The frameworks are used so that AI models become capable of generating entirely new data given their training set. 2. The research team at Facebook developed DeepFace, which can distinguish human faces almost as human beings do with an accuracy rate of 97.35%. DeepFace is a neural network consisting of nine layers. The network is trained on more than 4 million images taken from Facebook users. 3. Google has started using Sibyl to make predictions for its products. Sibyl is a machine learning system on a broader scale. The system consists of many new algorithms put together. It has significantly improved performance through parallel boosting and column-oriented data. In addition, it uses users' behavior for ranking products and advertising. 4. Eugene Goostman, an AI chatbot developed by three friends from Saint Petersburg in 2001, is considered to be the first AI chatbot to resemble human intelligence. This AI character is portrayed as a 13-year-old boy from Odessa, Ukraine, who has a pet guinea pig and a gynecologist father. Eugene Goostman passed the Turing test competition on 7 June 2014 at the Royal Society
2015	The first AI program "AlphaGo" beats a professional Go player. Go was a game that was initially impossible to teach a computer
2016	A group of scientists presents Face2Face during the Conference on Computer Vision and Pattern Recognition. Most of the "deepfake" software used in the present time is based on the logic and algorithms of Face2Face
2017	1. Autonomous or driverless cars are introduced in the U.S.A. by Waymo. 2. The famous paper "Attention is All You Need" is published, introducing the Transformer architecture based on the self-attention mechanism, which brings about significant progress in natural language processing
2021	Google DeepMind's AlphaFold 2 model places first in the CASP13 protein folding competition in the free modeling (FM) category, bringing a breakthrough in deep-learning-based protein structure prediction
2022	OpenAI and Google revolutionize large language models for mass use. Various applications of machine learning have started becoming part of daily activities

1.4 Why Machine Learning?

Before diving further into the book, one must have a clear view of the purpose and motives behind machine learning. Therefore, the following sections will discuss the purpose, motives, and importance of machine learning so one can implement machine learning in real-life scenarios.

1.4.1 Motivation

The motivation to create a multidimensional field as machine learning rose from the monotonous work humans had to do. With the increased usage of digital communication systems, smart devices, and the Internet, a massive amount of data are generated every moment. Searching and organizing through all those data every time humans need to solve any task is exhaustive, time-consuming, and monotonous. So instead of going through the laborious process of manually going through billions of data, human beings opted for a more automated process. The automated process aims to find relevant patterns in data and later use these patterns to evaluate and solve tasks. This was when the concept of programming took shape. However, even with programming, humans had to explicitly code or instruct the machines what to do, when, and how to do it. To overcome the new problem of coding every command for machines to understand, humans came up with the idea of making machines learn themselves the way humans do—simply by recognizing patterns.

1.4.2 Purpose

The purpose of machine learning is to make machines intelligent and automate tasks that would otherwise be tedious and susceptible to human errors. The use of machine learning models can make the tasks to be performed both more accessible and more time-efficient.

For example, consider a dataset $(x, y) = (0, 0); (1, 1); (2, 2); (3, ?)$. Here, to define the relationship between x and y, y can be expressed as a function of x, i.e., $y = f(x) = \theta x$. Such a representation of the two elements of the dataset is referred to as the model. The purpose of ML is to learn what θ is from the existing data and then apply ML to determine that $\theta = 1$. This knowledge can then be utilized to find out the value of the unknown value of y when $x = 3$. In the next chapters, we will learn how to formulate the hypothetical model $y = \theta x$ and how to solve for the values of θ.

1.4.3 Importance

Just like machines, the science of machine learning was devised with a view to making human tasks easier. Data analysis used to be a tedious and laborious job, prone to many errors when done manually. But thanks to machine learning, all humans have to do is provide the machine with the dataset or the source of the dataset, and the machine can analyze the data, recognize a pattern, and make valuable decisions regarding the data.

Another advantage of ML lies in the fact that humans do not need to tell the machine each step of the work. The machine itself generates the instructions after learning from the input dataset. For example, an image recognition model does not require telling the machine about each object in an image. In the case of supervised learning, we only need to tell the machine about the labels (such as cow or dog) along with their attributes (such as facial proportions, size of the body, size of ears, presence of horns, etc.), and the machine will automatically identify the labeled objects from any image based on the marked attributes.

ML is also essential in case of forecasting unknown or future data trends. This application is extremely valuable for creating business plans and marketing schemes and preparing resources for the future. For example, ML can help predict the future growth of solar module installations, even to as far as 2050 or 2100, based on historical price trends. Compared to other forecasting tools and techniques, ML can predict values with higher accuracy and can consider many additional parameters that cannot be incorporated into the definite forecasting formulae used in traditional forecasting tools, such as statistical data extrapolation.

1.5 Prerequisite Knowledge to Learn Machine Learning

Machine learning is an advanced science; a person cannot just dive into the world of ML without some rudimentary knowledge and skills. To be able to understand the ML concepts, utilize the algorithms, and apply ML techniques in practical cases, a person must be equipped with several subjects in advanced math and science, some of which are discussed in the following subsections.

This section demonstrates only the topics an ML enthusiast must know prior to learning ML. The topics are not covered in detail here. The elaborate discussions with relevant examples may be studied from [4].

1.5.1 Linear Algebra

Linear algebra is the branch of mathematics that deals with linear transformations. These linear transformations are done using linear equations and linear functions. Vectors and matrices are used to notate the necessary linear equations and linear functions. A good foundation in linear algebra is required to understand the more profound intuition behind different ML algorithms. The following section talks about the basic concepts of linear algebra.

1.5.1.1 Linear Equations

Linear equations are mathematically easier to describe and can be combined with non-linear model transformations. There are two properties of an equation to be termed linear—homogeneity and superposition. For modeling linear systems, the knowledge of linear equations can be convenient. An example of a linear equation is $p_1x_1 + p_2x_2 + \cdots + p_nx_n + q = 0$, where $x_1, x_2 \ldots, x_n$ are the variables, $p_1, p_2 \ldots, p_n$ are the coefficients, and q is a constant.

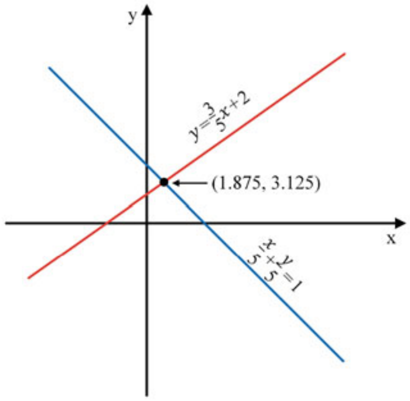


Fig. 1.4 Examples of two straight lines having linear equations

Using linear algebra, we can solve the equations of Fig. 1.4, i.e., we can find the intersection of these two lines. The equations for the two lines are as follows:

$$y = \frac{3}{5}x + 2, \tag{1.2}$$

$$\frac{x}{5} + \frac{y}{5} = 1. \tag{1.3}$$

Now by solving Eq. 1.3, we get

$$\begin{aligned} x + y &= 5, \\ \Rightarrow x + \left(\frac{3}{5}x + 2\right) &= 5, \\ \Rightarrow 8x &= 15, \\ \Rightarrow x &= 1.875. \end{aligned}$$

Putting the value of x in Eq. 1.2, we get $y = 3.125$. So the intersection point is $(x, y) = (1.875, 3.125)$.

1.5.1.2 Tensor and Tensor Rank

A tensor is a general term for vectors and matrices. It is the data structure used in ML models. A tensor may have any dimension. A scalar is a tensor with zero dimensions, a vector is a tensor with one dimension, and a matrix has two dimensions. Any tensor with more than two dimensions is called an n-dimensional tensor. Vectors and matrices are further discussed below.

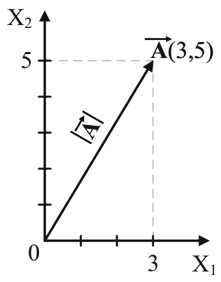


Fig. 1.5 Example of vector $\bar{A}$

1.5.1.2.1 Vector

A vector is a uni-dimensional array of numbers, terms, or elements. The features of the dataset are represented as vectors. A vector can be represented in geometric dimensions. For instance, a vector [3, 5] can be geometrically represented in a 2-dimensional space, as shown in Fig. 1.5. This space can be called a vector space or feature space. In the vector space, a vector can be visualized as a line having direction and magnitude.

1.5.1.2.2 Matrix

A matrix is a 2-dimensional array of scalars with one or more columns and one or more rows. A vector with more than one dimension is called a matrix. The number of rows and columns is expressed as the dimension of that matrix. For example, a matrix with a 4 × 3 dimension contains 4 rows and 3 columns.

Matrix operations provide more efficient computation than piece-wise operations for machine learning models. The two-component matrices must have the same dimension for addition and subtraction. For matrix multiplication, the first matrix's column size and the second matrix's row size must be identical. If a matrix with dimension $m \times n$ is multiplied by a matrix with dimension $n \times p$, then the result of this multiplication will be a matrix with dimension $m \times p$.

Equation 1.4 shows matrix A with a dimension of 2 × 3 and matrix B with a dimension of 3 × 1. Therefore, these two matrices can be multiplied as it fulfills the matrix multiplication condition. The output of the multiplication will be matrix C, shown in Eq. 1.5. It has the dimension of 2 × 1.

$$A = \begin{bmatrix} 1\ 2\ 3 \\ 4\ 5\ 6 \end{bmatrix}; B = \begin{bmatrix} 11 \\ 12 \\ 13 \end{bmatrix}. \tag{1.4}$$

$$Product,\ C = \begin{bmatrix} 74 \\ 182 \end{bmatrix}. \tag{1.5}$$

Some fundamental matrices are frequently used, such as row matrix, square matrix, column matrix, identity matrix, etc. For example, a matrix consisting of only a row is known as a row matrix, and a matrix consisting of only one column is known as a column matrix. A matrix that consists of an equal number of rows and columns is called a square matrix. A square matrix with all 1's along its main diagonal and all 0's in all the non-diagonal elements is an identity matrix. Examples of different matrices are given in Fig. 1.6.

1.5.1.2.3 Rank vs. Dimension

Rank and dimension are two related but distinct terms in linear algebra, albeit they are often used interchangeably in machine learning. In a machine learning perspective, each column of a matrix or a tensor represents each feature or subspace. Hence, the dimension of its column (i.e., subspace) will be the rank of that matrix or tensor.

1.5.1.2.4 Comparison Between Scalar, Vector, Matrix, and Tensor

A scalar is simply a numerical value with no direction assigned to it. A vector is a one-dimensional array of numbers that denotes a specific direction. A matrix is a two-dimensional array of numbers. Finally, a tensor is an *n*-dimensional array of data.

According to the aforesaid quantities, scalars, vectors, and matrices can also be regarded as tensors but limited to 0, 1, and 2 dimensions, respectively. Tables 1.3 and 1.4 summarize the differences in the rank or dimension of these four quantities with examples.

$$\begin{bmatrix} 5 & 2 \\ -6 & 1 \end{bmatrix} \qquad \begin{bmatrix} 4 & 1 \\ 2 & -1 \\ -7 & 5 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Square matrix 2 x 2 | Rectangular matrix 3 x 2 | Zero matrix 3 x 5

$$\begin{bmatrix} 5 & -1 & 0 & 3 \end{bmatrix} \qquad \begin{bmatrix} 1 \\ 2 \\ -7 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Row matrix 1 x 4 | Column matrix 3 x 1 | Identity matrix 3 x 3

Fig. 1.6 Examples of different matrices with their dimensions

Table 1.3 Comparison between scalar, vector, matrix, and tensor

Rank/dimension	Object
0	Scalar
1	Vector
2 or more	$m \times n$ matrix
Any	Tensor

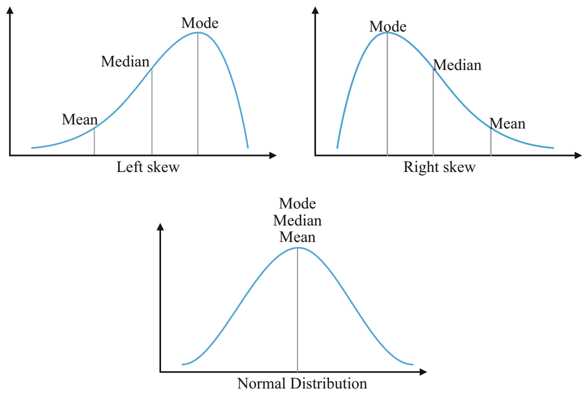
Table 1.4 Examples of scalar, vector, matrix, and tensor

Scalar	Vector	Matrix	Tensor
1	$\begin{bmatrix}1\\2\end{bmatrix}$	$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}$	$\begin{bmatrix}[1\ 2] & [3\ 4]\\ [5\ 6] & [7\ 8]\end{bmatrix}$

Table 1.5 Definition of mean, median, and mode

Name	Definition
Mean	The arithmetic average value
Median	The midpoint value
Mode	The most common value

Fig. 1.7 Graphical representation of mean, median, and mode



1.5.2 Statistics

Statistics is a vast field of mathematics that helps to organize and analyze datasets. Data analysis is much easier, faster, and more accurate when machines do the job. Hence, ML is predominantly used to find patterns within data. Statistics is the core component of ML. Therefore, the knowledge of statistical terms is necessary to fully utilize the benefits of ML.

1.5.2.1 Measures of Central Tendency

The three most common statistical terms used ubiquitously in diverse applications are mean, median, and mode. These three functions are the *measures of central tendency* of any dataset, which denote the central or middle values of the dataset. Table 1.5 lays down the distinctions between these three terms. These are the measures of central tendency of a dataset. Figure 1.7 gives a graphical representation of mean, median, and mode.

Three examples are demonstrated here and in Table 1.6. Let us consider the first dataset: {1, 2, 9, 2, 13, 15}. To find the mean of this dataset, we need to calculate the summation of the numbers. Here, the summation is 42. The dataset has six data points. So the mean of this dataset will be $42 \div 6 = 7$. Next, to find the median of the dataset, the dataset needs to be sorted in ascending order: {1, 2, 2, 9, 13,

Table 1.6 Several datasets and their measures of central tendency

Dataset	{1, 2, 9, 2, 13, 15}	{0, 5, 5, 10}	{18, 22, 24, 24, 25}
Mean	7	5	22.6
Median	5.5	5	24
Mode	2	5	24

15}. Since the number of data points is even, we will take the two mid-values and average them to calculate the median. For this dataset, the median value would be $(2 + 9) \div 2 = 5.5$. For the mode, the most repeated data point has to be considered. Here, 2 is the mode for this dataset. This dataset is left-skewed, i.e., the distribution of the data is longer toward the left or has a long left tail.

Similarly, if we consider the dataset {0, 5, 5, 10}, the mean, median, and mode all are 5. This dataset is normally distributed. Can you calculate the mean, median, and mode for the right-skewed dataset {18, 22, 24, 24, and 25}?

1.5.2.2 Standard Deviation

The standard deviation (SD) is used to measure the estimation of variation of data points in a dataset from the arithmetic mean. A complete dataset is referred to as a population, while a subset of the dataset is known as the sample. The equations to calculate the population's SD and the sample's SD are expressed as Eqs. 1.6 and 1.7, respectively.

$$SD\ for\ population,\ \sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}. \tag{1.6}$$

Here, σ symbolizes the population's SD, i is a variable that enumerates the data points, x_i denotes any particular data point, μ is the arithmetic mean of the population, and N is the total number of data points in the population.

$$SD\ for\ sample,\ s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}. \tag{1.7}$$

Here, s symbolizes the sample's SD, i is a variable that enumerates the data points, x_i denotes any particular data point, $\overline{x}$ is the arithmetic mean of the sample, and N is the total number of data points in the sample.

A low value of SD depicts that the data points are scattered reasonably near the dataset's mean, as shown in Fig. 1.8a. On the contrary, a high value of SD depicts that the data points are scattered far away from the mean of the dataset, covering a wide range as shown in Fig. 1.8b.

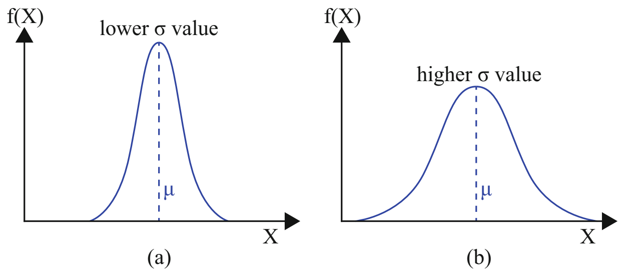


Fig. 1.8 Standard deviation of data

1.5.2.3 Correlation

Correlation shows how strongly two variables are related to each other. It is a statistical measurement of the relationship between two (and sometimes more) variables. For example, if a person can swim, he can probably survive after falling from a boat. However, correlation is not causation. A strong correlation does not always mean a strong relationship between two variables; it could be pure coincidence. A famous example in this regard is the correlation between ice cream sales and shark attacks. There is a strong correlation between ice cream sales and shark attacks, but shark attacks certainly do not occur due to ice cream sales. Correlation can be classified in many ways, as described in the following sections.

1.5.2.3.1 Positive, Negative, and Zero Correlation

In a positive correlation, the direction of change is the same for both variables, i.e., when the value of one variable increases or decreases, the value of the other variable also increases or decreases, respectively. In a negative correlation, the direction of change is opposite for both variables, i.e., when the value of one variable increases, the value of the other variable decreases, and vice versa. For zero correlation, the two variables are independent, i.e., no correlation exists between them. These concepts are elaborately depicted in Fig. 1.9.

1.5.2.3.2 Simple, Partial, and Multiple Correlation

The correlation between two variables is a simple correlation. But if the number of variables is three or more, it is either a partial or multiple correlation. In partial correlation, the correlation between two variables of interest is determined while keeping the other variable constant. For example, the correlation between the amount of food eaten and blood pressure for a specific age group can be regarded as a partial correlation. When the correlation between three or more variables is determined simultaneously, it is called a multiple correlation. For example, the relationship between the amount of food eaten, height, weight, and blood pressure can be regarded as a case of a multiple correlation.

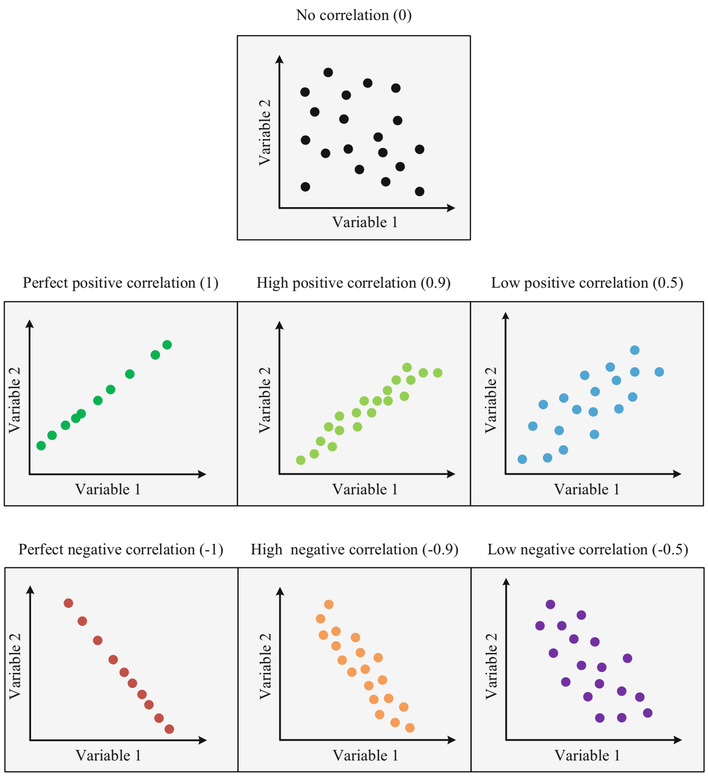


Fig. 1.9 Visualization of zero, positive, and negative correlation at various levels

1.5.2.3.3 Linear and Curvilinear Correlation

When the direction of change is constant at all points for all the variables, the correlation between them is linear. If the direction of change changes, i.e., not constant at all points, then it is known as curvilinear correlation, also known as non-linear correlation. A curvilinear correlation example would be the relationship between customer satisfaction and staff cheerfulness. Staff cheerfulness could improve customer experience, but too much cheerfulness might backfire.

1.5.2.3.4 Correlation Coefficient

The correlation coefficient is used to represent correlation numerically. It indicates the strength of the relationship between variables. There are many types of correlation coefficients. However, the two most used and most essential correlation coefficients are briefly discussed here.

Pearson's Correlation Coefficient

Pearson's Correlation Coefficient, also known as Pearson's r, is the most popular and widely used coefficient to determine the linear correlation between two variables. In other words, it describes the relationship strength between two variables based on the direction of change in the variables.

For the sample correlation coefficient,

$$r_{xy} = \frac{Cov(x, y)}{s_x s_y} = \frac{\frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{n-1}}{\sqrt{\frac{(x_i - \bar{x})^2}{n-1}}\sqrt{\frac{(y_i - \bar{y})^2}{n-1}}} = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2(y_i - \bar{y})^2}. \tag{1.8}$$

Here, r_{xy} = Pearson's sample correlation coefficient between two variables x and y; $Cov(x, y)$ = sample covariance between two variables x and y; $s_x,\ s_y$ = sample standard deviation of x and y; $\bar{x}, \bar{y}$ = average value of x and average value of y; n = the number of data points in x and y.

For the population correlation coefficient,

$$\rho_{xy} = \frac{Cov(x, y)}{\sigma_x \sigma_y} = \frac{\frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{n}}{\sqrt{\frac{(x_i - \bar{x})^2}{n}}\sqrt{\frac{(y_i - \bar{y})^2}{n}}} = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2(y_i - \bar{y})^2}. \tag{1.9}$$

Here, ρ_{xy} = Pearson's population correlation coefficient between two variables x and y; $Cov(x, y)$ = population covariance between two variables x and y; $\sigma_x,\ \sigma_y$ = population standard deviation of x and y; $\bar{x}, \bar{y}$ = average value of x and average value of y and n = the number of data points in x and y.

The value of Pearson's correlation coefficient ranges from -1 to 1. Here, -1 indicates a perfect negative correlation, and the value 1 indicates a perfect positive correlation. A correlation coefficient of 0 means there is no correlation. Pearson's coefficient of correlation is applicable when the data of both variables are from a normal distribution, there is no outlier in the data, and the relationship between the two variables is linear.

Spearman's Rank Correlation Coefficient

Spearman's Correlation Coefficient determines the non-parametric relationship between the ranks of two variables, i.e., the calculation is done between the rankings of the two variables rather than the data themselves. The rankings are usually determined by assigning rank 1 to the smallest data, rank 2 to the subsequent smallest data, and so on up to the largest data. For example, the data contained in a variable are $\{55, 25, 78, 100, 96, 54\}$. Therefore, the rank for that particular variable will be $\{3, 1, 4, 6, 5, 2\}$. By calculating the ranks of both variables, Spearman's rank correlation can be calculated as follows:

$$\rho = 1 - \frac{6\sum d_i^2}{n(n^2 - 1)}. \tag{1.10}$$

Here, ρ = Spearman's rank correlation coefficient, n = the number of data points in the variables, and d_i = rank difference in i-th data.

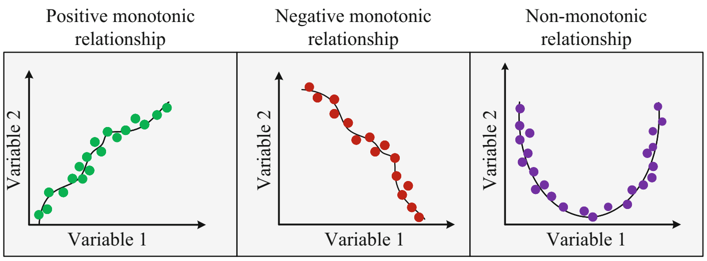


Fig. 1.10 Graphical representation of the monotonicity of the relationship

Pearson's correlation coefficient determines the linearity of the relationship, whereas Spearman's coefficient determines the monotonicity of the relationship. Graphical representation of the monotonicity of the relationship is depicted in Fig. 1.10.

Unlike in a linear relationship, the data change rate is not always the same in a monotonic relationship. If the data change rate is in the same direction for both variables, the relationship is positive monotonic. On the other hand, if the direction is opposite for both variables, the relationship is negative monotonic. The relationship is called non-monotonic when the direction of change is not always the same or opposite but rather a combination.

The value of Spearman's rank correlation coefficient lies between -1 and 1. A value of -1 indicates a perfect negative rank (negative monotonic) correlation, 1 indicates a perfect positive (positive monotonic) rank correlation, and 0 shows no rank correlation. Spearman's rank coefficient is used when one or more conditions of Pearson's coefficient are not fulfilled.

Besides these two correlation coefficients, Cramer's rank correlation coefficient (Cramer's τ), Kendall's v (Kendall's ϕ), point biserial coefficient, etc., are also used. The usage of the different correlation coefficients depends on the application and data type.

1.5.2.4 Outliers

An outlier is a data point in the dataset that holds different properties than all other data points and thus significantly varies from the pattern of other observations. This is the value that has the maximum deviation from the typical pattern followed by all other values in the dataset.

ML algorithms have a high sensitivity to the distribution and range of attribute values. Outliers have a tendency to mislead the algorithm training process, eventually leading to erroneous observations, inaccurate results, longer training times, and poor results.

Consider a dataset $(x, y) = \{(4, 12), (5,10), (6, 9), (8, 7.5). (9, 7), (13, 2), (15, 1), (16, 3), (21, 27.5)\}$. Here, x = water consumption rate per day and y = electricity consumption rate per day. In Fig. 1.11, we can see that these data are distributed in

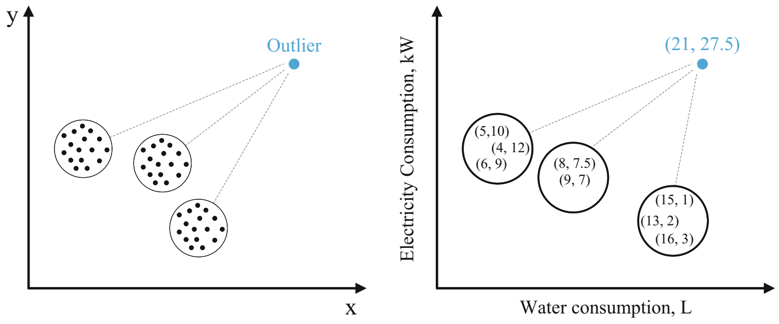


Fig. 1.11 Representation of an outlier. The black dots are enclosed within specific boundaries, but one blue point is beyond those circles of data. The blue point is an outlier

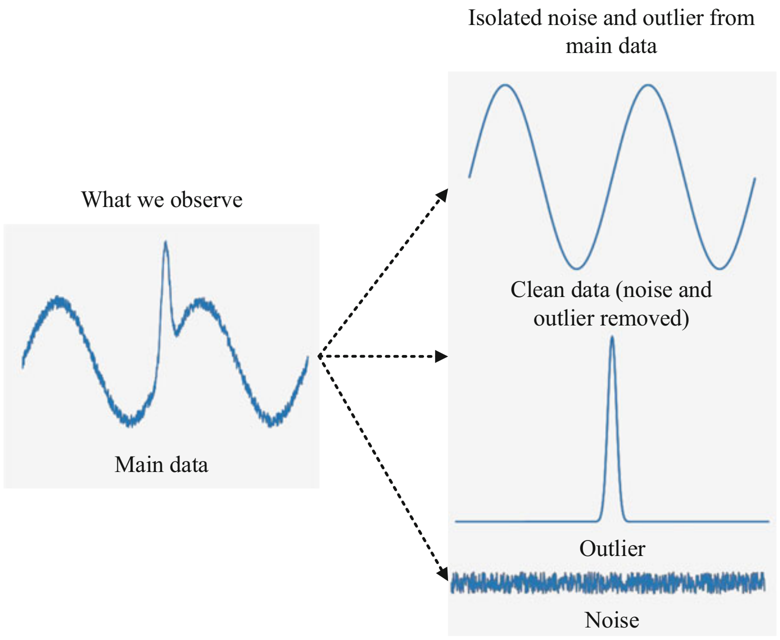


Fig. 1.12 Difference between outlier and noise

3 different groups, but one entry among these data cannot be grouped with any of these groups. This data point is acting as an outlier in this case.

It is to be noted that noise and outliers are two different things. While an outlier is significantly deviated data in the dataset, noise is just some erroneous value. Figure 1.12 visualizes the difference between outlier and noise using a signal.

Consider a list of 100 house prices, which mainly includes prices ranging from 3000 to 5000 dollars. First, there is a house on the list with a price of 20,000 dollars. Then, there is a house on the list with a price of −100 dollars. 20,000 is

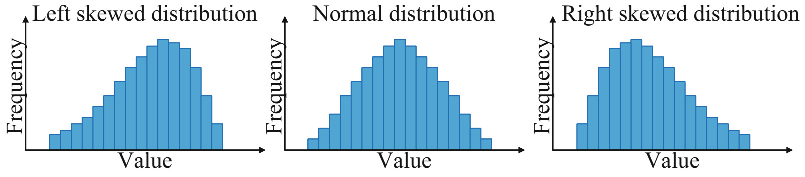


Fig. 1.13 Representation of different distribution of data using histogram

an outlier here as it significantly differs from the other house prices. On the other hand, -100 is a noise as the price of something cannot be a negative value. Since the outlier heavily biases the arithmetic mean of the dataset and leads to erroneous observations, removing outliers from the dataset is the prerequisite to achieving the correct result.

1.5.2.5 Histogram

A histogram resembles a column chart and represents the frequency distribution of data in vertical bars in a 2-dimensional axis system. Histograms have the ability to express data in a structured way, aiding data visualization. The bars in a histogram are placed next to each other with no gaps in between. Histogram assembles the data in bars providing a clear understanding of the data distribution. The arrangement also provides a clear understanding of data distribution according to its features in the dataset.

Figure 1.13 illustrates three types of histograms, with a left-skewed distribution, a normal distribution, and a right-skewed distribution.

1.5.2.6 Errors

The knowledge of errors comes handy when evaluating the accuracy of an ML model. Especially when the trained model is tested against a test dataset, the output of the model is compared with the known output from the test dataset. The deviation of the predicted data from the actual data is known as the error. If the error is within tolerable limits, then the model is ready for use; otherwise, it must be retrained to enhance its accuracy.

There are several ways to estimate the accuracy of the performance of an ML model. Some of the most popular ways are to measure the mean absolute percentage error (MAPE), mean squared error (MSE), mean absolute error (MAE), and root mean squared error (RMSE). In the equations in Table 1.7, n represents the total number of times the iteration occurs, t represents a specific iteration or an instance of the dataset, e_t is the difference between the actual value and the predicted value of the data point, and y_t is the actual value.

The concept of errors is vital for creating an accurate ML model for various purposes. These are described to a deeper extent in Sect. 2.2 in Chap. 2.

Table 1.7 Different types of errors

Name of error	Equation
Mean squared error	$MSE = \frac{1}{n}\sum_{t=1}^{n} e_t^2$
Root mean squared error	$RMSE = \sqrt{\frac{1}{n}\sum_{t=1}^{n} e_t^2}$
Mean absolute error	$MAE = \frac{1}{n}\sum_{t=1}^{n} \lvert e_t \rvert$
Mean absolute percentage error	$MAPE = \frac{100\%}{n}\sum_{t=1}^{n} \left\lvert \frac{e_t}{y_t} \right\rvert$

1.5.3 Probability Theory

Probability is a measure of how likely it is that a specific event will occur. Probability ranges from 0 to 1, where 0 means that the event will never occur and 1 means that the event is sure to occur. The probability is defined as the ratio of the number of desired outcomes to the total number of outcomes.

$$P(A) = \frac{n(A)}{n}, \tag{1.11}$$

where $P(A)$ denotes the probability of an event A, $n(A)$ denotes the number of occurrences of the event A, and n denotes the total number of probable outcomes, also referred to as the sample space.

Let us see a common example. A standard 6-faced die contains one number from 1 to 6 on each of the faces. When a die is rolled, any one of the six numbers will be on the upper face. So, the probability of getting a 6 on the die is ascertained as shown in Eq. 1.12.

$$P(6) = \frac{1}{6} = 0.167 = 16.7\%. \tag{1.12}$$

Probability theory is the field that encompasses mathematics related to probability. Any learning algorithm depends on the probabilistic assumption of data. As ML models deal with data uncertainty, noise, probability distribution, etc., several core ideas of probability theory are necessary, which are covered in this section.

1.5.3.1 Probability Distribution

In probability theory, all the possible numerical outcomes of any experiment are represented by random variables. A probability distribution function outputs the possible numerical values of a random variable within a specific range. Random variables are of two types: discrete and continuous. Therefore, the probability distribution can be categorized into two types based on the type of random variable involved—probability density function and probability mass function.

1.5.3.1.1 Probability Density Function

The possible numerical values of a continuous random variable can be calculated using the probability density function (PDF). The plot of this distribution is continuous. For example, in Fig. 1.14, when a model is looking for the probability of people's height in 160–170 cm range, it could use a PDF in order to indicate the total probability that the continuous random variable range will occur. Here, $f(x)$ is the PDF of the random variable x.

1.5.3.1.2 Probability Mass Function

When a function is implemented to find the possible numerical values of a discrete random variable, the function is then known as a probability mass function (PMF). Discrete random variables have a finite number of values. Therefore, we do not get a continuous curve when the PMF is plotted. For example, if we consider rolling a 6-faced die, we will have a finite number of outcomes as visualized in Fig. 1.15.

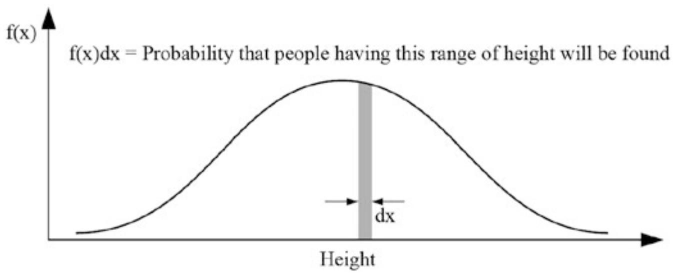


Fig. 1.14 Example of the probability density function

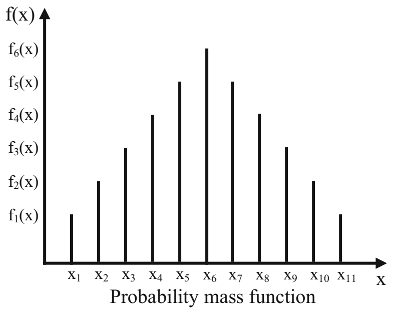


Fig. 1.15 Example of the probability mass function

1.5.3.2 Gaussian or Normal Distribution

The cumulative probability of normal random variables is presented in Gaussian or normal distribution. The graph depends on the mean and the standard distribution of the data. In a standard distribution, the mean of the data is 0, and the standard deviation is 1. A normal distribution graph plot is a bell curve, as depicted in Fig. 1.16. Hence, it is also called a bell curve distribution.

The equation representing the Gaussian or normal distribution is

$$P(x) = \frac{1}{\alpha\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\alpha^2}}. \tag{1.13}$$

Here $P(x)$ denotes the probability density of the normal distribution, α denotes the standard deviation, μ denotes the mean of the dataset, and x denotes a data point.

1.5.3.3 Bernoulli Distribution

A probability distribution on the Bernoulli trial is the Bernoulli distribution. Bernoulli trial is an experiment or event that has only two outcomes. For example, tossing a coin can be regarded as a Bernoulli trial as it can have only two outcomes—*head* or *tail*. Usually, the outcomes are observed in terms of success or failure. In this case, we can say getting a *head* will be a success. On the other hand, not getting a *head* or getting a *tail* would be a failure. The Bernoulli distribution has been visualized in Fig. 1.17, which plots the probability of two trials.

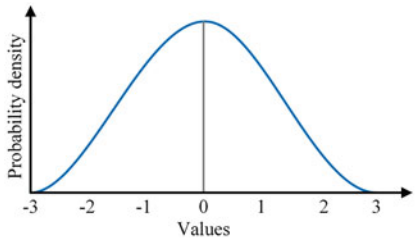


Fig. 1.16 The normal distribution

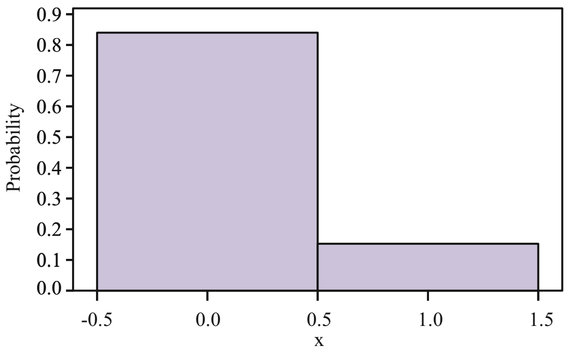


Fig. 1.17 The Bernoulli distribution

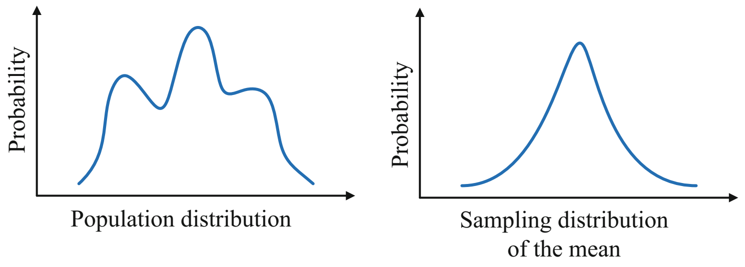


Fig. 1.18 Graphical demonstration of the central limit theorem

1.5.3.4 Central Limit Theorem

Consider a large dataset of any distribution. The central limit theorem states that *irrespective of the distribution of the numbers in the dataset, the arithmetic mean of data samples extracted from the main dataset will have a normal distribution.* The larger the sample size, the closer the mean will be to a normal distribution. The theorem has been demonstrated in Fig. 1.18. It can be seen that the population does not follow a normal distribution, but when the mean is sampled from it, the sampling forms a normal distribution.

1.5.4 Calculus

Newton's calculus is ubiquitously useful in solving a myriad of problems. One of the most popular algorithms of ML is the gradient descent algorithm. The gradient descent algorithm, along with backpropagation, is useful in the training process of ML models, which heavily depend on calculus. Therefore, differential calculus, integral calculus, and differential equations are all necessary aspects to be familiar with prior to studying ML.

1.5.4.1 Derivative and Slope

A derivative can be defined as the rate of change of a function with respect to a variable. For example, the velocity of a car is the derivative of the displacement of the car with respect to time. The derivative is equivalent to the slope of a line at a specific point. The slope helps to visualize how steep a line is. A line with a higher slope is steeper than a line with a lower slope. The concept of slope is depicted in Fig. 1.19.

$$slope, m = \frac{rise}{run} = \frac{\Delta y}{\Delta x}. \tag{1.14}$$

There is wide use of derivatives in ML, particularly in optimization problems, such as gradient descent. For instance, in gradient descent, derivatives are utilized

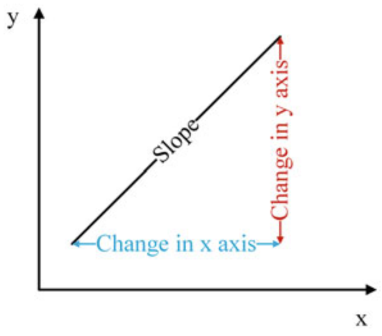


Fig. 1.19 Illustration of the concept of slope

to find the steepest path to maximize or minimize an objective function (e.g., a model's accuracy or error functions).

1.5.4.2 Partial Derivatives

If a function depends on two or more variables, then the partial derivative of the function is its derivative with respect to one of the variables, keeping the other variables constant. Partial derivatives are required for optimization techniques in ML, which use partial derivatives in order to adjust the weights to meet the objective function. Objective functions are different for each problem. So the partial derivative helps to decide whether to increase or decrease the weights to make an adjustment to the objective function.

1.5.4.3 Maxima and Minima

For a non-linear function, the highest peak or the maximum value refers to the maxima, and the lowest peak or the lowest value refers to the minima. In other words, the point at which the derivative of a function is zero is defined as the maxima or the minima. These are the points where the value of the function stays constant, i.e., the rate of change is zero. This concept of maxima and minima is necessary for minimizing the cost function (difference between the ground value and output value) of any ML model.

A *local minima* is the value of a function that is smaller than neighboring points but not necessarily smaller than all other points in the solution space. A *global minima* is the smallest value of the function to exist in that entire solution space. The case is the same for global and local maxima. A *local maxima* is the value of a function larger than neighboring points but not necessarily larger than all other points in the solution space. A *global maxima* is the largest value of the function to exist in that solution space. Figure 1.20 demonstrates global and local maxima and minima in a solution space.

1.5.4.4 Differential Equation

A differential equation (DE) represents the relationship between one or more functions and their derivatives with respect to one or more variables. DEs are

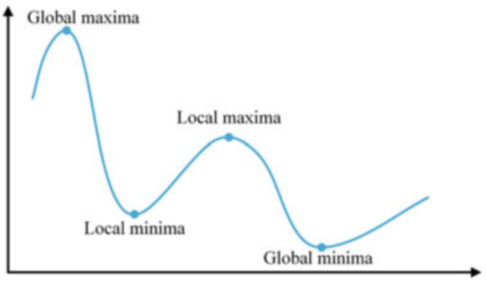


Fig. 1.20 Representation of maxima and minima

Table 1.8 Differential equations with their degree and order

Equation	Order	Degree
$\frac{d^3x}{dx^3} + 6x\frac{dy}{dx} = e^y$	3	1
$\frac{dy}{dx} + \left(\frac{d^2y}{dx^2}\right)^3 = 7\text{x}$	2	3
$\frac{d^2y}{dx^2} + \left(\frac{dy}{dx}\right)^3 = 7\text{x}$	2	1

highly useful in system modeling, and thus, they can be utilized in ML for dynamic modeling, specifically in neural networks.

The following is an example of a differential equation.

$$\frac{d^2y}{dx^2} + 4x = 1. \tag{1.15}$$

1.5.4.4.1 Order and Degree

In differential equations, the highest order of differentiation used in the equation is the order of the equation. The degree of a differential equation is the power of its highest derivative. For example, this is a fourth-order, first-degree differential equation:

$$\frac{d^4y}{dx^4} + \left(\frac{d^2y}{dx^2}\right)^2 + 4\frac{dy}{dx} - 6x = 0. \tag{1.16}$$

Here, the highest derivative is $\frac{d^4y}{dx^4}$. The order of the highest derivative is 4, so this is a fourth-order differential equation. The power of the highest derivative is 1, and therefore, this is a first-degree differential equation. Some more examples are shown in Table 1.8.

1.5.4.4.2 Ordinary and Partial Differential Equation

As discussed earlier, differential equations can have more than one variable. When an equation consists of differentials with respect to one variable, it is an ordinary differential equation (ODE). On the other hand, when the equation involves differentials with respect to more than one variable, they are known as partial differential equations (PDEs). The d symbol and ∂ symbol are used for ordinary differential and partial differential, respectively.

For example: $\frac{d^2y}{dx^2} + \frac{dy}{dx} + 1 = 0$ is an ODE and $\frac{\partial^2 y}{\partial x^2} + \frac{\partial y}{\partial x} + 1 = 0$ is a PDE.

1.5.4.4.3 Linear and Non-linear Equation

Equations can have both dependent and independent variables. These variables can have higher powers depending on the type of equation. When differential equations contain dependent variables with degree 1, they are considered linear differential equations. On the other hand, if the differential equations contain dependent variables with a higher degree, they are regarded as non-linear differential equations.

For example, in the equation $\frac{d^2y}{dx^2}+\frac{dy}{dx}+1 = 0$, the degree of the highest derivative is 1. So it is a linear equation. Again, the equation $\left(\frac{dy}{dx}\right)^2 + x = 0$ has 2 as its degree of highest derivative. So is an example of a non-linear equation.

1.5.5 Numerical Analysis

Numerical analysis uses numerical approximation for mathematical analysis and is a widely deployed tool in ML applications. This branch of mathematics uses specific algorithms and relies on iterations of the algorithm to reach solutions to mathematical problems. Numerical analysis primarily deals with and applies specific algorithms to discrete numerical data. Some parts of numerical analysis, such as function approximation, Fourier transform, numerical linear algebra techniques, compressed sensing, numerical linear algebra, interpolation methods, and optimization, are useful for ML. Without those sorts of tools, building anything that will outperform a human is difficult. Support vector machines use convex optimization, neural networks depend on gradient descent, and most machine learning algorithms use some sort of numerical linear algebra, which is especially important when they are applied to large datasets.

Two prominent methods of numerical analysis are described here with worked-out examples—the Newton–Raphson method and the Gauss–Seidel method.

1.5.5.1 Newton–Raphson Method

Newton–Raphson method, also widely known as Newton's method, is a commonly used algorithm to find the root value of an equation. It successively approximates the root on the foundation that a straight tangent line is used to approximate a continuous and differentiable function. Thus, Newton's method initially guesses a value and starts the approximation from there.

A continuously differentiable function $f(x)$ is given, of which the root needs to be determined. Let us say the initial guess for the root is $x = x_0$. According to the Newton–Raphson method, the formula to approximate the first value of the root will be

$$x_1 = x_0 - \frac{f(x_0)}{f^{'}(x_0)}, \tag{1.17}$$

where x_1 is the first approximation root value, x_0 is the initial guessed root value, $f(x_0)$ is the function, and $f^{'}(x_0)$ is the first derivative.

For n successive iterations, the value for the next approximate root will be

$$x_{n+1} = x_n - \frac{f(x_n)}{f^{'}(x_n)}, \tag{1.18}$$

where x_{n+1} is the next approximation root value, x_n is the current approximated root value, $f(x_n)$ is the function, and $f^{'}(x_n)$ is the first derivative.

In this way, the approximate roots are calculated iteratively until convergence is reached, i.e., the same value of the approximate root is found at least twice. Then, the last value of the approximate root will be the root of the given function. The following example will demonstrate the use of the Newton–Raphson method for solving equations.

Example 1.1 Find the root of the equation $x^3 - 3x - 10 = 0$ using the Newton–Raphson method.

Solution to Example 1.1
Using Eq. 1.18, we get

$$x_{(n+1)} = x_n - \frac{f(x_n)}{f'(x_n))}; \qquad x_{(n+1)} = x_n - \frac{x_n^3 - 3x_n - 10}{3x_n^2 - 3},$$

where

$$f(x) = x^3 - 3x - 10,$$

$$f'(x) = 3x^2 - 3.$$

$$\text{Here,} \quad f(0) = -10, \qquad f(1) = -12,$$

$$f(2) = -8, \qquad f(3) = 8.$$

Here, $f(2)$ resulted a negative number, while $f(3)$ was positive. Any marginal value is a good starting point, so starting with 2 or 3 is a viable option. In this example, the initial value is $x_0 = 3$.

Let us start the iterations now:

$$\begin{aligned}
x_1 &= x_0 - \frac{x_0^3 - 3x_0 - 10}{3x_0^2 - 3}, \\
&= 3 - 0.333333, \\
&= 2.66667. \\
x_2 &= x_1 - \frac{x_1^3 - 3x_1 - 10}{3x_1^2 - 3}, \\
&= 2.66667 - 0.0525253, \\
&= 2.61414. \\
x_3 &= x_2 - \frac{x_2^3 - 3x_2 - 10}{3x_2^2 - 3}, \\
&= 2.61414 - 0.00125285, \\
&= 2.61289. \\
x_4 &= x_3 - \frac{x_3^3 - 3x_3 - 10}{3x_3^2 - 3}, \\
&= 2.61289 - 7.04037 \times 10^{-7}, \\
&= 2.61289. \\
x_5 &= x_4 - \frac{x_4^3 - 3x_4 - 10}{3x_4^2 - 3}, \\
&= 2.61289 - 2.21923 \times 10^{-13}, \\
&= 2.61289.
\end{aligned}$$

As two consecutive values of x are equal, the last value of x is considered the equation's root. So, the root of the given equation $x^3 - 3x - 10 = 0$ is $x = 2.61289$.

1.5.5.2 Gauss–Seidel Method

The Gauss–Seidel method, also known as the Liebmann method, is another popular method for solving linear equations iteratively. This method is also widely known as the *successive displacement method*, which derives from the fact that the subsequent unknown is determined using the immediate previous unknown in the current iteration. The set of linear equations should be arranged in a diagonally dominant form.

Assume the following system of linear equations is given:

$$\begin{aligned}a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1,\\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2,\\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3.\end{aligned} \tag{1.19}$$

For finding the root of the equations, the initial guess is started from $(x_1, x_2, x_3) = (0, 0, 0)$. Then, the given equations are reoriented according to prominent variables in the equation. The initial guess is applied to each of the equations for converging. After each iteration, the values of variables are updated to reach convergence at an early rate. If the desired level of accuracy is met, the iterations stop, resulting in the root of the equations.

Equation 1.19 can also be represented in matrix form as

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}. \tag{1.20}$$

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}. \tag{1.21}$$

$$B = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}. \tag{1.22}$$

So the equation we get in matrix form is

$$AX = B. \tag{1.23}$$

Now, as the matrices A and B are known, the goal is to find the value of the matrix X to find the solution to the system of linear equations. The following example clearly demonstrates the process of solving a set of equations using the Gauss–Seidel method.

Example 1.2 Obtain the solution of the following system using the Gauss–Seidel iteration method.

$$\begin{aligned}20x_1 + x_2 - 2x_3 &= 17,\\ 3x_1 + 20x_2 - x_3 &= -18,\\ 2x_1 - 3x_2 + 20x_3 &= 25.\end{aligned}$$

Solution to Example 1.2
From the given equations, we can write

$$x_1^{k+1} = \frac{1}{20}\left(17 - x_2^k + 2x_3^k\right),$$
$$x_2^{k+1} = \frac{1}{20}\left(-18 - 3x_1^{k+1} + 2x_3^k\right),$$
$$x_3^{k+1} = \frac{1}{20}\left(25 - 2x_1^{k+1} + 3x_2^{k+1}\right),$$

where k represents the number of iterations.

Initial guess $(x_1, x_2, x_3) = (0, 0, 0)$.

1st Approximation

$$x_1^1 = \frac{1}{20}\Big[17 - (0) + 2(0)\Big] = 0.85,$$
$$x_2^1 = \frac{1}{20}\Big[-18 - 3(0.85) + (0)\Big] = -1.0275,$$
$$x_3^1 = \frac{1}{20}\Big[25 - 2(0.85) + 3(-1.0275)\Big] = 1.01088.$$

2nd Approximation

$$x_1^2 = \frac{1}{20}\Big[17 - (-1.0275) + 2(1.01088)\Big] = 1.00246,$$
$$x_2^2 = \frac{1}{20}\Big[-18 - 3(1.00246) + (1.01088)\Big] = -0.99983,$$
$$x_3^2 = \frac{1}{20}\Big[25 - 2(1.00246) + 3(-0.99983)\Big] = 0.99978.$$

3rd Approximation

$$x_1^3 = \frac{1}{20}\Big[17 - (-0.99983) + 2(0.99978)\Big] = 0.99997,$$
$$x_2^3 = \frac{1}{20}\Big[-18 - 3(0.99997) + (0.99978)\Big] = -1.00001,$$
$$x_3^3 = \frac{1}{20}\Big[25 - 2(0.99997) + 3(-1.00001)\Big] = 1.$$

4th Approximation

$$x_1^4 = \frac{1}{20}\Big[17 - (-1.00001) + 2(1)\Big] = 1,$$

$$x_2^4 = \frac{1}{20}\Big[-18 - 3(1) + (1)\Big] = -1,$$

$$x_3^4 = \frac{1}{20}\Big[25 - 2(1) + 3(-1)\Big] = 1.$$

So, the solution to the given system of equations is $(x_1, x_2, x_3) = (1, -1, 1)$.

1.5.6 Gradient Descent

A gradient is a vector that stores the partial derivatives of multi-variable functions, and gradient descent is an optimization algorithm that helps to find the minima of a differentiable function by repetitively moving along the direction of the steepest descent determined by the negative gradient, as shown in Fig. 1.21.

The parameters of ML models must be constantly updated to decrease the cost or error function to achieve the best possible result. This parameter update is made using gradient descent. In addition, gradient descent provides directions (up–down or positive–negative) for the update to be made. Chapter 2 discusses more on this topic.

1.5.7 Activation Functions

An activation function is a deciding function that decides the weight of an input in the output. These functions are frequently used in artificial neural networks and will be explained in Chapter 3. To get more extensive insight into activation functions, you can read this excellent paper by Nwankpa et al. [5], where they provide an in-depth exposition of the different types of activation functions and present their usage in deep learning and neural network applications.

1.5.8 Programming

Machine learning requires a beginner to be acquainted with basic programming skills, with first-hand knowledge of data types, arrays, variables, functions, loops,

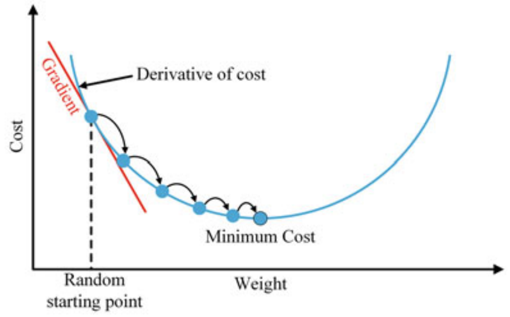


Fig. 1.21 Gradient descent and minimum cost

conditional statements, etc. Furthermore, the algorithms for machine learning are implemented through programming. Therefore, the algorithms can be made more efficient with proficient programming skills and easily tweaked to solve different problems.

1.5.8.1 Variables and Constants

A variable is used to contain information. This information can be of any type, such as numbers, characters, text, logical values, or even an array. A variable has a name to identify it. For example, we want to store the sum of two numbers as a variable. We can name the variable as *sum* and assign it as the sum of the two numbers.

```
sum = 5+9
```

When we write this line in a program, the program will assign a blank memory space to store the sum of 5 and 9.

Variables may be classified as *global* and *local*. Local variables are declared and known only within a specific scope. For instance, it can be declared and used within a function only. It cannot be used outside the function. The variable will be unknown to the program outside its designated scope. On the other hand, a global variable is known even outside the function.

Programming Example 1.1

The difference between global variables and local variables is shown in Listing 1.1, followed by its output and explanation in Table 1.9. The code compares a global variable to a local variable. A variable can be accessed depending on where the variable is declared in the code. The code ends with an error, which is vital in understanding which variable can be accessed and which cannot.

```python
# Declaring global variable
globalvariable = 63

# Building a function
def insidescope():
    localvariable = 63
    print("Inside the scope: ")
    print("This is global variable: ", globalvariable)
    print("This is local variable: ", localvariable)
    return localvariable

# Calling the function
insidescope()

# Printing the variables
print("Outside the scope: ")
print("This is global variable: ", globalvariable)
print("This is local variable: ", localvariable)
```

Listing 1.1 Example of global and local variables in Python

Table 1.9 Explanation of Listing 1.1

Line number	Description
1–2	Creating global variable
4–10	Building a function to understand the scope of local variable
13	Calling the function
15–18	Printing the variables
18	Gives error “NameError: name ‘localvariable’ is not defined”

Output of Listing 1.1:

```
Inside the scope:
This is global variable:  63
This is local variable:  63
Outside the scope:
This is global variable:  63
NameError: name 'localvariable' is not defined
```

The global variable is valid throughout the program, but the local variable only applies within the function. So, when the program was asked to print the local variable outside the function, it returned an error.

A constant is a parameter with a pre-defined value that, unlike a variable, does not change over time. It is initialized when it is declared. The values, such as the value of pi and the acceleration due to gravity, are constants. To indicate that a variable is constant, its name is usually written in uppercase letters.

1.5.8.1.1 Good Practices for Naming Variables

Variables are assigned definite names that are used to refer to the variables throughout the scope of the variable. There are certain rules for naming the variables in all programming languages. Some general rules for naming the variables in a program are delineated below:

- A variable name cannot match a keyword. Keywords are a pre-defined set of words in any programming language with special meanings. For example, the name *print* cannot be used as a variable name because the word “*print*” is a keyword. It refers to the command that tells a program to print something. Different programming languages have different keywords that are unique to them.
- The name cannot start with a numerical value. It should start with a letter or an underscore. For example, *99names* is an invalid name, but *names99* and *_names* are valid variable names.
- Variables are case-sensitive. This means that if we name a variable as *sum*, we cannot use it interchangeably as *Sum*. The uppercase and lowercase letters must be used consistently.

- The name of the variable should be relevant to the purpose of the variable. For example, for storing the sum of five numbers, the preferred name of the variable is *sum* or *sum5*, instead of a single character name such as *s* or a random name or character such as *x*. The relevance of the variable name with its purpose helps in understanding the code better.

In addition, there are many other naming conventions for variables.

1.5.8.2 Data Types

Data types refer to the type of data used in the programs. The data can range from integer numbers, fractional numbers, characters or text, strings, Boolean or logical arguments, and so on. Some common data types are as follows:

- *int*: This data type refers to integer values, such as 2, 19, 400, 50,000, etc.
- *float*: This data type implies fractional values, as well as integers. For example, *float* may refer to 5.5, 22/7, 3.00, etc.
- *str*: This data type includes textual data and is commonly included within ‘ ’ or “ ” marks. For example, ‘city’, ‘987-654-321’, and “This is a dummy line.” are string data types.
- *bool*: This data type has only two values: true or false, also shown as 1 or 0.

Python programming language contains the following built-in data types:

1. **Numeric**: Numeric data are basically integers, floats, or complex numbers in Python. All sorts of operations can be done on numeric data:

    - *Integer*: Integers can range from negative to positive numbers without any fractions, e.g., 25, 3, 985, etc.
    - *Floating Point Number*: Floating point numbers represent fractional numbers, e.g., 3.5, 22/7, etc.
    - *Complex Numbers*: By definition, complex numbers have a real part and an imaginary part, e.g., 2+3j, −3−2j, etc.

2. **String**: Any character or text can be represented with strings. To use any character or text as a string, they are declared using double quotes. Different types of operations can be performed on strings, such as slicing, concatenation, repetition, etc.
3. **List**: As the name suggests, a list contains more than one item. It can contain different types of items together. A list is created using square brackets “[ ],” where the items are kept using commas. A list can be modified after it is created.
4. **Tuple**: A tuple is like a list except that it cannot be modified once created. Parentheses “( )” are used to create tuples.
5. **Set**: A set keeps items in an unordered way; so, indexing for a set is not defined the way it is defined for a list or tuple. Duplicate elements are not contained in sets. Like tuples, sets cannot be modified either once created.

Table 1.10 Examples of different data types

Example	Data type
a = 50	Integer
a = 50.25	Float
a = 5+2j	Complex
a =True	Bool
a = "hello world"	String
a = [4, 1, 6.3, "world", 6+2j]	List
a = (4, 1, 6.3, "world", 6+2j)	Tuple
a = {4, 1, 6.3, "world", 6+2j}	Set
a = {1:4, 2:1, 3:6.3, 4:"world", 5:6+2j}	Dictionary

6. **Dictionary**: A dictionary also stores items in an unordered way, like sets, but the user can define the index. To store elements in a dictionary, the user needs to define both the value of the item and the index of the individual item. A dictionary can be modified after it has been created.
7. **Boolean**: Boolean data consist of two values. It can be either "true" or "false."

Table 1.10 demonstrates some examples of different data types.

1.5.8.3 Conditional Statements

Conditional statements execute an operation if a test condition is satisfied. They can be primarily divided into three categories:

- **If statement**: It consists of only one "if condition." The condition of the if statement is checked. The respective block or operation is executed if the condition is fulfilled. Otherwise, the respective block or operation is not executed.

```
if a % 2 == 0:
    print("a is even")
```

  Here, it is checked if the value of a is divisible by 2 or not. If a is divisible by 2, then the statement `print("a is even")` is executed. Otherwise, the statement is simply ignored.
- **If...else statements**: It also consists of only one "if condition." The condition of the if statement is checked. If the condition is fulfilled, then the respective block is executed. The difference here with the if statement is that if the condition is not fulfilled, then the "else block" is executed.

```
if b % 2 == 0:
    print("b is even")
else:
    print("b is odd")
```

Here, it is checked if the value of *b* is divisible by 2 or not. If *b* is divisible by 2, then the statement `print("b is even")` is executed. Otherwise, the statement `print("b is odd")` is executed.

- **If...elif...else statements**: It consists of more than one condition. The conditions are checked. The condition that is fulfilled first, the respective block under that condition is executed. The “else block" is executed if no condition is fulfilled.

```
if a > b:
    print("a is greater")
elif b > a:
    print("b is greater")
else:
    print("a and b are equal")
```

Here, it is checked that if the value of *a* is greater than the value of *b*, then the statement `print("a is greater")` is executed. If the condition is not fulfilled, then the next condition is checked: whether the value of *b* is greater than the value of *a*. If this condition is fulfilled, the statement `print("b is greater")` is executed. If this condition is also not fulfilled, then the else statement is executed.

- **Nested if statements:** It may consist of one or more than one condition. The difference here is that if one condition is fulfilled, it proceeds to check other conditions under that particular condition. On the other hand, if that condition is not fulfilled, the other conditions under that particular condition are not checked.

```
if a > b:
    if a % 2 == 0:
        print("a is greater and even")
    else:
        print("a is greater and odd")
elif b > a:
    print("b is greater")
else:
    print("a and b are equal")
```

Here, the first condition is checked, and if the value of *a* is greater than *b*, then it proceeds to check the conditions whether *a* is divisible by 0 or not. On the other hand, if the value of *a* is not greater than *b*, then the divisibility condition of *a* is not checked.

Programming Example 1.2

A Python program demonstrating the use of conditional statements is given in Listing 1.2 followed by its output and explanation in Table 1.11. The conditional statements mostly participate in decision-making and performing checks on different requirements. The code shows the implementation of different conditional statements using two variables, *a* and *b*. The code uses normal and nested conditional statements to determine whether *a* and *b* are even or odd and greater or smaller in value.

```
 1 a = 40
 2 b = 13
 3
 4 if a % 2 == 0:
 5     print("a is even")
 6
 7 if b % 2 == 0:
 8     print("b is even")
 9 else:
10     print("b is odd")
11
12 if a > b:
13     print("a is greater")
14 elif b > a:
15     print("b is greater")
16 else:
17     print("a and b are equal")
18
19 if a > b:
20     if a % 2 == 0:
21         print("a is greater and even")
22     else:
23         print("a is greater and odd")
24 elif b > a:
25     print("b is greater")
26 else:
27     print("a and b are equal")
```

Listing 1.2 Example of conditional statements using Python

Output of Listing 1.2:

```
a is even
b is odd
a is greater
a is greater and even
```

1.5.8.4 Loops

Loops are used for repetitive tasks. A loop statement is executed when a certain operation has to be performed repeatedly until a specified condition is satisfied. Figure 1.22 demonstrates structures of different types of loops. There are three main types of loops—for loop, while loop, and do...while loop:

Table 1.11 Explanation of Listing 1.2

Line number	Description
1–2	Creating two integer variables
4–5	If statement
7–10	If...else statement
12–17	If...elif...else statement
19–27	Nested if statement

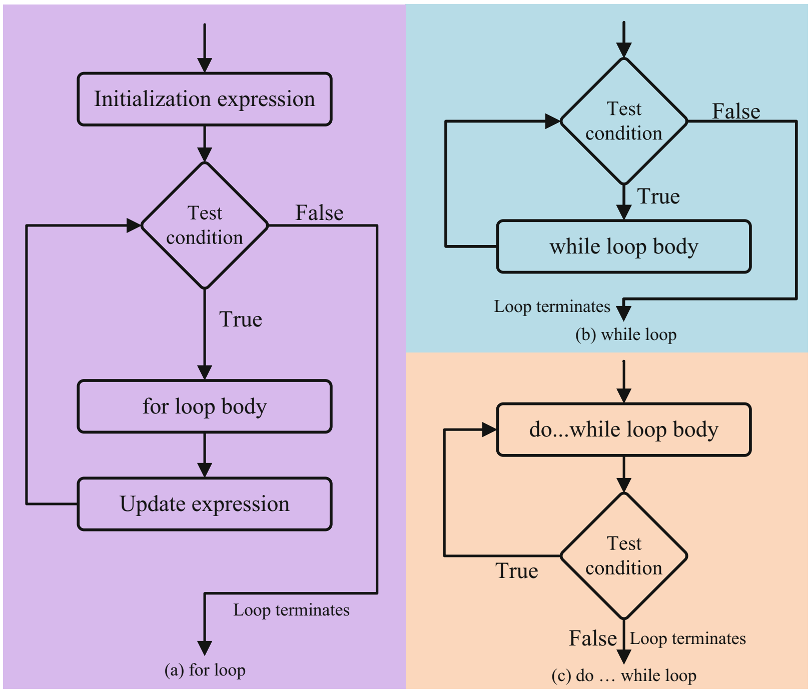


Fig. 1.22 Three types of loops: *for loop*, *while loop*, and *do...while loop*

1. **For loop:** The *"for loop"* consists of a condition and an execution body. The loop executes the body for a pre-defined number of iterations and simultaneously checks for the validity of the condition. The loop continues as long as it finds the condition valid or until a pre-defined number of iterations.
   Say an array of colors is given:
   `colors = ['red', 'black', 'orange', 'blue', 'pink'].`
   Using *for loop*, the colors in the array `colors` can be accessed as:

   ```
   for color in colors:
   print(color)
   ```

   Here, the variable "color" iterates through the data sequentially in the array `colors`.
   Another function, `range()`, can be used to access the data using *for loop*. The parameters for the `range()` function are `range(start, stop, step)`. The parameters used in `range()` are:

   - `start`—It specifies the initial index for the loop to start iterating. If not specified, the default value for "start" is 0.

- `stop`—The value for “stop" has to be specified; it is not optional. The loop iterates till the stop value (excluding it).
- `step`—It specifies the incremental step for the loop. If not specified, the default value for “step" is 1.

  Some examples of `range()` are explained below:

- `range(6)`: For loop will initiate from 0 and iterate for all values to 5 (six places less than 6).
- `range(1, 6)`: For loop will initiate from 1 and iterate for all values to 5. This one is also up to less than 6.
- `range(0, 6, 2)`: For loop will initiate from 0 and will increment and iterate at an increment of 2 steps to output the values (0, 2, 4). This one is also up to less than 6.

2. **While loop:** The *“while loop"* also consists of a condition and an execution block. The number of iterations for the “while loop" is not pre-defined. The *while* loop will keep executing its execution block as long as the condition is valid. Once the condition is not fulfilled, the loop stops its execution. Here, the iteration number is incremented in the execution block.

```
i=0
while i < 5:
    print(i)
    i += 1
```

   The initialization has to be made before the execution of the loop. The loop will keep executing as long as the value of *i* is less than 5.
3. **Do...while loop:** In a *“do...while loop,"* the execution block is executed first, the iteration number is incremented, and then the condition is checked. If the condition is not fulfilled, the loop stops there. Otherwise, it proceeds with its execution. In a *“while loop"*, the condition is checked at first, but in a *“do...while loop,"* the condition is checked last.

Another special type of loop, known as the *nested loop*, involves one loop inside another loop. This loop is used when there are more than one condition to be satisfied.

Programming Example 1.3

A Python program demonstrating the use of conditional statements is given in Listing 1.3 followed by its output and explanation in Table 1.12. The listing uses two lists named `colors` and `shapes` to demonstrate the use of different types of loops as well as conditional statements.

```python
colors = ['red', 'black', 'orange', 'blue', 'pink']
shapes = ['circle', 'square', 'triangle']

for color in colors:
   print(color)
print("---------------")

for color in colors:
    if color == 'orange':
        break
    print(color)
print("---------------")

for color in colors:
    if color == 'orange':
        continue
    print(color)
print("---------------")

for color in colors:
    for shape in shapes:
        print(color, shape)
print("---------------")

for a in range(5):
  print(a)
print("---------------")

for a in range(1,5):
  print(a)
print("---------------")

for a in range(1,6,2):
  print(a)
print("---------------")

i = 0
while i < 5:
  print(i)
  i += 1
```

Listing 1.3 Example of looping operations using Python

Output of Listing 1.3:

```
red
black
orange
blue
pink
---------------
red
black
---------------
```

```
red
black
blue
pink
---------------
red circle
red square
red triangle
black circle
black square
black triangle
orange circle
orange square
orange triangle
blue circle
blue square
blue triangle
pink circle
pink square
pink triangle
---------------
0
1
2
3
4
---------------
1
2
3
4
---------------
1
3
5
---------------
0
1
2
3
4
```

1.5.8.5 Array

Array is a type of data structure that can store the same type of data in sequential order. In Python, lists are used to implement arrays. Different types of operations can be performed on arrays, such as traversing, inserting, deleting, etc. The indexing in arrays starts from zero as shown in Fig. 1.23.

Table 1.12 Explanation of Listing 1.3

Line number	Description
1–2	Creating two lists
4–6	Looping through all the elements
8–12	Breaking at color orange
14–18	Skipping color orange
20–23	Nested loop
25–27	Printing from 0 to 4
29–31	Printing from 1 to 4
33–35	Printing from 1 to 5, incrementing the value of “a" by 2
37–40	While loop

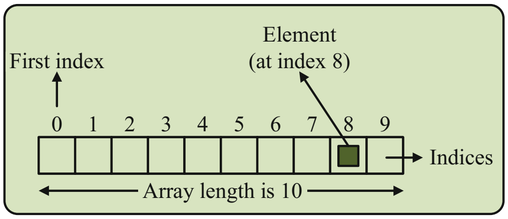


Fig. 1.23 Representation of array in programming languages

Programming Example 1.4

A Python program demonstrating the use of different operations on an array is given in Listing 1.4 followed by its output and explanation in Table 1.13. An array named `colors` is utilized in the listing to demonstrate how array elements can be accessed one at a time. Again, it shows the process of looping through the array to update, append, remove, and perform some other tasks.

```
# creating a list/array
colors = ['red', 'black', 'orange', 'blue']

# Accessing array elements
print("The first color in the array is", colors[0])  ## red
print("The second color in the array is", colors[1]) ## black

# looping through all the elements in the array
for color in colors:
    print(color) ## red, black, orange, blue

# Slicing
print(colors[1:3])  ## ['black', 'orange']
print(colors[0:4])  ## ['red', 'black', 'orange', 'blue']
print("---------------")

# Updating
colors[2] = 'green'
for color in colors:
    print(color)      ## red, black, green, blue
print("---------------")

colors.append('pink')
for color in colors:
    print(color)      ## red, black, green, blue, pink
print("---------------")

colors.insert(3, 'yellow')
for color in colors:
    print(color)    ## red, black, green, yellow, blue, pink
print("---------------")

# Delete
colors.remove("pink")
for color in colors:
    print(color)      ## red, black, green, yellow, blue
print("---------------")

colors.pop(2)
for color in colors:
    print(color)      ## red, black, yellow, blue
```

Listing 1.4 Example of different operations performed on an array in Python

Output of Listing 1.4:

```
The first color in the array is red
The second color in the array is black
red
black
orange
blue
['black', 'orange']
['red', 'black', 'orange', 'blue']
---------------
red
black
green
blue
---------------
red
black
green
blue
pink
---------------
red
black
green
yellow
blue
pink
---------------
```

```
red
black
green
yellow
blue
---------------
red
black
yellow
blue
```

Table 1.13 Explanation of Listing 1.4

Line number	Description
1	Creating array
4–6	Accessing one element from at a specific index
9–10	Accessing all the elements from the array
12–15	Accessing elements in the range from beginning to end-1
17–21	Updating the value at a certain index
23–26	Adding an element at the end of an array
28–31	Inserting an element at a specific index
33–37	Removing an element from the array without concern for the index
39–41	Removing element at certain index

1.5.8.6 Vector

Vector was introduced in this chapter earlier in section “Vector". In this section, we will see the different kinds of arithmetic operations that can be performed on vectors using Python. In machine learning, these operations come in handy.

Programming Example 1.5

A Python program demonstrating the use of different arithmetic operations on a vector is given in Listing 1.5 followed by its output and explanation in Table 1.14. The code showcases methods of creating vectors. Different arithmetic operations between two vectors are also discussed in the code.

```
# importing numpy library
import numpy as np

# creating row vectors
a = np.array([2, 4, 6])
print("Row vector a: ",a)
b = np.array([1, 2, 3])
print("Row vector b: ",b)

# creating column vectors
```

```
s = np.array([[2],
              [4],
              [6]])
print("Column vector s: ",s)
t = np.array([[1],
              [2],
              [3]])
print("Column vector t: ",t)

# addition
c = a + b
print("Addition: ",c)

# substraction
d = a - b
print("Substraction: ",d)

# mutiplication
e = a * b
print("Multiplication: ",e)

# division
f = a / b
print("Division: ",f)

# dot product
g = a.dot(b)
print("Dot product: ",g)

# scalar multiplication
h = 0.5 * a
print("Scalar multiplication", h)
```

Listing 1.5 Example of arithmetic operations performed on vectors using Python

Output of Listing 1.5:

```
Row vector a:  [2 4 6]
Row vector b:  [1 2 3]
Column vector s:  [[2]
 [4]
 [6]]
Column vector t:  [[1]
 [2]
 [3]]
Addition:  [3 6 9]
Substraction:  [1 2 3]
Multiplication:  [ 2  8 18]
Division:  [2. 2. 2.]
Dot product:  28
Scalar multiplication [1. 2. 3.]
```

Table 1.14 Explanation of Listing 1.5

Line number	Description
2	Importing NumPy library
4–8	Creating, printing row vectors
10–18	Creating, printing column vectors
20–22	Addition of vectors a and b
20–22	Subtraction of vectors a and b
24–26	Addition of vectors a and b
28–30	Multiplication of vectors a and b
32–34	Division of vectors a and b
36–38	Dot products of vectors a and b
40–42	Scalar multiplication with vector a

1.5.8.7 Functions

Functions are self-sufficient lines of code that can execute a specific action. They are comparable to a black box that can take some arguments as input, perform an operation using the inputs, and return an output. The function can be called from the main program or other functions and can be used repeatedly.

Suppose we are to create a function named *mean* that can find the arithmetic mean of a set of numbers. After writing the instructions for the function, we only need to feed the function with a dataset whose mean will be determined. Then, the function is executed, and the arithmetic mean of the input dataset is obtained. Whenever we need to find the arithmetic mean in any place within the program, we only need to call the function *mean*, and we can get the result within a moment without having to write the same instructions over and over again. Thus, the use of functions saves time and minimizes the necessary lines of code by adding the reusability feature within the code.

The syntax for a function in Python is given below:

```
def "user defined function name" (parameters):
    function block
    return variables
```

The function block is initiated with the syntax `def`. Here, the name of the function is defined by the users. The naming of the function follows the same conventions of naming variables. The parameters or list of parameters are defined in parentheses. In the function block, the code is written, which is to be performed by the function. Python is sensitive to indentation. So, the spacing in lines should be constantly checked while coding in Python. The statement `return` ends the definition of the function and returns whatever the user defines to be returned. The function ends at its last line without returning value if the return statement is not given.

Programming Example 1.6

A Python program demonstrating a function for finding the arithmetic mean is given in Listing 1.6 followed by its output and explanation in Table 1.15. The code declares a user-defined function to find out the arithmetic mean of a list of numbers.

```
1  #defining function
2  def calculate_mean1(list):
3    a = len(list)
4    sum = 0
5    for i in list:
6      sum += i
7    result1 = sum / a
8    return result1
9
10 def calculate_mean2(list):
11   a = len(list)
12   sum = 0
13   for i in list:
14     sum += i
15   result2 = sum / a
16   print("The mean calculated from the second function: ", result2
       )
17
18 list_of_numbers = [1, 2, 3, 4, 5, 6]
19 result = calculate_mean1(list_of_numbers)
20 print("The mean calculated from the first function: ", result)
21 calculate_mean2(list_of_numbers)
```

Listing 1.6 Example of a user-defined function to estimate the mean of a list of numbers using Python

Output of Listing 1.6:

```
The mean calculated from the first function:  3.5
The mean calculated from the second function:  3.5
```

Table 1.15 Explanation of Listing 1.6

Line number	Description
2–8	Defining a function that returns the value of the result
10–16	Defining a function that does not return any value
18	The list of numbers of which mean has to be calculated
19–20	Calling the first function and printing the result
21	Calling the second function

1.6 Programming Languages and Associated Tools

Python and R are the two most highly used programming languages for developing ML programs. In addition, several other languages, such as Julia, Java, Javascript, LISP (List Processing), C++, C, PHP, etc., are used by ML professionals. No language can be selected as the best among these languages since each has distinct features, packages, built-in libraries, and other attributes, making them suitable for several tasks. It is mostly about preference rather than being *the best* programming language. Someone skilled at Python will choose to use Python over the other languages no matter what benefits the other languages may accrue. Table 1.16 enlists four programming languages and provides a comparative view of their key features to be considered in the field of ML.

1.6.1 Why Python?

Python will be used as the only programming language in all examples of this book due to its ease of use, popularity, and the large, friendly, helpful, and interactive community Python encompasses. It is open-source, highly used in academic and research-based works, and is recommended by experts in almost every field. It is very efficient in terms of the amount of code needed to be written. The short, simple lines of Python code with obvious implications can be easily handled by beginners and are easy to read, debug, and expand. Python is also a cross-platform programming language, implying that it can run well on all operating systems and computers.

Python has often been nicknamed the *Swiss army knife of programming languages* because of its versatile nature and a wide range of functionalities. It contains many packages that can cater to almost all possible applications.

Table 1.16 Comparison of different programming languages. Python and R are most commonly used for machine learning applications

Features	Python	R	Java	C++
Learning curve	Smooth	Steep	Smooth	Smooth
Packages/ libraries	A lot	Moderate	Moderate	A few
Syntax/code readability	Easy	Easy	Moderate	Moderate
Built-in ML techniques	A few	A lot	Moderate	Moderate
Performance on repetitive tasks	Better	Moderate	Moderate	Moderate
Mostly used by	Data scientist/ engineers	Data analyst/ statistician	Developer	Embedded computer hardware

1.6.2 Installation

Machine learning problems are implemented using different algorithms based on the problem type. The implementation of these algorithms can be very lengthy and time-consuming. Frameworks and libraries are used to make the implementation of the algorithms easier.

Libraries consist of written codes of different functions and operations that users can reuse. Frameworks also consist of written codes of different functions and operations, plus they consist of other tools and necessary stuff for the application. Frameworks provide a skeleton for the developers to work with. Libraries and frameworks save developers from many complex implementations and save time from writing the same code repeatedly. There are different types of libraries to deal with different things. For example, we have text processing libraries, graphics libraries, data manipulation, and scientific computation. The most used ML libraries are NumPy, Pandas, Scipy, Theano, Keras, Scikit-learn, Matplotlib, etc., while the most common frameworks are PyTorch and TensorFlow.

An integrated development environment (IDE) is used as an environment to write and compile the codes. An IDE contains a source code editor, compiler, debugger, etc. For our purpose, we will be using Spyder and Jupyter Notebook. These IDEs already contain many ML libraries required. Anaconda software will be used to install Spyder and Jupyter Notebook.

1.6.3 Creating the Environment

Before experiencing problem solving using ML, we first need to create the necessary environment for implementing and running the codes. Next, we will need to install the necessary software for writing codes and running them. Finally, we will need to install the necessary frameworks and libraries after installing and setting up the software. The installation process is discussed step by step in the following sections.

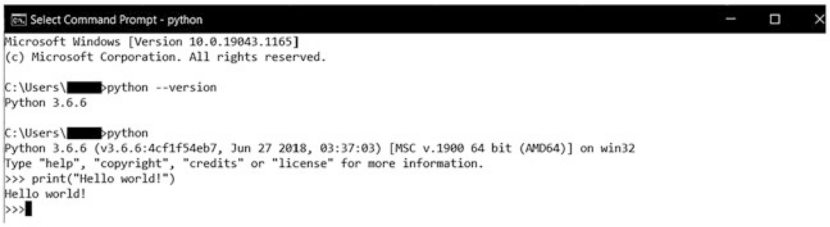
1.6.3.1 Creating the Environment in Windows

1.6.3.1.1 Installing Python

We will be working with Python3. To download Python3, go to https://www.python.org/downloads/ and download the latest version shown for Windows. To check that Python3 has been successfully installed, run the following command on the command prompt as shown in Fig. 1.24.

1.6.3.1.2 Installing Anaconda

Download Anaconda navigator from https://www.anaconda.com/products/individual. The installation screen should look like the steps shown in Fig. 1.25.



```
Microsoft Windows [Version 10.0.19043.1165]
(c) Microsoft Corporation. All rights reserved.

C:\Users\[redacted]>python --version
Python 3.6.6

C:\Users\[redacted]>python
Python 3.6.6 (v3.6.6:4cf1f54eb7, Jun 27 2018, 03:37:03) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello world!")
Hello world!
>>>
```

Fig. 1.24 Checking Python version on command prompt

1.6.3.2 Creating the Environment in MacOS

1.6.3.2.1 Installing Python

Go to https://www.python.org/downloads/ and download the latest version that is shown for MacOS. The Python version can be checked following Fig. 1.26. To check that Python3 has been successfully installed, run the following command on terminal:

```
python3 --version
```

1.6.3.2.2 Installing Anaconda

Download Anaconda navigator from https://www.anaconda.com/products/individual. The installation screen should resemble Fig. 1.27.

Install and launch Jupiter notebook and Spyder IDE from the screen shown in Fig. 1.28.

1.6.3.3 Installing Necessary Libraries

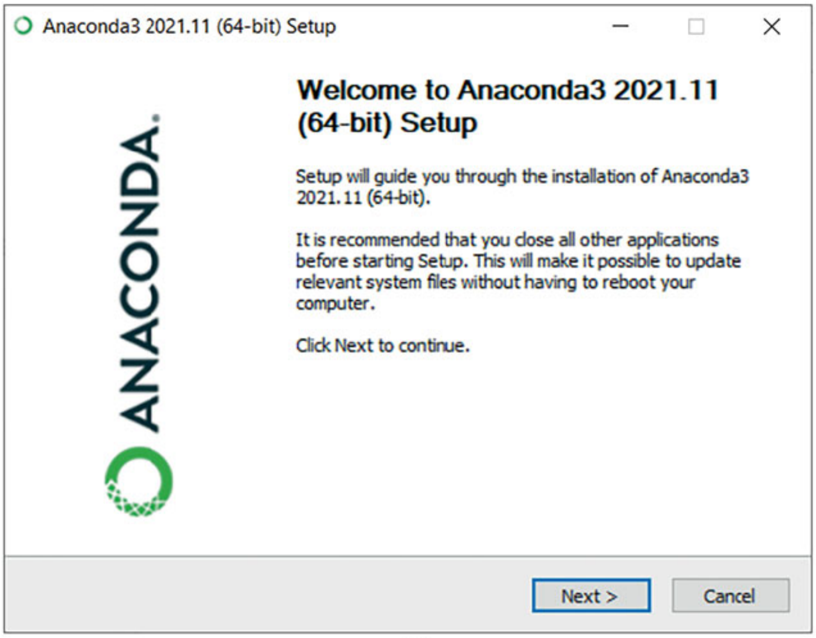
To use the TensorFlow library, we need to create the environment and install the necessary libraries within that environment. To create a TensorFlow environment named “test," run the following command in the terminal:

```
conda create -n test tensorflow
```

Now to install the necessary libraries inside this environment, at first, we need to activate the environment “test." To do this, run the following commands on the terminal:

```
conda activate test
pip3 install NumPy
pip3 install panda
pip3 install script
pip3 install matplotlib
```

Jupyter Notebook and Spyder IDE need to be installed for each environment. The way to access applications in the desired environment is shown in Figs. 1.29, 1.30, and 1.31.



Anaconda3 2021.11 (64-bit) Setup

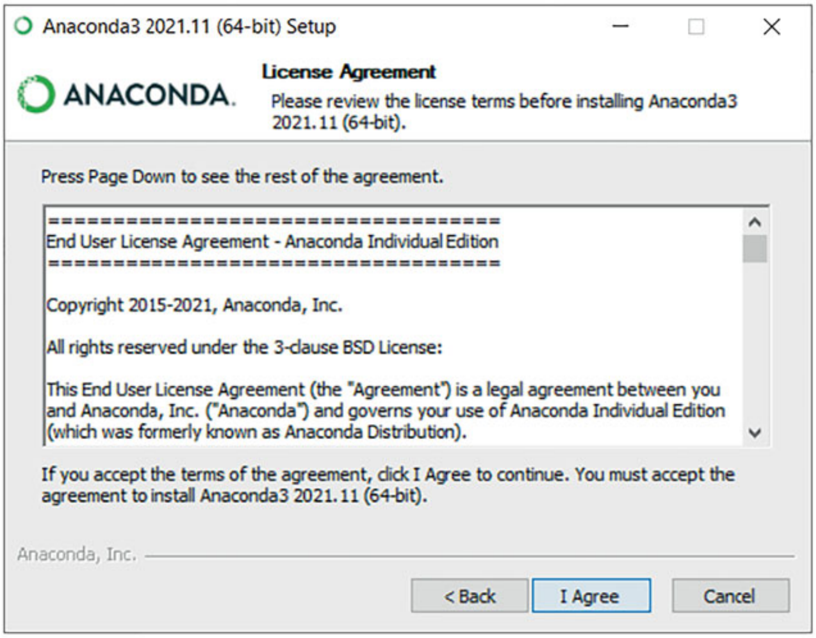
Welcome to Anaconda3 2021.11 (64-bit) Setup

Setup will guide you through the installation of Anaconda3 2021.11 (64-bit).

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

(a)



Anaconda3 2021.11 (64-bit) Setup

License Agreement

Please review the license terms before installing Anaconda3 2021.11 (64-bit).

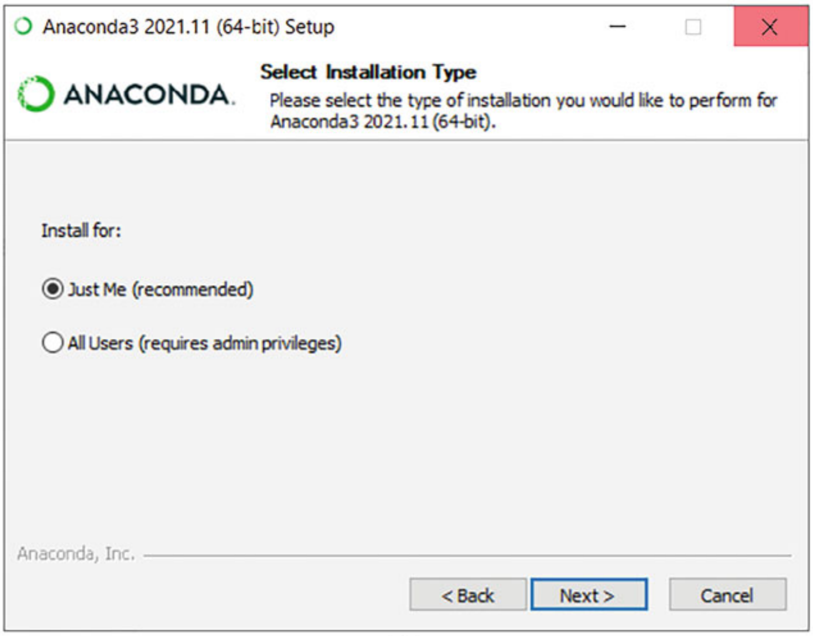
Press Page Down to see the rest of the agreement.

```
===================================
End User License Agreement - Anaconda Individual Edition
===================================

Copyright 2015-2021, Anaconda, Inc.

All rights reserved under the 3-clause BSD License:

This End User License Agreement (the "Agreement") is a legal agreement between you
and Anaconda, Inc. ("Anaconda") and governs your use of Anaconda Individual Edition
(which was formerly known as Anaconda Distribution).
```

If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install Anaconda3 2021.11 (64-bit).

(b)

Fig. 1.25 The six steps of Anaconda installation in Windows OS. (**a**) Step 1 of Anaconda installation. (**b**) Step 2 of Anaconda installation. (**c**) Step 3 of Anaconda installation. (**d**) Step 4 of Anaconda installation. (**e**) Step 5 of Anaconda installation. (**f**) Step 6 of Anaconda installation


Anaconda3 2021.11 (64-bit) Setup

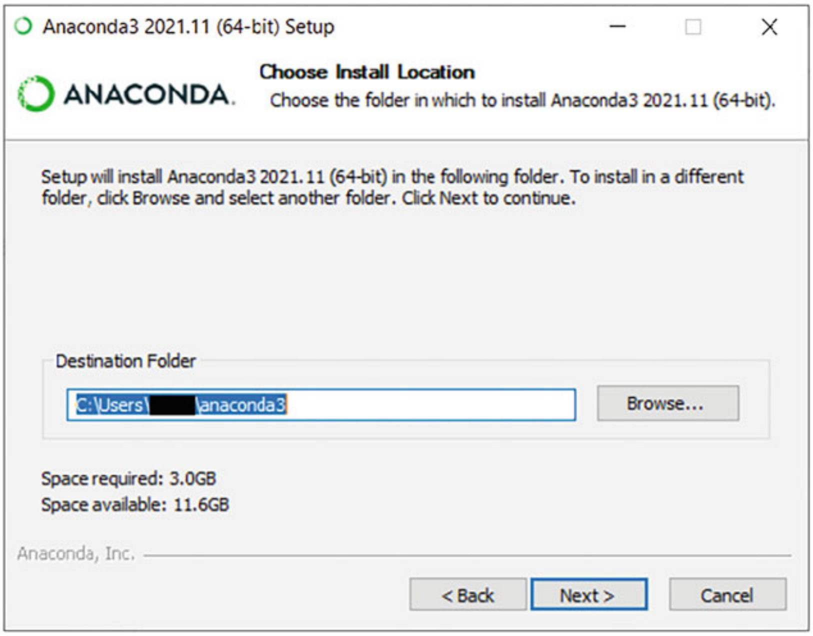
Select Installation Type

Please select the type of installation you would like to perform for Anaconda3 2021.11 (64-bit).

Install for:

[✓ Just Me (recommended)] All Users (requires admin privileges)

(c)


Anaconda3 2021.11 (64-bit) Setup

Choose Install Location

Choose the folder in which to install Anaconda3 2021.11 (64-bit).

Setup will install Anaconda3 2021.11 (64-bit) in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.

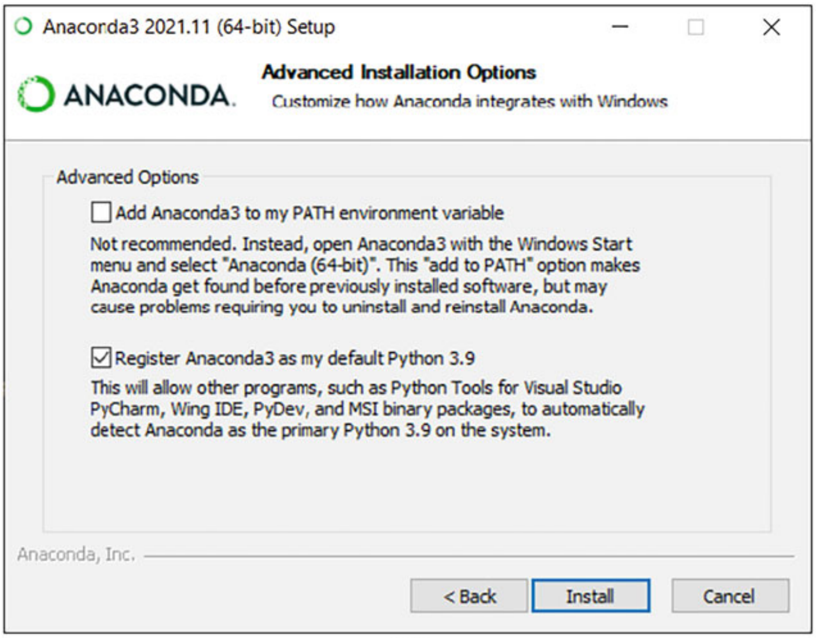
Destination Folder

C:\Users\[redacted]\anaconda3

Space required: 3.0GB
Space available: 11.6GB

(d)

Fig. 1.25 (continued)



Advanced Installation Options

Customize how Anaconda integrates with Windows

Advanced Options

☐ Add Anaconda3 to my PATH environment variable

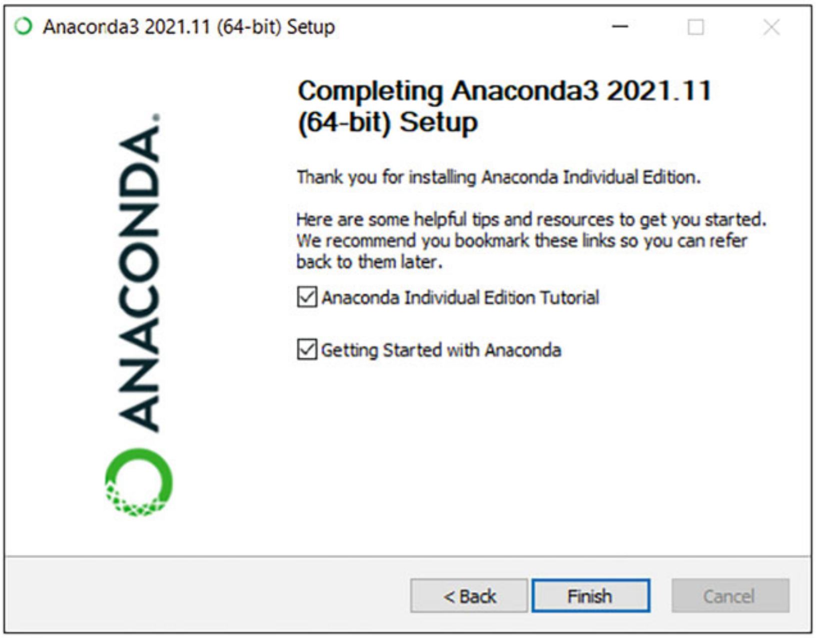
Not recommended. Instead, open Anaconda3 with the Windows Start menu and select "Anaconda (64-bit)". This "add to PATH" option makes Anaconda get found before previously installed software, but may cause problems requiring you to uninstall and reinstall Anaconda.

☑ Register Anaconda3 as my default Python 3.9

This will allow other programs, such as Python Tools for Visual Studio PyCharm, Wing IDE, PyDev, and MSI binary packages, to automatically detect Anaconda as the primary Python 3.9 on the system.

Anaconda, Inc.

(e)



Completing Anaconda3 2021.11 (64-bit) Setup

Thank you for installing Anaconda Individual Edition.

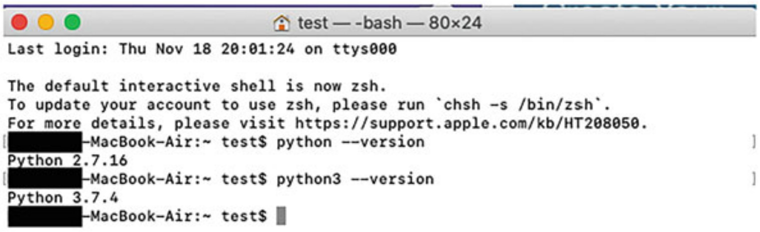
Here are some helpful tips and resources to get you started. We recommend you bookmark these links so you can refer back to them later.

☑ Anaconda Individual Edition Tutorial

☑ Getting Started with Anaconda

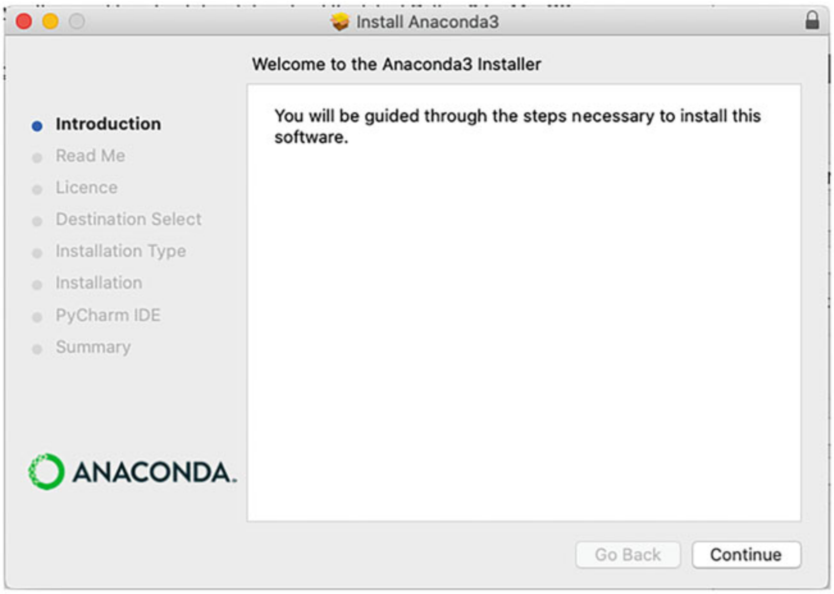
(f)

Fig. 1.25 (continued)



```
Last login: Thu Nov 18 20:01:24 on ttys000

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
[redacted]-MacBook-Air:~ test$ python --version
Python 2.7.16
[redacted]-MacBook-Air:~ test$ python3 --version
Python 3.7.4
[redacted]-MacBook-Air:~ test$
```

Fig. 1.26 Checking Python version on terminal on macOS



Welcome to the Anaconda3 Installer

You will be guided through the steps necessary to install this software.

Fig. 1.27 Installing Anaconda in macOS

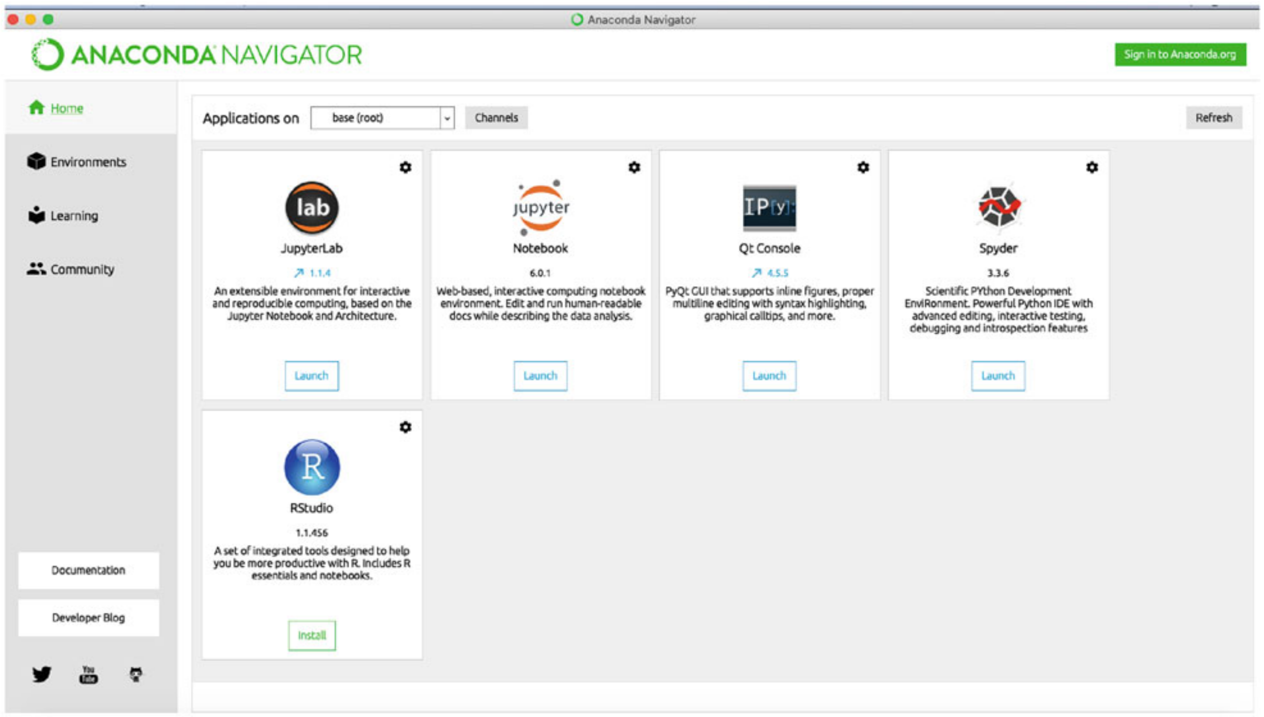
To install the PyTorch framework on Windows, run the following command on the Anaconda prompt.

```
conda install pytorch torchvision torchaudio cpuonly
-c pytorch
```

To install the PyTorch framework on MacOS, run the following command on the terminal.

```
conda install pytorch torchvision -c pytorch
pip3 install torch torchvision
```

Besides, *Google Colab* offers a cloud-based solution for Python notebooks free of cost. It even offers free graphics processing unit (GPU) access for hardware acceleration with some limitations. Additional computing power can be accessed



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

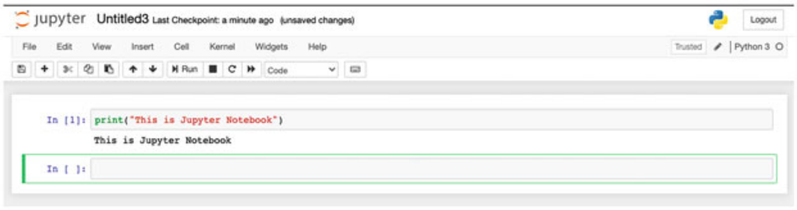
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

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

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

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

Fig. 1.28 Anaconda Navigator screen on macOS


jupyter Untitled3 Last Checkpoint: a minute ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
In [1]: print("This is Jupyter Notebook")
        This is Jupyter Notebook

In [ ]:
```

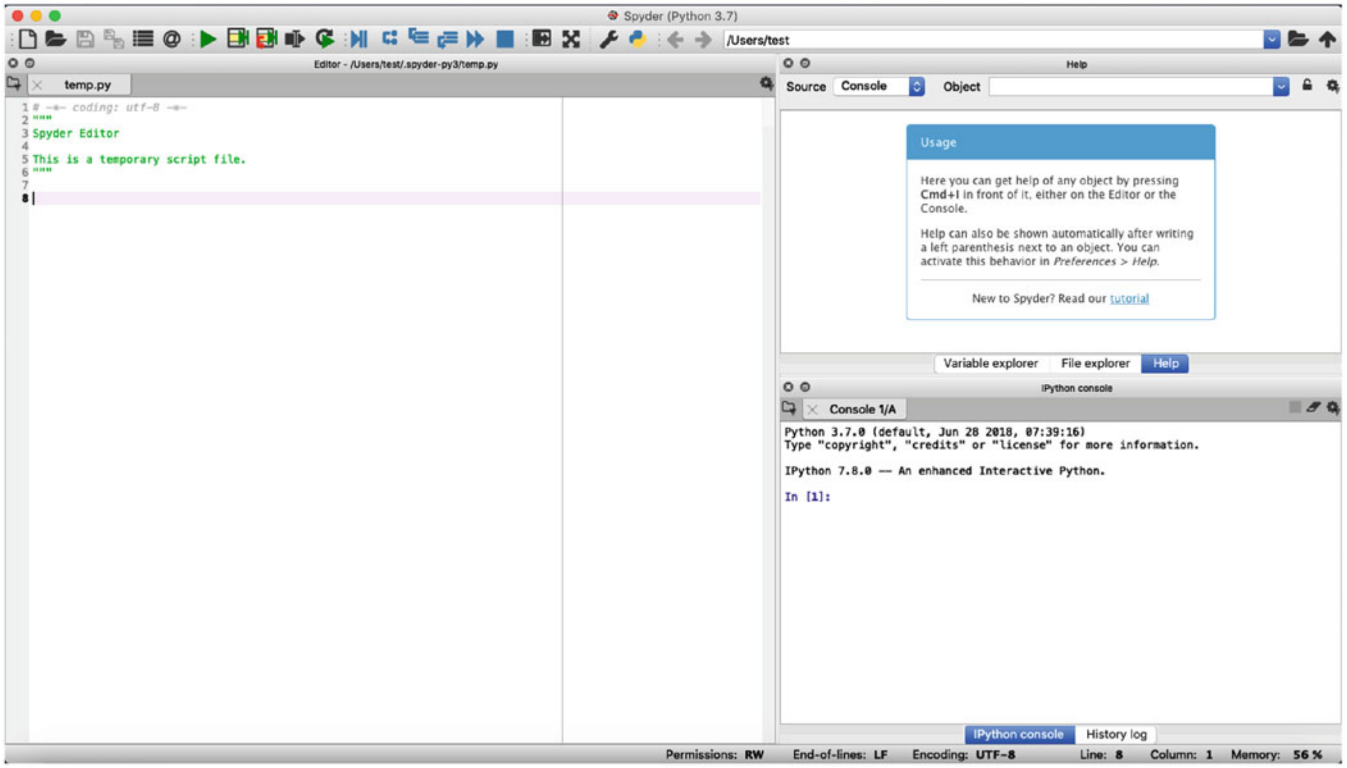
Fig. 1.29 Jupyter Notebook on macOS

through a paid subscription. Another good online platform for online machine learning practice is *Kaggle*. It provides a large collection of open-source datasets and cloud-based computing resources (CPU—Central Processing Unit, GPU, and TPU—Tensor Processing Unit) free of cost. Users can also participate in various machine learning competitions and engage in community discussion through the Kaggle platform.

1.7 Applications of Machine Learning

Machine learning is ubiquitous today in the twenty-first century. It can be applied in all domains of science and engineering. The expanse of ML is so vast that even if one is not directly using it, one needs to have an understanding of ML to study the works of others. It is an all-pervading tool in engineering research and development. ML is a versatile field of science, finding applications not only in engineering problems but also in computer science, economics, business analytics, marketing, finances, social networks, travel and hospitality management, and so on. This book is intended for engineers, so only some engineering applications of ML are explored in detail in this section:

- **Data Analysis:** Data analysis is an obvious application of ML. With large datasets, manual effort is simply pointless, ineffective, and prone to errors, not to mention the monotony of the work. ML algorithms can efficaciously handle large datasets, analyze them, detect patterns, and produce the intended results.
- **Signal and Image Processing:** The role of ML is unparalleled in the case of image recognition, classification, editing, and processing.
- **Spam Filter:** The spam folder in your email inbox is one of the most highly cited applications of ML. By analyzing patterns of spam mail encountered by the users, ML algorithms detect spam mail and directly send them to the spam folder.
- **Fraud detection:** When human detection falls short, the superior power of ML helps to detect fraudulent activities very efficiently.



```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""

```

Usage

Here you can get help of any object by pressing **Cmd+I** in front of it, either on the Editor or the Console.

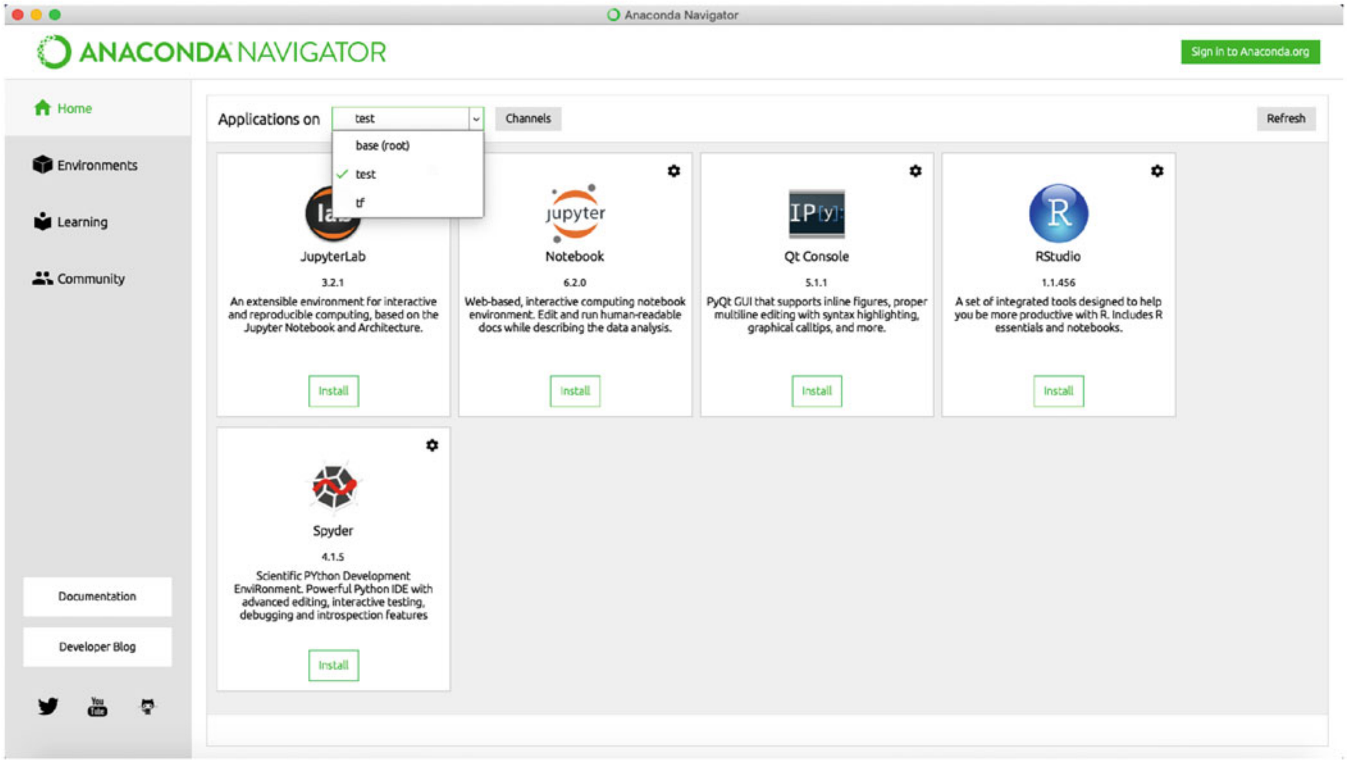
Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Help*.

New to Spyder? Read our tutorial

```
Python 3.7.0 (default, Jun 28 2018, 07:39:16)
Type "copyright", "credits" or "license" for more information.

IPython 7.8.0 -- An enhanced Interactive Python.

In [1]:
```

Permissions: RW   End-of-lines: LF   Encoding: UTF-8   Line: 8   Column: 1   Memory: 56 %

Fig. 1.30 Spyder IDE on macOS



JupyterLab
3.2.1
An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.

Notebook
6.2.0
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

Qt Console
5.1.1
PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.

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

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

Fig. 1.31 Installation of Jupyter Notebook and Spyder IDE for each environment on macOS

- **Healthcare system:** ML applications are not strictly limited to mathematics and engineering but also pervade in biomedical systems, particularly for disease identification through symptom analysis.
- **Clustering or Segmentation:** Clustering or segmenting similar groups of data is a prominent type of unsupervised ML technique. From any random dataset, an unsupervised ML algorithm can easily detect patterns and cluster data points based on its observations.
- **Robotics:** Artificial intelligence lies at the heart of modern robotics. ML is so intricately involved with the field of robotics that a new branch named *robot learning* has emerged that involves a robot gathering experiences and adapting to its environment using ML algorithms.
- **Forecasting:** ML is highly used in forecasting the weather, predicting natural disasters, and electrical load or demand forecasting in power systems. The long-term climate patterns can also be predicted using ML algorithms.
- **Simulation:** Simulations are used to run tests and predict probabilities of certain aspects using a well-established model. ML can be combined with simulation to achieve more remarkable results and solve many problems. ML can be applied to the input data before feeding it to the simulation or during the simulation process. It can even be applied to the output after the simulation is processed.
- **Control systems:** ML has deeply penetrated the world of control systems to such an extent that a sub-field of ML is known as machine learning control, which deals with optimal control problems using ML algorithms. These control systems can be implemented in electric power systems, air conditioning and refrigeration systems, satellite systems, navigation of deep water vessels, robotics, and so on.
- **Natural Language Processing (NLP):** ML has widespread applications in NLP, where information is extracted from human speech or text, and actions are taken accordingly. For example, you texted a friend saying that you were going to buy a new phone. Your data are recorded and processed immediately. The cross-app connectivity will make phone ads appear on your apps, such as Facebook, YouTube, and even your browser. This is an example of NLP, and it utilizes ML algorithms for analyzing the text. Besides digital advertising, NLP finds applications in translating languages, making chatbots, personal assistants like Siri and Alexa, sentiment analysis, auto-correct in text applications, and many more. It would be folly not to mention the breakthrough technology called *Chat GPT* at this point. Chat Generative Pre-trained Transformer (Chat GPT) is an AI chatbot developed by OpenAI. It is an NLP tool that uses ML algorithms to generate text responses based on input text prompts.
- **Computer Vision:** Computer vision is the technology that enables computers and other machines to be able to see and glean information as we do. Deep learning is necessary for computer vision applications. This technology is inextricably used in self-driving cars, facial recognition systems, image searches on search engines, captcha codes, security and protection systems, and so on.
- **Cybersecurity:** ML has widespread applications in ensuring the security of wireless networks. Network intrusion detection and prevention systems are built using ML algorithms.

- **Holography:** ML, particularly deep learning, is used for generating digital holograms, which are used for securely storing vital information.
- **Power System:** Load forecasting is a key function in all operational power systems. ML techniques can be used to forecast load demand, detect faulty conditions, predict overloads, clear faults, control interconnected systems, control switching operations, and so on.
- **Power Electronics:** Power electronics can be benefited from ML in the design, control, and maintenance stages. Design optimization of power electronics devices, intelligent controller design and application, anomaly detection in inverter signal, and remaining useful life prediction of supercapacitors—ML techniques contribute to all of these. With the application of ML, self-aware and self-adaptive power electronic solutions can be developed.
- **Agriculture:** Although agriculture is one of mankind's oldest practices, modern science and technology have revolutionized agricultural tools and techniques worldwide. The world has entered into the era of *digital agriculture*, also known as *smart farming*, which combines the magic of ML, AI, and data science with a touch of other technologies. ML can be used for crop management, prediction, and estimation of farming parameters to optimize the economic efficiency of livestock production systems, such as cattle and egg production.
- **Manufacturing and automation:** Industrial manufacturing processes require several cascaded, automated machines that can run efficiently and deliver the right products at the right time. ML can also help in the automation process and in maintenance and monitoring.
- **Game Development:** Modern game development scenario has heavy machine learning application. From environmental elements generation, improved non-playable character (NPC) intelligence, and environmental responsiveness improvement to high-end game playability at a higher framerate—ML has significantly improved many aspects of modern game development and overall gaming experience.

1.8 Conclusion

This chapter intends to introduce readers to the amazing and versatile world of Machine Learning. Understandably, the complex math involved might overwhelm beginners at this stage, but that is okay. Once you get the hang of it, ML will not seem so complicated anymore. This chapter assumes that a reader knows nothing about machine learning but is keen to learn the fundamentals. Accordingly, the basic idea and workflow, the technical terminologies, the prerequisite knowledge, and the diverse applications of machine learning are explored in depth in this chapter. Hopefully, by the end of this chapter, you have a fair understanding of what machine learning is all about. We will enter into the details in the next chapter. In the next chapter, we will learn about the different criteria for evaluating and selecting models.

1.9 Key Messages from This Chapter

- Machine learning is required to achieve remarkable applications in daily lives in the data-driven world.
- The four stages in a typical machine learning workflow are dataset collection, data preprocessing, model training, and model evaluation.
- Basic mathematics, statistics, and programming knowledge is required to implement and understand machine learning concepts.
- Python and R programming languages are widely used to implement machine learning algorithms. We will use Python in this book.
- The basics of Python programming can be learned through practice, which will be beneficial to implementing machine learning algorithms efficiently.
- Machine learning applications have significantly made living standards easier in all sectors, from healthcare, security, and entertainment to business, agriculture, and engineering.

1.10 Exercise

1. Define artificial intelligence and machine learning. Why do we need them?
2. Describe the basic workflow of a machine learning algorithm.
3. What is meant by optimization?
4. Define and distinguish between each set of the following terms:
   (a) Objective function, cost function, lost function
   (b) Algorithm, model, technique
   (c) Data science, machine learning, artificial intelligence, deep learning
   (d) Tensor, vector, matrix
   (e) Rank, dimension
   (f) Mean, median, mode
   (g) Global variable, local variable
5. What are the benefits of the Python programming language?
6. What are the applications of machine learning in electrical engineering?
7. Give some programming examples using loop and if statements such as:
   (a) Given an array, check whether the sum of any two numbers in the array equals a target sum. If yes, output 1, and if not, output 0.
   (b) Print the Fibonacci series up to a pre-defined n number.
   (c) Say you have an array with increasing and decreasing sequences. Find the highest peak in that array. For example, in the array $[1, 2, 3, 2, 0]$, the third element is the highest in value, i.e., the peak. Similarly, in the array $[-1, -2, -100, 99, 98, 97]$, the fourth element is the peak.

8. A list of marks obtained by twenty students in an exam is given: [70, 67, 56, 90, 78, 68, 87, 89, 87, 85, 86, 76, 75, 69, 74, 74, 84, 83, 77, 88]. Now, write three different functions in Python to determine the following:
   (a) The `mean()` to calculate the mean mark.
   (b) The `maxmin()` to find the highest and the lowest mark.
   (c) The `std\_dev()` to calculate the standard deviation.
   (d) Repeat the above three steps using NumPy library functions.
9. Find the output of the following code:

```python
bands = ['Linkin Park', 'System Of A Down', 'Metallica', '
    Megadeth', 'Evanescence', 'Poets of the Fall']

print("My favourite bands are: ")
for band in bands:
    if band == 'Megadeth':
        continue
    print(band)

bands[3] = 'Imagine Dragons'

print("Now the list of my favourite bands are: ")
for band in bands:
    print(band)

```

10. Perform matrix multiplication on the following matrix pairs both mathematically and using Python:
    (a) $A = \begin{bmatrix} 1 & 4 \\ 7 & 8 \end{bmatrix}$, $B = \begin{bmatrix} 2 & 9 \\ 11 & 16 \end{bmatrix}$.
    (b) $A = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \\ 1 & 3 & 5 \end{bmatrix}$, $B = \begin{bmatrix} 8 & 10 & 12 \\ 5 & 10 & 15 \\ 1 & 2 & 3 \end{bmatrix}$.
    (c) $A = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \\ 1 & 3 & 5 \end{bmatrix}$, $B = \begin{bmatrix} 1 & 9 \\ 7 & 6 \\ 4 & 3 \end{bmatrix}$.
11. Write a function in Python that calculates the square of a matrix. The code should take the following as inputs from the user:
    (a) The dimension of the matrix, and
    (b) The matrix values

    Sample Input:
    Dimension: 2
    Matrix: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

    Sample Output: $\begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$

12. For the following matrices, determine $A \times B$ and $B \times A$ using Python. Print "*Matrix multiplication is invalid!*" if the multiplication operation cannot be performed:
    (a) $A = \begin{bmatrix} 0.5 & 0.8 \\ -0.96 & 1.1 \end{bmatrix}$, $B = \begin{bmatrix} -0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}$.
    (b) $A = \begin{bmatrix} 0.1 & 0.56 & 0.38 \\ 0.77 & 0.1 & -0.87 \\ 0.84 & -0.91 & 0.1 \end{bmatrix}$, $B = \begin{bmatrix} 0.25 & -0.33 & 0.65 & 1.13 \\ -0.68 & 0.34 & 1 & -0.98 \\ -0.33 & 1.63 & 0.47 & -0.44 \end{bmatrix}$.
13. Matrix A = $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, Matrix B = $\begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$, Output = $\begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \\ 14 & 16 & 19 \end{bmatrix}$.

    A sample code for multiplying Matrix A and Matrix B is provided below. Find the syntax and logical errors in the code to give the given output.

```
matrixA = [[1,2,3] [4,5,6] [7,8,9]]
matrixB = [[2 0 0], [0 2 0], [0 0 2]]
Output = [[0,0,0], [0,0,0], [0,0,0]]

for i in range(0,2):
    for j in range(3):
        sum = 0
        for k in range(3):
            sum += matrixA[j][k] * matrixB[k][j]
        result[j][i] = sum

print(Output)

```

References

1. Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development, 3*, 210–229.
2. Mitchell, T. M. (1997). *Machine learning*. McGraw-Hill.
3. McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics, 5*(4), 115–133.
4. Deisenroth, M. P., Faisal, A. A., & Ong, C. S. (2020). *Mathematics for machine learning*. Cambridge University Press.
5. Nwankpa, C., Ijomah, W., Gachagan, A., & Marshall, S. (2018). Activation functions: Comparison of trends in practice and research for deep learning. *Preprint arXiv:1811.03378*.

Evaluation Criteria and Model Selection 2

2.1 Introduction

When a building is built, it demands much more than just bricks and cement. Solid planning and analytical studies are followed by a rough sketch, simulations, editing, and many other stages before the foundation is set. Similarly, machine learning (ML) is not only about algorithms and models, but it also requires testing the models, choosing the suitable algorithm, observing the performance, minimizing the errors, optimizing the model, and many other criteria. A crucial step in the ML workflow is appropriate model selection, and this process depends on various *evaluation criteria*. In common practice, the ML model is selected by comparing different potential model candidates for a given task. This comparison is based on pre-defined criteria, which differ from task to task. The success of an ML-based solution for a particular application depends on finding the best possible ML model. So, choosing the appropriate evaluation criteria is of paramount importance. In this chapter, we are going to study the evaluation criteria for an ML model and know why they are important. Moreover, we shall also discuss the process of choosing an ML algorithm for creating a model and the key factors to be analyzed before choosing an algorithm.

2.2 Error Criteria

The concept of errors and the different types of errors have been introduced in Sect. 1.5.2.6 in Chap. 1. Error criteria or loss function is required in machine learning to evaluate model performance. Generally, a small value of error is desired. This small error value can sometimes be overlooked because it has an insignificant effect on model performance and produces almost accurate results. This small error value varies based on each specific problem and application. However, this negligible value is different for all problems. For example, a negligible error value

E. Hossain, *Machine Learning Crash Course for Engineers*,
https://doi.org/10.1007/978-3-031-46990-9_2

for house price prediction may be equivalent to a significant error for medical data prediction. In the following sections, we will discuss four types of errors and three types of loss functions in greater detail.

2.2.1 MSE

The mean squared error (MSE) is calculated by taking the mean of the sum of the square of the difference (error) between the actual value and predicted value of all data points. The MSE is also called *average squared error*. MSE is sensitive to outliers.

Suppose the total number of data points in a sample is n. The output form of the ML model for the ith data point is $y^{(i)}$, and the actual value is $\hat{y}^{(i)}$; then the MSE is calculated as shown in Eq. 2.1:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2. \tag{2.1}$$

2.2.2 RMSE

As the name implies, the root mean squared error (RMSE) is defined as the square root of the MSE. So, Eq. 2.2 calculates the RMSE from Eq. 2.1. As the square of the error is taken in the MSE, it gives value in positive real number; thus, the square root of the MSE is also a real number. RMSE is also sensitive to outliers.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2}. \tag{2.2}$$

2.2.3 MAE

The mean absolute error (MAE) is formulated as the mean of the sum of the absolute value of the difference between the actual value and predicted value of all data points. Compared to MSE, rather than taking the square of the error, the absolute value of the error is considered in MAE, as shown in Eq. 2.3. MAE is robust to the negative effects of the presence of outliers.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y^{(i)} - \hat{y}^{(i)}|. \tag{2.3}$$

2.2.4 MAPE

In calculating the mean absolute percentage error (MAPE), the ratio of the difference between the actual value and predicted value to the actual value is calculated first. Then, the percentage of the average of the sum of these ratios over all the data points is called MAPE. It is also known as the *mean absolute percentage deviation (MAPD)*. MAPE is also robust to the negative effects of the presence of outliers.

$$MAPE = \frac{100}{n} \sum_{i=1}^{n} \left| \frac{y^{(i)} - \hat{y}^{(i)}}{\hat{y}^{(i)}} \right|. \tag{2.4}$$

In conclusion, *in the presence of outliers, MAE and MAPE are preferable to be used as error criteria. MSE and RMSE should be avoided because of their sensitivity to outliers*. We want to select the error criteria based on the dataset and algorithms we want to use. Sensitivity toward any elements in the dataset or algorithms, such as outliers, may result in undesirable outputs.

2.2.5 Huber Loss

Peter Jost Huber, a Swiss statistician, proposed this loss function in 1964 [1]. It is used mainly for regression problems. This function is less sensitive to data outliers than squared errors (MSE, RMSE) because the squared error is taken only at specific intervals. This problem is resolved by combining the squared error function and absolute value function in a piecewise function format as follows:

$$L = \frac{1}{n} \begin{cases} \frac{1}{2}(y^{(i)} - \hat{y}^{(i)})^2, & for\ |y^{(i)} - \hat{y}^{(i)}| \leq \delta; \\ \\ \sum_{i=1}^{n} \delta(|y^{(i)} - \hat{y}^{(i)}| - \frac{1}{2}\delta), & otherwise. \end{cases} \tag{2.5}$$

Here, δ is an arbitrary small value. The function above is quadratic in nature for small error values and shows linear characteristics for larger error values. The two parts of the function have equal value and slope at $|y^{(i)} - \hat{y}^{(i)}| = \delta$. This function and some variants can also be used for classification problems.

Example 2.1 The output of an ML model for a certain regression problem is $\hat{y} = [0.7, 1.1, 1.5, 1.9, 2.3, 2.7, 3.1]$. The true values corresponding to these predicted values are $y = [1.08, 1.2, 1.4, 2.1, 1.9, 7, 2.9]$. Determine the MSE, MAE, and Huber loss. Assume $\delta = 1.35$.

Solution to Example 2.1
Here, no of data points, $n = 7$.

$$
\begin{aligned}
MSE &= \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2,\\
&= \frac{\begin{array}{l}(1.08-0.7)^2+(1.2-1.1)^2+(1.4-1.5)^2+(2.1-1.9)^2\\ \quad +(1.9-2.3)^2+(7-2.7)^2+(2.9-3.1)^2\end{array}}{7},\\
&= \frac{18.894}{7},
\end{aligned}
$$

$$\therefore\ MSE = 2.699.$$

$$
\begin{aligned}
MAE &= \frac{1}{n}\sum_{i=1}^{n}|y^{(i)} - \hat{y}^{(i)}|,\\
&= \frac{\begin{array}{l}|1.08-0.7|+|1.2-1.1|+|1.4-1.5|+|2.1-1.9|+|1.9-2.3|\\ \quad +|7-2.7|+|2.9-3.1|\end{array}}{7},
\end{aligned}
$$

$$\therefore\ MAE = 0.811.$$

$$
\text{Huber loss, } L = \frac{1}{n}\begin{cases} \frac{1}{2}(y^{(i)} - \hat{y}^{(i)})^2, & for\ |y^{(i)} - \hat{y}^{(i)}| \le \delta;\\[2ex] \sum_{i=1}^{n}\delta(|y^{(i)} - \hat{y}^{(i)}| - \frac{1}{2}\delta), & otherwise.\end{cases}
$$

Here, only $|y_6 - \hat{y}_6| > \delta$, i.e., $|7 - 2.7| = 4.3 > 1.35$.

For this value, $L_6 = \delta\left(|y_6 - \hat{y}_6| - \frac{1}{2}\delta\right) = 1.35 \times \left(4.3 - \frac{1.35}{2}\right) = 4.894.$

Now, Huber loss

$$
\begin{aligned}
&= \frac{\begin{array}{l}0.5\{(1.08-0.7)^2+(1.2-1.1)^2+(1.4-1.5)^2+(2.1-1.9)^2\\ \quad +(1.9-2.3)^2+(2.9-3.1)^2\}+4.894\end{array}}{7},\\
&= \frac{0.2022+4.894}{7},
\end{aligned}
$$

$\therefore$ Huber loss $= 0.728$.

Programming Example 2.1

Example 2.1 is solved in Python using a user-defined function in Listing 2.1 followed by its output and explanation in Table 2.1. The code calculates the MSE and the MAE separately, using a user-defined function, to determine the value of the Huber loss. The Huber loss incorporates these two depending on the `delta` value.

```python
import numpy as np

# MSE function
def MSE(y, y_hat):

  y, y_hat = np.array(y), np.array(y_hat)            # Converting python list into numpy array
  dif = np.subtract(y, y_hat)                        # Subtraction operation
  squared_dif = np.square(dif)                       # Squaring the terms

  return np.mean(squared_dif)                        # Taking the mean of squared terms

# MAE function
def MAE(y, y_hat):

  y, y_hat = np.array(y), np.array(y_hat)

  return np.mean(np.abs(y - y_hat))                  # Taking the mean of the absolute values

# Huber Loss function
def huber_loss(y, y_hat, delta=1.0):

  y, y_hat = np.array(y), np.array(y_hat)
  huber_mse = 0.5*np.square(y - y_hat)               # MSE part of Huber Loss
  huber_mae = delta*(np.abs(y -y_hat) - 0.5*delta)  # MAE part of Huber Loss

  # Taking the mean of conditional error values
  return np.mean(np.where(np.abs(y-y_hat) <= delta, huber_mse, huber_mae))


y = [1.08, 1.2, 1.4, 2.1, 1.9, 7, 2.9]
y_hat = [0.7, 1.1, 1.5, 1.9, 2.3, 2.7, 3.1]

mse = MSE(y=y, y_hat=y_hat)
mae = MAE(y=y, y_hat=y_hat)
huber = huber_loss(y=y, y_hat=y_hat, delta=1.35)
print("MSE = {}\nMAE = {}\nHuber Loss = {}".format(mse, mae, huber))
```

Listing 2.1 Using a user-defined function in Python to solve Example 2.1

Output of Listing 2.1:

```
MSE = 2.6992
MAE = 0.8114285714285715
Huber Loss = 0.7279928571428573
```

Table 2.1 Explanation of Listing 2.1

Line number	Description
1	Importing NumPy library
4–10	Defining function to calculate MSE
13–17	Defining function to calculate MAE
20–27	Defining function to calculate Huber loss
30–31	Data points taken in array
33–35	Using the user-defined functions
36	Printing the results

Programming Example 2.2
Example 2.1 can also be solved using a pre-defined function in Listing 2.2 followed by its output and explanation in Table 2.2. The code uses pre-defined functions to execute the same tasks as in Listing 2.1.

```
import numpy as np
```

Listing 2.2 Using a pre-defined function to solve Example 2.1

Output of Listing 2.2:

```
MSE = 2.6992
MAE = 0.8114285714285715
Huber Loss = 0.7279928922653198
```

Table 2.2 Explanation of Listing 2.2

Line number	Description
1–3	Importing libraries
5–6	Data points taken in array
8–9	The arrays are converted into NumPy arrays
12–16	Calculations using pre-defined functions
18	Printing the results

2.2.6 Cross-Entropy Loss

The concept of cross-entropy comes from information theory, but it is extensively used in machine learning for classification problems. In 1948, a famous American mathematician, electrical engineer, and cryptographer, Claude Shannon, introduced the information entropy concept, also known as *Shannon entropy* [2]. The value of entropy of a random variable indicates the average "information" or "uncertainty" level of that particular random variable's possible outcomes. For example, if a discrete random variable X has $x_1, x_2, x_3, \ldots, x_n$ possible outcomes with a probability of $P(x_1)$, $P(x_2)$, $P(x_3), \ldots, P(x_n)$, the entropy of X will be as shown in Eq. 2.6:

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log P(x_i). \tag{2.6}$$

The base of the logarithm varies with applications. Usually, the base is taken as 2. A higher value of $H(X)$ indicates a higher level of uncertainty, i.e., information. From the expression of entropy, the cross-entropy loss can be defined as Eq. 2.7:

$$L_{CE} = -\sum_{i=1}^{n} y_i \log(\hat{y}_i), \tag{2.7}$$

where n is the number of classes. For binary classification, $n = 2$. So, Eq. 2.7 becomes Eq. 2.8:

$$L_{CE} = -\sum_{i=1}^{n} y_i \log(\hat{y}_i) = -[ylog(\hat{y}) + (1-y)\log(1-(\hat{y}))]. \tag{2.8}$$

The cross-entropy is regarded as a loss function in machine learning models for the classification task; hence, it is known as cross-entropy loss. The weights associated with machine learning classification models are updated by minimizing the cross-entropy loss to make the model's output as close as possible to original (truth) values.

Example 2.2 The output of a three-class classification model for a specific input is $\hat{y} = [0.02, 0.97835, 0.00165]$. The ground truth value corresponding to this output is $y = [0, 1, 0]$. Determine the log loss or cross-entropy loss.

Solution to Example 2.2

$$\begin{aligned}\text{Cross-entropy loss} &= -\sum_{i=1}^{n} y_i \log(\hat{y}_i),\\ &= -(0 \times \log(0.02) + 1 \times \log(0.97835)\\ &\quad + 0 \times \log(0.00165)),\\ &= -\log(0.97835),\\ &= 0.0095.\end{aligned}$$

So, the cross-entropy loss is 0.0095.

Programming Example 2.3
Example 2.2 is solved in Python using a user-defined function in Listing 2.3, followed by its output and explanation in Table 2.3. The code utilizes Eq. 2.7 to find out the categorical cross-entropy loss.

```python
import numpy as np

def categorical_cross_entropy(y, y_hat):

  y, y_hat = np.array(y), np.array(y_hat)
  ce = -np.sum(np.multiply(y,np.log10(y_hat)))

  return ce

y = [0, 1, 0]
y_hat = [0.02,0.97835,0.00165]
ce_loss = categorical_cross_entropy(y=y, y_hat=y_hat)

print("Cross-Entropy Loss = {}".format(ce_loss))
```

Listing 2.3 Using a user-defined function to solve Example 2.2

Output of Listing 2.3:

```
Cross-Entropy Loss = 0.009505750655915440
```

2.2.7 Hinge Loss

Hinge loss is often introduced to machine learning for classification tasks. Apart from errors, it is a great decision criterion for classification tasks, especially binary classification. It penalizes the misclassification to create a distinct margin between

Table 2.3 Explanation of Listing 2.3

Line number	Description
1	Importing library function
3–8	Defining the function
10–11	Data points taken in array
12	Calculations using the user-defined function
14	Printing the results

different classes. Even if the classification result is correct, hinge loss will still incur a penalty if the margin from the decision boundary is not large enough.

This loss function or error criterion is particularly used in maximal-margin classification for support vector machine (SVM) classifiers, although it has some usage in neural networks [3]. The hinge loss can be defined as shown in Eq. 2.9:

$$l(y) = \max(0, 1 - t.y), \tag{2.9}$$

where t is the truth label that should be mapped in the $\{-1,1\}$ range. y is the output of the SVM classifier. So, the output of linear SVM is given by Eq. 2.10:

$$y = wx + b, \tag{2.10}$$

where x and b are the parameters of the hyperplane. A more detailed explanation of SVM is given in Chap. 3. If $|y| \geq 1$, and t and y have the same symbols, then $l(y) = 0$, i.e., the loss becomes 0. On the other hand, if $|y| < 1$, then the loss increases linearly with y regardless of their symbol.

2.3 Distance Metrics

Distance metrics can be used to get an idea about the distance between two places, points, objects, etc. For example, distance metrics can tell how close or how far two points are from each other. Furthermore, some machine learning algorithms, such as KNN classification, K-means clustering, self-organizing map (SOM), SVM, etc., rely on the closeness between data points. Therefore, this "closeness" estimation can be considered a "distance" calculation. For this reason, knowing different distance metrics used in machine learning is essential. In the following sections, we are going to explore seven types of distance metrics with relevant diagrams and equations.

2.3.1 Euclidean Distance

The most popularly used distance metric is Euclidean distance. The Euclidean function calculates the square root of the sum of the differences between two different data objects. Assume two data objects a and b, where each data object has

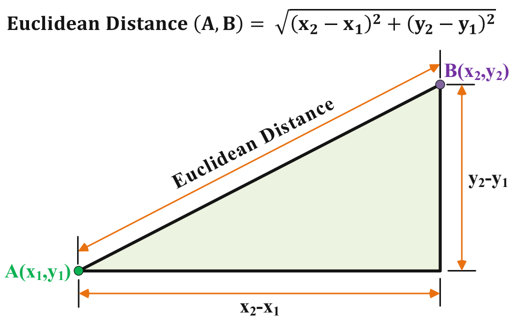

Euclidean Distance $(A, B) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
B(x2,y2)
Euclidean Distance
y2-y1
A(x1,y1)
x2-x1

Fig. 2.1 Euclidean distance between two points on a 2-D plane

n attributes, $a = (p_1, p_2, p_3, \dots, p_n)$ and $b = (q_1, q_2, q_3, \dots, q_n)$. The function to calculate the Euclidean distance between a and b is given in Eq. 2.11.

$$d(a, b) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}. \tag{2.11}$$

Figure 2.1 depicts the Euclidean distance between two data points $A(x_1, y_1)$ and $B(x_2, y_2)$ with two features x and y. Many algorithms such as the k-nearest neighbors algorithm and k-means algorithm use Euclidean distance as the objective function where the data dimension is low. However, in the case of higher dimensionality, the Euclidean distance may not perform effectively.

2.3.2 Cosine Similarity and Cosine Distance

Cosine similarity gives an estimation of the similarity between two or more vectors. The mathematical concept of cosine similarity is quite simple. It is the cosine of the angle between two vectors. Cosine similarity can be calculated by dividing the dot product of two vectors by the product of the magnitude of the two vectors. If θ is the angle between two vectors $\overrightarrow{A}$ and $\overrightarrow{B}$, the cosine similarity can be calculated using Eq. 2.12.

$$\cos(\theta) = \frac{\overrightarrow{A}\,.\,\overrightarrow{B}}{||\overrightarrow{A}||\;||\overrightarrow{B}||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}. \tag{2.12}$$

The two vectors $\overrightarrow{A}$ and $\overrightarrow{B}$ are non-zero, and they exist within an inner product space. The value of the cosine similarity metric ranges from -1 to 1, where -1 means total dissimilarity and 1 means total similarity between the two vectors. Cosine distance can be calculated from cosine similarity. Cosine distance is opposite to cosine similarity. The greater the distance, the less the similarity between data objects. The cosine distance is calculated as shown in Eq. 2.13.

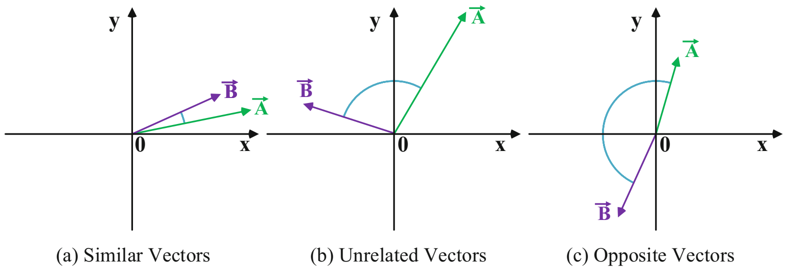


Fig. 2.2 Cosine similarity between two (**a**) similar, (**b**) unrelated, and (**c**) opposite vectors. The similar vectors have nearly 0° angle between them. The angle between unrelated vectors would be around 90° or a bit more. Two opposite vectors would have nearly 180° angle between them. There is no correlation between the length of the vectors and their cosine similarity

$$d(\vec{A}, \vec{B}) = 1 - \cos(\theta) = 1 - \frac{\vec{A} . \vec{B}}{|| \vec{A} || \, || \vec{B} ||}. \tag{2.13}$$

Figure 2.2 depicts the different cases of determining cosine similarity. Cosine similarity is preferable when two data objects vary in their size. The Euclidean distance between these two data objects could be huge due to their length mismatch, even if they have similarities. This similarity can be captured by cosine similarity because the angle remains constant regardless of the size of the data objects.

2.3.3 Manhattan Distance

Another popular distance metric is the Manhattan distance, also referred to as *city block distance* or *taxicab metric*. The intuition behind these names derives from calculating the shortest distance between any two blocks along the vertical and horizontal axes. The summation of absolute differences between two points is the Manhattan distance. The formula for calculating the Manhattan distance between two data points $a = (x_1, x_2, x_3, \ldots, x_n)$ and $b = (y_1, y_2, y_3, \ldots, y_n)$ is given by Eq. 2.14.

$$d(a, b) = |x_1 - y_1| + |x_2 - y_2| + \cdots + |x_n - y_n|. \tag{2.14}$$

Figure 2.3 depicts the Manhattan distance between two data points A and B with two features, x and y. The Manhattan distance is preferred over the Euclidean distance in the case of higher dimensionality. It also works best for discrete attributes.

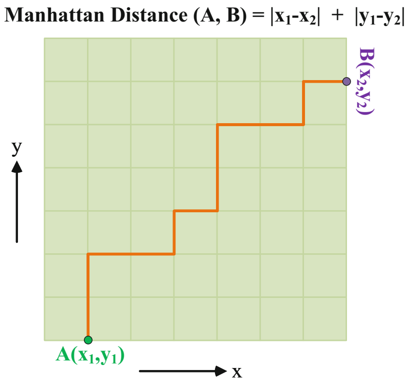


Fig. 2.3 Manhattan distance between two points on a 2-D plane

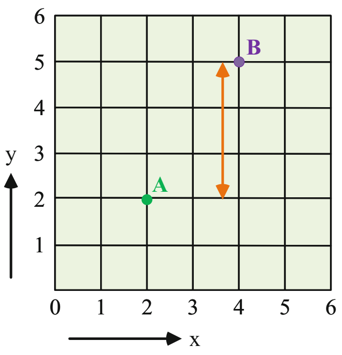


Fig. 2.4 Chebyshev distance between two points(A(2,2) and B(4,5)) on a 2-D plane

2.3.4 Chebyshev Distance

Chebyshev distance, also referred to as the *supremum distance*, is the maximum difference in the attributes between two data objects. The formula for calculating the Chebyshev distance between two data points $a = (x_1, x_2, x_3, \ldots, x_n)$ and $b = (y_1, y_2, y_3, \ldots, y_n)$ is given by Eq. 2.15.

$$d(a, b) = \max(|x_1 - y_1|,\ |x_2 - y_2|, \ldots,\ |x_n - y_n|). \tag{2.15}$$

Figure 2.4 depicts the Chebyshev distance for two points A and B. The Chebyshev distance can be applied to specific logistical problems only. For instance, it can determine the minimum moves required by the king on a chessboard to move from one square to another.

2.3.5 Minkowski Distance

A more generalized form of the Euclidean distance and the Manhattan distance is the Minkowski distance. The formula for calculating the Minkowski distance between

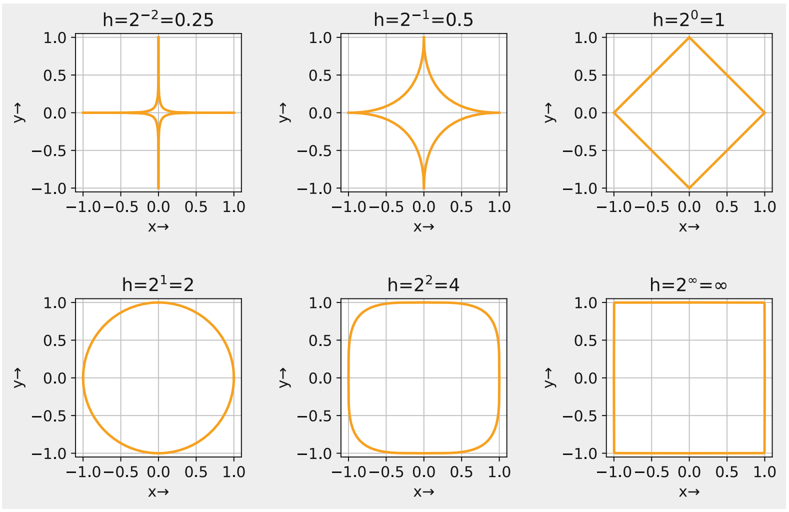


Fig. 2.5 The change in unit circles with various values of h in Minkowski distance. The path of the unit circles represents points with the same Minkowski distance

two data points $a = (x_1, x_2, x_3, \ldots, x_n)$ and $b = (y_1, y_2, y_3, \ldots, y_n)$ is given by Eq. 2.16.

$$d(a, b) = \sqrt[h]{(x_1 - y_1)^h + (x_2 - y_2)^h + \cdots + (x_n - y_n)^h}, \tag{2.16}$$

where $h \geq 1$. For $h = 1$, *the formula for the Minkowski distance calculates the Manhattan distance, and for* $h = 2$, *it calculates the Euclidean distance.* Figure 2.5 shows the results for different values of h between two data points A and B with two features x and y.

Let us now explore two mathematical examples related to the five distance metrics that we have discussed so far.

Example 2.3 A(3,4) and B(5,2) are two points on a 2-dimensional XY plane. Determine the Euclidean distance, Manhattan distance, Chebyshev distance, cosine similarity, and Minkowski distance between them. Take $h = 2$ for Minkowski distance.

Solution to Example 2.3
Euclidean distance, $d_e(A, B) = \sqrt{(3-5)^2 + (4-2)^2} = 2\sqrt{2}$.

Manhattan distance, $d_m(A, B) = |3 - 5| + |4 - 2| = 4$.

Chebyshev distance, $d_c(A, B) = \max(|3 - 5|, |4 - 2|) = 4$.

Cosine similarity, $d_{\cos}(A, B) = \cos(\theta) = \dfrac{\vec{A}\,.\,\vec{B}}{||\vec{A}||\;||\vec{B}||}$.

Now,

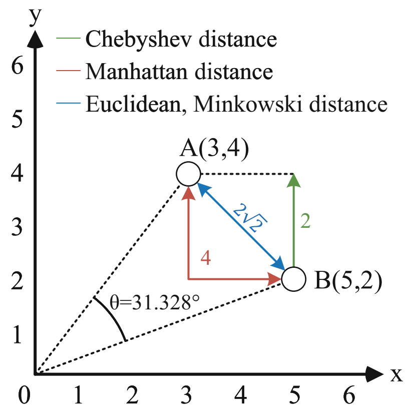


Fig. 2.6 Various distance metrics between A(3,4) and B(5,2)

$||\overrightarrow{A}|| = \sqrt{3^2 + 4^2} = 5$ and $||\overrightarrow{B}|| = \sqrt{5^2 + 2^2} = \sqrt{29}$.
$\overrightarrow{A} . \overrightarrow{B} = (3 \times 5) + (4 \times 2) = 15 + 8 = 23$.
So, $\cos(\theta) = \dfrac{23}{5 \times \sqrt{29}} \approx 0.8542$.
$\therefore \theta = \cos^{-1}(0.8542) = 31.328°$.
Cosine distance, $d_{cosine}(A, B) = 1 - \cos(\theta) = 1 - 0.8542 = 0.1458$.
Minkowski distance,
$d_{mk}(A, B) = \sqrt[h]{|3-5|^h + |4-2|^h} = \sqrt{|3-5|^2 + |4-2|^2} = 2\sqrt{2}$.
Here, the Minkowski distance and the Euclidean distance are equal.
Figure 2.6 illustrates the different distance metrics of Example 2.3.

Example 2.4 $A = [1, 3, 5, 7, 9]$ and $B = [2, 4, 6, 8, 12]$ are two data points. Determine the Euclidean distance, Manhattan distance, Chebyshev distance, cosine similarity, and cosine distance between this pair of data points.

Solution to Example 2.4
Euclidean distance,
$d_e(A, B) = \sqrt{(1-2)^2 + (3-4)^2 + (5-6)^2 + (7-8)^2 + (9-12)^2} = \sqrt{13} = 3.605$.

Manhattan distance,
$d_m(A, B) = |1-2| + |3-4| + |5-6| + |7-8| + |9-12| = 7$.

Chebyshev distance,
$d_c(A, B) = \max(|1-2|, |3-4|, |5-6|, |7-8|, |9-12|) = \max(1, 1, 1, 1, 3) = 3$.

Cosine similarity, $d_{\cos}(A, B) = \cos(\theta) = \dfrac{\overrightarrow{A} . \overrightarrow{B}}{||\overrightarrow{A}|| \, ||\overrightarrow{B}||}$.

Now,
$||\overrightarrow{A}|| = \sqrt{1^2 + 3^2 + 5^2 + 7^2 + 9^2} = \sqrt{165}$.
$||\overrightarrow{B}|| = \sqrt{2^2 + 4^2 + 6^2 + 8^2 + 12^2} = 2\sqrt{66}$.

$\overrightarrow{A}.\overrightarrow{B} = (1 \times 2) + (3 \times 4) + (5 \times 6) + (7 \times 8) + (9 \times 12) = 208.$

So, $\cos(\theta) = \dfrac{208}{\sqrt{165} \times 2\sqrt{66}} \approx 0.997.$

$\therefore \theta = \cos^{-1}(0.997) = 4.728^\circ.$

Cosine distance, $d_{cosine}(A, B) = 1 - \cos(\theta) = 1 - 0.997 = 0.003.$

Programming Example 2.4

Instead of doing all the math by ourselves, we can use Python to do the task more quickly. Listing 2.4 demonstrates a Python program with a user-defined function to solve Example 2.4, followed by the output of the program and the explanation in Table 2.4. The listing generates user-defined functions for calculating the Euclidean distance, cosine similarity, and the cosine distance between two vectors, A and B. The Manhattan and Chebyshev distances are not shown in the program and have been left as an exercise for the readers.

```python
import numpy as np
from numpy.linalg import norm

# Euclidean Distance function
def euclidean_distance(A, B):

  A, B = np.array(A), np.array(B)
  sum = np.sum(np.square(A - B))

  return np.sqrt(sum)

# Cosine Similarity function
def cosine_similarity(A, B):

  A, B = np.array(A), np.array(B)
  cosine = np.dot(A, B)/(norm(A)*norm(B))

  return cosine

# Implement the rest of metrics yourself

A = [1, 3, 5, 7, 9]
B = [2, 4, 6, 8, 12]

euc = euclidean_distance(A=A, B=B)
cos = cosine_similarity(A=A, B=B)
d_cos = 1 - cos                         # Cosine distance = 1-cos
    (theta)

print("Euclidean Distance = {}\nCosine Similarity = {}\nCosine
    distance = {}".format(euc, cos, d_cos))
```

Listing 2.4 Using a user-defined function to solve Example 2.4

Table 2.4 Explanation of Listing 2.4

Line number	Description
1–2	Importing libraries.
4–10	Defining function for calculating Euclidean distance
13–18	Defining function for calculating cosine similarity
22–23	Data points taken in array
25–27	Calculations using the user-defined function
29	Printing the results

Output of Listing 2.4:

```
Euclidean Distance = 3.605551275463989
Cosine Similarity = 0.9965965959318528
Cosine distance = 0.003403404068147209
```

2.3.6 Hamming Distance

The similarity between two strings of equal length can be determined by the Hamming distance. This metric can be applied to only binary features. If two equal-length strings can be assumed as rows of binary (0 or 1) features, the distance or similarity between them can be calculated by comparing each binary string. It is done by counting the number of positions with different string values.

Practically, the Hamming distance is calculated by performing a logical XOR operation between the two strings and summing the non-zero values from the resulting string. Then, the Hamming distance is normalized by dividing the distance by the length of the strings to have a generalized estimation of the Hamming distance independent of the length of the strings. For example, we have two binary strings of the same length—00110110 and 10111011. The logical XOR operation between them will be

$$00110110 \oplus 10111011 = 10001101. \tag{2.17}$$

So, the hamming distance will be

$$d(00110110,\, 10111011) = sum(10001101) = 4. \tag{2.18}$$

This is also shown in Fig. 2.7. The normalized Hamming distance will be

$$d(00110110,\, 10111011) = \frac{d(00110110,\, 10111011)}{length\ of\ string} = \frac{4}{8} = 0.5. \tag{2.19}$$

In machine learning practice, Hamming distance is calculated from natural language strings. For example, the two strings “Euclidean” and “Manhattan” have

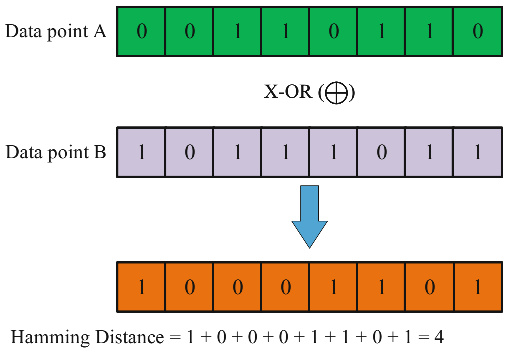


Fig. 2.7 The Hamming distance between two data points

the same length. Therefore, if we consider the Hamming distance output for each similar string element pair as 0 and each different string element pair as 1, then the Hamming distance will be

$$d(Euclidean, Manhattan) = sum(111111100) = 7. \tag{2.20}$$

This equals the number of different letter pairs between the two strings (Euclidean, Manhattan).

Programming Example 2.5
This calculation uses Python in Listing 2.5, followed by its output and explanation in Table 2.5. This code generates a function to calculate the Hamming distance between two strings. Then it uses the function to find the Hamming distance between strings A and B.

```python
# Function for Hamming distance calculation

def hamming_distance(s1, s2):
  dist = 0
  if len(s1)!=len(s2):
      print("Strings are not equal")
  else:
      for x, (i, j) in enumerate(zip(s1, s2)):
          if i != j:
              print(f'Characters do not match {i,j} in {x}')
              dist+=1
  return f'Hamming Distance = {dist}'

A = "Euclidean"
B = "Manhattan"

hamming = hamming_distance(A, B)
print(hamming)
```

Listing 2.5 Using a user-defined function to calculate Hamming distance

Table 2.5 Explanation of Listing 2.5

Line number	Description
3–12	Defining function for calculating Hamming distance
14–15	Data points A and B to be checked
17	Calculations using the user-defined function
18	Printing the results

Output of Listing 2.5:

```
Characters do not match ('E', 'M') in 0
Characters do not match ('u', 'a') in 1
Characters do not match ('c', 'n') in 2
Characters do not match ('l', 'h') in 3
Characters do not match ('i', 'a') in 4
Characters do not match ('d', 't') in 5
Characters do not match ('e', 't') in 6
Hamming Distance = 7
```

2.3.7 Jaccard Similarity and Jaccard Distance

In 1910, Paul Jaccard, a botany and plant physiology professor at ETH Zurich, published a similarity coefficient called *Coefficient de Communauté*, now known as Jaccard Similarity Index. This distance metric measures the similarity between finite sets. It is defined as the ratio between the intersection size and the union size between two sets. This can also be referred to as Intersection over Union (IoU), as shown in Fig. 2.8. For example, for two finite sets A and B, the Jaccard similarity, $J(A, B)$, is calculated as shown in Eq. 2.21.

$$J(A, B) = IoU = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}. \tag{2.21}$$

Certainly, $0 \leq J(A, B) \leq 1$, where 0 means no similarity at all and 1 means complete similarity. If both A and B are empty sets, i.e., $A = \varnothing$ and $B = \varnothing$, then $J(A, B) = 1$. The Jaccard distance, $d(A, B)$, can be calculated from the Jaccard similarity as shown in Eq. 2.22.

$$d(A, B) = 1 - J(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}, \tag{2.22}$$

where $0 \leq d(A, B) \leq 1$. A Jaccard distance of 0 means complete similarity, and a Jaccard distance of 1 means no similarity. If $A = \varnothing$ and $B = \varnothing$, then $d(A, B) = 0$.

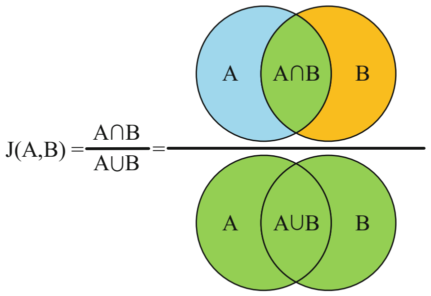


Fig. 2.8 Visualization of Jaccard similarity

In natural language processing (NLP), the Jaccard similarity determines the similarity between two sentences or documents from the total number of each word. Jaccard similarity is used in object identification applications to determine the bounding box accuracies.

Let us now have a look at Example 2.5, which is a problem with Jaccard similarity and Jaccard distance.

Example 2.5 Two commonly used sample sentences in NLP for determining similarity are "This is a foo bar sentence." and "This sentence is similar to a foo bar sentence." Determine the Jaccard similarity and Jaccard distance. Ignore the words "is," "a," and "to" as they do not have much influence on similarity.

Solution to Example 2.5

Set of words in the first sentence, $S_1 = \{"This", "foo", "bar", "sentence", "."\}$.

Set of words in the second sentence, $S_2 = \{"This", "sentence", "similar", "foo", "bar", "."\}$.

Now, $S_1 \cap S_2 = \{"This", "foo", "bar", "sentence", "."\}$.

There are 5 items in set $S_1 \cap S_2$. $\therefore |S_1 \cap S_2| = 5$.

Similarly, $S_1 \cup S_2 = \{"This", "similar", "foo", "bar", "sentence""."\}$
$\therefore |S_1 \cup S_2| = 6$.

Therefore, Jaccard similarity, $J(S_1, S_2) = \dfrac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = \dfrac{5}{6} = 0.833$.

And Jaccard distance, $d(S_1, S_2) = 1 - J(S_1, S_2) = 1 - 0.833 = 0.167$.

Programming Example 2.6

Example 2.5 is solved in Python in Listing 2.6 using a user-defined function. The output of the code is also provided, and the explanation can be found in Table 2.6. The code uses the Natural Language Toolkit (NLTK) library to successfully generate the Jaccard similarity index and Jaccard distance between two input sentences A and B.

```python
# Importing Natural Language Toolkit library
# NLTK associated packages are downloaded

import nltk
nltk.download('punkt')
nltk.download('stopwords')

from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

def jaccard_similarity(s1, s2):

  # Tokenizing sentences, i. e., splitting the sentences into
    words
  S1_list = word_tokenize(s1)
  S2_list = word_tokenize(s2)

  # Getting the English stopword collection
  sw = stopwords.words('english')

  # Creating word sets corresponding to each sentence
  S1_set = {word for word in S1_list if not word in sw}
  S2_set = {word for word in S2_list if not word in sw}

  print(f'Word set Sentence 1 = {S1_set}')
  print(f'Word set Sentence 2 = {S2_set}')

  I = set(S1_set).intersection(set(S2_set))      # Intersection
    operation
  U = set(S1_set).union(set(S2_set))             # Union operation

  print(f'Intersection = {I}')
  print(f'Union = {U}')

  IoU = len(I)/len(U)                            # Intersection
    over Union (IoU)

  return IoU

A = "This is a foo bar sentence ."
B = "This sentence is similar to a foo bar sentence ."

J = jaccard_similarity(A, B)
d_J = 1 - J                                              # Jaccard
     distance = 1 - J
print("Jaccard Similarity Index = {}\nJaccard distance = {}".
    format(J, d_J))
```

Listing 2.6 Using a user-defined function to solve Example 2.5

Output of Listing 2.6:

```
Word set Sentence 1 = {'bar', 'sentence',
                       'This', '.', 'foo'}
```

Table 2.6 Explanation of Listing 2.6

Line number	Description
1–2	Importing libraries
4–10	Defining function for calculating Euclidean distance
13–18	Defining function for calculating cosine similarity
22–23	Data points taken in array
25–27	Calculations using the user-defined function
29	Printing the results

```
Word set Sentence 2 = {'similar', 'bar', 'sentence',
                       'This', '.', 'foo'}
Intersection = {'bar', 'sentence', 'This', '.', 'foo'}
Union = {'similar', 'bar', 'sentence', 'This', '.',
       'foo'}
Jaccard Similarity Index = 0.8333333333333334
Jaccard distance = 0.16666666666666663
```

2.4 Confusion Matrix

A confusion matrix gives a comprehensible understanding of the performance of a given classification model. Whether an evaluation score is meaningful or not can be understood using a confusion matrix. For instance, an accuracy score of 90% would imply that the performance of the model is very high and efficient. However, consider the case that the model perfectly predicts all 90 data from one class and entirely fails to predict the remaining 10 data from another class. This case would prove that a model with an accuracy score of 90% can still be irrelevant and wrong. In cases like this, the confusion matrix helps depict the actual scenario.

The terms used in the confusion matrix in Fig. 2.9 are described below:

1. **True Positive (TP):** These are the positive cases, and the model correctly predicted them as positive cases.
2. **False Positive (FP):** These are the cases that are not positive, but the model predicted them as positive. This error is a type 1 error.
3. **True Negative (TN):** These are the negative cases, and the model correctly predicted them as negative cases.
4. **False Negative (FN):** These are the cases that are actually positive, but the model has incorrectly predicted them as negative cases. This error is a type 2 error.

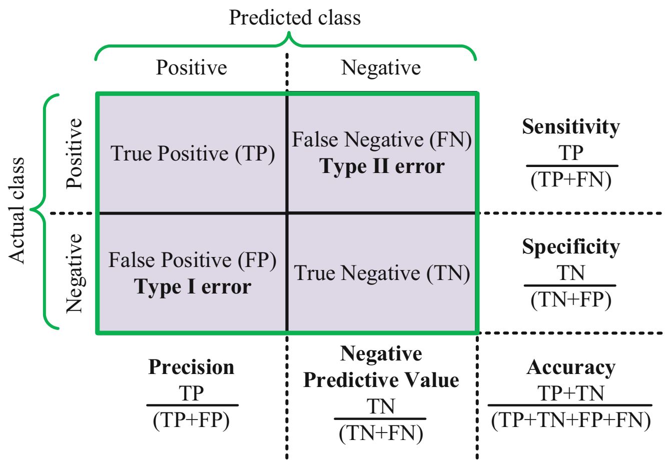


Fig. 2.9 Confusion matrix and the metrics associated with it

2.4.1 Accuracy

Accuracy is the simplest form of evaluation score. It is defined by the number of correctly predicted observations over the total number of observations. The accuracy score is valid when the dataset is balanced; each class in the dataset has an equal number of data objects. However, for an imbalanced dataset, the accuracy score can be meaningless as it does not provide a detailed insight into the model's performance. The formula for calculating accuracy for any model from the confusion matrix is shown in Eq. 2.23:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}. \tag{2.23}$$

2.4.2 Precision and Recall

Suppose a case of an imbalanced dataset is given, where 98 data belong to a positive class, and 2 data belong to a negative class. Now, if the model trained can predict all the 98 data of the positive class correctly but fails to predict the 2 data of the negative class, then the accuracy score of the model will be 98%. Here, an accuracy score of 98% gives a false sense of an effective and efficient model, but the model fails to predict an entire class in reality. In such circumstances, precision and recall scores are preferable over accuracy.

2.4.2.1 Precision

The precision score is used to understand the ratio of correctly predicted positive cases among all the positive cases predicted by the model. It is used when the system is required to have a low false-positive rate. For instance, a rescue team must detect injured human beings in a disaster area. Sending help to areas where incorrectly detected injured human beings would waste resources, time, and energy. Therefore, the system must have a low false-positive rate for an efficient and effective rescue operation. Higher precision means a lower false-positive rate.

The precision score is calculated as shown in Eq. 2.24 by dividing the number of true positives (TP) by the summation of the number of true positives and the number of false positives (FP):

$$Precision = \frac{TP}{TP + FP}. \tag{2.24}$$

2.4.2.2 Recall

The recall score, also called *sensitivity*, is used to calculate the true-positive rate. It is calculated as the ratio of correctly predicted positive cases to all the cases in the actual positive class. It is used when the system is required to have a low false-negative rate. For instance, if an injured human being goes undetected by the system, they will not be helped by the rescue team, which is a considerable risk. Sometimes a rescue mission can afford some false positives (detecting non-injured human beings as injured) but not false negatives (failing to detect actual injured human beings). A higher recall score means a lower false-negative rate. Mathematically, recall can be defined as shown in Eq. 2.25:

$$Recall = \frac{TP}{TP + FN}. \tag{2.25}$$

There is a general trade-off between precision and recall. A system cannot simultaneously have a high precision score and a high recall score. A high precision leads to a poor recall score and vice versa. Figure 2.10 demonstrates the distinction between precision and recall.

2.4.3 F1 Score

The F1 score is the harmonic mean of precision and recall score. It is used when both the precision and recall scores need to be considered for the system. For instance, Fig. 2.10 shows that neither a high precision nor a high recall is best for a rescue mission. In cases like this, the F1 score best evaluates the model's performance. The best value of the F1 score of a model is 1. The formula to calculate the F1 score is shown in Eq. 2.26:

$$F1\ score = 2 \times \frac{Precision \times Recall}{Precision + Recall}. \tag{2.26}$$

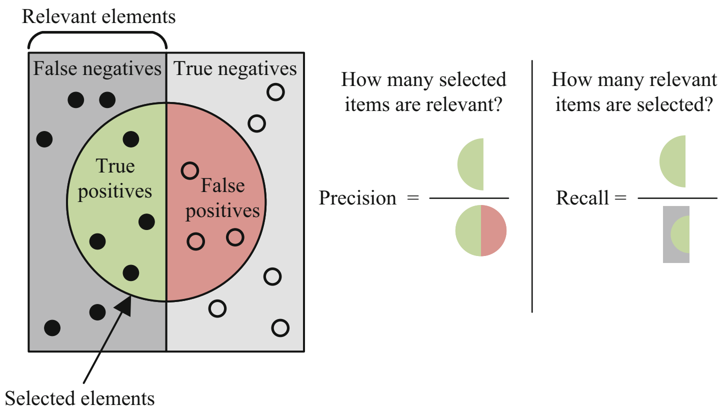


Fig. 2.10 Distinction between precision and recall

Now we will study an example related to the confusion matrix, accuracy, precision, recall, and F1 score.

Example 2.6 Suppose we have built a cat–dog classifier model. For testing the model, we have given it 100 images of cats and dogs containing 50 images of cats and 50 images of dogs. 48 images of cats have been classified correctly, while the other 2 are classified as dogs. On the other hand, only 30 images of dogs are classified correctly, while the other 20 images are classified as cats.

Develop a confusion matrix from this model output. What is the accuracy of this classifier model? Also determine the precision, recall, and F1 score of this model.

Solution to Example 2.6
Let us assume that "Cat" is the positive class and "Dog" is the negative class. So:

- True positive, TP = is cat and is classified as cat = 48.
- True negative, TN = is not cat and is not classified as cat = is dog and is classified as dog = 30.
- False positive, FP = is not cat but is classified as cat = 50 − 30 = 20.
- False negative, FN = is cat but is classified as dog = 2.

So, the confusion matrix is: $\begin{pmatrix} TP & FN \\ FP & TN \end{pmatrix} = \begin{pmatrix} 48 & 2 \\ 20 & 30 \end{pmatrix}$.

Accuracy = $\frac{TP + TN}{TP + TN + FP + FN} = \frac{48 + 30}{48 + 30 + 20 + 2} = \frac{78}{100} = 0.78 \approx$ 78%.

Precision = $\frac{TP}{TP + FP} = \frac{48}{48 + 20} = \frac{48}{68} = 0.706 \approx 70.6\%$.

Recall = $\frac{TP}{TP + FN} = \frac{48}{48 + 2} = \frac{48}{50} = 0.96 \approx 96\%.$

F1 score = $2 \times \frac{Precision \times Recall}{Precision + Recall} = 2 \times \frac{0.706 \times 0.96}{0.706 + 0.96} = 0.814.$

The model seems to perform better for cat images than dog images. For this reason, the precision score of this model is somewhat lower, even though it has a higher recall score.

2.5 Model Parameter and Hyperparameter

The term *learning* in ML indicates that an ML model learns something from input data. However, what is learned in an ML model? For example, in regression models, linear or logistic regression coefficients are learned; the weights and biases are learned in neural networks; cluster centroids of clustering algorithms are learned, and so on. These learned coefficients, weights, bias, cluster centroids, etc., are known as the *parameters* in ML. Parameters are said to be internal to the model as the value of parameters is supposed to change during training.

On the other hand, model hyperparameters cannot be estimated from data. The learning process of an ML algorithm depends on model hyperparameters. It directly controls the learned parameters of an ML model. Hyperparameters can be considered external to the model because the hyperparameter values are set before model training starts, and they remain constant throughout the training process. Batch size, pooling size, kernel size, the number of iterations or epochs, choice of optimization algorithms and learning rate of optimization algorithms, the number of clusters in clustering algorithms, choice of loss/cost function, choice of activation function, and the number of hidden layers in a neural network, etc., are regarded as model hyperparameters.

2.6 Hyperparameter Space

Different ML algorithms come with different types of hyperparameters. The hyperparameters associated with different ML algorithms are as follows:

- Polynomial Regression:
  - Degree of polynomials
- Logistic Regression:
  - Regularization
  - Class weights
- Decision Tree and Random Forest:
  - Maximum tree depth
  - Minimum split samples
  - Minimum number of leaf samples
  - Maximum number of features

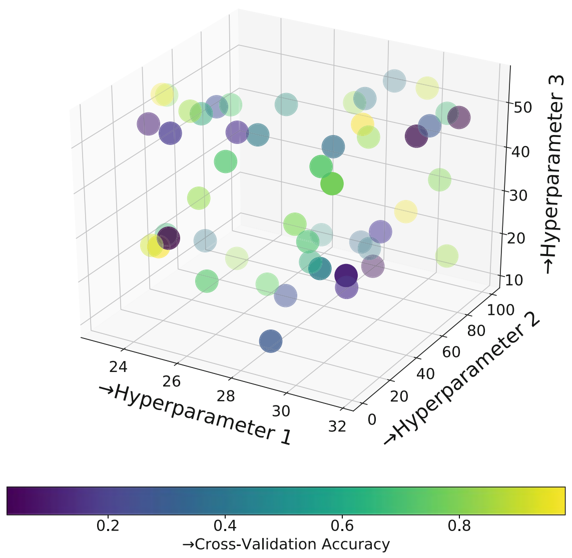


Fig. 2.11 Hyperparameter space consisting of three different hyperparameters. Points indicating higher cross-validation accuracy are desired

  - Split quality criterion
- Neural Network:
  - Number of layers
  - Number of neurons
  - Activation function
  - Optimizer algorithm
  - Learning rate

If a system has several possible solutions, the set of those possible solutions is called the *search space*. Similarly, an ML model can be developed using many sets of hyperparameters. These sets of hyperparameters can be referred to as *hyperparameter space*, which is demonstrated in Fig. 2.11. The hyperparameter space can be visualized as an *n*-dimensional plot where *n* is the number of hyperparameters, and each axis represents each hyperparameter. Each point in the hyperparameter space represents a set of hyperparameters. The optimal point in this hyperparameter space is found by performing *hyperparameter tuning*.

2.7 Hyperparameter Tuning and Model Optimization

An ML model consists of some hyperparameters and parameters. The job of an ML model is to learn the parameters to give the correct hypothesis in a given context, but the structure of the ML model depends on the hyperparameters. It is evident that an optimized ML model is expected for the final application, so a proper choice of hyperparameters is required. Identifying the correct set of hyperparameters to achieve an optimized ML model is known as *hyperparameter tuning*. It plays a crucial part in building an ML model. The steps associated with hyperparameter tuning are given below:

1. Visualize the data and understand the problem.
2. Select the best possible ML algorithm suitable for that problem.
3. Split the dataset into three sets—train set, validation set, and test set.
4. Determine the list of parameters and create the hyperparameter space (HS).
5. Select the most suitable method for searching the optimal set of hyperparameters from the HS and apply that.
6. Implement cross-validation.
7. Evaluate the model score.
8. Repeat steps 5, 6, and 7 until the best possible model score is achieved. The hyperparameter set with the best model score is expected to be optimal.

Sometimes, ML models fail to perform well on real-world data even if they had performed well in the training stage. This *performance degradation* happens when the ML model has already learned some test data during the training stage. Data sharing between the train, validation, and test sets causes the model to overfit and, in turn, underperform in the application stage. This phenomenon is known as *data leakage*. For this reason, the whole dataset is split before performing cross-validation.

Some hyperparameter tuning and model optimization techniques are described in the following sections.

2.7.1 Manual Search

This is the most straightforward hyperparameter tuning technique. In this method, the hyperparameters are selected in a trial-and-error method. One can create a hyperparameter space and select the best possible hyperparameters from there. With proper expertise and experience, this method can sometimes help build an ML model quickly. This method also does not require any extra computational resources. However, this method is not recommended most of the time because the probability of achieving an optimized model using this method is low.

2.7.2 Exhaustive Grid Search

This is an exhaustive or brute-force search method. The steps followed in this method are as follows:

1. The hyperparameter space is created from all available sets of hyperparameters.

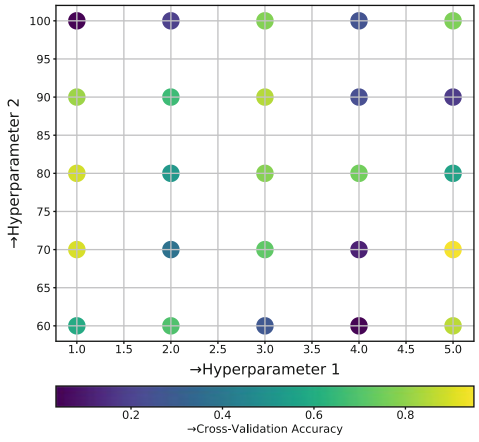


Fig. 2.12 Exhaustive grid search in a two-dimensional hyperparameter space

2. Cross-validation is performed on all the possible combinations of the hyperparameters.
3. The combination with the best evaluation metrics is picked to be the hyperparameter of the model.

The grid search method is time-consuming since it evaluates all possible combinations of hyperparameters. If the dataset is huge, it takes a lot of time, and an exhaustive search method is not recommended. This search method can find the best possible set of hyperparameters, but this might also make the ML model more prone to overfitting. The exhaustive grid search method is depicted in Fig. 2.12.

2.7.3 Halving Grid Search

The grid search method can be made faster by implementing a *successive halving algorithm (SHA)*. In this method, the hyperparameters are evaluated using fewer resources first, i.e., the training process of the ML model is stopped early so that fewer computing resources are required. Then half of the hyperparameter sets are eliminated based on model performance. This process is repeated until one hyperparameter set remains. The computational resources are increased each time the process is repeated. This method requires significantly less time than only grid search, as more resources are allocated to more potential models.

2.7.4 Random Search

As the name implies, this method randomly selects different combinations of hyperparameters and performs cross-validation on the dataset instead of searching the entire hyperparameter space, as depicted in Fig. 2.13. It is faster than grid

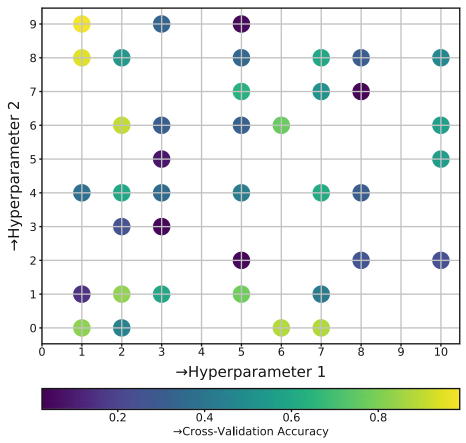


Fig. 2.13 Random search in a two-dimensional hyperparameter space

search, but sometimes it can miss the best combination of hyperparameters due to selecting randomly. It may still be very time-staking for massive datasets and hyperparameters, but it is less prone to model overfitting than an exhaustive grid search.

2.7.5 Halving Random Search

This searching method also utilizes a successive halving algorithm (SHA) like a halving grid search. The difference is that the hyperparameter sets are selected randomly from hyperparameter space. This method is also faster than any brute-force method.

2.7.6 Bayesian Optimization

A function that produces output without revealing its internal mechanisms is called a *black-box function*. Sometimes, ML models are considered black block functions. So, it brings a black-box optimization problem. If that black-box function is computationally cheaper, optimizing it using previous methods is convenient.

The Bayesian optimization technique becomes handy when the model or the black-box function is computationally expensive. Just as grid search and random search, the Bayesian optimization technique also trains multiple ML models from multiple hyperparameter sets from the hyperparameter space. However, instead of going through all hyperparameter sets or randomly picking them, Bayesian optimization considers the model performance on previous hyperparameter sets. It selects the future hyperparameters based on that information. This method is expected to perform much faster than the previously discussed methods.

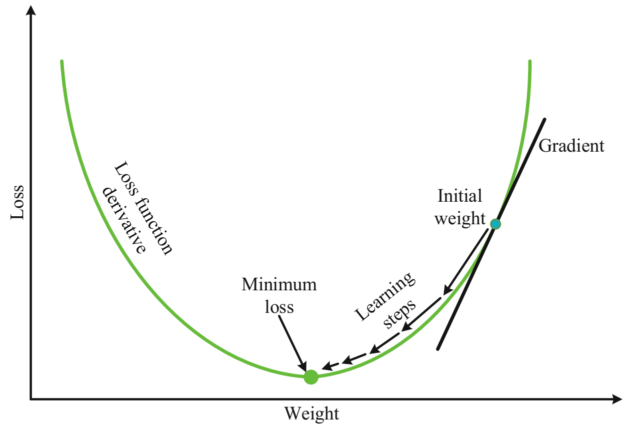


Fig. 2.14 Gradient descent visualization

2.7.7 Gradient-Based Optimization

The primary purpose of training specific ML models would be to achieve the point where the loss or cost function has the minimum value. This loss minimization is done in a gradient-based optimization technique. This method helps achieve an optimal set of parameters (not hyperparameters), i.e., weights, biases, etc.

For example, imagine a hiker has lost his sense of direction but can still sense whether he is approaching his destination or not. Based on his sense, he can adjust his path and eventually find his destination. This is done in gradient-based optimization as depicted in Fig. 2.14.

The loss values are plotted against a parameter, i.e., weight. The gradient of this loss curve is found by taking its derivative. The parameters are updated based on the gradient values. The minimum loss is found where the derivative is equal to or close to zero. The parameters corresponding to this minima are selected for the model. This is known as the *gradient descent algorithm*, which is used to apply gradient-based optimization. Based on the parameter updating techniques, there are many types of gradient descent (GD) algorithms, such as Batch/Vanilla Gradient Descent, Stochastic Gradient Descent (SGD), Mini-batch Gradient Descent, etc.

The convergence speed in GD algorithms largely depends on the *learning rate*, which can again be considered a hyperparameter. The learning rate determines the length of the incremental learning steps shown in Fig. 2.14.

2.7.8 Evolutionary Algorithm

The evolutionary algorithm mimics biological mechanisms. It solves problems by emulating the reproduction pattern of living organisms. It is mainly based on the Darwinian evolution theory. The solutions (i.e., model output due to a hyperparam-

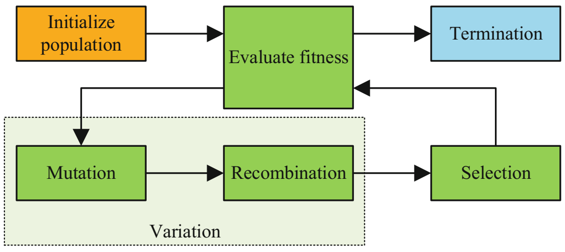


Fig. 2.15 Evolutionary algorithm process that mimics the biological process

eter set) may be considered individual organisms from a specific hyperparameter space. These individual organisms create a population sample. The evolutionary algorithms often include selection, reproduction, mutation, and recombination functions. The population is then evaluated for fitness based on model performance. The trimmest fit individuals are eliminated, and the hyperparameter sets evolve with each iteration or generation. This is loosely analogous to *Darwin's survival of the fittest*.

The whole process is shown in a diagram in Fig. 2.15. Evolutionary algorithms perform excellently at optimizing models but cannot always find the best possible solution. They also require high computational resources.

2.7.9 Early Stopping

The early stopping approach can be used on both continuous and discrete operations. This approach is mainly used to prevent the model from overfitting. The problem of overfitting a model is discussed in the next section. This method is especially relevant when the search space is massive, and the computational costs are significantly expensive.

2.7.10 Python Coding Example for Hyperparameter Tuning Techniques

The *Wisconsin Breast Cancer* dataset [4] contains measurements for breast cancer cases. The aim is to predict whether the breast tumors are malignant or benign. So, this is a binary classification problem.

A random forest classifier algorithm has been used for this classification purpose. Do not worry about the ML algorithm or the different types of hyperparameters related here, as these will be discussed in detail in the later chapters. Just focus on the performance of different hyperparameter tuning techniques shown here.

Programming Example 2.7
To start with, we need to import the necessary libraries and packages. Then, we will download and prepare the dataset. Listing 2.7 imports necessary library functions

for hyperparameter tuning. Additionally, it imports the breast cancer dataset from the scikit-learn library.

```python
# ----------------------Importing Libraries----------------------
from sklearn import datasets
import pandas as pd
from scipy.stats import randint as sp_rand
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.experimental import enable_halving_search_cv
from sklearn.model_selection import GridSearchCV,
    HalvingGridSearchCV
from sklearn.model_selection import RandomizedSearchCV,
    HalvingRandomSearchCV
from datetime import datetime


# ------------------------Loading Dataset------------------------
# Downloading the Breast Cancer Dataset from Scikit-learn Library
dataset = datasets.load_breast_cancer()
data = pd.DataFrame(dataset.data, columns = dataset.feature_names
    )
data['target'] = dataset.target
data.head()

# Creating dataset variables
X = dataset.data
y = dataset.target

# Splitting Dataset
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, random_state=42)
```

Listing 2.7 Importing libraries and preprocessing

After importing the library and packages, loading the dataset, creating dataset variables, and splitting the dataset into test data and train data, we will implement the different techniques of hyperparameter tuning one by one.

2.7.10.1 Manual Search

At first, we will go through the hyperparameter space manually. Then, we will build the ML models based on the hyperparameter space. Next, the models are evaluated on the accuracy metric to compare the performance. We have used the Random Forest Classifier algorithm to build the models, which will be discussed in Chap. 3.

Programming Example 2.8
Listing 2.8 depicts the Python code for the manual search technique.

```python
# ---------------------Hyperparameter Space----------------------
params_1 = {'n_estimators': 10, 'criterion': 'entropy',
            'max_features': 15, 'min_samples_split': 6,
            'min_samples_leaf': 8, 'bootstrap': True}
params_2 = {'n_estimators': 50, 'criterion': 'entropy',
            'max_features': 30, 'min_samples_split': 8,
            'min_samples_leaf': 11, 'bootstrap': True}
params_3 = {'n_estimators': 80, 'criterion': 'gini',
            'max_features': 30, 'min_samples_split': 10,
            'min_samples_leaf': 6, 'bootstrap': False}


# ---------------------Create and Fit Model----------------------
model_1 = RandomForestClassifier(**params_1)
model_2 = RandomForestClassifier(**params_2)
model_3 = RandomForestClassifier(**params_3)

# Training Models
model_1.fit(X_train, y_train)
model_2.fit(X_train, y_train)
model_3.fit(X_train, y_train)

# Results
print(f'Model 1 accuracy: {model_1.score(X_test, y_test)}')
print(f'Model 2 accuracy: {model_2.score(X_test, y_test)}')
print(f'Model 3 accuracy: {model_3.score(X_test, y_test)}')
```

Listing 2.8 Hyperparameter manual search

Output of Listing 2.8:

```
Model 1 accuracy: 0.956140350877193
Model 2 accuracy: 0.956140350877193
Model 3 accuracy: 0.9473684210526315
```

2.7.10.2 Grid Search

Next, we will implement the grid search method using pre-defined functions. The hyperparameter is created first, and then the models are built upon the hyperparameter space. After training the models, the accuracy is evaluated for each model.

Programming Example 2.9

Listing 2.9 depicts the Python code for the grid search technique.

```python
# ---------------------Hyperparameter Space----------------------
h_space = {'n_estimators': [30, 60, 80, 100],
           'criterion':['gini', 'entropy'],
           'max_features': [10, 20, 25, 30],
           'min_samples_split':[5, 11],
           'min_samples_leaf': [5, 11],
           'bootstrap': [True, False]}
```

```python


# ---------------------Create and Fit Model----------------------
random_forest_clf = RandomForestClassifier()

# Creating and training models
# The datetime function has been used to calculate operation time
start = datetime.now()
models = GridSearchCV(random_forest_clf, param_grid = h_space, cv
    =5)
models.fit(X_train, y_train)
end = datetime.now()

# Getting 5-fold cross-validated score
scores = models.cv_results_['mean_test_score']
# Getting best hyperparameters
best_hparams = models.best_params_

print(f'Duration: {end-start}')
print(f'Best model training Score: {max(scores)}')
print(f'Best hyperparameters: {best_hparams}')
```

Listing 2.9 Hyperparameters creation for grid search

Output of Listing 2.9:

```
Duration: 0:05:16.174259
Best model training Score: 0.9604395604395604
Best hyperparameters: {'bootstrap': True,
'criterion': 'entropy', 'max_features': 10,
'min_samples_leaf': 5,
'min_samples_split': 5, 'n_estimators': 80}
```

Programming Example 2.10
Next, the model is built using the best set of hyperparameter combinations, as shown in Listing 2.10.

```python
# ------------------------Training Model-------------------------
# Training model with best hyperparameters from Grid Search
best_model = RandomForestClassifier(bootstrap = True,
                                    criterion = 'entropy',
                                    max_features = 10,
                                    min_samples_leaf = 5,
                                    min_samples_split =  5,
                                    n_estimators = 80)
best_model.fit(X_train, y_train)
print('Best model accuracy: {best_model.score(X_test, y_test)}')
```

Listing 2.10 Grid search from hyperparameters

Output of Listing 2.10:

```
Best model accuracy: 0.9649122807017544
```

2.7.10.3 Halving Grid Search

We will now use the halving grid search method. This is also implemented using pre-defined functions.

Programming Example 2.11
Listing 2.11 depicts the Python code for the halving grid search technique.

```python
# ---------------------Create and Fit Model----------------------
# The hyperparameter space is the same as Grid Search
start = datetime.now()
# The datetime function has been used to calculate operation time
models = HalvingGridSearchCV(random_forest_clf,
                             param_grid = h_space, cv=5)
models.fit(X_train, y_train)
end = datetime.now()


# Getting 5-fold cross-validated score
scores = models.cv_results_['mean_test_score']
# Getting best hyperparameters
best_hparams = models.best_params_

print(f'Duration: {end-start}')
print(f'Best model training Score: {max(scores)}')
print(f'Best hyperparameters: {best_hparams}')
```

Listing 2.11 Model for halving grid search technique

Output of Listing 2.11:

```
Duration: 0:02:38.249521
Best model training Score: 0.9555555555555555
Best hyperparameters: {'bootstrap': True,
'criterion': 'gini',
'max_features': 10, 'min_samples_leaf': 5,
'min_samples_split': 5, 'n_estimators': 30}
```

Programming Example 2.12
Next, the model is built using the best set of hyperparameter combinations, as shown in Listing 2.12.

```python
# ------------------------Training Model------------------------
# Training model with best hyperparameters from Halving Grid
   Search
best_model = RandomForestClassifier(bootstrap = True,
                                    criterion = 'gini',
                                    max_features = 10,
                                    min_samples_leaf = 5,
                                    min_samples_split = 5,
                                    n_estimators = 30)
best_model.fit(X_train, y_train)
print(f'Best model accuracy: {best_model.score(X_test, y_test)}')
```

Listing 2.12 Halving grid search technique

Output of Listing 2.12:

```
Best model accuracy: 0.956140350877193
```

2.7.10.4 Random Search

Now, we will implement the random search method. Again, the pre-defined functions will be used for implementing the random search along the hyperparameter space.

Programming Example 2.13

Listing 2.13 depicts the Python code for the random search technique.

```python
# ---------------------Hyperparameter Space----------------------
h_space = {'bootstrap': [True, False],
           'criterion': ['gini', 'entropy'],
           'max_features': sp_rand(2, 30),
           'min_samples_split': sp_rand(2, 11),
           'min_samples_leaf': sp_rand(2, 11),
           'n_estimators': sp_rand(30, 100)}


# ---------------------Create and Fit Model----------------------
start = datetime.now()
models = RandomizedSearchCV(random_forest_clf,
    param_distributions = h_space, cv=5,
                            random_state = 42)
models.fit(X_train, y_train)
end = datetime.now()

# Getting 5-fold cross-validated score
scores = models.cv_results_['mean_test_score']
# Getting best hyperparameters
best_hparams = models.best_params_

print(f'Duration: {end-start}')
print(f'Best model training Score: {max(scores)}')
print(f'Best hyperparameters: {best_hparams}')
```

Listing 2.13 Hyperparameters creation for random search method

Output of Listing 2.13:

```
Duration: 0:00:12.967197
Best model training Score: 0.956043956043956
Best hyperparameters: {'bootstrap': True,
'criterion': 'entropy', 'max_features': 16,
'min_samples_leaf': 9, 'min_samples_split': 6,
'n_estimators': 53}
```

Programming Example 2.14
Next, the model is built using the best set of hyperparameter combinations, as shown in Listing 2.14.

```python
# ------------------------Training Model-------------------------
# Training model with best hyperparameters from Random Search
best_model = RandomForestClassifier(bootstrap = True,
                                    criterion = 'entropy',
                                    max_features = 16,
                                    min_samples_leaf = 9,
                                    min_samples_split = 6,
                                    n_estimators = 53)
best_model.fit(X_train, y_train)
print(f'Best model accuracy: {best_model.score(X_test, y_test)}')
```

Listing 2.14 Random search method

Output of Listing 2.14:

```
Best model accuracy: 0.9649122807017544
```

2.7.10.5 Halving Random Search

Finally, we will implement the halving random search method. Again, the predefined functions will be used for implementing the halving random search along the hyperparameter space.

Programming Example 2.15
Listing 2.15 depicts the Python code for the halving random search technique.

```python
# ---------------------Create and Fit Model----------------------
# The hyperparameter space is the same as Random Search
start = datetime.now()
models = HalvingRandomSearchCV(random_forest_clf,
                               param_distributions = h_space,
                               cv=5,
                               random_state = 42)
models.fit(X_train, y_train)
end = datetime.now()

```

```
# Getting 5-fold cross-validated score
scores = models.cv_results_['mean_test_score']
# Getting best hyperparameters
best_hparams = models.best_params_

print(f'Duration: {end-start}')
print(f'Best model training Score: {max(scores)}')
print(f'Best hyperparameters: {best_hparams}')
```

Listing 2.15 Model for halving random search method

Output of Listing 2.15:

```
Duration: 0:00:14.086538
Best model training Score: 0.9666666666666666
Best hyperparameters: {'bootstrap': True,
'criterion': 'gini',
'max_features': 9, 'min_samples_leaf': 3,
'min_samples_split': 2, 'n_estimators': 77}
```

Programming Example 2.16
Next, the model is built using the best set of hyperparameter combinations, as shown in Listing 2.16.

```
# ------------------------Training Model------------------------
# Training model with best hyperparameters from Halving Random
    Search
best_model = RandomForestClassifier(bootstrap = True,
                                    criterion = 'gini',
                                    max_features = 9,
                                    min_samples_leaf = 3,
                                    min_samples_split = 2,
                                    n_estimators = 77)
best_model.fit(X_train, y_train)
print(f'Best model accuracy: {best_model.score(X_test, y_test)}')
```

Listing 2.16 Halving random search method

Output of Listing 2.16:

```
Best model accuracy: 0.956140350877193
```

Both random search and halving random search take almost the same amount of time and give almost identical scores, but they require significantly less time than brute-force (grid search and halving grid search) methods.

2.8 Bias and Variance

Bias and variance are two reducible errors in machine learning. For accurate prediction by the ML models, a low value of bias and variance is desired. However, both cannot be achieved at the same time. This section will teach us about bias, variance, and the bias–variance trade-off:

1. **Bias:** Bias is the difference in the predicted and actual values of the target variable. When training a model on a specific dataset, the model makes some assumptions about the dataset. Based on this assumption, the model is trained. Two types of scenarios need to be dealt with while working with biases—low bias and high bias:
   (a) **Low bias:** Low bias indicates that the model makes fewer assumptions about the dataset, which helps the model learn about new features, thus increasing the performance of the model. However, low bias can make it time-consuming for the model to train. One of the significant characteristics of non-linear algorithms is low bias, such as decision trees, k-nearest neighbors, support vector machines, etc.
   (b) **High Bias:** High bias indicates that the model has made more assumptions about the dataset. These assumptions speed up the training process for the model. However, because of a higher number of assumptions, the model cannot adapt to new features on the dataset. So if the assumptions made prior to training are incorrect, the model's performance decreases. Usually, linear algorithms have high bias, i.e., linear regression, linear discriminant analysis, and logistic regression.
2. **Variance:** Variance is a measure of how the output of a model will vary if a different portion of the training data is used one at a time. In ideal circumstances, it is expected that the variation of the model's output based on different portions of the training data should not be significant. However, if the variation is significant, the algorithm has failed to establish a concrete relation between the features and the target variable. There are also two scenarios to discuss for variance—low and high:
   (a) **Low variance:** A low variance indicates moderate fluctuation of the model's output based on a different portion of the training data.
   (b) **High variance:** A high variance indicates significant fluctuation of the model's output based on different portions of the training data, resulting in the model's failure to relate between input and output. The model has to learn repeatedly from the dataset to make a relation. Thus, it learns even the minuscule features, i.e., noise from the dataset. Eventually, the model does not perform well with unobserved data.

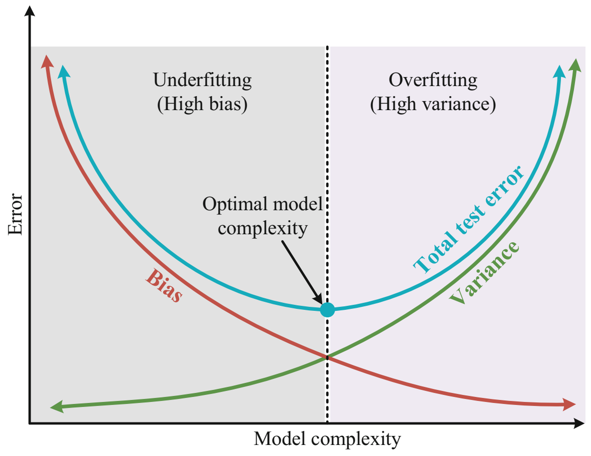


Fig. 2.16 The trade-off between bias and variance

2.8.1 Bias–Variance Trade-off

There exists a trade-off between bias and variance, which can be observed from Fig. 2.16. If the bias of the model is increased, the variance will decrease. On the contrary, if the model's variance is increased, the bias will decrease. The target of building a model is always to perform well with unobserved data. A low bias and high variance can make a model overfit the training dataset. On the other hand, a model with high bias and low variance may underfit the dataset. Therefore, a balance between bias and variance is required to be determined while training the models.

2.9 Overfitting and Underfitting

While training a model on the dataset, the goal is to find a good fit. A balance must be found so the model does not overfit or underfit. The techniques for finding a good fit vary depending on the models used and the dataset being worked with:

1. **Overfitting:** When a model overlearns a dataset, to the extent of learning all the noise and errors in a dataset, it is said that the model has overfitted. An *overfit model* fails to perform well for unobserved data. Therefore, the model loses its credibility for not predicting correctly. The model has been overfitted when train accuracy is significantly higher than the test accuracy. Overfitting occurs when the model has low bias and high variance.

   Several preventive measures can be taken to prevent the model from overfitting. One way can be to stop the training early to prevent it from learning trivial features from the dataset. Cross-validation is also an effective way to prevent overfitting. If the model is trained with a larger dataset, the probability of overfitting lessens. Regularization and ensembling techniques also help stop the overfitting of the model.

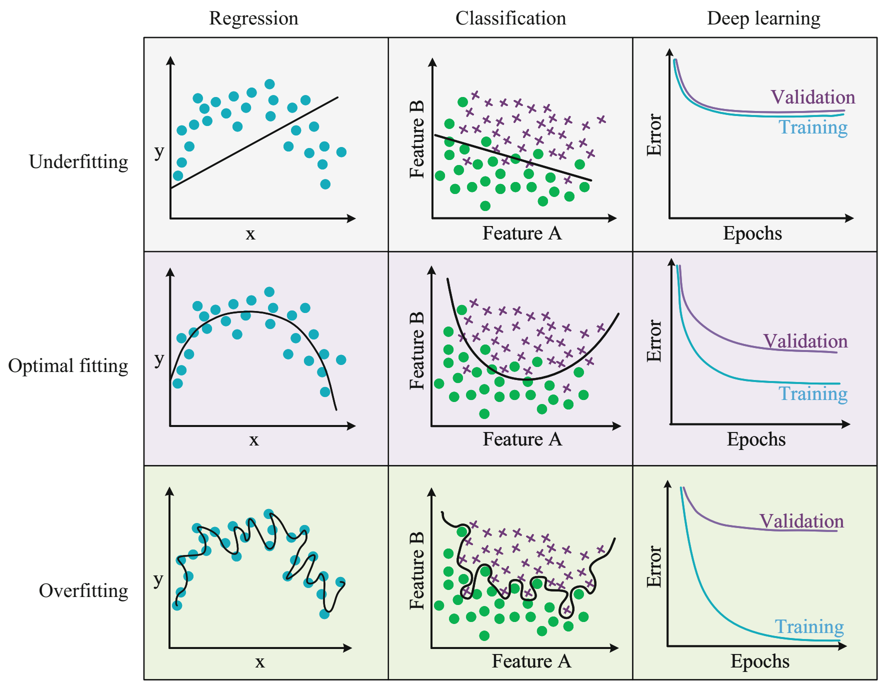


Fig. 2.17 The effect of overfitting and underfitting on training data

2. **Underfitting:** When a model fails to learn anything from the dataset, it is called an *underfit model*. It is straightforward to identify an underfit model. The train accuracies are very poor. Therefore, the model has to be trained for a more extended amount of time to prevent underfitting. Another way to prevent underfitting is to increase the number of features. A model is underfitted when the model has high bias and low variance.

The effect of overfitting and underfitting on training data is illustrated in Fig. 2.17.

2.10 Model Selection

The technique to pick a model among many built models that would be the most relevant and efficient to solve the model.

There are two approaches to model selection:

- **Probabilistic Methods:** Picks a model based on the performance and complexity of training data.

- **Resampling Methods:** Picks a model based on the performance and complexity of test data.

2.10.1 Probabilistic Methods

Probabilistic measures work well if linear models are used, e.g., regression or logistic regression. Three of the most used approaches of probabilistic model selection are discussed below.

2.10.1.1 Akaike Information Criterion (AIC)

The AIC method is one of the probabilistic methods widely used for model selection [5]. It is named after Hirotugu Akaike, a Japanese statistician who gave the formula for this method. *The lowest AIC-scored model is selected among the given models and used to perform respective tasks.* The AIC method focuses more on model performance with respect to the training dataset than model complexity. However, it does not penalize the complexity of the model. Therefore, complex models can get selected if their performances are good. The formula for calculating AIC is given by Eq. 2.27:

$$AIC = -\frac{2}{N} * LL + 2 * \frac{k}{N}, \tag{2.27}$$

where AIC denotes the Akaike Information Criterion score, N denotes the existing number of examples that are present in the training dataset, LL denotes the log likelihood of the model based on the training dataset, and k denotes the present number of parameters in the model.

2.10.1.2 Bayesian Information Criterion (BIC)

The BIC is another probabilistic model selection method very similar to the AIC method [6]. This method, also known as the *Schwarz Information Criterion*, was developed by Gideon E. Schwarz, who introduced the Bayesian adaption for the method, thus naming it the Bayesian Information Criterion. *The model with the lowest BIC score is selected as the best model among the given models.* With the increase in model complexity, the penalty for the model is also increased by the BIC method, thus lowering the chance of selecting the complex models. The formula for calculating the BIC is given by Eq. 2.28:

$$BIC = -2 * LL + \log(N) * k, \tag{2.28}$$

where BIC denotes the Bayesian Information Criterion, log() denotes the natural logarithm, N denotes the example size present in the training dataset, LL denotes the log likelihood of the model based on the training dataset, and k denotes the present number of parameters in the model.

2.10.1.3 Minimum Description Length (MDL)

The MDL is another widely used probabilistic method for model selection. The model with the least MDL score is selected as the best model on a given training set. The formula for calculating the MDL is given by Eq. 2.29:

$$MDL = L(h) + L(D|h), \tag{2.29}$$

where MDL denotes the Minimum Description Length, h is the model, D denotes the predictions outputted by the model, $L(h)$ denotes the number of bits that are needed to represent the model, and $L(D|h)$ denotes the number of bits that are needed to represent the model's outputted predictions on the training dataset. The target is to minimize both $L(h)$ and $L(D|h)$.

2.10.2 Resampling Methods

The resampling method focuses more on evaluating a model on out-of-sample data. Three of the most used approaches for resampling model selection are discussed below.

2.10.2.1 Random Train/Test Splits

This technique divides the dataset into two smaller datasets: train set and test set. The train set is used to train and build the model. The test set is the new unobserved data for evaluating the model. The predictions made on the test set by the model are compared to the expected results to make an evaluation.

The random train/test split technique is efficient when the available data are large enough. However, this technique fails to evaluate the model appropriately with a small dataset. Again, if the dataset does not contain all possible cases of the specific problem to be worked with, this technique will fail to evaluate the model performance properly.

2.10.2.2 Cross-Validation

This method is also known as *K-fold cross-validation*. This is a statistical method to evaluate the performance of a model. This method is a much more stable way to evaluate data than based on training and testing datasets.

Conventionally, we use 75% ($\frac{3}{4}$) of the data to train the data and 25% ($\frac{1}{4}$) of the data to test the model. The idea is to see if the model can predict the test data correctly. However, when the model does not have much data to train or faces any unseen behavior, it cannot predict correctly.

This is where cross-validation comes in handy. Rather than dividing the whole dataset into 4 parts, the dataset is divided into k parts (k-folds). For each of these folds, k $\in$ 1,. . . , K, all but the k-th fold is trained and tested upon the k-th fold. Upon completing the test on all the folds, all the errors are averaged, and the total error percentage is calculated.

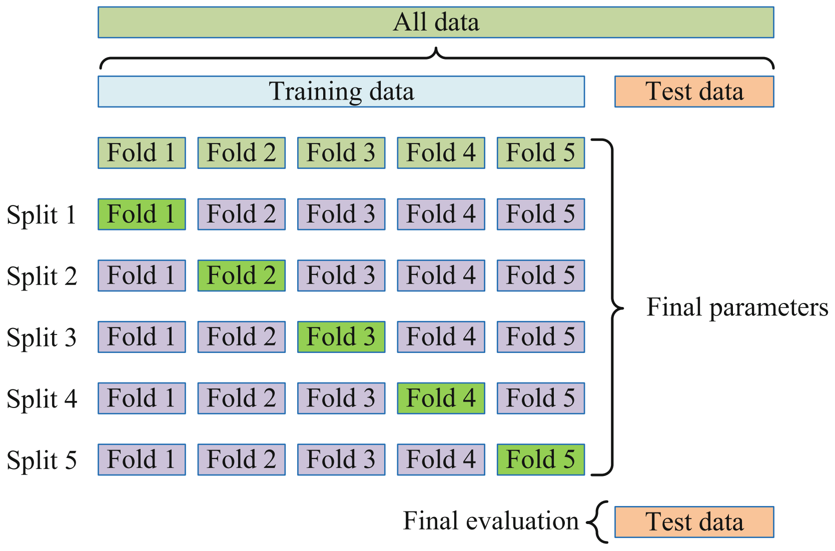


Fig. 2.18 Fivefold cross-validation visualization

Figure 2.18 shows the visualization of the fivefold cross-validation. First, the entire training dataset is split into five folds. Then, a final test set is kept aside for final evaluation. During each iteration, a different fold is used as the validation set (marked in green) for performing test validation, and the rest folds are used to train the model. For example, fold one is used as the validation set, and folds 2–5 are used to train the model in the first iteration. Then, for the next iteration, fold 2 is used as the validation set, and the rest of the folds are used as train sets. Finally, after five iterations, the cross-validation is done, and the model is tested on a final test set that is kept aside.

2.10.2.3 Bootstrap

Bootstrap is a statistical resampling method used to evaluate the model's performance. Bootstrap sampling repeatedly takes small samples from a dataset with replacement and tries to estimate the population parameter. To understand the technique, one should be familiarized with out-of-bag samples. After drawing one sample, the model is fit on that sample and evaluated on the data that are not included. These non-included data objects are out-of-bag samples.

The bootstrap sampling algorithm is given as follows:

1. Determine the number of bootstrap samples.
2. Determine the size of the sample to work with.
3. For every bootstrap sample:
   (a) Draw samples, each time replacing the previous data objects.
   (b) Make necessary statistical calculations on the sample.
4. Make necessary calculations on sample statistics.

2.11 Conclusion

In this chapter, we learned different strategies for evaluating machine learning models and selecting appropriate models. At first, we studied the different error criteria, such as the MSE, RMSE, MAE, MAPE, Huber loss, cross-entropy loss, and Hinge loss. Next, we studied different types of distance metrics for measuring the distance between two points, such as the Euclidean distance, the cosine similarity and cosine distance, the Manhattan distance, the Chebyshev distance, the Minkowski distance, the Hamming distance, and Jaccard similarity and Jaccard distance. After that, we read about the confusion matrix and some associated terms such as accuracy, precision, recall, and the F1 score, followed by an overview of model parameters, hyperparameters, and hyperparameter spaces and a detailed section on the different methods of hyperparameter tuning and model optimization. Then, we studied the concepts of bias, variance, overfitting, and underfitting. Finally, we studied the different probabilistic and resampling methods of model selection. In summary, this chapter talked us through the pre- and post-modeling stages of machine learning modeling so that we know which models to select and how to evaluate their performance. In the next chapter, we will study different types of machine learning algorithms to help us in the modeling stage.

2.12 Key Messages from This Chapter

- Building any model is not sufficient. It should be efficient and relevant to the problem and context.
- The different error criteria help us to evaluate the models. The minimum value of the error is desired.
- Hyperparameters of the models are required to be tuned so that efficient models can be built.
- A low value of bias and variance is desired. However, both cannot be achieved simultaneously, so we have to go for a bias–variance trade-off.
- A good machine learning model should avoid overfitting or underfitting the data.
- After building models, necessary methods are required to apply to select the best model. After selection, the model is applied to real-life applications.

2.13 Exercise

1. How should a model be selected for machine learning? Briefly discuss the steps.
2. What is meant by error criteria in machine learning? Classify different error criteria and mention their usage.
3. What is determined by distance metrics in machine learning? Mention different distance metrics and their usage.

4. What are parameters and hyperparameters in machine learning? Mention and briefly explain some hyperparameter tuning techniques.
5. Define bias and variance. How do bias and variance relate to overfitting and underfitting? How do we prevent overfitting and underfitting of a model?
6. Consider two arrays of equal size: $A = [1, 2, 3, 4, 5]$ and $B = [0, 0.9, 1.4, 1.7, 7]$. Find the MSE, RMSE, MAE, and MAPE.
7. The output of a machine learning model is $\hat{y} = [80, 86, 89, 95, 98]$. The corresponding ground truth value is $y = [81, 84.5, 150, 95.6, 99]$. Calculate the MSE and the Huber Loss. Which error metric is less affected by an outlier? Assume $\delta = 5$.
8. A binary classifier can also be regarded as a multiclass classifier. For a certain binary classifier, the predicted value, $\hat{y} = 0, 98$ for $y = 1$. If this is taken as a multiclass classifier, then $\hat{y} = [0.2, 0.98]$ and $y = [0, 1]$. Calculate the binary cross-entropy loss and the multiclass cross-entropy loss. Do we get similar results in both cases?
9. The test accuracy of an ML model is 98%, and the training accuracy is 90%. Does the model overfit or underfit? Give some reasons and list some mitigation techniques (e.g., increase/decrease train data, increase/decrease model size, etc.).
10. For $h = 1$ and $h = 2$, we get the Manhattan distance and Chebyshev distance, respectively, from the Minkowski distance. Can you prove it?
11. Given two data objects: A(7, 30, 0, 9, 87) and B(4, 67, 2, 54, 5). Perform the following:
    (a) Calculate:
        i. Euclidean distance
        ii. Manhattan distance
        iii. Chebyshev distance
        iv. Cosine distance
        v. Minkowski distance ($h = 3$)
    (b) Write Python programs to calculate the above.
12. Two DNA samples of different species are as follows:
    (a) Sample A = AATGCATTCG
    (b) Sample B = AATCGATTGC
    Which distance metric will you choose to determine the similarity between these two samples and why? Write a Python program to determine the similarity between these two DNA samples.
13. Two sentences are given—"The men are playing soccer on the beach" and "The boys are playing soccer near the beach." Write Python program(s) to determine cosine similarity and Jaccard similarity. Do these distance metrics agree? [Hint: Follow the code snippet below to get the required numerical data for cosine similarity.]

```python
'''
Get two different sets of words for the 2 different
    sentences.
The two sets are denoted here as S1_set and S2_set. Follow
    the
method described in Example~2.5 for getting S1_set and
    S2_set.
'''

# Word count list for 1st sentence. Declared as empty list.
S1_list = []
# Word count list for 2nd sentence. Declared as empty list.
S2_list = []

# Getting the union operation output between the word sets of
     two sentences.
# This is exactly the same union operation as the Jaccard
    similarity example.
U = S1_set.union(S2_set)

for word in U:
    if word in S1_set: S1_list.append(1) # Getting word list
    in binary format for 1st sentence
    else: S1_list.append(0)
    if word in S2_set: S2_list.append(1) # Getting word list
    in binary format for 2nd sentence
    else: S2_list.append(0)

# Now calculate the cosine similarity between S1_list and
    S2_list

```

14. A binary classifier model for classifying two objects A and B is developed. The necessary information associated with testing the model is as follows:
    (a) Total number of samples = 250
    (b) Total number of samples of A = 130
    (c) Total number of samples of B = 120
    (d) Number of samples correctly classified for A = 128
    (e) Number of samples correctly classified for A = 128
    (f) Number of samples correctly classified for B = 110

    Determine confusion matrix, accuracy, precision, recall, and F1 score by taking A as positive and B as negative class. Repeat the operation by taking A as a negative class and B as a positive class. Which metrics give the same value, and which give different values for these two conditions? Why?

References

1. Huber, P. J. (1992). Robust estimation of a location parameter. In *Breakthroughs in statistics* (pp. 492–518). Springer.
2. Shannon, C. E. (1951). Prediction and entropy of printed English. *Bell System Technical Journal, 30*(1), 50–64.
3. Gentile, C., & Warmuth, M. K. K. (1998). Linear hinge loss and average margin. In *Advances in Neural Information Processing Systems* (Vol. 11).
4. Mangasarian, O., Street, N., Wolberg, W., & Street, W. (1995). Breast Cancer Wisconsin (Diagnostic). *UCI Machine Learning Repository*. https://doi.org/10.24432/C5DW2B
5. Stoica, P., & Selen, Y. (2004). Model-order selection: A review of information criterion rules. *IEEE Signal Processing Magazine, 21*(4), 36–47.
6. Schwarz, G. (1978). Estimating the dimension of a model. *The Annals of Statistics, 6*, 461–464.

Machine Learning Algorithms 3

3.1 Introduction

The world of machine learning is not a small one. There are lots of techniques and algorithms that can be utilized in various applications. Machine learning is broadly divided into several categories. However, the two main categories are *supervised learning* and *unsupervised learning*. Supervised learning involves labeled data and can be either a classification or a regression problem. On the other hand, unsupervised learning involves unlabeled data, and the algorithm detects patterns within the dataset. This type of learning is mainly used for clustering problems, although several more applications of this category exist. A fusion of supervised and unsupervised learning approaches is made in *semi-supervised learning*, which involves a small amount of labeled data (similar to supervised learning) with a large amount of unlabeled data (similar to unsupervised learning).

Each type of machine learning constitutes several algorithms. This chapter unfolds the basics of each algorithm with practical examples and relevant code with explanation. The reader is advised to walk through the codes and execute them to ensure better learning. Since machine learning requires a huge amount of data for training the models, we will talk about datasets and their preprocessing strategies first.

3.2 Datasets

We get a dataset when we collect a set of corresponding data from a particular statistical population and arrange them in some manner. The robustness of a model significantly depends on the dataset preparation. Some standard datasets are readily available that can be used as base datasets and can be altered or modified based on the necessity for conducting machine learning research. The values of the dataset are usually numerical, although some datasets may contain Boolean or string data.

E. Hossain, *Machine Learning Crash Course for Engineers*,
https://doi.org/10.1007/978-3-031-46990-9_3

A dataset is best arranged in a tabular manner, where each row corresponds to a single data entry and each column represents a specific variable associated with all the available data in that dataset.

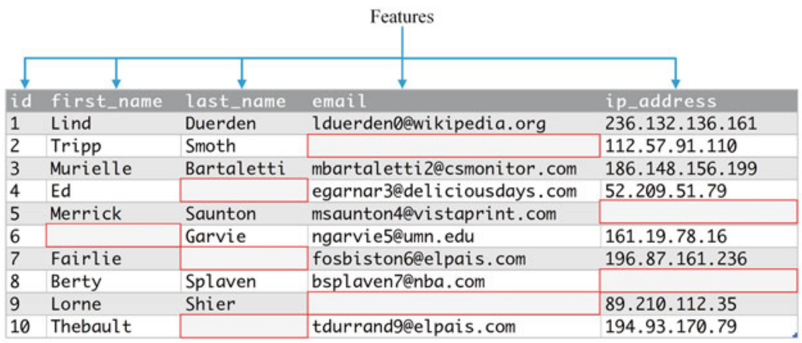
A dataset can be categorized based on its size. When the dataset becomes very large (larger than a petabyte) and contains complex data features, the data processing power requirements grow exponentially to the extent that traditional computing systems fail to comprehend. This massive dataset is then referred to as *big data*. Dataset size ranging from terabyte to megabyte may be referred to as a large dataset, whereas a dataset in the kilobyte range is a small dataset. This book mainly deals with large datasets in this sense. In this section, we will explore data wrangling, feature scaling, different data types, and data splitting.

3.2.1 Data Wrangling

Data wrangling, or *data munging*, is the process of converting a raw dataset into a digestible format for the ML algorithm. Data wrangling refines the data into a suitable format that can be readily accessed and analyzed by the ML algorithm. The process involves preprocessing, compensating for missing data, feature engineering, and converting the data to an appropriate type. Thus, data wrangling makes the data usable for various analytic purposes.

3.2.1.1 Preprocessing

Data preprocessing is a step wherein data is transformed so that machines can easily understand it. For implementing ML techniques on data, features are a pivotal factor. Therefore, features have been introduced earlier in Chap. 1 as part of the ML workflow. A feature, also known as *variables* or *fields*, is a characteristic of each data point of a specific dataset. For instance, Fig. 3.1 portrays a dataset with five features: the id number, first name, last name, email address, and IP address, and the dataset contains some missing data marked in red. A *label* is a feature that an ML algorithm



id	first_name	last_name	email	ip_address
1	Lind	Duerden	lduerden0@wikipedia.org	236.132.136.161
2	Tripp	Smoth		112.57.91.110
3	Murielle	Bartaletti	mbartaletti2@csmonitor.com	186.148.156.199
4	Ed		egarnar3@deliciousdays.com	52.209.51.79
5	Merrick	Saunton	msaunton4@vistaprint.com	
6		Garvie	ngarvie5@umn.edu	161.19.78.16
7	Fairlie		fosbiston6@elpais.com	196.87.161.236
8	Berty	Splaven	bsplaven7@nba.com	
9	Lorne	Shier		89.210.112.35
10	Thebault		tdurrand9@elpais.com	194.93.170.79

Fig. 3.1 A dataset having five features and some missing data

extracts from the dataset for each data point. The features have different data types, and some values are missing. The main objective of data preprocessing is to allow machines to understand the features of the whole data frame. This involves feature scaling, removing missing values, data augmentation, etc., which are discussed in the following sections. In a tabular representation of datasets, features can typically be seen as column headers.

3.2.1.2 Missing Data

Finding every piece of missing data while working on a vast dataset is implausible. This is generally because of errors during data collection, validation, human error, data merging between different datasets, etc. Missing data can cause an error if the ML algorithm is implemented on such a faulty dataset. Hence, these missing or corrupted data must be either changed or corrected or removed during the preprocessing stage to ensure the smooth performance of the ML algorithm.

The following list demonstrates some methods to eliminate the missing data from a large dataset. The dataset is read into a Pandas data frame, $ipAddress$. Here, $dropna()$ is a method already implemented in the Pandas package. So, instead of writing vigorous codes from scratch, we can simply use this Pandas method directly. Pandas has their official documentation on how to deal with missing values [1].

1. **Eliminating missing values**
   The command is `ipAddress.dropna()`
2. **Eliminating selected rows or columns**
   The command is `ipAddress.dropna([1, 2, 3])`
3. **Eliminating missing values in selected rows or columns**
   The command is `ipAddress.dropna("email", axis=1)`
4. **Eliminating missing values based on condition**
   The command is `ipAddress.dropna[df["email"] == "bsplaven7@nba.com"]`
5. **Eliminating null values**
   The command is `ipAddress.dropna[df["email"].notnull()]`

3.2.1.3 Imputation

Many datasets often have missing values. These values are a hurdle to properly implementing the ML algorithm. ML algorithms require datasets to be complete. Imputation can be a better way of filling in missing data.

Programming Example 3.1

Listing 3.1 shows a general method of using imputation followed by its output and explanation in Table 3.1. The code shows the imputation process when dealing with missing numbers. The missing values in this scenario are replaced with the average of the adjacent numbers.

```
import numpy as np
from sklearn.impute import SimpleImputer

imp = SimpleImputer(missing_values=np.nan, strategy='mean')
imp.fit([[1, 5],
         [2, 10],
 7            [7, 35],
 8            [15, 75],
 9            [0, 0]])
10
11 X = [[1, 5],
12      [2, 10],
13      [np.nan, 2],
14      [7, 35],
15      [6, np.nan],
16      [15, 75],
17      [0, 0]]
18
19 print(imp.transform(X))
```

Listing 3.1 Data imputation

Output of Listing 3.1:

```
[[ 1.   5.]
 [ 2.  10.]
 [ 5.   2.]
 [ 7.  35.]
 [ 6.  25.]
 [15.  75.]
 [ 0.   0.]]
```

The code requires `numpy` and `sklearn` packages as prerequisites. The code to install these packages is as follows:

```
pip install numpy
pip install sklearn
```

We used the `mean` of the existing values to get the remaining values, which can be seen in Line 4. We can also use `median`, `most_frequent`, and `constant` instead of `mean`.

- `mean`: The missing values are replaced by the mean of each column; it works with only numeric data.
- `median`: The missing values are replaced by the median of each column; it works with only numeric data.
- `most_frequent`: Then the missing values are replaced by the most frequent value or the mode of each column; it works with strings or numeric data. For multiple modes, only the smallest mode is returned.

Table 3.1 Explanation of Listing 3.1

Line number	Description
1–2	Importing libraries
4	Creating an instance of imputation
5–9	Fitting the imputation to existing data
11–17	Declaring a list with blank values
19	Predicting null values using mean strategy

- `constant` : The missing values are replaced by `fill\_value`; it works with strings or numeric data.

3.2.2 Feature Scaling

When working with different features or variables, it is almost certain that we have data on different features in different ranges. For instance, a dataset has two features: temperature and humidity. The values for temperature range from 25 to 35 °C, and the values for humidity range from 70 to 90%. The range is different for the two features, so they cannot be compared. This range difference is also unsuitable for ML algorithms to work with. Therefore, a sort of transformation is done on the attribute values so that all features come within a comparable and workable range. The transformation done on the data for this purpose is *feature scaling*. There are different methods for feature scaling, three of which are standardization, normalization, and data augmentation. These are described in the following sections.

3.2.2.1 Standardization

Standardization is a very popular method for feature scaling. After standardizing the dataset, we get a mean of zero and a unit standard deviation. This method is used primarily in cases where the data distribution follows the normal distribution. Standardizing the data does not bring the data to a specific pre-defined range. Therefore, this feature scaling method is not affected by outliers. Whenever a feature is standardized, each data is first subtracted from the mean and then divided by the standard deviation. The formula to perform standardization of the data is given in Eq. 3.1:

$$z = \frac{x - \mu}{\sigma}, \tag{3.1}$$

where z represents the standardized value, x is the attribute value, μ is the respective attribute mean, and σ is the respective attribute standard deviation.

3.2.2.2 Normalization

Normalization is another popularly used method for feature scaling. Normalization is a technique to convert different numeric values into a common range without distorting the differences between the values. It scales all the feature values within the range [0,1] or [−1,1]. Since there is a specific range value normalizing the data, the outliers present in the dataset affect this method. Normalization is beneficial when the dataset's data distribution is unknown. The formula to perform normalization of the data is given in Eq. 3.2:

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}}, \tag{3.2}$$

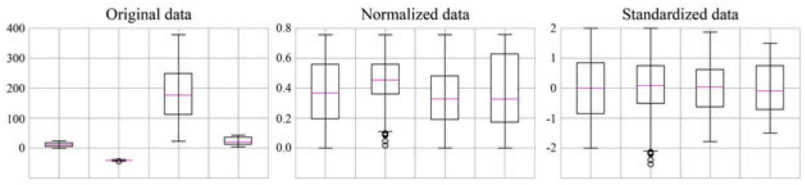


Fig. 3.2 Difference between standardization and normalization

where x' is the normalized value, x is the attribute value, $x_{\min}$ is the minimum value of the respective attribute, and $x_{\max}$ is the maximum value of the respective attribute.

The difference between standardization and normalization has been demonstrated through a box and whisker plot in Fig. 3.2. The normalized data brings all the original values to the 0 to 1 range, while the standardized dataset has a mean of zero.

Example 3.1 Suppose a weather dataset has two features—temperature and humidity. Apply feature scaling on the dataset using standardization and normalization methods.

Temperature	25	21	30	35	34	33	32	33	24	23
Humidity	78	75	88	72	83	79	76	88	85	77

Solution to Example 3.1
Standardization
For the temperature feature, we calculate the mean μ first.

$$\mu = \frac{25+21+30+35+34+33+32+33+24+23}{10} = \frac{290}{10} = 29. \tag{3.3}$$

Next, we calculate the standard deviation σ for temperature:

$$\sigma = \sqrt{\frac{1}{N}\sum(x_i-\mu)^2} = \sqrt{\frac{244}{10}} = 4.9396. \tag{3.4}$$

Applying the standardization method to the first temperature value, i.e., $x = 25$, we get

$$z = \frac{x-\mu}{\sigma} = \frac{25-29}{4.9396} = -0.8097. \tag{3.5}$$

Similarly, we get the standardized values given in Table 3.2 for temperature and humidity features:

Table 3.2 The standardized values for temperature and humidity

Temperature	Humidity
−0.8097	−0.3962
−1.6195	−0.9622
0.2024	1.4905
1.2146	−1.5283
1.0122	0.5471
0.8097	−0.2075
0.6073	−0.7735
0.8097	1.4905
−1.0122	0.9245
−1.2146	−0.5849

Table 3.3 The normalized values for temperature and humidity

Temperature	Humidity
0.2857	0.3750
0.0000	0.1875
0.6428	1.0000
1.0000	0.0000
0.9285	0.6875
0.8571	0.4375
0.7857	0.2500
0.8571	1.0000
0.2142	0.8125
0.1428	0.3125

Normalization

For the temperature, $x_{\min} = 21$ and $x_{\max} = 35$.

Applying the normalization method to the first temperature value, i.e., $x = 25$, we get

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}} = \frac{25 - 21}{35 - 21} = 0.2857. \tag{3.6}$$

Similarly, we get the normalized values given in Table 3.3 for temperature and humidity features:

3.2.2.3 Data Augmentation

While working with a dataset, we may often come across the issue of insufficient data. A larger dataset is always desired as it helps the models to make generalizations easily. A dataset that is huge in quantity and contains different possible data scenarios is helpful for the model to perform efficiently. The more data fed to the model during the training phase, the better the model performs. *Data augmentation* is a technique that synthetically increases the training data size by creating modified

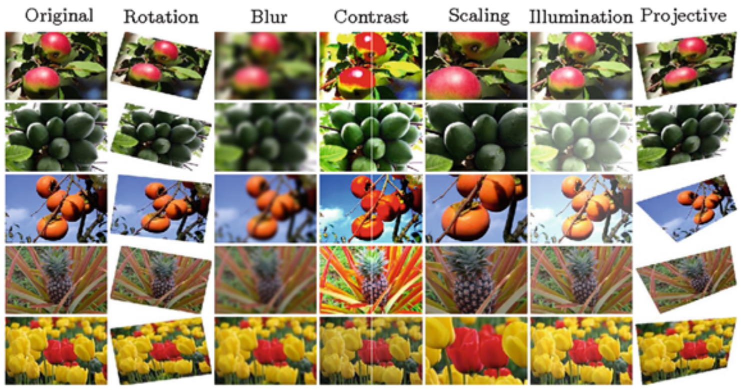


Fig. 3.3 Examples of different data augmentation techniques [2]

data from the existing ones. Various techniques can be applied to the existing dataset to increase the amount and variety of data.

Figure 3.3 presents an example of different techniques to achieve data augmentation, such as rotation, blurring, contrast, scaling, illumination, projection, etc. Scaling the images creates the zoom-in effects, resulting in the diversity of data. Another technique is arbitrarily rotating the original image, varying the degree values to diversify the dataset. Augmentation can also be achieved by tweaking color components such as hue, intensity, saturation, etc. An object can be viewed from different angles. Changes in perspective points or projections are done to achieve the effect of viewing the object from different angles.

3.2.3 Data Types

Machine learning algorithms deal with different types of data. Therefore, different data types require different approaches to ML algorithms. The data types can be categorized as (i) sequential and non-sequential and (ii) stationary and non-stationary. These are discussed in the following sections.

3.2.3.1 Sequential vs. Non-sequential Data Type

Sequential data types are those that have a definite sequence, such as lists, strings, tuples, byte sequences, byte arrays, and range objects. The elements in such data types are accessible through their indexes, which indicate their position in the sequence and start from 0.

On the contrary, *non-sequential data types* have no sequence, such as dictionaries and sets. There is no maintained order among the elements in non-sequential data types. Figure 3.4 depicts a visual representation of sequential data and non-sequential data. For sequential data, an array is shown. The array is serially indexed.

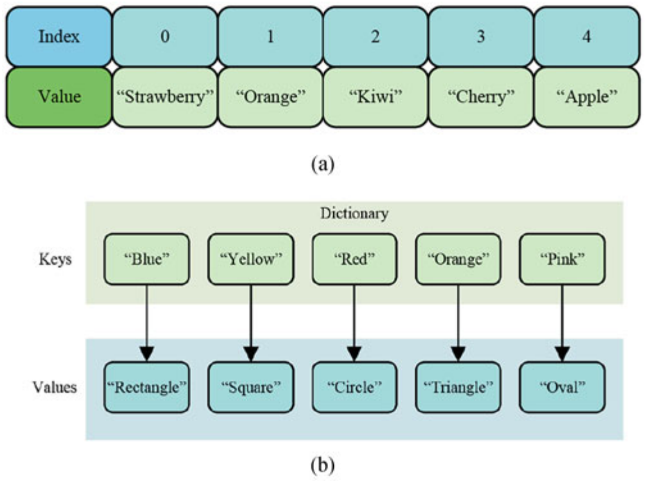


Fig. 3.4 Examples of sequential vs. non-sequential data types. (**a**) Sequential data type: an array. (**b**) Non-sequential data type: a dictionary

On the contrary, a dictionary is shown with no serial indexing for non-sequential data.

3.2.3.2 Stationary vs. Non-stationary Data Type

A stationary dataset is known to have constant statistical properties, such as mean, variance, etc., over time. Data in such datasets are easy to forecast since their properties do not change with time.

On the contrary, the dataset whose statistical properties change over time is known as non-stationary data. These types of data cannot be forecasted or modeled due to their variable nature. Non-stationary data have trends, cycles, or seasonality in them. Figure 3.5 shows an example of stationary and non-stationary data.

3.2.4 Data Splitting

Data splitting has been previously mentioned in Chap. 1 as a step in the ML workflow. Splitting data serves the purpose of appropriately training an ML model. The proper splitting prevents a model from overfitting, ensures precise assessment, and improves the performance of a model. The most common way of data splitting is to split the dataset into two subsets—one train dataset and another test dataset. But a convention of splitting into three subsets is also familiar. In this case, the dataset is split into train, validation, and test datasets. These two ways of data splitting are illustrated in Fig. 3.6. The percentages mentioned in the figure can vary according to use cases.

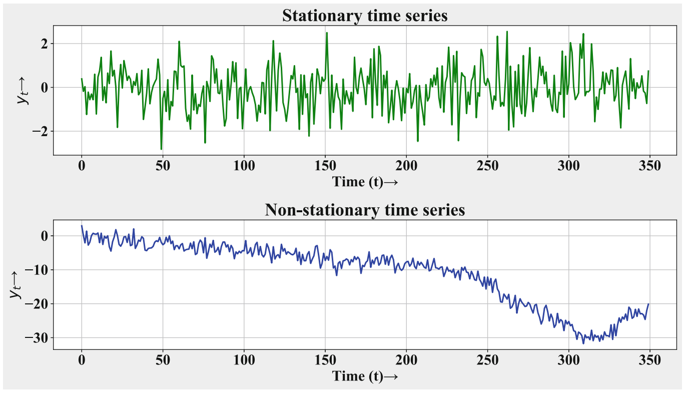


Fig. 3.5 Stationary vs. non-stationary data

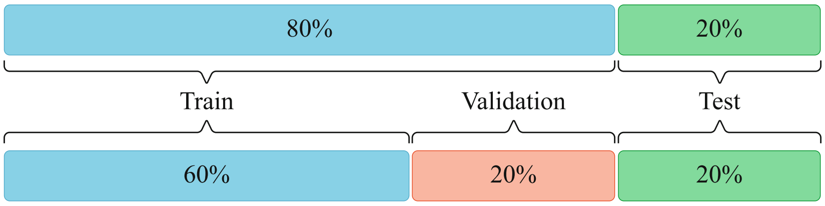


Fig. 3.6 The two ways of data splitting: training and testing data (top) and training, validation, and testing data (bottom)

1. **The training dataset:** The training dataset is the subset used to train the model. It is used by the model to recognize any pattern or relationships in the data. In most cases, it is the largest chunk (nearly 60–80%) of the data.
2. **The validation dataset:** The validation dataset, also known as the *development dataset*, is the subset used to understand the performance of the various models and parameters and select the best model. In other words, it is used to get an unbiased evaluation of how the model performance changes when using different models and parameters instead of those that were used to train the model.
3. **The testing dataset:** The testing dataset is used to evaluate the model performance on unseen data. How well it predicts the test data determines the model's accuracy.

Apart from this splitting technique, there are some that are often used for special purposes, such as:

- **k-Fold Cross-Validation:** The dataset, in this case, is divided into k subsets (also called k-folds). Then these subsets are used to train and validate the model. In contrast to the normal training and validating process, it uses a different subset each time for validating and the rest of the subsets for training. More details on cross-validation can be found in Sect. 2.10.2.2 of Chap. 2.

- **Stratified Sampling:** This method is used so that all the subsets contain each class or label in an equal distribution after splitting. It ensures the data is correctly distributed within the train, test, and validation datasets.

The most common method for splitting is scikit-learn's `train_test_split()` function, whose uses are demonstrated below:

1. For train–test split:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split (
features, labels, test_size=0.2, random_state=42, shuffle=
False)
```

2. For train-validation-test split:

```
X_train_test, X_test, y_train_test, y_test =
train_test_split (signals, labels, test_size=0.2)

X_train, X_val, y_train, y_val = train_test_split (
X_train_test, y_train_test, test_size=0.3)
```

Sometimes, a label is not passed through the function for splitting.

```
X_train, X_test =
train_test_split (signal_df.values, test_size=0.2)
```

Different parameters can be passed into the function. The most common parameters are $train\_size$, $test\_size$, $random\_state$, and $shuffle$, which consecutively define train dataset size, test dataset size, the randomness of the data, and whether the data should be randomly shuffled before splitting. The parameter $shuffle$ is True by default. If you set the test dataset size, the remaining data will be used as the training dataset. Consequently, no need to mention the train dataset size in general. The $random\_state$ parameter controls the shuffling but ensures the splitting is reproducible across multiple function calls; e.g., `random_state=42` means a specific random state is called for reproducing the same shuffle. In the case of classification tasks, the parameter $stratify$ needs to be used to have approximately the same distribution of target classes as the original dataset.

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, stratify=y)
```

Data splitting can cause the model performance drop in some specific cases. One of them is in the case of overfitting. For instance, in a power system, most faults (about 90% time) are single line-to-ground (SLG) faults. If the data taken from a live system is directly used for training, it will overfit the SLG label and try to predict everything as SLG. Maintaining the integrity of the dataset is crucial to preserve the original data distribution and prevent any data leakage. This is necessary to ensure that the testing set does not interfere with the training process and that it does not degrade model performance.

In most cases, the train–test split or train–validation–test split is adequate in general. However, data splitting is done according to the dataset and model characteristics for special purposes.

3.3 Categorization of Machine Learning Algorithms

Machine learning is an infinitely vast field of science beyond hard-and-fast categorization. Generally, in the most standard references, ML algorithms are divided into supervised learning and unsupervised learning. In some other texts, semi-supervised learning is presented as the third type. Therefore, it can be rightly stated that there is no strict categorization of the algorithms, and whichever notion you decide to follow is correct. However, in this book, the following main types of ML algorithms are discussed in depth:

1. Supervised learning
2. Deep learning
3. Time series forecasting
4. Unsupervised learning
5. Semi-supervised learning
6. Reinforcement learning

Let us now dive into the world of machine learning.

3.4 Supervised Learning

Supervised learning is the most commonly used technique in ML. The technique involves training the model with a labeled dataset, i.e., with definite features or labels. The first task in dealing with any problem is to understand the nature of the problem, i.e., whether the problem demands a supervised or an unsupervised algorithm. If the problem contains an initial dataset, then the chances are high that it might require a supervised algorithm to solve (albeit it may require a semi-supervised approach too). Supervised learning can be primarily classified into two broad categories—*regression* and *classification*. If it is clear that a supervised approach can solve the problem, then the next consideration should be whether it is a regression problem or a classification problem. This decision is necessary to help decide which ML algorithm to use and which algorithm will provide the solution with the minimum errors. In the following sections, we will explore regression and classification in detail.

3.4.1 Regression

The regression analysis is defined as a set of statistical methods used to determine the relationships between one dependent variable and one or more independent variables. This technique is beneficial for predicting the value of a data point in a continuous dataset.

Suppose we are to find out the price of solar photovoltaic modules in the year 1995, and we already have the information for some random years, such as 1980, 1983, 1989, 1990, 1993, 1999, 2003, and 2005. Notice that the years are not equally spread apart or follow any definite pattern. In that case, having no other verified source, ML models can be used to find out the price in 1995. An ML algorithm will help to determine the unknown value based on the available information on the prices in the other years. Such a problem is known as a regression problem, which may also be called *data interpolation* in Layman's terms.

Some commonly deployed regression models are elaborated on in the following sections, along with implementations in Python.

3.4.1.1 Simple Linear Regression

Simple linear regression is a linear regression model involving a single dependent variable and a single independent variable. It is expressed using the general equation for a straight line in a regular two-dimensional rectangular coordinate system as

$$y = \beta_0 + \beta_1 x, \tag{3.7}$$

where x and y are the independent and dependent variables, respectively, β_0 is the intercept from the y-axis, and β_1 is the slope.

Simple linear regression is also termed ordinary least squares (OLS). It tries to minimize (hence the name "least") the sum of squares for error. The error is the difference between the actual data point and its predicted value, as shown in Eq. 3.9. Since it can be positive or negative, it is squared to yield a positive quantity consistently. Although fitting a simple linear regression might not always be accurate, it can try to fit in various environments, as portrayed in Fig. 3.7.

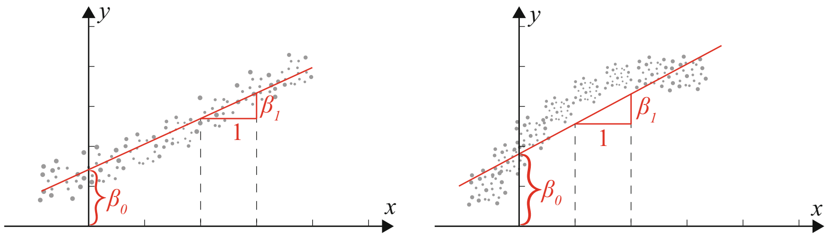


Fig. 3.7 Fitting straight line (left) and curve line (right) using a simple linear regression model

Equation 3.7 gives us the expected or predicted values based on regression. The actual or observed values will be different from this value. For instance, if we consider case i, the fitted value into Eq. 3.7 is given by

$$\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i. \tag{3.8}$$

The difference between observed values and predicted values leads us to residuals. The sum of squared residuals is called the residual sum of squares (RSS), which is given by

$$RSS = L = \sum_{i=1}^{n} \left(y_i - \hat{y}_i \right)^2. \tag{3.9}$$

A penalty term can be added to the OLS equation to obtain a higher model accuracy for complex data, which adds a bias with some values. The bias is symbolized with λ, the regularization parameter. These are known as L1 regularization (LASSO regression) and L2 regularization (ridge regression). A model that minimizes both bias and variance is deemed the best.

3.4.1.2 LASSO Regression

The term LASSO is an acronym for *Least Absolute Shrinkage and Selection Operator*. It was coined by Robert Tibshirani in 1996 and was intended for linear regression models. LASSO regression uses the L1 regularization technique to reduce overfitting and improve prediction accuracy. In addition, shrinkage is used in this technique to shrink the data points closer to their mean. LASSO regression is useful for creating simple models with high levels of multicollinearity. The L1 penalty function is given by the absolute value: $\lambda \times |slope|$. The equation for LASSO regression with standardized features is expressed as

$$L_{lasso}(\hat{\beta}) = \sum_{i=1}^{n} \left(y_i - \sum_{j} x_i^j \hat{\beta}_j \right)^2 + \underbrace{\lambda \sum_{j=1}^{m} \left| \hat{\beta}_j \right|}_{\text{penalty function}} . \tag{3.10}$$

Here, λ is the amount of shrinkage.

- If $\lambda = 0$, it is the same as linear regression, and all features are considered. The residual sum of squares is only considered to build a predictive model.
- If $\lambda = \infty$, no feature is considered. As λ approaches infinity, it eliminates more and more features.

When λ increases, the bias increases, and the variance decreases. On the contrary, when λ decreases, the bias decreases, and the variance increases.

A key disadvantage of LASSO regression is the loss of information during the elimination of all correlated variables except one. This loss of information hampers the model's accuracy.

3.4.1.3 LASSO LARS Regression

The Least-Angle Regression (LARS) algorithm was developed by Bradley Efron, Trevor Hastie, Iain Johnstone, and Robert Tibshirani in 2004 [3]. The algorithm can fit linear regression models to high-dimensional data. LARS is comparable with forward stepwise regression. At every step, the feature with the highest correlation with the target is determined. In the case of multiple features with the same correlation, LARS propagates at a direction equiangular to all the features. LassoLars is a LASSO model implemented using the LARS algorithm.

3.4.1.4 Ridge Regression

Ridge regression is a special case of Tikhonov regularization, which can help reduce overfitting. This technique uses L2 regularization. The penalty function for L2 regularization is $bias \times slope^2$. The equation for ridge regression with cost function on the right side of the equation is

$$L_{ridge}(\hat{\beta}) = \sum_{i=1}^{n}\left(y_i - \sum_j x_i^j \hat{\beta}_j\right)^2 + \lambda \sum_{j=1}^{m} \hat{\beta}_j^2. \tag{3.11}$$

3.4.1.5 Elastic Net Regression

Elastic net regression is a convex combination of ridge regression and LASSO regression. Besides setting and choosing a λ value, elastic net also permits tuning the mixing parameter named α, where $\alpha = 0$ corresponds to ridge and $\alpha = 1$ to LASSO.

The purpose of the elastic net regression method is to minimize the loss function, a combination of the RSS and the regularization terms. The loss function is expressed with the help of the equation:

$$L_{elastic}(\hat{\beta}) = \frac{\sum_{i-1}^{n}(y_i - x_i^j \hat{\beta})^2}{2n} + \lambda\left(\frac{1-\alpha}{2}\sum_{j=1}^{m} \hat{\beta_j}^2 + \alpha \sum_{j=1}^{m} |\hat{\beta}_j|\right). \tag{3.12}$$

3.4.1.6 Support Vector Regression

A support vector machine (SVM) is a set of supervised learning techniques that can be used for regression, classification, and detection of outliers. The use of SVM in regression problems is known as support vector regression (SVR). SVR has three types of implementations: SVR, NuSVR, and LinearSVR [4]. LinearSVR is faster than SVR but only considers the linear kernel, and NuSVR implements a slightly different formulation than SVR and LinearSVR.

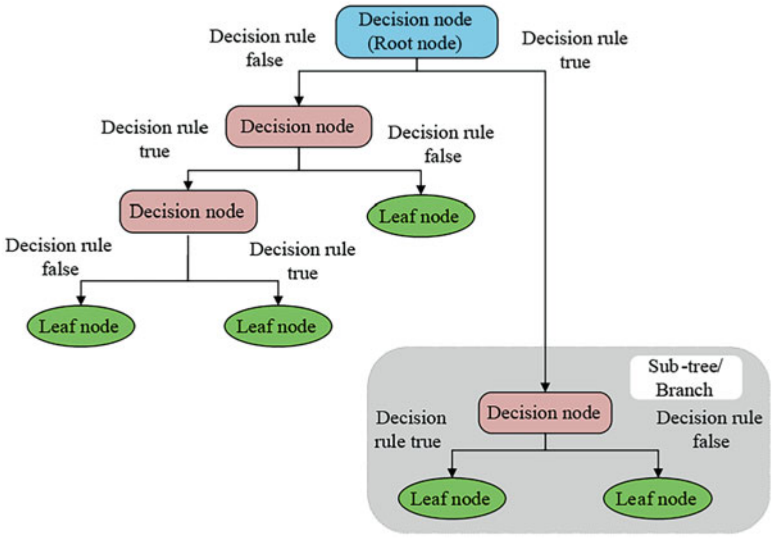


Fig. 3.8 Basic theme of the decision tree algorithm

3.4.1.7 Decision Tree Regression

The decision tree algorithm comprises two types of nodes: the decision node and the leaf node, as shown in Fig. 3.8. A decision node is a point wherein a condition is checked and a decision is made, whereas a leaf node is a point where a pure dataset (i.e., with the same features) is obtained, and no further splits can be done. A decision tree algorithm repeatedly splits the dataset into two parts until a pure leaf node is obtained in each part.

In simple terms, a dataset is categorized based on a series of yes or no decisions about a certain condition for the features at each node that eventually determine the fate of a particular data point. For regression problems, the decisive condition is determined based on the reduction of the variance of the dataset. The formula to calculate the variance is

$$Variance = \frac{1}{n} \sum (y_i - \bar{y})^2. \tag{3.13}$$

3.4.1.8 Random Forest Regression

The decision tree algorithm has the disadvantage of being highly sensitive to the training dataset, and so the decision tree can yield completely different results by a change of any single data. This is where the random forest algorithm comes into play. From the training dataset, several data points are chosen randomly to create a new dataset, and for each data point, a couple of features are selected for building the decision tree for that new dataset. This process is termed as *bootstrapping*. The random forest algorithm involves using several decision trees (hence the name as multiple trees make up a forest, and the trees and the corresponding features are

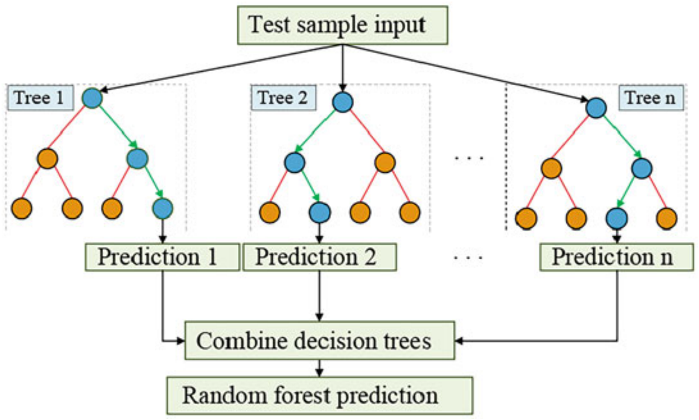


Fig. 3.9 Basic theme of the random forest algorithm for regression problems

Table 3.4 Advantages and disadvantages of the random forest algorithm

Advantages	Disadvantages
The problem of overfitting is mitigated	The algorithm is complex
The accuracy of prediction is high	Higher computational resources are necessary
Data scaling is not necessary	The prediction process is time-consuming

chosen randomly from the training dataset). In the case of a regression problem, the average of the results from each random forest decision tree is taken as the final prediction for the new data point. This process is demonstrated in Fig. 3.9. The advantages and disadvantages of the random forest algorithm are given in Table 3.4.

3.4.1.9 Bayesian Ridge Regression

Bayesian ridge regression is a type of ridge regression that uses the Bayesian theorem, i.e., probabilistic terms for the analysis. The penalty function of the L2 regularization of ridge regression is estimated using the Bayesian theorem. For two different events A and B, the Bayesian theorem can be expressed as follows:

$$P(A|B) = \frac{P(B|A).P(A)}{P(B)}, \tag{3.14}$$

where

$P(A)$ and $P(B)$ are the prior probabilities of events A and B occurring, respectively, without any condition.
$P(A|B)$ is the conditional probability of event A occurring given that B is true.
$P(B|A)$ is the conditional probability of event B occurring given that A is true.
$P(B) \neq 0$.

3.4.1.10 Multiple Linear Regression

Multiple linear regression (MLR) estimates the value of a variable dependent on two or more independent variables. In other words, when one parameter is linked to two or more factors, MLR is used. It is determined using the following equation:

$$y = \beta_\circ + \beta_1 X_1 + \cdots + \beta_n X_n + \epsilon, \tag{3.15}$$

where

$X_1, \ldots, X_n$ are the independent variables.
$\beta_\circ$ is the intercept from the y-axis.
$\beta_1, \ldots \beta_n$ are the regression coefficients.
y is the unknown dependent variable or the target variable.
ϵ is the error.

3.4.1.11 Polynomial Regression

In linear regression, a straight line links all the scattered data points in a two-dimensional plane. On the other hand, polynomial regression uses an nth degree polynomial to fit all the scattered data points.

Programming Example 3.2

In this elaborate example, we show a simple linear regression model used to determine the relationship between advertisements on TV, radio, and newspaper with the amount of sales. The program is broken down into segments with a clear description of the steps. Where applicable, the outputs of the steps are provided after each step.

```python
# Importing necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import pearsonr
from sklearn.metrics import r2_score
```

```python
# Reading and initializing dataset
dataset = pd.read_csv("advertising.csv")
dataset.head()
```

Output:

```
        TV     Radio    Newspaper    Sales
0    230.1      37.8         69.2     22.1
1    44.5       39.3         45.1     10.4
```

```
2   17.2        45.9        69.3       12.0
3   151.5       41.3        58.5       16.5
4   180.8       10.8        58.4       17.9
```

```python
# Detailed info about the dataset
dataset.info()
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   TV         200 non-null    float64
 1   Radio      200 non-null    float64
 2   Newspaper  200 non-null    float64
 3   Sales      200 non-null    float64
dtypes: float64(4)
memory usage: 6.4 KB
```

```python
# Checking the number of null data in the dataset
dataset.isnull().sum()
```

Output:

```
TV           0
Radio        0
Newspaper    0
Sales        0
dtype: int64
```

We calculate Pearson's correlation coefficients between advertisements on different media (TV, radio, and newspapers) and the amount of sales. This operation will decide on which data the linear regression will be performed.

```python
# Checking the correlation between different types of
    advertisement and sales
tv_corr, _ = pearsonr(dataset["TV"], dataset["Sales"])
radio_corr, _ = pearsonr(dataset["Radio"], dataset["Sales"])
news_corr, _ = pearsonr(dataset["Newspaper"], dataset["Sales"])

print("TV Correlation: {}\nRadio Correlation: {}\nNewspaper
    Correlation: {}\n".format(
    tv_corr, radio_corr, news_corr
))
```

Output:

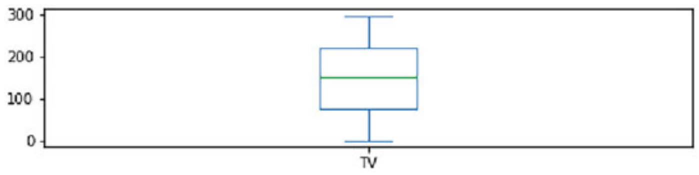
```
TV Correlation: 0.9012079133023307
Radio Correlation: 0.34963109700766914
Newspaper Correlation: 0.1579600261549263
```

Here, we can see that the correlation coefficient for TV data is the highest. So, we will consider the TV data for our linear regression model.

```python
# whether TV data contain outliers or not from box plot
dataset["TV"].plot(kind='box', subplots=True, layout=(4,1),
    figsize=(8,8)# Checking)
```

Output:

```
TV  AxesSubplot(0.125,0.71587;0.775x0.16413)
dtype: object
```



```python
# String data from dataset into variable
tv = dataset["TV"]
sales = dataset["Sales"]
```

The input data is standardized before applying it to the linear regression model.

```python
# Maximum value from TV data
tv_max = max(tv)
# Minimum value from TV data
tv_min = min(tv)

# Mean value from TV data
tv_mu = np.mean(tv)
# Standard deviation from TV data
tv_sigma = np.std(tv)

# Applying standard scaling on TV data
# This scaled data will be used as input data
tv = (tv-tv_mu) / (tv_sigma + 1e-6)
tv = np.array(tv)
tv = np.reshape(tv, (-1,1))
```

The equation for simple linear regression is $y = \beta_0 + \beta_1 x$. In our case, the equation is $y = sales$ and $x = TV$. So, we can write $sales = \beta_0 \times 1 + \beta_1 \times TV$. For this reason, we will need to add a column of ones to our input data.

```python
# Creating a vector of ones
# It has the same length as TV data
X_ones = np.ones_like(tv)

# The one vector is concatenated to the TV data
# The concatenation is done row-wise
# Thus the input data now contains two columns
X = np.concatenate((X_ones, tv), axis=1)

# The sales data are stored as output
y_true = np.array(sales)
y_true = np.reshape(y_true, (-1,1))
```

```python
# Defining MSE function
# MSE is used as loss function here
def loss_mse(y_hat, y_true):

  loss = np.mean((y_hat - y_true)**2)

  return loss
```

We calculate the gradients are calculated for backpropagation purposes, which is required for performing gradient descent. Section 3.5.2 provides more details on this.

```python
# Function for gradient calculation
# Gradients are required for implementing the Gradient Descent
    algorithm
def grad(y_hat, y_true, x):

  # The expression for gradients are calculated manually
  grad_c = 2 * np.mean(y_hat - y_true)
  grad_m = 2 * np.mean((y_hat - y_true) * x)

  return [grad_c, grad_m]
```

```python
# Function for implementing model training
def train(x, y_true, params, learning_rate):

  # Model prediction as y_hat
  y_hat = (params * x).sum(axis=1)
  y_hat = np.reshape(y_hat, (-1,1))

  # Loss calculation from the loss function (MSE)
  loss = loss_mse(y_hat, y_true)

  # The next two steps are required for model optimization
  # These two steps are the core learning process
  # Gradient computation
  grads = grad(y_hat, y_true, x)

  # Parameter update
  new_params = params - learning_rate*np.array(grads)

  return loss, new_params
```

```
# Splitting the dataset
# First 180 datapoints are selected as training data
X_train = X[:180]
y_train = y_true[:180]
# Last 20 datapoints are selected as test data
X_test = X[180:]
y_test = y_true[180:]
```

```
# Number of iteration or epoch
epoch = 50
# Learning rate (alpha)
alpha = 0.1

# Empty list for storing training losses at each epoch
losses = []
# Both learnable parameters are initialized as 1
params = np.ones((1,2), dtype=np.float64)

# Commencing the training of the linear regression model
# Loop is used to iterate through every epoch
for i in range(epoch):
  loss, new_params = train(X_train, y_train, params,
    learning_rate=alpha)
  params = new_params
  losses.append(loss)
  print("Epoch {}   Loss: {}".format(i+1, loss))
```

Output:

```
Epoch 1   Loss: 217.74026464292425
Epoch 2   Loss: 136.62414046936394
Epoch 3   Loss: 87.5254734940006
Epoch 4   Loss: 57.85860889022964
Epoch 5   Loss: 39.91014552885682
Epoch 6   Loss: 28.983105073246357
Epoch 7   Loss: 22.23810175095949
Epoch 8   Loss: 17.972700566127372
Epoch 9   Loss: 15.175399587907759
Epoch 10   Loss: 13.25110878757679
Epoch 11   Loss: 11.853278577725417
Epoch 12   Loss: 10.78202906486732
Epoch 13   Loss: 9.922891293871576
Epoch 14   Loss: 9.210369755605814
Epoch 15   Loss: 8.606553371397444
Epoch 16   Loss: 8.08876179764178
Epoch 17   Loss: 7.642552764900449
Epoch 18   Loss: 7.257863963352305
Epoch 19   Loss: 6.9269539292918205
Epoch 20   Loss: 6.643350802398563
Epoch 21   Loss: 6.401347667439824
Epoch 22   Loss: 6.195780958046366
Epoch 23   Loss: 6.021945437742041
Epoch 24   Loss: 5.87556739504495
Epoch 25   Loss: 5.7527964841889725
...
Epoch 47   Loss: 5.172144547281532
Epoch 48   Loss: 5.170878793327431
Epoch 49   Loss: 5.16985094633049
Epoch 50   Loss: 5.169016486362349
```

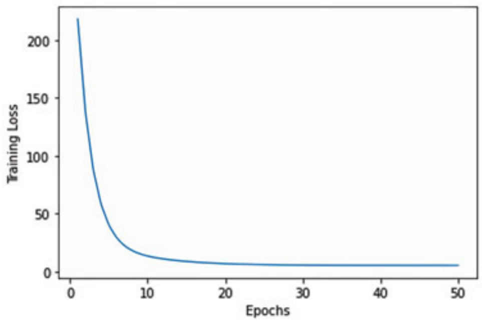
```
# Training loss curve visualization
fig = plt.figure()
epochs = np.arange(1, epoch+1)
plt.plot(epochs, losses)
plt.xlabel('Epochs')
plt.ylabel('Training Loss')
plt.show()
```

The output of the above code is the graph illustrated in Fig. 3.10.

Figure 3.10 shows that the losses gradually decrease with respect to the number of epochs and eventually become quite constant at some points. This is the expected behavior. The coefficient of determination, R^2, scores are calculated for model evaluation. The closer the value of R^2 is to 1, the better.

```
# Calculation of training R2 score
train_pred = (params*X_train).sum(axis=1)
train_pred = np.reshape(train_pred, (-1,1))
```

Fig. 3.10 Output for training loss vs. epochs



```python
train_r2 = r2_score(y_train, train_pred)

# Calculation of test R2 score
test_pred = (params*X_test).sum(axis=1)
test_pred = np.reshape(test_pred, (-1,1))
test_r2 = r2_score(y_test, test_pred)

print("Training R2 score: {}\nTest R2 score: {}\n".format(
    train_r2, test_r2))
```

Output:

```
Training R2 score: 0.8092780603962393
Test R2 score: 0.8324002955411913
```

```python
# Subplots for linear regression output
fig, (ax1, ax2) = plt.subplots(1,2, figsize=(15,7))

# Plotting output on training data
x = X_train[:, 1]
y = params[:, 0] + params[:, 1]*x

ax1.scatter(tv[:180], sales[:180], label='Training Data')
ax1.plot(x, y, c='r', label='Linear Regression Model')
ax1.set(xlabel='TV Advertisement', ylabel='Sales', title='
    Training Output')
ax1.legend()

# Plotting output on test data
x = X_test[:, 1]
y = params[:, 0] + params[:, 1]*x

ax2.scatter(tv[180:], sales[180:], label='Test Data')
ax2.plot(x, y, c='r', label='Linear Regression Model')
ax2.set(xlabel='TV Advertisement', ylabel='Sales', title='Test
    Output')
ax2.legend()
```

Output: The output of the above code is the graphs illustrated in Fig. 3.11. The output line indicates that the line has fitted both the training data and test data quite well. Thus, a successful linear regression model has been trained.

So far, we have discussed different types of regression algorithms. Now, we will study the classification algorithms in the next section.

3.4.2 Classification

Classification, a type of supervised learning technique, is one of the most highly used real-world applications of ML at present. It simply refers to classifying or labeling data from a dataset based on the features provided in the training data.

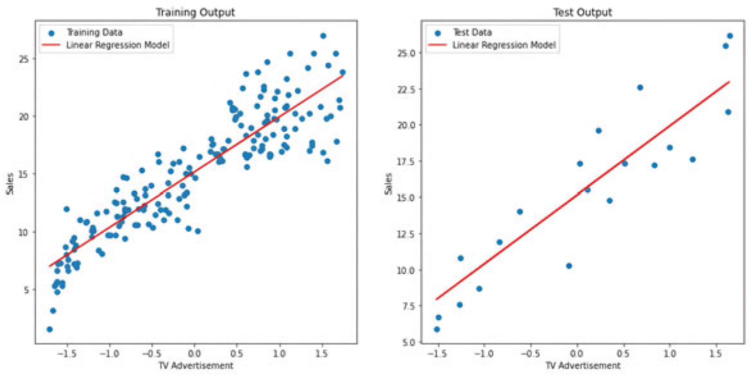


Fig. 3.11 Training and test output

Say we are to classify the students of a class as good, average, or weak based on several factors, such as their grades, attendance, interaction during classes, homework, extracurricular activities, and behavior. Each of these five factors is a feature, and the type assigned to the students (good, average, or weak) is the label of the three defined classes. This is an example of a classification problem. The classes are set such that whenever a new student enters, he/she can be easily classified into one of the three classes based on the features. An ML model prepares these classes, and when fed with the features of a new student, it can easily classify the new student into a suitable class. This task might sound easy, but when it comes to classifying hundreds of students (not to mention six times the features), the task might be daunting as a manual effort. ML can come in handy in such cases of dealing with a large number of data.

In the following sections, we shall discuss the most popular ML algorithms for classification problems and explore several worked-out examples using Python.

3.4.2.1 Logistic Regression

The name logistic *regression* might sound misleading, but it is indeed a linear model for solving classification problems. Therefore, it is also referred to as the maximum-entropy classification (MaxEnt), logit regression, or the log-linear classifier. In this model, the logistic function (Eq. 3.16) is used to model the probabilities of the possible outcomes of a single trial.

$$f(x) = \frac{L}{1 + e^{-k(x-x_\circ)}}, \tag{3.16}$$

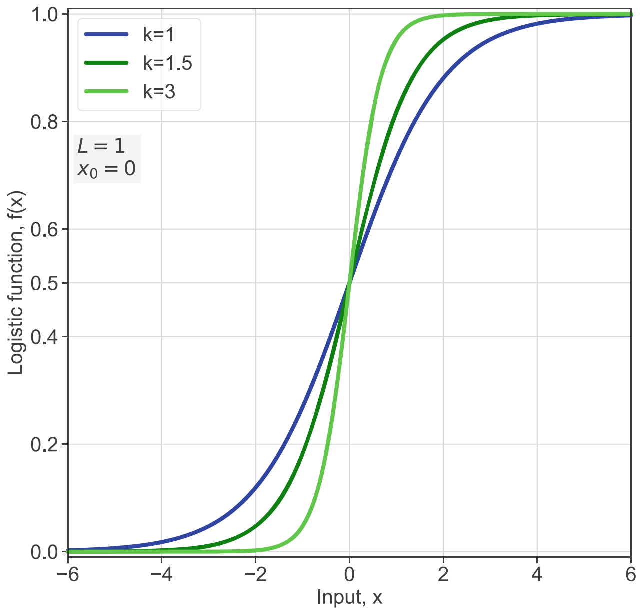


Fig. 3.12 Logistic function

where

L is the maximum value of the curve.
k is the logistic growth rate or steepness of the curve.
$x_\circ$ is the x-value at the midpoint of the curve.
A standard logistic sigmoid (S-shaped) function or curve is shown in Fig. 3.12.

3.4.2.2 k-Nearest Neighbor (KNN)

The k-nearest neighbor (KNN) is one of the most popular and straightforward algorithms in supervised learning. It can be used to solve regression and classification problems, although it is more frequently used for the latter. Two common adjectives of the KNN algorithm are that it is *non-parametric* (i.e., it makes no underlying assumptions regarding the data distribution) and a *lazy learner* (i.e., it does not immediately learn from the training dataset).

The Euclidean distance is measured from “k”-nearest neighboring points to the data point that has to be classified. For example, among “k”-nearest neighboring points, if the majority of the nearest neighboring points belong to Class A, then the data point is classified as Class A. Figure 3.13 demonstrates the KNN algorithm where $k = 3$. Here, it can be seen that two of the nearest neighboring points belong to the class “true,” and one nearest neighboring point belongs to the class “false.” Therefore, the data point is classified as a class “true” data point. The advantages and disadvantages of KNN are discussed in Table 3.5.

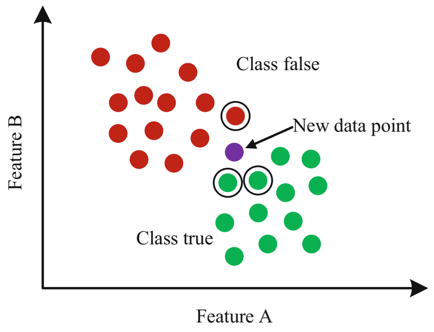


Fig. 3.13 KNN algorithm with $k = 3$

Table 3.5 Advantages and disadvantages of the KNN algorithm

Advantages	Disadvantages
It has a simple implementation	It has a high computational cost
It is good for large training data	It requires the determination of the value of K each time

Example 3.3 A dataset is given with two features "A" and "B" and respective labels as True or False. Using the KNN algorithm, classify the new data point (6, 5) where $k = 3$.

Feature A	Feature B	Label
5	2	True
5	4	True
7	4	False
8	6	False
7	3	False

Solution to Example 3.3
We will calculate the Euclidean distance of the new data point (6, 5) from every data point in the dataset.

Feature A	Feature B	Label	Euclidean distance
5	2	True	$\sqrt{(5-6)^2 + (2-5)^2} = 3.16$
5	4	True	$\sqrt{(5-6)^2 + (4-5)^2} = 1.41$
7	4	False	$\sqrt{(7-6)^2 + (4-5)^2} = 1.41$
8	6	False	$\sqrt{(8-6)^2 + (6-5)^2} = 2.24$
7	3	False	$\sqrt{(7-6)^2 + (3-5)^2} = 2.24$

Now we will rank it in ascending order of the Euclidean distances.

Feature A	Feature B	Label	Euclidean distance
5	4	True	$\sqrt{(5-6)^2+(4-5)^2}=1.41$
7	4	False	$\sqrt{(7-6)^2+(4-5)^2}=1.41$
8	6	False	$\sqrt{(8-6)^2+(6-5)^2}=2.24$
7	3	False	$\sqrt{(7-6)^2+(3-5)^2}=2.24$
5	2	True	$\sqrt{(5-6)^2+(2-5)^2}=3.16$

Since $k = 3$, we will pick the three nearest data points according to their Euclidean distance. The three nearest data points to the new data point are (5, 4), (7, 4), and (8, 6). Now according to their respective labels, we have to classify the new data point. Data point (5, 4) classifies the data point as "True," whereas both the data points (7, 4) and (8, 6) classify the new data point as "False." Since most of the k-nearest neighbor is classifying the new data point as "False," the new data point (6, 5) is "False."

3.4.2.3 Support Vector Classification

The use of SVM in classification problems is known as support vector classification (SVC). SVC has three types of implementations: SVC, NuSVC, and LinearSVC [4]. The SVM algorithm works by creating a decision boundary or hyperplane among the data to create different classes. The outermost data point in a class nearest to the hyperplane is called a *support vector*. Support vectors are used to create this decision boundary or hyperplane. A straight line would suffice for linear data as a hyperplane. However, for non-linear data, a straight hyperplane will fail to segregate the classes properly. Therefore, kernels are needed to determine the different shapes of hyperplanes according to the data arrangement as depicted in Fig. 3.14.

Kernel machines are types of functions for analyzing the patterns within datasets. The different types of kernel functions can be sigmoid, linear, non-linear, polynomial, and radial basis functions (RBFs).

3.4.2.4 Naive Bayes

The naive Bayes methods are a family of supervised learning algorithms using Bayes' theorem with the "naive" assumption of conditional independence between each feature pair given the value of the class variable [4]. According to the Bayes theorem, for a class variable y and dependent feature vectors $x_1, x_2, \ldots, x_n$, the following relationship is applicable:

$$P(y|x_1, x_2, \ldots, x_n) = \frac{P(y)P(x_1, x_2, \ldots, x_n|y)}{P(x_1, x_2, \ldots, x_n)}. \tag{3.17}$$

The following naive conditional independence assumption is made:

$$P(x_i|y, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = P(x_i|y). \tag{3.18}$$

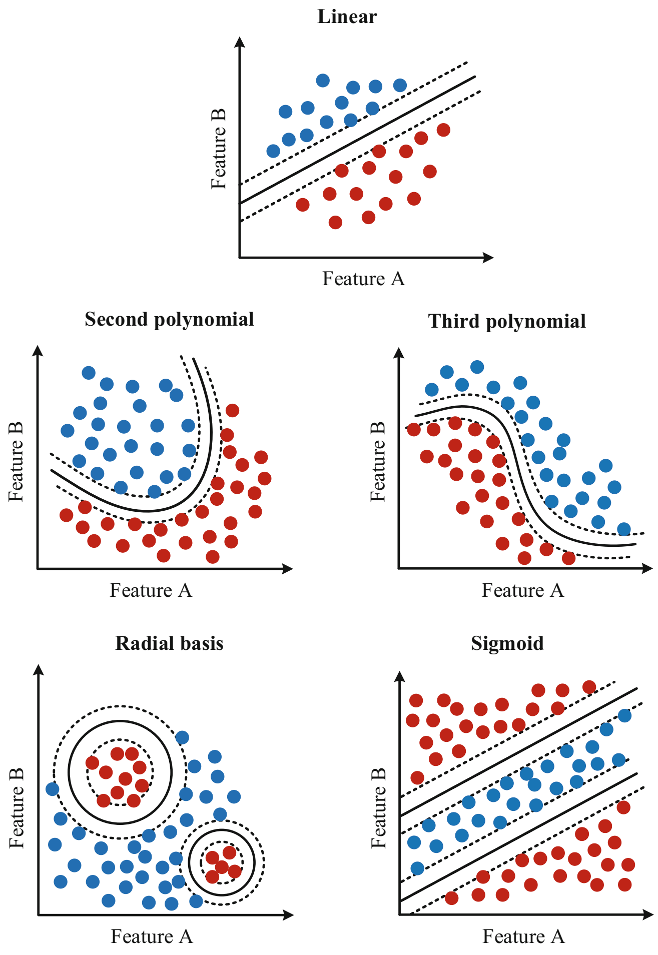


Fig. 3.14 Logistic function

This relationship can be simplified for all i as

$$P(y|x_1, x_2, \ldots, x_n) = \frac{P(y)\prod_{i=1}^{n} P(x_i|y)}{P(x_1, x_2, \ldots, x_n)}. \tag{3.19}$$

Since $P(x_1, x_2, \ldots, x_n)$ is a constant, the above equation can be represented as

$$P(y|x_1, x_2, \ldots, x_n) \propto P(y)\prod_{i=1}^{n} P(x_i|y). \tag{3.20}$$

$$\hat{y} = \arg\max_{x} P(y)\prod_{i=1}^{n} P(x_i|y). \tag{3.21}$$

Advantages of Naive Bayes Algorithms

The advantages of the naive Bayes algorithms are enlisted as follows [4]:

- These algorithms work well for real-world applications despite the oversimplified assumptions.
- A small amount of training data is sufficient for estimating the parameters.
- These algorithms are very fast in comparison to more sophisticated methods.
- Each feature distribution can be independently estimated as a 1D distribution, which avoids dimensionality.

However, a disadvantage of the Naive Bayes algorithm is that it is a lousy estimator because it is based on the assumption that the features are independent.

3.4.2.5 Gaussian Naive Bayes

This is a special type of naive Bayes algorithm. The Gaussian Naive Bayes algorithm assumes that the continuous features associated with each class are distributed in a normal or a Gaussian distribution. The dataset is first distributed into classes, and then the mean and variance of each class are determined. Then the probability density of x_i of class y is given by the following equation:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i-\mu_y)^2}{2\sigma_y^2}\right). \tag{3.22}$$

The mean of the values is μ_y and the Bessel corrected variance of the values is σ_y^2. Here, the maximum likelihood is used to estimate the parameters σ_y and μ_y.

3.4.2.6 Decision Tree Classification

As described in Sect. 3.4.1.7, a decision tree algorithm repeatedly splits the dataset until a pure leaf node is obtained in each part. For classification problems, different algorithms of decision tree construction use different attribute selection measures (ASMs). In this section, we will talk about five ASMs.

Entropy

Entropy is the measure of randomness or disorder in the dataset. The formula to calculate the entropy is

$$Entropy(A) = -P_{yes}\log_2(P_{yes}) - P_{no}\log_2(P_{no}), \tag{3.23}$$

where P_{yes} is the probability of the attribute being yes, and P_{no} is the probability of the attribute being no.

The entropy of an attribute can be measured using the following equation:

$$E(T, X) = \sum P(X)E(X), \tag{3.24}$$

where X is the attribute whose entropy we want to calculate, T is the target feature, $P(X)$ is the probability of the attribute, and $E(X)$ is the entropy of the attribute.

Information Gain

The information gain is used to measure how much information can be obtained from a certain feature based on its entropy. For instance, the ID3 algorithm uses entropy and information gain as ASM. To calculate the information gain, the following equation is used:

$$Information\ Gain(T,\ X) = Entropy(T) - Entropy(T, X), \tag{3.25}$$

where T is the feature variable and X is the attribute.

Split Information

The formula to calculate split information is given below:

$$Split\ Info_A(D) = -\sum \frac{|D_j|}{|D|} \times \log_2\left(\frac{|D_j|}{|D|}\right), \tag{3.26}$$

where A is the attribute, D_j is the frequency of attribute values, and D is the total number of attributes.

Gain Ratio

If the information gain of an attribute is divided by the split information, then we get the gain ratio of that attribute. The C4.5 algorithm uses the gain ratio and split information as the ASM. The gain ratio is calculated using the following equation:

$$Gain\ Ratio(A) = \frac{Information\ Gain(A)}{Split\ Info\ (A)}, \tag{3.27}$$

where A is the attribute.

Gini Index

The *Gini index* is simply the measure of impurity in the dataset. The Classification And Regression Tree (CART) algorithm uses the Gini index. The formula to calculate the Gini index is given below:

$$Gini = 1 - \sum (P_i)^2, \tag{3.28}$$

where P_i is the probability of the ith attribute.

Example 3.4 A dataset for playing golf is given in Table 3.6. Construct a decision tree using entropy and information gain as the attribute selection measures.

Table 3.6 Dataset for playing golf [5]

Outlook	Temperature	Humidity	Windy	Play Golf
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Overcast	Hot	High	Weak	Yes
Rainy	Mild	High	Weak	Yes
Rainy	Cool	Normal	Weak	Yes
Rainy	Cool	Normal	Strong	No
Overcast	Cool	Normal	Strong	Yes
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Rainy	Mild	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes
Overcast	Mild	High	Strong	Yes
Overcast	Hot	Normal	Weak	Yes
Rainy	Mild	High	Strong	No

Table 3.7 Datapoint counts for the target class **Play Golf**

Yes	No	Total
9	5	14

Solution to Example 3.4 [6]
Here, the attributes are:

- ▶ Outlook: Sunny, Overcast, Rainy
- ▶ Temperature: Hot, Mild, Cool
- ▶ Humidity: High, Normal
- ▶ Wind: Weak, Strong

And the target class is **Play Golf: Yes, No**.

First, we will count the number of Yes and No for this target class (Table 3.7).

Now we will calculate the entropy for the target class Play Golf:

$$\begin{aligned}\mathrm{E}(\text{Play Golf}) &= -P_{yes} \cdot \log_2 \left(P_{yes}\right) - P_{no} \cdot \log_2 \left(P_{no}\right) \\ &= -\frac{9}{14} \cdot \log_2 \left(\frac{9}{14}\right) - \frac{5}{14} \cdot \log_2 \left(\frac{5}{14}\right) = 0.94.\end{aligned}$$

Next, we will calculate the information gain of each attribute to determine the root node (Table 3.8).
Calculating the entropy for each branch, we get

$$\mathrm{E}(\text{Sunny}) = -\frac{2}{5} \cdot \log_2 \left(\frac{2}{5}\right) - \frac{3}{5} \cdot \log_2 \left(\frac{3}{5}\right) = 0.97$$

Table 3.8 The data point counts in the outlook feature

		Play Golf		Count
		Yes	No	
Outlook	Sunny	2	3	5
	Overcast	4	0	4
	Rainy	3	2	5
Total				14

Table 3.9 Information gain of the four attributes, Outlook, Humidity, Windy, and Temperature

Attribute	Information gain
Outlook	0.25
Humidity	0.15
Windy	0.05
Temperature	0.03

$$E(\text{Overcast}) = -\frac{4}{4} \cdot \log_2\left(\frac{4}{4}\right) - \frac{0}{4} \cdot \log_2\left(\frac{0}{4}\right) = 0$$

$$E(\text{Rainy}) = -\frac{3}{5} \cdot \log_2\left(\frac{3}{5}\right) - \frac{3}{5} \cdot \log_2\left(\frac{3}{5}\right) = 0.97.$$

Next, we have to calculate the entropy of **Outlook** with respect to the target class.

$$\begin{aligned} E(\text{Play Golf, Outlook}) &= P(\text{Sunny}) \cdot E(\text{Sunny}) + P(\text{Overcast}) \cdot E(\text{Overcast}) \\ &\quad + P(\text{Rainy}) \cdot E(\text{Rainy}) \\ &= \frac{5}{14} \cdot 0.97 - \frac{4}{14} \cdot 0 + \frac{5}{14} \cdot 0.97 = 0.69. \end{aligned}$$

So, we get the information gain of **Outlook**:

$$\begin{aligned} \text{Information Gain (Outlook)} &= \text{E(Play Golf)} - \text{E(Play Golf, Outlook)} \\ &= 0.94 - 0.69 = 0.25. \end{aligned}$$

Similarly, we can calculate the information gain of all the attributes as given in Table 3.9.

It can be seen that **Outlook** has the highest information gain among all the attributes, so it is assigned as the root node. The next step is to split the **Outlook** node. As calculated earlier, it is seen that the entropy of **Overcast** is 0. Therefore, further splitting for **Overcast** will not be needed. We have found one leaf node, and Fig. 3.15 visualizes the corresponding decision tree at this stage.

However, we can see that the entropy values for **Sunny** and **Rainy** are not zero. Therefore, we need further splitting of the branches.

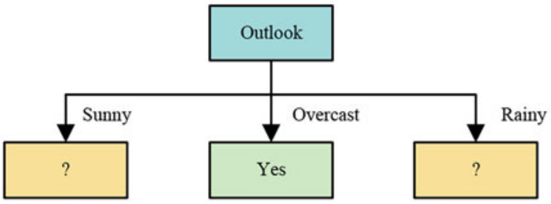


Fig. 3.15 Outlook has been selected as the root node as it has the highest information gain

Table 3.10 Dataset for playing golf when the Outlook is Sunny

		Play Golf		
		Yes	No	Count
Humidity	High	0	3	3
	Normal	2	0	2
Total				5

Table 3.11 Entropy of the Humidity

E(High)	$-\frac{0}{3}\cdot\log_2\left(\frac{0}{3}\right)-\frac{3}{3}\cdot\log_2\left(\frac{3}{3}\right)$	0
E(Normal)	$-\frac{2}{2}\cdot\log_2\left(\frac{2}{2}\right)-\frac{0}{2}\cdot\log_2\left(\frac{0}{2}\right)$	0

So, first, we attempt to split the **Sunny** branch further. Now, we have to calculate the information gain of the rest of the attributes with respect to **Sunny** (Table 3.10).

First, we calculate the entropy for each branch (Table 3.11).

Now, we calculate the entropy for **Humidity**.

$$\begin{aligned}\mathrm{E(Sunny, Humidity)} &= \mathrm{P(High)}\cdot\mathrm{E(High)}+\mathrm{P(Normal)}\cdot\mathrm{E(Normal)}\\ &= \frac{3}{5}\cdot 0+\frac{2}{5}\cdot 0\\ &= 0.\end{aligned}$$

We get the information gain for **Humidity** as

$$\begin{aligned}\text{Information Gain (Humidity)} &= \text{E(Sunny)} - \text{E(Sunny, Humidity)}\\ &= 0.97-0\\ &= 0.97.\end{aligned}$$

Similarly, we will calculate the information gain for the remaining attributes as given in Table 3.12.

Since **Humidity** gives the highest information gain, this will be the next assigned node. As the calculated entropy values for the branches **High** and **Normal** are zero, there will be no need for further splitting and hence reached leaf nodes. Figure 3.16 depicts this updated version of the decision tree.

Table 3.12 Information gain of the three attributes, Humidity, Windy, and Temperature

Attribute	Information gain
Humidity	0.97
Windy	0.02
Temperature	0.57

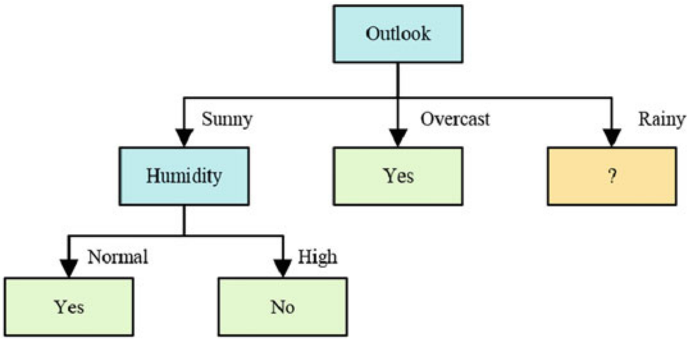


Fig. 3.16 Humidity has been selected as the next node

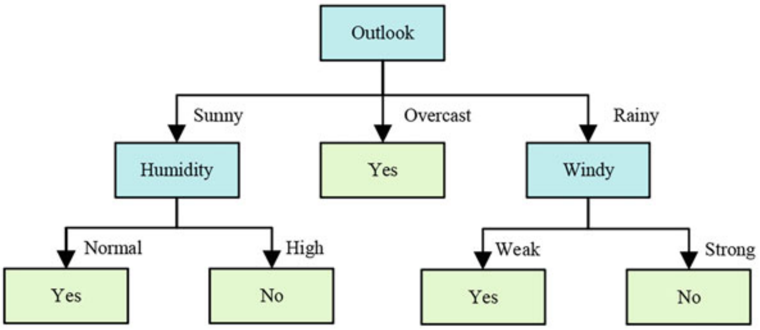


Fig. 3.17 The complete decision tree for the dataset in Table 3.6 using entropy and information gain as ASM

Similarly, we repeat the process of splitting the branch **Rain** and determining the next decision node. We will find that the next decision node will be **Windy**, and the branches will have 0 entropy. The construction of the decision tree is complete as shown in Fig. 3.17.

3.4.2.7 Random Forest Classification

The basic concept of the random forest algorithm has been explained in Sect. 3.4.1.8. Then, the initial bootstrapping process is carried out similarly to the regression problems. In the case of a classification problem, based on the results from each decision tree of the random forest, the label found from the majority of the trees is selected as the final label for the new data point. This process is known as *aggregation*.

Programming Example 3.3
Listing 3.2 provides the implementation procedure of different classifiers. The classifiers use four different datasets. Each classifier is trained on train samples and visualized for test samples. The output of the listing is visualized in Fig. 3.18 followed by the explanation in Table 3.13.

```
# --------------------Importing Libraries------------------------
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles,
    make_classification
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.datasets import make_gaussian_quantiles


# ----------Defining classifier models and datasets--------------
names = ["KNN", "SVM", "RBF SVM", "Gaussian Process",
         "Gaussian NB", "Decision Tree", "Random Forest"]

# 6 classifiers are demonstrated here
models = [
    KNeighborsClassifier(3),  # KNN
    SVC(kernel="linear", C=0.02), # SVM
    SVC(gamma=2, C=1), # Non-Linear SVM
    GaussianProcessClassifier(0.99 * RBF(1.0)), # Gaussian
    classifier
    GaussianNB(), # NB
    DecisionTreeClassifier(max_depth=6), # Decision tree
    RandomForestClassifier(max_depth=6, n_estimators=10,
    max_features=2), # Random Forest
    ]


datasets = [make_moons(noise=0.1, random_state=1),
            # creates a moon shape 2 class dataset

            make_classification(n_features=2, n_redundant=0,
                                n_informative=1,
                                n_clusters_per_class=1),
            # creates a separable classification dataset

            make_gaussian_quantiles(n_features=2, n_classes=2),
            # creates two gaussian circle dataset

            make_classification(n_features=2, n_redundant=0,
                                n_informative=2)
            # creates a dataset with two class overlap
            ]


Fig = plt.figure(figsize=(30, 10))
count = 1


# -------------------Training and Plotting-----------------------
for idxx, data in enumerate(datasets):
    X, y = data
    # pre-processing
    X = StandardScaler().fit_transform(X)
    X_train, X_test, y_train, y_test = \
        train_test_split(X, y, test_size=.5, random_state=10)

    # Minimum and maximum range for creating the plot mesh
    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                         np.arange(y_min, y_max, 0.02))


    # Using the scikit-learn plotting of decision boundary
    example
    cm = plt.cm.gray
    cm_bright = ListedColormap(['#F000A0', '#00FFAA'])
    ax = plt.subplot(len(datasets), len(models) + 1, count)
    if idxx == 0:
        ax.set_title("Different types of data")

    # plotting test samples
    ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=
    cm_bright, alpha=0.6,
               edgecolors='k')
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xticks(())
    ax.set_yticks(())
    count += 1 # Counting column


    for name, classifier in zip(names, models):
        ax = plt.subplot(len(datasets), len(models) + 1, count)
        classifier.fit(X_train, y_train) # Training
        acc = classifier.score(X_test, y_test) *100 # Accuracy

        # Decision boundary plotting
        # Point in the mesh [x_min, x_max]x[y_min, y_max]
        if hasattr(classifier, "decision_function"):
            hh = classifier.decision_function(np.c_[xx.ravel(),
                                                      yy.ravel()])
        else:
            hh = classifier.predict_proba(np.c_[xx.ravel(),
                                                  yy.ravel()])[:,1]

        # Put the result into a color plot
        hh = hh.reshape(xx.shape)
        ax.contourf(xx, yy, hh,cmap=cm, alpha=.8)


        # Ploting test samples
        ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=
    cm_bright,
                   edgecolors='k', alpha=0.8)

        ax.set_xlim(xx.min(), xx.max())
        ax.set_ylim(yy.min(), yy.max())
        ax.set_xticks(())
        ax.set_yticks(())
        if idxx == 0:
            ax.set_title(name)
        ax.text(xx.max() - .3, yy.min() + .3, ('%.2f' % acc).
    lstrip('0'),
                size=15, color= 'red', horizontalalignment='right
    ')
        count += 1 # counting within each row


plt.savefig('./results/classifier.png', bbox_inches='tight')
```

Listing 3.2 Performance of different classifiers [7]

3.5 Deep Learning

The growth of computational resources at the beginning of the twenty-first century has expedited the development of modern artificial intelligence (AI). Deep learning (DL) lies at the center of this AI research and development. Many branches of deep learning gradually adopted different variations of AI models popularly known as *deep neural network*. As hardware computational resources keep growing, deep neural networks (DNNs) have become extremely popular in demonstrating exceptional performance in image recognition, speech recognition, medical applications, and whatnot. This chapter summarizes the core algorithms of existing deep neural networks. To do so, first, we look at the core element of every neural network, formally known as a *neuron*.

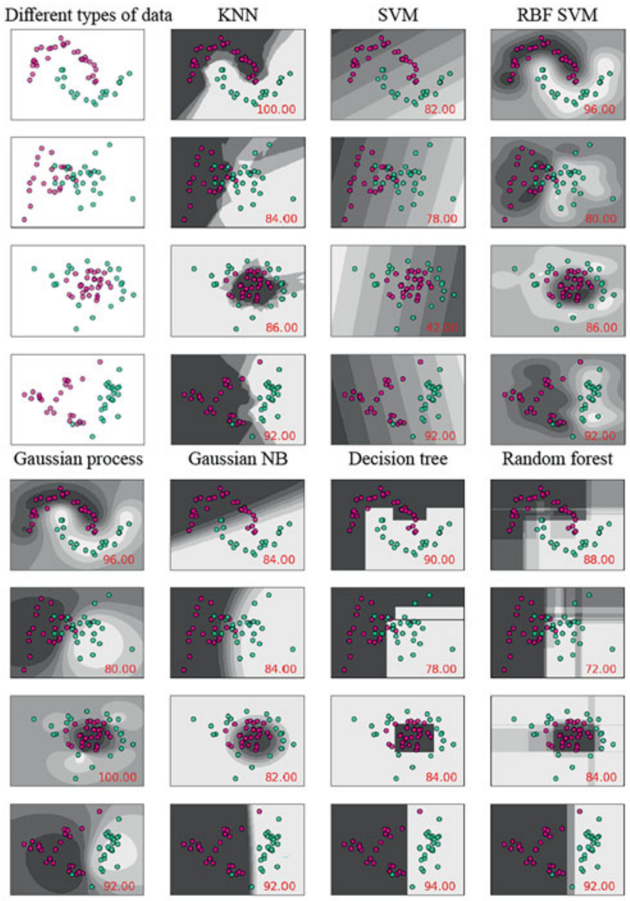


Fig. 3.18 Performance of different classifiers on four types of different data for a two-class classification problem

3.5.1 What Is a Neuron?

Inspired by the biological concept of neurons in the human brain, in 1958, psychologist Frank Rosenblatt proposed a neuron in an AI model. Neurons are the building block of a neural network. In a modern neural network, billions of neurons combine to perform a specific task. While neurons are formed using a basic mathematical formula, billions of them put together can do wonders in solving complex practical problems.

As displayed in Fig. 3.19, a formal structure of a neuron takes an input $\boldsymbol{x} \in \mathbb{R}^2$ which has two dimensions, where x_1 and x_2 are the two-dimensional features. Now consider a simple computation where two variables w_1 and w_2 form a simple mathematical model of a neuron, given by Eq. 3.29.

Table 3.13 Explanation of Listing 3.2

Line number	Description
2–15	Using scikit-learn modules
18–31	Defining the models
34	Moon shape two-class dataset
37–39	A separable Gaussian two-class dataset
42	Gaussian data with circle
45	Gaussian data with two overlap classes
59	Preprocessing the data
63–84	Preparing the plot for decision boundary
89–90	Classifier training and accuracy computation
92–118	Plotting the decision boundary

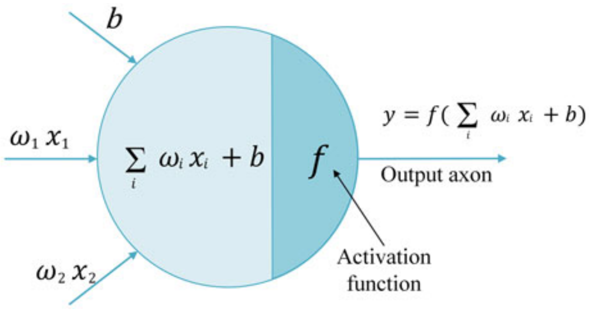


Fig. 3.19 The structure of a neuron used in neural networks

$$\begin{aligned} y &= f(\hat{y}), \\ &= f(w_1x_1 + w_2x_2 + b), \\ &= \sigma(w_1x_1 + w_2x_2 + b). \end{aligned} \tag{3.29}$$

Here, w_1 and w_2 form a simple feed-forward neural network with two weights and a bias, b. Here f can be any activation function, such as the sigmoid function (σ). The goal of a neuron is to predict an output based on the weights and bias values. In order to use this neuron to predict the future output, the weights and bias values must be trained or adjusted based on the existing data. We call this the training stage of a neural network. In the next section, we will discuss the details of the training process of a neuron.

3.5.2 Backpropagation and Gradient Descent

Backpropagation is the key algorithm in training a neuron of a deep neural network. We need backpropagation because the weights and biases will be updated based on the gradient information of an error function with respect to weights and biases. Let us assume that the values of x_1 and x_2 at the input will ideally generate a target output y_t. The goal is to learn the values of w_1, w_2, and b so that we can project the target value at the output of the neuron. The error function can be a simple mean squared error function given by

$$\mathcal{L} = (y - y_t)^2. \tag{3.30}$$

Recall the gradient descent algorithm discussed in Sect. 1.5.6 to minimize this error. We need to compute the gradient of $\mathcal{L}$ with respect to each trainable parameter (e.g., weights and biases) so that the values of $\mathcal{L}$ can be minimized. For example, the gradient of Eq. 3.30 with respect to w_1 would be given by

$$\frac{\partial \mathcal{L}}{\partial w_1} = 2(y - y_t)\frac{\partial y}{\partial w_1}. \tag{3.31}$$

However, $y = \sigma(w_1x_1 + w_2x_2 + b)$ is not directly differentiable with respect to w_1. To counter this, a formal approach is to apply the chain rule, as shown in the equation below:

$$\frac{\partial \mathcal{L}}{\partial w_1} = 2(y - y_t)\frac{\partial y}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w_1} = 2(y - y_t)\frac{\partial \sigma(\hat{y})}{\partial \hat{y}}\frac{\partial (w_1x_1 + w_2x_2 + b)}{\partial w_1}. \tag{3.32}$$

Thus the final gradient would be equal to

$$\frac{\partial \mathcal{L}}{\partial w_1} = 2(y - y_t)\frac{\partial \sigma(\hat{y})}{\partial \hat{y}}x_1. \tag{3.33}$$

The formulation can even be simplified for the sigmoid case as for sigmoid function $\frac{\partial \sigma(\hat{y})}{\partial \hat{y}} = \sigma(\hat{y})(1 - \sigma(\hat{y}))$. In a similar way, the $\frac{\partial \mathcal{L}}{\partial w_2}$ and $\frac{\partial \mathcal{L}}{\partial b}$ can also be computed:

$$\frac{\partial \mathcal{L}}{\partial w_2} = 2(y - y_t)\sigma(\hat{y})(1 - \sigma(\hat{y}))x_2, \tag{3.34}$$

$$\frac{\partial \mathcal{L}}{\partial b} = 2(y - y_t)\sigma(\hat{y})(1 - \sigma(\hat{y})). \tag{3.35}$$

Once the gradient is computed at each training step, the values of weights and biases can be updated based on the gradient descent formula:

$$w_1 = w_1 - \alpha\frac{\partial \mathcal{L}}{\partial w_1}, \tag{3.36}$$

$$w_2 = w_2 - \alpha\frac{\partial \mathcal{L}}{\partial w_2}, \tag{3.37}$$

Table 3.14 Data for neuron training

Features	Output
(1,1)	0.25
(−1,−1)	−0.25

$$b = b - \alpha \frac{\partial \mathcal{L}}{\partial b}. \tag{3.38}$$

The backpropagation algorithm continues until convergence, i.e., the target output y_t and neuron-generated output y are almost equal. This is a widely adopted approach to training a neuron, even in a DNN, where the number of neurons may reach billions. Next, we will go through a very simple coding example of training and testing a neuron.

Example 3.5 Train a neuron so that it gives the output equal to 0.25 when the inputs are $x_1 = 1$ and $x_2 = 1$; else, it will give the output equal to -0.25 when the inputs are $x_1 = -1$ and $x_2 = -1$. The dataset is given below in Table 3.14.

Solution to Example 3.5
Programming Example 3.4
The code for training the neuron is presented in Listing 3.3, followed by its output and explanation in Table 3.15. The listing addresses a two-class classification problem. A class, `Neuron`, is defined using a basic neural network, which is trained to minimize the MSE loss. The training process occurs over a set number of epochs, during which the code performs a forward pass and calculates gradients. Weights and biases are then updated using gradient descent.

```python
import numpy as np
import math

# This is a two class classification problem which will be solved
     using the sample neuron
# ----------------------Declaring Variables----------------------
learning_rate = 0.1 # Learning rate
trainig_epochs = 400 # Trainig epochs


# ------------------------Data Preparation-----------------------
# Two features for two classes for trainig
x_class1_feat1 = 1
x_class1_feat2 = 1
x_class2_feat1 = -1
x_class2_feat2 = -1
y_class1 = 0.25 # Target for class 1
y_class2 = -0.25 # Target for class 2
```

```python


# --------------Defining the Neuron and Activation---------------

def sigmoid(x):
    "Sigmoid function output of a scalar value x"
    return 1.0 / (1.0 + math.exp(-x))

class Neuron(object):
    "This function will take two input x1 and x2 to produce an
    output y neuron output"
    def __init__(self):

        self.w1 = 0.25 # Initializing the weight1 to 0.25
        self.w2 = 0.25 # Initializing the weight2 to 0.25
        self.b = 0.25 # Initializing the bias to 0.25

    def update_weights_biases (self,grad_w1,grad_w2,grad_b,
    learning_rate):

        # Gradient update using the gradient descent formula
        self.w1 = self.w1 - learning_rate * grad_w1
        self.w2 = self.w2 - learning_rate * grad_w2
        self.b = self.b - learning_rate * grad_b

        return

    def sigmoid(self,x): # Sigmoid function
        return 1.0 / (1.0 + math.exp(-x))

    def grad_w1(self,x1,output,target): # Gradient of W1
        gradient  =  2 * (output-target) * output*(1-output) * x1
        return gradient

    def grad_w2(self,x2,output,target): # Gradient of W2
        gradient  = 2 * (output-target) * output*(1-output) * x2
        return gradient

    def grad_b(self,output,target): # Gradient of bias
        gradient  = 2 * (output-target) * output*(1-output)
        return gradient


    def forward (self, x1,x2):
        # forward path
        y_in = np.sum( self.w1*x1 + self.w2*x2 + self.b ) # W*X +
     b
        y = self.sigmoid(y_in)

        return y

def error_function (x,y):
    "Computes the MSE between x and y"
    cost = np.sum((x-y) ** 2) # Mean Squred Error
    return  cost


# Defining the neuron
neuron = Neuron()


# -----------------Traininig Loop for the Neuron-----------------

for i in range(trainig_epochs):

        # Forward class1 and cost
        output1 = neuron.forward(x_class1_feat1,x_class1_feat2 )
        cost1 = error_function (output1,y_class1)

        # Gradient
        grad_w1 = neuron.grad_w1(x_class1_feat1,output1,y_class1)
        grad_w2 = neuron.grad_w2(x_class1_feat2,output1,y_class1)
        grad_b = neuron.grad_b(output1,y_class1)

        # Update
        neuron.update_weights_biases (grad_w1,grad_w2,grad_b,
    learning_rate)

        # Forward class2 and cost
        output2 = neuron.forward(x_class2_feat1,x_class2_feat2)
        cost2 = error_function (output2,y_class2)

        # Gradient
        grad_w1 = neuron.grad_w1(x_class2_feat1,output2,y_class2)
        grad_w2 = neuron.grad_w2(x_class2_feat2,output2,y_class2)
        grad_b = neuron.grad_b(output2,y_class2)

        # Update
        neuron.update_weights_biases (grad_w1,grad_w2,grad_b,
    learning_rate)

        if i == 0:
            print("Initial Iteration values")
            print("Cost =", cost1+cost2)
            print("weight1 = ", neuron.w1)
            print("weight2 = ", neuron.w2)
            print("bias = ", neuron.b)

print("Final Solution")
print("Cost =", cost1+cost2)
print("weight1 = ", neuron.w1)
print("weight2 = ", neuron.w2)
print("bias = ", neuron.b)
```

Listing 3.3 Neuron training code.

Table 3.15 Explanation of the training and testing code example of a neuron presented in Listing 3.3

Line number	Description	Equations
1–2	Using NumPy and math modules	
6–7	Defining the hyper-parameters	
12–17	Preparing the dataset as per Table 3.14	
22–24	Sigmoid function	Provided in Table 3.17
30–32	Initialize w_1, w_2, b	This is a random guess
37–39	Updating learnable parameter	Equations 3.36, 3.37, and 3.38
46–55	Gradient of Loss with respect to w_1, w_2, b	Equations 3.33, 3.34, and 3.35
61–62	Output computation	Equation 3.29
68	Mean Squared error	Provided in Table 1.7
81–87	Compute output and gradient for one class	
90	Update weights and biases	
93–99	Compute output and gradient for second class	
102	Update weights and biases again	
104–115	Printing outputs	

Output of Listing 3.3:

```
Initial Iteration values
Cost = 0.6636567265251024
weight1 =  0.26545948169695005
weight2 =  0.26545948169695005
bias =  0.19713416231123965
Final Solution
Cost = 0.0720120329701401
weight1 =  0.735913945982327
weight2 =  0.735913945982327
bias =  -2.5201486542891023
```

3.5.3 Artificial Neural Network (ANN)

Artificial neural networks (ANNs) are popularly known as *feed-forward networks*. An ANN usually consists of many neurons stacked together at specific layers. A standard ANN should have an input layer, one or multiple hidden layers, and one output layer as displayed in Fig. 3.20.

Here, the input layer takes multiple features as the inputs. For example, in this chapter, we will mainly demonstrate a standard handwritten dataset problem known as *MNIST*, which stands for *Modified National Institute of Standards and Technology* and is shown in Fig. 3.21 [8, 9]. The MNIST is a popular visual recognition dataset for image classification problems. It contains grayscale images with size 28 × 28. This handwritten digit dataset has ten (0–9) different classes.

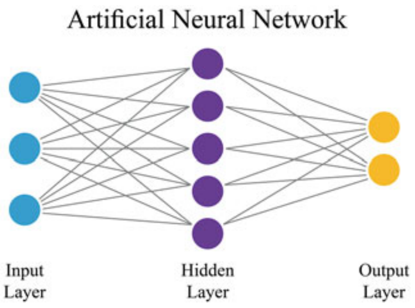


Fig. 3.20 The input layers, hidden layers, and the output layer of a typical artificial neural network

Fig. 3.21 MNIST handwritten dataset [8]

Programming Example 3.5
Next, we will look at a PyTorch example in Listing 3.4 to learn how to apply ANN in this kind of classification problem. The code explanation is provided in Table 3.16. The code performs a training and an evaluation cycle for the ANN model on the MNIST dataset for simple digit classification. The dataset is preprocessed before splitting into training and testing datasets. The ANN is implemented using the cross-entropy loss function and stochastic gradient descent optimizer. The weights are updated through forward and backward propagation. Once the training is complete, the model's performance is evaluated using the test function on the test dataset.

```python
# -------------------------Torch Modules-------------------------
from __future__ import print_function
import numpy as np
import pandas as pd
import torch.nn as nn
import math
import torch.nn.functional as F
import torch
from torch.nn import init
import torch.optim as optim
from torchvision import datasets, transforms
from torchvision import models
import torch.nn.functional as F

```

```

# ---------------------------Variables---------------------------
mean = [0.5] # For Normalization
std = [0.1]

BATCH_SIZE =128 # Batch size
Iterations = 20
learning_rate = 0.01


# -------Commands to download and prepare the MNIST dataset------
train_transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
        ])

test_transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
        ])


train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('./mnist', train=True, download=True,
                       transform=train_transform),
        batch_size=BATCH_SIZE, shuffle=True) # train dataset
test_loader = torch.utils.data.DataLoader(
        datasets.MNIST('./mnist', train=False,
                         transform=test_transform),
        batch_size=BATCH_SIZE, shuffle=False) # test dataset


# -------------------------Defining ANN--------------------------
class ANN(nn.Module):
    def __init__(self):
        super(ANN, self).__init__()
        self.l1 = nn.Linear(784, 100)  # input layer 784 for
    mnist and 100 neurons hidden layer
        self.relu = nn.ReLU() # activation function
        self.l3 = nn.Linear(100, 10) ## from 100 neuron hidden
    layer to output 10 layer for 10 digits

    def forward(self, x):
        x = torch.flatten(x, 1) ## making the 28 x 28 images into
     a 784 dimension input
        x = self.l1(x)
        x = self.relu(x)
        x = self.l3(x)
        return x

# defining ANN model
model = ANN()

```

```python
## Loss function
criterion = torch.nn.CrossEntropyLoss() # pytorch's cross entropy loss function

# Definin which paramters to train only the ANN model parameters
optimizer = torch.optim.SGD(model.parameters(),learning_rate)

# Defining the training function
# Train baseline classifier on clean data
def train(model, optimizer,criterion,epoch):
    model.train() # setting up for training
    for batch_idx, (data, target) in enumerate(train_loader): # data contains the image and target contains the label = 0/1/2/3/4/5/6/7/8/9
        optimizer.zero_grad() # setting gradient to zero
        output = model(data) # forward
        loss = criterion(output, target) # loss computation
        loss.backward() # back propagation here pytorch will take care of it
        optimizer.step() # updating the weight values
        if batch_idx % 100 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))


# To evaluate the model
# Validation of test accuracy
def test(model, criterion, val_loader, epoch):
    model.eval()
    test_loss = 0
    correct = 0

    with torch.no_grad():
        for batch_idx, (data, target) in enumerate(val_loader):

            output = model(data)
            test_loss += criterion(output, target).item() # Sum up batch loss
            pred = output.max(1, keepdim=True)[1] # Get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item() # If pred == target then correct +=1

    test_loss /= len(val_loader.dataset) # Average test loss
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.4f}%)\n'.format(
        test_loss, correct, val_loader.sampler.__len__(),
        100. * correct / val_loader.sampler.__len__() ))

```

```

# Training the ANN
for i in range(Iterations):
    train(model, optimizer,criterion,i)
    test(model, criterion, test_loader, i) # Testing the the
    current ANN
```

Listing 3.4 ANN Example

Output of Listing 3.4:

```
Train Epoch: 0 [0/60000 (0%)] Loss: 2.461659
Train Epoch: 0 [12800/60000 (21%)] Loss: 0.594805
Train Epoch: 0 [25600/60000 (43%)] Loss: 0.318740
Train Epoch: 0 [38400/60000 (64%)] Loss: 0.300817
Train Epoch: 0 [51200/60000 (85%)] Loss: 0.354926

Test set: Average loss: 0.0022, Accuracy: 9169/10000 (91.6900%)

Train Epoch: 1 [0/60000 (0%)] Loss: 0.332806
Train Epoch: 1 [12800/60000 (21%)] Loss: 0.337252
Train Epoch: 1 [25600/60000 (43%)] Loss: 0.180066
Train Epoch: 1 [38400/60000 (64%)] Loss: 0.173537
Train Epoch: 1 [51200/60000 (85%)] Loss: 0.310849

Test set: Average loss: 0.0018, Accuracy: 9360/10000 (93.6000%)

Train Epoch: 2 [0/60000 (0%)] Loss: 0.267431
Train Epoch: 2 [12800/60000 (21%)] Loss: 0.146986
Train Epoch: 2 [25600/60000 (43%)] Loss: 0.224672
Train Epoch: 2 [38400/60000 (64%)] Loss: 0.221327
Train Epoch: 2 [51200/60000 (85%)] Loss: 0.281612

Test set: Average loss: 0.0015, Accuracy: 9446/10000 (94.4600%)

... ... ...
... ... ...
... ... ...

Train Epoch: 18 [0/60000 (0%)] Loss: 0.054460
Train Epoch: 18 [12800/60000 (21%)] Loss: 0.030821
Train Epoch: 18 [25600/60000 (43%)] Loss: 0.076624
Train Epoch: 18 [38400/60000 (64%)] Loss: 0.084313
Train Epoch: 18 [51200/60000 (85%)] Loss: 0.109154

Test set: Average loss: 0.0007, Accuracy: 9729/10000 (97.2900%)

Train Epoch: 19 [0/60000 (0%)] Loss: 0.085309
Train Epoch: 19 [12800/60000 (21%)] Loss: 0.034182
Train Epoch: 19 [25600/60000 (43%)] Loss: 0.035189
Train Epoch: 19 [38400/60000 (64%)] Loss: 0.057047
Train Epoch: 19 [51200/60000 (85%)] Loss: 0.082779

Test set: Average loss: 0.0007, Accuracy: 9734/10000 (97.3400%)
```

Table 3.16 Explanation of the ANN code presented in Listing 3.4

Line number	Description
2–13	Using NumPy, pandas, and Torch modules
17–18	Mean and standard deviation for data normalization
20–22	List of hyper-parameters
26–34	Converting to tensor and normalization
37	Train loader has the train dataset 60k images
42	Test loader has 10k test images
48–60	Describing ANN
51	First linear layer 784 input features and 100 output
52	ReLu Activation
53	Output linear layer 100 input and 10 output class
56	Flattens a 28×28 image into 1D 784 features
66	Defining the cross-entropy loss of PyTorch
69	Using PyTorch SGD solver as optimizer
73–84	Training function
74	Setting the model for training
75	Loading the data and target for training
77–79	Computes output and calculates loss and backpropagation
80	This line updates the weights and other trainable parameters
89–110	Test function
90	Setting the model for evaluation
94	We do not need gradient here (i.e., no grad)
97–99	Computes output, calculates loss, and predicts
100	If predict = target then correct prediction
109–111	Training and testing the ANN

3.5.4 Convolutional Neural Network

A convolutional neural network (CNN) is a popular network architecture used for image classification purposes. Most of the image classification tasks nowadays use some form of CNN. Typically, a CNN consists of a convolution layer, a pooling layer, an activation function, a batch normalization (BN) layer, and a fully connected (FC) layer. These layers are described in the following sections.

3.5.4.1 Convolution Layer

In Fig. 3.22, a basic convolution operation is visualized. A convolutional layer contains filters/weights (i.e., kernel) and biases that are trainable parameters. The figure displays how these weights and biases are used to compute the output of a convolution layer from its inputs. In a practical CNN, there are many input channels and output channels. Figure 3.22 shows a 7×7 input and a 3×3 filter which goes through a series of multiplication and addition operations (1, 2, 3, 4, 5, 6, 7, 8, and 9). In each consecutive figure (from $1\rightarrow2$, $2\rightarrow3$), the kernel window shifts by

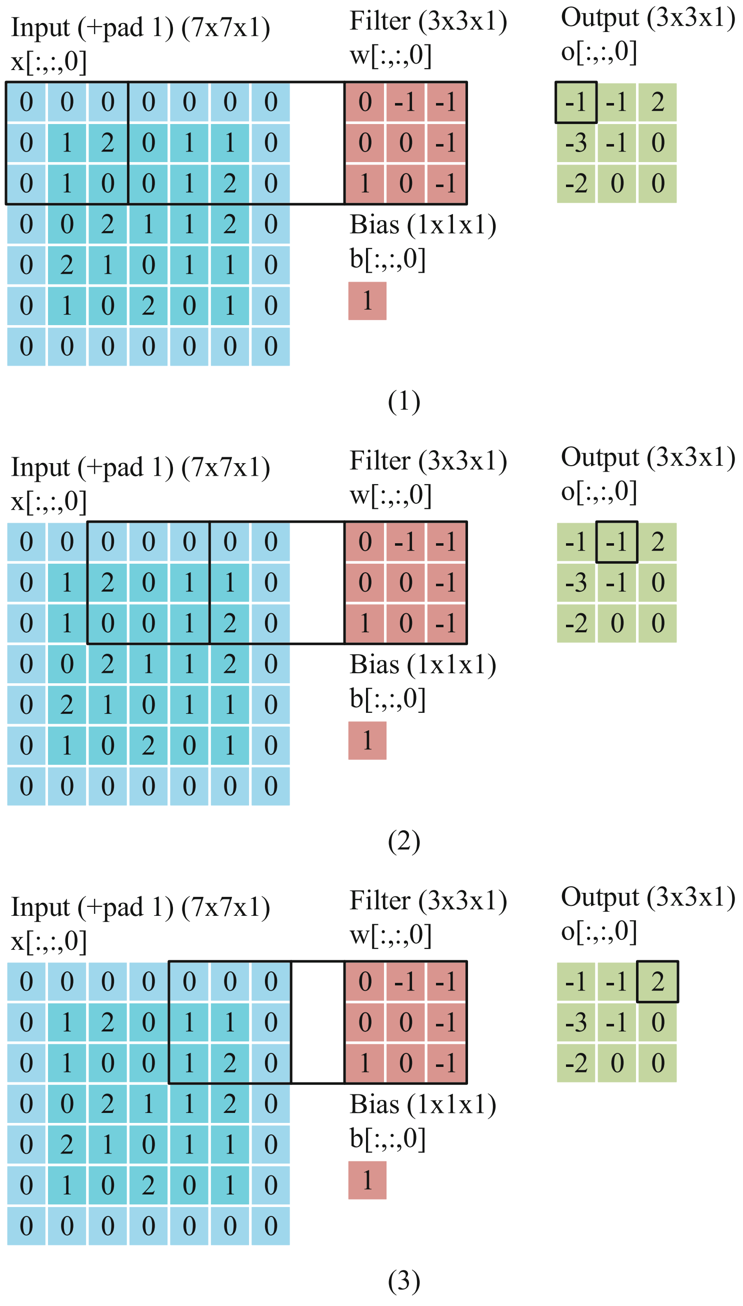
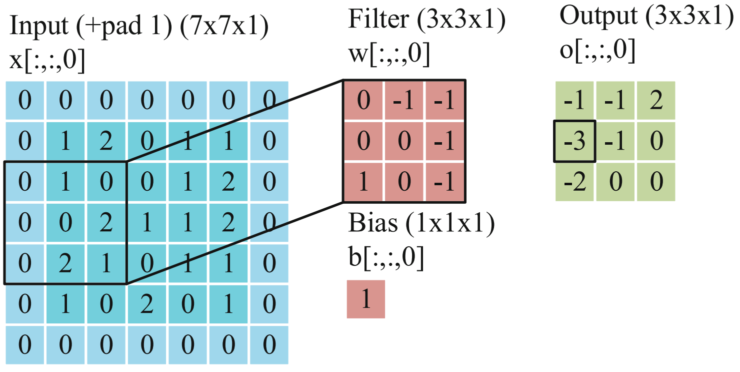
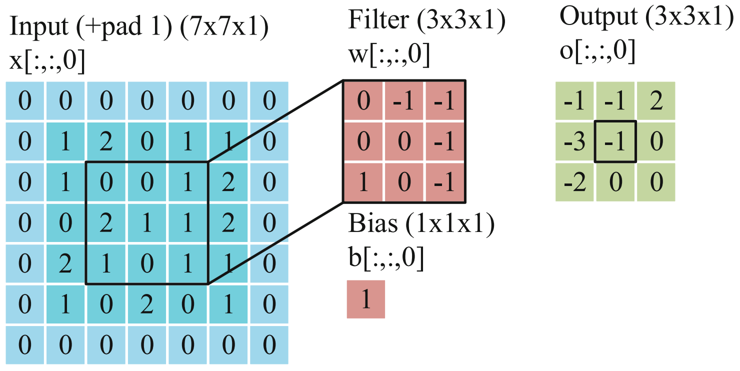


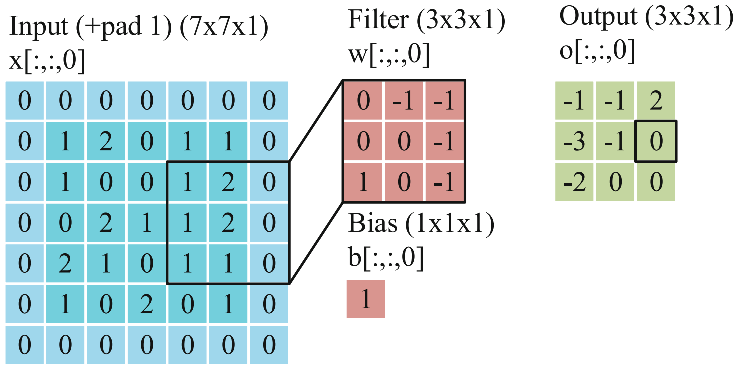
Fig. 3.22 Visualization of convolution operation with one input channel and one output channel using one filter. A 3×3 filter with padding $= 1$ and stride $= 2$ is used on an input size of 6×6. At each window the corresponding output block is computed by multiplying each element of a kernel and input and then adding them with the bias



(4)

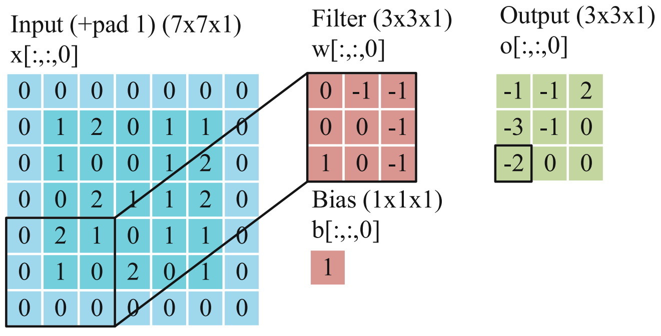


(5)

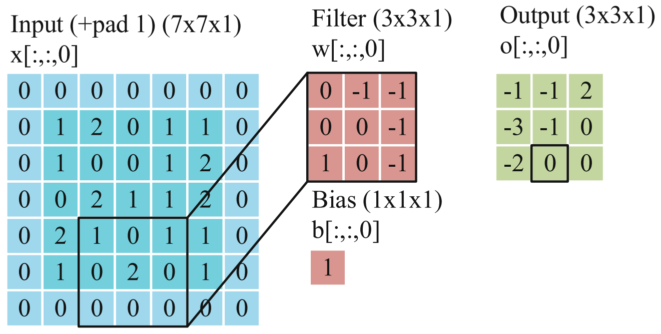


(6)

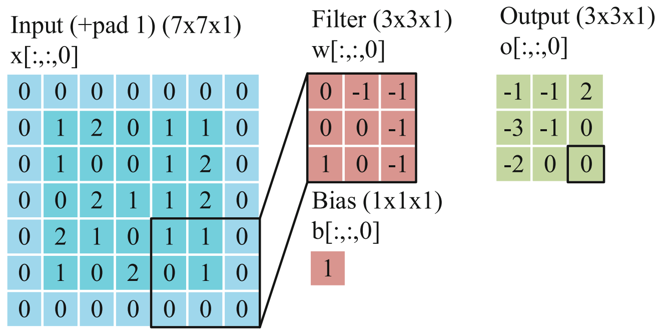
Fig. 3.22 (continued)



(7)



(8)



(9)

Fig. 3.22 (continued)

2 places to the right. This sliding window step is known as stride (e.g., 2 in this case). Additionally, since 7 (input size) is not divisible by 3 (kernel width), the input is zero-padded on both sides to make it 9×9. Finally, the output at each step is computed by an element-wise dot product between the kernel and the input element and adding each element with the bias (e.g., 1).

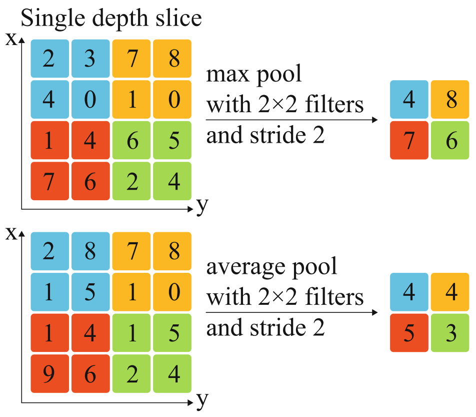


Fig. 3.23 Visualization of maximum pooling at the top and average pooling at the bottom

3.5.4.2 Pooling Layer

A pooling layer is usually used in between two convolutional layers. The purpose of the pooling operation is to reduce the activation size of the previous convolutional layer. A pooling layer generally reduces the activation map size and helps reduce memory consumption during training, as displayed in Fig. 3.23. There are different pooling layers, including but not limited to max pooling and average pooling. Maximum pooling (or max pooling) generates the maximum value in each patch of the feature map as the output. Average pooling determines the average value of each patch (e.g., 2×2) on the feature map.

3.5.4.3 Activation Functions

The concept of activation function has been introduced in Chap. 1. The activation functions that are generally used in CNN models are discussed in the following paragraphs. Table 3.17 summarizes the seven types of activations functions discussed here with their equations. The graphical representations of the functions are illustrated in Fig. 3.24.

1. **Binary Step Function:** A binary step function produces an output equal to 1 if a certain threshold value of the input is reached and outputs 0 if the input is less than the threshold. Since the output is constant, the gradient of the function is zero, causing a hindrance in the backpropagation process.
2. **Identity Function:** The identity function creates an output signal equal to the input signal. It outputs the same value as its input, so, technically, this function is similar to the input multiplied by 1. Based on the input, the outputs of the identity function may range from $-\infty$ to $+\infty$. The derivative of this function is constant, so it is not possible to backpropagate to the original function.
3. **Sigmoid Function:** The sigmoid function resembles an S-shaped curve, as shown in Fig. 3.24. The value of the sigmoid function ranges from 0 to 1, which is why it is extensively used in probabilistic applications. Furthermore, the function is differentiable, implying that its slope can be determined. In ML, the sigmoid

Table 3.17 Different activation functions used in machine learning. Here, a is constant, and x is variable

Name of function	Equation
Identity function	$f(x) = x$
Binary step function	$f(x) = \begin{cases} 0, & \text{when } x < 0 \\ 1, & \text{when } x \geq 0 \end{cases}$
Sigmoid	$f(x) = \frac{1}{1 + e^{-x}}$
tanh	$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
Rectified linear unit (ReLU)	$f(x) = \begin{cases} 0, & \text{when } x < 0 \\ x, & \text{when } x \geq 0 \end{cases}$
Parametric rectified linear unit (PReLU)	$f(x) = \begin{cases} \alpha x, & \text{when } x < 0 \\ x, & \text{when } x \geq 0 \end{cases}$
Exponential linear unit (ELU)	$f(x) = \begin{cases} \alpha.(e^z - 1), & \text{when } x < 0 \\ x, & \text{when } x \geq 0 \end{cases}$

function is useful for mapping predictions to a probability. The sigmoid function was coined by Pierre François Verhulst in three papers from 1838 to 1847 in his attempts to model population growth by adjusting the exponential growth model.

4. **Hyperbolic Tangent Function (tanh):** The functionality of tanh is similar to the sigmoid function up to some level, as depicted in Fig. 3.24. However, unlike the sigmoid function, tanh outputs in the range of -1 to 1. While developing an ML model, it can be useful to let the model decide which direction to go and how much change should be done in the process.
5. **Rectified Linear Unit (ReLU):** ReLU is a linear function that provides an output equal to the input only when the input is greater than zero and outputs 0 otherwise. The output values range from 0 to $+\infty$, i.e., always non-negative since the inputs must be non-negative for a non-zero output. Backpropagation from the derivative of the function is possible in this case. The ReLU function was first proposed by Nair and Hinton in their 2010 publication [10], which has been cited over 13,000 times, thus proving the usefulness of the function in ML applications, particularly in deep learning.
6. **Parametric Rectified Linear Unit (PReLU):** The Parametric Rectified Linear Unit (PReLU) is an activation function used in neural networks which solves the issue of vanishing gradient problems of other activation functions. Furthermore, the PReLU activation function can adapt to drastic changes in the slope by using backpropagation. The range of this function depends on the convergence value of α, which in turn depends on the training data. The PReLU function was first laid down by He et al. in a conference paper in 2015 [11], having more than 11,500 citations.

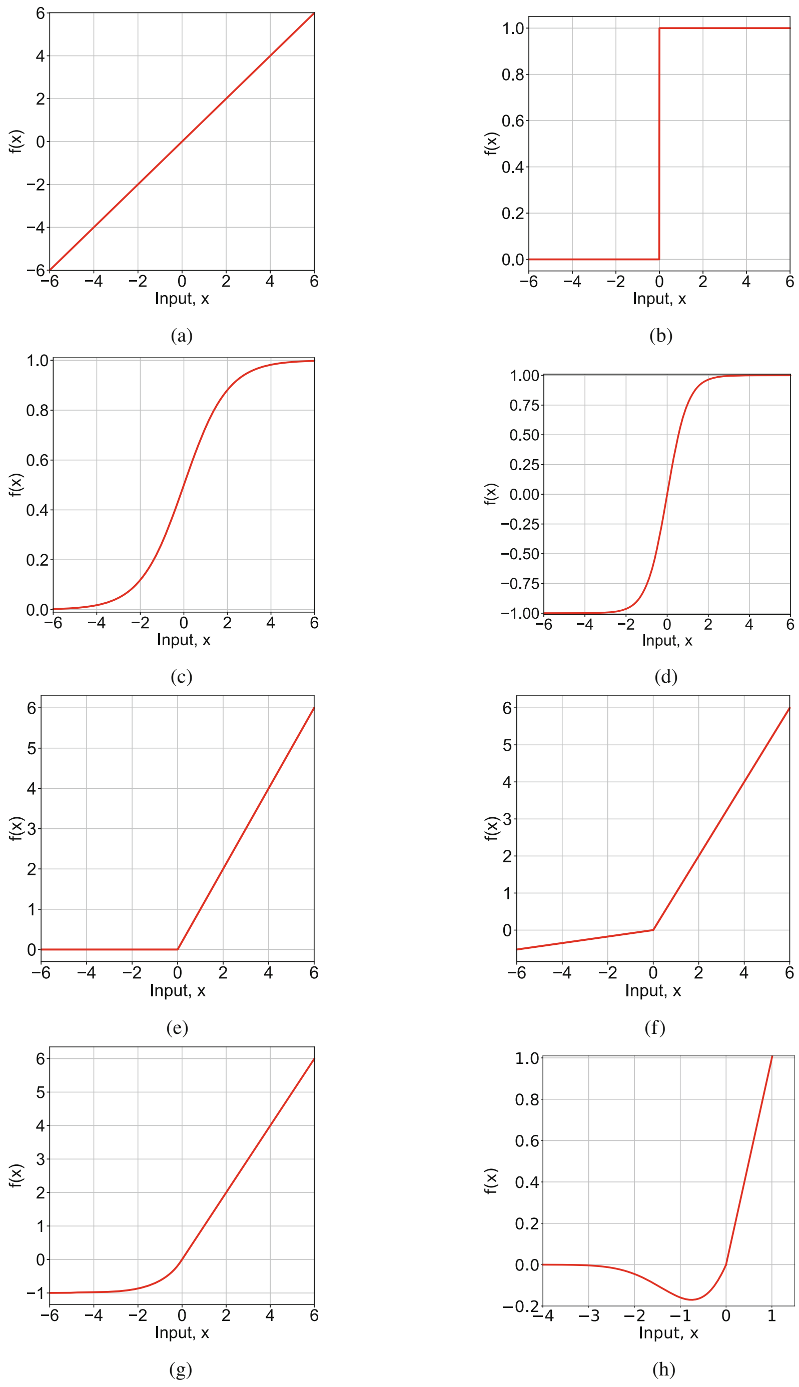


Fig. 3.24 The graphical illustration of different activation functions used in machine learning. (**a**) Identity function. (**b**) Binary step function. (**c**) Sigmoid function. (**d**) Tanh function. (**e**) Rectified linear unit. (**f**) Parametric rectified linear unit. (**g**) Exponential linear unit. (**h**) GELU

7. **Exponential Linear Unit (ELU):** The Exponential Linear Unit (ELU) is one of the activation functions used in neural networks. The ELUs use negative values to bring mean unit activation values to almost zero with way less computational complexity. While other activation functions, such as LReLUs and PReLUs, also use zero values, the ELU provides a more stable deactivation state unaffected by noise. Clevert et al. first proposed the ELU function in 2015 [12].
8. **Gaussian Error Linear Unit (GELU):** The GELU thresholds inputs based on their values rather than their signs, unlike ReLU. Furthermore, it is a function that multiplies its input by a cumulative density function (CDF) which enables the combination of non-linearity and stochasticity. Equation 3.39 presents the mathematical formula for GELU [13]:

$$GELU(x) = xP(X \leq x) = x\Phi(x). \tag{3.39}$$

Here, $\Phi(x)$ is the CDF for Gaussian distribution. This CDF can be expressed using different approximations. For error function (erf) approximation, GELU can be represented as

$$GELU(x) = x\Phi(x) = x.\frac{1}{2}\left[1 + erf\left(\frac{x}{\sqrt{2}}\right)\right]. \tag{3.40}$$

For tanh approximation, GELU can be expressed as shown in the equation below:

$$GELU(x) \approx \frac{1}{2}x\left(1 + \tanh\left[\frac{x}{\sqrt{2}}\left(x + 0.044715x^3\right)\right]\right). \tag{3.41}$$

GELU is a non-convex and non-monotonic non-linear function with curvature and thus can approximate complicated functions better than ReLU and other linear units. GELU has been implemented in BERT, ROBERTa, ALBERT, and many other state-of-the-art natural language processing (NLP) models.

3.5.4.4 Dropout

Dropout is a popular regularization technique. These regularization techniques are used in neural networks to prevent overfitting. Usually, a CNN/DNN suffers from overfitting when the model has a large number of trainable parameters. Dropout has a pre-defined rate that defines the probability of a neuron being dropped during one iteration of the training process to resolve the overfitting issue. The pre-defined rate is known as the *dropout rate*. Dropout helps remove specific neurons during training to compensate for the overfitting of a DNN model.

3.5.4.5 Batch Normalization

A batch normalization (BN) is a layer that scales the output between different convolutional layers. Typically, a BN would contain additional trainable parameters as well. In Eq. 3.42, we show a channel-wise (c) BN normalization operation that takes x_c as input across a particular channel c.

$$O_c = \gamma_c * \frac{(x_c - \mu_c)}{\sqrt{\sigma_c^2 + \epsilon)}} + \beta_c, \tag{3.42}$$

where O_c is the output of channel c, μ_c is the channel-wise mean, and σ_c is the standard deviation. γ_c and β_c are the trainable parameters that are trained using the gradient descent algorithm. Finally, ϵ is a small value that ensures BN does not suffer from the divide by zero operation.

3.5.4.6 Optimizers

An optimizer is an algorithm or a function that minimizes the loss function by adjusting the attributes of the neural network, such as weights and learning rates. In this section, we will discuss five different types of optimizers.

3.5.4.6.1 Gradient Descent

The theory of gradient descent has already been discussed in Sect. 2.7.7 as a hyperparameter tuner. Here, we discuss gradient descent briefly as an optimizer. This algorithm iterates the entire dataset each time to look for the optimal solution. However, in search of global minima, the gradient descent optimizer may converge to local minima and fail to recover from it. Moreover, this is a slower and computationally expensive optimizer. The formula the gradient descent optimizer uses to update its parameters is shown as follows:

$$w_{i+1} = w_i - \alpha.\nabla_{w_i} J(w_i), \tag{3.43}$$

where w is the weight parameter, i is the iteration index, α is the learning rate, J is the cost function, and ∇ is used as an operator expressing the gradient of a function at a particular point, i.e., $\nabla_{w_i} J$ is the gradient of the cost function J at w_i.

3.5.4.6.2 Stochastic Gradient Descent (SGD)

To solve the issue of computation costs of gradient descent optimizer, SGD has been introduced. It does not iterate the entire dataset each time. Instead, it iterates a randomly shuffled dataset partition each time to reach the optimal solution. This approach makes SGD computationally faster and cheaper than gradient descent optimizers. However, this approach may make SGD prone to noise and outliers in the dataset as it only considers a random portion of the dataset at a time. The formula for the SGD optimizer is given below:

$$w_{i+1} = w_i - \alpha.\nabla_{w_i} J\left(x^i y^i; w_i\right), \tag{3.44}$$

where w is the weight parameter, i is the iteration index, α is the learning rate, J is the cost function, x is the randomly shuffled dataset partition on a single observation, and y is labeled data.

3.5.4.6.3 Mini Batch Stochastic Gradient Descent (MB-SGD)

Mini Batch Stochastic Gradient Descent (MB-SGD) is a variant of an SGD optimizer. It would be more appropriate to state that the MB-SGD optimizer is an optimized combination of gradient descent and SGD. First, it divides the entire dataset into different partitions called batches. Then it calculates the gradient for each batch and, finally, an average gradient mini-batch. This average gradient is used to update the weight parameters. This optimizer is more computationally cost-efficient and time sufficient. As the MB-SGD optimizer is faster compared to the previous optimizers, it can be used on extensive datasets. The formula for MB-SGD optimizer is given below:

$$w_{i+1} = w_i - \alpha.\nabla_{w_i} J\left(x^{i:i+b}, y^{i:i+b}, w_i\right), \tag{3.45}$$

where w is the weight parameter, i is the iteration index, α is the learning rate, J is the cost function, x is the randomly shuffled dataset partition, y is labeled data, and b is the size of a single batch.

3.5.4.6.4 RMSprop Optimizer

The two gradients in the root mean squared propagation (RMSprop) algorithm are first compared for signs. If they both have the same sign, then we are moving in the right direction and can, therefore, slightly raise the step size. However, we must reduce the step size if they have the opposite indications. After limiting the step size, we may proceed with the weight update. The algorithm's main goal is to shorten the number of function evaluations necessary to obtain the local minima, hence quickening the optimization process. The algorithm divides the gradient by the square root of the mean square and keeps a moving average of the squared gradients for each weight.

3.5.4.6.5 Adam Optimizer

The Adam optimization algorithm has been developed as an extension to the classical gradient descent algorithms. It has been designed to maintain the advantages of adaptive gradient and RMSProp algorithms. The term Adam is derived from *adaptive moment estimation*. Adam optimizer uses bias correction and the estimations of the first and second moments of the gradient for adjusting the learning rate for each weight of the neural network.

Adam optimizer is computationally efficient and fast. It does not require huge memory space. For extensive datasets, the Adam optimizer is the appropriate algorithm. Adam optimizers work well with noisy and sparse gradients. It can be applied if the dataset contains non-stationary data.

3.5.4.7 Fully Connected Layer

We already discussed an example of a fully connected linear layer in ANN. In a classification problem, the last fully connected layer in a CNN network will always have an output neuron equal to the amount of classification class. For example, in our previous MNIST classification problem, the output class has 10 neurons and 10 output classes (e.g., 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9).

Programming Example 3.6
Next, we look at the example of a CNN modeling in Listing 3.5. The code implements a CNN model for digit classification on the MNIST dataset. The problem statement is exactly similar to the previous ANN model example in Listing 3.4. The only difference is we replace the ANN model with a CNN architecture in lines 52–57. The CNN architecture contains two convolution layers, two dropout layers, and two fully connected linear layers.

```
1  # -------------------------Torch Modules-------------------------
2  from __future__ import print_function
3  import numpy as np
4  import pandas as pd
5  import torch.nn as nn
6  import math
7  import torch.nn.functional as F
8  import torch
9  from torch.nn import init
10 import torch.optim as optim
11 from torchvision import datasets, transforms
12 from torchvision import models
13 import torch.nn.functional as F
14
15
16 # ---------------------------Variables---------------------------
17 mean = [0.5] # for Normalization
18 std = [0.1]
19 # batch size
20 BATCH_SIZE =128
21 Iterations = 20
22 learning_rate = 0.01
23
24
25 # -------Commands to download and prepare the MNIST dataset------
26 train_transform = transforms.Compose([
27         transforms.ToTensor(),
28         transforms.Normalize(mean, std)
29         ])
30
31 test_transform = transforms.Compose([
32         transforms.ToTensor(),
33         transforms.Normalize(mean, std)
34         ])
35
36
37 train_loader = torch.utils.data.DataLoader(
38         datasets.MNIST('./mnist', train=True, download=True,
39                        transform=train_transform),
        batch_size=BATCH_SIZE, shuffle=True) # train dataset

test_loader = torch.utils.data.DataLoader(
        datasets.MNIST('./mnist', train=False,
                        transform=test_transform),
        batch_size=BATCH_SIZE, shuffle=False) # test dataset

# -------------------------Defining CNN--------------------------
# Pytorch official Example site: https://github.com/pytorch/
    examples/blob/master/mnist/main.py
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output


# defining CNN model
model = CNN()

## Loss function
criterion = torch.nn.CrossEntropyLoss() # pytorch's cross entropy
     loss function

# definin which paramters to train only the ANN model parameters
optimizer = torch.optim.SGD(model.parameters(),learning_rate)

# defining the training function
# Train baseline classifier on clean data
def train(model, optimizer,criterion,epoch):
    model.train() # setting up for training
    for batch_idx, (data, target) in enumerate(train_loader): #
    data contains the image and target contains the label =
    0/1/2/3/4/5/6/7/8/9
```

```
        optimizer.zero_grad() # setting gradient to zero
        output = model(data) # forward
        loss = criterion(output, target) # loss computation
        loss.backward() # back propagation here pytorch will take care of it
        optimizer.step() # updating the weight values
        if batch_idx % 100 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))


# to evaluate the model
## validation of test accuracy
def test(model, criterion, val_loader, epoch):
    model.eval()
    test_loss = 0
    correct = 0

    with torch.no_grad():
        for batch_idx, (data, target) in enumerate(val_loader):

            output = model(data)
            test_loss += criterion(output, target).item() # sum up batch loss
            pred = output.max(1, keepdim=True)[1] # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item() # if pred == target then correct +=1

    test_loss /= len(val_loader.dataset) # average test loss
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.4f}%)\n'.format(
        test_loss, correct, val_loader.sampler.__len__(),
        100. * correct / val_loader.sampler.__len__() ))


## training the ANN
for i in range(Iterations):
    train(model, optimizer,criterion,i)
    test(model, criterion, test_loader, i) #Testing the the current ANN
```

Listing 3.5 CNN Example [14]

Output of Listing 3.5:

```
Train Epoch: 0 [0/60000 (0%)] Loss: 2.345949
Train Epoch: 0 [12800/60000 (21%)] Loss: 0.751740
Train Epoch: 0 [25600/60000 (43%)] Loss: 0.497667
Train Epoch: 0 [38400/60000 (64%)] Loss: 0.405288
Train Epoch: 0 [51200/60000 (85%)] Loss: 0.235314

Test set: Average loss: 0.0013, Accuracy: 9519/10000 (95.1900%)

Train Epoch: 1 [0/60000 (0%)] Loss: 0.320085
Train Epoch: 1 [12800/60000 (21%)] Loss: 0.134681
Train Epoch: 1 [25600/60000 (43%)] Loss: 0.241163
Train Epoch: 1 [38400/60000 (64%)] Loss: 0.128346
Train Epoch: 1 [51200/60000 (85%)] Loss: 0.114656

Test set: Average loss: 0.0007, Accuracy: 9707/10000 (97.0700%)

Train Epoch: 2 [0/60000 (0%)] Loss: 0.185234
Train Epoch: 2 [12800/60000 (21%)] Loss: 0.182811
Train Epoch: 2 [25600/60000 (43%)] Loss: 0.138322
Train Epoch: 2 [38400/60000 (64%)] Loss: 0.165857
Train Epoch: 2 [51200/60000 (85%)] Loss: 0.154645

Test set: Average loss: 0.0005, Accuracy: 9780/10000 (97.8000%)

... ... ...
... ... ...
... ... ...

Train Epoch: 18 [0/60000 (0%)] Loss: 0.050605
Train Epoch: 18 [12800/60000 (21%)] Loss: 0.099548
Train Epoch: 18 [25600/60000 (43%)] Loss: 0.040686
Train Epoch: 18 [38400/60000 (64%)] Loss: 0.040548
Train Epoch: 18 [51200/60000 (85%)] Loss: 0.040939

Test set: Average loss: 0.0002, Accuracy: 9890/10000 (98.9000%)

Train Epoch: 19 [0/60000 (0%)] Loss: 0.020478
Train Epoch: 19 [12800/60000 (21%)] Loss: 0.038522
Train Epoch: 19 [25600/60000 (43%)] Loss: 0.008525
Train Epoch: 19 [38400/60000 (64%)] Loss: 0.057977
Train Epoch: 19 [51200/60000 (85%)] Loss: 0.088406

Test set: Average loss: 0.0002, Accuracy: 9893/10000 (98.9300%)
```

3.5.4.8 Why Is CNN So Popular?

CNN is the most popular deep neural network nowadays for image classification, object recognition, and object tracking tasks. First, the convolution operation can precisely capture an image's semantic information and extract detailed feature information. Second, the CNN architecture is much more efficient. It has a smaller parameter size than a fully connected neural network. Hence, CNN models are ideal for edge and mobile devices with low memory and computing resources.

3.5.4.9 State-of-the-Art Model Architecture

A typical CNN architecture, also known as ConvNet architecture, is characterized by alternate layers of CONV and pooling, followed by one or more FC layers at the end [15]. Sometimes an FC layer is supplanted by a global average pooling layer. Several mapping functions, regulatory units such as BN and dropout, etc. are utilized to optimize the performance of the ConvNet. The arrangement of the CNN component layers is pivotal in designing new architectures aiming for better performances. Most of the computation time and memory space is invested in the early CONV layers, and the majority of the parameters are in the FC layers at the end of the network.

Some commonly used ConvNet architectures are described in this section. A comparison of the architectures is provided in Table 3.18.

1. **LeNet:** The LeNet architecture is one of the first successful applications of CNN. It was developed by Yann LeCun et al. in 1989–1998 [16]. It was used to read zip codes, handwritten digits, etc.
2. **AlexNet:** AlexNet was the first CNN architecture used in computer vision, and it was developed by Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton in 2012 [17]. It is very similar to LeNet except that it is bigger and deeper and involves several CONV layers stacked on top of one another.
3. **GoogLeNet:** The GoogLeNet, also recognized as Inception-V1, is a 22-layer deep CNN developed by Szegedy et al. from Google in 2014 [18]. The inception module used in this architecture significantly decreased the number of parameters in the network to only 4 million compared to the 60 million in AlexNet. The

Table 3.18 Comparison among the CNN architectures [15]

Name of architecture	Parameters	Depth	Test accuracy %
LeNet	0.06 M	5	MNIST 99.05
AlexNet	60 M	8	ImageNet 57
GoogLeNet	4 M	22	ImageNet 93.3
VGGNet	138 M	19	ImageNet 92.7
ResNet	25.6 M	152	ImageNet 96.4

reduction is achieved through the use of average pooling rather than using FC layers at the top of the ConvNet.

4. **VGGNet:** VGGNet was introduced by Karen Simonyan and Andrew Zisserman from the Visual Geometry Group (by Oxford University) in 2014 [19] and is known for its simplicity. It uses only 3×3 CONV layers stacked atop one another in an ascending order of depth. The use of max pooling helps to reduce the volume size. However, this architecture is costly in terms of evaluation, training time, memory consumption, and a number of parameters (140 million).
5. **ResNet:** The Residual Network, or ResNet, was proposed by Kaiming He et al. in 2015 [20]. This network has special skip connections, which is an extensive use of BN. The purpose of the skip connection is to enhance the gradient computation process during backpropagation. Such connection in a deeper network solves the vanishing gradient problem in deep neural network (DNN).

3.5.5 Recurrent Neural Network (RNN)

Convolution neural networks can capture the semantic information of an image, but to adapt to sequential data, we will adopt Recurrent Neural Network (RNN). To classify character-level words as a series of characters, RNN combines input and hidden state output to generate a class label as output for a specific word, as displayed in Fig. 3.25.

The network in Fig. 3.25 has two linear layers that operate on a hidden and an input state. The network operates by taking input and a prior hidden state from the sequence of data. Then it will compute the classification output and a future hidden state. The output layer uses a LogSoftmax function to generate the output probability of the classes. We have demonstrated a practical example of Natural

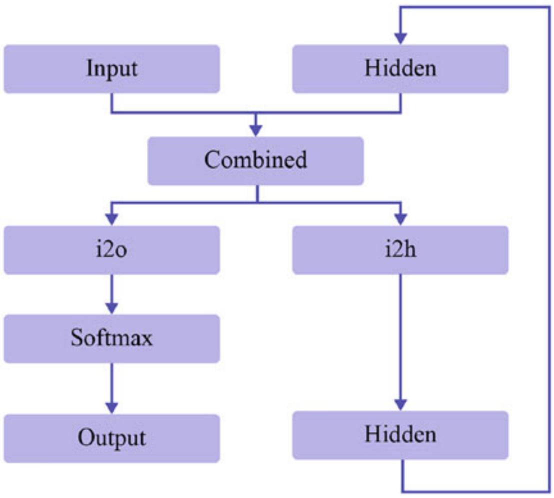


Fig. 3.25 Architecture of an RNN model containing hidden state, combining layer, and output layer

Language Processing using RNN in Sect. 4.9.4 of Chap. 4. In our example, we will classify surnames from 18 different languages using the PyTorch official example of RNN [21].

3.5.6 Generative Adversarial Network (GAN)

Deep adversarial networks have gained recent popularity in generating fake images, videos, and other tasks. It was first proposed by IAN Goodfellow [22]. It consists of two neural networks competing with each other in a min–max game. First, a network known as *generator* outputs fake images from random noise which closely matches an underlying target distribution. The second network is known as *discriminator*, which functions as a decipher between the clean and fake images (Fig. 3.26). Next, we describe the functionality of each GAN module and an overall training scheme:

1. **Discriminator.** The discriminator is trained to differentiate between a clean image from the original distribution and a fake image generated from the generator. During training, it ignores the generator loss function. It is trained based on a binary classification loss function only to identify true/fake images.
2. **Generator.** The generator takes random input and transforms it into a target distribution data instance. The goal is to fool the discriminator into misclassifying this data point as true data. Hence, generator training takes the distribution loss into account.
3. **Overall Training.** The overall training of GAN should alternate between the generator and the discriminator. Typically, the discriminator is trained for a few iterations and then the generator; this process of alternate training keeps repeating until convergence. In the literature, there are different loss functions that can train both generator and discriminator jointly. A typical min–max loss function using the Kullback–Leibler (KL) divergence loss can be formulated as

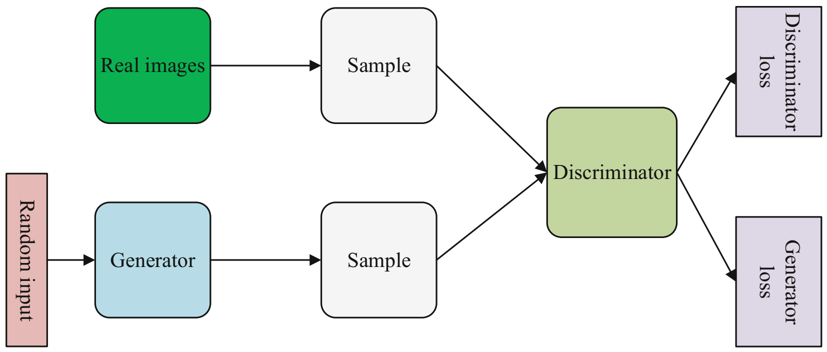


Fig. 3.26 GAN network

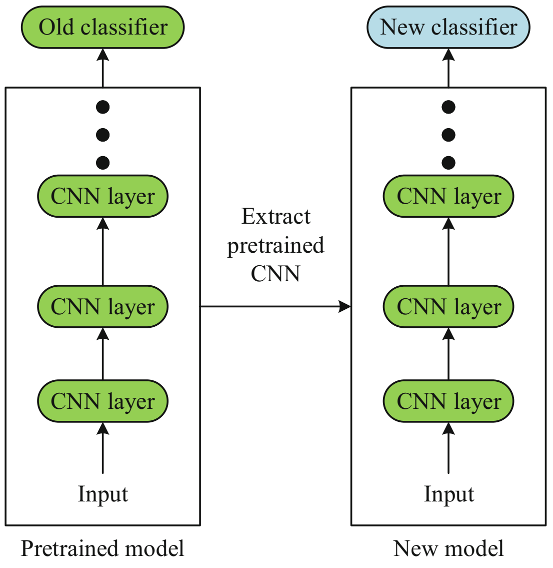


Fig. 3.27 Transfer learning process

$$E_x[\log(D(x))] + E_z[\log(1 - D(G(z)))], \tag{3.46}$$

where $D(x)$ is the discriminator's estimate of the probability that real data instance x is real, $G(z)$ is the generator's output for random noise z, and $D(G(z))$ is the discriminator's estimate of the probability that a fake instance is real. Another popular loss function for GAN training is Wasserstein loss. We would encourage the reader to read some survey GAN papers to get familiar with a wide range of GAN architectures [23]. We will present an example of a GAN algorithm in a robotics application in Chap. 5.

3.5.7 Transfer Learning

Transfer learning refers to transferring one information domain into a different application domain. For example, a neural network is trained on a 10-class classification problem. Next, we want this pre-trained model to adapt to a new knowledge domain (e.g., 100-class classification problem). A common approach to transferring this 10-class classifier model into a 100-class classifier would be to fix only a part of the model to replace the last classification layer, as displayed in Fig. 3.27, because the initial convolution layers capture the high-level features of the given data. Such high-level features are common in general across similar types of datasets. However, the final few layers, especially the last layer, perform the classification task on the extracted features. Hence, a common strategy in transfer learning applications would be to fix the first few layers of the pre-trained model and then only train the last classification layer after replacing it with new class configurations. We will demonstrate an example of multi-domain transfer learning in Chap. 7.

3.6 Time Series Forecasting

The statistical or machine learning technique called time series forecasting aims to model past time series data in order to predict future time points. A time series is a collection of observations made repeatedly over time, whether daily, weekly, monthly, or annually. Time series analysis entails creating models to characterize the observed time series and comprehend the "why" underlying its dataset. This involves making predictions and interpretations based on the available data. Time series forecasting uses the best-fitting model to predict future observations based on the intricate processing of current and historical data. In this section, we will study three algorithms for time series forecasting: ARIMA, SARIMA, and LSTM.

3.6.1 ARIMA

Auto-Regressive Integrated Moving Average (ARIMA) is a class of models that defines different characteristics of a time series depending on the previous behavior of that time series. The term ARIMA is made of initials of three different terms, which are:

- *Auto-regression (AR):* indicating a model that uses the relationship between current and past values. It is represented by p.
- *Integrated (I):* indicating the data is either stationary or made stationary by differencing. It is represented by d.
- *Moving Average (MA):* indicating a model which linearly depends on the forecast of that model, and also the errors are linear functions of past errors. It is represented by q.

Together, these three terms are represented by the p, d, and q parameters. These parameters are represented by integer values in the model to better understand which model is being used. For example, if we use the first order for auto-regression, difference a time series twice to make it stationary, and use the third order for moving average, then we can say that we are using the ARIMA(1, 2, 3) model.

Before going into the method, we will study a few terms that will help us understand the concepts:

1. **Stationary Time Series:** A time series whose statistical properties, such as mean, standard deviation, variance, and covariance, do not vary with time is known as a stationary time series. A time series is stationary if it fulfills three conditions—constant mean with time, constant variance with time, and constant auto-correlation with time.

2. **White noise:** A time series is a mix of signals and noise. Mathematically,

$$\text{Time series}(t) = \text{Signal}(t) + \text{Noise}(t). \tag{3.47}$$

When we fit a model to the series, we can predict the signal portion of the time series but not the noise portion. A signal is known as a white noise if its mean is zero and has a constant standard deviation with time, and the correlation between lags is zero. If a time series is truly white noise, it maintains all these three conditions of being white noise. So, it is unpredictable, and we should stop trying to fit any model into it.

3. **Lag or Backshift Operator:** The lag (L) or the backshift operator (B) operates on an element of a time series and produces the previous element. In a time series, if a specific point in time is t, then x_{t-j} is called the jth lag of x_t.

$$B^j x_t \equiv x_{t-j}. \tag{3.48}$$

$$\nabla xt = x_t - x_{t-1} = x_t - Bx_t = (1 - B)x_t. \tag{3.49}$$

3.6.1.1 The Auto-regressive Process

In the auto-regressive process, the future data points are forecasted based on the past values in the series. Starting our forecast with respect to the past 1 value, which is AR(1). Here the past value is Y_{t-1}. We can now see that Y_{t-1} depends on Y_{t-2}, and so on.

$$\begin{aligned}
Y_t &= c + \phi Y_{t-1} + e_t, \\
Y_t &= c + \phi^2 Y_{t-2} + \phi Y_{t-1} + e_t, \\
&\vdots \\
Y_t &= \frac{c}{1-\phi} + \phi^t Y_1 + \phi^{t-1} e_2 + \phi^{t-2} e_3 + \cdots + e_t.
\end{aligned}$$

The first observation term ($\phi^t Y_1$) in the last equation signifies that the first observation still matters in any future forecast, even if it is very small. We use stationary conditions to minimize this effect, which is $|\phi| < 1$.

A time series that is a linear function of p past values, i.e., AR(p) is given by

$$Y_t = c + \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \cdots + \phi_p Y_{t-p} + e_t. \tag{3.50}$$

3.6.1.2 The Moving Average Process

In the moving average process, the future values are forecasted based on past errors. Similar to the auto-regressive process, we can write

$$Y_t = c + \theta e_t - 1 + r_t. \tag{3.51}$$

A time series that is a linear function of q past errors, MA(q), is given by

$$Y_t = c + e_t + \theta_1 e_{t-1} + \theta_2 e_{t-2} + \cdots + \theta_q e_{t-q}. \tag{3.52}$$

3.6.1.3 The Differencing Process

Combining auto-regression (AR) and moving average (MA) with differencing (I), we will get the Auto-Regression Integrated Moving Average (ARIMA) process [24, 25].

$$Y'_t = c + \phi_1 Y'_{t-1} + \cdots + \phi_p Y'_{t-p} + \theta_1 e_{t-1} + \cdots + \theta_q e_{t-q} + e_t, \tag{3.53}$$

where Y_t is the differenced series, and we needed to difference only once to make the time series stationary. But at times, the time series might have to be differenced more than once.

The above equation can be expressed with the backshifting operator B for easier calculation:

$$\underset{\substack{\uparrow \\ \text{AR}(p)}}{(1 - \phi_1 B - \cdots - \phi_p B^p)} \quad \underset{\substack{\uparrow \\ \text{Degree of} \\ \text{differencing}(d)}}{(1 - B)^d Y_t} \quad = c + \underset{\substack{\uparrow \\ \text{MA}(q)}}{(1 + \theta_1 B + \cdots + \theta_q B^q)e_t} \quad .$$

3.6.1.4 Determining the Order

With the help of the autocorrelation plot (ACF) and partial autocorrelation plot (PACF), the order of a model can be determined. The ACF plot basically visualizes the relationship between y_t and y_{t-q}. The PACF is quite similar to ACF, showing the relationship between two time series datapoints, y_t and y_{t-p}, but intervening y_t removed. Often, the Akaike information criterion (AIC) and the Bayesian information criterion (BIC) are also used to determine the best order for a model.

There are a few statistical tests to check the stationarity of a time series. If a time series is stationary, then $d = 0$. If the series needs to be differenced n times, then $d = n$. One of the most used statistical tests is the Augmented Dickey–Fuller (ADF) test. It is a type of unit root test, indicating a time series being stationary. The ADF test can calculate the p-value, which is the probability of achieving a result equal to or more extreme than the actual observed under the assumption that there is no relationship between the two sets of data here (null hypothesis). The p-value can indicate whether the time series is stationary or not.

- p-value $> 0.05 \rightarrow$ non-stationary
- p-value $\leq 0.05 \rightarrow$ stationary.

Programming Example 3.7
Listing 3.6 is basically for the purpose of determining the model order of ARIMA and Table 3.19 explains the listing. After reading the CSV data, the data is plotted to see what the dataset actually looks like (Fig. 3.28). Then, the dataset is checked using the ADF test to determine whether the data is stationary. Based on the *p*-value, we decide that these data are stationary. The `auto_arima` function is used to find the best model for ARIMA. Here, `auto_arima` uses the ADF test to fit a model, determine the error, and choose the model with the least error.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller
from pmdarima import auto_arima
import warnings
warnings.filterwarnings("ignore")


# -------------------------Reading Data-------------------------
df = pd.read_csv('./data/MaunaLoaDailyTemps.csv',
                 index_col='DATE',
                 parse_dates=True).dropna()


# -------------------------Initial Plot-------------------------
df['AvgTemp'].plot(figsize=(12,6))
plt.show()


# ----------------------Check If Stationary----------------------
dftest = adfuller(df['AvgTemp'], autolag='AIC')
print('ADF Value: {0:.2f} \tP-Value: {1:.2f} \nNo of Lags: {2} \t
    \tNo Of Observations: {3} \nCritical Values:'.format(
    dftest[0], dftest[1], dftest[2], dftest[3]))

for i, values in dftest[4].items():
  print("\t\t\t\t",i, " :", values)


# -------------------Finding Best Model Order-------------------
arima_model = auto_arima(df['AvgTemp'], error_action="ignore",
                          stepwise=True, test='adf',
                          suppress_warnings=True)

print(arima_model.summary())
```

Listing 3.6 Determination of ARIMA model order

Output of Listing 3.6:

```
ADF Value: -6.55    P-Value: 0.00
No of Lags: 12      No Of Observations: 1808
Critical Values:
 1%   : -3.433972018026501
 5%   : -2.8631399192826676
 10%  : -2.5676217442756872


                          SARIMAX Results
=============================================================
Dep. Variable                y   No. Observations        1821
Model       SARIMAX(1, 0, 5)   Log Likelihood      -4139.680
Date        Tue, 10 Aug 2021   AIC                  8295.360
Time                  192246   BIC                  8339.417
Sample                     0   HQIC                 8311.613
Covariance Type          opg
=============================================================
             coef   std err        z       Pz    [0.025   0.975]
-------------------------------------------------------------
intercept  1.2241    0.367    3.338   0.001     0.505    1.943
ar.L1      0.9736    0.008  123.206   0.000     0.958    0.989
ma.L1     -0.1327    0.024   -5.573   0.000    -0.179   -0.086
ma.L2     -0.2113    0.024   -8.696   0.000    -0.259   -0.164
ma.L3     -0.2159    0.024   -9.010   0.000    -0.263   -0.169
ma.L4     -0.1352    0.023   -5.888   0.000    -0.180   -0.090
ma.L5     -0.0506    0.024   -2.065   0.039    -0.099   -0.003
sigma2     5.5306    0.174   31.778   0.000     5.189    5.872
=============================================================
Ljung-Box (L1) (Q)         0.08   Jarque-Bera (JB)        20.30
Prob(Q)                    0.77   Prob(JB)                 0.00
Heteroskedasticity (H)     0.81   Skew                    -0.17
Prob(H) (two-sided)        0.01   Kurtosis                 3.39
============================================================
```

Programming Example 3.8

Listing 3.7 is the program for fitting data and predicting output values and Table 3.20 explains the listing. First, the data is read in CSV form. The train and test data are separated into two different arrays. The model with the least error is determined in Listing 3.6. So, the order of this model will be ARIMA(1,0,5).

A model is trained with these data, which predicts in the range of test data. These are compared and RMS error is calculated. The forecasted data and actual data are plotted side by side in Fig. 3.29. In this figure, the ARIMA model predicts the trend

Table 3.19 Explanation of the ARIMA model selection code example presented in Listing 3.6

Line number	Description
1–7	Importing some required modules
9–10	Reading data using read_csv function of Pandas module
17–18	Plotting the dataset with respect to date index
22–27	Printing the five outputs of Augmented Dickey–Fuller test
30–32	Search for the best model for this particular dataset using pmdarima module
34	Printing the best model and its summary

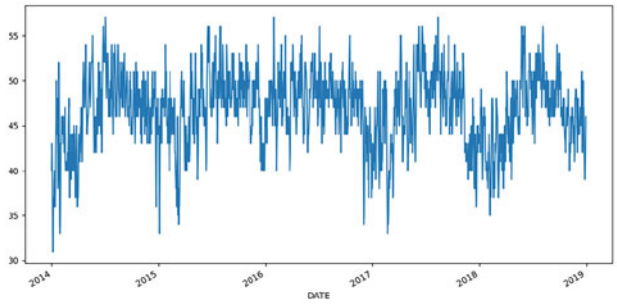


Fig. 3.28 Initial plot of the dataset used

component of the data. That is why the predicted data is a smooth line and the actual data have a lot of deviation. Figure 3.33 is a great example of different components of a dataset.

Another model is trained (again using ARIMA(1,0,5) order) with the whole dataset and then forecasted similarly. The predicted result is reflected in Fig. 3.30.

```python
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA
from math import sqrt
from sklearn.metrics import mean_squared_error


# -------------------------Reading Data-------------------------
df=pd.read_csv('./data/MaunaLoaDailyTemps.csv',index_col='DATE',
    parse_dates=True)
df=df['AvgTemp'].dropna()


# ----------------------Splitting Dataset----------------------
size = int(len(df)*0.80)
train, test = df[0:size], df[size:len(df)]


# -------------------Training with 80% Data---------------------
model=ARIMA(train,order=(1,0,5))
model_fit=model.fit()
model_fit.summary()


# ------------Prediction & Setting Index for Plot---------------
rng = pd.date_range(start='2017-12-30',end='2018-12-29')
prediction=model_fit.predict(start=len(train), end=len(train)+len
    (test)-1, typ='levels')
prediction.index = rng
```

```


# ------------------------Ploting Data-------------------------
plt.figure(figsize=(12,6))
prediction.plot(label="Prediction", legend=True)
test.plot(label="Test Data", legend=True)
plt.show()


# -------------------------Evaluation--------------------------
mean=test.mean()
print('Mean:',mean)
rmse = sqrt(mean_squared_error(test, prediction))
print('Test RMSE: %.3f' % rmse)


# -------------------Training with Full Data---------------------
full_model = ARIMA(df,order=(1,0,5))
full_model_fit = full_model.fit()


# ------------------Prediction & Setting index------------------
rng2 = pd.date_range(start='2018-12-30', end='2019-02-28', freq='
    D')
full_prediction = full_model_fit.predict(start=len(df),end=len(df
    )+2*30,typ='levels').rename('ARIMA Predictions')
full_prediction.index = rng2


# --------------------Plotting with Main Data--------------------
df.plot(figsize=(12,6), label="Full data", legend=True)
full_prediction.plot(figsize=(12,6), label="Full prediction",
    legend=True)

plt.show()
```

Listing 3.7 ARIMA Model Implementation

Output of Listing 3.7:

```
Mean: 46.3041095890411
Test RMSE: 3.787
```

3.6.2 Seasonal ARIMA

Although ARIMA is widely used for forecasting univariate time series data, it has a limitation. It cannot forecast data that has a seasonal component. Seasonal ARIMA, or SARIMA, is an extension of ARIMA that can handle data with seasonality.

In a time series, the property of seasonality makes the time series predictable over a certain period of time. However, seasonality is different from the cyclic property. Seasonality is prevalent over a fixed period of time; that is, it has a fixed

Table 3.20 Explanation of the ARIMA code example presented in Listing 3.7

Line number	Description
1–5	Importing Pandas, Matplotlib, ARIMA, and some required modules
9–10	Reading data using read_csv function of Pandas module
14–15	Splitting the dataset into train and test datasets
19–21	Fitting the train dataset with order (1.0,5) and also getting model fit summary
26	Making predictions
25–27	Setting index for newly predicted dataset
31–34	Plotting with test dataset to compare them both
38–39	Finding mean of test data and output it
40–41	Finding Root Mean Square Error
45–46	Fitting the whole dataset
50–52	Getting predictions and setting index to date values for proper plotting
56–58	Plotting the prediction with actual data
54–60	Search for the best model for this particular dataset using pmdarima module
62	Printing the best model and its summary

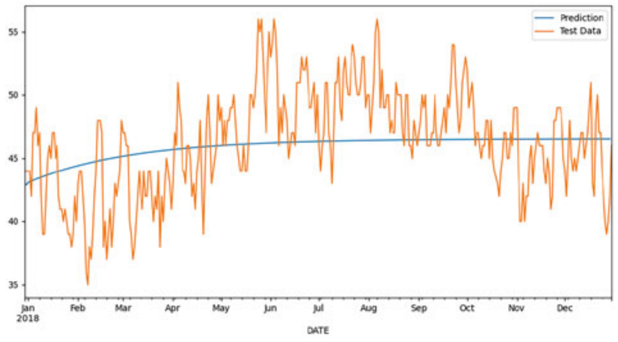


Fig. 3.29 Test data forecast using ARIMA model

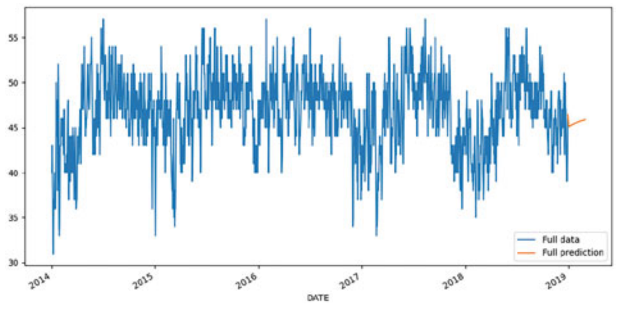


Fig. 3.30 Out of sample data forecast using ARIMA model

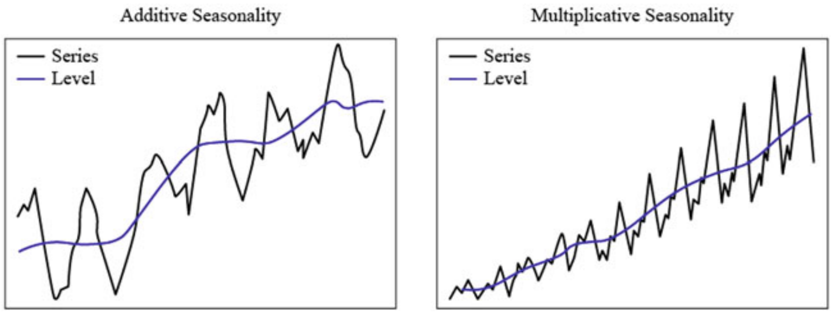


Fig. 3.31 Additive and multiplicative seasonality in a time series dataset

frequency. On the other hand, if the period of time fluctuates and does not have a fixed frequency, it is termed a cycle.

A time series is a combination of signal and noise, and a signal constitutes different components, such as trends ($T(t)$), seasonality ($S(t)$), and residuals ($E(t)$). It is not mandatory for all series to have trends, seasonality, and residuals. But a series can have any combination of these three components. The three components can be decomposed from the signal using the decomposition process. As shown in Fig. 3.31, there are two approaches to decomposing the components of a time series:

1. Additive approach: $Y(t) = T(t) + S(t) + E(t)$.
2. Multiplicative approach: $Y(t) = T(t) \times S(t) \times E(t)$.

In addition to the ARIMA terms (p, d, q), four other terms are required to express an order of the SARIMA model. These are the seasonal AR (P), seasonal difference order (D), seasonal MA (Q), and the length of season (m). So, the total order of a SARIMA model is written as follows [26]:

$$\text{SARIMA}(p, d, q)(P, D, Q)_m. \tag{3.54}$$

The SARIMA model is a bit complex to express directly, so we use the backshift operator to describe it easily. Three examples are shown below:

1. SARIMA$(1, 1, 1)(1, 1, 1)_4$ is expressed (without constant) as

$$\underset{\text{AR}(1)}{\underset{\uparrow}{(1-\phi_1 B)}}\ \underset{\text{AR}(1)_4}{\underset{\uparrow}{(1-\Phi_1 B^4)}}\ \underset{d=1}{\underset{\uparrow}{(1-B)}}\ \underset{D=1}{\underset{\uparrow}{(1-B^4)}}y_t = \underset{\text{MA}(1)}{\underset{\uparrow}{(1+\theta_1 B)}}\ \underset{\text{MA}(1)_4}{\underset{\uparrow}{(1+\Theta_1 B^4)}}e_t.$$

2. SARIMA$(1, 0, 4)(2, 0, 2)_{12}$ is expressed as

$$\underset{\text{AR}(1)}{\underset{\uparrow}{(1-\phi_1 B)}} \qquad \underset{\text{AR}(2)_{12}}{\underset{\uparrow}{(1-\Phi_1 B^{12}-\Phi_2 B^{24})}}y_t$$

$$= \underset{constant}{\underset{\uparrow}{\theta_0}} + \underset{\text{MA}(4)}{\underset{\uparrow}{(1-\theta_1 B-\theta_2 B^2-\cdots-\theta_4 B^4)}}\ \underset{\text{MA}(2)_{12}}{\underset{\uparrow}{(1-\Theta_1 B^{12}-\Theta_2 B^{24})}}e_t.$$

3. SARIMA$(0, 0, 1)(2, 1, 2)_{12}$ is expressed (without constant) as

$$\underset{\text{AR}(2)_{12}}{\underset{\uparrow}{(1-\Phi_1 B^{12}-\Phi_2 B^{24})}}\ \underset{D=1}{\underset{\uparrow}{(1-B^{12})}}y_t = \underset{\text{MA}(1)}{\underset{\uparrow}{(1+\theta_1 B)}}\ \underset{\text{MA}(2)_{12}}{\underset{\uparrow}{(1+\Theta_1 B^{12}+\Theta_2 B^{24})}}.$$

You may refer to the references [27, 28] for further reading.

Programming Example 3.9
Listing 3.8 is almost similar to Listing 3.6 and Table 3.21 explains the listing. This determines the best order we can use for the SARIMA model. The data used in this case is non-stationary data. The dataset is monthly passenger data for an airplane company. The objective is to predict the future data using the past data successfully. The data is first plotted to get a brief idea about the data (Fig. 3.32). Then ADF test checks whether the data is stationary. Seasonal decomposition is done to separate different components (Fig. 3.33). Then, different models are grid-searched to find the best one (one for the lowest error).

```
#Dataset: https://www.kaggle.com/rakannimer/air-passengers
import matplotlib.pyplot as plt
import pandas as pd
from statsmodels.tsa.stattools import adfuller
from pmdarima.arima import auto_arima
from statsmodels.tsa.seasonal import seasonal_decompose as sd

# -------------------------Reading Data--------------------------
passenger_data =pd.read_csv("./data/AirPassengers.csv")
passenger_data.Month = pd.to_datetime(passenger_data.Month)
passenger_data = passenger_data.set_index("Month")
plt.plot(passenger_data.Passengers)
plt.xlabel("Year")
plt.ylabel("No of Passengers")
plt.show()


# ----------------------Check If Stationary----------------------
dftest = adfuller(passenger_data, autolag='AIC')
print('ADF Value: {0:.2f} \tP-Value: {1:.2f} \nNo of Lags: {2} \t
    \tNo Of Observations: {3} \nCritical Values:'.format(
    dftest[0], dftest[1], dftest[2], dftest[3]))

for i, values in dftest[4].items():
  print("\t\t\t\t",i, " :", values)


# ---------------------Seasonal Decomposition--------------------
comp = []
comp.append(passenger_data["Passengers"])

#Seasonal Decomposition
components = sd(passenger_data["Passengers"], model='additive')

trend_component = components.trend
comp.append(trend_component)

seasonal_component = components.seasonal
comp.append(seasonal_component)

residual_component = components.resid
comp.append(residual_component)


comp_names = ["Original", "Trend", "Seasonal", "Residual"]

plt.figure(figsize=(12, 6))
for i in range(4):
    plt.subplot(411 + i)
    plt.plot(comp[i], label=comp_names[i], color='red')
    plt.legend(loc=2)
plt.show()


# -------------------Finding Best Model Order--------------------
sarima_model=auto_arima(passenger_data["Passengers"],start_p=1,d
    =1,start_q=1,
                      max_p=5,max_q=5,m=12,
                      start_P=0,D=1,start_Q=0,max_P=5,max_D=5,
    max_Q=5,
                      seasonal=True,
                      error_action="ignore",
                      suppress_warnings=True,
                      stepwise=True,n_fits=50,test='adf')

print(sarima_model.summary())
```

Listing 3.8 Determination of SARIMA model order [29, 30]

Output of Listing 3.8:

```
ADF Value:  0.81    P-Value:  0.99
Num Of Lags:  13    Num Of Observations: 130
Critical Values:
             1% :  -3.4816817173418295
```

```
            5% :  -2.8840418343195267
            10% :  -2.578770059171598


                               SARIMAX Results
   ==============================================================
   Dep. Variable:                     y  No. Observations:   144
   Model:     SARIMAX(0,1,1)x(2,1,[],12)  Log Likelihood -505.589
   Date:                Tue, 10 Aug 2021  AIC            1019.178
   Time:                        20:01:12  BIC            1030.679
   Sample:                             0  HQIC           1023.851
   Covariance Type:                  opg
   ==============================================================
               coef    std err         z    P>|z|    [0.025    0.975]
   --------------------------------------------------------------
   ma.L1     -0.3634     0.074    -4.945    0.000    -0.508    -0.219
   ar.S.L12  -0.1239     0.090    -1.372    0.170    -0.301     0.053
   ar.S.L24   0.1911     0.107     1.783    0.075    -0.019     0.401
   sigma2   130.4480    15.527     8.402    0.000   100.016    60.880
   ==============================================================
   Ljung-Box (L1) (Q):          0.01   Jarque-Bera (JB):         59
   Prob(Q):                     0.92   Prob(JB):               0.10
   Heteroskedasticity (H):      2.70   Skew:                   0.15
   Prob(H) (two-sided):         0.00   Kurtosis:               3.87
   ==============================================================
```

Table 3.21 Explanation of the SARIMA model selection code example presented in Listing 3.8

Line number	Description
1–6	Importing Pandas, Matplotlib, ARIMA, and pmdarima modules
8–11	Reading data using read_csv function of Pandas module and setting dates as index
12–15	Plotting the dataset
8–24	Performing the Dickey–Fuller test
32–41	Decomposing into three different components considering additive model
44–51	Plotting different components of the dataset with the original dataset
55–61	Search for the best model for this particular dataset using pmdarima module
63	Printing the best model and its summary

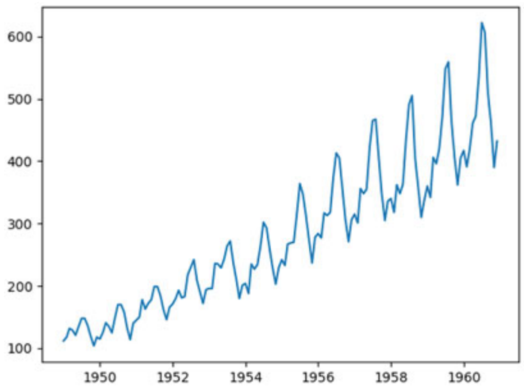


Fig. 3.32 Initial plot of the dataset used for the SARIMA model

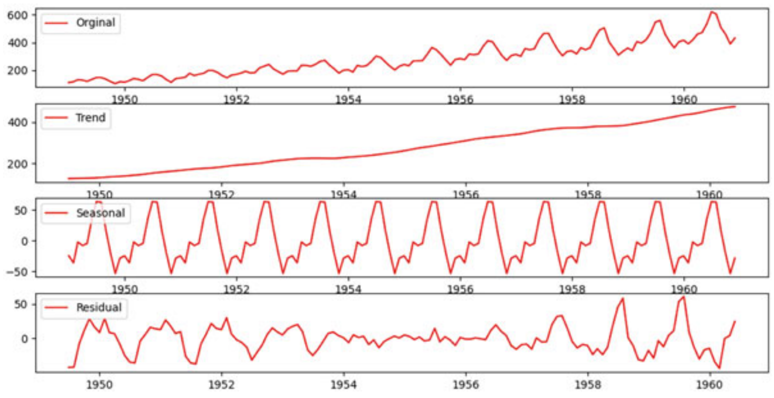


Fig. 3.33 Seasonal decomposition of the data

Programming Example 3.10
Listing 3.9 (explained in Table 3.22) first reads all the CSV data. Data are then split into train and test data. The forecast is done based on train data in the range of test data and compared with the test data. The comparison is shown in Fig. 3.34. The forecasted value is evaluated using the mean value and the RMSE value. The full data is trained again to forecast out of the sample data. The forecasted values are plotted in Fig. 3.35.

```python
#Dataset: https://www.kaggle.com/rakannimer/air-passengers
import matplotlib.pyplot as plt
import pandas as pd
from statsmodels.tsa.statespace.sarimax import SARIMAX
from math import sqrt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split


# -------------------------Reading Data-------------------------
passenger_data =pd.read_csv("./data/AirPassengers.csv")
passenger_data.Month = pd.to_datetime(passenger_data.Month)
passenger_data = passenger_data.set_index("Month")


# -----------------------Splitting Dataset-----------------------
X_train, X_test = train_test_split(passenger_data, test_size=0.2,
                                random_state=42, shuffle=False)


# --------------------Training with 80% Data---------------------
SARIMA=SARIMAX(X_train["Passengers"],
             order=(0,1,1),
             seasonal_order=(2,1,1,12))
SARIMA_fit=SARIMA.fit()
```

```python


# ------------Prediction & Plotting with Test Data---------------
trained_results=SARIMA_fit.predict(len(X_train),len(
    passenger_data)-1)
trained_results.plot(legend=True)
X_test["Passengers"].plot(legend=True)
plt.show()


# --------------------------Evaluation---------------------------
mean=X_test["Passengers"].mean()
print('Mean:',mean)
rmse = sqrt(mean_squared_error(X_test, trained_results))
print('Test RMSE: %.3f' % rmse)


# -------------------Training with Full Data---------------------
year_to_forecast = 5
model_full=SARIMAX(X_train["Passengers"],
                order=(0,1,1),
                seasonal_order=(2,1,1,12))
model_fit_full=model_full.fit()


# -----------------Predicting out-of-sample data
    ------------------
forecast=model_fit_full.predict(start=len(passenger_data),
                      end=(len(passenger_data)-1)+
    year_to_forecast*12,
                      typ="levels")


# ------------------Plotting with Original Data
    --------------------
plt.figure(figsize=(12, 6))

data_sets = [(X_train, 'Training Data', 'green'),
             (X_test, 'Test Data', 'blue'),
             (trained_results, 'In-sample Forecast', 'black'),
             (forecast, 'Out-of-sample Forecast', 'red')]

for data, label, color in data_sets:
    plt.plot(data.index, data, label=label, color=color)

plt.legend(loc=2)
plt.xlabel("Time")
plt.ylabel("Passenger Count")
plt.title("Seasonal Forecasting with SARIMA")
plt.grid(True)
plt.show()
```

Listing 3.9 SARIMA Model Implementation

Table 3.22 Explanation of the SARIMA code example presented in Listing 3.9

Line number	Description
1–7	Importing Pandas, Matplotlib, SARIMAX, and some required modules
11–13	Reading data using read_csv function of Pandas module and setting date as index
17–18	Splitting the dataset into train and test datasets
22–25	Fitting the train dataset with order (0,1,1)(2,1,1,12)
29	Making predictions
30–32	Plotting with test dataset to compare them both
36–37	Finding mean of test data and output it
38–39	Finding Root Mean Square Error
43–47	Fitting the whole dataset
51–53	Getting predictions for the whole dataset
57–72	Plotting the prediction with actual data

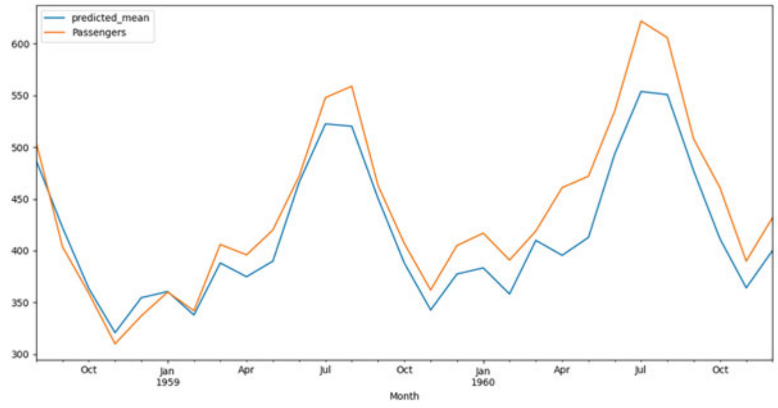


Fig. 3.34 Test data forecast using the SARIMA model

Output of Listing 3.9:

```
Mean: 440.3103448275862
Test RMSE: 32.712
```

3.6.3 Long Short-Term Memory (LSTM)

Proposed by Horchreiter et al. in 1996 [31], long short-term memory (LSTM) is an artificial recurrent neural network (RNN) architecture. It is used in deep learning to predict output from time series or sequential data. As the name suggests, LSTM has a memory capability to store information about long-term dependency. Similar to RNN, it can be one-directional or bidirectional, and a stack of LSTM can be used to capture features from more complex data. LSTM is widely used in speech

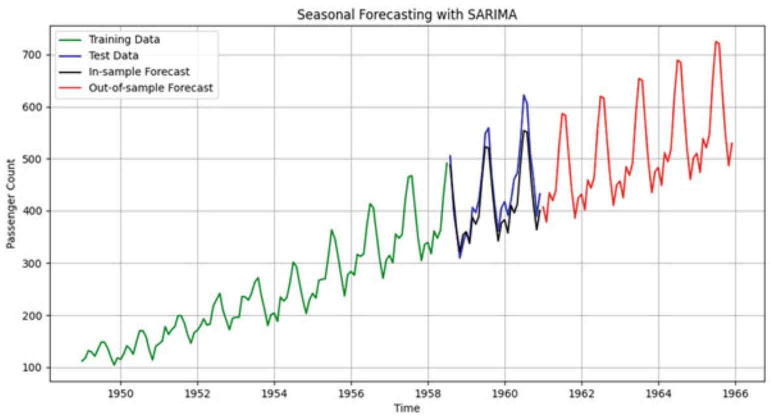


Fig. 3.35 Out of sample data forecast using SARIMA model

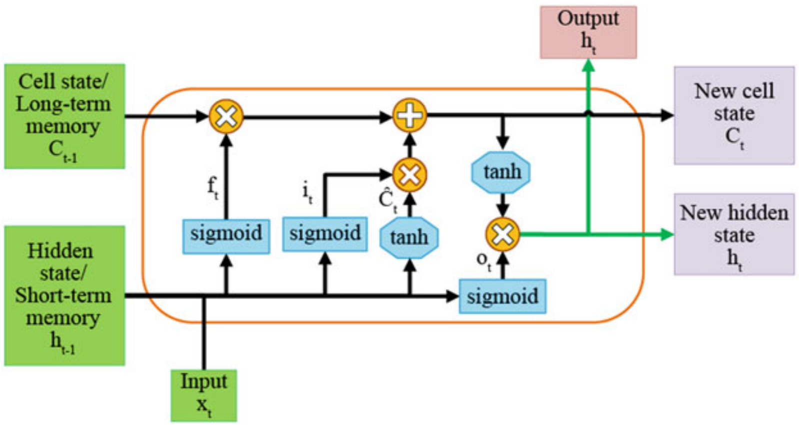


Fig. 3.36 An LSTM model consists of input, hidden state, cell state memory, and activation functions (e.g., sigmoid and tanh)

recognition, music generation, sentiment analysis, translation, image captioning, handwriting recognition, and many other fields.

Unlike a simple RNN, the core idea in LSTM is to use gates in order to control the flow of information from one LSTM unit to another using cell state. Using this cell state (C_{t-1} in Fig. 3.36), the information can run through an entire chain of the network without vanishing. Thus, long-term dependency is achieved by solving the short-term problem of RNN. Depending on the previous unit's output (h_{t-1}) and input (x_t), the sigmoid layer called the *forget gate layer* outputs a number f_t, which is between 0 and 1, and this controls the amount of information that will be eliminated from the cell state. Then new information is stored in the cell state using the *input gate layer* that is also a sigmoid layer, and the new value ($\hat{C}_t$) is determined using the tanh layer from the combined or concatenated form of (x_t and h_{t-1}). Then this new information is multiplied with i_t and added to the cell state, and now this new cell state (C_t) is ready to go to the next unit.

In the case of output (h_t), it is determined how much of the cell state will be passed. So using the sigmoid layer, from x_t and h_{t-1}, o_t is determined. Then, using tanh, the new cell state(C_t) is mapped between -1 and 1, which is finally used to determine the output. The copied version of the output is also passed to the next unit as the new hidden state. Finally, the operation of one LSTM unit is completed. The equation governing the operation of every block of one LSTM unit has been given from Eqs. 3.55 to 3.60 [31].

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f). \tag{3.55}$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i). \tag{3.56}$$

$$\hat{C} = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C). \tag{3.57}$$

$$C_t = f_t * C_{t-1} + i_t * \hat{C}. \tag{3.58}$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o). \tag{3.59}$$

$$h_t = o_t * \tanh(C_t). \tag{3.60}$$

Here, W_f, W_i, W_c, and W_o are weights and b_f, b_i, b_c, and b_o are biases of their corresponding layers. We have already seen that we deal with numbers to predict from the LSTM or any other ML model. But in the field of natural language processing (NLP), we deal with sentences composed of words. So in the case of sentiment analysis or any task in the field of NLP, we need a method to somehow convert these sentences to numbers, which is done by *tokenizing*. Later by embedding, the token or a number is converted into a vector. Vector represents the word in such a way that in the vector space, it expresses the meaning, context, as well as the semantics of that word. So words with similar context or meaning will be put close in the vector space, such as the words "dog" and "cat." It should be noted that every vector must be of the same length. This vector representation is called *word embedding*. We will not go into the details of word embedding; rather, we will use the *embedding* layer of the TensorFlow module.

Programming Example 3.11

In this example, we will use one-directional many-to-one LSTM for sentiment analysis using the IMDB movie review dataset. IMDB dataset has 50,000 movie reviews for the NLP task. Every positive and negative review has positive and negative numbers accordingly. So, if the model outputs a positive number, the review is positive and vice versa. As many mathematical operations are involved in implementing the LSTM network from scratch and optimization is also required, we will use the LSTM layer of the Keras library, which is included in the TensorFlow module. It will take care of these critical mathematical operations.

The code is provided in Listing 3.10 with its explanation in Table 3.23. Also, a complete diagram of the overall task has been given in Fig. 3.37. The code embodies a sentiment analysis procedure using an LSTM model implementation. The IMDB review dataset from the TensorFlow dataset is used here. After preprocessing, the `TextVectorization` layer converts the provided dataset into a numerical format. A class, `myCallback`, is defined to stop training after reaching 87% accuracy. The model utilizes a binary cross-entropy loss function and an Adam optimizer. The process involves training the model with the training dataset, followed by an evaluation of its performance using the test dataset (Fig. 3.38). For further understanding, refer to the following references [32, 33].

```
#Source: https://www.tensorflow.org/text/tutorials/
    text_classification_rnn
import numpy as np
import tensorflow as tf
import tensorflow_datasets as tfds
from tensorflow.keras import preprocessing
from tensorflow.keras.layers.experimental.preprocessing import
    TextVectorization
from tensorflow.keras.layers import Embedding, LSTM, Dense,
    Dropout
import matplotlib.pyplot as plt


# --------------------Load IMDB Review Dataset-------------------
imdb_dataset = tfds.load('imdb_reviews', as_supervised=True)
train_ds, test_ds = imdb_dataset['train'], imdb_dataset['test']
     # Split train and test data

# Declare buffer size to avoid overlap in data processing
BUFFER_SIZE = 100000
BATCH_SIZE = 64
# Shuffle the data and confugure for performance
autotune = tf.data.AUTOTUNE
train_ds = train_ds.shuffle(BUFFER_SIZE).batch(BATCH_SIZE).
    prefetch(autotune)
test_ds = test_ds.shuffle(BUFFER_SIZE).batch(BATCH_SIZE).prefetch
    (autotune)

# Declare vocabulary size
Vocabulary_size = 1000
# Vectorization to convert text into corresponding number
vectorization = TextVectorization(max_tokens=Vocabulary_size)
# Extract only text from train data
train_text = train_ds.map(lambda text, labels: text)
# Map text to number using vectorization
vectorization.adapt(train_text)
```

```

# Creating callback to stop after 87% accuracy of the model
THRESHOLD = 0.87


class myCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={}):
        if(logs.get('accuracy') > THRESHOLD):
            print(f"Reached {THRESHOLD*100}% accuracy")
            self.model.stop_training = True


# --------------------Define the LSTM model----------------------
model = tf.keras.Sequential([
                            vectorization,
                            Embedding(input_dim=len(vectorization.get_vocabulary()),
                                      output_dim=32, mask_zero=True),
                            LSTM(32),
                            Dropout(0.2),
                            Dense(32, activation=tf.nn.relu),
                            Dense(1)
                            ])

# Compile defined model
model.compile(loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
                optimizer=tf.keras.optimizers.Adam(1e-4),metrics=['accuracy'])

# Fit the model on training data
history = model.fit(train_ds, epochs=10, callbacks=[myCallback()])

model.summary() # Summary of the model


# ---------Plot Loss and Accuracy with respect to Epochs---------
fontsize = 20
linewidth = 3
plt.figure(figsize=(8, 8))
plt.plot(history.history['accuracy'], color="green", linewidth=linewidth)
plt.plot(history.history['loss'], color="red", linewidth=linewidth)
plt.xlabel("Epochs", fontsize=fontsize)
plt.ylabel("Accuracy, Loss", fontsize=fontsize)
plt.legend(["Accuracy", "Loss"], fontsize=fontsize)
plt.ylim(0, 1)
plt.grid()
plt.show()

```

```python
# Evaluate loss and accuracy over test dataset
test_loss, test_acc = model.evaluate(test_ds, verbose=0)
print(f"Test loss:{test_loss}, Test accuracy:{test_acc}")


# -------------------Function for Prediction---------------------
def predict(text):
    predictions = model.predict(np.array([text]))
    if predictions >= 0:
        print("Positive review!!")
    else:
        print("Negative review!!")

# Predict sentiment for given review
text = """Completely time waste!! Don't waste your time, rather
    sleeping is better"""
predict(text)
text = """I love beautiful movies. If a film is eye-candy with
    carefully designed decorations,
          masterful camerawork, lighting, and architectural
    frames, I can forgive anything else in """
predict(text)
```

Listing 3.10 Sentiment analysis using LSTM [34]

Output after line 59 of Listing 3.10:

```
Epoch 1/10
391/391 [==============================] - 62s 131ms/step -
loss: 0.6887 - accuracy: 0.5032
Epoch 2/10
391/391 [==============================] - 52s 131ms/step -
loss: 0.4939 - accuracy: 0.7617
Epoch 3/10
391/391 [==============================] - 52s 130ms/step -
loss: 0.3785 - accuracy: 0.8356
Epoch 4/10
391/391 [==============================] - 52s 130ms/step -
loss: 0.3391 - accuracy: 0.8542
Epoch 5/10
391/391 [==============================] - 52s 130ms/step -
loss: 0.3231 - accuracy: 0.8633
Epoch 6/10
391/391 [==============================] - 53s 131ms/step -
loss: 0.3166 - accuracy: 0.8668
Epoch 7/10
391/391 [==============================] - 53s 131ms/step -
loss: 0.3094 - accuracy: 0.8702
Reached 87.0% accuracy
```

Output after line 62 of Listing 3.10:

```
Model: "sequential"
_______________________________________________________________
 Layer (type)                Output Shape              Param #
===============================================================
 text_vectorization (TextVec  (None, None)              0
 torization)

 embedding (Embedding)       (None, None, 32)        32000

 lstm (LSTM)                 (None, 32)              8320

 dropout (Dropout)           (None, 32)              0

 dense (Dense)               (None, 32)              1056

 dense_1 (Dense)             (None, 1)               33

=================================================================
Total params: 41,409
Trainable params: 41,409
Non-trainable params: 0
```

Table 3.23 Explanation of the sentiment analysis code using LSTM presented in Listing 3.10

Line number	Description
2–8	Importing Matplotlib, TensorFlow module, LSTM, and embedding layers
4	Importing IMDB movie review dataset from TensorFlow dataset
10–13	Loading dataset and splitting into test and train datasets
17	Initialize batch size. Batch size 64 means 64 reviews will be fed at once into the LSTM model
18–21	Shuffling test and train datasets to avoid bias
23–30	Initializing vocabulary size and tokenizing or vectorization is done in order to map every word to a fixed number
32–40	Initializing accuracy threshold and defining callback function to terminate training process at this threshold
36–52	Defining the whole sentiment analysis model
45–46	Vectorization is not a layer; it just creates a sequence of tokens for a review, while the embedding layer outputs a sequence of vectors for the same review. These vectors are trainable; thus, embedding is achieved after training based on context, meaning, and semantics. Here, mask_zero is set to true to handle variable sequence lengths.
48–52	Defining LSTM model with 32 units. This unit means the dimension of the output (h_t in Fig. 3.36). The number of individual units depends on the sequence of the given review. The LSTM unit iterates through each embedding vector given by the embedding layer and passes output to the next unit to finally give an output of dimension (batch_size, unit)—in our example (64,32). This output is further processed in two dense layers to output the number to predict review.
54–56	Compile the defined LSTM model
58–59	Training process begins here
61	Model summary with the total trainable parameter is shown here
64–75	Loss and accuracy of the model are plotted with respect to epochs
77–79	Accuracy and loss of the trained model are determined over the test dataset
81–87	Function to predict sentiment for the given sentence is defined
90–95	Prediction is performed (you can give your review to predict sentiment)

Output after line 79 of Listing 3.10:

```
Test loss:0.323529988527298, Test accuracy:0.8559200167655945
```

Output after lines 92 and 95 of Listing 3.10:

```
Negative review!!
Positive review!!
```

3.7 Unsupervised Learning

Sometimes, datasets do not consist of labeled data that can be used to train the model for the purpose of classification or regression. In such cases as image captioning, automated speech translation, self-driving cars, recommendation systems, and

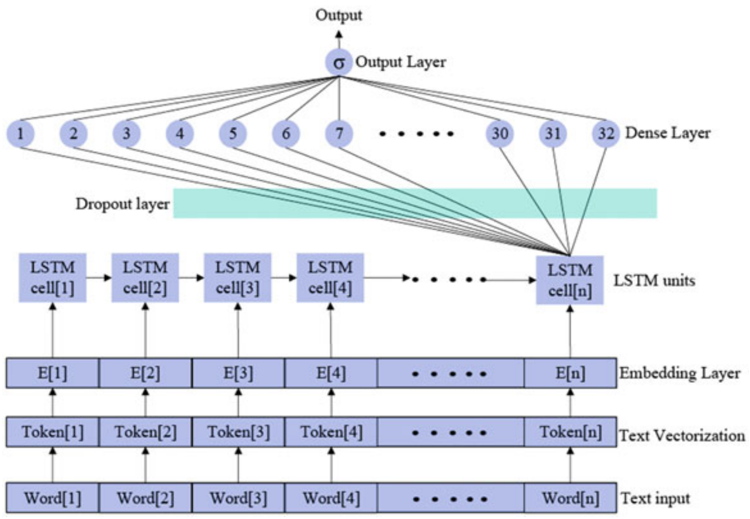
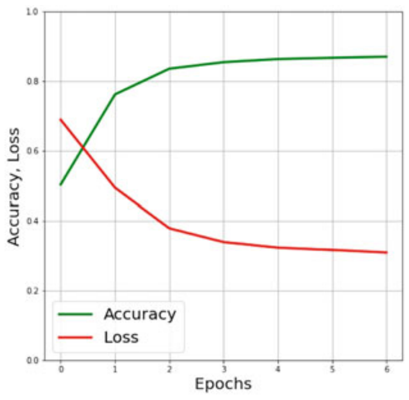


Fig. 3.37 Overall structure of Listing 3.10

Fig. 3.38 Output after line 75 of Listing 3.10



others, unsupervised learning is used. Unsupervised learning is a method where the models are provided with unlabeled data to explore and understand different patterns and structures from the data. The model then categorizes the data into some groups according to the similarities and dissimilarities in the data patterns and structures. Unsupervised learning is instrumental to obtaining insightful observations from underlying patterns. Furthermore, this machine learning method does not require tedious manual input from the user to label the dataset. In this section, we will talk about clustering, dimensionality reduction, and association learning.

3.7.1 Clustering

In unsupervised learning, similarities and dissimilarities in data patterns and structures are explored. Then, this knowledge is used to group them into multiple groups. These groups are formed so that the data in the same group have the most similarity and the data from different groups have the most dissimilarity. These groups are called *clusters*, and the method to form these groups is called *clustering*.

The characteristic of good clustering is that intra-clusters should have the most similarity and inter-clusters should have the most dissimilarity. In Fig. 3.39 [36], the datapoints in cluster A have the most similarity, and they have the most dissimilarity from the datapoints in cluster B. The distance metrics are used to measure the similarities and dissimilarities among clusters. Four algorithms to perform clustering are discussed in the following sections.

3.7.1.1 K-Means Clustering

In k-means clustering, all the observations are divided into a certain number of clusters, and each observation belongs to the cluster with the nearest mean. This is an unsupervised learning method, so labeled data are not required. The only input parameter is the number of clusters (k).

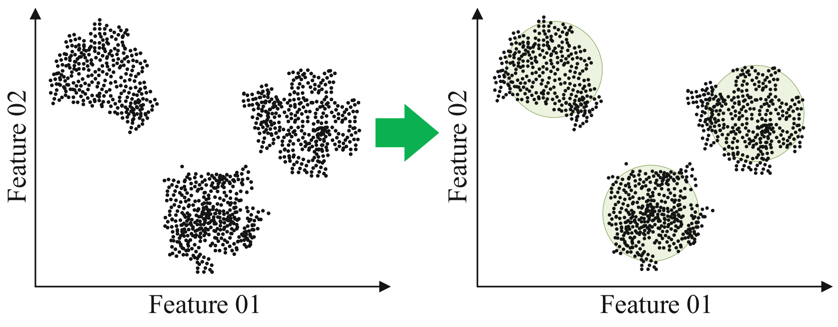


Fig. 3.39 Cluster identification [35]

3.7.1.1.1 The Elbow Method

The elbow method is the most used method for finding the best possible number of clusters. In this case, the sum of squared errors (SSEs) or distances is measured for a range of probable clusters for each observation, using Eq. 3.61:

$$\mathrm{SSE} = \sum_{i=1}^{m} (x_i - c_i)^2 , \tag{3.61}$$

where x_i is a data point and c_i is the appointed centroid. Then, a plot is made to understand the number of clusters better. We take the point after which the SSE decreases linearly as the elbow point. Thus, we find the optimal number of clusters [37–39].

Programming Example 3.12

Listing 3.11 utilizes the elbow method to determine the optimal number of clusters. The output of the code is illustrated in Fig. 3.40 followed by its explanation in Table 3.24. The code generates a synthetic dataset and iterates through different numbers of clusters to find out the SSE of each cluster. When the SSEs are plotted with respect to the number of clusters, an elbow can be seen, which indicates the proper number of clusters for the generated dataset.

```python
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs


# ---------------------Generate Sample Data----------------------
n_samples = 1500
random_state = 170
X, y = make_blobs(n_samples=n_samples, random_state=random_state)


# ---------Finding Out SSE Value for Number of Clusters----------
k_rng = range (1,10)
sse = []
for k in k_rng:
    km = KMeans(n_clusters=k)
    km.fit(X)
    sse.append(km.inertia_)


# ---------------------------Plotting----------------------------
plt.xlabel('n')
plt.ylabel('sse')
plt.plot(k_rng,sse)
plt.show()
```

Listing 3.11 Number of cluster detecting using elbow method [40]

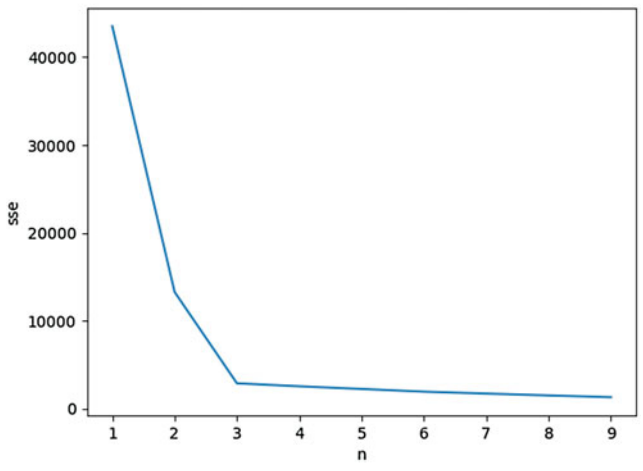


Fig. 3.40 Cluster elbow method

Table 3.24 Explanation of the elbow method coding example presented in Listing 3.11

Line number	Description
1–3	Importing Matplotlib, k-means, and make_blob modules
7–9	Creating blobs as sample data
13–18	Testing what number of clusters (limited to 1–10 for fast processing) gives the best Sum of Squared Error (SSE)
22–25	Plotting SSE vs. the number of clusters to find the elbow

3.7.1.1.2 The k-Means Algorithm

The k-means algorithm uses an iterative approach to divide a set of N samples into several disjoint clusters. It aims to choose centroids that minimize the sum of the squared errors (SSEs) between the centroid and the data points, as shown by the following equation:

$$\text{SSE} = \sum_{i=0}^{n} \min_{\mu_j \in C} (||x_i - \mu_j||^2). \tag{3.62}$$

The algorithm has two steps, as shown below. Alternating between these steps completes the clustering process.

1. **Assignment step:** Each data point is assigned to the cluster with the nearest mean.
2. **Update step:** The means for the data points assigned to each cluster are recomputed.

The means are commonly called the *cluster centroids*. The cluster centroids are not data points, although they live in the same space. The algorithm converges when the assignments no longer change. A drawback of the algorithm is that the formation of clusters is affected by outliers, which cause them to form entirely different clusters.

Programming Example 3.13
Listing 3.12 is an example of k-means clustering using a synthetic dataset that was used in Listing 3.11. The original data points and the clustered data points are compared in Fig. 3.41 after the clustering operation is done using the k-means function. The explanation is presented in Table 3.25.

```python
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs


# -----------Generate Sample Data & Initial Plotting-------------
plt.figure(figsize=(12,4))
X, y = make_blobs(n_samples=1500, cluster_std=0.5, random_state
    =0)
plt.subplot(121)
plt.scatter(X[:, 0], X[:, 1])
plt.title("Original data")


# ---------------------KMeans Clustering------------------------
y_pred = KMeans(n_clusters=3, random_state=0).fit_predict(X)
plt.subplot(122)
plt.scatter(X[:, 0], X[:, 1], c=y_pred)
plt.title("Clustered data")
plt.show()
```

Listing 3.12 K-means clustering [40]

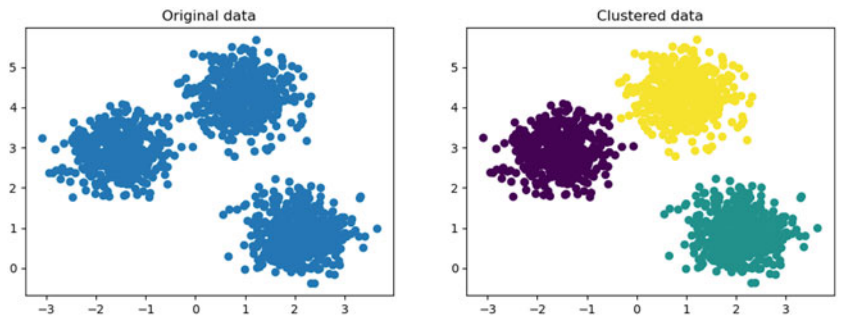


Fig. 3.41 K-means clustering output

Table 3.25 Explanation of the k-means code example presented in Listing 3.12

Line number	Description
1–3	Importing Matplotlib, k-means, and make_blob modules
8	Creating blobs as sample data to apply k-means algorithm
7–11	Creating sample data and plotting them
15	Using the k-means function from scikit-learn to divide all the blobs into three different clusters
16–19	Plotting with original blobs and visually identifying three different clusters in different colors

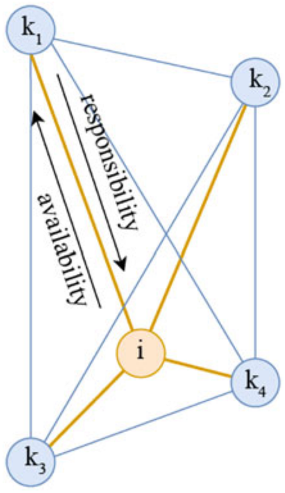


Fig. 3.42 Availability and responsibility of five data points

3.7.1.2 Affinity Propagation Clustering

Affinity propagation is a clustering method that determines the number of clusters (k) on its own. In each cluster, there is only one exemplar. This data point is the most significant for a cluster, and this point is determined collectively by all the data points of that cluster. Affinity propagation is a network where each data point sends messages to all other data points, conveying their willingness to become an exemplar.

For affinity propagation, two concepts are fundamental: availability and responsibility. These are basically represented as graphs or matrices [41], as depicted in Fig. 3.42.

1. **Availability:** Availability tells how appropriate it is for point i to choose point k as its exemplar. It is represented as $a(i, k)$. The availability of sample k to be the exemplar of sample i is given by

$$a(i, k) \leftarrow \min[0, r(k, k) + \sum_{i' \ s.t. \ i' \notin \{i,k\}} r(i', k)]. \tag{3.63}$$

2. **Responsibility:** Responsibility tells how befitting point k is to be an exemplar for point i. It is represented as $r(i, k)$. The responsibility of a sample k to be the exemplar of sample i is given by [42]

$$r(i, k) \leftarrow s(i, k) - \max[a(i, k') + s(i, k')\forall k' \neq k]. \tag{3.64}$$

Here $s(i, k)$ refers to the similarity between samples i and k.

The values for r and a are initialized to zero. The calculation is iterated until convergence. To avoid numerical oscillations when updating the messages, a damping factor λ is introduced to the iteration process. The updated equations for responsibility and availability are as follows, where t indicates the number of iterations:

$$r_{t+1}(i, k) = \lambda \cdot r_t(i, k) + (1 - \lambda) \cdot r_{t+1}(i, k), \tag{3.65}$$

$$a_{t+1}(i, k) = \lambda \cdot a_t(i, k) + (1 - \lambda) \cdot a_{t+1}(i, k). \tag{3.66}$$

Programming Example 3.14
The code for implementing affinity propagation is presented in Listing 3.13, followed by its output in Fig. 3.43 and explanation in Table 3.26. A synthetic dataset is processed with the `AffinityPropagation` function from scikit-learn. The output compares the original data points with the clustered data points.

```
from sklearn.cluster import AffinityPropagation
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
from itertools import cycle


# -----------Generate Sample Data & Initial Plotting-------------
X, labels_true = make_blobs(n_samples=300, cluster_std=0.5,
    random_state=0)
plt.figure(figsize=(12,4))
plt.subplot(121)
plt.plot(X[:, 0], X[:, 1],'.')
plt.title("Original data")


# -----------------Compute Affinity Propagation------------------
af = AffinityPropagation(preference=-50).fit(X)
cluster_centers_indices = af.cluster_centers_indices_
labels = af.labels_

cluster_num = len(cluster_centers_indices)


```

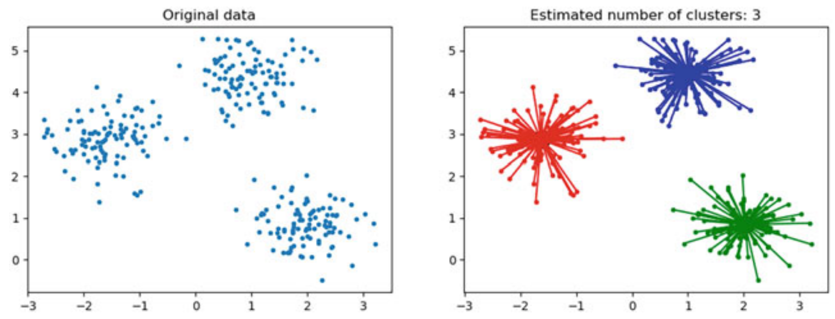


Fig. 3.43 Output of affinity propagation

Table 3.26 Explanation of the affinity propagation code example presented in Listing 3.13

Line number	Description
1–4	Importing Matplotlib, affinity propagation, and make_blob and cycle modules
8	Creating blobs as sample data
8–12	Creating sample data and plotting them
16–18	Using affinity propagation function to find out different sets of clusters where the preference is chosen to be low for getting a small number of clusters
20	Determining the number of clusters set by the affinity propagation algorithm
24–35	Plotting original data with clustered data

```
23 # -------------------------Plot Results--------------------------
24 plt.subplot(122)
25 colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')
26 for k, col in zip(range(cluster_num), colors):
27     class_members = labels == k
28     cluster_center = X[cluster_centers_indices[k]]
29     plt.plot(X[class_members, 0], X[class_members, 1], col + '.')
30     plt.plot(cluster_center[0], cluster_center[1], 'o',
       markerfacecolor=col, markeredgecolor='k', markersize=14)
31     for x in X[class_members]:
32         plt.plot([cluster_center[0], x[0]], [cluster_center[1], x
       [1]], col)
33 
34 plt.title('Estimated number of clusters: %d' % cluster_num)
35 plt.show()
```

Listing 3.13 Affinity propagation implementation [43]

3.7.1.3 Mean-Shift Clustering

The mean-shift algorithm does not require any prior knowledge of the number of clusters. Within a region, it updates the centroid candidates (i.e., kernels) to be the mean of the neighboring points. A post-processing stage then filters the candidates to remove the near-duplicates, and the final set of centroids is obtained.

The mean-shift algorithm follows a set of instructions [44]. These are:

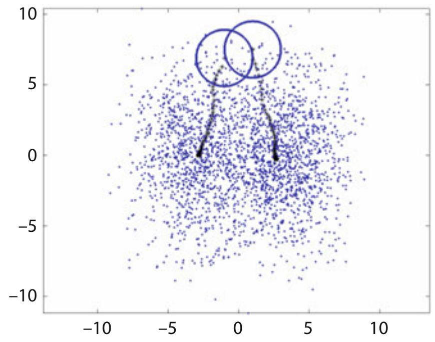


Fig. 3.44 Finding the center of mass within each kernel [45]

1. Initialize kernels that cover every data point.
2. Calculate the center of mass within the kernel. This step is represented by Fig. 3.44.
3. Translate the kernel to the center of the mean.
4. Repeat steps 2 and 3 until convergence.

For any iteration t, if the candidate centroid is x_i, then the candidate is updated in the following way:

$$x_i^{t+1} = x_i^t + m(x_i^t), \tag{3.67}$$

where $N(x_i)$ refers to the samples within the neighborhood of x_i, and m is the mean-shift vector that is calculated for each centroid in the direction of a region of the maximum increase in the density of points.

The value of m is calculated using the following equation:

$$m(x_i) = \frac{\sum_{x_j \in N(x_i)} K(x_j - x_i)x_j}{\sum_{x_j \in N(x_i)} K(x_j - x_i)}. \tag{3.68}$$

Thus, the centroid is updated until it becomes the mean of the samples in its neighborhood.

Programming Example 3.15
Listing 3.14 presents a code that uses the mean-shift algorithm on a synthetic dataset. The estimated number of clusters is shown in the title of the plot in Fig. 3.45. The explanation of the code is presented in Table 3.27. In the mean-shift clustering, the algorithm takes data points and bandwidth as input. Using proper bandwidth is essential for mean shift, which in this case is determined using the `estimate\_bandwidth` function from scikit-learn. The parameter `quartile` controls how small or big the bandwidth will be.

```python
import numpy as np
from sklearn.cluster import MeanShift, estimate_bandwidth
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
from itertools import cycle


# ---------------------Generate Sample Data----------------------
X, _ = make_blobs(n_samples=10000,cluster_std=0.5,random_state=0)


# ---------------------------Plotting----------------------------
plt.figure(figsize=(12,4))
plt.subplot(121)
plt.plot(X[:, 0], X[:, 1],'.')
plt.title("Original data")


# --------------Compute Clustering with MeanShift----------------
bandwidth = estimate_bandwidth(X, quantile=0.2, n_samples=500)

ms = MeanShift(bandwidth=bandwidth, bin_seeding=True)
ms.fit(X)
labels = ms.labels_
cluster_centers = ms.cluster_centers_

labels_unique = np.unique(labels)
n_clusters_ = len(labels_unique)


# -------------------------Plot Results--------------------------
plt.subplot(122)
colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')
for k, col in zip(range(n_clusters_), colors):
    my_members = labels == k
    cluster_center = cluster_centers[k]
    plt.plot(X[my_members, 0], X[my_members, 1], col + '.')
    plt.plot(cluster_center[0], cluster_center[1], 'o',
    markerfacecolor=col,
             markeredgecolor='k', markersize=14)
plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```

Listing 3.14 Mean-shift implementation [46]

3.7.1.4 DBSCAN: Density-Based Spatial Clustering of Applications with Noise

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a density-based clustering algorithm, as the name suggests. The algorithm has two important parameters:

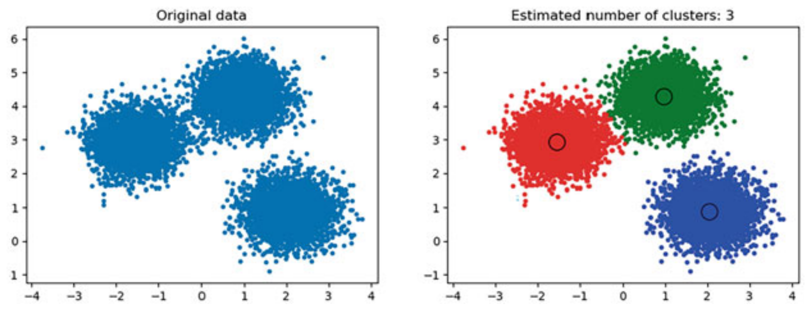
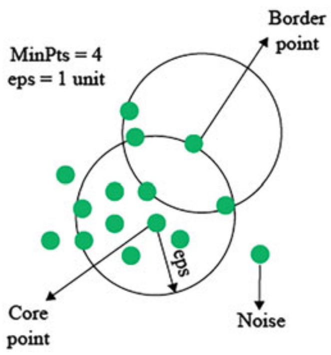


Fig. 3.45 Mean-shift output

Table 3.27 Explanation of the mean-shift clustering code example presented in Listing 3.14

Line number	Description
1–4	Importing Matplotlib, MeanShift, make_blob, NumPy, and cycle modules
9	Creating blobs as sample data
13–16	Plotting sample data
20	Setting bandwidth using estimate_bandwidth function from scikit-learn
22–28	Using MeanShift function and fitting sample data to determine clusters
32–41	Plotting original data with clustered data

Fig. 3.46 The concept of DBSCAN clustering



1. `eps`: If the distance between two points is lower than or equal to `eps`, then they are considered neighbors.
2. `min_samples`: It is the minimum number of neighbors (data points) within the `eps` radius.

The core point, border point, and noise are also essential terminologies for this algorithm. Figure 3.46 can help understand these terminologies.

The points neighboring `eps` are identified along with the core points (visited with more than `min_samples` neighbors). If the core point does not already belong to a cluster, it will be assigned to a new one. Then, we can get a complete cluster if we find all its density-connected points and assign them to the same cluster as the core point. The points that do not belong to any cluster are noise [47–49].

Programming Example 3.16
Listing 3.15 presents a Python implementation of a clustering algorithm, DBSCAN. The code uses `StandardScaler` from scikit-learn to scale the dataset. The parameters `eps` and `min_samples` are used as inputs of the DBSCAN function. The scaled data points and the clustered data points are compared in Fig. 3.47. Table 3.28 incorporates the explanation of the code in this listing.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler


# ---------------------Generate Sample Data----------------------
X, labels_true = make_blobs(n_samples=1000, cluster_std=0.5,
    random_state=0)

X = StandardScaler().fit_transform(X)

plt.figure(figsize=(12,4))
plt.subplot(121)

plt.plot(X[:, 0], X[:, 1],'b.')

plt.title("Original data")


# ------------------------Compute DBSCAN-------------------------
db = DBSCAN(eps=0.3, min_samples=5).fit(X)
# the maximum distance between two samples (eps) is 0.3, and
# the minimum number of samples in a neighbourhood for a point to
     be considered as a core point is 5
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_

# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_noise_ = list(labels).count(-1)

print('Estimated number of clusters: %d' % n_clusters_)
print('Estimated number of noise points: %d' % n_noise_)


# -------------------------Plot Result--------------------------
plt.subplot(122)
```

```

# Black removed and is used for noise instead.
unique_labels = set(labels)
colors = [plt.cm.Spectral(each)
          for each in np.linspace(0, 1, len(unique_labels))]
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = [0, 0, 0, 1]

    class_member_mask = (labels == k)

    xy = X[class_member_mask & core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=14)

    xy = X[class_member_mask & ~core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=6)
plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```

Listing 3.15 DBSCAN implementation [50]

Output of Listing 3.15:

```
Estimated number of clusters: 3
Estimated number of noise points: 3
```

3.7.2 Dimensionality Reduction

In the modern world, objects are represented by digital data. Very high-dimensional data signals are required to represent these objects. When we need to work with such large amounts of data, for example, when we need to analyze thousands of people's facial identification or search over a large database for matching bank accounts, we

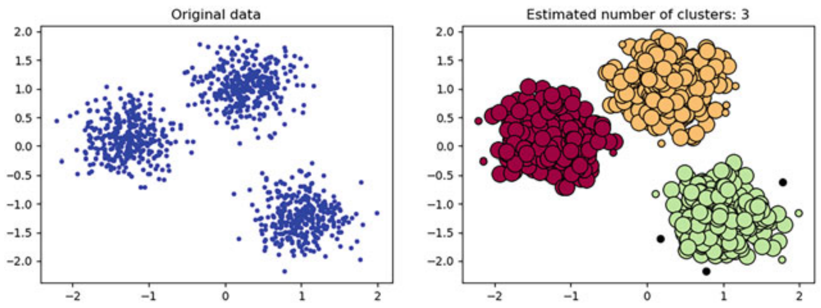


Fig. 3.47 DBSCAN output diagram

Table 3.28 Explanation of the DBSCAN code example presented in Listing 3.15

Line number	Description
1–5	Importing Matplotlib, DBSCAN, make_blob, NumPy, and StandardScaler modules
9	Creating blobs as sample data
13	Processing (scaling) the data to something useable
15–18	Plotting sample data
22–27	Using DBSCAN function and fitting sample data to determine clusters
30–31	Counting the number of clusters and the number of noise points
33-34	Printing the number of clusters and the number of noise points
41–43	Setting up colors and labels
44–59	Plotting clustered data with noise points marked in black color

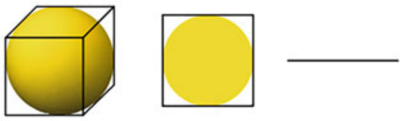


Fig. 3.48 Dimensionality reduction concept

need to reduce the dimension of the data without losing important information [51–53]. Figure 3.48 expresses the concept of dimensionality reduction. The first image is a sphere, which is a three-dimensional object. Then, its dimension is reduced to two when it becomes a circle. Again, when it becomes a line segment, its dimension further reduces to one.

In this section, we will discuss three techniques for dimensionality reduction: principal component analysis, linear discriminant analysis, and singular value decomposition.

3.7.2.1 Principal Component Analysis (PCA)

The principal component analysis (PCA) is a prominent technique for dimensionality reduction. The main idea is to reduce the dimensionality of a dataset that still contains most of the information. Suppose we have a set of d-dimensional data ($y_i \in \mathbb{R}^d; i = 1, \ldots, N$). PCA is handy in finding a linear manifold of a dimension q containing most of the data variables, which is lower than the initial dimension ($q < d$). Now if the mean is $\overline{y} = \frac{1}{N}\sum_{i=1}^{N} y_i$, then PCA computes the covariance matrix of these data as

$$C = \frac{1}{N-1}\sum_{i=1}^{N}(\overline{y} - y)(\overline{y} - y)^T. \tag{3.69}$$

After determining the eigenvalues and corresponding eigenvectors, we can define a $d \times q$ matrix, and after proper mapping, we will find the principal components [52–57].

Programming Example 3.17
Listing 3.16 (explained in Table 3.29) is an example of dimensionality reduction using PCA. The dataset we are using has data regarding the classification of mushrooms. Assuming the data has no null value, a label encoder represents the features numerically. The class data is stored in the first column, so we separate class features as `label` and others as `features_columns`. These datasets are now properly scaled. The scaling, in this case, is not for using a classifier. Scaling may impact how PCA analyzes these data.

PCA is implemented using a scikit-learn module to calculate the variance of each feature. Then, these data are sorted and plotted according to the variance of these features (maximum variance to minimum variance). The plot in Fig. 3.49 shows that 95% of the data are represented with up to the 15th feature. Thus, deducting the remaining features will not affect the data very much.

```
#Dataset: https://www.kaggle.com/datasets/uciml/mushroom-
    classification
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split

# -------------------------Reading Data--------------------------
df = pd.read_csv('./data/mushrooms.csv')


# ---------Encode These String Characters into Integers----------
encoder = LabelEncoder()


# Applying transformation
for column in df.columns:
    df[column] = encoder.fit_transform(df[column])

X = df.iloc[:,1:23]
Y = df.iloc[:, 0]


# ----------------------Scale the Features-----------------------
ss = StandardScaler()
X = ss.fit_transform(X)


# -------------------------Fit and Plot--------------------------
pca = PCA()
X_pca = pca.fit_transform(X)
explained_variance = pca.explained_variance_

# Plot before dimension reduction
```

```
35 plt.figure(figsize=(12, 4))
36 plt.subplot(121)
37 plt.bar(range(22), explained_variance, alpha=0.5, align='center',
        label='individual variance')
38 plt.title("Original data")
39
40
41 # Reduce the number of features using PCA
42 pca_reduced = PCA(n_components=15)
43 X_pca_reduced = pca_reduced.fit_transform(X)
44 explained_variance_reduced = pca_reduced.explained_variance_
45
46 # Plot after dimension reduction
47 plt.subplot(122)
48 plt.bar(range(15), explained_variance_reduced, alpha=0.5, align='
      center')
49 plt.title("Dimension Reduced")
50 plt.show()
```

Listing 3.16 PCA algorithm

3.7.2.2 Linear Discriminant Analysis (LDA)

The linear discriminant analysis (LDA) distinguishes the samples from the training dataset with respect to the class values. This method searches for a linear combination of input variables. The linear combination should be such that the amount of separation between the samples of different class centroids or means is the maximum and the amount of separation between the samples of the same class is the minimum.

When the class centroids are far from each other, it performs best. In case the class centroids are common in more than one class, it will be impossible for LDA to distinguish the samples of those classes [57–62].

Table 3.29 Explanation of the PCA code example presented in Listing 3.16

Line number	Description
1–7	Importing PyPlot, pandas, and some of sklearn modules for labeling, scaling, and implementation of PCA
9	Reading CSV data
13	Encoder for encoding non-numerical features
17–18	Encoding with label encoder
20–21	Splitting up the features (columns) and labels
25–26	Scaling features using StandardScaler [Optional]
30–32	Using PCA to fit and transform in order to get the list of features that have
	the most variance
34–38	Plotting the features with the order of maximum and minimum variance
42–50	Considering the first 15 features as top features plotting the features again

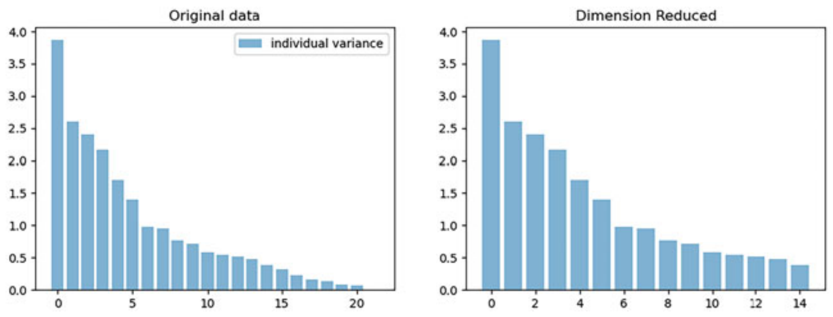


Fig. 3.49 PCA output

LDA works in the following three major steps to generate a new plot separating the two data points:

1. Based on the distance between the class centroids, the separability is determined.
2. The distance between samples and centroids for different classes is calculated.
3. The lower dimensional space has to be constructed to maximize the variance (the distance between the class centroids) between different classes.

Programming Example 3.18

Listing 3.17 (explained in Table 3.30) is an example of dimensionality reduction using LDA. The dataset used represents different features of the passengers of the ship Titanic.

After reading the data, a null check is done. Upon finding any null values in the `Age` & `Embarked` column, the null value will be replaced by the `median` & `"S"` accordingly. The non-numeric values in the `Sex` & `Embarked` column will be replaced by numeric values using `LabelEncoder()`. Reshaping the `Age` & `Fare` column as these will be scaled later using the `StandardScaler()`.

Once the data are split into training and testing sets, the model is fitted to the training set and used to predict the testing set. The printed output displays the accuracy of the predictions. The dataset is again transformed with a specific number of components expected for LDA and fitted the model to features and labels. The original and reduced features are compared and printed out for a better idea. A logistic regression classifier is used on these data. If the accuracy test is run again, the change in performance is noticed.

```python
#Dataset: https://www.kaggle.com/competitions/titanic/data?select
    =train.csv
import pandas as pd
import numpy as np
from sklearn.metrics import accuracy_score, f1_score
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.discriminant_analysis import
    LinearDiscriminantAnalysis as LDA
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# -------------------------Reading Data--------------------------
df = pd.read_csv("./data/train.csv")
df["Age"].fillna(df["Age"].median(), inplace=True)
df["Embarked"].fillna("S", inplace=True)
encoder = LabelEncoder()


# ------------------------Label Encoding-------------------------
encoder.fit(df["Sex"])



df_temp = encoder.transform(df["Sex"])
df["Sex"] = df_temp
encoder.fit(df["Embarked"])
df_temp = encoder.transform(df["Embarked"])
df["Embarked"] = df_temp


# Reshape data
agesArray = np.array(df["Age"]).reshape(-1, 1)
faresArray = np.array(df["Fare"]).reshape(-1, 1)


# Scale the X
ss = StandardScaler()
df["Age"] = ss.fit_transform(agesArray)
df["Fare"] = ss.fit_transform(faresArray)


# -------------------------Split Dataset-------------------------
X = df.drop(labels=['PassengerId', 'Survived'], axis=1)
Y = df['Survived']


# Splitting & fitting train data
xtrain, xval, ytrain, yval = train_test_split(X, Y, test_size
    =0.2, random_state=27)

lda_model = LDA()
lda_model.fit(xtrain, ytrain)
lda_predictions = lda_model.predict(xval)
lda_acc = accuracy_score(yval, lda_predictions)
lda_f1 = f1_score(yval, lda_predictions)

print("LDA Model - Accuracy: {}".format(lda_acc))
print("LDA Model - F1 Score: {}".format(lda_f1))

```

```python

# ----------------------LDA Transformation----------------------
lda_new = LDA(n_components=1)
lda_new.fit(X, Y)
X_lda = lda_new.transform(X)


# Printing result
print('Original feature #:', X.shape[1])
print('Reduced feature #:', X_lda.shape[1])


# Splitting with the new features and run the classifier
x_train_lda, x_val_lda, y_train_lda, y_val_lda = train_test_split
    (X_lda, Y, test_size=0.2, random_state=27)

logistic_regression = LogisticRegression()
logistic_regression.fit(x_train_lda, y_train_lda)
logreg_predictions = logistic_regression.predict(x_val_lda)
logreg_acc = accuracy_score(y_val_lda, logreg_predictions)
logreg_f1 = f1_score(y_val_lda, logreg_predictions)
print("Logistic Regression Model - Accuracy: {}".format(
    logreg_acc))
print("Logistic Regression Model - F1 Score: {}".format(logreg_f1
    ))
```

Listing 3.17 LDA algorithm

Output of Listing 3.17:

```
Accuracy: 0.8100558659217877
F1 Score: 0.734375

Original feature #: 7
Reduced feature #: 1

Accuracy: 0.8212290502793296
F1 Score: 0.7500000000000001
```

3.7.2.3 Singular Value Decomposition (SVD)

The singular value decomposition (SVD) technique is similar to PCA, but more general. The motivation of SVD is to make the process of calculation with matrices easier. We basically reduce the columns of the matrix.

In this case, as shown in Fig. 3.50, we are assuming that matrix A can be represented by three other matrices—an orthogonal matrix (U), a diagonal matrix (Σ), and the transpose of another orthogonal matrix (V). U has m×m items, Σ has m×n items, V^t has n×n items, and matrix A has m×n items.

As U and V are orthogonal and Σ is a diagonal matrix, we can deduce the following equations for the three matrices:

$$UU^T = U^TU = I. \tag{3.70}$$

Table 3.30 Explanation of the LDA code example presented in Listing 3.17

Line number	Description
1–7	Importing NumPy, pandas, and some of sklearn modules
11–13	Reading CSV data and filling in any missing data in the case of the Age and Embarked feature
14	Encoder for non-numerical features
18–26	Encode both the Sex and Embarked feature
30–31	Converting to arrays and reshaping the arrays
35–37	Scaling both features using StandardScaler
41–42	Selecting necessary features and label
46	Splitting test and train data
48–52	Fit training data and validate predict with actual values
54–55	Print metrics for the predictions
59–61	Transforming the features (with a specific number of desired components for LDA)
65–66	Printing out the number of original and reduced features
70–76	Executing the classifier again to notice performance change
77–78	Print metrics for the new predictions

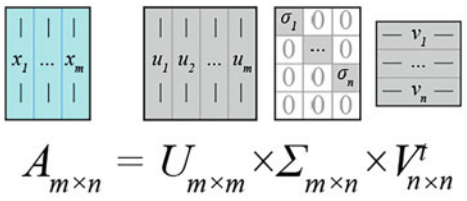


Fig. 3.50 A matrix is represented by three other matrices

$$VV^T = V^T V = I. \tag{3.71}$$

$$\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \ldots \geq \sigma_n \geq 0. \tag{3.72}$$

Each term of the diagonal matrix Σ is greater than the next term. This indicates that the terms multiplied with the next terms will have less (or no) value over the reconstruction of matrix A. As both the singular matrices are multiplied with the diagonal matrix, we can easily assume that the dependency on the right-most variables of matrix A will reduce to some extent. Thus, by dividing a large set of data into three matrices, SVD reduces the dimensions of the dataset [53, 57, 61, 63–65].

Programming Example 3.19
Listing 3.18 is an application of SVD and Table 3.31 explains the listing. Figure 3.51a is the image working as input. The file size of the image is 1.15 MB. Here the objective is to reduce the file size without compromising the image quality.

There are three different functions defined in Listing 3.18 for easily performing this operation. These three functions are:

- **Loading image:** This function uses the Image module from the PIL library to open and then converts the image data into an array. Furthermore, it separates three different channels (Red, Blue, and Green) and returns these as an array.
- **Channel compression:** This is the function where we truly use the singular value decomposition. Here, three matrices (u, s, and v) are separated from the `color_channel` matrix using the same concept from Fig. 3.50. In this case, `n` or `singular_val_lim` is the first `n-terms` considered for SVD among `m` total terms. Then, the three matrices are multiplied to get a compressed but non-distorted image.
- **Image compression:** This function basically uses the previous function for compressing each channel and outputs the image.

The program initiates and uses these three functions to get a compressed image of the input image. The resulting image size is 779 KB. This is a 32.3% compression in file size.

```python
import numpy as np
from numpy.linalg import svd as singular_value_decomposition
from numpy import zeros as create_zeros_matrix
from numpy import matmul as matrix_multiplication
from PIL import Image


# ------------------Class for Image Compression------------------
class ImageCompressor:
    def __init__(self, file_path, singular_value_limit):
        self.file_path = file_path
        self.singular_value_limit = singular_value_limit

    def load_image(self):
        image = Image.open(self.file_path)
        image_channels = np.array(image)
        r = image_channels[:, :, 0]
        g = image_channels[:, :, 1]
        b = image_channels[:, :, 2]
        return r, g, b


# ---------Function for compressing each color channels----------
    @staticmethod
    def compress_channel(channel, limit):
        u, s, v = singular_value_decomposition(channel)
        compressed_channel = create_zeros_matrix((channel.shape
    [0], channel.shape[1]))
        n = limit
```

```
29
30         left_matrix = matrix_multiplication(u[:, 0:n], np.diag(s)[0:n, 0:n])
31         inner_compressed = matrix_multiplication(left_matrix, v[0:n, :])
32         compressed_channel = inner_compressed.astype('uint8')
33         return compressed_channel
34
35 # ----Function for compressing & combining all color channels----
36     def compress_and_combine_channels(self):
37         red_channel, green_channel, blue_channel = self.load_image()
38
39         compressed_red = self.compress_channel(red_channel, self.singular_value_limit)
40         compressed_blue = self.compress_channel(blue_channel, self.singular_value_limit)
41         compressed_green = self.compress_channel(green_channel, self.singular_value_limit)
42
43         red_array = Image.fromarray(compressed_red)
44         blue_array = Image.fromarray(compressed_blue)
45         green_array = Image.fromarray(compressed_green)
46
47         compressed_image = Image.merge("RGB", (red_array, green_array, blue_array))
48         compressed_image.show()
49         compressed_image.save("./results/compressed.jpg")
50
51
52 # ----------------------Image Compression------------------------
53 if __name__ == "__main__":
54     file_path = "data/alex-knight-2EJCSULRwC8-unsplash.jpg"
55     singular_value_limit = 1200
56
57     compressor = ImageCompressor(file_path, singular_value_limit)
58     compressor.compress_and_combine_channels()
```

Listing 3.18 SVD algorithm

Table 3.31 Explanation of the SVD code example presented in Listing 3.18

Line number	Description
1–5	Importing NumPy, SVD, and Image modules
9	Creating a class for compression
14–20	Function for loading the image and splitting Red, Green, and Blue into different arrays
25–33	Function for compressing each channel using the SVD technique and returning compressed array of that channel
57	Loading image and splitting each channel
57	Compressing each channel and getting the compressed image

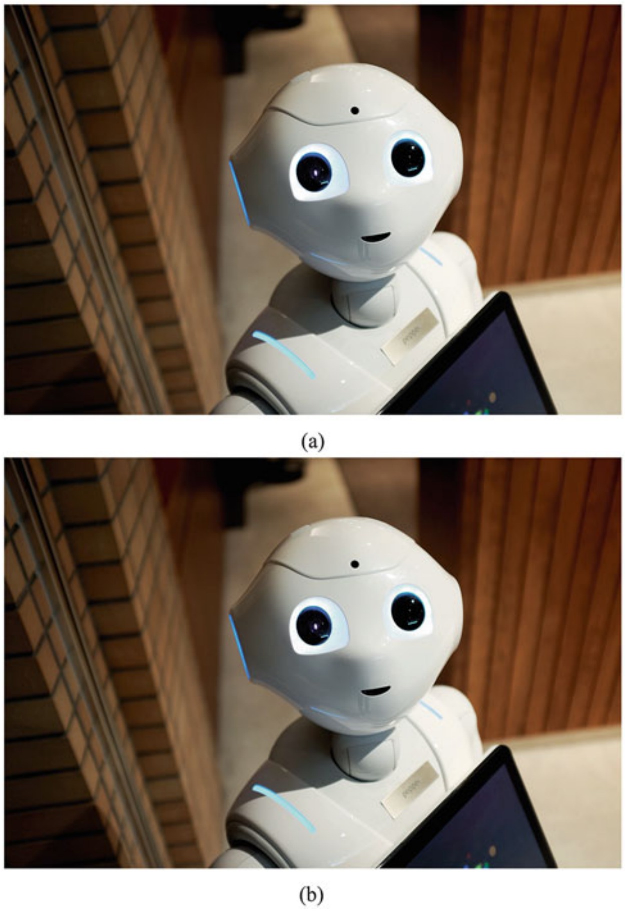


Fig. 3.51 The effect of SVD on the image of a robot. (**a**) An image of a robot before the SVD technique. Photo by Alex Knight on Unsplash [66]. (**b**) An image of a robot after the SVD technique (32% compressed in size)

3.7.3 Association Learning

The association rule is used in market basket analysis, bioinformatics, web usage mining, and intrusion detection. From transaction history, the association rule explores the pattern and relationships in the data. The difference between the association rule and collaborative filtering is that the former studies data from the transaction of all the users as a group, while the latter focuses on one user to identify similar items, which is used in e-commerce websites to recommend similar items.

The association rule is written as {Corn flakes, Bread}→{Milk}, meaning that if a user purchases corn flakes and bread, he will also buy milk in the same transaction. Here the set {Corn flakes, Bread} is called the *antecedent* and {Milk} is called the *consequent*. There can be multiple items in both of these sets. The *itemset* is the

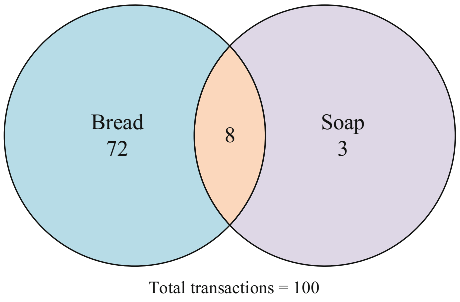


Fig. 3.52 An example to demonstrate the confidence and lift concept

combined list of all the items in antecedent and consequent. Here, the itemset is {Corn flakes, Bread, Milk}.

Three metrics will help us understand association learning: support, confidence, and lift. These three terms are discussed below. The example in Fig. 3.52 will support the description.

1. **Support:** Support is the frequency of an itemset in all the considered transactions. A low support implies that there is insufficient information on the relationship between the items, and we cannot derive conclusions from such a rule. Mathematically,

$$\text{Support}(\{\text{X}\} \rightarrow \{\text{Y}\}) = \text{P}(\text{X} \cup \text{Y}) = \frac{\text{frequency(X,Y)}}{\text{total transactions}}. \tag{3.73}$$

   For instance, the itemset {Bread, Milk} has high support as it is purchased commonly. However, the itemset {Bread, Soap} has low support as this itemset is not purchased frequently.
2. **Confidence:** Confidence is the probability of an item being in the itemset provided another item is present. Regardless of the antecedent, the confidence of a frequent consequent will be high, as it is brought frequently with other items. Mathematically,

$$\text{Confidence}(\{\text{X}\} \rightarrow \{\text{Y}\}) = \frac{\text{Support } (X \cup Y)}{\text{Support X}} = \frac{\text{frequency(X,Y)}}{\text{frequency(X)}}. \tag{3.74}$$

   In Fig. 3.52, the confidence for {Soap}→{Bread} is $8 \div (8+3) = 0.73$, which is a high confidence. But these products have less association, so something is wrong here. The next metric, “Lift,” will eradicate this confusion.
3. **Lift:** Lift indicates the correlation between two items in an itemset. For lift, at first, we assume that the two items are independent. Then, lift is the ratio of the observed support to the expected support. Mathematically,

$$\text{Lift}(\{\text{X}\} \rightarrow \{\text{Y}\}) = \frac{\text{Support } (X \cup Y)}{\text{Support X } \times \text{ Support Y}}. \tag{3.75}$$

- If lift = 1, there is no correlation between X and Y.
- If lift > 1, there is a positive correlation between X and Y.
- If lift < 1, there is a negative correlation between X and Y.

For the first case, the confidence for {Soap}→{Bread} was 0.8. Now the probability of having bread without having statistics of soap is $80 \div 100 = 0.8$. So lift is $0.73 \div 0.8 = 0.91$. As the lift < 1, there is a negative correlation between soap and bread.

Now to generate a rule from the transactions, we need to generate a list of items that will consist of items that occur at least once in the transactions. So first we need to generate itemset like {Corn flakes, Bread, Milk} and then need to extract the rule from a subset of this such as {Corn flakes, Milk}, {Bread, Milk}, etc. [67].

3.7.3.1 Apriori Algorithm

Rule generation is computationally expensive as a huge number of rules are required even for a few items. The Apriori algorithm gives a systematic approach to reducing the number of rules, thus making it more computationally efficient and practically realizable. Let us look at the steps of using the Apriori algorithm.

1. **Generating itemset:** First, all the frequent itemsets that are present at least a minimum number of times are extracted. This can be less than the total number of items. The brute-force approach would be highly time-consuming, but the Apriori principle can do this efficiently. This principle is "All subsets of a frequent itemset must also be frequent." For example, {Bread}→{Milk} has a greater or equal number of transactions than {Corn flakes, Bread}→{Milk}. Thus if {Corn flakes, Bread}→{Milk} has a support value of 0.2, then {Bread}→{Milk} will have a support value ≥ 0.2, which is anti-monotone property of support, thus dropping itemset increase or keeping the support value same.

   Thus, a frequent itemset is generated using support value ≥ minimum support for length = 1,2,3,..., by checking the threshold each time. This is the Apriori algorithm, which prunes the item at every step for the next itemset.
2. **Generating rule:** Now, from the generated frequent itemset, the rules are extracted. A candidate from itemset for rules is made. For example, the candidates from {Corn flakes, Bread}→{Milk} will be:

- (Bread, Milk → Corn flakes)
- (Corn flakes, Milk → Bread)
- (Corn flakes → Bread, Milk) etc.

The confidence is checked from this list and the rules with confidence greater than a minimum confidence value are sorted out. Thus again, pruning is done based on this confidence value. Now these extracted rules satisfy the minimum support and minimum confidence value.

Finally, on these rules, the highest value of lift is searched to make a decision. All these steps can be performed using a standard library.

Here, we will use a standard Python package, named *mlxtend.frequent_patterns*, having "Apriori" and association rules modules. Refer to Table 3.32 to understand the code given in Listing 3.19. For further reading, please check reference [68].

Programming Example 3.20
Listing 3.19 demonstrates association rule mining using the Apriori algorithm. The code reads a bakery transaction dataset (Fig. 3.53), preprocesses it (Fig. 3.54), and converts it into a crosstab format (Fig. 3.55), which is a way to present the relationship between two or more categorical variables in a tabular format. Implementing the Apriori algorithm, the listing efficiently identifies item sets that frequently occur and meets a minimum support threshold of 4% (Fig. 3.56). From there, it generates association rules with a minimum confidence level of 50%, allowing for a comprehensive understanding of the relationships between items in bakery transactions (Fig. 3.57a). The resulting rules are then sorted by lift, providing a clear picture of the most highly associated item sets in transaction patterns (Fig. 3.57b). The explanation of the code is presented in Table 3.32. The output of this code can be best viewed in an IPython Notebook environment.

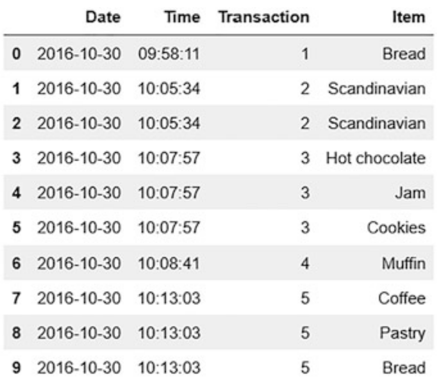
```python
#Data: https://github.com/viktree/curly-octo-chainsaw/blob/master
    /BreadBasket_DMS.csv

import pandas as pd
import numpy as np
from mlxtend.frequent_patterns import apriori, association_rules

#Loading dataset
dataset = pd.read_csv('data/BreadBasket_DMS.csv')

print(dataset.head(10))

#Dropping Duplicate Transaction
dataset = dataset.drop_duplicates()

#Taking Date, Time, Transaction and Item columns
print(dataset[['Date', 'Time', 'Transaction', 'Item']].head(10))

#Convert transacton & item into Crosstab
transaction = pd.crosstab(index= dataset['Transaction'], columns=
     dataset['Item'])
print(transaction.head(10))

#Removing "NONE"
transaction = transaction.drop(['NONE'], axis = 1)

print(transaction.head(10))
```

```

#Frequent itemset with min support = 4%
frequent_itemset = apriori(df = transaction, min_support= 0.04,
    use_colnames= True)
frequent_itemset.sort_values(by = 'support', ascending = False)

#Rule with minimun confidence = 50%
Rules = association_rules(frequent_itemset, min_threshold= 0.5)
print(Rules.head())

#Sorting results by lift to get highly associated itemsets
print(Rules.sort_values(by='lift', ascending= False).head())
```

Listing 3.19 Association learning using the Apriori algorithm [69, 70]

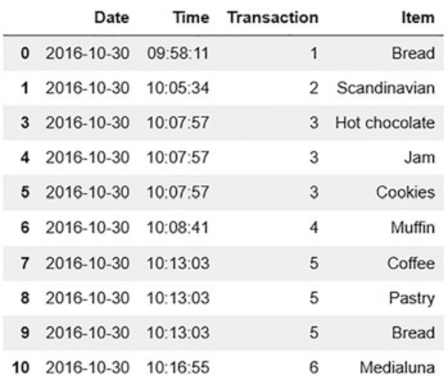
3.7.3.2 ECLAT Algorithm

The Equivalence Class Clustering and bottom-up Lattice Traversal (ECLAT) is a popular association algorithm used for mining data for frequent items in a dataset. It is a more computationally efficient variant of the Apriori algorithm. Unlike the Apriori algorithm, it follows the depth-first search method (i.e., in a vertical manner) of a graph rather than the breadth-first search method (i.e., in a horizontal manner).



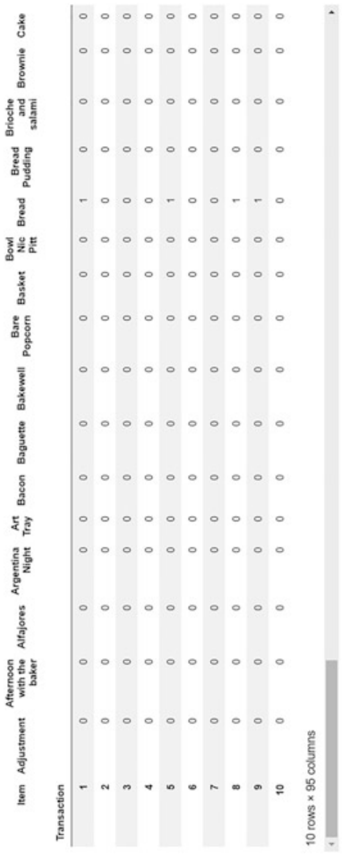
	Date	Time	Transaction	Item
0	2016-10-30	09:58:11	1	Bread
1	2016-10-30	10:05:34	2	Scandinavian
2	2016-10-30	10:05:34	2	Scandinavian
3	2016-10-30	10:07:57	3	Hot chocolate
4	2016-10-30	10:07:57	3	Jam
5	2016-10-30	10:07:57	3	Cookies
6	2016-10-30	10:08:41	4	Muffin
7	2016-10-30	10:13:03	5	Coffee
8	2016-10-30	10:13:03	5	Pastry
9	2016-10-30	10:13:03	5	Bread

Fig. 3.53 Output after line 10 of Listing 3.19



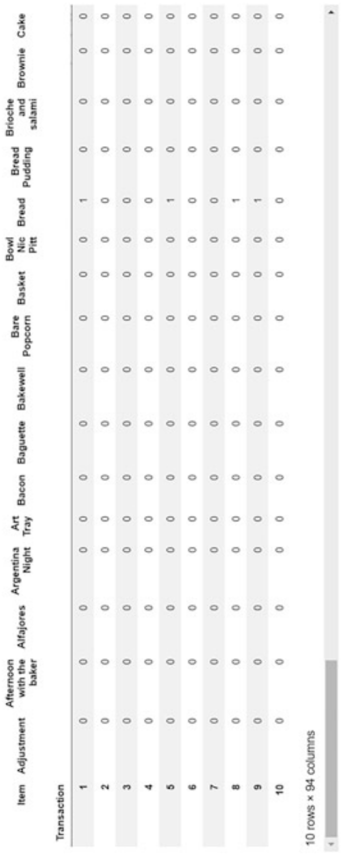
	Date	Time	Transaction	Item
0	2016-10-30	09:58:11	1	Bread
1	2016-10-30	10:05:34	2	Scandinavian
3	2016-10-30	10:07:57	3	Hot chocolate
4	2016-10-30	10:07:57	3	Jam
5	2016-10-30	10:07:57	3	Cookies
6	2016-10-30	10:08:41	4	Muffin
7	2016-10-30	10:13:03	5	Coffee
8	2016-10-30	10:13:03	5	Pastry
9	2016-10-30	10:13:03	5	Bread
10	2016-10-30	10:16:55	6	Medialuna

Fig. 3.54 Output after line 16 of Listing 3.19



```
Item         Adjustment  Afternoon with the baker  Alfajores  Argentina Night  Art Tray  Bacon  Baguette  Bakewell  Bare Popcorn  Basket  Bowl Nic Pitt  Bread  Bread Pudding  Brioche and salami  Brownie  Cake
Transaction
1            0           0                         0          0                0         0      0         0         0             0       0              1      0              0                   0        0
2            0           0                         0          0                0         0      0         0         0             0       0              0      0              0                   0        0
3            0           0                         0          0                0         0      0         0         0             0       0              0      0              0                   0        0
4            0           0                         0          0                0         0      0         0         0             0       0              0      0              0                   0        0
5            0           0                         0          0                0         0      0         0         0             0       0              1      0              0                   0        0
6            0           0                         0          0                0         0      0         0         0             0       0              0      0              0                   0        0
7            0           0                         0          0                0         0      0         0         0             0       0              0      0              0                   0        0
8            0           0                         0          0                0         0      0         0         0             0       0              1      0              0                   0        0
9            0           0                         0          0                0         0      0         0         0             0       0              1      0              0                   0        0
10           0           0                         0          0                0         0      0         0         0             0       0              0      0              0                   0        0

10 rows × 95 columns
```

(a)



```
Item         Adjustment  Afternoon with the baker  Alfajores  Argentina Night  Art Tray  Bacon  Baguette  Bakewell  Bare Popcorn  Basket  Bowl Nic Pitt  Bread  Bread Pudding  Brioche and salami  Brownie  Cake
Transaction
1            0           0                         0          0                0         0      0         0         0             0       0              1      0              0                   0        0
2            0           0                         0          0                0         0      0         0         0             0       0              0      0              0                   0        0
3            0           0                         0          0                0         0      0         0         0             0       0              0      0              0                   0        0
4            0           0                         0          0                0         0      0         0         0             0       0              0      0              0                   0        0
5            0           0                         0          0                0         0      0         0         0             0       0              1      0              0                   0        0
6            0           0                         0          0                0         0      0         0         0             0       0              0      0              0                   0        0
7            0           0                         0          0                0         0      0         0         0             0       0              0      0              0                   0        0
8            0           0                         0          0                0         0      0         0         0             0       0              1      0              0                   0        0
9            0           0                         0          0                0         0      0         0         0             0       0              1      0              0                   0        0
10           0           0                         0          0                0         0      0         0         0             0       0              0      0              0                   0        0

10 rows × 94 columns
```

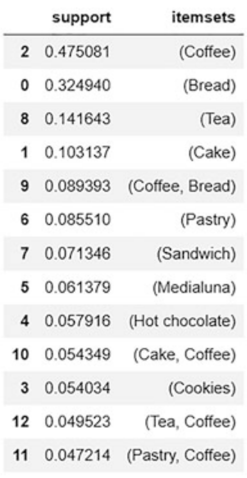
(b)

Fig. 3.55 Output of Listing 3.19. (**a**) Output after line 20. (**b**) Output after line 25

As the depth first search has lesser computational requirements, ECLAT is faster than Apriori.

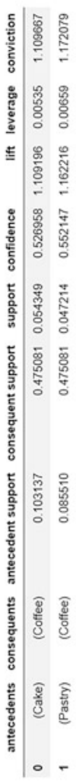
Table 3.33 shows a transaction database in both a horizontal and a vertical manner. The concept of "Transaction ID Set (TIDset)" is important for ECLAT. From the table, it can be observed that item I4 has been present in TID 1, 3, and 4 in the horizontal transaction section. The TIDset of I4 can be expressed as TIDset(I4) = {1, 3, 4}, which is also shown in the vertical transaction section of the table.

The main idea of ECLAT is to use TIDset for scanning frequent 1-itemsets, creating and refining frequent 2-itemsets and frequent 3-itemsets, and repeating this procedure until there are no more possible sets of candidate items to be found [71]. The process of the ECLAT algorithm is depicted in Fig. 3.58, which uses the table for the input.



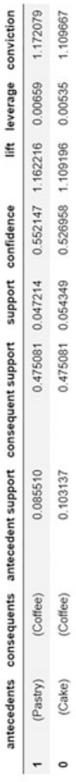
	support	itemsets
2	0.475081	(Coffee)
0	0.324940	(Bread)
8	0.141643	(Tea)
1	0.103137	(Cake)
9	0.089393	(Coffee, Bread)
6	0.085510	(Pastry)
7	0.071346	(Sandwich)
5	0.061379	(Medialuna)
4	0.057916	(Hot chocolate)
10	0.054349	(Cake, Coffee)
3	0.054034	(Cookies)
12	0.049523	(Tea, Coffee)
11	0.047214	(Pastry, Coffee)

Fig. 3.56 Output after line 29 of Listing 3.19



	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction
0	(Cake)	(Coffee)	0.103137	0.475081	0.054349	0.526958	1.109196	0.00535	1.109667
1	(Pastry)	(Coffee)	0.085510	0.475081	0.047214	0.552147	1.162216	0.00659	1.172079

(a)



	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction
1	(Pastry)	(Coffee)	0.085510	0.475081	0.047214	0.552147	1.162216	0.00659	1.172079
0	(Cake)	(Coffee)	0.103137	0.475081	0.054349	0.526958	1.109196	0.00535	1.109667

(b)

Fig. 3.57 Output of Listing 3.19. (**a**) Output after line number 33. (**b**) Output after line number 36

Table 3.32 Explanation of the association learning code using Apriori presented in Listing 3.19

Line number	Description
1–2	Reference of data and code
3–5	Required library and modules
7–10	Loading dataset and displaying first 10 entries
12–13	Dropping duplicate transaction
15–16	Displaying date, time, transaction, and item columns
18–20	Converting to crosstab and displaying
22–25	Removing “NONE” entries
27–29	Generating frequent itemset with the minimum support of 0.04
31–33	Extracting rules with a minimum confidence of 0.5
35–36	Sorting result according to lift in order to make decision

Table 3.33 Sample transaction database for association learning

Horizontal transaction (Apriori)		Vertical transaction (ECLAT)	
TID	Itemset	TID	Itemset
1	I0, I1, I4	I0	1, 4, 5, 7, 8, 9
2	I1, I3	I1	1, 2, 3, 4, 6, 8, 9
3	I1, I2	I2	3, 5, 6, 7, 8, 9
4	I0, I1, I3	I3	2, 4
5	I0, I2	I4	1, 8
6	I1, I2		
7	I0, I2		
8	I0, I1, I2, I4		
9	I0, I1, I2		

3.8 Semi-supervised Learning

We have already discussed supervised and unsupervised learning so far. We use labeled data in supervised learning, whereas in unsupervised learning, we use unlabeled data to train the model. For semi-supervised learning, we use a mixture of a small amount of labeled data with a huge amount of unlabeled data in the training process. So, a little supervision is present to guide the model. Though a lot of unlabeled data are used compared to the labeled data, this mixture improves the learning accuracy by a considerable amount.

In the real world, data acquisition of labeled data is expensive, involves a lot of effort and time, and requires experts in that particular field. On the other hand, the production of unlabeled data is inexpensive and easy. So semi-supervised learning plays a vital role in modeling these real-world data. Some real-world examples are web page classification, web content classification, text-based image retrieval, speech analysis, protein sequence classification, etc. In this section, we are going to explore semi-supervised GAN and semi-supervised classification.

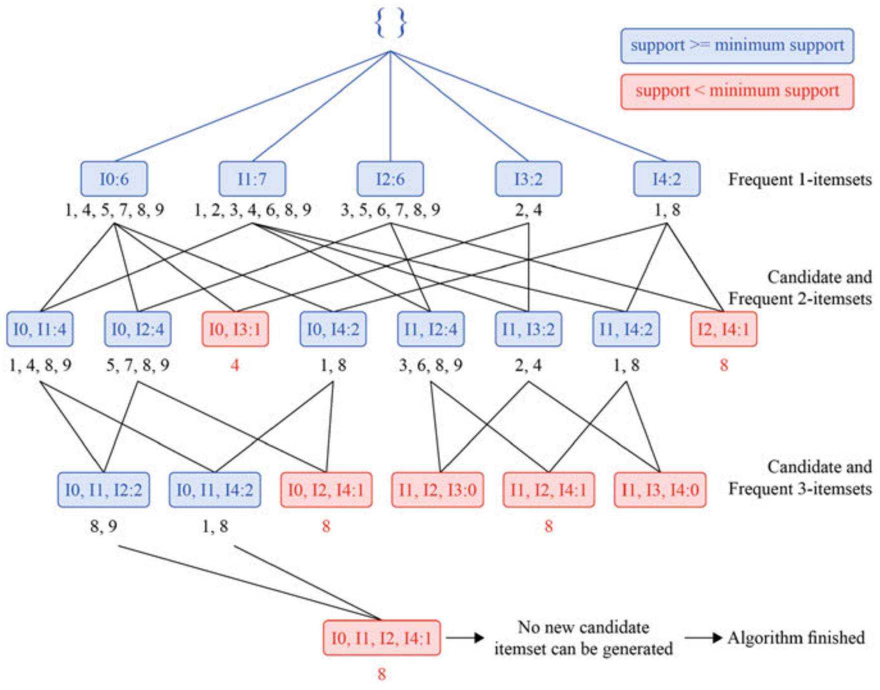


Fig. 3.58 The process of ECLAT algorithm

3.8.1 Semi-supervised GAN (SGAN)

We have already introduced the working principle of the GAN algorithm in Sect. 3.5.6 as a part of deep learning. In this section, we will show the use of GAN in semi-supervised learning algorithms. For example, in the ANN section (Sect. 3.5.3), we built and trained an ANN model to classify digits from the MNIST handwritten dataset, where around 60,000 images have been used. However, if we use SGAN, we can train the model with very little data.

The architecture of the SGAN model is presented in Fig. 3.59. There are 10 extra softmax output layers for the classification and 1 "Fake/Real" unit for the GAN. So this model has three components (Generator, Discriminator, and Classifier). Here, the discriminator acts as the classifier too. Here, the MNIST generative model is learned simultaneously with the training of the classifier. If we eliminate the generator and "Fake/Real" class in Fig. 3.59, this model is the same as a classifier model with convolution and dense layers. Regardless of the small training dataset, accuracy is still comparable with the baseline classifier. This is because the discriminator is also acting as the classifier; as a result, it is learning feature extraction during the training of the classifier and at the time of training the GAN model. As a result, this is a very data-efficient classifier. At the end of the training, we can independently use the discriminator as a classifier without the generator

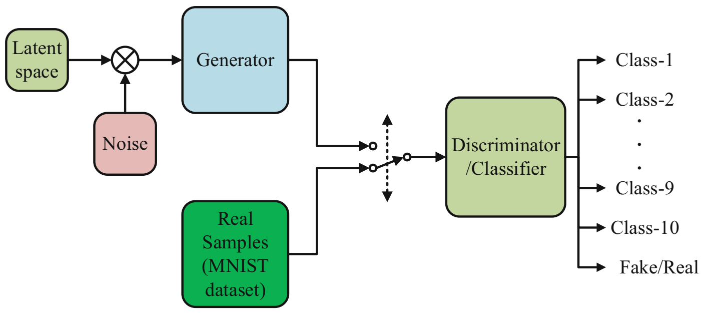


Fig. 3.59 Architecture of SGAN model

model; the generator can produce data similar to the MNIST handwritten dataset. The code for this SGAN using PyTorch has been given in Listing 3.20 and a description of the code in Table 3.34.

Programming Example 3.21
Listing 3.20 provides a code for implementing an SGAN to generate images resembling handwritten digits from the MNIST dataset. The process involves defining and training the generator and discriminator networks. The generator network utilizes a noise vector to produce an image, whereas the discriminator network takes an image and delivers a validity score (real or fake) along with a label (image class). The model is trained using binary cross-entropy loss for the adversarial component and cross-entropy loss for the auxiliary component. The accuracy of the discriminator is evaluated and printed for both authentic and generated images individually. The code is explained in Table 3.34.

```
#https://gitee.com/nj520/PyTorch-GAN/blob/master/implementations/
    sgan/sgan.py
#Paper: https://arxiv.org/abs/1606.01583
import numpy as np
import math
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torchvision import datasets
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
import torch

#Loading dataset
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,)
    ),
                                ])
dataset = datasets.MNIST('./mnist', download=True, train=True,
    transform=transform)
```

```
dataloader = torch.utils.data.DataLoader(dataset, batch_size=64,
    shuffle=True)

#Variables
latent_dim = 100
img_size = 32
num_epochs = 3
batch_size = 64
num_classes = 10

#Defining generator
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.init_size = img_size // 4
        self.l1 = nn.Sequential(nn.Linear(latent_dim, 128 * self.
    init_size ** 2))

        self.conv_blocks = nn.Sequential(
            nn.BatchNorm2d(128),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 128, 3, stride=1, padding=1),
            nn.BatchNorm2d(128, 0.8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 64, 3, stride=1, padding=1),
            nn.BatchNorm2d(64, 0.8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 1, 3, stride=1, padding=1),
            nn.Tanh()
        )

    def forward(self, noise):
        out = self.l1(noise)
        out = out.view(out.shape[0], 128, self.init_size, self.
    init_size)
        img = self.conv_blocks(out)
        return img

#Defining discriminator
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.conv_blocks = nn.Sequential(
                              nn.Conv2d(1, 16, 3, 2, 1),
                              nn.LeakyReLU(0.2, inplace=True),
                              nn.Dropout2d(0.25),

                              nn.Conv2d(16, 32, 3, 2, 1),
                              nn.LeakyReLU(0.2, inplace=True),
                              nn.Dropout2d(0.25),
                              nn.BatchNorm2d(32, 0.8),

                              nn.Conv2d(32, 64, 3, 2, 1),
                              nn.LeakyReLU(0.2, inplace=True),
                              nn.Dropout2d(0.25),
                              nn.BatchNorm2d(64, 0.8),

                              nn.Conv2d(64, 128, 3, 2, 1),
                              nn.LeakyReLU(0.2, inplace=True),
                              nn.Dropout2d(0.25),
                              nn.BatchNorm2d(128, 0.8)
                            )

        ds_size = img_size // 2 ** 4 # The height and width of
    downsampled image

        # Output layers
        self.adv_layer = nn.Sequential(nn.Linear(128 * ds_size **
     2, 1), nn.Sigmoid())
        self.aux_layer = nn.Sequential(nn.Linear(128 * ds_size **
     2, num_classes + 1), nn.Softmax())

    def forward(self, img):
        out = self.conv_blocks(img)
        out = out.view(out.shape[0], -1)
        validity = self.adv_layer(out)
        label = self.aux_layer(out)

        return validity, label

#Loss functions
adversarial_loss = torch.nn.BCELoss()
auxiliary_loss = torch.nn.CrossEntropyLoss()

#Initialize generator and discriminator
generator = Generator()
discriminator = Discriminator()

#Optimizers
optimizer_G = torch.optim.Adam(generator.parameters(), lr=0.0001,
     betas=(0.5, 0.999))
optimizer_D = torch.optim.Adam(discriminator.parameters(), lr
    =0.0001, betas=(0.5, 0.999))

#  Training
for epoch in range(num_epochs):
    for i, (imgs, labels) in enumerate(dataloader):

        batch_size = imgs.shape[0]

        # Adversarial ground truths
        valid = Variable(torch.FloatTensor(batch_size, 1).fill_
    (1.0), requires_grad=False)
```

```
        fake = Variable(torch.FloatTensor(batch_size, 1).fill_(0.0), requires_grad=False)
        fake_aux_gt = Variable(torch.LongTensor(batch_size).fill_(num_classes), requires_grad=False)

        # Configure input
        real_imgs = Variable(imgs.type(torch.FloatTensor))
        labels = Variable(labels.type(torch.LongTensor))

        ###Train Generator

        optimizer_G.zero_grad()

        # Sample noise and labels as generator input
        z = Variable(torch.FloatTensor(np.random.normal(0, 1, (batch_size, latent_dim))))

        # Generate a batch of images
        gen_imgs = generator(z)

        # Loss measures generator's ability to fool the discriminator
        validity, _ = discriminator(gen_imgs)
        g_loss = adversarial_loss(validity, valid)

        g_loss.backward()
        optimizer_G.step()

        ###Train Discriminator

        optimizer_D.zero_grad()

        # Loss for real images
        real_pred, real_aux = discriminator(real_imgs)
        d_real_loss = (adversarial_loss(real_pred, valid) + auxiliary_loss(real_aux, labels)) / 2

        # Loss for fake images
        fake_pred, fake_aux = discriminator(gen_imgs.detach())
        d_fake_loss = (adversarial_loss(fake_pred, fake) + auxiliary_loss(fake_aux, fake_aux_gt)) / 2

        # Total discriminator loss
        d_loss = (d_real_loss + d_fake_loss) / 2


        # Calculate discriminator accuracy
        pred = np.concatenate([real_aux.data.numpy(), fake_aux.data.numpy()], axis=0)
        gt = np.concatenate([labels.data.numpy(), fake_aux_gt.data.numpy()], axis=0)
        d_acc = np.mean(np.argmax(pred, axis=1) == gt)

        d_loss.backward()
        optimizer_D.step()

        # Calculate discriminator mnist accuracy
        d_acc_mnist = np.mean(np.argmax(real_aux.data.numpy(),
    axis=1) == labels.data.numpy())
        d_acc_fake = np.mean(np.argmax(fake_aux.data.numpy(),
    axis=1) == fake_aux_gt.data.numpy())

        print(
            "[Epoch %d/%d] [Batch %d/%d] [D loss_real: %f,
    loss_fake: %f, acc_mnist: %d%%, acc_fake: %d%%] [G loss: %f]"
             % (epoch, num_epochs, i, len(dataloader),
    d_real_loss.item(), d_fake_loss.item(), 100*d_acc_mnist, 100*
    d_acc_fake, g_loss.item())
        )
```

Listing 3.20 Semi-supervised GAN [72, 73]

Output of Listing 3.20:

```
[Epoch 0/3] [Batch 0/938] [D loss_real: 1.550232,
loss_fake: 1.539532, acc_mnist: 4%, acc_fake: 9%]
[G loss: 0.703786]
[Epoch 0/3] [Batch 1/938] [D loss_real: 1.550391,
loss_fake: 1.539539, acc_mnist: 4%, acc_fake: 3%]
[G loss: 0.703941]
[Epoch 0/3] [Batch 2/938] [D loss_real: 1.549383,
loss_fake: 1.539355, acc_mnist: 7%, acc_fake: 6%]
[G loss: 0.703729]
.....
.....
.....
.....
[Epoch 2/3] [Batch 932/938] [D loss_real: 0.918866,
loss_fake: 0.773814, acc_mnist: 70%, acc_fake: 100%]
[G loss: 8.777421]
[Epoch 2/3] [Batch 933/938] [D loss_real: 0.900834,
loss_fake: 0.772494, acc_mnist: 76%, acc_fake: 100%]
[G loss: 8.913239]
[Epoch 2/3] [Batch 934/938] [D loss_real: 0.943198,
loss_fake: 0.772426, acc_mnist: 68%, acc_fake: 100%]
[G loss: 8.530702]
[Epoch 2/3] [Batch 935/938] [D loss_real: 0.923197,
loss_fake: 0.776373, acc_mnist: 70%, acc_fake: 100%]
[G loss: 8.833928]
[Epoch 2/3] [Batch 936/938] [D loss_real: 0.912454,
loss_fake: 0.774332, acc_mnist: 75%, acc_fake: 100%]
[G loss: 8.576003]
```

```
[Epoch 2/3] [Batch 937/938] [D loss_real: 0.873888,
loss_fake: 0.772086, acc_mnist: 87%, acc_fake: 100%]
[G loss: 8.941827]
```

3.8.2 Semi-supervised Classification

A typical semi-supervised learning problem starts with a bunch of unlabeled data and a very small fraction of the labeled data. The training algorithm remains the same as a supervised learning algorithm (e.g., Naive Bayes or SVD). However,

Table 3.34 Explanation of the SGAN code presented in Listing 3.20

Line number	Description
3–11	Importing NumPy and PyTorch
13–18	Loading and transforming MNIST handwritten dataset for training
20–25	Defining variables such as latent dimension and the number of classes
27–53	Defining generator using convolution layer, which takes noise as input and generates an image using further up-sampling and conv layer
55–93	Defining discriminator
60–79	Regular sequential convolution layer, which determines the fake image and also the class of the image
83–85	Output layer of discriminator. First, using a sigmoid layer fake of the real image is determined, that is, the adversarial output layer, and then using auxiliary layer class is determined using the softmax layer. These two layers are independent of each other, and both take input from the sequential convolution layer
87–93	Executes outputs from the adversarial and auxiliary layers. So discriminator has two output validity and label of the input image
95–97	Loss function for the generator and the discriminator is defined
99–101	Initialize the discriminator and the generator from the predefined class
103–105	Optimizer for the discriminator and the generator is initialized
107–170	Training of the model performed here
113–116	Ground truths are initialized
119–120	Real images and corresponding labels are converted into an appropriate form
127	Noise is extracted to generate a fake image
130	Fake images are generated
133–137	Loss is calculated, and backpropagation is performed for generator
144–152	Loss for real images and fakes images; thus the total discriminator loss is calculated
156–161	Backpropagation for discriminator is performed here
164–165	Accuracy for MNIST dataset is calculated
169	Overall metrics are printed

the following changes need to be made in order to handle the unlabeled data and complete the learning process:

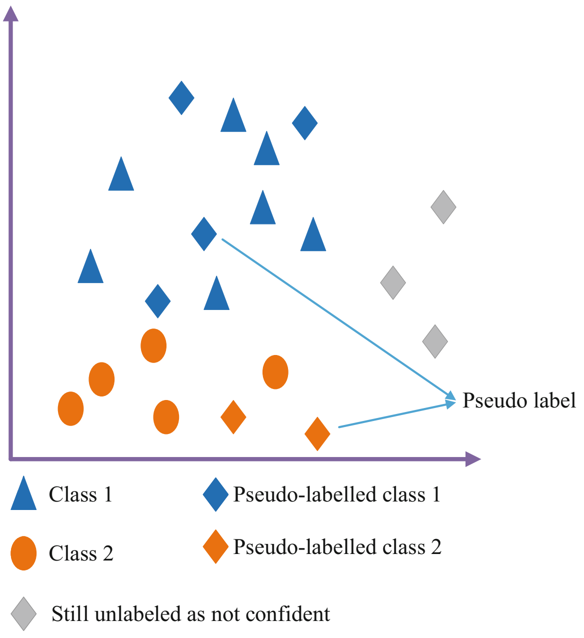
1. Train a classifier (e.g., SVM and Naive Bayes) on the labeled training data.
2. Collect the labels of the unlabeled data from the pre-trained model at the first stage.
3. Only include those unlabeled observations into the previous label dataset with a higher confidence score at step 2.
4. Use the augmented data to retrain the previous pre-trained model to complete the training.

Adopting the above procedure can turn any supervised learning into a semi-supervised learning scheme. Figure 3.60 depicts semi-supervised classification by showing how a small amount of labeled data can be used to improve the performance of a model on a large amount of unlabeled data using pseudo-labeling. Labeling the unlabeled data with the output predicted by neural networks is called *pseudo-labeling*.

3.9 Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning where the agent learns by sequential interactions at discrete time steps with the environment.

Fig. 3.60 Semi-supervised classification



Reinforcement learning has applications in the field of robotics, self-driving cars, healthcare, industry automation, natural language processing, trading finance, and robot motion control.

We have already seen that ML models are trained using some sort of labeled or unlabeled data. However, RL algorithms learn by interacting with the environment, even if there is no previous experience with that environment, so no dataset is necessary for training in RL. In the learning process, as experiences are acquired by the data (which are being fed to the agent in real time), a neural network, CNN, or RNN that we have already learned can be used for learning, though a simple gradient method cannot be used.

As described in Fig. 3.61, in RL, we have an agent that interacts with the environment by performing actions. Actions are taken based on the state of the environment. The state (frame of the video in case of a game) of the environment is fed to the agent; then, the policy network (which can be a neural network) transforms the input (state) into the output, which is an action. The policy is written as $\pi(a|s)$, which tells the probability of taking an action a under state s. In addition, the policy gradient is used to backpropagate compared to the normal gradient used in the conventional ML model. After applying the action, the current state s_t is changed into a new state s_{t+1}, and a reward (positive reward) is given to the agent if the action is right; otherwise, a penalty is given for the wrong action.

The main goal of the training in RL is to optimize the policy network using policy gradient to receive as much reward as possible across an episode. An *episode* can be defined as the sequence of interactions until the environment reaches the final state from the initial state. For example, if a robot is being trained to walk, the episode is the interactions done from the first action to the last action when the robot will walk properly (positive reward) or fall down (negative reward). After an episode, the environment is reset to a standard starting state, s_0, or to a random starting state sampled from the distribution; a positive reward is also given, and a normal policy gradient is applied to increase the probability of that action in the future. On the other hand, for a negative reward or for a penalty, the policy gradient is applied with a negative sign to reduce the probability of that action.

In Fig. 3.62, the sequence of state changes, applying an action, and gaining reward has been shown. The first action a_t is applied based on the current state

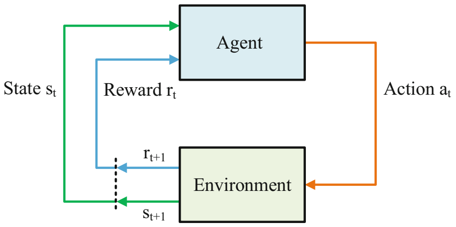


Fig. 3.61 Reinforcement learning

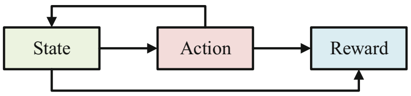


Fig. 3.62 Sequence of action, state, and reward

s_t, after which the state is changed into s_{t+1} due to the performed action a_t, and the reward r_{t+1} is given based on the final state s_{t+1}. It should be noted that r_t is the reward for the transition of the state from s_{t-1} into s_t.

Now we shall discuss some terms briefly.

1. **Policies (π) and value functions:** Policy map state from the state space to the probabilities of taking possible action for each state. On the other hand, if state and state–action pair is given, then the expected return can be extracted from the value function and can be analyzed.
2. **Optimal policy:** A policy will be considered better than the other if it has a greater expected return than the compared one for all states.
3. **Optimal state-value function:** This gives the largest expected return possible by any given policy for each state. Mathematically an optimal state-value function is given by

$$v_*(s) = \max_\pi v_\pi(s). \tag{3.76}$$

4. **Optimal Action-Value Function:** This is the optimal Q-value function that gives the largest expected return by any policy for each state–action pair. Mathematically an optimal Q-function is given by

$$q_*(s,a) = \max_\pi q_\pi(s,a). \tag{3.77}$$

5. **Bellman Optimality Equation:** q_* must fulfil the Bellman Optimality equation:

$$q_*(s,a) = E\left[R_{t+1} + \gamma \max_{a'} q_*\left(s', a'\right)\right]. \tag{3.78}$$

   The expected reward from starting in state s with executing action a while following optimum policy is the expected reward for action pair (s,a), which is R_{t+1} plus max possible discounted return which can be found from the next action pair (s', a').
6. **Exploration:** In exploration methods, the agents try to explore different actions to accumulate knowledge about each possible action rather than accumulating rewards through repeating certain actions over and over again. Exploration techniques tend to find the best overall action or decision.
7. **Exploitation:** In exploitation methods, the agents try to accumulate as many rewards as possible by utilizing already known actions. Exploitation techniques tend to find the best action or decision based on current knowledge.
8. **Regret:** We humans tend to regret our decision when it fails to achieve the desired outcome. The same goes for RL. Regret can be seen as a difference between the rewards accumulated through the actions taken and the highest possible accumulated rewards if the most optimal action was taken.

If an action a is taken, then the action value will be the mean reward for action a, that is,

$$Q(a) = E[r|a]. \tag{3.79}$$

The optimal action value will be found from optimal action a^*:

$$V^* = Q(a^*) = \max_{a \in A} Q(a). \tag{3.80}$$

Then the regret for one particular step t is

$$l_t = E[V^* - Q(a_t)]. \tag{3.81}$$

And the total regret is

$$l_t = E \sum_{\tau=1}^{t} [V^* - Q(a_t)]. \tag{3.82}$$

Regret may be considered analogous to the loss function. More rewards can be accumulated by minimizing regret.

3.9.1 Multi-armed Bandit Problem

A famous problem in the ML domain, specifically in the RL domain, is the *multi-armed bandit problem*, also known as the *k-armed bandit problem*. This problem represents the *exploration vs. exploitation dilemma* in RL very well.

Suppose you live in a big city with lots of restaurants. You have visited some of the restaurants, but not all. Of all the restaurants you have visited, you get the most satisfaction from a few of them. To achieve maximum satisfaction, you can go to a restaurant you already like or visit a new one with the hope that they can do better. This dilemma is illustrated in Fig. 3.63. Replace the city, restaurant visits, yourself, and your satisfaction with the environment, actions, agent, and rewards, respectively, and then you get the formal statement of a multi-armed bandit problem. Here, the term *bandit* is derived from casino slot machines with one arm or one lever, except this problem deals with k levers.

The multi-armed bandit can be expressed as a tuple—$\langle A, R \rangle$, where A is the set of k actions or arms, and R is the set of rewards. Now, the unknown probability distribution over the set of rewards will be given by the equation:

$$R_r^a = P\left[\frac{r}{a}\right]. \tag{3.83}$$

Here r and a are individual rewards and actions where $r \in R$ and $a \in A$. For a particular step t, the agent selects an action a_t. For this action, the accumulated

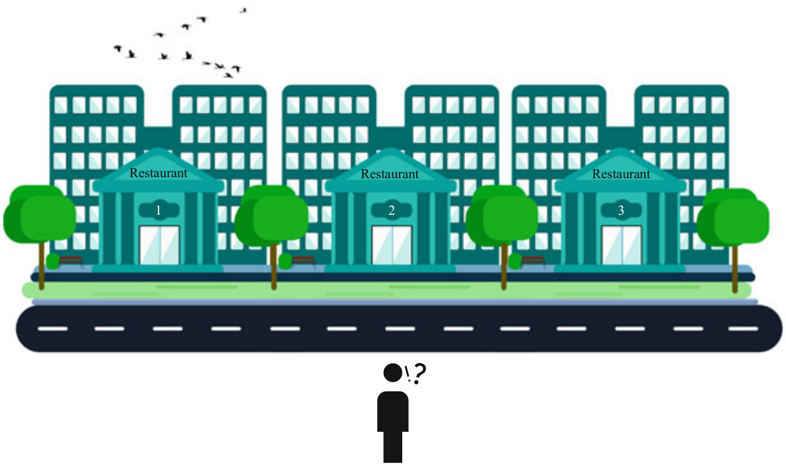


Fig. 3.63 Visual representation of the restaurant dilemma example of the multi-armed bandit problem

reward from the environment will be r_t. Here, $a_t \in A$ and $r_t \in R$. The cumulative reward will then become $\sum_{\tau=1}^{t} r_t$.

The purpose of the agent (yourself) is to maximize the amount of cumulative rewards (maximum satisfaction). In RL, there are various techniques to solve this problem. Some of these will be briefly discussed in the following sections.

3.9.1.1 The Greedy Strategy

This is a simple action-value strategy. The aim is to select the action with the highest reward. This action is known as the greedy action. If there are multiple greedy actions, then the selection is done randomly. The greedy action can be estimated as follows:

$$a_t^* = \underset{a \in A}{\arg\max}\, Q_t(a). \tag{3.84}$$

Although the greedy algorithm focuses on accumulating as many rewards as possible, it has a tendency to stick to the sub-optimal action.

3.9.1.2 The Epsilon ($\in$)-Greedy Strategy

The greedy strategy can miss the most optimal action. A solution to this problem can be to explore a bit with a certain probability, say $\in$, every once in a while and behave greedily for the rest of the time (with $1-\in$ probability).

In short, select $a_t = \arg\max_{a \in A} Q_t(a)$ with probability $1-\in$ and then select a random action with probability $\in$. In this case, the probability of obtaining the optimal action should be near certainty (greater than $1-\in$).

3.9.1.3 Upper Confidence Bound (UCB)

This is a non-greedy method, i.e., this method does not exclusively focus on the current known action. In UCB, the action is selected based on the optimal action estimation and the uncertainty of that selection. This can be achieved from the following equation:

$$a_t = \underset{a \in A}{\arg\max} \left[Q_t(a) + c\sqrt{\frac{ln(t)}{N_t(a)}} \right], \tag{3.85}$$

where $N_t(a)$ is the number of actions a taken prior to time t, and c ($c > 0$) is the degree of exploration. The squared-root term in Eq. 3.85 determines the level of uncertainty of the action [74]. As the name suggests, UCB focuses on exploration and exploitation based on a confidence boundary. The agent optimistically chooses the actions in case of uncertainty that has the highest upper bound. This boundary decreases as it explores more and more.

3.9.1.4 Thompson Sampling

Thomson sampling utilizes the Bayes theorem to determine the reward distribution of various arms of the multi-armed bandit. The posterior distribution of rewards is determined from the Bayes theorem as follows [75]:

$$P(R|h_t) = \frac{P(h_t|R)P(h_t)}{P(R)}, \tag{3.86}$$

where $P(R|h_t)$ is the posterior distribution, $P(h_t|R)$ is the likelihood of obtaining R given h_t, and $P(h_t)$ indicates the prior belief on h_t distribution.

The reward distribution R is sampled from the posterior distribution. Then the action value becomes $Q(a) = E[r|a]$, where $r \in R$. Then the action a_t is selected based on this sampled value:

$$a_t = \underset{a \in A}{\arg\max}\, Q(a). \tag{3.87}$$

Thus, this method selects action by making trade-offs between exploration and exploitation to maximize the reward accumulation.

3.9.1.5 Q-Learning

The Q-learning technique can be used to learn the optimal policy we already discussed in the previous section. After selecting the optimal Q-function, the RL algorithm is applied to determine the optimal policy to find the action that maximizes the Q-function for each state.

The Q-function takes input from a state and an action for a given policy and returns the expected return, for which Eq. 3.78 can be used. For Q-learning, at first, the *Q-table* (shown in Fig. 3.64) is initialized. Its dimension is found by the

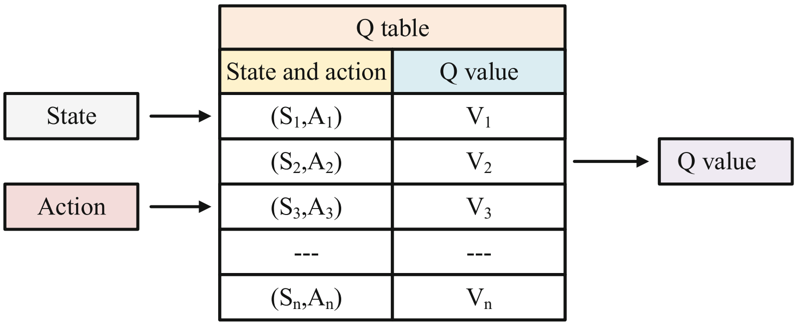
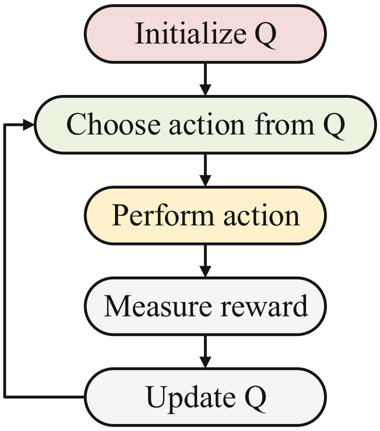


Fig. 3.64 The Q-table used in Q-learning

Fig. 3.65 The steps for training the Q-learning algorithm



number of actions and the number of states. Normally it is initialized with zeros, and after each episode, the Q-table is updated after taking action and according to the reward. The formula for determining the new Q-value for a state–action pair (s, a) combination at a given point in time t is given by Eq. 3.88.

$$q^{\text{new}}(s,a) = (1-\alpha)\underbrace{q(s,a)}_{\text{old value}} + \alpha \overbrace{\left(R_{t+1} + \gamma \times \underbrace{\max_{a'} q\left(s', a'\right)}_{\text{max estimate of future value}}\right)}^{\text{learned value}}. \tag{3.88}$$

Here, α denotes the learning rate, R_{t+1} stands for reward, and γ stands for discount rate.

For taking action, the agent randomly selects exploration and exploitation during the training in order to make a trade-off between them. The typical steps for training using the Q-learning algorithm are given in Fig. 3.65.

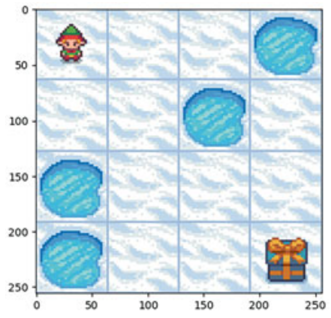


Fig. 3.66 The map visualized in Listing 3.21

Here, the new Q-value is the summation of the weighted old value and the new learned value. This topic will be clarified in the following example.

Programming Example 3.22
We will demonstrate the Q-learning algorithm using the "FrozenLake" environment from "Open AI Gymnasium" [76]. The reader is advised to read the environment description from the given reference. In this environment, the start point is marked with a "Player Character," and the final point is pointed with a "Gift Box." The agent can move left, right, up, and down (Fig. 3.66). The goal is to reach the gift without falling into any hole. If the agent can reach the goal traversing the frozen surface, a reward of +1 is given, and if the agent falls into a hole, there is 0 reward. After both cases, the game is over, and a new game starts from the starting point.

The Python code for this problem is provided in Listing 3.21 with its output following it (Fig. 3.67). Table 3.35 explains the code.

```
# ----------------------Importing Libraries----------------------
import numpy as np
import gymnasium as gym
import random
from IPython.display import clear_output
import matplotlib.pyplot as plt
from collections.abc import Sequence


# ----------------------Initiate Environment---------------------
custom_map = ["SFFH",
              "FFHF",
              "HFFF",
              "HFFG"]

env_gym = gym.make('FrozenLake-v1', desc=custom_map, render_mode=
    "rgb_array")

env_gym.reset()
plt.imshow(env_gym.render())
```

```


# --------------------------Parameters---------------------------
# Generating Q-table
a_size = env_gym.action_space.n
s_size = env_gym.observation_space.n
# Initializing Q-table with zero
Q_table = np.zeros((s_size, a_size))

# Total number of episodes
num_episodes = 9000
# Maximum number of steps agent is allowed to take within a
    single episode
maximum_step_each_episode = 90

learn_rate = 0.1
discount_rate = 0.99

exploration_rate = 1
max_exploration_rate = 1
min_exploration_rate = 0.01
exploration_decay_rate = 0.001

def check_state(state):
    if isinstance(state, Sequence):
        return state[0]
    else:
        return state


# ---------------------------Training----------------------------
# List to hold reward from each episodes
all_rewards = []

# Q-learning algorithm
for episode in range(num_episodes):
    # Initialize new episode params
    state = env_gym.reset()
    state = check_state(state)
    # Done parameter keep to keep track of episode when finish
    finished_flag = False
    rewards_current_episode = 0

    # For each time step within this episode
    for step in range(maximum_step_each_episode):

        # Choosing between exploration and exploitation
        exploration_rate_threshold = random.uniform(0, 1)
        if exploration_rate_threshold > exploration_rate:
            action = np.argmax(Q_table[state, :])
        else:
            action = env_gym.action_space.sample()

```

```python
        outs = env_gym.step(action)
        if len(outs) == 4:
            observation, reward, finished_flag, _ = env_gym.step(action)
        else:
            observation, reward, terminated, truncated, _ = env_gym.step(action)
            finished_flag = terminated

        # Update Q-table for Q(s,a)
        Q_table[state, action] = Q_table[state, action] * (1 - learn_rate) + learn_rate * (
                reward + discount_rate * np.max(Q_table[observation, :]))

        state = observation
        rewards_current_episode += reward

        if finished_flag == True:
            break

    # Exploration rate decay using exponential decay
    exploration_rate = min_exploration_rate + (max_exploration_rate - min_exploration_rate) * np.exp(
        -exploration_decay_rate * episode)

    all_rewards.append(rewards_current_episode)

# Calculate and print the average reward per thousand episodes
rewards_per_thousand_episodes = np.split(np.array(all_rewards), num_episodes / 1000)
count = 1000

print("Average reward summary:")
for r in rewards_per_thousand_episodes:
    print(count, ": ", str(sum(r / 1000)))
    count += 1000

# Updated Q-table
print("Updated Q-table:")
print(Q_table)



# -----------------Iterations of agent playing-------------------
# Watching the agent playing with best actions from the Q-table
for episode in range(4):
    state = env_gym.reset()
    state = check_state(state)
    finished_flag = False
    print("============================================")
    print("EPISODE: ", episode+1)

    for step in range(maximum_step_each_episode):

        action = np.argmax(Q_table[state, :])
        outs = env_gym.step(action)

        if len(outs) == 4:
            observation, reward, finished_flag, _ = env_gym.step(
    action)
        else:
            observation, reward, terminated, truncated, _ =
    env_gym.step(action)
            finished_flag = terminated

        if finished_flag:
            plt.imshow(env_gym.render())
            if reward == 1:
                print("The agent reached the Goal")
            else:
                print("The agent fell into a hole")

            print("Number of steps", step)

            break

        state = observation

env_gym.close()
```

Listing 3.21 Reinforcement learning using OpenAI frozen lake environment [77]

Output after Line 19 of Listing 3.21:
Output after Line 102 of Listing 3.21:

```
Average reward summary:
1000 :  0.04900000000000004
2000 :  0.20200000000000015
3000 :  0.3820000000000003
4000 :  0.5420000000000004
5000 :  0.6120000000000004
6000 :  0.6680000000000005
7000 :  0.6570000000000005
8000 :  0.6500000000000005
9000 :  0.6680000000000005
```

Output after Line 106 of Listing 3.21:

```
Updated Q-table:
[[5.50579416e-02 5.47402216e-02 5.23440673e-02 5.33719782e-02]
 [7.92652688e-02 6.57612005e-02 6.42740679e-02 6.91404070e-02]
 [1.82775052e-02 1.92826368e-02 3.80034093e-04 1.98099025e-02]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [4.46883633e-02 2.93065726e-02 4.40962298e-02 4.92429277e-02]
 [6.88198255e-02 3.57139646e-02 5.30218450e-02 4.51624282e-02]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [6.21450787e-02 0.00000000e+00 4.56000428e-03 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [9.10521593e-02 8.81886916e-02 1.57974619e-01 6.67273365e-02]
 [3.20439214e-02 4.18211567e-01 2.72005863e-01 1.94268929e-01]
 [7.85680685e-02 1.65205077e-01 1.21904562e-01 1.74734821e-02]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [7.67063001e-02 2.14511902e-01 3.95164778e-01 1.89810501e-01]
 [1.91633412e-01 8.17657490e-02 8.67790192e-02 5.06268910e-02]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]]
```

Output after Line 143 of Listing 3.21:

```
============================================
EPISODE:  1
The agent fell into a hole
Number of steps 6
============================================
EPISODE:  2
The agent fell into a hole
Number of steps 2
============================================
EPISODE:  3
The agent fell into a hole
Number of steps 11
============================================
EPISODE:  4
The agent reached the Goal
Number of steps 16
```

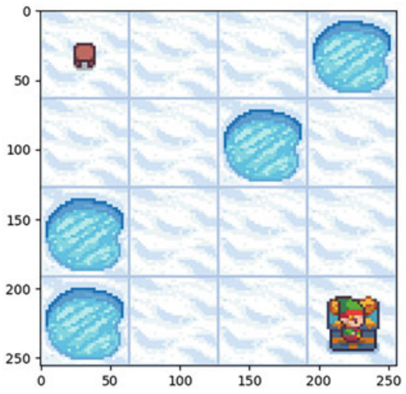


Fig. 3.67 Output of Listing 3.21

Table 3.35 Explanation of the Q-learning example using frozen lake environment presented in Listing 3.21

Line number	Description
1–7	Importing necessary libraries and modules
10–19	Define a custom environment and visualize the map
23–40	Set various parameters for Q-learning
42–46	A function to keep state value always as an integer
50–106	Q-learning algorithm to train the agent
51	List to hold rewards from each episode
54–93	Loop for training the Q-table
66–70	Decide whether to explore or exploit based on the exploration rate
72–81	Perform an environment step and update Q-table using Eq. 3.88
90	Decay exploration rate over episodes
100–102	Calculate and print average reward per 1000 episodes
106	Print the updated Q-table
111–143	Watch the agent play using the Q-table
130–137	Display the environment and outcomes
180	Close the environment

3.10 Conclusion

This chapter introduces the readers to different types of algorithms used in machine learning. At first, the different types of datasets and dataset preprocessing strategies are discussed. Then, a large section is dedicated to supervised learning, which includes regression and classification techniques with relevant examples. Then comes a section on deep learning, which includes important concepts such as gradient descent and backpropagation, artificial neural networks, convolutional neural networks, recurrent neural networks, and generative adversarial networks. The chapter also discusses time series forecasting techniques, such as ARIMA, SARIMA, and LSTM. Next, the chapter covers unsupervised learning (clustering, dimensionality reduction, and association learning), semi-supervised learning, and reinforcement learning as well. The topics are explained briefly, and some examples are implemented in Python using built-in libraries. Overall, this chapter provides a decent familiarity with machine learning algorithms and demonstrates numerous programming examples that will help the readers to clearly understand the practical applications of the theories. In the next chapter, we will study machine learning applications in signal processing.

3.11 Key Messages from This Chapter

- In supervised learning, ML models are explicitly given the features and target labels for them to learn and train.

- Data without labeling require unsupervised learning. Without any manual intervention, ML models pick up the critical patterns within data.
- Semi-supervised learning is used when we have a limited amount of labeled data. The dataset consists of both labeled and unlabeled data.
- When the size of the dataset becomes massive and computations become more complex and time-consuming, deep learning is preferred over machine learning.

3.12 Exercise

1. How are input data handled before implementing ML algorithms? Mention the steps.
2. Why is data augmentation required, and how is it implemented?
3. The datasets often contain missing data. How can this problem be addressed?
4. What do stationary and non-stationary time series signify? Differentiate between them using an example.
5. Why do we need different types of ML algorithms? Briefly discuss.
6. Discuss the relationship between deep learning and neural networks. What are the advantages of deep learning methods?
7. Briefly describe the learning process of a single neuron and develop a simple neural network structure using multiple neurons.
8. How is a CNN architecture built? Briefly discuss some state-of-the-art CNN architectures.
9. Describe the differences between the following:
   (a) Supervised, semi-supervised, and unsupervised algorithm
   (b) CNN and RNN
   (c) Classification, regression, and clustering
10. Briefly explain the mechanism of reinforcement learning.
11. Every year, a massive amount of CO_2 is emitted from electricity generation worldwide. To reduce CO_2 emissions and plan our energy strategy accordingly, it is essential to gather an idea about future CO_2 emissions. Therefore, develop a 10-year CO_2 emission forecasting model. The dataset and its description are available here: https://www.kaggle.com/datasets/txtrouble/carbon-emissions.
12. Perform a multiclass classification on the iris dataset [78] using the Random Forest classifier. Load the iris dataset using the scikit-learn library as follows:

```python
from sklearn import datasets

iris = datasets.load_iris()
X = iris.data
y = iris.target

```

13. Develop a multiclass classifier for the MNIST dataset. Implement the same classifier for the Fashion MNIST dataset. Compare the results.

14. Implement the IMDB movie rating sentiment analysis using LSTM and bidirectional LSTM. Which one performs better and why?
15. Implement Q-learning algorithm in the CartPole environment. The CartPole environment is available on OpenAI-gym. A description of this environment is available on their website: https://gymnasium.farama.org/environments/classic_control/cart_pole/
16. Using the golf playing dataset from Table 3.6, construct a decision tree using Gain Ratio as the attribute selection measure.

References

1. *Working with missing data*. https://pandas.pydata.org/docs/user_guide/missing_data.html#
2. Pawara, P., Okafor, E., Schomaker, L., & Wiering, M. (2017). Data augmentation for plant classification. In *Advanced concepts for intelligent vision systems* (pp. 615–626). Springer International Publishing.
3. Efron, B., Hastie, T., Johnstone, I., & Tibshirani, R. (2004). Least angle regression. *The Annals of Statistics, 32*(2), 407–499.
4. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research, 12*, 2825–2830.
5. Quinlan, J. R. (986). Induction of decision trees. *Machine Learning, 1*(1), 81–106.
6. noushin.gauhar. *Constructing a decision tree: Entropy & Information gain - Decision Tree - Learn with Gauhar*—learnwithgauhar.com. https://learnwithgauhar.com/constructing-a-decision-tree-entropy-information-gain/. Accessed September 07, 2023.
7. *Classifier comparison*—scikit-learn.org. https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html. Accessed September 07, 2023.
8. *MNIST dataset*. https://www.tensorflow.org/datasets/catalog/mnist
9. Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE, 86*(11), 2278–2324.
10. Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted Boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10).*
11. He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proceedings of the IEEE International Conference on Computer Vision* (pp. 1026–1034).
12. Clevert, D.-A., Unterthiner, T., & Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (ELUs). *Preprint arXiv:1511.07289.*
13. Hendrycks, D., & Gimpel, K. (2016). Gaussian error linear units (GELUs). *Preprint arXiv:1606.08415.*
14. *Examples/mnist/main.py at main · PyTorch/examples*—github.com. https://github.com/pytorch/examples/blob/main/mnist/main.py. Accessed September 07, 2023.
15. Khan, A., Sohail, A., Zahoora, U., & Qureshi, A. S. (2020). A survey of the recent architectures of deep convolutional neural networks. *Artificial Intelligence Review, 53*(8), 5455–5516.
16. LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE, 86*(11), 2278–2324.
17. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems, 25*, 1097–1105.
18. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 1–9).

19. Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *Preprint arXiv:1409.1556*.
20. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770–778).
21. Robertson, S. (2017). *NLP from scratch: Classifying names with a character-level RNN*. https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html
22. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2020). Generative adversarial networks. *Communications of the ACM, 63*(11), 139–144.
23. Pan, Z., Yu, W., Yi, X., Khan, A., Yuan, F., & Zheng, Y. (2019). Recent progress on generative adversarial networks (GANs): A survey. *IEEE Access, 7*, 36322–36333.
24. Prabhakaran, S. *Arima model – complete guide to time series forecasting in python*. https://www.machinelearningplus.com/time-series/arima-model-time-series-forecasting-python/
25. Athanasopoulos, G., & Hyndman, R. J. *Non-seasonal ARIMA models*. https://otexts.com/fpp2/non-seasonal-arima.html
26. Athanasopoulos, G., & Hyndman, R. J. *Seasonal ARIMA models*. https://otexts.com/fpp2/seasonal-arima.html
27. Pathak, P. *How to create an ARIMA model for time series forecasting in python*. https://www.analyticsvidhya.com/blog/2020/10/how-to-create-an-arima-model-for-time-series-forecasting-in-python/
28. Palachy, S. *Stationarity in time series analysis*. https://towardsdatascience.com/stationarity-in-time-series-analysis-90c94f27322
29. Brownlee, J. (2018). *A gentle introduction to SARIMA for time series forecasting in Python*. https://machinelearningmastery.com/sarima-for-time-series-forecasting-in-python/
30. Sarkar, S. *Time series forecasting and analysis: ARIMA and seasonal-ARIMA*. https://medium.com/analytics-vidhya/time-series-forecasting-and-analysis-arima-and-seasonal-arima-cacaf61ae863
31. Hochreiter, S., & Schmidhuber, J. (1997). *Long short-term memory*. https://direct.mit.edu/neco/article/9/8/1735-1780/6109. November 1997.
32. Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011). Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, Portland, Oregon, USA, June 2011 (pp. 142–150). Association for Computational Linguistics.
33. Olah, C. *Understanding lstm networks*. https://colah.github.io/posts/2015-08-Understanding-LSTMs/
34. TensorFlow. *Text classification with an RNN*. https://www.tensorflow.org/text/tutorials/text_classification_rnn/
35. Priy, S. *Clustering in machine learning*. https://www.geeksforgeeks.org/clustering-in-machine-learning/
36. Google Developers. *What is clustering?* https://developers.google.com/machine-learning/clustering/overview
37. Google Developers. *k-means advantages and disadvantages*. https://developers.google.com/machine-learning/clustering/algorithm/advantages-disadvantages/
38. Sckit-learn. *Clustering*. https://scikit-learn.org/stable/modules/clustering.html#k-means
39. Simplilearn. *K-means clustering algorithm: Applications, types, and how does it work*. https://www.simplilearn.com/tutorials/machine-learning-tutorial/k-means-clustering-algorithm/
40. *Demonstration of k-means assumptions*—scikit-learn.org. https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_assumptions.html#sphx-glr-auto-examples-cluster-plot-kmeans-assumptions-py. Accessed September 10, 2023.
41. Maklin, C. *Affinity propagation algorithm explained*. https://towardsdatascience.com/unsupervised-machine-learning-affinity-propagation-algorithm-explained-d1fef85f22c8/
42. Sckit-Learn. *Affinity propagation*. https://scikit-learn.org/stable/modules/clustering.html#affinity-propagation/

43. *Demo of affinity propagation clustering algorithm*—scikit-learn.org. https://scikit-learn.org/stable/auto_examples/cluster/plot_affinity_propagation.html#sphx-glr-auto-examples-cluster-plot-affinity-propagation-py. Accessed September 07, 2023.
44. Niebles, J. C., & Krishna, R. (2016). *K-means & mean-shift clustering - Stanford university*.
45. Comaniciu, D., Meer, P. (2002). Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 24*(5), 603–619.
46. *A demo of the mean-shift clustering algorithm*—scikit-learn.org. https://scikit-learn.org/stable/auto_examples/cluster/plot_mean_shift.html. Accessed September 07, 2023.
47. Sckit-learn. *Dbscan*. https://scikit-learn.org/stable/modules/clustering.html#dbscan
48. Dey, D. *Dbscan clustering in ml | density based clustering*. https://www.geeksforgeeks.org/dbscan-clustering-in-ml-density-based-clustering/
49. Dobilas, S. *Dbscan clustering algorithm — how to build powerful density-based models*. https://towardsdatascience.com/dbscan-clustering-algorithm-how-to-build-powerful-density-based-models-21d9961c4cec/
50. *Demo of DBSCAN clustering algorithm*—scikit-learn.org. https://scikit-learn.org/stable/auto_examples/cluster/plot_dbscan.html#sphx-glr-auto-examples-cluster-plot-dbscan-py. Accessed September 09, 2023.
51. GeeksforGeeks. *Introduction to dimensionality reduction*. https://www.geeksforgeeks.org/dimensionality-reduction/
52. Kramer, O. (2013). *Dimensionality reduction with unsupervised nearest neighbors* (1st ed.). Springer.
53. Wang, J. (2012). *Geometric structure of high-dimensional data and dimensionality reduction* (1st ed.). Springer.
54. Scikit-learn. *Principal component analysis*. https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html
55. Jaadi, Z. *A step-by-step explanation of principal component analysis (PCA)*. https://builtin.com/data-science/step-step-explanation-principal-component-analysis
56. Brems, M. *A one-stop shop for principal component analysis*. https://towardsdatascience.com/a-one-stop-shop-for-principal-component-analysis-5582fb7e0a9c
57. Cadima, J., & Jolliffe, I. T. (2016). Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society A, 374*, 20150202.
58. Ali, A. *Dimensionality reduction(PCA and LDA) with practical implementation*. https://medium.com/machine-learning-researcher/dimensionality-reduction-pca-and-lda-6be91734f567
59. Brownlee, J. *Linear discriminant analysis for dimensionality reduction in python*. https://machinelearningmastery.com/linear-discriminant-analysis-for-dimensionality-reduction-in-python/
60. Nelson, D. *Dimensionality reduction in python with scikit-learn*. https://stackabuse.com/dimensionality-reduction-in-python-with-scikit-learn/
61. Kumar, V. *Practical approach to dimensionality reduction using PCA, LDA and Kernel PCA*. https://analyticsindiamag.com/practical-approach-to-dimensionality-reduction-using-pca-lda-and-kernel-pca/
62. Scikit-learn. *Linear discriminant analysis*. https://scikit-learn.org/stable/modules/generated/sklearn.discriminant_analysis.LinearDiscriminantAnalysis.html
63. Mahendru, K. *Master dimensionality reduction with these 5 must-know applications of singular value decomposition (SVD) in data science*. https://www.analyticsvidhya.com/blog/2019/08/5-applications-singular-value-decomposition-svd-data-science/
64. Putalapattu, R. *Montecarlo calculation of Pi*. https://github.com/rameshputalapattu/jupyterexplore/blob/master/jupyter_interactive_environment_exploration.ipynb
65. Putalapattu, R. *Jupyter, python, image compression and SVD — an interactive exploration*. https://medium.com/@rameshputalapattu/jupyter-python-image-compression-and-svd-an-interactive-exploration-703c953e44f6
66. Unsplash (2017). *Photo by Alex Knight on Unsplash*—unsplash.com. https://unsplash.com/photos/2EJCSULRwC8. Accessed September 09, 2023.

67. Garg, A. *Complete guide to association rules (1/2)*. https://towardsdatascience.com/association-rules-2-aa9a77241654/
68. Garg, A. *Complete guide to association rules (2/2)*. https://towardsdatascience.com/complete-guide-to-association-rules-2-2-c92072b56c84/
69. Umredkar, R. K. *Guide to association rule mining from scratch*. https://analyticsindiamag.com/guide-to-association-rule-mining-from-scratch/
70. Venkataramanan, V. *Breadbasket dataset*. https://github.com/viktree/curly-octo-chainsaw/blob/master/BreadBasket_DMS.csv/
71. Zhang, C., Tian, P., Zhang, X., Liao, Q., Jiang, Z. L., & Wang, X. (2019). HashEclat: An efficient frequent itemset algorithm. *International Journal of Machine Learning and Cybernetics, 10*(11), 3003–3016.
72. Odena, A. *Semi-supervised GAN*. https://gitee.com/nj520/PyTorch-GAN/blob/master/implementations/sgan/sgan.py.
73. Odena, A. (2016). *Semi-supervised learning with generative adversarial networks*. https://arxiv.org/abs/1606.01583
74. Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT Press.
75. Agrawal, S., & Goyal, N. (2012). Analysis of Thompson sampling for the multi-armed bandit problem. In *Conference on Learning Theory* (pp. 39–1). JMLR Workshop and Conference Proceedings.
76. OpenAI. *Frozenlake-v1 environment for reinforcement learning*. https://gymnasium.farama.org/environments/toy_text/frozen_lake
77. Deeplizard. *OpenAI gym and python set up for q-learning*. https://deeplizard.com/learn/video/QK_PP_2KgGE.
78. *Iris Species*. https://www.kaggle.com/datasets/uciml/iris

4 Applications of Machine Learning: Signal Processing

4.1 Introduction

Information from the real world is represented via signals, and the analysis, synthesis, and modification of these signals are called signal processing. Fast and accurate feature extraction, pattern recognition, and data interpretation capabilities of machine learning models have made machine learning a lucrative tool for signal processing. Image processing, image classification, natural language processing, adversarial input attack, etc., are all applications of machine learning and deep learning for signal processing.

Deep learning is a subset of machine learning in which algorithms are organized in layers to build a *artificial neural network* that can learn and make decisions independently. In deep learning, a model learns to execute *classification tasks* directly from texts, images, or sound. It is based on the neural network architecture. *Signal processing* is an area of electrical engineering that deals with modeling and analyzing data representations of physical occurrences. Images, movies, music, and sensor data are all examples of signals. Signal data of fine standards are difficult to derive, and there is a lot of noise and fluctuation in it. Machine learning is well-suited to the limitations and solutions demanded by signal processing issues. *Image classification* is one of the most dominant aspects where deep learning excels. The task of assigning a label to an image is called *image classification*. For increased accuracy, deep learning involves a *convolutional neural network (CNN)* for image processing. In order to have a deeper understanding of how machines look at and label an image using deep learning, we need to explore CNN profoundly. This chapter will further discuss the application of machine learning in different spheres of our lives. Here, we talk about seven different ML applications: image classification, neural style transfer, anomaly detection, adversarial input attack, malicious input detection, and natural language processing.

E. Hossain, *Machine Learning Crash Course for Engineers*,
https://doi.org/10.1007/978-3-031-46990-9_4

4.2 Signal and Signal Processing

We perceive our surrounding environment through our senses—vision, touch, hearing, etc. Similarly, various types of devices we use today for different purposes interact with us and the surrounding environment. Nevertheless, how does this interaction occur? An agent—whether a human, an animal, or any device—acts on the decisions made in their brain, i.e., the processing unit. The decisions are made based on different necessary information gathered from the environment, which comes through a form known as a signal. Theoretically, *a signal can be considered a mathematical function that depends on one or more variables carrying information about a particular phenomenon*. From a practical perspective, we can think of a signal as a *quantity that varies with respect to time, distance, temperature, or any other quantity, conveying specific information*. A signal should be measurable and sensible—manually or with the help of a machine. Radio signals, temperature, EEG, speech, image, voltage, and current are commonly known signals.

A signal can be very simple, such as a plain sinusoidal wave. However, most of the natural signals are way more complex. We can think of these signals as a combination of many sinusoidal signals with different amplitudes and frequencies. If we want to find the individual sinusoidal signals that constitute the original signal, we can perform *Fourier transform* on that particular signal. This mathematical operation is an instance of *signal processing*. Of course, numerous other types of signal processing methods are available besides the Fourier transform, such as wavelet transform, data compression, modulation, and digital signal processing. In short, *the mathematical operations we perform on signals to get insight into the signals and uncover information that is otherwise impossible to gather from direct observation of the signals are known as signal processing*. Signal processing is done for signal analysis, optimization, and denoising purposes.

Real-world signals are not ideal signals, often complex, and contain noise. Gathering information from these signals can be a very complex and time-consuming process. In our current state of technology, we produce a huge amount of data every moment. So, different types of signals are abundant, be it images, speech, or others. We can develop efficient machine learning algorithms and models to find the hidden structures or patterns from these signals, where we can get the necessary information. We can automate and perform complex signal processing operations if we implement machine learning for signal processing tasks.

4.3 Image Classification

When we look at a car, a cat, an apple, or a cloud, our brain instantly recognizes it from past exposure to it. The human brain can recall from memory and does not have a problem recognizing objects as they are. We do not need to deeply analyze objects in order to identify them. However, a machine cannot recognize an object as effortlessly as a human does. But machine learning gives machines the ability to identify images, too.

If this process of identifying and labeling the object of the image is done with computer analysis, it is called *image classification*, which is a prominent application of deep learning [1]. A number of classes or labels are given, and the computer assigns one of the classes to the image. For example, the computer does not give a direct output, whether the image is of a cat or a dog. Instead, it gives a probability that the image may be 80% cat and 20% dog. The class of the image is determined with the higher probability value.

In this section, we will learn how image classification works, its applications and challenges, and one worked-out example of an image classification problem using ML.

4.3.1 Image Classification Workflow

The workflow of image classification can be broken down into five steps, as shown in Fig. 4.1. The steps are briefly described below:

1. **Data collection:** The first and foremost step is to define the problem and define the type of dataset required. What kind of image is required? How will the data be collected? How much data would be collected? Images can be collected from online sources using *web scrapping methods*. Besides, raw image datasets can be made using cameras or drones based on domain requirements. Many established datasets can also be used if they fulfill the domain-specific requirements.
2. **Data augmentation:** The concept of data augmentation has been discussed in Sect. 3.2.2.3 of Chap. 3. One crucial way to overcome image classification challenges is to create a dataset with all possible scenarios. The model becomes robust if we can create and train the model with a very diverse dataset. Furthermore, with a diverse dataset that includes all possible cases, the model becomes more capable of detecting all possible cases.
3. **Building and training model:** Generally, CNN is used for image classification. The concept of CNN has been discussed in Sect. 3.5.4 of Chap. 3. Building a

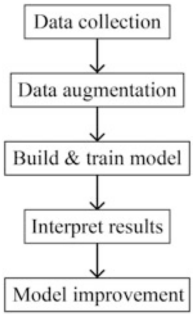


Fig. 4.1 Workflow of image classification

CNN model is comprehensible using Keras or PyTorch APIs. Once the dataset has been built, it needs to be trained on the dataset. For this purpose, the dataset is split into train, validation, and test sets. The train set is used to train the model, the validation set is used for hyperparameter tuning, and the test set is used to evaluate the model.

4. **Interpreting results:** Comprehensibility of the results is crucial for image classification. It does not matter what the result is if it produces no meaningful interpretation. Determining and defining the metrics used to interpret the results is essential. To gain insights into the data, we can use the *confusion matrix* visualization and a dependent performance variable to improve. For example, we can learn more about what kinds of images are under-represented when characterizing the class by visually representing incorrectly categorized cases.
5. **Model improvement:** Based on the evaluation of the results, the model can be tweaked and trained again to create better results. The dataset may even be tweaked to create diverse and balanced data representing all possible cases.

4.3.2 Applications of Image Classification

Image classification has significant applications in almost every field. Some of these applications are concisely discussed below to give a picture of the scope of image classification. In addition, researchers are finding more innovative and valuable scopes of image classification in various fields:

1. **Autonomous Cars:** An autonomous car uses image classification to identify other vehicles, pedestrians, pavements, blocks, traffic signals, etc., while driving around. It utilizes different types of sensors to take input from their surroundings and uses image classification to identify objects. After image classification, object localization (locating the presence of an object in the image using a bounding box) is performed, and the car decides its move.
2. **Security Systems:** There are many applications of image classification in the security sector. One such application is the facial recognition system. The face unlock system on our mobile devices primarily uses facial recognition systems. Similarly, the retina, iris, and fingerprint recognition systems are some prominent examples of image classification in security systems.
3. **Reverse Image Search:** Reverse image search is a process of searching for something using an image. The image is given as input to the search engine. After the classification, the search engine provides information about the image's subject.
4. **Healthcare:** There is ample application of image classification in the world of healthcare. For example, medical images from different tests are used to analyze and diagnose abnormalities and diseases, such as brain tumors and cancer. Another application of image classification in the healthcare industry is used to aid people with visual impairment navigate their daily lives.

5. **Manufacturing Industries:** In manufacturing industries, image classification is used to inspect and manage the quality of products without the manual intervention of human beings. Therefore, it saves time, as a human is not required to inspect each product thoroughly. Moreover, the inspection is more precise compared to human inspection, which is prone to errors.
6. **Land Mapping:** Another great use of image classification is land mapping. For example, geo-images can be analyzed to identify different types of terrains, such as forests, deserts, urban areas, agricultural lands, etc. This process is constructive to planning out where to build farmland or where to build residential areas.
7. **Agriculture:** In the agricultural field, image classification is also used for various purposes, such as plant disease detection, animal monitoring, pest detection, quality control, harvest monitoring, and more. Constant human intervention is not required, which makes the process more reliable and fast.

4.3.3 Challenges of Image Classification

A machine often struggles to learn from images due to its inherent limitations in identifying objects. For instance, an object may appear different in different images for several reasons, such as the following. These challenges make it hard for the machine to correctly identify images:

1. **Orientation Variation:** Some of the reasons that cause this variation are the orientation and positioning of the object, the environment's lighting, the angle at which the image is taken, the background, etc. For example, a cat's silhouette appears different when it is standing, sitting, lying down, or running.
2. **Object Positioning:** An image can be captured in various orientations and angles. For example, the head of the cat can be straight, tilted to an angle, or put down. Recognizing a cat in all these positions is a very trivial task for the human eyes. However, it is not such an easy task for a machine. Figure 4.2 shows how a

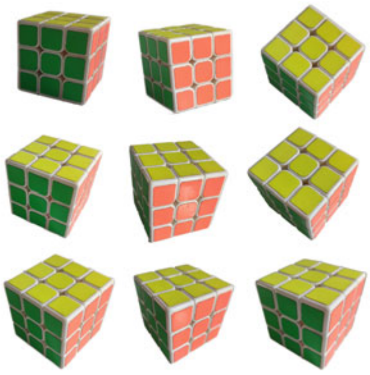


Fig. 4.2 Different orientation and positioning of one object

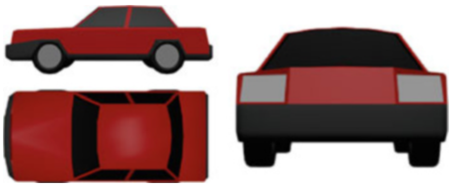


Fig. 4.3 Top view, front view, and side view of the same object that makes image classification challenging

Rubik's cube can look different from various angles and positions. A human eye can instinctively tell that the object is a Rubik's cube. However, it becomes challenging for the machines to do so.

3. **Feature Similarity:** The human eye can easily distinguish between a cat and a dog. However, a machine finds it hard to distinguish between a cat and a dog because they visually have many similar features, such as four limbs, one tail, two ears, two eyes, etc.
4. **Background:** The human eye can easily recognize objects with various backgrounds. However, if the background is too cluttered or too similar to the object, it is hard even for the human eyes to distinguish objects from the background. So, a cluttered background makes image classification more challenging.
5. **Viewpoint Variation:** The same object can appear differently due to the viewing perspective, i.e., top, side, or front views. For example, a car's top, side, and front views are entirely different, as shown in Fig. 4.3. Nevertheless, we never fail to recognize a car with our human eyes. However, this is deemed to be a difficult task for the computer.
6. **Different Lighting:** The images of an object with different environmental lightings but the same orientation, angle, view, etc., can challenge the computer to identify it. For instance, a big green tree with all its leaves would appear differently in images on a clear, foggy, or rainy day. Again, the tree looks completely different when all the leaves have fallen out. This difference makes image classification especially challenging. Figure 4.4 shows how the same portrait of a human face can vary in different lighting angles.

4.3.4 Implementation of Image Classification

Many extensive works have been done on deep-learning-based image classification techniques. The ability of an artificially intelligent model to identify an image has already been implemented in several prominent applications such as the *Apple Face ID lock*. In the ANN section of Chap. 3, we already discussed an image classification problem, e.g., the MNIST handwritten dataset. In addition to the MNIST dataset, PyTorch's official website provides popular image classification datasets such as CIFAR-10 and CIFAR-100 [2]. Figure 4.5 shows the CIFAR-10 dataset for image classification.

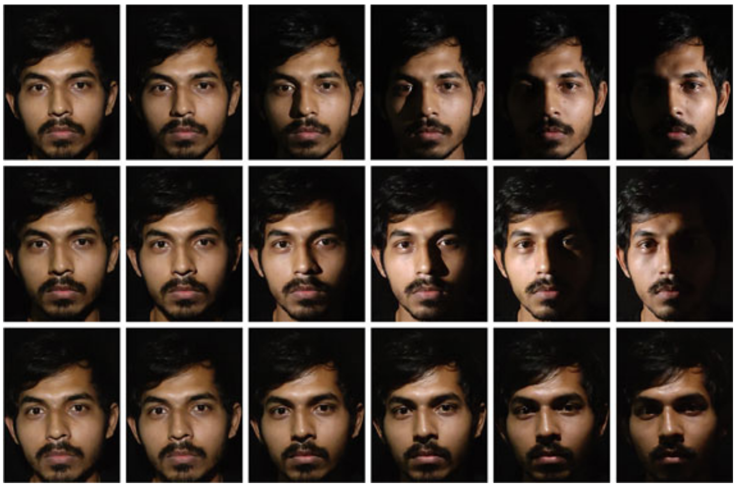


Fig. 4.4 Different lighting of the same object makes it seem to appear different. (Image courtesy: Shetu Mohanto)

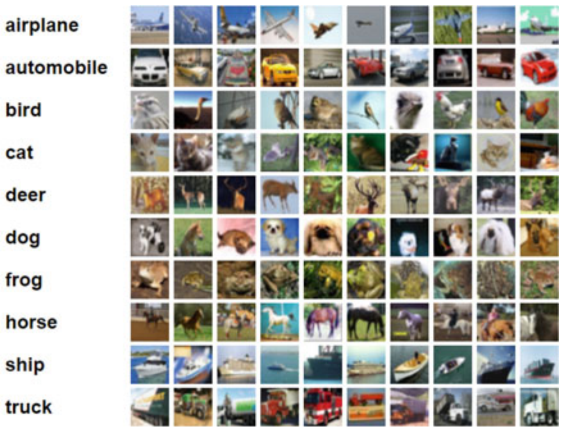


Fig. 4.5 CIFAR-10 dataset for image classification into 10 categories [2]

Programming Example 4.1

A program to download and process CIFAR-10 and CIFAR-100 datasets using PyTorch commands is shown in Listing 4.1. The parameters of these datasets are enlisted in Table 4.1. The listing defines train transformation and test transformation, which transforms the data to tensor and normalizes the data. The train transformation also includes random cropping and random horizontal flipping. Similarly, a separate data loader for both train and test data is also defined, which utilizes the transformations mentioned earlier. It is worth noting that all the sections are almost identical for both datasets.

```python
from torch.utils.data import DataLoader as DL
import torchvision.transforms as Trans
import torchvision.datasets as ds
BATCH_SIZE = 5

# -----Commands to Download and Prepare the CIFAR10 Dataset-----

train_transform = Trans.Compose([
    Trans.RandomCrop(32, padding=4),
    Trans.RandomHorizontalFlip(),
    Trans.ToTensor(),
    Trans.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994,
    0.2010)),
])

test_transform = Trans.Compose([
    Trans.ToTensor(),
    Trans.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994,
    0.2010)),
])

# train dataset
train_dataloader = DL(ds.CIFAR10('./data', train=True,
                                 download=True,
                                 transform=train_transform),
                      batch_size=BATCH_SIZE, shuffle=True)

# test dataset
test_dataloader = DL(ds.CIFAR10('./data', train=False,
                                transform=test_transform),
                     batch_size=BATCH_SIZE, shuffle=False)
print('CIFAR10 Dataset Pre-processing Done')


# -----Commands to Download and Prepare the CIFAR100 Dataset-----

train_transform = Trans.Compose([
    Trans.RandomCrop(32, padding=4),
    Trans.RandomHorizontalFlip(),
    Trans.ToTensor(),
    Trans.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994,
    0.2010)),
])

test_transform = Trans.Compose([
    Trans.ToTensor(),
    Trans.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994,
    0.2010)),
])

# train dataset
train_dataloader = DL(ds.CIFAR100('./data', train=True,
                                  download=True,
                                  transform=train_transform),
                      batch_size=BATCH_SIZE, shuffle=True)

# test dataset
test_dataloader = DL(ds.CIFAR100('./data', train=False,
                                 transform=test_transform),
                     batch_size=BATCH_SIZE, shuffle=False)
print('CIFAR100 Dataset Pre-processing Done')
```

Listing 4.1 Loading and preprocessing CIFAR dataset

Table 4.1 Parameters of the CIFAR-10 and CIFAR-100 datasets

Parameters	CIFAR-10	CIFAR-100
Class	10	100
Image Per Class	6000	600
Test Sample	10,000	100 (Per Class)
Training Sample	50,000	500 (Per Class)
Categorization of Class	Not applicable	20 Superclasses[a]

[a]Each image of the CIFAR-100 dataset comes with a "fine" label (the class to which it belongs) and a "coarse" label (the superclass to which it belongs)

Once downloaded and prepared for the classification task, we will demonstrate the application of classifying the CIFAR-10 data model in this section. In addition, we leave the task of classifying the CIFAR-100 dataset as an exercise at the end of the chapter. However, in comparison to Listing 3.5 (CNN example), the following modifications are necessary to modify the MNIST classification problem into a CIFAR-10/CIFAR-100 classification task:

1. MNIST is a grayscale dataset with only one input channel, but both CIFAR-10 and CIFAR-100 are color images with 3 input channels. Hence, *line 51* of Listing 3.5 should have 3 input channels instead of 1.
2. For CIFAR-100, there are 100 output classes. So, the final classification layer should have 100 output neurons instead of 10.

By applying the above modifications to the MNIST classification problem listed in Listing 3.5, we can perform image classification of both the CIFAR-10 and CIFAR-100 datasets. This task will also be left as an exercise to classify the CIFAR dataset using Listing 3.5, which uses the PyTorch module.

Programming Example 4.2

We will perform an image classification on the CIFAR-10 dataset using the TensorFlow and Keras modules. We need to use the GPU to accelerate the model's training process. For this purpose, make sure the notebook in Google Colab is connected to the GPU. You can set the "*Hardware accelerator*" to GPU in "*Notebook settings*"

Table 4.2 Architecture of the model used in Listing 4.2

Layer (type)	Output shape	Parameters
conv2d (Conv2D)	(None, 30, 30, 64)	1792
max_pooling2d (MaxPooling2D)	(None, 15, 15, 64)	0
dropout (Dropout)	(None, 15, 15, 64)	0
conv2d_1 (Conv2D)	(None, 13, 13, 128)	73,856
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 128)	0
dropout_1 (Dropout)	(None, 6, 6, 128)	0
conv2d_2 (Conv2D)	(None, 4, 4, 256)	295,168
conv2d_3 (Conv2D)	(None, 2, 2, 256)	590,080
max_pooling2d_2 (MaxPooling2D)	(None, 1, 1, 256)	0
dropout_2 (Dropout)	(None, 1, 1, 256)	0
flatten (Flatten)	(None, 256)	0
dense (Dense)	(None, 128)	32,896
dense_1 (Dense)	(None, 100)	12,900
dense_2 (Dense)	(None, 80)	8080
dense_3 (Dense)	(None, 60)	4860
dense_4 (Dense)	(None, 10)	610

from the "*Edit*" on the menu bar. We are going to build a simple sequential CNN model for image classification. The architecture of the model is given in Table 4.2.

The Python code to implement the problem is provided in Listing 4.2, which is explained in Table 4.3.

We need to preprocess the data before feeding the model with data. The pixel values of the images range from 0 to 255. Therefore, they need to be normalized to the range of 0 to 1. The label data are categorical data. The labels are as follows: "Airplane," "Automobile," "Bird," "Cat," "Deer," "Dog," "Frog," "Horse," "Ship," "Truck." *One-hot encoding* is performed on the label data, as CNN models cannot work with categorical data.

The outputs are visualized in Fig. 4.6. We can see that the first output, a frog, has been inaccurately predicted as a deer with 82% confidence. The second output has been accurately predicted as a cat with 85% confidence. The model's accuracy can be increased by tweaking and making changes to the architecture, which we will leave as a task for you to do.

```
# ----------------------Importing Modules------------------------
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.utils import to_categorical, plot_model
from tensorflow.keras import models, layers
from keras.layers import Conv2D, MaxPooling2D, Flatten , Dense,
    Activation,Dropout


# -------------------Loading CIFAR-10 Dataset------------------
(xtrain,ytrain),(xtest,ytest)= keras.datasets.cifar10.load_data()

#Preprocessing data
xtrain = xtrain/255
xtest = xtest/255
ytrain=to_categorical(ytrain)
ytest=to_categorical(ytest)


# ----------------------Creating CNN Model----------------------
model=models.Sequential()
model.add(layers.Conv2D(64,(3,3),input_shape=(32,32,3),activation
    ='relu'))
model.add(layers.MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.25))

model.add(layers.Conv2D(128,(3,3),activation='relu'))
model.add(layers.MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.25))

model.add(layers.Conv2D(256,(3,3),activation='relu'))
model.add(layers.Conv2D(256,(3,3),activation='relu'))
model.add(layers.MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.25))

model.add(layers.Flatten(input_shape=(32,32)))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(100, activation='relu'))
model.add(layers.Dense(80, activation='relu'))
model.add(layers.Dense(60, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
model.compile(optimizer='adam', loss='categorical_crossentropy',
    metrics=['accuracy'])
model.summary()

#Train model
xtrain2=xtrain.reshape(50000,32,32,3)
xtest2=xtest.reshape(10000,32,32,3)
model.fit(xtrain2,ytrain,epochs=40,batch_size=56,verbose=True,
    validation_data=(xtest2,ytest))


# ---------------------------Evaluation--------------------------
test_loss, test_acc = model.evaluate(xtest2, ytest)
print("accuracy:", test_acc)

#Visualising the output
predictions=model.predict(xtest2)
```

```
57 class_labels = ["airplane","automobile","bird","cat","deer","dog"
       ,"frog","horse","ship","truck"]
58 def visualize_output(predicted_label, true_label, img):
59   plt.xticks([])
60   plt.yticks([])
61   plt.imshow(img, cmap=plt.cm.binary)
62   predicted_label_index =np.argmax(predicted_label)
63   true_label_index=np.argmax(true_label)
64 
65   if predicted_label_index == true_label_index:
66     color='blue' # accurate prediction
67   else:
68     color='red' # inaccurate prediction
69 
70   plt.xlabel("{} {:2.0f}% ({})".format(class_labels[
       predicted_label_index], 100*np.max(predicted_label),
       class_labels[true_label_index]), color=color)
71 
72 plt.figure(figsize=(12,6))
73 visualize_output(predictions[1], ytest[1], xtest[1])
74 
75 plt.show()
```

Listing 4.2 Implementation of image classification using the CIFAR-10 dataset

Table 4.3 Explanation of the image classification code in Listing 4.2

Line number	Description
1–8	Importing modules
12	Downloading CIFAR-10 dataset
15–16	Performing normalization
17–18	Performing one-hot encoding for label data
22	Creating a sequential CNN model
23–25	Adding first convolutional block
27–29	Adding second convolutional block
31–34	Adding third convolutional block
36	Adding flattening layer
37–40	Adding fully connected layer
41	Adding the output layer
42	Compiling the model
43	The summary of model
45–48	Training the model
52–53	Evaluation of the model
55–70	Visualizing the output
72–75	Visualizing a random output

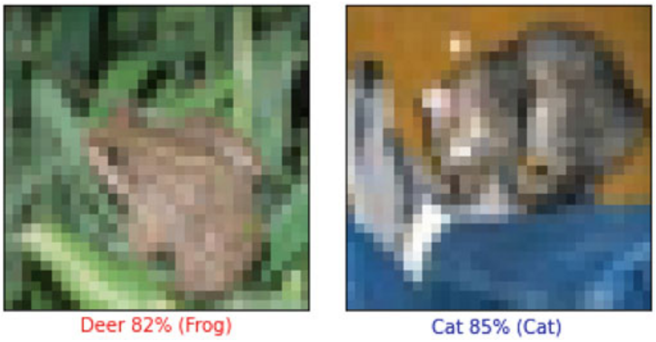


Fig. 4.6 Output of Listing 4.2

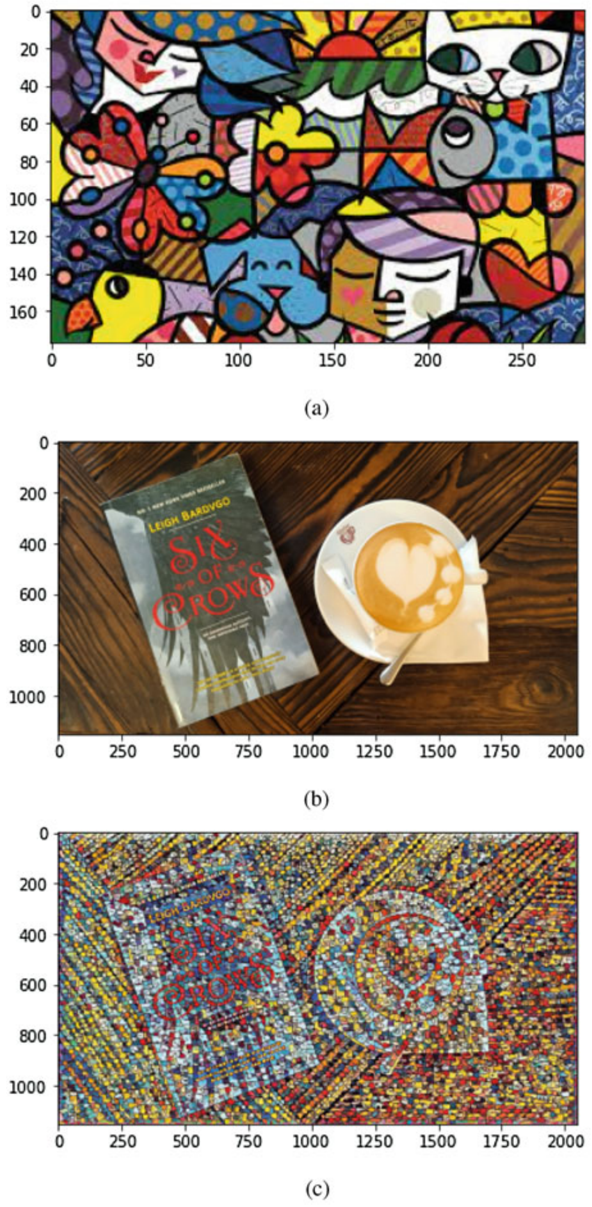
4.4 Neural Style Transfer (NST)

Neural Style Transfer [3] is an optimization method that blends two images—the *content image* and the *style image*—for imposing the style of the style image onto the content image to create the final output image. NST is used to change the style of an image (content image) based on any other image (style image) style or painting. For example, one may want the style of *The Starry Night* painting by *Vincent van Gogh* onto their own portrait by applying NST.

The neural network used in NST is trained on neither the content image nor the style image. Rather, it is used to extract the features from both images. The feature maps are crucial for NST. The lower layers of a CNN model contain detailed features and information about the image, such as pixel colors and textures. However, it does not contain the arrangement information of the pixels. Therefore, the lower layers are used as a feature extractor for style images. The higher layers of the CNN contain the features of the content of an image, that is, the arrangement of the pixels in the image. Thus, the higher layers are used as a content extractor for the content image. Then these two feature information, style features and content features, are blended to produce the final product. The big idea is that the output style should be the same as the style image, and the content of the output should be the same as the content image.

Programming Example 4.3
An implementation of NST is shown in this example using a pre-trained model from TensorFlow Hub in Python, whose output is provided in Fig. 4.7. For style image, it uses a colorful painting shown in Fig. 4.7a and a content image shown in Fig. 4.7b, which consists of coffee and a book. It is clear from the output image shown in Fig. 4.7c that the style image's style has been transferred from the style image.

Fig. 4.7 Output of the NST implementation in Listing 4.2. (**a**) Style image [4]. (**b**) Content image. (Image courtesy: Noushin Gauhar). (**c**) Output image



The code to implement the NST example is given in Listing 4.3 with its explanation in Table 4.4.

```
1  import tensorflow_hub as hub
2  import tensorflow as tf
3  from matplotlib import pyplot as plt
4  import numpy as np
5  import cv2
6
7  model = hub.load('https://tfhub.dev/google/magenta/arbitrary-
       image-stylization-v1-256/2')
8
9
10 def load_image(img_path):
11     img = tf.io.read_file(img_path)
12     img = tf.image.decode_image(img, channels=3)
13     img = tf.image.convert_image_dtype(img, tf.float32)
14     img = img[tf.newaxis, :]
15     return img
16
17 content = load_image('./data/content_image.png')
18 style = load_image('./data/style.png')
19
20 output = model(tf.constant(content), tf.constant(style))[0]
21
22 cv2.imwrite('./results/output.png', cv2.cvtColor(np.squeeze(
       output)*255, cv2.COLOR_BGR2RGB))
```

Listing 4.3 Implementation of Neural Style Transfer using TensorFlow Hub model

4.4.1 NST Applications

Some popular applications of NST are discussed below:

1. **Gaming:** NST manipulates the gaming interface video. Google introduced Stadia [5] during a press conference at the 2019 Game Developers Conference, which uses NST to impose different styles of texture and color palettes into the gaming world. The integration of NST with the gaming interface has given the power

Table 4.4 Explanation of the NST code in Listing 4.3

Line number	Description
1–5	Importing modules
7	Importing TensorFlow Hub model
10–15	Function for loading images
17–18	Loading content and style images
20	Performing NST
22	Saving output

of mapping different textures from different paintings, images, videos, and more into the virtual interface of games.

2. **Virtual reality:** The field of virtual reality is still in the experimental phase with NST. NST is used to transfer the style of real-life surroundings to the virtual world to give the experience of different scenarios. It also gives the experience of living and roaming around in famous paintings. One can experience the surrealism of living in *The Starry Night* painting.
3. **Image and video editor:** One of the explicit examples of NST is various photo and video editing tools and software. Many software are available now both on computers and mobile devices. These software use NST to impose the style of different paintings or templates on input images. This allows users to create creative portraits, selfies, and images of themselves. It also enables users to be creative in editing their video clips. Some of the widely used applications that use NST for photo and video editing are Prisma, Video Star (Art Studio, iOS), Painter's Lens (iOS), Looq (iOS), Instapainting's AI Painter (Desktop), Arbitrary Style Transfer in the Browser (Desktop), etc.

4.5 Feature Extraction or Dimensionality Reduction

In Chap. 3, we have already discussed the possibility of using Principal Component Analysis (PCA) as a dimensionality reduction method. This section demonstrates another dimensionality reduction technique ideal for large input dimensions, popularly known as *AutoEncoder* [6].

Autoencoders consist of two key parts—*encoder* and *decoder*. The encoder can project input data $\boldsymbol{X}$ to a latent space vector $\boldsymbol{Z}$. A latent vector is a low-dimensional representation of the information contained in $\boldsymbol{X}$. A decoder can decode the encoded latent space $\boldsymbol{Z}$ back to the original input space as $\hat{\boldsymbol{X}}$. The sample illustration of an autoencoder is shown in Fig. 4.8.

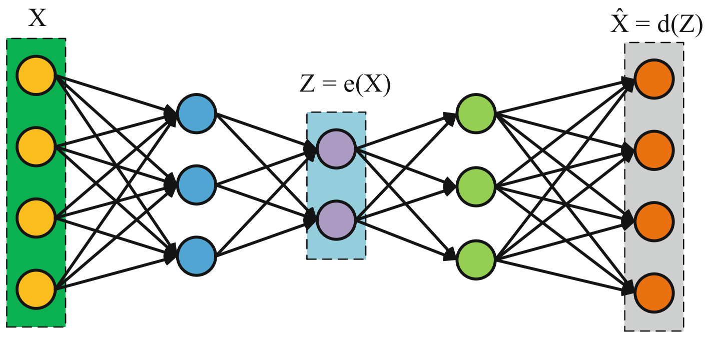


Fig. 4.8 A sample autoencoder

Programming Example 4.4
We will demonstrate an example of the MNIST dataset where we extract two-dimensional features from 784 input features of the image. The autoencoder is defined using three classes: `Encoder`, `Decoder`, and `Autoencoder`. The `Encoder` encodes images to a latent space, while the `Decoder` reconstructs input images from the latent space. The `Autoencoder` combines both of these to complete the autoencoder model. The Python code is given in Listing 4.4 with its explanation in Table 4.5.

```python
# -------------------------Torch Modules-------------------------
from __future__ import print_function
import numpy as np
import pandas as pd
import torch.nn as nn
import math
import torch.nn.functional as F
import torch
from torch.nn import init
import torch.optim as optim
from torch.utils.data import DataLoader as DL
import torchvision.transforms as Trans
import torchvision.datasets as ds
from torchvision import models
import torch.nn.functional as F
import matplotlib.pyplot as plt;
# ---------------------------Variables---------------------------
# batch size
BATCH_SIZE = 128
Iterations = 20
feature_dims = 2

# ------Commands to Download and Prepare the MNIST Dataset-------
train_transform = Trans.Compose([
    Trans.ToTensor(),
])
test_transform = Trans.Compose([
    Trans.ToTensor(),
])  # no normalization for Autoencoder training

# train dataset
train_dataloader = DL(ds.MNIST('./mnist', train=True,
                               download=True,
                               transform=train_transform),
                      batch_size=BATCH_SIZE, shuffle=True)
# test dataset
test_dataloader = DL(ds.MNIST('./mnist', train=False,
                              transform=test_transform),
                     batch_size=BATCH_SIZE, shuffle=False)


# -------------------Defining the Autoencoder--------------------
class Encoder(nn.Module):
    def __init__(self, feature_dims):
        # Encoder part of the Autoencoder whcih projects x to a latent space z
        super(Encoder, self).__init__()
        self.linear1 = nn.Linear(784, 512)
        self.linear2 = nn.Linear(512, feature_dims)

    def forward(self, x):
        x = torch.flatten(x, start_dim=1)
        x = F.relu(self.linear1(x))
        return self.linear2(x)


class Decoder(nn.Module):
    def __init__(self, feature_dims):
        # Decoder part of the latent space which project the intermediate features back to reconstruct the image
        super(Decoder, self).__init__()
        self.linear1 = nn.Linear(feature_dims, 512)
        self.linear2 = nn.Linear(512, 784)

    def forward(self, z):
        z = F.relu(self.linear1(z))
        z = torch.sigmoid(self.linear2(z))
        return z.reshape((-1, 1, 28, 28))


class Autoencoder(nn.Module):
    def __init__(self, feature_dims):
        # combining the encoer and decoder to create the auto encoder
        super(Autoencoder, self).__init__()
        self.encoder = Encoder(feature_dims)
        self.decoder = Decoder(feature_dims)

    def forward(self, x):
        z = self.encoder(x)
        return self.decoder(z)


# defining Autoencoder model
model = Autoencoder(feature_dims)

# defining which paramters to train only the CNN model parameters
optimizer = torch.optim.Adam(model.parameters())


# -------------------Training of Autoencoder---------------------
# Train baseline classifier on clean data
def train(model, optimizer, epoch):
    model.train()  # setting up for training
    for id_batch, (data, target) in enumerate(
93              train_dataloader):  # data contains the image and
        target contains the label = 0/1/2/3/4/5/6/7/8/9
94          optimizer.zero_grad()  # setting gradient to zero
95          output = model(data)  # forward
96          loss = ((output - data) ** 2).sum()  # MSE loss
97          loss.backward()  # back propagation here pytorch will
        take care of it
98          optimizer.step()  # updating the weight values
99          if id_batch % 100 == 0:
100             print('Epoch No: {} [ {:.0f}% ]   \tLoss: {:.3f}'.
        format(
101                     epoch, 100. * id_batch / len(train_dataloader),
        loss.item()))
102
103
104
105 ## training the Autoencoder
106 for i in range(Iterations):
107     train(model, optimizer, i)  # train Function
108
109
110 # plotting function
111 def plot(model, data_loader):
112     for i, (x, y) in enumerate(data_loader):
113         z = model.encoder(x)
114         z = z.detach().numpy()
115         plt.scatter(z[:, 0], z[:, 1], c=y, cmap='tab10')
116         plt.xlabel("Value of Feature 1")
117         plt.ylabel("Value of Feature 2")
118         plt.savefig("./results/features.png")
119
120
121 # plotting the latent space feature
122 plot(model, train_dataloader)
```

Listing 4.4 Autoencoder Feature Extraction

Table 4.5 Explanation of the name classification task listed in Listing 4.4

Line number	Description
2–14	Using our previous Torch modules
16–21	Declaring the variables
24–41	Download and prepare the MNIST dataset
43–54	Encoder model
56–66	Decoder model
68–77	Autoencoder model
88–100	Training function; in particular, line 93 uses MSE loss between decoder output and input data
106–113	Plot function

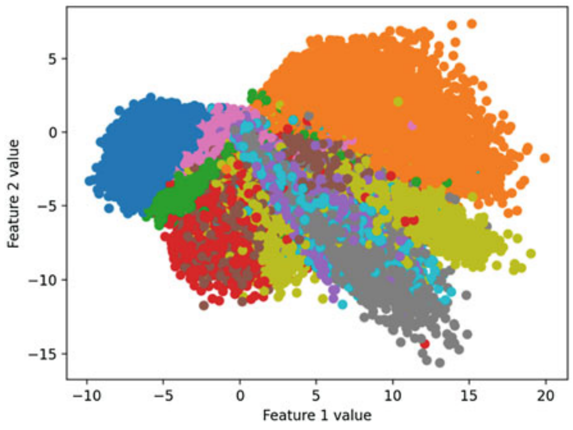


Fig. 4.9 Two-dimensional feature extracted from the MNIST digits. Each color represents a specific class of image

The code in Listing 4.4 takes 28 × 28 grayscale MNIST images as input and plots their extracted features in 2D. The extracted 2D features of the ten classes have been shown in ten colors in Fig. 4.9. The features extracted in Fig. 4.9 can be used to perform classification tasks instead of the original digit images.

4.6 Anomaly or Outlier Detection

An anomaly or outlier is something that significantly and statistically deviates from what is expected. The outlier points have different statistical characteristics than the rest of the dataset. For example, a scenario exists where a person regularly drives from point A to point B during office hours. If suddenly it is seen from their driving history that a person has driven from point A to point C, it is seen as an anomaly. It may raise suspicions about whether the person is in danger or involved in some criminal activities. There are many critical applications of outlier detection.

Figure 4.10 shows four types of outliers—local, global, contextual, and collective. These types are discussed below:

1. **Local Outliers:** When a data point significantly differs from its neighboring points in the dataset, it is called a local outlier. For instance, in a daily load curve, if there is a sudden peak demand at midnight, it is a local outlier. This peak is not usual at midnight, but if the full-day curve is observed, it is still lower than the evening peak.
2. **Global Outliers:** When the data point deviates significantly from all the existing data points in the dataset, it is called a global outlier. It is also often referred to as a point anomaly. The diagram in Fig. 4.10 point B visually depicts deviation from the rest of the set. This is a case of a global outlier. For instance, a person spends $500 on average in a week. If the person suddenly spends more than 1 million dollars in one week, this will be a prime case of a global outlier.

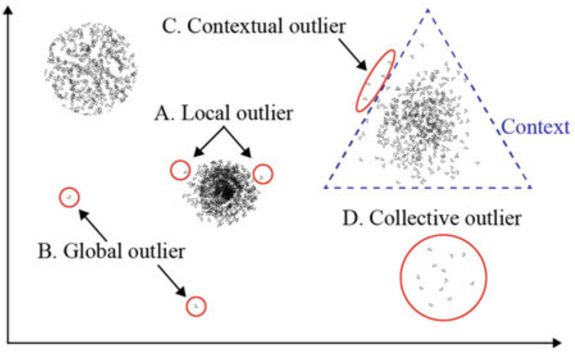


Fig. 4.10 Different types of outliers

3. **Contextual Outliers:** When the concern arises about whether a data point is an anomaly or not based on a specific context, it is considered a contextual outlier. It is often referred to as a conditional outlier because whether the data point would be viewed as an outlier or not is conditioned on its context. Time is often considered the context in this case. For instance, a temperature of 30 °C is expected in the summer, but this would be a case of a contextual outlier in the winter because this high temperature is not expected in winter.
4. **Collective Outliers:** When a set of data points significantly deviate from the rest of the dataset, it is called a collective outlier. For instance, if a sudden increase is seen in a transaction in the stock market within specific groups, it can be identified as a collective outlier.

Next, we will discuss how outliers can be detected.

4.6.1 How Does It Work?

For simpler cases, simpler statistical methods such as standard deviations and interquartile range are used for outlier detections. There are also some other outlier detection algorithms that are used in the case of high-dimensional feature space, i.e., when many input variables are present. Two outlier detection algorithms used for such purposes are isolation forest and local outlier factor. These four methods of outlier detection are briefly discussed below.

4.6.1.1 Standard Deviation

The standard deviation can be used to detect outliers or anomalies for datasets that follow the Gaussian distribution. The data points that fall outside the range from three standard deviations below the mean ($\mu - 3\sigma$) to three standard deviations above the mean ($\mu + 3\sigma$) are considered outliers. These are depicted in Fig. 4.11. Here, μ denotes the mean of the dataset, and σ denotes the standard deviation.

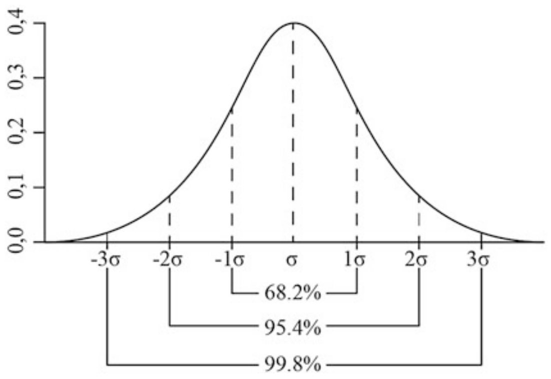


Fig. 4.11 Data points that fall outside the $\mu \pm 3\sigma$ are outliers

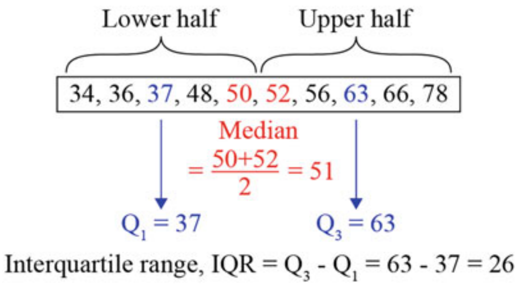


Fig. 4.12 Calculating Q1, Q3, and IQR for a dataset

4.6.1.2 Inter-quartile Range (IQR)

For datasets with a skewed distribution, the inter-quartile range can detect outliers or anomalies. The data points that fall below $Q1 - 1.5IQR$ or fall above $Q3 + 1.5IQR$ are considered outliers. Here, Q1 is the 25th percentile of the dataset, Q3 is the 75th percentile of the dataset, and IQR is the inter-quartile range that is calculated as $Q3 - Q1$. Figure 4.12 denotes the Q1, Q3, and IQR of a dataset.

4.6.1.3 Isolation Forest

The core concept of isolation forest, an unsupervised model, is similar to random forest and is built using decision trees. Randomly subsampled data is processed in an isolation forest using a tree structure based on randomly chosen attributes. Since it took more cuts to isolate the samples that traveled further into the tree, they are less likely to be abnormalities. Therefore, the data that get on shorter branches are considered anomalies or outliers because it shows that they had few similar attributes. Liu et al. [7] proposed the method in 2008 and have stated in their paper that this method leverages two quantitative properties of anomalies: "i) they are the minority consisting of fewer instances, and ii) they have attribute values that are very different from those of normal instances."

4.6.1.4 Local Outlier Factor (LOF)

LOF is a density-based outlier technique [8]. It mainly detects local outliers and does not deal with global outliers. It basically works by detecting data points in the feature space that have sparse density, i.e., scattered away from the rest of the dataset. LOF assigns an anomaly score to each data point based on their density level or local neighborhood in the feature space. The points with an anomaly score that exceeds a threshold are detected as outliers. This threshold is set by a domain expert given the scenario for which the outliers are being detected.

4.6.2 Applications of Anomaly Detection

Some of the significant applications of anomaly detection are given below:

1. **Fraud detection:** A person usually maintains a habitual event while using their credit card. Usually, they spend a specific amount and do not exceed the amount. Again, they usually maintain a habit of buying from certain places. If the credit card gets stolen, the spending pattern of the thief will not match the owner's spending pattern. The thief will exceed in spending amount and will buy from random places. This anomaly may raise suspicion and block the credit card from being used.
2. **Security:** Any network has almost consistent traffic in its course. When a hacker or intruder tries to break into the network, the number of traffic deviates drastically. This drastic change in traffic in a network may suggest an attack on the network. According to the detection, the defense system may take the necessary steps and save the network.
3. **Business transaction:** Transactions in stock markets are a regular occurrence. Every transaction follows a pattern with a slight deviation. When suddenly a drastic change is seen in the transaction within specific groups, this alerts the stock market, and immediate actions are taken to control the transaction. This sudden change of transaction may occur when one group is trying to dominate the stock market.
4. **Healthcare:** During different medical tests, anything that is not normal can be detected using outlier detection methods. According to the detection, further necessary tests and treatments can be done to treat the patient. For instance, normal cells and cancerous cells act differently. If a sudden change is found in the cell's characteristic, it will be marked and tested further to see whether it is a case of cancer or any other disease.

4.6.3 Challenges of Anomaly Detection

There exist many challenges in detecting anomalies. Some of the challenges are briefly discussed below:

1. **Modeling of outlier data:** Generally, when training models, we train them with normal data. Any data that statistically deviate from the normal data are

considered outlier data. This process is basically how outliers are detected. Therefore, it is essential that normal data are modeled very carefully because if not, outlier detection will be erroneous. Modeling normal data is a challenging task. We need to define every possible normal behavior of data and create a dataset accordingly. Often it becomes difficult to define the boundary line that differentiates the normal data from outlier data. This makes outlier detection challenging.

2. **Noise vs. Outliers:** The presence of noise is inevitable in real-world datasets. Noise and outliers are different concepts, and they are not the same. Outliers are valid data, except they are generated statistically differently from normal data. On the other hand, noises are unwanted data point in the dataset. They can be missing values, incorrect values, duplicate values, etc. The presence of noise throws off the expected characteristic of normal data, and it becomes challenging to retrieve normal data and model them, thus making the work of detecting outliers difficult.
3. **Domain-specific knowledge:** Not all statistically deviated data is considered an outlier in every context. Therefore, only some outlier detection is significant in some scenarios. We need to establish outlier detection based on context and application. The characteristics of normal data have to be defined based on the domain outlier detection used. Again, the level of deviation and input type is different for different applications. For instance, the slightest deviation can be considered in medical fields. On the other hand, for fraud detection in credit cards, the deviation needs to be significant enough to be detected as an outlier.
4. **Interpretability:** It is not only challenging to detect outliers but also to interpret why the detected data are outliers. What is the purpose of a detected outlier in a specific case, and what are the imposed conditions for those specific data points to be outliers? For application-based outlier detection, it is crucial to interpret the condition and criterion of outliers.

4.6.4 Implementation of Anomaly Detection

The *Beijing Multisite Air-Quality dataset* [9] will be used to implement anomaly detection. This time series dataset includes hourly air pollutants data from 12 nationally controlled air-quality monitoring sites. However, data for only one site will be considered for implementation. We will implement the local outlier factor algorithm for this purpose.

Programming Example 4.5
The Python code is provided in Listing 4.5 with its explanation in Table 4.6. The code preprocesses the dataset. The `LocalOutlierFactor` class from scikit-learn was used to identify and remove outliers from the training data, and the refined data were trained through a linear regression model.

```
Dataset: https://www.kaggle.com/datasets/sid321axn/beijing-
    multisite-airquality-data-set

# ----------------------Importing Modules------------------------
import numpy as np
import pandas as pd
import missingno
import matplotlib.pyplot as plt
import seaborn as sb
from sklearn.neighbors import LocalOutlierFactor
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error


# -------------------------Reading Data--------------------------

dataset = pd.read_csv("./data/PRSA_Data_Aotizhongxin_20130301
    -20170228.csv")


# preprocessing dataset
dataset.isnull().sum()
df = dataset.dropna()

X = df[['PM10', 'CO']]
Y = df['PM2.5']

X_train, X_test, y_train, y_test = train_test_split(X, Y,
    test_size=0.20, random_state=1)


# ------------------Local Outlier Factor (LOF)-------------------
model = LocalOutlierFactor(n_neighbors= 35 , contamination= 0.1)
predict = model.fit_predict(X_train)
mask = predict != -1
X_train, y_train = X_train.iloc[mask, :], y_train.iloc[mask]


model = LinearRegression()
model.fit(X_train, y_train)
yhat = model.predict(X_test)
mae = mean_absolute_error(y_test, yhat)
print('MAE: %.3f' % mae)
```

Listing 4.5 Outlier detection using the LOF algorithm

4.7 Adversarial Input Attack

Recently, the deep learning models have been shown to be vulnerable to slight malicious noise formally known as *adversarial example* [10, 11]. In Fig. 4.13, we display an example of an adversarial image where a malicious attacker can add

Table 4.6 Explanation of the outlier detection code in Listing 4.5

Line number	Description
3–12	Importing modules
17	Reading dataset
21	Checking if null values exist
22	Dropping all the rows that contain null values
24	Selecting features
25	Target variable
27	Splitting the dataset into train and test sets
31–32	Fitting LOF model
33	Selecting rows without outliers
34	Setting trainset without outliers
37–41	Fitting and evaluating the model

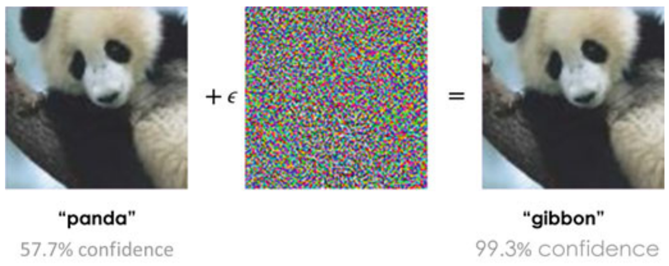


Fig. 4.13 Adversarial example of a panda image. (Source: Open AI Blog [12])

slight, imperceptible noise to the incoming *panda* image with the objective of misclassifying it into a wrong output class *gibbon*.

Among the popular adversarial input attacks are *Fast Gradient Sign Method (FGSM)* [10] and *Projected Gradient Descent (PGD)* method [11]. A typical FGSM algorithm follows the mathematical strategy expressed by Eq. 4.1.

$$\underbrace{\hat{\boldsymbol{x}}}_{\text{adversarial example}} = \underbrace{\boldsymbol{x}}_{\text{clean data}} + \underbrace{\epsilon \cdot \operatorname{sign}\big(\nabla_{\boldsymbol{x}} \mathcal{L}(g(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{t})\big)}_{perturbation}. \tag{4.1}$$

In Eq. 4.1, the attacker takes the clean data $\boldsymbol{x}$ and adds a perturbation, which is the sign of the loss function of a neural network with respect to the input data scaled by ϵ. Here ϵ is the L_∞ norm between the clean data $\boldsymbol{x}$ and the adversarial data $\hat{\boldsymbol{x}}$. Next, we take our MNIST image classification problem from Chap. 3 and apply the above algorithm to generate adversarial examples of each test image.

Programming Example 4.6

A simple adversarial example generation code is given in Listing 4.6 with the attack module. The code explanation can be found in Table 4.7. The FGSM test involves testing machine learning models with adversarial attacks. This code trains a CNN model on the MNIST dataset and performs an FGSM attack on the trained model. A perturbed image is created by adjusting each pixel of the input image. The attack function evaluates the attack's success rate by computing the test loss and accuracy on the perturbed data. It also saves some of the original and perturbed images for visualization.

```python
# -------------------------Torch Modules-------------------------
from __future__ import print_function
import numpy as np
import pandas as pd
import torch.nn as nn
import math, torch
import torch.nn.functional as F
from torch.nn import init
import torch.optim as optim
from torchvision import datasets as ds
from torchvision import transforms as Trans
from torchvision import models
import torch.nn.functional as F
from torchvision.utils import save_image
from torch.utils.data import DataLoader as DL
# --------------------------Variables--------------------------
# for Normalization
mean = [0.5]
std = [0.5]
# batch size
bs = 128
Iterations = 5
learn_rate = 0.01


# ------Commands to download and perpare the MNIST dataset-------
train_transform = Trans.Compose([
    Trans.ToTensor(),
    Trans.Normalize(mean, std)
])

test_transform = Trans.Compose([
    Trans.ToTensor(),
    Trans.Normalize(mean, std)
])

# train dataset
train_loader = DL(ds.MNIST('./mnist', train=True, download=True,
    transform=train_transform),
                  batch_size=bs, shuffle=True)

```

```
# test dataset
test_loader = DL(ds.MNIST('./mnist', train=False, transform=
    test_transform),
                 batch_size=bs, shuffle=False)


# --------------------------Defining CNN--------------------------
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output


# defining CNN model
model = CNN()

# Loss function
criterion = torch.nn.CrossEntropyLoss()  # pytorch's cross
    entropy loss function

# defining which parameters to train only the CNN model
    parameters
optimizer = torch.optim.SGD(model.parameters(), learn_rate)


# -----------------------Training Function-----------------------
# Train baseline classifier on clean data
def train(model, optimizer, criterion, epoch):
    model.train()  # setting up for training
    for id, (data, target) in enumerate(
            train_loader):  # data contains the image and target
    contains the label = 0/1/2/3/4/5/6/7/8/9
        optimizer.zero_grad()  # setting gradient to zero
```

```python
        output = model(data)  # forward
        loss = criterion(output, target)  # loss computation
        loss.backward()  # back propagation here pytorch will take care of it
        optimizer.step()  # updating the weight values
        if id % 100 == 0:
            print('Train Epoch No: {} [ {:.0f}% ]   \tLoss: {:.6f}'.format(
                epoch, 100. * id / len(train_loader), loss.item()))


# ----------------------Testing Function-----------------------
# validation of test accuracy
def test(model, criterion, val_loader):
    model.eval()
    test_loss = 0
    correct = 0

    with torch.no_grad():
        for id, (data, target) in enumerate(val_loader):
            output = model(data)
            test_loss += criterion(output, target).item()  # sum up batch loss
            pred = output.max(1, keepdim=True)[1]  # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()  # if pred == target then correct +=1

    test_loss /= len(val_loader.dataset)  # average test loss
    print('\nTest set: \nAverage loss: {:.4f}, \nAccuracy: {}/{} ({:.4f}%)\n'.format(
        test_loss, correct, val_loader.sampler.__len__(),
        100. * correct / val_loader.sampler.__len__()))


# training the CNN
for i in range(Iterations):
    train(model, optimizer, criterion, i)
    test(model, criterion, test_loader)  # Testing the the current CNN


# ----------------------FGSM Attack Code-----------------------
def fgsm(model, data, target, epsilon=0.1, data_min=0, data_max=1):
    'this function takes a data and target model as input and produces a adversarial image to fool model'
    data_min = data.min()
    data_max = data.max()
    model.eval()  # evaluation mode
    perturbed_data = data.clone()  # data setup

    perturbed_data.requires_grad = True
    output = model(perturbed_data)  # ouput
    loss = F.cross_entropy(output, target)  # loss

    if perturbed_data.grad is not None:
        perturbed_data.grad.data.zero_()

    loss.backward()  # backward loss

    # Again set gradient requirement to true
    perturbed_data.requires_grad = False

    with torch.no_grad():
        # Create the perturbed image by adjusting each pixel of
    the input image
        perturbed_data += epsilon * perturbed_data.grad.data.sign
    ()
        # Adding clipping to maintain [min,max] range, default
    0,1 for image
        perturbed_data.clamp_(data_min, data_max)

    return perturbed_data


# ---------------Evaluating the Attack Success Rate--------------
def Attack(model, criterion, val_loader):
    model.eval()
    test_loss = 0
    correct = 0

    for id, (data, target) in enumerate(val_loader):

        adv_img = fgsm(model, data, target, epsilon=0.3)
        if id == 0:  # saving the image
            save_image(data[0:100], './results/data' + '.png',
    nrow=10)
            save_image(adv_img[0:100], './results/adv' + '.png',
    nrow=10)
        output = model(adv_img)
        test_loss += criterion(output, target).item()  # sum up
    batch loss
        pred = output.max(1, keepdim=True)[1]  # get the index of
     the max log-probability
        correct += pred.eq(target.view_as(pred)).sum().item()  #
    if pred == target then correct +=1

    test_loss /= len(val_loader.dataset)  # average test loss
    print('\nTest set: \nAverage loss: {:.4f}, \nAccuracy After
    Attack: {}/{} ({:.4f}%)\n'.format(
        test_loss, correct, val_loader.sampler.__len__(),
        100. * correct / val_loader.sampler.__len__()))


```

```
# Executing the attack code
Attack(model, criterion, test_loader)
```

Listing 4.6 Adversarial Examples

Table 4.7 Explanation of the adversarial attack generation code in Listing 4.6

Line number	Description
1–15	Import necessary libraries and modules
16–23	Define variables for data normalization and hyperparameters
27–35	Define data transformations for training and testing datasets
37–43	Create data loaders for the MNIST dataset
46–70	Define a CNN model with two convolutional layers, dropout layers, and fully connected layers
74	Create an instance of the CNN model
77	Define the loss function for training
80	Define the optimizer SGD for training
84–96	Define a training function that iterates through training data, computes loss, performs backpropagation, and updates model weights
100–116	Define a testing function to evaluate the accuracy of the model on the test dataset
119–122	Train the CNN model for a specified number of iterations, printing loss during training, and testing the model after each iteration
126–151	FGSM attack generation code
128–129	Taking the minimum and maximum values of the data
130	Setting the model to evaluation mode
133–135	Computing the loss with respect to the input image
137–138	Setting the gradient of the input to zero before calling backward
140	Calling the backpropagation function
147	Adding the noise using the gradient sign similar to Equation 4.1
149	Ensuring pixel values of the adversarial examples are within the L_∞ norm
154–174	Evaluates the attack methodology
162	Generating the adversarial image; the rest of the function is similar to the test function of Listing 3.5
178	Calling the attack function

We will get the following output if we apply the above part of the code at the end of Listing 3.5.

Output of Listing 4.6:

```
Train Epoch No: 0 [ 0% ]    Loss: 2.317856
Train Epoch No: 0 [ 21% ]    Loss: 1.753910
Train Epoch No: 0 [ 43% ]    Loss: 0.838018
Train Epoch No: 0 [ 64% ]    Loss: 0.636633
Train Epoch No: 0 [ 85% ]    Loss: 0.532179

Test set:
```

```
Average loss: 0.0026,
Accuracy: 9058/10000 (90.5800%)

Train Epoch No: 1 [ 0% ]    Loss: 0.515063
Train Epoch No: 1 [ 21% ]    Loss: 0.461006
Train Epoch No: 1 [ 43% ]    Loss: 0.350277
Train Epoch No: 1 [ 64% ]    Loss: 0.460339
Train Epoch No: 1 [ 85% ]    Loss: 0.309240

Test set:
Average loss: 0.0018,
Accuracy: 9354/10000 (93.5400%)

Train Epoch No: 2 [ 0% ]    Loss: 0.249620
Train Epoch No: 2 [ 21% ]    Loss: 0.244345
Train Epoch No: 2 [ 43% ]    Loss: 0.470304
Train Epoch No: 2 [ 64% ]    Loss: 0.302229
Train Epoch No: 2 [ 85% ]    Loss: 0.275732

Test set:
Average loss: 0.0014,
Accuracy: 9490/10000 (94.9000%)

Train Epoch No: 3 [ 0% ]    Loss: 0.249589
Train Epoch No: 3 [ 21% ]    Loss: 0.248986
Train Epoch No: 3 [ 43% ]    Loss: 0.220321
Train Epoch No: 3 [ 64% ]    Loss: 0.183422
Train Epoch No: 3 [ 85% ]    Loss: 0.180923

Test set:
Average loss: 0.0011,
Accuracy: 9584/10000 (95.8400%)

Train Epoch No: 4 [ 0% ]    Loss: 0.165640
Train Epoch No: 4 [ 21% ]    Loss: 0.224271
Train Epoch No: 4 [ 43% ]    Loss: 0.146689
Train Epoch No: 4 [ 64% ]    Loss: 0.220537
Train Epoch No: 4 [ 85% ]    Loss: 0.167554

Test set:
Average loss: 0.0010,
Accuracy: 9601/10000 (96.0100%)

Test set:
Average loss: 0.0128,
Accuracy After Attack: 4530/10000 (45.3000%)
```

In Fig. 4.14, we show some sample adversarial images that a human can recognize clearly. However, such maliciously designed samples can greatly fool a neural network. For Listing 4.6, the attack can fool the target neural network and degrade the accuracy to 45.3%. In the next section, we will train a binary classifier to detect such adversarial examples.

4.8 Malicious Input Detection

Once we generate adversarial examples with malicious noise, the next question would be: can we detect such an anomaly (malicious noise)? In this section, we train a convolution neural network (CNN) to detect adversarial samples from clean MNIST images. We will train a binary classifier whose output would be 1 for adversarial images and 0 if the incoming image is a clean sample.

Programming Example 4.7
The Python code for this problem is given in Listing 4.7 with its explanation in Table 4.8. A CNN class is defined within the script to build a CNN model specifically designed for the MNIST dataset. An instance of this class is then created and loaded with pre-trained weights from a file. The code also implements a `Detector` class to detect adversarial examples using a binary classifier. An optimizer is defined to facilitate the updating of the parameters of this class during training. The `fgsm` function is also implemented in the script, which employs the FGSM attack. This function takes a model, data, target, and an optional epsilon value as input and generates an adversarial image intended to deceive the model.

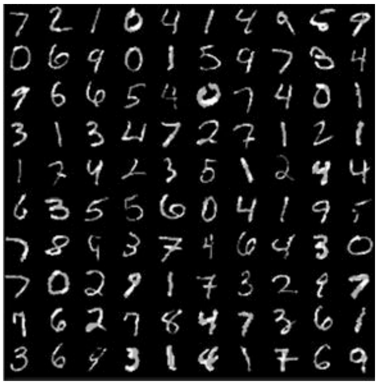


Fig. 4.14 Sample adversarial examples

```
# -------------------------Torch Modules-------------------------
from __future__ import print_function
import numpy as np
import pandas as pd
import torch.nn as nn
import math
import torch.nn.functional as F
import torch
from torch.nn import init
import torch.optim as optim
from torch.utils.data import DataLoader as DL
import torchvision.transforms as Trans
import torchvision.datasets as ds
from torchvision import models
import torch.nn.functional as F
# ---------------------------Variables---------------------------
mean = [0.5] # for Normalization
std = [0.5]

BATCH_SIZE =128
Iterations = 2
learning_rate = 0.001


# ------Commands to Download and Perpare the MNIST Dataset-------
train_transform = Trans.Compose([
        Trans.ToTensor(),
        Trans.Normalize(mean, std)
        ])

test_transform = Trans.Compose([
        Trans.ToTensor(),
        Trans.Normalize(mean, std)
        ])


train_dataloader = DL(
        ds.MNIST('./mnist', train=True, download=True,
                       transform=train_transform),
        batch_size=BATCH_SIZE, shuffle=True) # train dataset

test_dataloader = DL(
        ds.MNIST('./mnist', train=False,
                        transform=test_transform),
        batch_size=BATCH_SIZE, shuffle=False) # test dataset


# -----------------Loading the pre-trained model-----------------
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output

# defining CNN model
model = CNN()
# loading a pre-trained model
model = torch.load('./data/mnist.pt')


# -----------------------Defining Detector-----------------------
class Detector(nn.Module):
    def __init__(self):
        super(Detector, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1,
    stride=1)
        self.maxpool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1,
    stride=1)
        self.maxpool2 = nn.MaxPool2d(2)
        self.linear1 = nn.Linear(7*7*64, 200)
        self.linear2 = nn.Linear(200, 1)


    def forward(self, x):

        out = self.maxpool1(F.relu((self.conv1((x)))))
        out = self.maxpool2(F.relu((self.conv2(out))))
        out = out.view(out.size(0), -1)
        #print(out.size())
        out = F.relu((self.linear1(out)))
        out = F.sigmoid(self.linear2(out))
        return out

# defining the detector
D= Detector()

```

```python


# loss function for binary classification
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(D.parameters(), lr=learning_rate)


# -----------------------FGSM Attack Code------------------------
def fgsm(model, data, target, epsilon=0.1, data_min=0, data_max
    =1):
        'this function takes a data and target model as input and
     produces a adversarial image to fool model'
        data_min = data.min()
        data_max = data.max()
        model.eval() # evaluation mode
        perturbed_data = data.clone() # data setup

        perturbed_data.requires_grad = True
        output = model(perturbed_data) # ouput
        loss = F.cross_entropy(output, target) # loss

        if perturbed_data.grad is not None:
            perturbed_data.grad.data.zero_()

        loss.backward() ## backward loss

        # Again set gradient requirement to true
        perturbed_data.requires_grad = False

        with torch.no_grad():
            # Create the perturbed image by adjusting each pixel
    of the input image
            perturbed_data += epsilon*perturbed_data.grad.data.
    sign()
            # Adding clipping to maintain [min,max] range,
    default 0,1 for image
            perturbed_data.clamp_(data_min, data_max)
        return perturbed_data


# ---------------------Training and Testing----------------------
# Train the detector model
def train(model, adv, optimizer,criterion,epoch):
    model.train() # setting up for training
    for id_batch, (data, target) in enumerate(train_dataloader):
    # data contains the image and target contains the label =
    0/1/2/3/4/5/6/7/8/9
        optimizer.zero_grad() # setting gradient to zero

        # clean data loss
        output = model(data) # forward
        real_labels = torch.zeros(target.size()[0], 1) # clean
    sample labels = 0
```

```
        loss = criterion(output, real_labels) # loss computation

        # adversarial data loss
        adv_img = fgsm(adv, data, target, epsilon=0.3) ## geerating the adversarial samples
        output1 = model(adv_img)
        fake_labels = torch.ones(target.size()[0], 1) # adversarialsample label =1

        loss1 = criterion(output1, fake_labels)
        loss = (loss+ loss1)/2 # overall loss function
        loss.backward() # back propagation here pytorch will take care of it

        optimizer.step() # updating the weight values
        if id_batch % 100 == 0:
            print('Epoch No: {} [ {:.0f}% ]   \tLoss: {:.3f}'.format(
                epoch, 100. * id_batch / len(train_dataloader), loss.item()))



# --------------Evaluating the Attack Success Rate---------------
# Validation of detection rate of malicious samples
def test(model, adv,val_loader, epoch):
    model.eval()
    test_loss = 0
    correct = 0

    for id_batch, (data, target) in enumerate(val_loader):
            adv_img = fgsm(adv, data, target, epsilon=0.3)
            output = model(adv_img)

            for i in range(data.size()[0]):
                if output [i] > 0.9:
                    correct +=1

    print('\n Detection Rate:',
        100. * correct / 10000 )


# Training the detector and testing
for i in range(Iterations):
    train(D, model,optimizer,criterion,i)
    test(D, model, test_dataloader, i) # Testing the the current CNN
```

Listing 4.7 Malicious Input Detection

Table 4.8 Explanation of the detection code in Listing 4.7

Line number	Description
2–76	Similar as prior code in Listing 3.5
78	Loading a pre-trained model trained to classify MNIST images
81–103	Detector model is a CNN architecture with one output neuron
108	The loss function presented here is a binary cross-entropy loss
109	Using the ADAM optimizer
113–138	FGSM attack code
142–165	Train function
149	Creating real image label 0
155	Creating fake image label 1
158	Combining both losses
170–184	Testing the detection rate of malicious samples

Output of Listing 4.7:

```
Epoch No: 0 [ 0% ]    Loss: 0.694
Epoch No: 0 [ 21% ]    Loss: 0.001
Epoch No: 0 [ 43% ]    Loss: 0.000
Epoch No: 0 [ 64% ]    Loss: 0.000
Epoch No: 0 [ 85% ]    Loss: 0.000

 Detection Rate: 100.0

Epoch No: 1 [ 0% ]    Loss: 0.000
Epoch No: 1 [ 21% ]    Loss: 0.000
Epoch No: 1 [ 43% ]    Loss: 0.000
Epoch No: 1 [ 64% ]    Loss: 0.000
Epoch No: 1 [ 85% ]    Loss: 0.000

 Detection Rate: 100.0
```

The above code has a detection rate of 100%, indicating the binary classifier can always detect malicious adversarial samples from clean data provided that the malicious sample generation method is FGSM. This demonstrates a supervised detection problem of any incoming data, which shows an anomaly compared to the original clean data.

4.9 Natural Language Processing

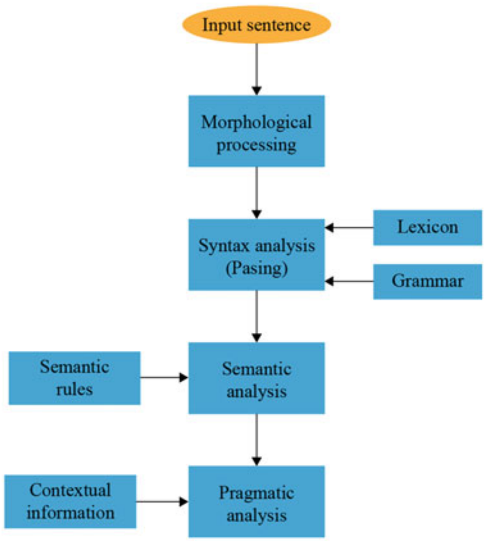
Natural language processing (NLP) is the subfield of artificial intelligence concerned with giving the computer the human capacity to process language. Humans read texts and hear audio and can instinctively understand, interpret, and act accordingly. The goal of NLP is for the machine to take in both audible and visual text, which is natural language, and interpret it. It is challenging for computers to reach the human level in terms of processing languages. NLP has a broad real-world application in many fields, including medical, business, and educational industries. Researchers are pushing boundaries further and further to perfect the art of NLP. *IBM Watson* [13] is one of the most significant accomplishments in the field of NLP.

4.9.1 How Does NLP Work?

Figure 4.15 also gives a visual representation of the workflow of NLP. The four basic steps of NLP are described in this section:

1. **Morphological processing:** In this step, the input text is broken down into sets of tokens. Paragraphs are broken down into sentences, then sentences are broken down into words, and words are broken down into units. For example, the word "unbreakable" can be broken into un, break, and able.
2. **Syntax Analysis:** This step checks whether the sentences are developed using correct grammatical rules or not. According to syntax analysis, the sentence "I

Fig. 4.15 Workflow of NLP



eat brick" is valid because it is formed correctly according to English grammar rules, although the sentence makes no sense.
3. **Semantic Analysis:** This step analyzes and checks if the sentences or phrases hold accurate meanings according to their dictionary meanings. The previous example shown, "I eat brick," would be rejected in semantic analysis because the sentence has no meaning.
4. **Pragmatic Analysis:** The relevance of the given text is checked during pragmatic analysis based on the present context. "Bring the pen from the holder that is blue"—here, either the pen is blue or the holder is blue. The sentence is open to two different interpretations.

4.9.2 Applications of NLP

NLP has revolutionized many aspects of modern technology. Some of the most common applications of NLP are stated below:

1. **Text prediction and auto-correction:** While texting on mobile devices or computers, the next word is often predicted and shown on the screen. The probability of the words is calculated, and the word with the highest probability is shown. There are rules by which the probability is calculated. Sometimes a misspelled word is automatically corrected. This process of auto-correction is also done in the same. The probability of the right spelling is calculated, and the word with the highest probability is replaced with the misspelled word.
2. **Email filtering:** It will be a tremendous job to review each email we receive daily. Email filtering is the process of categorizing emails according to their content and priority. Some emails have higher priority over others. It is easy to scroll through emails categorically. In this way, the most urgent emails are checked first. Emails are categorized into business, socials, marketing promotions, etc., categories.
3. **Grammar checker:** Different software are used to check for spelling and grammatical errors. These software are applications of NLP. The software checks and corrects the spelling when a sentence or paragraph is given as input. It also points out and rectifies grammatical errors. It can also paraphrase the text for better readability. According to the goal of the text, the software can point out if the writing is too formal or informal. It also provides better synonyms to make the writing more interesting.
4. **Spam detection:** Spam detection is used to scan and filter out spam or phishing emails and messages and send them to spam messages. Text classification is used for spam detection. A set of words or phrases is kept as indicators of this text classification. These words or phrases can be foul language, specific financial terms, inappropriate phrases, suspicious and unverified links, specific brand names, etc.
5. **Machine translation:** While translating a text from one language to another, it is not enough to replace the words. The translated form should also capture the true essence and tone of the authentic text. Machine translation is a modern NLP technology. *Google Translate* is a widely used example of machine translation.

6. **Sentiment analysis:** NLP has emerged as a crucial commercial tool to extract hidden data insights from social media channels. Sentiment analysis can examine the language used in social media postings, comments, reviews, and more to extract attitudes and emotions in response to products, promotions, and events. Businesses can use this information to create new products, launch new marketing initiatives, and more.
7. **Text summarization:** Text summarization is used to summarize massive amounts of digital text and produce summaries and synopses for indexes, research databases, etc. Text summarizing uses NLP techniques. The finest text summary software uses natural language generation (NLG) and semantic reasoning to summarize relevant context and conclusions.
8. **Speech recognition:** Speech recognition aims to take audio as input and output equivalent text. Then, it performs the necessary operations according to the equivalent text. Similar to image classification tasks, voice recognition tasks of AI models are extremely popular in the forms of *Apple Siri* [14], *Google Assistant* [15], and *Amazon Alexa* [14].

4.9.3 Challenges of NLP

When a word is misspelled, it is easy for humans to detect and replace it with the correct spelling. However, it is hard for the computer to detect and correct the spelling. There are many challenges to NLP, some of which are discussed below briefly:

1. **Multiple meanings:** Many words have more than one meaning. For example, in English, the word "blue" means color and sadness. It is instinctive for humans to interpret the meaning of "blue" based on the context. For example, "*I am feeling blue*"—We can easily understand that the word "blue" means sadness. However, it is difficult for machines to interpret the meanings instinctively.
2. **Irony and sarcasm:** Human beings use irony and satirical phrases. A person who had an awful day might say sarcastically, "*I had the best day today!*" A human being will quickly understand the sarcasm based on the given scenario and delivery tone. Nevertheless, it is hard for machines to pick up on sarcastic phrases.
3. **Synonyms:** Many words have synonymous meanings but are sometimes used differently to express feelings. "*You look good*" is a very generic compliment, and it is given in for formalities even though they might not look so good. "*You look amazing!*" tells that the person looks really good. It is not just for the sake of formalities. It is challenging for machines to pick up on this degree with synonymous words.
4. **Ambiguity:** Ambiguity makes NLP very challenging. In case of lexical or semantic ambiguity, some words take different meanings based on the context. For example, the word "water" can be used as both a noun and a verb in English. "*Please give me a glass of water*"—here, "water" is used as a noun. "*Please*

do not forget to water the plants"—here, "water" is used as a verb. In case of syntactic or structural ambiguity, the sentence structure creates a confusing meaning, such as "*I spotted the guy with the telescope.*" Who has the telescope in this sentence? Did I spot the guy using the telescope, or was the guy holding a telescope? These types of ambiguity are hard to understand for machines.

5. **Domain-specific language:** Language used by various firms and industries might vary considerably. For instance, an NLP processing model required for the processing of medical records might differ significantly from one required for processing legal papers. Although many analysis tools that have been trained for particular disciplines are available now, specialized companies may still need to develop or train their own models.

4.9.4 Implementation of NLP

As an application of NLP, *speech recognition* will be implemented in this section. In this task, we try to demonstrate one voice recognition application using a language dataset from PyTorch's official website [16]. The dataset has the names of different people in 18 different languages. The list of the 18 different languages is given below:

1. Polish	4. Japanese	7. English	10. French	13. Greek	16. Dutch
2. Russian	5. Italian	8. Scottish	11. German	14. Spanish	17. Chinese
3. Czech	6. Portuguese	9. Arabic	12. Irish	15. Korean	18. Vietnamese

Programming Example 4.8
In this task, we will train an RNN model to learn to classify the names into a specific class. For example, *Akrivopoulos* is a Greek name, and *Chang* is a Korean name. Next, we provide a sample code for the language recognition task based on the speech (e.g., name). Listing 4.8 shows the Python code for this task, and Table 4.9 explains the code. The code defines an RNN class implementing a simple RNN model, which incorporates two linear layers and a softmax layer. The forward method of the class takes an input tensor and a hidden state tensor as input and computes the output and the updated hidden state of the model. The forward method of the class accepts an input tensor and a hidden state tensor as input, computes the output, and updates the model's hidden state.

```
## Datset: https://pytorch.org/tutorials/intermediate/
    char_rnn_classification_tutorial.html
## This is the RNN word classification problem in PyTorch
    Official Website
from __future__ import unicode_literals, print_function, division
from io import open
import glob
import os
```

```python
import unicodedata
import string
import torch
import torch.nn as nn
import random
import time
import math

# ---------------------Data Preparation--------------------------
def findFiles(path): return glob.glob(path)


all_letters = string.ascii_letters + " .,;'"
n_letters = len(all_letters)

# Turn a Unicode string to plain ASCII from stack overflow https
    ://stackoverflow.com/a/518232/2809427
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
        and c in all_letters
    )



# Build the category_lines dictionary, a list of names per
    language
category_lines = {}
all_categories = []

# Read a file and split into lines
def readLines(filename):
    lines = open(filename, encoding='utf-8').read().strip().split
    ('\n')
    return [unicodeToAscii(line) for line in lines]

for filename in findFiles('data/names/*.txt'):
    category = os.path.splitext(os.path.basename(filename))[0]
    all_categories.append(category)
    lines = readLines(filename)
    category_lines[category] = lines # category_lines is a
    dictionary mapping each category (language) to a list of
    lines (names).

n_categories = len(all_categories)

# category_lines is a dictionary mapping each category (language)
     to a list of lines (names).


# ---------One hot vector represnetation of letters--------------

# Find letter index from all_letters, e.g. "a" = 0
def letterToIndex(letter):
    return all_letters.find(letter)

# Just for demonstration, turn a letter into a <1 x n_letters> Tensor
def letterToTensor(letter):
    tensor = torch.zeros(1, n_letters)
    tensor[0][letterToIndex(letter)] = 1
    return tensor

# Turn a line into a <line_length x 1 x n_letters>,
# or an array of one-hot letter vectors
def lineToTensor(line):
    tensor = torch.zeros(len(line), 1, n_letters)
    for li, letter in enumerate(line):
        tensor[li][0][letterToIndex(letter)] = 1
    return tensor

print(letterToTensor('J'))

print(lineToTensor('Jones').size())



# -------------------------RNN Model----------------------------
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size) ## the basic blocks of RNN again is the simple linear layers
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

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

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

n_hidden = 128 ## We keep the basic structure similar to pytorch official implementation
rnn = RNN(n_letters, n_hidden, n_categories)

# ------------Some important function for training---------------
```

```
102
103 def categoryFromOutput(output):
104     "This function interprets the output of using topk function
        to locate which language category the output indicates"
105     top_n, top_i = output.topk(1)
106     category_i = top_i[0].item()
107     return all_categories[category_i], category_i
108
109 ## the following lines 118-130 randomly slects some training
        samples in different catergory
110
111 def randomChoice(l):
112     return l[random.randint(0, len(l) - 1)]
113
114 def randomTrainingExample():
115     category = randomChoice(all_categories)
116     line = randomChoice(category_lines[category])
117     category_tensor = torch.tensor([all_categories.index(category
        )], dtype=torch.long)
118     line_tensor = lineToTensor(line)
119     return category, line, category_tensor, line_tensor
120
121 for i in range(10):
122     category, line, category_tensor, line_tensor =
        randomTrainingExample()
123     print('category =', category, '/ line =', line)
124
125 # loss function
126 criterion = nn.NLLLoss()
127
128 learning_rate = 0.005 # learning rate
129
130
131 # ------------------------Training loop--------------------------
132 def train(category_tensor, line_tensor):
133
134     """Each loop of training will
135     Create input and target tensors
136     Create a zeroed initial hidden state
137     Read each letter in and
138     Keep hidden state for next letter
139     Compare final output to target
140     Back-propagate
141     Return the output and loss"""
142
143     hidden = rnn.initHidden()
144
145     rnn.zero_grad()
146
147     for i in range(line_tensor.size()[0]):
148         output, hidden = rnn(line_tensor[i], hidden)
149
150     loss = criterion(output, category_tensor)
```

```
    loss.backward()

    # Add parameters' gradients to their values, multiplied by
    learning rate
    for p in rnn.parameters():
        p.data.add_(p.grad.data, alpha=-learning_rate)

    return output, loss.item()

## some hyper parameters
n_iters = 100000
print_every = 5000



# Keep track of losses
current_loss = 0
all_losses = []

def timeSince(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)

start = time.time()

# training loop
for iter in range(1, n_iters + 1):
    category, line, category_tensor, line_tensor =
    randomTrainingExample()
    output, loss = train(category_tensor, line_tensor)
    current_loss += loss

    # Print iter number, loss, name and guess
    if iter % print_every == 0:
        guess, guess_i = categoryFromOutput(output)
        correct = ' ' if guess == category else ' (%s)' %
    category
        print('%d %d%% (%s) %.4f %s / %s %s' % (iter, iter /
    n_iters * 100, timeSince(start), loss, line, guess, correct))

    all_losses.append(current_loss /1)
    current_loss = 0


# -------------------------Evaluation----------------------------

## evlauation function return the output
def evaluate(line_tensor):
    hidden = rnn.initHidden()

```

```
200      for i in range(line_tensor.size()[0]):
201          output, hidden = rnn(line_tensor[i], hidden)
202
203      return output
204
205  ##predicts the output language of a given class
206  def predict(input_line, n_predictions=3):
207      print('\n> %s' % input_line)
208      with torch.no_grad():
209          output = evaluate(lineToTensor(input_line))
210
211          # Get top N categories
212          topv, topi = output.topk(n_predictions, 1, True)
213          predictions = []
214
215          for i in range(n_predictions):
216              value = topv[0][i].item()
217              category_index = topi[0][i].item()
218              print('(%.2f) %s' % (value, all_categories[
         category_index]))
219              predictions.append([value, all_categories[
         category_index]])
220
221
222  # Testing how accurate the output will be:
223  predict('Dovesky')
224  predict('Daheri')
225  predict('Jackson')
226  predict('Satoshi')
```

Listing 4.8 Speech Recognition [16]

Table 4.9 Explanation of the name classification code in Listing 4.8

Line number	Description
3–13	Using Unicode and Torch modules
16–28	Data preparation and converting letter into ASCII code
33–34	Two empty dictionary per language
36–45	Reading the text files in data folder
54–56	Converting letter into index
59–70	Converting the index into one-hot coded value
79–97	Describing RNN model
101–128	Defining some important functions for training
133–191	Training function and training loop
196–203	Evaluation function
205–220	Prediction function
223–226	Evaluating four names whose output is shown

Output of Listing 4.8:

```
> Dovesky
(-0.70) Polish
(-1.36) Russian
(-1.59) Czech

> Daheri
(-0.97) Japanese
(-1.73) Italian
(-1.98) Portuguese

> Jackson
(-0.68) English
(-1.40) Scottish
(-2.25) Russian

> Satoshi
(-0.97) Japanese
(-1.07) Arabic
(-2.16) Italian
```

As displayed above, our trained RNN model clearly predicts Dovesky as Polish, Daheri as Japanese, Jackson as English, and Satoshi as a Japanese name. In this way, we can use the program to identify the origin of any name.

4.10 Conclusion

In our increasingly digitalized world, signals are present everywhere. Machine learning has stepped into the world of signal processing to help in multidimensional tasks. This chapter focuses on the signal processing applications of machine learning and deep learning. It covers the concepts, applications, and Python implementations of a wide range of signal processing use cases, such as image classification, neural style transfer, feature extraction or dimensionality reduction, anomaly/outlier detection, adversarial input attack, malicious input detection, and natural language processing. All these topics have wonderful real-world applications that are used daily worldwide. The readers will gain a concise overview of signal processing and study the different ways in which machine learning can make signal processing easy and interesting. In the next chapter, we will study machine learning applications in energy systems.

4.11 Key Messages from This Chapter

- The applications of machine learning and deep learning for signal processing have become integrated into our lives, from email writing to mobile face lock ID systems.
- Machine learning can be used for image classification applications whose practical implementations are boundless.
- Neural style transfer allows adopting the style of one image into another image, thus resulting in a stylized image.
- Adversarial input attack and malicious input detection are useful in the cyber-security domain and have important practical implementations.
- Natural language processing is bringing about a massive revolution today, and its applications are used everyday by everyone.

4.12 Exercise

1. Describe the relationship between signal processing and machine learning. Show some examples.
2. Sometimes, the available image dataset might not be sufficient to properly train a machine learning model. Explain how you will respond to this problem if you cannot collect external data.
3. Define feature and feature extraction. Why is feature extraction/dimensionality reduction required?
4. What is meant by adversarial input attack? Show a workaround to this problem:
5. (a) Define natural language and natural language processing (NLP).
   (b) Briefly describe some aspects of NLP.
6. Modify the CIFAR-10 image classification model architecture from Listing 4.2 to improve model performance.
7. Create and train a CIFAR-100 image classifier. Use CNN for image classification.
8. Capture a portrait of yours and stylize that image with another image of your choice by applying neural style transfer.
9. Create an AutoEncoder to project the MNIST classifier dataset into the latent space of 20 dimensions and use these 20 features to create an MNIST image classifier.
10. Create a PGD adversarial attack algorithm by modifying the FGSM attack in Listing 4.6. PGD attack algorithm can be found in [11].
11. Train a classifier with both clean and adversarial images so that the model can recognize both clean and noisy images into the correct class. This is popularly known as adversarial training [11].

References

1. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems, 25*, 1097–1105.
2. Krizhevsky, A., Hinton, G., et al. (2009). *Learning multiple layers of features from tiny images*.
3. Gatys, L. A., Ecker, A. S., & Bethge, M. (2015). A neural algorithm of artistic style. *arXiv preprint arXiv:1508.06576*.
4. Britto, R. https://www.pakocampo.com/garden/
5. Wiggers, K. *Google's Stadia uses style transfer ML to manipulate video game environments*. https://venturebeat.com/ai/googles-stadia-uses-style-transfer-ml-to-manipulate-video-game-environments/
6. Wang, Y., Yao, H., & Zhao, S. (2016). Auto-encoder based dimensionality reduction. *Neurocomputing, 184*, 232–242.
7. Liu, F. T., Ting, K. M., & Zhou, Z.-H. (2008). Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining* (pp. 413–422). IEEE.
8. Breunig, M. M., Kriegel, H.-P., Ng, R. T., & Sander, J. (2000). LOF: Identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (pp. 93–104).
9. Dong, A., He, J., Xu, Z., Chen, S. X., Zhang, S., & Guo, B. *Beijing Multisite Air-Quality dataset*. https://archive.ics.uci.edu/ml/datasets/Beijing+Multi-Site+Air-Quality+Data
10. Goodfellow, I. J., Shlens, J., & Szegedy, C. (2014). Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*.
11. Madry, A., Makelov, A., Schmidt, L., Tsipras, D., & Vladu, A. (2018). Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*.
12. Huang, S., Duan, Y., Abbeel, P., Clark, J., Goodfellow, I., Papernot, N. (2017). *Attacking machine learning with adversarial examples—openai.com*. https://openai.com/research/attacking-machine-learning-with-adversarial-examples [Accessed 09 Sep 2023].
13. High, R. (2012). The era of cognitive systems: An inside look at IBM Watson and how it works. *IBM Corporation, Redbooks, 1*, 16.
14. Hoy, M. B. (2018). Alexa, Siri, Cortana, and more: An introduction to voice assistants. *Medical Reference Services Quarterly, 37*(1), 81–88.
15. Google. *Google voice assistant*. https://assistant.google.com/
16. Robertson, S. *NLP from scratch: Classifying names with a character-level RNN*. https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

5 Applications of Machine Learning: Energy Systems

5.1 Introduction

Since the beginning of the electrical power systems implementation, it has undergone numerous significant changes, improvements, and evolution, such as the transformation from DC power to AC power and now to smart grid systems. Historically, conventional energy systems have been passive in nature since the power and information flow are unidirectional. The smart grid system incorporates various renewable energy sources and distributed energy resources into the conventional grid. The integration of different energy resources requires mutual communication, data, and power flow between them; thus, the grid system becomes active in nature. This bidirectional power and information flow helps more efficient and reliable energy system operations and management.

The smart grid ecosystem's interconnected communication and management strategy is executed through various dedicated advanced communication devices, protocols, phasor measurement units (PMU), and Internet of Things (IoT) devices. These operations result in a large amount of data generation and accumulation. Insights from these data can be utilized to ensure the most efficient and optimal smart grid application. The application of machine learning (ML) models comes into play in this data-centric decision-making process. For example, historical load usage data can be used to train predictive ML models for short and long-term electrical load forecasting. Various measurement data from PMUs can be used to train ML models to detect and classify different electrical faults in real-time to provide accurate and instantaneous protective measures.

Renewable energy integration plays a major part in smart grid systems. In order to establish a sustainable renewable energy-based solution, proper long-term planning of renewable energy establishment is essential. ML-based predictive models can play a significant part in this regard. Moreover, ML can be implemented to provide real-time power factor control and correction mechanism more efficiently

E. Hossain, *Machine Learning Crash Course for Engineers*,
https://doi.org/10.1007/978-3-031-46990-9_5

and quickly. ML models can also be applied in load monitoring, energy security, and even as a helping hand for protection against cyber vulnerabilities and exploits.

Thus, data-driven ML-based methods have revolutionized energy systems' operations and management structures. Independent energy market ecosystem, the ever-increasing renewable energy penetration in power systems along with increasing power electronics applications, application of phasor measurement units (PMU), advanced communication, and Internet of Things (IoT) infrastructures in smart grid ecosystem, and increased hardware capabilities—all of these contribute to the generation and acquisition of a large amount of data in the energy domain. These collected and related synthetic data can be utilized to train and deploy various predictive and discriminative (classification) machine learning models to ensure more precise and robust design, control, management, system optimization, and system response in smart energy systems.

This chapter focuses on building a basic understanding of ML applications in energy systems, which will act as a knowledge base for further study and research in this particular domain. At first, we will study load forecasting and then dive into fault analysis studies, followed by future trend prediction in renewable energy systems, reactive power control, and power factor correction.

5.2 Load Forecasting

Load forecasting refers to predicting the electrical power demand and ensuring adequate supply to meet the demand. This is helpful in formulating planning and operational strategies for continuous and reliable operation of the electrical generation, transmission, and distribution systems [1]. Electric load forecasting can be of several types based on duration, such as long-term, medium-term, short-term, and very short-term load forecasting, corresponding to annual, monthly, single-day, and hourly load forecasting [2].

This section demonstrates a practical example of electrical load forecasting using the hourly energy consumption data at Northern Illinois [3]. This section will use a special kind of RNN, popularly known as the *Long Short-Term Memory (LSTM)* network, to perform the forecasting task.

An LSTM network consists of two states in comparison to just one in RNN—the *cell state* and the *hidden state*, which is depicted in Fig. 5.1. Each input enters the model, combines with the hidden short-term memory, and then goes through a set of activation functions (e.g., softmax, tanh). Finally, the output is produced through arithmetic operations (e.g., addition/multiplication) with the cell state long-term memory. At the same time, the model generates a new cell state and a hidden state for future inputs. This type of model is ideal for predicting a series of sequential data.

The dataset is illustrated graphically in Fig. 5.2, where the energy load (GW) is displayed with respect to each year. The sliding window takes a portion of the year as the training input X and predicts the subsequent load output as Y. The idea of a sliding window is similar to the moving average concept, where a larger window

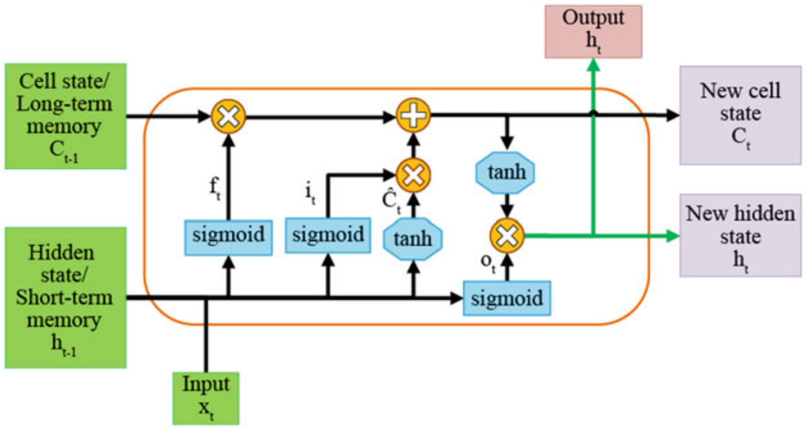


Fig. 5.1 An LSTM model consists of input, hidden state, cell state memory, and activation functions (e.g., sigmoid, tanh)

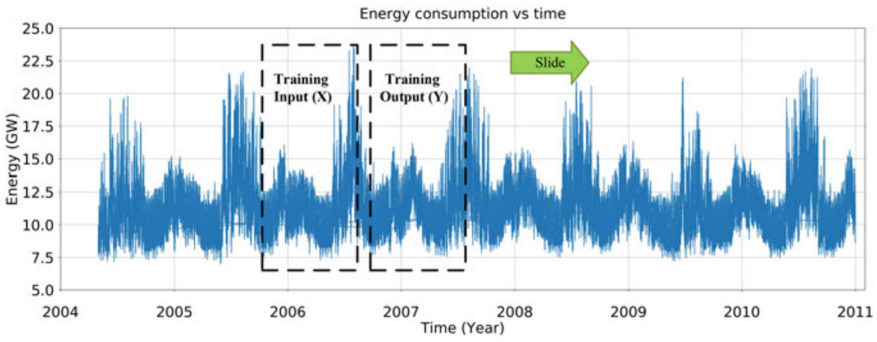


Fig. 5.2 Forecasting data at Northern Illinois: Energy (MW) vs. Year plot. Each sliding window contains a pair of training data (X,Y)

size will reduce our dataset size, and a smaller window size will make each data sample oscillate without any visible pattern for the training model. In this example, we chose a moderate window size of 30.

Programming Example 5.1
In Fig. 5.3, we demonstrate the necessary steps to follow in the PyTorch code to build a predictor for the forecasting prediction. Listing 5.1 shows the code for building the load forecasting model for the said dataset. First, we must process the data and prepare the training samples (X,Y) in lines 67–109 of Listing 5.1. Next, we define the LSTM model as shown in Fig. 5.1 in lines 123–150 of Listing 5.1. Finally, the training and evaluation of the model are presented in lines 154–223 of Listing 5.1. The overall summary of the code is presented in Table 5.1. The code is written for an IPython or Jupyter Notebook environment.

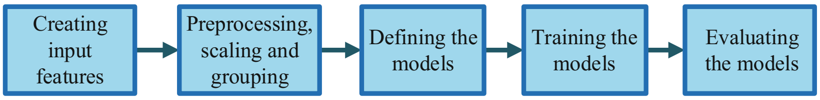


Fig. 5.3 Steps for building the load forecasting model. First, the data goes through a preparation step, then a processing step, then we define the LSTM model and train the given dataset. Finally, we evaluate the model using a hold-out validation dataset

```python
# Dataset: https://www.kaggle.com/robikscube/hourly-energy-
    consumption

# -------------------------Torch Modules-------------------------
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import TensorDataset as ds
from torch.utils.data import DataLoader as DL
from tqdm.notebook import tqdm
from sklearn.preprocessing import MinMaxScaler
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


# -----------------------Hyper-Parameters-------------------------
# to keep track of index column
tar_idx = 0
in_idx = range(5)

# Define window_size period
kw = 30

# batch size
bs = 256

# number of training iterations
iters = 5
# number of layers
layers = 3
# learning rate
lr = 0.001


# -----------------------Data Preperation-------------------------
data_path = "./data/NI_hourly.csv"
filename = "NI_hourly.csv"


# prepares the dataset for training and testing
def data_prep(data_values, kw, in_idx, tar_idx):

    '''
    This function creates a sliding window of the data and each
    slice will be a potential input to the model with a target
    label
    '''

    # creates input and label for training and testing
    inputs = np.zeros((len(data_values) - kw, kw, len(in_idx)))
    target = np.zeros(len(data_values) - kw)

    # this loop creates input containing samples from kw window
    and target value
    for i in range(kw, len(data_values)):

        inputs[i - kw] = data_values[i - kw:i, in_idx]
        target[i - kw] = data_values[i, tar_idx]

    inputs = inputs.reshape(-1, kw, len(in_idx))
    target = target.reshape(-1, 1)
    print(inputs.shape, target.shape)

    return inputs, target


# This dictionary re-scale the target during evaluation
target_scalers = {}
trainX = []
testX = {}
testY = {}


# --------Reading the File: North Ilinois Power Load (MW)--------
df = pd.read_csv(f'{data_path}', parse_dates=[0])

#reading each input
df['Hours'] = df['Datetime'].dt.hour
df['Day_of_weeks'] = df['Datetime'].dt.dayofweek
df['Months'] = df['Datetime'].dt.month
df['Day_of_year'] = df['Datetime'].dt.dayofyear
df = df.sort_values("Datetime").drop('Datetime', axis=1)

# scaling the input
scale = MinMaxScaler()
target_scale = MinMaxScaler()
data_values = scale.fit_transform(df.values)

# target scaling for evaluation
target_scale.fit(df.iloc[:, tar_idx].values.reshape(-1, 1))
target_scalers[filename] = target_scale

# prepare dataset
inputs, target = data_prep(data_values, kw, in_idx=in_idx,
    tar_idx=tar_idx)
```

```


testing_percent = int(0.2*len(inputs)) # 20 percent will be used
    for testing

if len(trainX) == 0:
    trainX = inputs[:-testing_percent]
    trainY = target[:-testing_percent]
else:
    trainX = np.concatenate((trainX, inputs[:-testing_percent]))
    trainY = np.concatenate((trainY, target[:-testing_percent]))
testX[filename] = (inputs[-testing_percent:])
testY[filename] = (target[-testing_percent:])


# prepare train data
train_load = ds(torch.from_numpy(trainX), torch.from_numpy(trainY
    ))
train_dataloader = DL(train_load, shuffle=True, batch_size=bs,
    drop_last=True)

# checking GPU availability
is_cuda = torch.cuda.is_available()

# If GPU available then train on GPU
device = torch.device("cuda") if is_cuda else torch.device("cpu")





# --------------------Defining the LSTM Model--------------------

class LSTMModel(nn.Module):
    def __init__(self, input_dimension, hidden_dimension,
    output_dimension, layers):

        "LSTM model"

        super(LSTMModel, self).__init__()
        self.hidden_dimension = hidden_dimension
        self.layers = layers

        self.lstm = nn.LSTM(input_dimension, hidden_dimension,
    layers,
                            batch_first=True, dropout=0.1)
        # lstm layer
        self.fc = nn.Linear(hidden_dimension, output_dimension)


    def forward(self, x, h):
        # forward path
        out, h = self.lstm(x, h)
        out = self.fc(F.relu(out[:, -1]))
        return out, h

    def init_hidden(self, bs):
        w = next(self.parameters()).data

        h = (w.new(self.layers, bs, self.hidden_dimension).zero_
    ().to(device),
                      w.new(self.layers, bs, self.hidden_dimension).
    zero_().to(device))
        return h


# ------------------------Train Function-------------------------

def train_model(train_dataloader, learning_rate, hidden_dimension
    , layers, num_of_epoch):

    ## training parameters
    input_dimension = next(iter(train_dataloader))[0].shape[2]
    output_dimension = 1

    model = LSTMModel(input_dimension, hidden_dimension,
    output_dimension, layers)
    model.to(device)

    #   Mean Squared Error
    loss_criterion = nn.MSELoss()
    Adam_optimizer = torch.optim.Adam(model.parameters(), lr=
    learning_rate)

    model.train()  # set to train mode


    # Training start
    for iteration in range(1, num_of_epoch+1):

        h = model.init_hidden(bs)
        avg_loss_cal = 0.

        for data_values, target in train_dataloader:

            h = tuple([e.data for e in h])

            # as usual
            model.zero_grad()

            out, h = model(data_values.to(device).float(), h)
            loss = loss_criterion(out, target.to(device).float())

            # Perform backward differentiation
            loss.backward()
            Adam_optimizer.step()
            avg_loss_cal += loss.item()

        print(f"Epoch [{iteration}/{num_of_epoch}]: MSE: {
    avg_loss_cal/len(train_dataloader)}")
    return model
# Defining the model
model = train_model(train_dataloader, lr, 256,  layers,iters)


# --------------------------Test Phase---------------------------
def test_model(model, testX, testY, target_scalers):
    model.eval()
    predictions = []
    true_values = []

    # get data of test data for each state
    for filename in testX.keys():
        inputs = torch.from_numpy(np.array(testX[filename]))
        target = torch.from_numpy(np.array(testY[filename]))

        h = model.init_hidden(inputs.shape[0])

        # predict outputs
        out, h = model(inputs.to(device).float(), h)

        predictions.append(target_scalers[filename].
    inverse_transform(
            out.cpu().detach().numpy()).reshape(-1))

        true_values.append(target_scalers[filename].
    inverse_transform(
            target.numpy()).reshape(-1))

    # Merge all files
    f_outputs = np.concatenate(predictions)
    f_targets = np.concatenate(true_values)
    Evaluation_error = 100/len(f_targets) * np.sum(np.abs(
    f_outputs - f_targets) / (np.abs(f_outputs + f_targets))/2)
    print(f"Evaluation Error: {round(Evaluation_error, 3)}%")

    # list of targets/outputs for each state
    return predictions, true_values


predictions, true_values = test_model(model, testX, testY,
    target_scalers)


# --------------------------Visualizing--------------------------
plt.rcParams.update({'font.size': 18})
plt.figure(figsize=(12, 10))
plt.plot(predictions[0][-100:], "-r", color="r", label="LSTM
    Output", markersize=2)
```

```
plt.plot(true_values[0][-100:], color="b", label="True Value")
plt.xlabel('Time (Data points)')
plt.ylabel('Energy Consumption (MW)')
plt.title(f'Energy Consumption for North Illinois state')
plt.legend()
plt.savefig('./results/load_forecasting.png')
```

Listing 5.1 Load Forecasting code

Table 5.1 Explanation of the load forecasting code in Listing 5.1

Line number	Description
3–14	Importing PyTorch module
18–33	Setting hyperparameters
23	Sliding window size of 30 as describe by Fig. 5.2
37–38	The data folder contains the CSV file NI_hourly
42–62	Data generating function
72–104	Reading the North Illinois power generation data
107–109	Creating dataloader
112–115	Enable GPU if available
123–149	LSTM model as depicted in Fig. 5.1
154–192	A train function to perform forward, backward, and model optimization
194	Training function call
198–230	Evaluation function and generating prediction output
233–241	Plotting the actual and predicted output

Figure 5.4 shows the performance of the trained LSTM model in predicting the next energy consumption load at a given current state. The red curve—predicted by the LSTM model—closely matches the blue curve, which is the true load of the dataset with negligible error, proving the accuracy of the model for load forecasting. So, this LSTM model can be used to predict any sequential forecasting or future prediction problem, such as share market data, price of houses, and so on.

5.3 Fault/Anomaly Analysis

In electrical power systems, faults occur due to the passage of abnormal current flowing through one or more phases. For example, when a live wire touches a neutral wire, a *short circuit fault* occurs. Again, the current flows to the ground in the case of a *ground fault*. The detection and clearing of faults is important to ensure the reliable operation of the electrical power system. After successfully detecting the fault, the circuit breaker and other protective devices operate immediately to protect the system. ML techniques can be used for quick and accurate detection of electrical faults. Before learning the application of ML in fault detection, let us quickly review the different types of faults in electrical power systems.

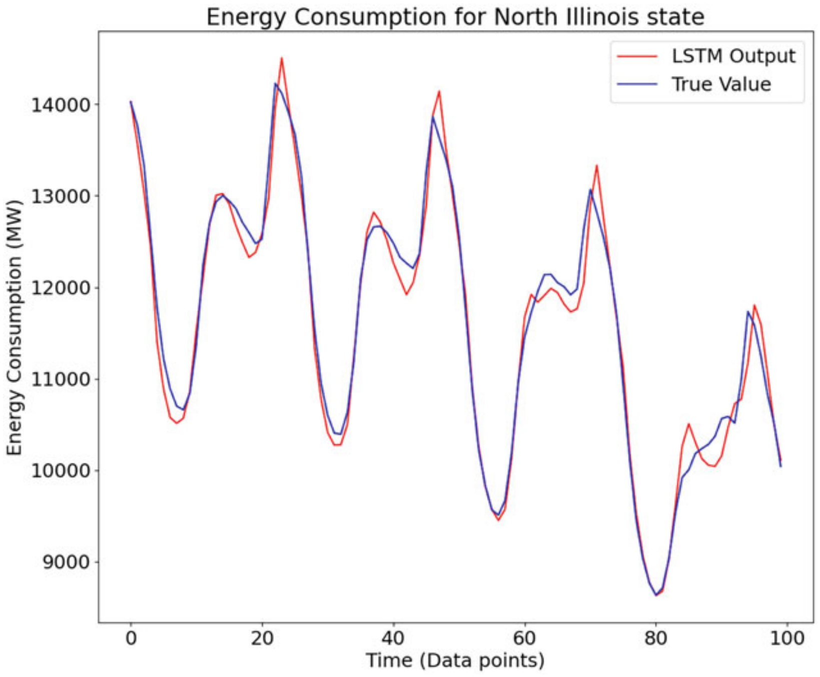


Fig. 5.4 The output of the LSTM model (red curve) closely matches the original data (blue curve)

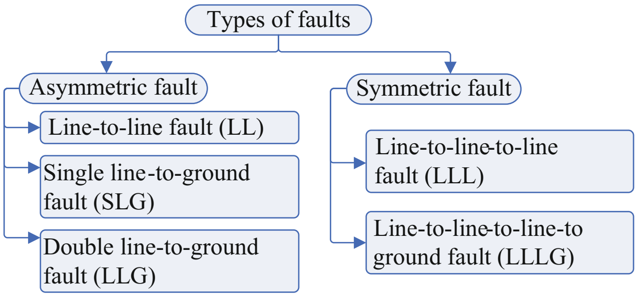

Types of faults
Asymmetric fault
Line-to-line fault (LL)
Single line-to-ground fault (SLG)
Double line-to-ground fault (LLG)
Symmetric fault
Line-to-line-to-line fault (LLL)
Line-to-line-to-line-to ground fault (LLLG)

Fig. 5.5 Different types of faults in three-phase transmission line

5.3.1 Different Types of Electrical Faults

The three-phase faults in electrical power systems can be broadly classified into asymmetric and symmetric faults. The complete classification is shown in Fig. 5.5.

1. **Asymmetric faults:** Asymmetric faults occur in only one or two phases in a three-phase system, for which these faults are more difficult to analyze. The asymmetric faults are of three types: line-to-line (LL) faults, single-line-to-

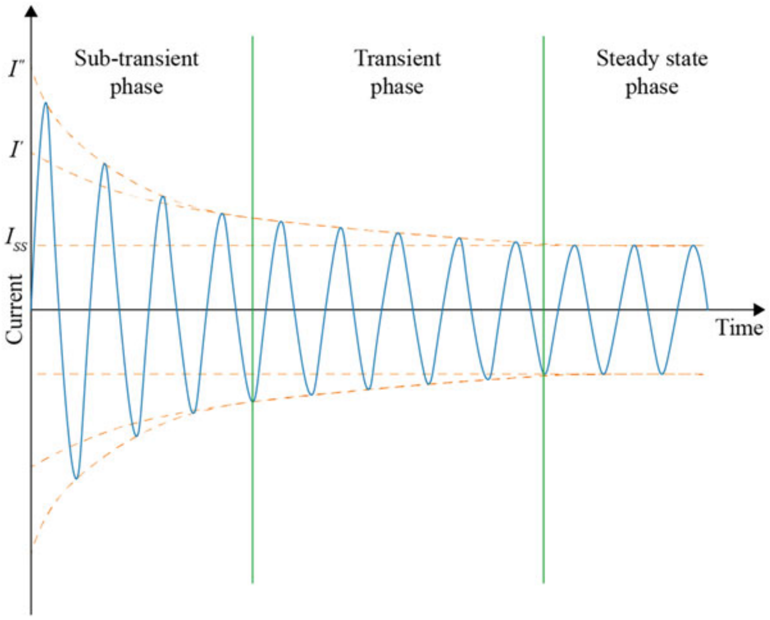


Fig. 5.6 Illustration of the sub-transient, transient, and steady-state periods during a short circuit fault condition

ground (SLG) faults, and double line-to-ground (LLG) faults. The SLG fault is the most common in power systems, making up about 65–70% of all faults. On the other hand, the LL faults comprise only 5–10% of the faults, and the LLG faults comprise 15–20% of the faults. The method of clearing the LL and LLG faults is similar, so they are grouped as one type of fault here.

2. **Symmetric faults:** Symmetric faults occur on all three phases of the three-phase system. So, the system remains balanced, but it causes severe damage to the equipment connected to the system as the largest amount of fault current flows in these faults. For this reason, the protective equipment of the system is rated based on the symmetric fault current ratings. The protective measures should have faster response capabilities because the amount of fault current is much higher for symmetrical faults than for asymmetric faults. Two types of symmetric faults are three-phase (3P) faults and triple-line-to-ground (LLLG) faults. Together, these faults constitute only 2–5% of the power system faults.

When a fault occurs in an electrical system, a significant change in current is observed. This change occurs in three main steps: the sub-transient, transient, and steady state. These steps are depicted in Fig. 5.6 and described as follows.

1. **Sub-transient state:** When a circuit is suddenly switched on, the voltage and the current do not immediately reach stable values. They start with a very high peak value. This temporary condition is known as the sub-transient state. The AC

current flowing through the circuit during the sub-transient state is known as the sub-transient current and is denoted by I'', shown in Fig. 5.6. The amplitude of this current can be up to ten times higher than the steady-state fault current. The sub-transient reactance X'' is expressed by the following equation, where E_A is the voltage:

$$X'' = \frac{E_A}{I''}. \tag{5.1}$$

2. **Transient state:** After a switching operation in a circuit, the state with the high peak values of the voltage and the current just after the sub-transient state is known as the transient state. This state is also temporary, but the peak values are slightly lesser than those in the sub-transient state. The AC current flowing through the circuit during the transient state is known as the transient current, denoted by I', shown in Fig. 5.6. The amplitude of this current can be up to five times higher than the steady-state fault current. The transient reactance X' is expressed by the following equation, where E_A is the voltage:

$$X' = \frac{E_A}{I'}. \tag{5.2}$$

3. **Steady State:** The steady state refers to the condition where the current and voltage within the circuit do not change. A definite amplitude and frequency are maintained. The AC current flowing through the circuit during the steady-state period is known as the steady-state current. It is denoted by I_{SS}, shown in Fig. 5.6. The steady-state reactance X_{SS} is expressed by the following equation, where E_A is the voltage:

$$X_{SS} = \frac{E_A}{I_{SS}}. \tag{5.3}$$

The knowledge of these three states is pivotal to understanding electrical fault analysis. In the next section, we will learn about electrical fault detection.

5.3.2 Fault Detection

We use a synthetic dataset, i.e., an artificially made dataset, for the fault detection task. The dataset is generated in a Simulink environment. Figure 5.7 shows the Simulink block diagram, which is used to generate the required dataset for the fault detection task. The simulation consists of a three-phase 60 Hz AC power source with 480 V phase-to-phase voltage. The three-phase fault block is used to generate the data during the fault condition (SLG, LL, LLG, and LLLG). The parameters of the fault block are shown in Fig. 5.8.

The faults can be emulated by checking the corresponding boxes in the "Fault between" section in the Block Parameters window. The output data are monitored

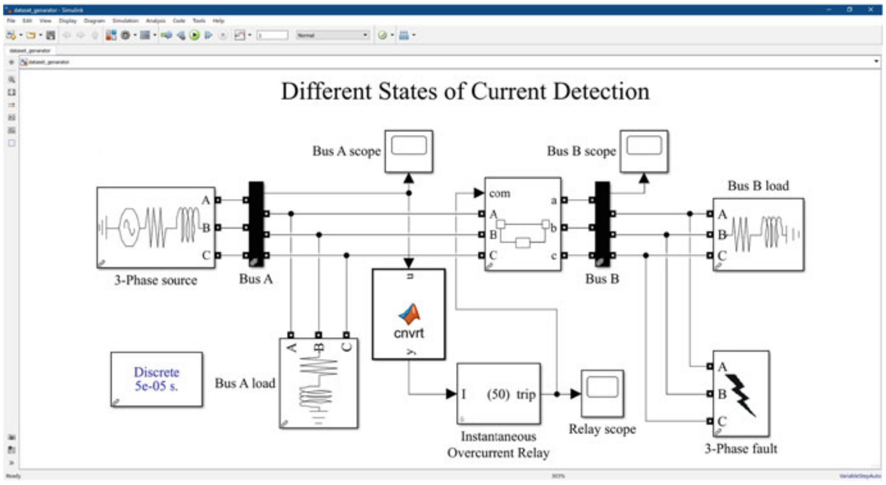
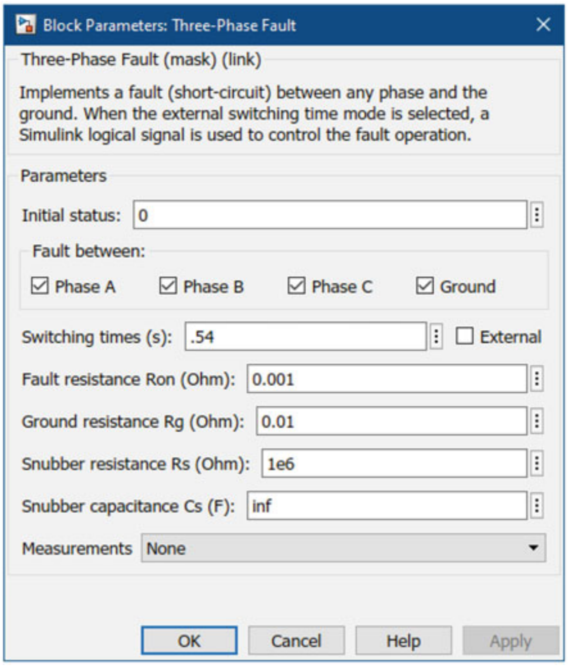


Fig. 5.7 Simulink block diagram for data generation


Block Parameters: Three-Phase Fault

Three-Phase Fault (mask) (link)

Implements a fault (short-circuit) between any phase and the ground. When the external switching time mode is selected, a Simulink logical signal is used to control the fault operation.

Parameters

Initial status: 0

Fault between:

☑ Phase A ☑ Phase B ☑ Phase C ☑ Ground

Switching times (s): .54 ☐ External

Fault resistance Ron (Ohm): 0.001

Ground resistance Rg (Ohm): 0.01

Snubber resistance Rs (Ohm): 1e6

Snubber capacitance Cs (F): inf

Measurements None

Fig. 5.8 Block parameters of the three-phase fault

through two different bus scopes—"Bus A Scope" and "Bus B Scope." The three-phase fault block is connected to Bus B to induce a fault in the system and is separated by an instantaneous overcurrent relay. The instantaneous overcurrent relay block was imported from MathWorks file exchange, developed by Rodney Tan [4]. The block has been modified to give its output in per-unit, and a block "cnvrt" has been implemented to match the per-unit system. Before and after the fault occurs, Bus A maintains a steady state.

The data from the steady state, transient, overload, and fault conditions are collected from the two bus scopes and are logged to create an array variable. Some samples of each current state are shown in Fig. 5.9. The array is then converted to

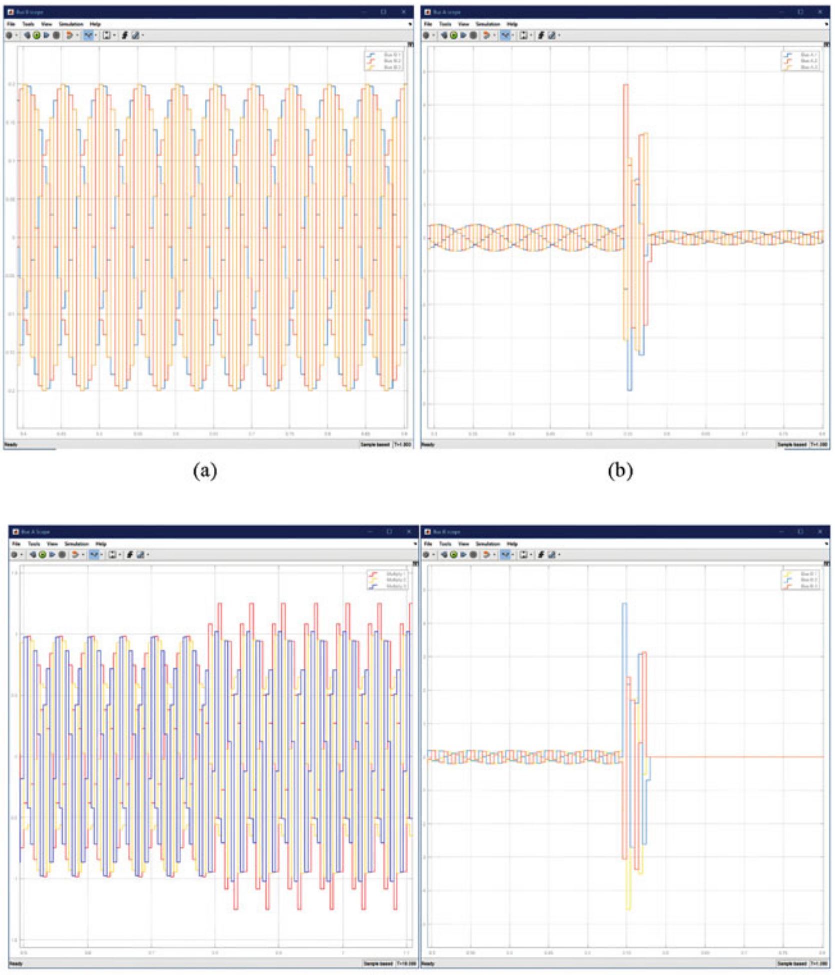


Fig. 5.9 Samples of different states of current. (**a**) Steady-state. (**b**) Transient. (**c**) Overload. (**d**) Fault condition

Table 5.2 Data participation of different current states

State	Steady-State	Transient	Overload	Fault
Class	0	1	2	3
Current Level	Up to 100%	115% to 175%	105%	More than 800%
Participation	50%	20%	10%	20%

Table 5.3 Dataset sample

ID	Class	0	0.005	0.01	...	...	0.99	0.995	1
1	1	−0.18512	0.39479	−0.05887	...	...	0.18512	−0.39479	0.058872
2	3	−0.10734	−0.12703	0.18584	...	...	0.10734	0.12703	−0.18584
...	...	...	...	...	...	...	...	...	...
499	0	−0.09221	0.197	−0.02954	...	...	0.092209	−0.197	0.029544
500	2	0.97743	−0.42154	−0.7169	...	...	−0.97743	0.42154	0.7169

a CSV file using MATLAB. The data in the dataset have been acquired by taking each current signal of 1 second time window, sampled at 5 ms. Table 5.2 shows the four current states considered for this dataset.

A sample of the synthesized dataset consisting of signal id, class, and time stamps is depicted in Table 5.3.

Programming Example 5.2
This fault detection task turns out to be a classification task, a four-class classification in our case. The features of this dataset are one-dimensional time series. Therefore, we will use a 1D CNN-based classifier for the fault detection task. Listing 5.2 narrates the code for the fault detection system, followed by its output, confusion matrix in Fig. 5.10 and explanation in Table 5.4.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from torch.utils.data import DataLoader as DL
from torch.utils.data import TensorDataset as ds
# ---------------Read the dataset & Preprocess-------------------
df_class = pd.read_csv("./data/Dataset.csv")

df_trans = df_class.iloc[:, 2:].T
```

```
df_norm = (df_trans - df_trans.min()) / (df_trans.max() -
    df_trans.min())


df = pd.concat([df_class.iloc[:, :2].T, df_norm]).T


encoder = LabelEncoder()
df['Class'] = encoder.fit_transform(df['Class'])

X = df.drop(['Class'], axis=1)
y = df['Class']


signals = torch.from_numpy(X.values).float()
signals = signals.unsqueeze(1)
target = torch.tensor(y, dtype=torch.long)


# -------------------Splitting the dataset-----------------------
X_train_test, X_test, y_train_test, y_test = train_test_split(
    signals, target, test_size=0.2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_test,
    y_train_test, test_size=0.2, random_state=42)


# ------------------------Define model--------------------------
class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__()
        self.conv1 = nn.Conv1d(1, 32, kernel_size=3)
        self.pool = nn.MaxPool1d(kernel_size=2)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(32 * ((signal_length // 2) - 1), 4)
        self.fc2 = nn.Linear(4, num_classes)
    def forward(self, x):
        x = self.conv1(x)
        x = self.pool(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.fc2(x)
        return x

# Set the hyperparameters
learn_rate = 0.001
bs = 1 #batch_size
no_of_epochs = 320
signal_length = X.shape[1]
num_classes = len(encoder.classes_)


# ----------------------Initialize model-------------------------
model = CNNModel()
```

```python

loss_criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learn_rate)

# Set the device to use GPU if available, otherwise use CPU
device = torch.device("cuda" if torch.cuda.is_available() else "
    cpu")
model.to(device)


training_dataset = ds(X_train, y_train)
training_loader = DL(training_dataset, batch_size=bs, shuffle=
    True)

validation_dataset = ds(X_val, y_val)
validation_loader = DL(validation_dataset, batch_size=bs)

test_dataset = ds(X_test, y_test)
test_loader = DL(test_dataset, batch_size=bs)
# Lists to store epoch and test accuracy
epoch_list = []
test_accuracy_list = []


# ------------------------Training loop--------------------------
for epoch in range(no_of_epochs):
    model.train()
    running_loss = 0.0
    for data, target in training_loader:
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        outputs = model(data)
        loss = loss_criterion(outputs, target)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    model.eval()
    val_loss = 0.0
    correct_flag = 0
    total = 0

    with torch.no_grad():
        for data, target in validation_loader:
            data, target = data.to(device), target.to(device)
            outputs = model(data)
            loss = loss_criterion(outputs, target)
            val_loss += loss.item()
            _, predicted_value = torch.max(outputs.data, 1)
            total += target.size(0)
            correct_flag += (predicted_value == target).sum().
    item()

    # Store epoch number and test accuracy
    epoch_list.append(epoch + 1)
    test_accuracy_list.append((correct_flag / total) * 100)

    print(f"Epoch No {epoch+1:3d}: Training Loss: {running_loss/
    len(training_loader):.4f},Validation Loss: {val_loss/len(
    validation_loader):.4f},\n              Validation Accuracy:
    {(correct_flag/total)*100:.2f}%")


# -------------------------Evaluation----------------------------
model.eval()
testing_loss = 0.0
correct_flag = 0
total = 0
predictions = []
true_labels = []
with torch.no_grad():
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)
        outputs = model(data)
        loss = loss_criterion(outputs, target)
        testing_loss += loss.item()
        _, predicted_value = torch.max(outputs.data, 1)
        total += target.size(0)
        correct_flag += (predicted_value == target).sum().item()
        predictions.extend(predicted_value.cpu().numpy())
        true_labels.extend(target.cpu().numpy())

print(f"Testing Loss: {testing_loss / len(test_loader):.4f},
    Testing Accuracy: {(correct_flag / total) * 100:.2f}%")

predictions = np.array(predictions)
true_labels = np.array(true_labels)

# Calculate confusion matrix
cm = confusion_matrix(true_labels, predictions)


# -------------------------Plotting------------------------------
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()
```

Listing 5.2 Code for the fault detection by different states of current detection

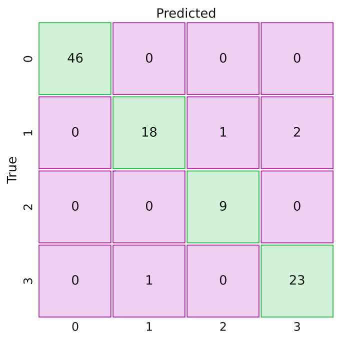


Fig. 5.10 Confusion matrix

Output of Listing 5.2:

```
Epoch   1: Train Loss: 1.1283, Validation Loss: 0.8917,
           Validation Accuracy: 70.00%
Epoch   2: Train Loss: 0.9092, Validation Loss: 0.8454,
           Validation Accuracy: 72.50%
Epoch   3: Train Loss: 0.7046, Validation Loss: 0.6727,
           Validation Accuracy: 78.75%
Epoch   4: Train Loss: 0.6034, Validation Loss: 0.6303,
           Validation Accuracy: 75.00%
Epoch   5: Train Loss: 0.5551, Validation Loss: 0.5597,
           Validation Accuracy: 75.00%
. . . . .
. . . . .
. . . . .
. . . . .
. . . . .
Epoch 318: Train Loss: 0.0275, Validation Loss: 0.0699,
           Validation Accuracy: 96.25%
Epoch 319: Train Loss: 0.0286, Validation Loss: 0.2215,
           Validation Accuracy: 92.50%
Epoch 320: Train Loss: 0.1019, Validation Loss: 0.1029,
           Validation Accuracy: 93.75%
Testing Loss: 0.2265, Testing Accuracy: 96.00%
```

Fault detection is the preliminary step to a safe and healthy network. The use of ML is a boon to accurately and quickly detect faults in the electrical power system. Next, we will study fault classification using ML.

5.3.3 Fault Classification

We should know the type of fault to implement the remedial action to eradicate the fault. So, in addition to fault detection, fault classification is also necessary. This section will apply artificial neural networks (ANN) to detect and classify faults in three-phase electrical transmission lines. Each fault is detected by observing the three-phase line currents and voltages, so these will be the input parameters for the ANN model. Also, we have seen that we now have three types of faults depending on their occurrence and method of clearing. So, there are a total of four parameters

Table 5.4 Explanation of the fault detection code example presented in Listing 5.2

Line number	Description
1–10	Importing library files
14–20	Data loading normalizing the data
23–24	Encoding the labels
26–32	Separating labels converting to tensors
36–37	Train, test, and validation data split
39–55	Defining CNN
57–62	Setting up hyperparameters
66	Initializing the CNN model
68–69	Loss function and optimizer defining
72–73	Code for using GPU, if available
76–83	Creating data loaders for training, testing, and validation data
85–86	List for accuracy vs. epoch plot
90–100	Training loop optimizes
102	Initialize model evaluation
107–115	After each epoch, the model is evaluated on the validation data from val_loader to calculate the validation loss and accuracy
118–119	Update epoch and accuracy list
121	Print Epoch number, train loss, validation loss, accuracy
124–130	Evaluation phase initialization
131–143	Tests predicted data with testing dataset
145–150	From “true_labels” and “predictions” creating confusion matrix
152–157	Plotting confusion matrix

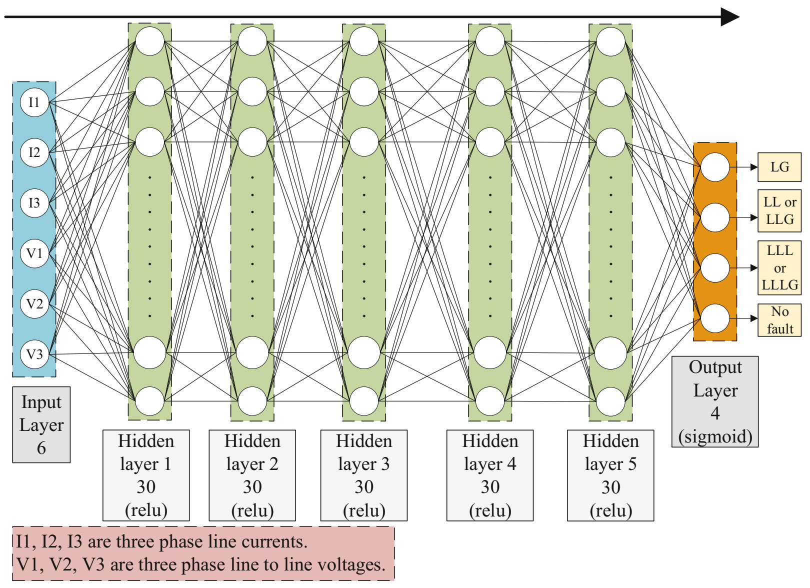


Fig. 5.11 ANN model used in Listing 5.3 for fault classification in transmission line

as output from the ANN model—three for the fault types and one for the no-fault or healthy case. The fault is detected from the largest value of the output layer. We will use five hidden layers—each of 30 units—in the ANN as shown in Fig. 5.11. The accuracy of the model is 100%, which is required for the safety of the transmission line.

Programming Example 5.3
Listing 5.3 shows the code for this, and its explanation is provided in Table 5.5.

The data for this example has been obtained from simulation using MATLAB or Simulink. Two 400 kV (rms) generators have been used to simulate the model. The total transmission length is 600 km, and the fault is considered at the middle of the line. The simulation is not discussed here since it is not relevant to the scope of this book. The dataset has been uploaded into the data folder of the GitHub repository named "data_fault_detection.csv"; this file contains the value of the 3-phase line currents and voltages and their corresponding one hot encoded vector for four types of output.

```python
#Source: https://www.kaggle.com/esathyaprakash/electrical-fault-
    detection-and-classification
#Paper: https://springerplus.springeropen.com/articles/10.1186/
    s40064-015-1080-x


# -------------------------Torch Modules-------------------------
import numpy as np
import pandas as pd
import torch.nn as nn
import math
import torch
from torch.nn import init
import torch.utils.data as data_utils
import torch.optim as optim
from torchvision import models
import torch.nn.functional as F


# ---------------------------Variables---------------------------
BATCH_SIZE = 128
Iterations = 100
learning_rate = 0.01


# ------------------Commands to Prepare Dataset------------------
data_set = pd.read_csv("./data/data_fault_detection.csv")
torch.manual_seed(18)

target_value = torch.tensor(data_set["Fault_type"].values.astype(
    np.float32))
input_value = torch.tensor(data_set.drop(columns = ["Fault_type"
    ]).values.astype(np.float32))

```

```
data_tensor = data_utils.TensorDataset(input_value, target_value)

train_set_size = math.floor(input_value.size()[0]*0.90)
test_set_size = input_value.size()[0]-train_set_size

train_set, test_set = torch.utils.data.random_split(data_tensor,
    [train_set_size, test_set_size])

train_loader = data_utils.DataLoader(dataset = train_set,
    batch_size = BATCH_SIZE, shuffle = True)
test_loader = data_utils.DataLoader(dataset = test_set,
    batch_size = BATCH_SIZE, shuffle = True)


# -------------------------Defining ANN-------------------------
class ANN(nn.Module):
    def __init__(self):
        super(ANN, self).__init__()
        self.l1 = nn.Linear(6, 30)
        self.relu = nn.ReLU()
        self.l2 = nn.Linear(30, 30)
        self.l3 = nn.Linear(30, 4)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.l1(x)
        x = self.relu(x)

        x = self.l2(x)
        x = self.relu(x)

        x = self.l2(x)
        x = self.relu(x)

        x = self.l2(x)
        x = self.relu(x)

        x = self.l2(x)
        x = self.relu(x)

        x = self.l3(x)
        x = self.sigmoid(x)
        return x

# defining ANN model
model = ANN()
## Loss function
criterion = torch.nn.CrossEntropyLoss()

# definin which paramters to train only the ANN model parameters
optimizer = torch.optim.SGD(model.parameters(), lr =
    learning_rate)

```

```python
# defining the training function
def train(model, optimizer, criterion,epoch):
    model.train()
    total_trained_data = 0
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target.type(torch.LongTensor))
        loss.backward()
        optimizer.step()
        total_trained_data += len(data)
        if (batch_idx !=0 and batch_idx % 30 == 0) or total_trained_data == len(train_loader.dataset):
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, total_trained_data, len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))


# ---------------------------Evaluation--------------------------
def test(model, criterion, val_loader, epoch):
    model.eval()
    test_loss = 0
    correct = 0

    with torch.no_grad():
        for batch_idx, (data, target) in enumerate(val_loader):
            output = model(data)
            test_loss += criterion(output, target.type(torch.LongTensor)).item()
            pred = output.max(1, keepdim=True)[1]
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(val_loader.dataset)
    if epoch:
        print('\nTrain set: Average loss: {:.4f}, Accuracy: {}/{} ({:.4f}%)\n'.format(
            test_loss, correct, val_loader.sampler.__len__(),
            100. * correct / val_loader.sampler.__len__() ))
    else:
        print('Test set: Average loss: {:.4f}, Accuracy: {}/{} ({:.4f}%)\n'.format(
            test_loss, correct, val_loader.sampler.__len__(),
            100. * correct / val_loader.sampler.__len__() ))

## training the ANN
for i in range(Iterations):
    train(model, optimizer,criterion,i)
    test(model, criterion, train_loader, i)
    test(model, criterion, test_loader, False)
```

```
def pred(data): #prediciton function for single data
    output = model(data)
    output = output.tolist()
    index = output.index(max(output))

    if index == 3:
        print("No fault detected.")
    else:
        string = ["SLG", "LL or LLG", "LLL or LLLG"]
        type_fault = string[index]
        print(f"Fault detected. Type of fault is {type_fault}.")

for input_data, _ in test_loader:
    for value in input_data:
        pred(value)
```

Listing 5.3 Transmission line fault detection using ANN [5,6]

Output after line 121 of Listing 5.3:

```
Train Epoch: 0 [3968/16093 (24%)] Loss: 1.119744
Train Epoch: 0 [7808/16093 (48%)] Loss: 1.088864
Train Epoch: 0 [11648/16093 (71%)] Loss: 0.936920
Train Epoch: 0 [15488/16093 (95%)] Loss: 0.947128
Train Epoch: 0 [16093/16093 (99%)] Loss: 0.918876
Test set: Average loss: 0.0073,
Accuracy: 11987/16093 (74.4858%)

Train Epoch: 1 [3968/16093 (24%)] Loss: 0.967388
Train Epoch: 1 [7808/16093 (48%)] Loss: 0.879457
Train Epoch: 1 [11648/16093 (71%)] Loss: 0.887392
Train Epoch: 1 [15488/16093 (95%)] Loss: 0.875771
Train Epoch: 1 [16093/16093 (99%)] Loss: 0.876310

Train set: Average loss: 0.0069,
Accuracy: 11950/16093 (74.2559%)
......
......
......
......
Train Epoch: 98 [3968/16093 (24%)] Loss: 0.743768
Train Epoch: 98 [7808/16093 (48%)] Loss: 0.743760
Train Epoch: 98 [11648/16093 (71%)] Loss: 0.743745
Train Epoch: 98 [15488/16093 (95%)] Loss: 0.743782
Train Epoch: 98 [16093/16093 (99%)] Loss: 0.743839

Train set: Average loss: 0.0058,
Accuracy: 16093/16093 (100.0000%)
```

```
Train Epoch: 99 [3968/16093 (24%)] Loss: 0.743809
Train Epoch: 99 [7808/16093 (48%)] Loss: 0.743778
Train Epoch: 99 [11648/16093 (71%)] Loss: 0.743793
Train Epoch: 99 [15488/16093 (95%)] Loss: 0.743708
Train Epoch: 99 [16093/16093 (99%)] Loss: 0.743869

Train set: Average loss: 0.0058,
Accuracy: 16093/16093 (100.0000%)

Test set: Average loss: 0.0058,
Accuracy: 1789/1789 (100.0000%)
```

Output after line 139 of Listing 5.3:

```
Fault detected. Type of fault is LL or LLG.
Fault detected. Type of fault is SLG.
Fault detected. Type of fault is SLG.
Fault detected. Type of fault is LLL or LLLG.
......
......
......
......
Fault detected. Type of fault is LLL or LLLG.
No fault detected.
Fault detected. Type of fault is SLG.
Fault detected. Type of fault is SLG.
Fault detected. Type of fault is LLL or LLLG.
Fault detected. Type of fault is LLL or LLLG.
```

Table 5.5 Explanation of the fault classification code example presented in Listing 5.3

Line number	Description
5–12	Importing Matplotlib, Pandas, PyTorch
18–21	Defining batch size, iteration number, and learning rate
24–39	Preparing dataset using pandas. Test-train split, data shuffle have also been done here
42–70	Defining ANN model as presented in Fig. 5.11
72–75	Defining loss function and model optimizer
77–94	Function to train the model has been defined here
97–118	Function to evaluate the model during and after the training
120–124	Training of the model begins here
126–136	Function to predict fault from data
138–140	A prediction of the faults has been demonstrated

Thus, we have studied different types of electrical faults and devised ML models to detect and classify them when they occur. In the next section, we will study partial discharge detection using ML.

5.3.4 Partial Discharge Detection

In electrical systems, a partial discharge occurs when the space between any two conductors is very small, and the voltage across the space is high enough to cause a localized dielectric breakdown. Electrical faults can lead to a partial discharge phenomenon in the power systems. Partial discharge gradually damages the power lines, and a prolonged partial discharge may lead to catastrophic failure and power outage. Overhead high voltage and medium voltage power lines span hundreds of miles in a typical grid system. A partial discharge might occur at any part of the line, but it is impractical to continuously inspect the power lines manually for any type of minor physical damage or disturbance that does not readily destroy the lines. So, it is essential to automatically detect any occurrence of partial discharge as early as possible to take necessary preventive measures.

A potential solution to this problem is to train a classifier on power line signals to detect the faulty signal and inspect that respective power line. Here, we will use the dataset from the VSB Power Line Fault Detection competition [7] for partial discharge detection.

Programming Example 5.4
We will make a 1D convolutional neural network (CNN)-based approach to train our partial discharge classifier shown in Listing 5.4 (explained in Table 5.6). A subset of the dataset is considered, maintaining a ratio between the number of positive and negative samples. If the full dataset were taken for training, it would overfit the training. From the balanced combination of the positive and negative samples, the training and testing datasets are created. All the data are reshaped to match the input layer of the CNN layer. The train and test dataset are created utilizing the train and test dataloader. A CNN model is defined, which incorporates convolutional, fully connected, and activation layers. As this is a binary classification task, the `BCELoss` is used as the loss function and the `Adam` optimizer is employed for optimization. Afterward, evaluation is done on both the training and test datasets. The code is written for the Python environment, but the computational time will be reduced if it is executed in any cloud IPython environment. Sometimes, the system cannot allocate enough memory to execute the code. In such cases, switch to cloud platforms.

```
# Dataset: https://www.kaggle.com/competitions/vsb-power-line-
    fault-detection/data
# Download train.parquet and metadata_train.csv files

```

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as Data
import numpy as np
import pandas as pd
import pyarrow.parquet as pq
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split


# ---------------Read the dataset & Preprocess-------------------
# Download from the dataset link and replace the path
subset_train = pq.read_pandas('../input/vsb-power-line-fault-
    detection/train.parquet',
                columns=[str(i) for i in range(5000)]).to_pandas()
# Read half of the data among 800000 samples
subset_train = subset_train.iloc[200000:600000, :]
subset_train.info()

metadata_train = pd.read_csv('../input/vsb-power-line-fault-
    detection/metadata_train.csv')
metadata_train.info()

# Reduce the sample sizes to stay within memory limits
S_decimation = subset_train.iloc[0:25000:8, :]
small_subset_train = S_decimation
small_subset_train = small_subset_train.transpose()
small_subset_train.index = small_subset_train.index.astype(np.
    int32)
train_dataset = metadata_train.join(small_subset_train, how='
    right')

### Uncomment the following to train on the full dataset
# subset_train = subset_train.transpose()
# subset_train.index = subset_train.index.astype(np.int32)
# train_dataset = metadata_train.join(subset_train, how='right')


# ----------Separating positive and negative samples-------------
positive_samples = train_dataset[train_dataset['target'] == 1]
positive_samples = positive_samples.iloc[:, 3:]

print("positive_samples data shape: " + str(positive_samples.
    shape) + "\n")
positive_samples.info()

y_train_pos = positive_samples.iloc[:, 0]
X_train_pos = positive_samples.iloc[:, 1:]
scaler = StandardScaler()
```

```python
scaler.fit(X_train_pos.T)  # Normalize the data set
X_train_pos = scaler.transform(X_train_pos.T).T

negative_samples = train_dataset[train_dataset['target'] == 0]
negative_samples = negative_samples.iloc[:, 3:]

print("negative_samples data shape: " + str(negative_samples.
    shape) + "\n")
negative_samples.info()

y_train_neg = negative_samples.iloc[:, 0]
X_train_neg = negative_samples.iloc[:, 1:]
scaler.fit(X_train_neg.T)
X_train_neg = scaler.transform(X_train_neg.T).T


# -------------------Splitting the dataset-----------------------
X_train_pos, X_valid_pos, y_train_pos, y_valid_pos =
    train_test_split ( X_train_pos,
                                    y_train_pos,
                                    test_size=0.3,
                                    random_state=0,
                                    shuffle=False)
X_train_neg, X_valid_neg, y_train_neg, y_valid_neg =
    train_test_split ( X_train_neg,
                                    y_train_neg,
                                    test_size=0.3,
                                    random_state=0,
                                    shuffle=False)

print("X_train_pos data shape: " + str(X_train_pos.shape))
print("X_train_neg data shape: " + str(X_train_neg.shape))
print("y_train_pos data shape: " + str(y_train_pos.shape))
print("y_train_neg data shape: " + str(y_train_neg.shape))

print("\nX_valid_pos data shape: " + str(X_valid_pos.shape))
print("X_valid_neg data shape: " + str(X_valid_neg.shape))
print("y_valid_pos data shape: " + str(y_valid_pos.shape))
print("y_valid_neg data shape: " + str(y_valid_neg.shape))


# -----------Combine positive and negative samples---------------
# Keeping the the samples balanced
# 550 and 270 is used to make sure
# a correct ratio of positive and negative samples
def combine_pos_and_neg_samples(pos_samples, neg_samples, y_pos,
    y_neg):
    X_combined = np.concatenate((pos_samples, neg_samples))
    y_combined = np.concatenate((y_pos, y_neg))
    combined_samples = np.hstack((X_combined, y_combined.reshape(
    y_combined.shape[0], 1)))
    np.random.shuffle(combined_samples)
    return combined_samples
```

```


train_samples = combine_pos_and_neg_samples(X_train_pos,
                                            X_train_neg[:550, :],
                                            y_train_pos,
                                            y_train_neg[:550])
X_train = train_samples[:, :-1]
y_train = train_samples[:, -1]

print("X_train data shape: " + str(X_train.shape))
print("y_train data shape: " + str(y_train.shape))
print("train_samples data shape: " + str(train_samples.shape))

validation_samples = combine_pos_and_neg_samples(X_valid_pos,
                                                 X_valid_neg[:270, :],
                                                 y_valid_pos,
                                                 y_valid_neg[:270])
X_valid = validation_samples[:, :-1]
y_valid = validation_samples[:, -1]

print("\nX_valid data shape: " + str(X_valid.shape))
print("y_valid data shape: " + str(y_valid.shape))
print("validation_samples data shape: " + str(validation_samples.
    shape))


# -----Reshape training and validation data for input layer------
X_train = X_train.reshape(-1, 1, 3125)
X_valid = X_valid.reshape(-1, 1, 3125)
print("X_train data shape: " + str(X_train.shape))
print("X_valid data shape: " + str(X_valid.shape))
print("y_train data shape: " + str(y_train.shape))
print("y_valid data shape: " + str(y_valid.shape))

X_valid = X_valid.astype(np.float32)
y_valid = y_valid.astype(np.float32)
X_train = X_train.astype(np.float32)
y_train = y_train.astype(np.float32)
print("Type of data: " + str(X_train.dtype))


# --------------------Normalize feature------------------------
print("Total samples in train dataset: " + str(np.sum(y_train)))
print("Total samples in validation dataset: " + str(np.sum(
    y_valid)))

def feature_normalize(data):
    mu = np.mean(data, axis=0)
    std = np.std(data, axis=0)
    return (data - mu) / std


X_valid = feature_normalize(X_valid)
```

```
X_train = feature_normalize(X_train)


# ---------------------Define dataloader-------------------------
class torch_Dataset(Data.Dataset):
    def __init__(self, x, y):
        self.x = torch.from_numpy(x)
        self.y = torch.from_numpy(y)

    def __getitem__(self, index):
        data = (self.x[index], self.y[index])
        return data

    def __len__(self):
        return len(self.y)


def training_loader(train_data, batch_size, shuffle):
    return torch.utils.data.DataLoader(train_data, batch_size,
    shuffle)

Train_dataset = torch_Dataset(X_train, y_train)
test_dataset = torch_Dataset(X_valid, y_valid)
train_loader = training_loader(Train_dataset, batch_size=1,
    shuffle=False)
test_loader = training_loader(test_dataset, batch_size=1, shuffle
    =False)


# ------------------------Define model---------------------------
class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__()

        # Convolutional layers
        self.conv1 = nn.Conv1d(1, 32, kernel_size=3, stride=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool1d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv1d(32, 64, kernel_size=3, stride=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool1d(kernel_size=2, stride=2)

        # Fully connected layers
        self.fc1 = nn.Linear(64 * 779, 128)
        self.relu3 = nn.ReLU()
        self.fc2 = nn.Linear(128, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)
```

```python

        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool2(x)

        x = x.view(x.size(0), -1)

        x = self.fc1(x)
        x = self.relu3(x)
        x = self.fc2(x)
        x = self.sigmoid(x)

        return x


model = CNNModel()
print(model)
optimizer = optim.Adam(model.parameters())
criterion = nn.BCELoss()


# ------------------------Training loop--------------------------
for epoch in range(10):
    losses = []
    for data, target in train_loader:
        output = model(data)
        target = target.view([1, 1])
        loss = criterion(output, target)
        losses.append(loss.item())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch + 1}: loss {sum(losses) / len(losses)}")


# -------------------------Evaluation----------------------------
def validate(model, train_loader, val_loader):
    accdict = {}
    for name, loader in [("train dataset", train_loader), ("test
    dataset   ", val_loader)]:
        correct = 0
        total = 0
        predictions = []
        true_labels = []
        with torch.no_grad():
            for imgs, labels in loader:
                imgs = imgs.float()
                outputs = model(imgs)
                predicted = torch.max(outputs)
                if (predicted > 0.5):
                    fault_detected = 1
                else:
                    fault_detected = 0
                total += labels.shape[0]
                correct += int((fault_detected == labels).sum())
                predictions.append(fault_detected)
                true_labels.append(round(labels.item()))


        print("Accuracy {0}: {1:.2f}(%)".format(name, 100 * (
    correct / total)))
        accdict[name] = correct / total

validate(model, train_loader, test_loader)
```

Listing 5.4 Power line fault or partial discharge detection

Output of Listing 5.4:

————-Read the dataset and Preprocess——————-

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400000 entries, 200000 to 599999
Columns: 5000 entries, 0 to 4999
dtypes: int8(5000)
memory usage: 1.9 GB
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8712 entries, 0 to 8711
Data columns (total 4 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   signal_id       8712 non-null   int64
 1   id_measurement  8712 non-null   int64
 2   phase           8712 non-null   int64
 3   target          8712 non-null   int64
dtypes: int64(4)
memory usage: 272.4 KB
```

————-Separating positive and negative samples——————-

```
positive_samples data shape: (332, 3126)

<class 'pandas.core.frame.DataFrame'>
Int64Index: 332 entries, 3 to 4940
Columns: 3126 entries, target to 224992
dtypes: int64(1), int8(3125)
memory usage: 1018.4 KB

negative_samples data shape: (4668, 3126)

<class 'pandas.core.frame.DataFrame'>
Int64Index: 4668 entries, 0 to 4999
Columns: 3126 entries, target to 224992
```

```
dtypes: int64(1), int8(3125)
memory usage: 14.0 MB
```

——————-Splitting the dataset————————

```
X_train_pos data shape: (232, 3125)
X_train_neg data shape: (3267, 3125)
y_train_pos data shape: (232,)
y_train_neg data shape: (3267,)

X_valid_pos data shape: (100, 3125)
X_valid_neg data shape: (1401, 3125)
y_valid_pos data shape: (100,)
y_valid_neg data shape: (1401,)
```

————Combine positive and negative samples——————

```
X_train data shape: (782, 3125)
y_train data shape: (782,)
train_samples data shape: (782, 3126)

X_valid data shape: (370, 3125)
y_valid data shape: (370,)
validation_samples data shape: (370, 3126)
```

——Reshape training and validation data for input layer——

```
X_train data shape: (782, 1, 3125)
X_valid data shape: (370, 1, 3125)
y_train data shape: (782,)
y_valid data shape: (370,)
Type of data: float32
```

————————Normalize feature—————————-

```
Total samples in train dataset: 232.0
Total samples in validation dataset: 100.0
```

—————————Define model—————————

```
CNNModel(
  (conv1): Conv1d(1, 32, kernel_size=(3,), stride=(1,))
  (relu1): ReLU()
  (pool1): MaxPool1d(kernel_size=2, stride=2, padding=0,
                     dilation=1, ceil_mode=False)
  (conv2): Conv1d(32, 64, kernel_size=(3,), stride=(1,))
  (relu2): ReLU()
  (pool2): MaxPool1d(kernel_size=2, stride=2, padding=0,
                     dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=49856, out_features=128, bias=True)
  (relu3): ReLU()
  (fc2): Linear(in_features=128, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

```
----------------Training loop----------------
Epoch 1: loss 0.8164422646736628
Epoch 2: loss 0.4948220962638653
Epoch 3: loss 0.1427917082643386
Epoch 4: loss 0.05073368033861952
Epoch 5: loss 0.0036601704486090457
Epoch 6: loss 0.00039702328483650052
Epoch 7: loss 0.00012066281708122702
Epoch 8: loss 6.312557864259138e-05
Epoch 9: loss 3.8254333689510956e-05
Epoch 10: loss 2.3528139867673672e-05
----------------Evaluation----------------
Accuracy train dataset: 100.00(%)
Accuracy test dataset  : 73.51(%)
```

From the result, we can see that our classifier has a 100% accuracy on the training data and 73.51% accuracy on the test data. This happened due to class imbalance and noise in the data. The test accuracy can be improved by applying proper denoising techniques on the dataset and some advanced ML concepts that are beyond this chapter's scope.

5.4 Future Trend Prediction in Renewable Energy Systems

Predicting the future price, product usage, demand or supply, etc. is very important for us to prepare for the best and worst possible situations in the future. This section demonstrates two real-life examples of future trend prediction in renewable energy systems—solar PV installed capacity prediction and wind power output prediction.

5.4.1 Solar PV Installed Capacity Prediction

Solar energy is the most abundant resource in the world. According to the latest findings by the International Energy Agency, solar photovoltaic (PV) system is on its way to surpassing natural gas by 2026 and coal by 2027 in terms of installed power capacity [8]. In 2022, solar PV systems had the highest growth among all renewable energy systems, consistent with the growth rate required for reaching Net Zero Emissions by 2050 [8]. Solar PV produces 4.5% of the global electricity and remains the third largest renewable electricity technology after hydropower and wind [8]. The annual solar PV capacity will continue to increase.

Table 5.6 Explanation of the power line fault detection code presented in Listing 5.4

Line number	Description
4–16	Import the necessary libraries
19–20	Read the dataset
23–24	Take half of the dataset for easier processing
26–27	Load metadata file
30–31	Select a subset of rows from the subset_train
32–34	It is then transposed, its index is converted to an integer data type, and it is joined with the metadata_train
36–39	Code snippet for using full data, for higher memory or computational limit
43–44	Select rows from the train_dataset where “target” = 1 (i.e., positive samples)
46–47	Shows information about the positive_samples
49–50	Separate the positive_samples into a target variable and a feature matrix
51–52	A StandardScaler is then fit to the transposed feature matrix
53	The feature matrix is then transformed using the fitted scaler and transposed
55–64	Repeat the process for negative samples
67–78	Split the positive and negative samples into training and validation sets
90–99	Define a function that concatenates the positive and negative samples
102–105	Use the function to combine the positive and negative samples
106–107	Separate target variable y_train and feature matrix X_train
113–118	Repeat the process for validation_samples
126–127	Reshape the training and validation feature matrices to have a shape of (-1, 1, 3125), where -1 means that NumPy will decide the best size/dimension
128–131	Information about the shapes of these arrays is then printed
133–137	Convert the data type of all variables to 32-bit floating point
140–151	Define a function that normalizes the features
155–158	Define a Dataset class that converts feature and target matrix to tensors
160–165	__getitem__ and __len__ methods to allow indexing and to get length
168–169	A function defined that create DataLoader object for the given dataset
171–174	Instances of torch_Dataset created and passed to the training_loader function
179–195	The __init__ method of this class defines the layers of the model, including 2 convolutional layers with activation, pooling, and fully connected layers
197–213	The forward method defines the forward pass of the model by passing the input x through each layer in turn and returning the output of the final layer
216–217	Create an instance of CNNModel class and print its architecture
218–219	Adam optimizer and binary cross entropy loss function is instantiated
223–233	Define a training loop that runs for 10 epochs in which loss is computed using the previously defined BCE loss function
237	Define a function that evaluates the model’s performance on both datasets
239	The function iterates over both data loaders and computes predictions
244	torch.no_grad() is used to avoid changing gradients while updating weights, as this would interfere with backpropagation
245–260	The number of correct predictions and the accuracy is computed

In this section, we will predict the installed capacity of solar PV systems in the coming years. An existing annual solar PV module installed capacity dataset is used for this example [9].

Programming Example 5.5
From Listing 5.5 (explained in Table 5.7) it can be seen that each year's data is read from a CSV file and separated into two arrays for further processing. One array contains the year value, and the other array contains the respective PV solar module installation value. A portion of the data is used to train the model, and a portion is kept for testing purposes. Then, we fit year values to a third-order polynomial. After setting the parameters of the Bayesian ridge regression, the model is fitted, and we get the predictions based on these. The training, test, and predicted data are plotted in Fig. 5.12. Next, the data from 1995 to 2020 are fitted to the model using a second-order polynomial and predicted up to 2040.

```
from sklearn.linear_model import BayesianRidge
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
from numpy import sqrt
import pandas as pd
import numpy as np


# --------------------------Reading Data-------------------------
data = pd.read_csv('./data/PV.csv',header=0)
X = data.iloc[:, 0].values  # values converts it into a numpy
    array
Y = data.iloc[:, 1].values
xtrain, xtest, ytrain, ytest=train_test_split(X, Y, test_size
    =0.1, shuffle=False)


# -------------------------Data Processing-----------------------
n_order = 3
Xtrain = np.vander(xtrain, n_order + 1, increasing=True)
Xtest = np.vander(xtest, n_order + 1, increasing=True)


# -----------------------Setting Parameter-----------------------
reg = BayesianRidge(tol=1e-18, fit_intercept=False, compute_score
    =True)


# -------------------------Fit and Predict-----------------------
reg.set_params(alpha_init=0.1, lambda_init=1e-15)
reg.fit(Xtrain, ytrain)
ymean = reg.predict(Xtest)

```

```

# ----------------------------Plotting----------------------------
plt.figure(figsize=(12,4))
plt.subplot(1, 2, 1)
plt.plot(xtest, ytest, color="blue", label="Test data")
plt.scatter(xtrain, ytrain, s=50, alpha=0.5, label="Training data
    ")
plt.plot(xtest, ymean, color="red", label="Predicted data")
plt.legend()


# ----------------------Fitting on Full Data----------------------
n_order = 2
Xfull = np.vander(X, n_order + 1, increasing=True)
Xpred = np.array([2025, 2030, 2035, 2040])
XPred = np.vander(Xpred, n_order + 1, increasing=True)


# Setting parameter
bay = BayesianRidge(tol=1e-18, fit_intercept=False, compute_score
    =True)


# Fit & predict
bay.set_params(alpha_init=0.1, lambda_init=1e-30)
bay.fit(Xfull, Y)
Yfull = bay.predict(XPred)


# Plotting
plt.subplot(1, 2, 2)
plt.scatter(X, Y, s=50, alpha=0.5, label="Training data")
plt.plot(Xpred, Yfull, color="red", label="Predicted data")
plt.legend()
plt.show()
```

Listing 5.5 Solar PV installed capacity prediction using the Bayesian ridge regression model

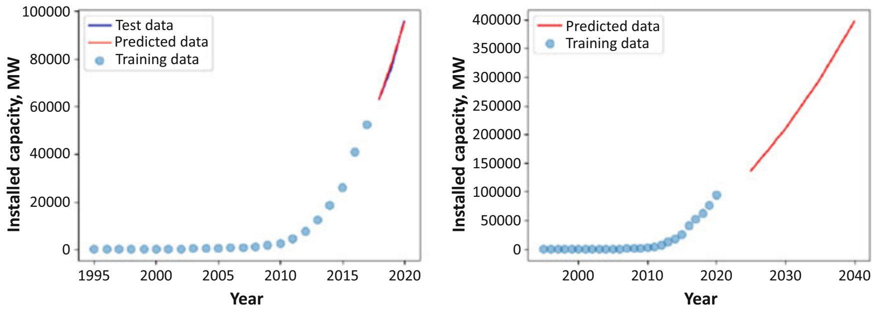


Fig. 5.12 Solar PV installed capacity future trend prediction with the input value

Table 5.7 Explanation of the solar PV installation future trend prediction code example presented in Listing 5.5

Line number	Description
1–7	Importing NumPy, pandas, and some of sklearn modules
11	Read data from csv file
12–13	Separate year and PV installation into different array
14	Separate train and test data
18–24	Preparing data and setting parameters to fit to a model
28–30	Fit and predict the test values
34–39	Plotting train data, test data, and predicted data
43–56	Fit the whole dataset and predict up to 2040
60–64	Plotting overall result for final prediction

5.4.2 Wind Power Output Prediction

Wind energy is undoubtedly a major renewable energy source, providing a pathway toward net zero carbon emission. The energy share of wind power has been growing rapidly since the early 2000s. According to the International Renewable Energy Agency (IRENA), the global wind generation capacity has increased by a factor of 98 in the last two decades—from around 7.5 GW in 1997 to around 733 GW in 2018 [10]. From 2009 to 2019, the overall wind power production increased by a factor of 5.2, reaching 1412 TWh annual production [10]. Wind electricity generation increased by a record 265 TWh in 2022, becoming the second-highest power generation sector among all renewable energy sources after solar PV [11].

Electricity generated by wind power is intermittent due to the wind's changing speed, direction, and flow. Therefore, power forecasting is essential to tackle wind power fluctuation in power system dispatch. Various predictive modeling has been used and implemented, from statistical methods to neural networks, for wind power forecasting.

Programming Example 5.6
Listing 5.6 shows the wind power predictor model code followed by its output and explanation in Table 5.8. Random forest regression has been used for wind power forecasting in the example presented in this section. The Texas Wind dataset from Kaggle has been used here [12]. From the dataset, 30% has been used for testing, and the rest has been used for training the regression model (Fig. 5.13).

```
1 # Dataset: https://www.kaggle.com/datasets/pravdomirdobrev/texas-
    wind-turbine-dataset-simulated
2 # ---------------Import the necessary libraries------------------
3 import pandas as pd; import numpy as np
4 import plotly.express as px
5 from sklearn.model_selection import train_test_split
```

```python
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

# --------------------Import the dataset-------------------------
df = pd.read_csv("./data/TexasTurbine.csv")
df.set_index("Time stamp", inplace=True)
print(df.head())


# -------------------Define X and y values-----------------------
X = df.drop(columns="System power generated | (kW)")
y = df["System power generated | (kW)"]


# --------------------Split the dataset-------------------------
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.30, random_state=None, shuffle=False)

print("X Train shape:", X_train.shape)
print("X Test shape:", X_test.shape)
print("Y Train shape:", y_train.shape)
print("Y Test shape:", y_test.shape)


# --------------------Create a RFR model-------------------------
RFR = RandomForestRegressor()


# ---------------------Train the model--------------------------
RFR.fit(X_train, y_train)
train_preds = RFR.predict(X_train)
test_preds = RFR.predict(X_test)


# ------Print the model score, train RMSE, and test RMSE---------
print("Model score:", RFR.score(X_train, y_train))
print("Train RMSE:", mean_squared_error(y_train, train_preds)
    **(0.5))
print("Test RMSE:", mean_squared_error(y_test, test_preds)**(0.5)
    )


# ----------Plot the predictions and actual values---------------
plt.figure().set_figwidth(12)
X_test["RFR Prediction"] = test_preds
X_test["System power generated | (kW)"] = y_test
x_index = np.linspace(0, 250, 250)
plt.plot(x_index, X_test["RFR Prediction"].tail(250), color='red'
    , linewidth=1, label='RFR prediction')
plt.plot(x_index, X_test["System power generated | (kW)"].tail
    (250), color='green', linewidth=1,label='Actual power
    generated')
```

```
plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.05),
    fancybox=True, shadow=True, ncol=5)
plt.show()
```

Listing 5.6 Wind turbine power output prediction

Output of Listing 5.6:

```
        System power generated|(kW)      Air temperature|(°C)
Jan 1, 12:00 am             1766.64                    18.263
Jan 1, 01:00 am             1433.83                    18.363
Jan 1, 02:00 am             1167.23                    18.663
Jan 1, 03:00 am             1524.59                    18.763
Jan 1, 04:00 am             1384.28                    18.963

[5 rows x 5 columns]
X Train shape: (6132, 4)
X Test shape: (2628, 4)
Y Train shape: (6132,)
Y Test shape: (2628,)
Model score: 0.9999943031205266
Train RMSE: 2.1351364248882674
Test RMSE: 9.312500866528874
```

5.5 Reactive Power Control and Power Factor Correction

An important application of ML is in power factor correction, which is of great importance in electrical power systems. Maintaining the power factor within a reasonable range is an essential regulatory mechanism in power systems. Typically, power factor correction is employed by controlling the reactive power through capacitor bank, relay, and conductor arrangements or by using phase advancers and synchronous condensers. However, there are some challenges with these traditional approaches, such as slower response time, mechanical faults, harmonics injection in the system, sizing and siting issues, and human errors.

Before going into the details of this section, we will review some basic concepts regarding power and power factor. The power triangle, shown in Fig. 5.14, depicts three quantities—real, reactive, and apparent power.

1. **Real power (P)** is the power that is used up by the loads in the system. It is also called true or active power and the wattful component of electric power. It is measured in Watt (W).
2. **Reactive power (Q)** is the power that moves back and forth within the circuit. It is not used up by the loads. It is the power that can act again and again, hence, the name reactive. It is also called the wattless component of electric power. It is measured in volt-ampere-reactive (VAR).

Table 5.8 Explanation of the power output code example presented in Listing 5.6

Line number	Description
3–7	Importing the necessary libraries
11–13	Read the dataset from file & setting index
17–18	Separate the feature matrix X and the target variable y from the dataframe
22	Split the data into training and testing sets
24–27	Print train & test data shape
31	Create an instance of the RandomForestRegressor as RFR
35	Fit the random forest regressor model to the training data, training the model on the features (X_train) and target variable (y_train)
36–37	Generate predictions using the trained model
41–43	Print the score of the model
46–54	Compare the predicted values and actual values from test set

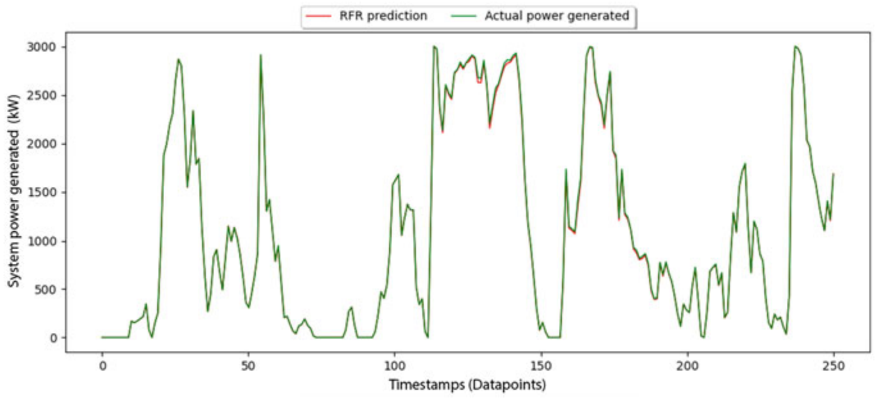
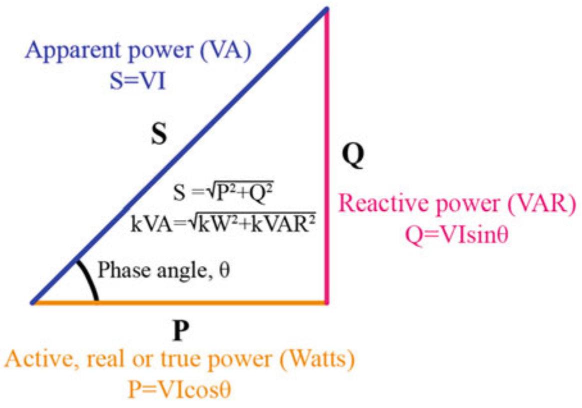


Fig. 5.13 Comparison of the (red) predicted values and the (green) actual values

Fig. 5.14 The power triangle



3. **Apparent power (S)** is the vector sum of the real and reactive power. The apparent power is a vector or complex quantity; the horizontal or real component is the real power, and the vertical or imaginary component is the reactive power. It is measured in volt-ampere (VA).

The **power factor (pf)** is the ratio of real power (P) to apparent power (S). It can also be expressed as the cosine of the phase angle θ between the voltage and the current,

$$pf = \frac{P}{S} = \cos\theta. \tag{5.4}$$

Equation 5.4 shows that the pf is a fraction ranging between 0 and 1. A pf of 1 is ideal, but it can be achieved only in purely resistive circuits. A pf of 0.85 means that of the total apparent power available, 85% is real power, and the remaining 15% is reactive power. The pf can be leading, lagging, or unity depending on the phase angle between the voltage and current. The pf is said to be unity when the voltage and the current are in phase with each other, i.e., when $\theta = 0°$. This is the case in purely resistive circuits. The pf is lagging for inductive loads, where the current lags the voltage, and leading for capacitive loads, where the current leads the voltage. In Fig. 5.15, for both the inductive and capacitive circuits, the phase angle is $\theta = 30°$, which means that the power factor is $\cos\theta = \cos 30° = 0.866$.

A higher value of pf is desired since a low pf can cause a myriad of problems, such as higher internal current, excessive heat, damage to the equipment, reduced output voltage, an overall expensive system, and higher bills. The pf can be kept high if the reactive power is low, but a low reactive power in the system also causes several problems. Reactive power is necessary for maintaining a constant voltage level in the utility grid. With low reactive power, the voltage drops, leading to poor performance of the equipment and a rise in the current. The rise of the current value can incur higher losses and, thus higher costs and create overloads, eventually leading to cascading failures. So, a trade-off between the pf and reactive power is

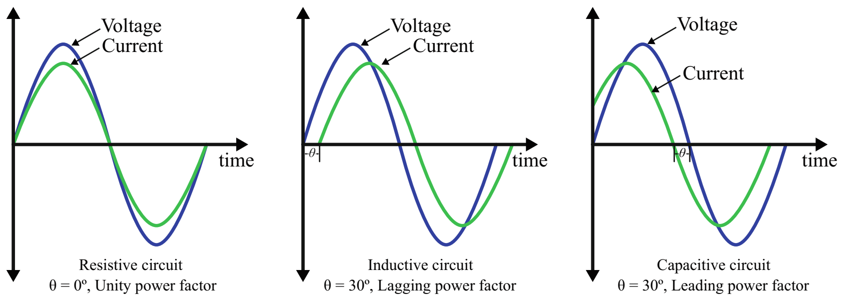


Fig. 5.15 Unity pf is found in a purely resistive circuit, lagging pf in an inductive circuit, and leading pf in a capacitive circuit

essential to ensure the smooth operation of the system. In addition to pf control techniques, a suitable reactive power compensation technique is also adopted in utility systems to maintain the balance between pf and reactive power.

ML can help facilitate many tasks related to reactive power control and power factor correction. For example, in the design phase, it can determine the type, location, and size of the compensator. ML can help with instant sizing, operation technique, switching sequence, and automation in the operation phase. In the maintenance phase, ML can be used for degradation modeling and unit replacement (partial or complete), transient responses, and grid service capability verification.

Most industrial loads are inductive, so capacitor banks inject reactive power into the system to increase the power factor. One use of ML (such as linear regression or ridge regression) is to estimate the size of the capacitor for controlling the pf [13]. This section applies ML to constantly monitor the pf and regulate the capacitor bank to ensure an acceptable range of reactive load.

As shown in Fig. 5.16, the monitoring system takes pf and voltage level as inputs to control the output capacitor bank level. It can operate between three discrete voltage levels, viz. 11, 33, and 69 kV, which are controlled by a switch at the beginning. Based on the voltage level, system frequency, and the pf value, the model outputs the required capacitance value for injecting the necessary reactive power into the system.

Now we want to create a dataset to train a regression model that will take the current power factor as input and return a capacitor bank value (μF) as output. Equation 5.4 can be rewritten as:

$$S = \frac{P}{pf}. \tag{5.5}$$

The reactive power Q can be measured in terms of kVAR, which can be calculated from the following equation:

$$Q = kVAR = \sqrt{S^2 - P^2}. \tag{5.6}$$

For a desired pf, $\hat{pf}$, we can compute the new apparent power as:

$$\hat{S} = \frac{P}{\hat{pf}}. \tag{5.7}$$

The new Q, $\hat{Q}$, or kVAR requirement would be

$$\hat{Q} = kVAR_{new} = \sqrt{\hat{S}^2 - P^2}. \tag{5.8}$$

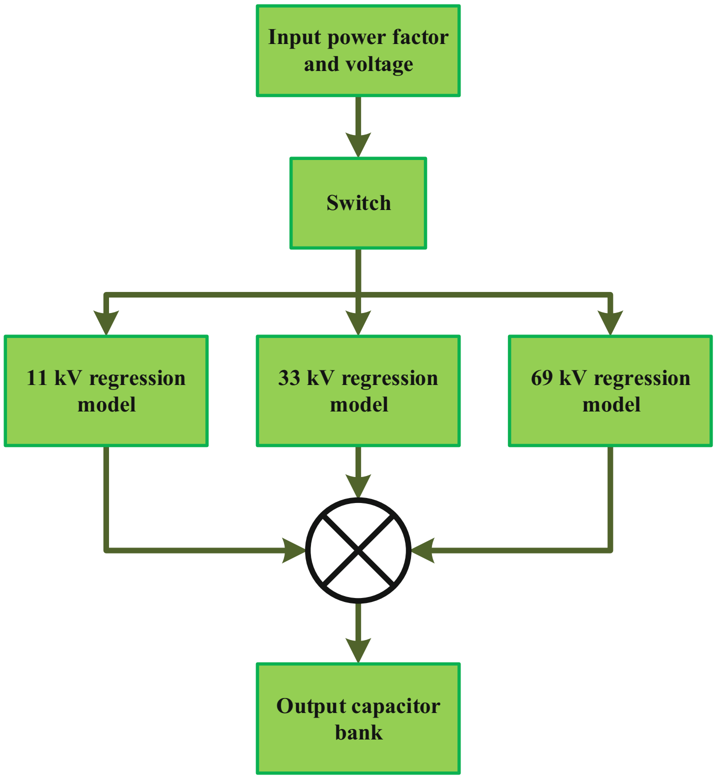


Fig. 5.16 Basic flowchart for the capacitor bank control regression model

So, the additional requirement of kVAR to maintain the same reactive power would be

$$\Delta kVAR = kVAR_{new} - kVAR. \tag{5.9}$$

If ω is the angular frequency, then the capacitor bank requirement in μF would be

$$C = \frac{\Delta kVAR}{\omega \times V^2} \times 10^9. \tag{5.10}$$

Programming Example 5.7
To obtain the train and test dataset, we generate a random pf set for a 24-hour period as the inputs and compute the capacitor bank values as a target value using Equation 5.10. Then, we train a Bayesian Ridge regression model to fit this input pf and target capacitor value. After training, at the inference phase, the regression

model will predict the required capacitor bank value for an input pf to the model. Listing 5.7 shows this model's code, and the code's explanation is given in Table 5.9.

```
# --------------------------Torch Modules--------------------------
from __future__ import print_function
import numpy as np
import torch
from sklearn.linear_model import BayesianRidge
from matplotlib import pyplot as plt


# ----------------------------Variables---------------------------
# parameters
Iterations = 3
frequency = 377 ## rad/s
voltage = torch.Tensor([11,33,69]) ## KV 11kv 33kv , 69kv
real = 2000 ## KW
pf  = 0.95 # power factor
Apower =  real / pf # aparent power
sizes =1024 # 24 hours day data generation
KVARs = np.sqrt(Apower ** 2 - real ** 2) # kvars
print("Target KVARs: ", KVARs)


# ------------------Commands to Prepare Dataset------------------
def create_powerfactor_dataset(sizes,real,KVARs,voltage,frequency
    ):
    ## generating the power factor with respect to time
    thetas = np.arange(sizes) ## creating list of theta
    data_pf = (90 + 10*np.cos(thetas/2) - 0.5 + 0.5* (2*np.random
    .rand(sizes) - 1))/100  # computing power factor dataset

    Apower =  real /data_pf
    new_KVARs = np.sqrt(Apower ** 2 - real ** 2)
    dels  = (new_KVARs - KVARs) ## required Kvar
    voltages = voltage.repeat(sizes)

    for k in range (len(dels)):
        if dels[k] <  0:
            dels[k] = 0
        else:
            dels [k] = dels[k] /(frequency * (voltages[k] ** 2))
    * 1000 # Capacitance in F, not μF

    return torch.Tensor(data_pf).view(sizes,1), torch.Tensor(dels
    ).view(sizes,1)


# -----------------Using BayesianRidge Regression----------------
regressor_11 = BayesianRidge()

```

```
regressor_33= BayesianRidge()

regressor_69 = BayesianRidge()


# ------------------------Plotting Function----------------------
def plot_func(data, y, target, v_level):
    plt.rcParams['font.sans-serif'] = ['Times New Roman']
    plt.rcParams.update({'font.size': 21})
    color1 = 'purple'
    color2 = 'dodgerblue'
    color3 = 'orange'

    fig, ax1 = plt.subplots(figsize=(12, 6))
    line1 = ax1.plot(y[-24:], color=color1, linewidth='1')
    line2 = ax1.plot(target[-24:], color=color3, linewidth='1')
    ax1.set_ylabel('Capacitor Bank values, F')
    ax1.tick_params(axis='y')
    ax2 = ax1.twinx()
    line3 = ax2.plot(data[-24:], color=color2, linewidth='1')
    ax2.set_ylabel('Power Factor')
    ax2.tick_params(axis='y')

    lines = line1 + line2 + line3
    labels = ['Capacitor Bank Predicted', 'Capacitor Bank Ideal',
     'Power Factor for '+str(v_level)+'kV']
    ax2.legend(lines, labels);
    ax1.set_xlabel('Time (Datapoints)')

    ax1.plot(0, 19 if v_level==11 else (2.2 if v_level==33 else
    0.54), marker="^", ms=12, color="k", transform=ax1.
    get_yaxis_transform(), clip_on=False)
    ax2.plot(1, 1.01, marker="^", ms=12, color="k", transform=ax2
    .get_yaxis_transform(), clip_on=False)
    ax1.set_ylim(-0.01, 19 if v_level == 11 else (2.2 if v_level
    == 33 else 0.54))
    ax2.set_ylim(0.79, 1.01)

    ax2.spines['top'].set_visible(False)
    ax1.spines['top'].set_visible(False)
    ax1.spines['bottom'].set_visible(False)
    ax1.spines['left'].set_visible(False)
    ax1.spines['right'].set_visible(False)

    fig.tight_layout()
    plt.savefig('./results/' + str(v_level) + 'control.png', dpi
    =300)
    plt.close()


# ----------------------Training and Testing---------------------
for i in range(Iterations):
    #Training
    ## 11KV case
    if i == 0:
        data_11, target_11 = create_powerfactor_dataset(sizes,
    real,KVARs,voltage[i],frequency)  ## creating the dataset
        regressor_11.fit(data_11[:1000], target_11[:1000].ravel()
    )  ## training
        y_11 = regressor_11.predict(data_11)   ## testing
        y_11[y_11<0] = 0

        plot_func(data_11, y_11, target_11, 11)

    if i == 1: ## 33kv case
        data_33, target_33 = create_powerfactor_dataset(sizes,
    real,KVARs,voltage[i],frequency)  ## creating the dataset

        regressor_33.fit(data_33[:1000], target_33[:1000].ravel()
    )  ## training
        y_33 = regressor_33.predict(data_33)   ## testing
        y_33[y_33<0] = 0

        plot_func(data_33, y_33, target_33, 33)

    if i == 2: # 69 Kv case
        data_69, target_69 = create_powerfactor_dataset(sizes,
    real,KVARs,voltage[i],frequency)  ## creating the dataset

        regressor_69.fit(data_69[:1000], target_69[:1000].ravel()
    )  ## training

        y_69 = regressor_69.predict(data_69)   ## testing
        y_69[y_69<0] = 0

        plot_func(data_69, y_69, target_69, 69)
```

Listing 5.7 Code for controlling the power factor using a capacitor bank

The result of our model is presented in Fig. 5.17. For each voltage level, e.g., 11/33/69 kV, our model predicted capacitor bank values (shown in orange) that match the ideal case (in green). Thus, a simple regression model can monitor the current pf level and regulate the capacitor bank accordingly. To utilize the full advantage of ML, mechanical switching in traditional power systems is a hindrance. So, ML usage in power systems is not yet widespread. However, the newer smart grid and microgrid systems utilize solid state or electronic switching, so the application of ML is increasing gradually.

5.6 Conclusion

Machine learning has become quite prevalent in energy systems, particularly electrical power systems. This chapter presents some of these applications, including electrical load forecasting, electrical fault detection and classification, partial dis-

Table 5.9 Explanation of the power factor correction code in Listing 5.7

Line number	Description
2–6	PyTorch Modules
9–19	Hyperparameters and computing the target kVAR
22–39	Creating dataset with power factor and corresponding require kVARs
37	The equation here differs from Eq. 5.10 as it calculates output in F (not μF)
43–47	Three Bayesian ridge regression model for each voltage
50–86	Function for plotting figure with pf and capacitor bank values
90	Initiating the training and testing loops for three different voltages
94	Dataset creation for 11 kV case
95–97	Training and testing using Bayesian Ridge regression
99	Plotting comparison of predicted and idea capacitor bank
101–108	Training, testing, and comparing results with idea data for 33 kV case
110–118	Repeating the process for 69 kV case

charge detection, future trend prediction of solar PV installed capacity, wind power forecasting, reactive power control, and power factor correction. The programming examples are presented in each section with detailed explanations and relevant output graphs. The clear narratives will help the reader to replicate the works easily and customize the programming examples as required. The knowledge gained from this chapter will prepare the readers to conduct further studies on advanced topics associated with the concepts of this chapter and help them initiate their own works in this particular research domain. In the next chapter, we will study machine learning applications in robotics.

5.7 Key Messages from this Chapter

- Modern energy systems are abundant in useful data, and proper utilization of these data can help ensure fast, reliable, and efficient system operation. A machine learning-based approach can play a significant role in this area.
- Machine learning models can provide energy systems with improved load forecasting capabilities to ensure proper operations, planning, and management strategy.
- Machine learning-based electrical fault detection and classification techniques improve energy security by providing fast and reliable system insights.
- Efficient planning of renewable energy facilities can be done through machine learning.
- Machine learning can increase energy system efficiency and stability by providing highly responsive, fast, and accurate power factor correction and control schemes.

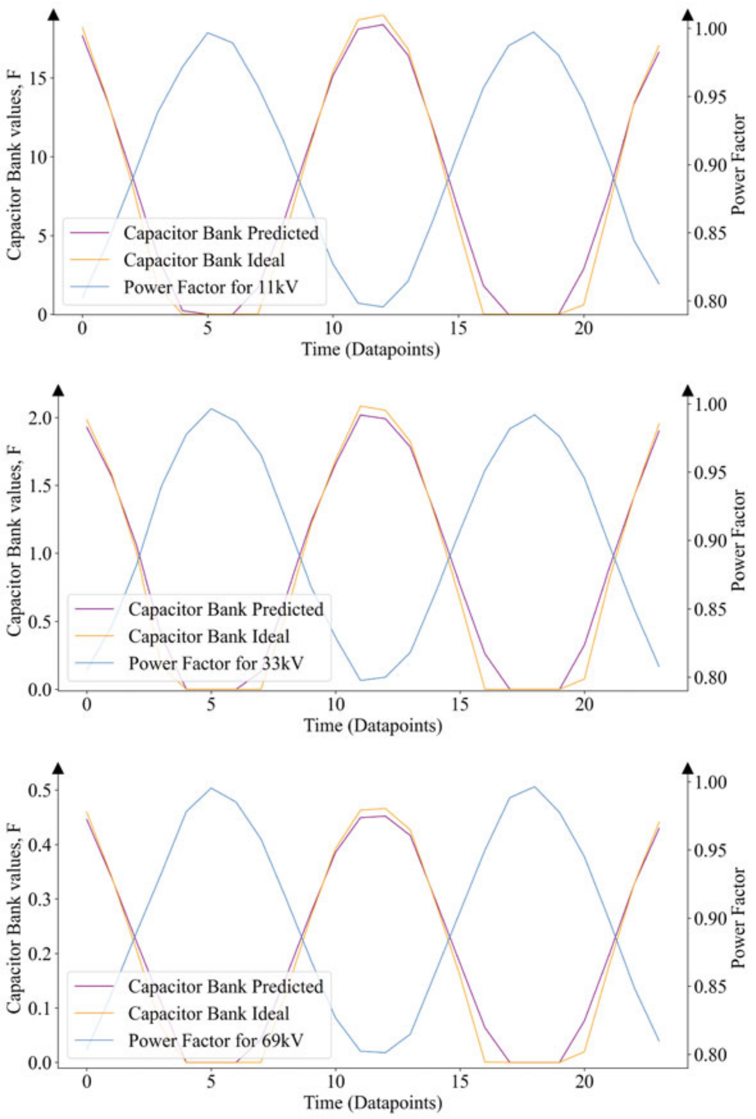


Fig. 5.17 Controlling the kVARs using capacitor bank through regression model for each voltage

5.8 Exercise

1. Define the following:
   (a) load forecasting
   (b) electrical fault
   (c) power factor
   (d) power triangle
2. Discuss how machine learning techniques can be utilized in electrical power systems.
3. What are the key advantages of load forecasting? How can ML contribute to this process?
4. Develop a load forecasting model on the Panama Case Study dataset [14] (Link: https://data.mendeley.com/datasets/byx7sztj59/1).
5. What are the consequences of having low power factor? How can ML contribute to solving this problem?
6. Develop a machine learning-based power factor correction model for a 20,000 kW, 132 kV line. Generate 1024–2048 random power factor values to use as the dataset.
7. Apply simple LSTM model to detect transmission line fault as done in Sect. 5.3.3 and compare results with the given one. List the benefits and drawbacks of utilizing LSTM over the given data format. [Hint: For fault analysis, we used time series data.]
8. Besides the topics discussed in this chapter, in what other applications can machine learning be used in energy systems? Give examples.

References

1. Rafi, S. H., Deeba, S. R., Hossain, E., et al. (2021). A short-term load forecasting method using integrated CNN and LSTM network. *IEEE Access, 9*, 32436–32448.
2. Al Mamun, A., Sohel, M., Mohammad, N., Sunny, M. S. H., Dipta, D. R., & Hossain, E. (2020). A comprehensive review of the load forecasting techniques using single and hybrid predictive models. *IEEE Access, 8*, 134911–134939.
3. Mulla, R. *PJM hourly energy consumption data*. https://www.kaggle.com/robikscube/hourly-energy-consumption
4. Tan, R. (2017). Instantaneous overcurrent relay, Mar 2017.
5. Prakash, E. S. *Electrical fault detection and classification*. https://www.kaggle.com/esathyaprakash/electrical-fault-detection-and-classification/
6. Jamil, M., Sharma, S. K., & Singh, R. (2015). Fault detection and classification in electrical power transmission system using artificial neural network. *SpringerPlus, 4*(1), 334.
7. VSB power line fault detection. https://www.kaggle.com/competitions/vsb-power-line-fault-detection/data
8. International Energy Agency. *Solar PV*. https://www.iea.org/energy-system/renewables/solar-pv
9. Tabassum, S., Rahman, T., Islam, A. U., Rahman, S., Dipta, D. R., Roy, S., Mohammad, N., Nawar, N., & Hossain, E. (2021). Solar energy in the United States: Development, challenges and future prospects. *Energies, 14*(23), 8142.

10. International Renewable Energy Agency. *Wind energy*. https://www.irena.org/Energy-Transition/Technology/Wind-energy
11. International Energy Agency. *Wind*. https://www.iea.org/energy-system/renewables/wind
12. Texas wind turbine dataset—Simulated. https://www.kaggle.com/datasets/pravdomirdobrev/texas-wind-turbine-dataset-simulated
13. Bayindir, R., Gok, M., Kabalci, E., & Kaplan, O. (2011). An intelligent power factor correction approach based on linear regression and ridge regression methods. In *2011 10th International Conference on Machine Learning and Applications and Workshops* (Vol. 2, pp. 313–315). IEEE.
14. Madrid, E. A. (2021). *Short-term electricity load forecasting (Panama case study)*. https://data.mendeley.com/datasets/byx7sztj59/1

Applications of Machine Learning: Robotics

6

6.1 Introduction

A *robot* is a programmable machine that can perform certain routine tasks via external or internal control. The primary purpose of a robot is to reduce human workload and make our work more effective in terms of money and effort. A robot may or may not resemble a human appearance. *Robotics* deals with the theoretical and engineering aspects of robots. Robot concepts, design, development, manufacture, operation, and control are all fundamental parts of robotics. The study of robotics and the investigation of robot capabilities and possible applications has expanded significantly in the twenty-first century. Industrial robots and various other robots are utilized nowadays to carry out monotonous tasks. They could appear as standard humanoid robots, robotic exoskeletons, or robotic arms. A robot or robotic system's actions are guided by a combination of computer programming and algorithms, a remotely controlled manipulator, actuators, control systems—action, processing, perception—real-time sensors, and an element of automation.

Traditionally, robots are programmed to perform tasks in a well-controlled environment, which renders them unable to perform various tasks in a changing environment. This problem can be resolved if robots can make their decisions optimally on their own through their perception of the outside environment. Different machine learning and artificial intelligence-based approaches can be taken to give robots their perception and decision-making power. This integration of machine learning and robotics is referred to as *robot learning*. Robot learning has been used for various tasks, such as handling items, categorizing objects, and even interacting linguistically with a human peer. Learning can occur through self-discovery or with the help of a human operator. Intelligent robots must gather knowledge from human input or sensors in order to learn. The processing unit of the robot will then compare the recently acquired data to the information that has already been recorded and decide the best course of action based on the data it has acquired.

This chapter talks about the application of machine learning in the field of robotics. A robot must be able to see or track objects, recognize them, and follow pre-defined instructions. All these aspects are covered in this chapter. This chapter first discusses computer vision, including object tracking, recognition, and image segmentation. At the end of this chapter is an example of a line follower robot that can follow a given non-linear path defined by an equation.

6.2 Computer Vision and Machine Vision

Vision is our ability to see and comprehend things. The same goes for computers and machines. The term *computer vision* refers to an automated image capture and analysis system governed by a computer. Computer vision mainly focuses on extracting as much information as possible from captured images. It is a subset of machine learning. Computer vision and machine learning aim to teach computers how to analyze and respond to data in a particular circumstance. While machine learning concentrates on other forms of data and seeks to address picture classification, object recognition, object segmentation, and object tracking in movies, computer vision is far more focused on imaging and visual data. Machine learning is a method that gives computers the ability to learn how to analyze and respond to data inputs based on precedents established by prior actions. With the integration of machine learning with computer vision technologies, it is possible to teach computers to see patterns in visual data, similar to how humans do.

When computer vision is applied to interacting with the physical world by means of some devices or machines, e.g., self-driving cars, autonomous drones, automated product inspection systems, etc., then it is called *machine vision*. Machine vision also deals with the post-image processing part, i.e., making decisions based on image analysis information. It enables machines to comprehend the visual environment to precisely recognize, categorize, and respond to objects. When performing computer vision tasks, the availability of computing resources is of little concern. As computer vision tasks are generally done using standard computing platforms, the computing resources are usually expandable and can be allocated as per requirement. In the case of machine vision, the executing devices usually have less computing power and are not expandable. So, hardware-related constraints play an important part in performing machine vision tasks.

In this section, we will discuss some machine vision-related tasks in robotics—object tracking, object recognition/detection, and image segmentation.

6.2.1 Object Tracking

One of the critical applications of robotics is object tracking. Any vision task, either a robot or a self-driving car, must first identify the object's location in front of them. Once the location is spotted accurately, the algorithm can then classify the located object using the image classification algorithms discussed in Chaps. 3 and 4.

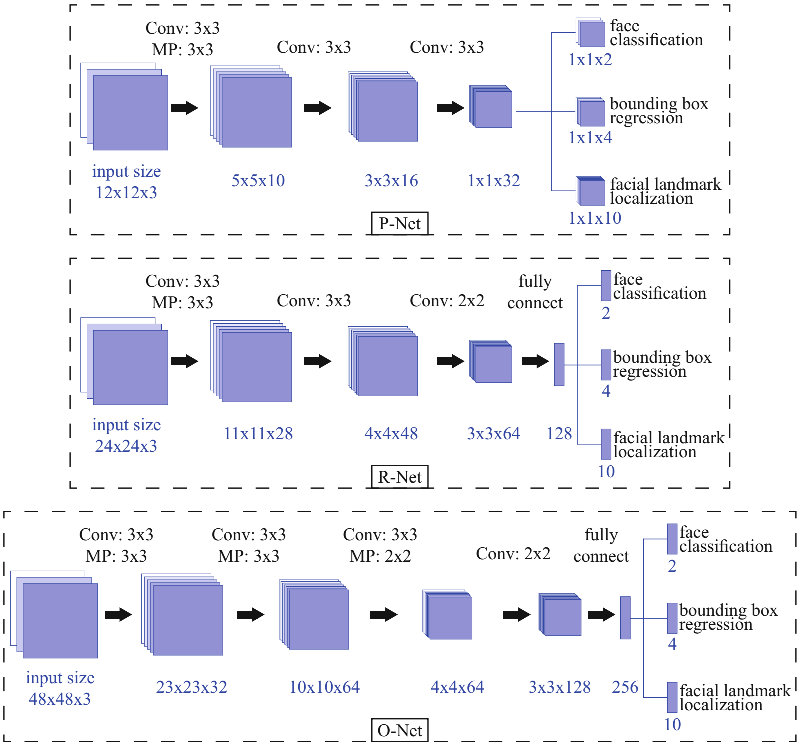


Fig. 6.1 The architecture of the MTCNN model [2]. It consists of three CNN architectures: P-Net, R-Net, and O-Net

This section will look at a famous face tracking PyTorch module known as the *FaceNet-PyTorch* [1]. For this purpose, first, we will study the MultiTask Cascaded Convolutional Neural Network (MTCNN) architecture for face detection and then study a face tracking example using the MTCNN.

6.2.1.1 The MTCNN Architecture

The MTCNN architecture [2] uses three cascaded CNN networks for face detection. This three-stage architecture of the MTCNN is illustrated in Fig. 6.1.

As shown in Fig. 6.2, the face tracking algorithm in MTCNN operates at three specific stages—the P-Net, the R-Net, and the O-Net. These three stages are described below:

1. **P-Net:** To obtain the candidate (target) facial windows and their bounding box regression vectors, the *P-Net or propose network* is used. The candidates are then calibrated using the bounding box regression vectors that have been estimated. After that, non-maximum suppression (NMS) is used to merge candidates with

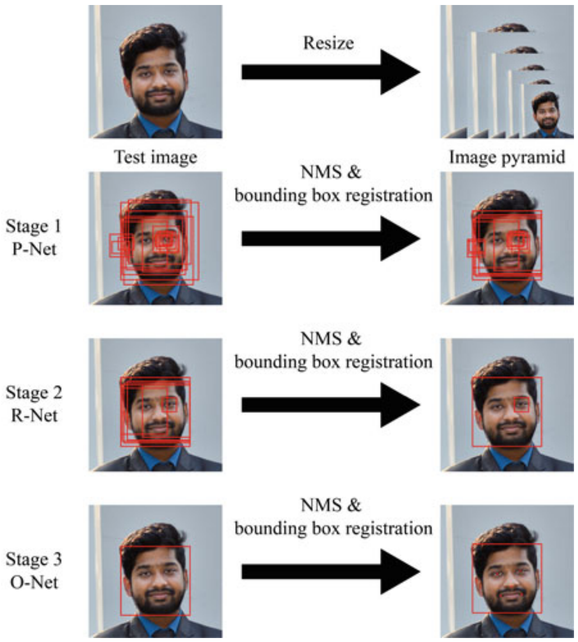


Fig. 6.2 Three working stages of MTCNN model using P-Net, R-Net, and O-Net [2]. Image courtesy: Ashraf Ul Islam Shihab

high overlap. NMS is a technique to discard candidates that fall below a probability bound.

2. **R-Net:** All candidates are fed into another CNN called the *refined network (R-Net)*, which rejects a large number of false candidates, calibrates using bounding box regression, and performs NMS.
3. **O-Net:** The output network (O-Net) is similar to the R-Net, but face regions are identified with more supervision in this stage. The network will output the positions of five facial landmarks in particular.

The main challenge here is to work with videos, which means tracking faces in consecutive frames one after the other rather than from single images. Another crucial challenge is detecting and tracking faces in real-time; otherwise, the main purpose of face tracking will fail.

6.2.1.2 Face Tracking Example Using MTCNN

Programming Example 6.1

We will demonstrate a simple face tracking example using the MTCNN module of FaceNet in Listing 6.1. The explanation of the code can be found in Table 6.1. The code loads a video using the pre-defined `mmcv.VideoReader` class. Each frame of the image is converted to an object and passed through the MTCNN to detect faces. The tracked faces are compiled together to make a video at the end.

```python
# ----------------------------Modules----------------------------
from facenet_pytorch import MTCNN
import torch
import numpy as np
import mmcv, cv2
from PIL import Image, ImageDraw


# --------------------------MTCNN Model--------------------------
mtcnn = MTCNN(keep_all=True, device='cpu')


# --------------Loading a Video for Face Recognition-------------
video = mmcv.VideoReader('./data/video.mp4')
frames = [Image.fromarray(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)) for frame in video]


## we will track each frame using
frames_tracked = []
for i, frame in enumerate(frames):
    print('\rTracking frame: {}'.format(i + 1), end='')

    # Detect faces within each boxes
    boxes, _ = mtcnn.detect(frame)

    # Draw a rectagle around the faces
    frame_draw = frame.copy()
    draw = ImageDraw.Draw(frame_draw)
    for box in boxes:
        draw.rectangle(box.tolist(), outline=(255, 0, 0), width=6)

    # Add to frame list
    frames_tracked.append(frame_draw.resize((640, 360), Image.BILINEAR))
print('\nDone')


# ---------------------Saving Tracked Video----------------------
dim = frames_tracked[0].size
fourcc = cv2.VideoWriter_fourcc(*'FMP4')
video_tracked = cv2.VideoWriter('./results/video_tracked.mp4', fourcc, 25.0, dim)
for frame in frames_tracked:
    video_tracked.write(cv2.cvtColor(np.array(frame), cv2.COLOR_RGB2BGR))
video_tracked.release()
```

Listing 6.1 Face tracking in a video frame

In Fig. 6.3, we demonstrate the results of Listing 6.1 in tracking a video frame with multiple faces. A pre-trained MTCNN model from FaceNet PyTorch can successfully detect the location of the faces. Usually, the object tracking datasets

Table 6.1 Explanation of the face tracking example in Listing 6.1

Line number	Description
2–6	PyTorch FaceNet and OpenCV to track faces in a video
10	Defining the MTCNN model
14–15	Loading a sample video
19–24	Detecting each frame face within a box
26–34	Drawing the rectangle
37–43	Saving the tracked video

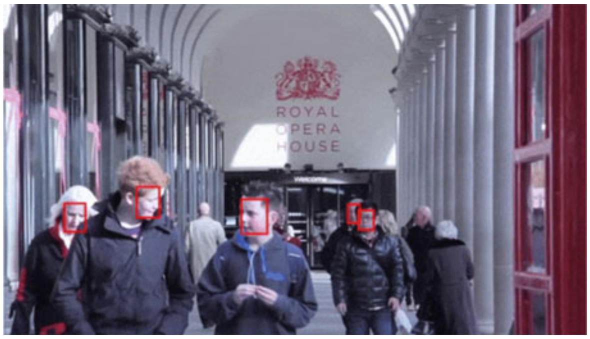


Fig. 6.3 A demonstration of face detection in a video frame. Video snippet courtesy: GitHub source [3]

are larger, and architectures are more complex than most domains we discussed in this book. Hence, we have presented a simple example of tracking faces in a demo video of FaceNet PyTorch [4]. For more analysis and application of object tracking, please check famous CNN architectures such as YOLO-V2 [5].

6.2.2 Object Recognition/Detection

Once the algorithm performs the localization of an object inside a single video frame, the next task is to identify the object. An object detection model should also be able to detect all the different objects within a frame. Modern architectures such as YOLO can simultaneously perform object tracking and detection tasks. However, discussing the details of object tracking models and their datasets is beyond the scope of this book. In the following sections, we focus on some practical object detection problems.

6.2.2.1 Applications of Object Recognition/Detection

Some object recognition or detection applications are discussed briefly to give the readers a picture of this broad spectrum of usage.

1. **Retail Stores:** To track customer footfall and acquire data on how customers spend their time, many retail shops strategically put people counting equipment. AI-based customer analysis that uses cameras to recognize and track customers can be used to better understand consumer interaction and experience, improve store layout, and streamline operations. Identifying queues to shorten wait times in retail establishments is a common use case.
2. **Agriculture:** Object detection can be used in several agricultural activities, including product quality assessment, animal monitoring, and counting. Machine learning algorithms can be used to identify damaged food even as it is being processed.
3. **Security Applications:** Object detection is the foundation for various security applications in video surveillance, such as using computer vision to automate inspection chores at remote locations or detect persons in dangerous or prohibited areas.
4. **Self-driving Cars:** Using object detection, self-driving cars may identify pedestrians, traffic signs, other vehicles, and more. For instance, Tesla's Autopilot AI mainly relies on object detection to identify potential risks from the environment and its surroundings, such as approaching vehicles or obstructions. An example is shown in the next section to demonstrate a solution to a real-life self-driving problem.
5. **Healthcare:** Numerous advances in the field of medicine have been made possible through object detection. Applying object detection and image segmentation on CT and MRI scans can provide fast and accurate medical diagnostics.
6. **Traffic Analysis:** For traffic analysis, object recognition is used to count and identify vehicles. It also identifies vehicles that stop in potentially hazardous locations, such as on highways or crosswalks.

6.2.2.2 Self-Driving Car: Traffic Sign

An important application for a self-driving autonomous car is identifying street traffic signs. To demonstrate the traffic sign recognition task, we use the German Traffic Sign Recognition Benchmark dataset [6]. As shown in Fig. 6.4, it has 42 different traffic signs with 40,000 labeled train images. Among them, we have used 80% of the labeled images as the training dataset and the remaining 20% as the test dataset for the recognition task.

Programming Example 6.2

For enabling the self-driving car to recognize traffic signs, we will use a ResNet-18 model [7], which has 18 layers with only one fully connected layer and several residual connections between convolution layers. Residual models such as ResNet-18 perform exceptionally well in classifying large datasets. Listing 6.2 demonstrates the Python program, and Table 6.2 explains the code.

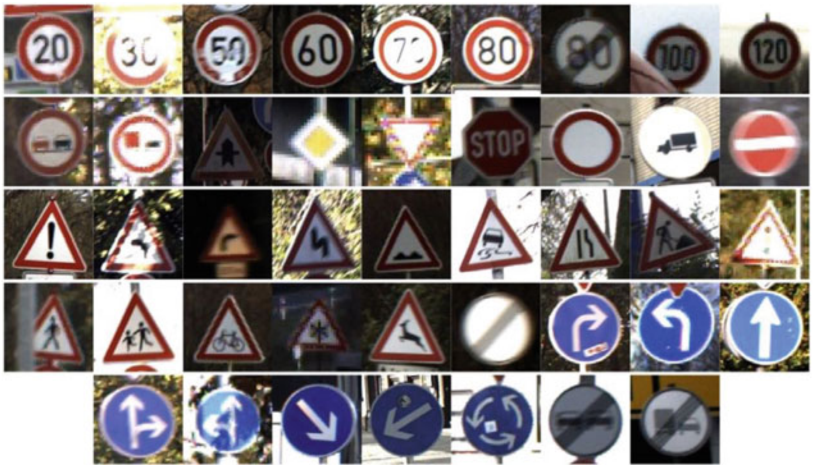


Fig. 6.4 The German traffic sign recognition benchmark [6]

```python
# -------------------------Torch Modules-------------------------
import numpy as np
import pandas as pd
import torch.nn as nn
import math
import torch.nn.functional as F
import torch
from torch.nn import init
import torch.optim as optim
from torchvision.datasets import ImageFolder as IF
from torchvision import models
import torch.nn.functional as F
import torchvision
from torchvision import transforms as Trans
from torch.utils.data import DataLoader as DL
from torch.utils.data import random_split
import time
import numpy as np
import os
import matplotlib.pyplot as plt
from torch.optim import SGD
# ---------------------------Variables---------------------------
bs = 128 #Batch Size
learning_rate = 0.005
Iterations = 5
CUDA_av = 1 # set to 1 for GPU training


# ----------------Prepare the German Sign Dataset----------------
# Define the transformations.
# To begin with, we shall keep it minimum
# Only resizing the images and converting them to PyTorch tensors
```

```python

data_transforms = Trans.Compose([
    Trans.Resize([112, 112]),
    Trans.ToTensor(),
    ])


# Create data loader for training and validation

train_directory = "../input/gtsrb-german-traffic-sign/Train"
train_dataset = IF(root = train_data_path, transform =
    data_transforms)

# Divide data into training and validation (0.8 and 0.2)
ratio = 0.8
n_train_examples = int(len(train_dataset) * ratio)
n_val_examples = len(train_dataset) - n_train_examples

train_dataset, validation_data = random_split(train_dataset, [
    n_train_examples, n_val_examples])

print(f"Training dataset samples: {len(train_dataset)}")
print(f"Validation dataset samples: {len(validation_data)}")

train_dataloader = DL(train_dataset, shuffle=True, batch_size =
    bs)
test_dataloader = DL(validation_data, shuffle=True, batch_size =
    bs)


# -------------------Defining a ResNet-18 Model------------------
class ModifiedBlock(nn.Module):
    expansion_factor = 1
    def __init__(self, input_channels, output_channels, stride=1)
    :
        super(ModifiedBlock, self).__init__()
        self.conv1 = nn.Conv2d(
            input_channels, output_channels, kernel_size=3,
    stride=stride, padding=1, bias=False)
        self.batch_norm1 = nn.BatchNorm2d(output_channels)
        self.conv2 = nn.Conv2d(output_channels, output_channels,
    kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.batch_norm2 = nn.BatchNorm2d(output_channels)

        self.shortcut_connection = nn.Sequential()
        if stride != 1 or input_channels != self.expansion_factor
    *output_channels:
            self.shortcut_connection = nn.Sequential(
                nn.Conv2d(input_channels, self.expansion_factor*
    output_channels,
                          kernel_size=1, stride=stride, bias=
    False),
                nn.BatchNorm2d(self.expansion_factor*
    output_channels)
            )

    def forward(self, x):
        out = F.relu(self.batch_norm1(self.conv1(x)))
        out = self.batch_norm2(self.conv2(out))
        out += self.shortcut_connection(x)
        out = F.relu(out)
        return out


class ModifiedResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=43):
        super(ModifiedResNet, self).__init__()
        self.input_channels = 64

        self.conv1 = nn.Conv2d(3, 64, kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.batch_norm1 = nn.BatchNorm2d(64)
        self.layer1 = self._make_layer(block, 64, num_blocks[0],
    stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1],
     stride=2)
        self.layer3 = self._make_layer(block, 256, num_blocks[2],
     stride=2)
        self.layer4 = self._make_layer(block, 512, num_blocks[3],
     stride=2)
        self.linear = nn.Linear(4608, num_classes)

    def _make_layer(self, block, output_channels, num_blocks,
    stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for stride in strides:
            layers.append(block(self.input_channels,
    output_channels, stride))
            self.input_channels = output_channels * block.
    expansion_factor
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.batch_norm1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, 4)
        out = out.view(out.size(0), -1)

        out = self.linear(out)
        return out

```

```python

def ModifiedResNet18():
    return ModifiedResNet(ModifiedBlock, [2, 2, 2, 2])


# defining CNN model
CNN_Model = ModifiedResNet18()
if CUDA_av == 1:
    CNN_Model = CNN_Model.cuda()
## Loss function
loss_criterion = torch.nn.CrossEntropyLoss() # pytorch's cross entropy loss function
if CUDA_av == 1:
    loss_criterion = loss_criterion.cuda()
# definin which paramters to train only the CNN model parameters
optimizer = SGD(CNN_Model.parameters(),learning_rate)

# defining the training function
# Train baseline classifier on clean data
def train_model(CNN_Model, optimizer,loss_criterion,epoch_no):
    CNN_Model.train() # setting up for training
    for id, (input_images, labels) in enumerate(train_dataloader): # data contains the image and target contains the label = 0/1/2/3/4/5/6/7/8/9
        if CUDA_av == 1:
            input_images, labels = input_images.cuda(), labels.cuda()
        optimizer.zero_grad() # setting gradient to zero
        output = CNN_Model(input_images) # forward
        loss = loss_criterion(output, labels) # loss computation
        loss.backward() # back propagation here pytorch will take care of it
        optimizer.step() # updating the weight values
        if id % 100 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch_no, id * len(input_images), len(train_dataloader.dataset),
                100. * id / len(train_dataloader), loss.item()))


# to evaluate the model
## validation of test accuracy
def test_model(CNN_Model, loss_criterion, val_loader, epoch_no):
    CNN_Model.eval()
    test_loss = 0
    correct_flag = 0

    with torch.no_grad():
        for id, (input_images, labels) in enumerate(val_loader):
            if CUDA_av == 1:
                input_images, labels = input_images.cuda() , labels.cuda()
```

```
166             output = CNN_Model(input_images)
167             test_loss += loss_criterion(output, labels).item()
168             pred = output.max(1, keepdim=True)[1]
169             correct_flag += pred.eq(labels.view_as(pred)).sum().item() # if pred == labels then correct_flag +=1
170 
171     test_loss /= len(val_loader.dataset)
172     print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.4f}%)\n'.format(
173         test_loss, correct_flag, val_loader.sampler.__len__(),
174         100. * correct_flag / val_loader.sampler.__len__() ))
175 
176 
177 ## training the CNN
178 for i in range(Iterations):
179     train_model(CNN_Model, optimizer,loss_criterion,i)
180     test_model(CNN_Model, loss_criterion, test_dataloader, i) # Testing the the current CNN
```

Listing 6.2 Traffic sign detection code [8]

Table 6.2 Explanation of the traffic sign detection example in Listing 6.2

Line number	Description
3–22	Import the required modules
25–28	Hyper-parameters; if you have GPU set CUDA_av = 1
30–56	Preparing the dataset
59–123	Defining the classifier model ResNet-18
126–135	Training setup
139–180	Same train and test functions used prior

Output of Listing 6.2:

```
Training dataset samples: 31367
Validation dataset samples: 7842
Train Epoch: 0 [0/31367 (0%)] Loss: 3.835121
Train Epoch: 0 [12800/31367 (41%)] Loss: 1.762058
Train Epoch: 0 [25600/31367 (81%)] Loss: 1.356717

Test set: Average loss: 0.0231, Accuracy: 2667/7842 (34.0092%)

Train Epoch: 1 [0/31367 (0%)] Loss: 2.470074
Train Epoch: 1 [12800/31367 (41%)] Loss: 0.797109
Train Epoch: 1 [25600/31367 (81%)] Loss: 0.488599

Test set: Average loss: 0.0094, Accuracy: 4972/7842 (63.4022%)

Train Epoch: 2 [0/31367 (0%)] Loss: 0.802156
Train Epoch: 2 [12800/31367 (41%)] Loss: 0.377860
Train Epoch: 2 [25600/31367 (81%)] Loss: 0.274211

Test set: Average loss: 0.0177, Accuracy: 3503/7842 (44.6697%)

Train Epoch: 3 [0/31367 (0%)] Loss: 1.994194
Train Epoch: 3 [12800/31367 (41%)] Loss: 0.163087
Train Epoch: 3 [25600/31367 (81%)] Loss: 0.170656

Test set: Average loss: 0.0071, Accuracy: 5945/7842 (75.8097%)

Train Epoch: 4 [0/31367 (0%)] Loss: 0.313496
Train Epoch: 4 [12800/31367 (41%)] Loss: 0.132293
Train Epoch: 4 [25600/31367 (81%)] Loss: 0.105552

Test set: Average loss: 0.0037, Accuracy: 6767/7842 (86.2918%)
```

The output of the above code shows that the ResNet-18 model can recognize 6,767 street signs among the total 7,842 test images with a success rate of 86.29%. We would encourage the reader to apply other advanced models, such as Mobile-Net-V2, and try to improve the accuracy even further.

Recent advances in computer vision have made it easy to train a convolution or fully connected neural network and perform classification tasks with high accuracy. We already presented the classification problem of hand-written digits of MNIST in Chap. 3. Also, we have demonstrated the classification problem of RGB images in Chap. 4 (CIFAR-10 dataset). In summary, we have covered a wide range of vision (e.g., grey-scale or RGB) classification examples throughout this book. Based on these samples, we can detect any other objects from our daily life.

6.2.3 Image Segmentation

A major objective of computer vision and machine vision is extracting information from target images. One way of extracting information from an image could be by isolating and marking different objects within an image, which can be done through *image segmentation*. Image segmentation is the process of identifying each pixel within an image and creating segments by isolating different objects from each other so that the identity and location of each object within an image can be easily determined.

Image segmentation can be classified into three main categories: semantic, instance, and panoptic.

1. **Semantic Segmentation:** In semantic segmentation, each object within an image is separated from the other, but the objects from the same class are given the same label. For example, if an image contains four different bottles, all the bottles will be labeled similarly. The main goal of semantic segmentation is to classify objects within an image at the pixel level. Semantic segmentation is used when counting the number of objects in each class is unnecessary. Its output contains the original

image background with separate masks to highlight different objects belonging to different classes.

2. **Instance Segmentation:** In instance segmentation, each object within an image is separated from the other and is labeled separately, even if those objects are of the same class. For example, if an image contains four different bottles, they will be separately labeled even though they are in the same class. The main goal of instance segmentation is to detect the addition or removal of objects belonging to any particular class to an image. It is especially useful where multiple instances of the same type of objects are present. The output of instance segmentation usually contains a black background with varying colored masks to highlight separate instances of a particular class.
3. **Panoptic Segmentation:** The term *pan* means *whole*, and *optic* relates to *vision*. Therefore, panoptic segmentation refers to the image segmentation process dealing with all the vision segmentation aspects within an image. This is achieved by combining instance segmentation and semantic segmentation. In panoptic segmentation, each object within an image is given two labels—one is the *class label*, and another is the *instance label* to which it belongs. Panoptic segmentation considers both countable and uncountable objects so that the image can be analyzed and segmented thoroughly.

Figure 6.5 shows different types of image segmentation. This chapter will look at an example of an image segmentation application in autonomous drone flight to ensure flight and landing safety. This is a semantic segmentation application, and we will use the U-Net architecture to implement our image segmentation task.

6.2.3.1 The U-Net Architecture

The U-Net architecture is a modified and evolved form of traditional convolutional neural networks developed in 2015 by Ronneberger et al. [10] to perform biomedical image segmentation. As shown in Fig. 6.6, the U-Net has a symmetric U-shaped architecture. It consists of two major parts: the contracting and expansive paths.

1. **Contracting Path:** The left portion of the U-shape of the network is known as the contracting path. It consists of traditional convolution layers. This path

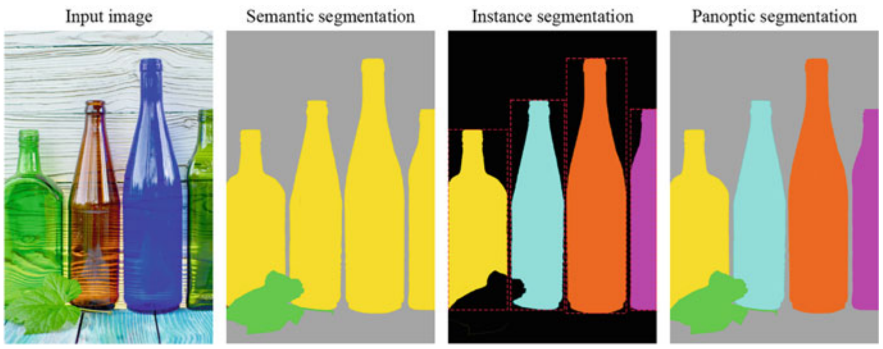


Fig. 6.5 Image segmentation categories. Photo by Andrey Haimin on Unsplash [9]

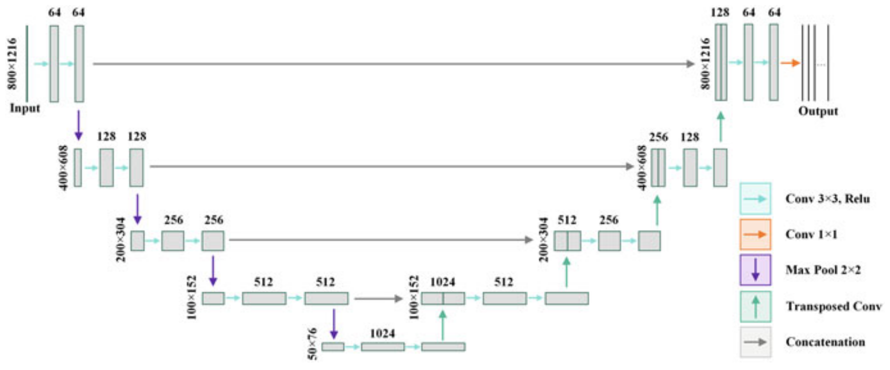


Fig. 6.6 U-Net architecture

follows the top-down approach. Each step consists of two convolution layers followed by a max pooling layer, which proceeds to the next step. In the first step, the resolution of the input image is unchanged, but the number of channels is increased to 64. After a 2×2 max pooling operation, the resolution of the input image becomes halved, and it proceeds to the second step, where the number of channels is further increased to 128 through two consecutive convolution layers. This operation is repeated several times until the number of channels becomes 1024. In this part of the network, the resolution of the input image gets halved at each step; that is why it is called the contracting path. In the lowest part of the network, there is no max pooling operation.

2. **Expansive Path:** The right portion of the U-shape of the network is known as the expansive path. This path follows the bottom-up approach. Each step consists of two convolution layers followed by a transposed convolution layer. A transposed convolution layer is an up-sampling technique that expands the size of its input by adding some padding to the input. Here, the output resolution of the transposed convolution operation is the same as the output resolution of the corresponding contracting path step. Some information gets altered or missing during the up-sampling process, so the up-sampled image is concatenated with the corresponding contracting path image to retain information for more precise segmentation output. This operation is repeated several times until the image's resolution becomes the same as the input. In the uppermost step, the last layer is a convolution layer of shape 1×1 to map the segmented output. The number of channels gets doubled at each step in this path; that is why it is called the expansive path.

There is no dense layer in U-Net architecture. The operating process of this network consisting of contracting and expansive path is analogous to the encoder-decoder operation. The input and output resolution of the U-Net is the same. This architecture is able to provide good segmentation performance even if the input dataset size is small.

6.2.3.2 Aerial Semantic Segmentation Example

An application of image segmentation in robotics is the aerial semantic segmentation performed by unmanned aerial vehicles (UAVs).

Programming Example 6.3

To demonstrate an example, we will use the semantic drone dataset [11], which is also available in Kaggle. We first load the dataset and apply several data augmentation techniques to improve model performance. We then use the U-Net architecture using PyTorch to perform semantic aerial segmentation. Table 6.3

Table 6.3 Explanation of the state detection code example presented in Listing 6.3

Line number	Description
1–18	Importing necessary library files
18–22	Reading the images and masks
24–28	Iterate each file in the directories to append them in an object
31–32	Split train, test, and validation dataset
37–49	Defines a train dataset class and initializes the instance variables
50–57	Reads an image, converts the color space, and applies the transformation
66–67	If patches are set to True, split the image and mask into patches
71–79	Defines the tiles method to split the image and mask into patches
84–93	Defines transformations for training data and for validation data
98–105	Create a dataset and data loader for training and validation loop
110–115	Creates an instance of the U-Net model where 'mobilenet_v2' with pre-trained weights from 'imagenet'
120–125	Defines a function that calculates the accuracy given the output predictions
127–129	Defines a function that returns the learning rate of the optimizer
131–140	Defines a function that trains the model for the specified number of epochs
142–143	Iterates over the specified number of epochs for training
147–148	Iterates over the training data batches
149–152	If patch = True; retrieves and reshapes the tensor
154–186	Computes the loss function and accuracy
188–194	Checks if current val_loss is lower than previous; saves least one
196–202	If it is higher, not_improve counter increases
204–205	Appends the average accuracy of the epoch
218–220	Sets the maximum learning rate, total number of epochs, and weight decay
222–223	CrossEntropyLoss criterion and AdamW optimizer used
224–227	Creates a learning rate scheduler that adjusts the learning rate
232–243	Plot accuracy and loss function improvement with each epoch
247–273	Define a test dataset properties as before
276–277	Transformation and dataset is created
281–305	Defines functions to calculate the accuracy
313–315	Create a path for saving figure, if it does not exist
317–333	Plot random images to cross-validate results

explains the code of Listing 6.3. The model performance is shown in Fig. 6.7 while Fig. 6.8 shows the final output. The listing is written for the IPython or Jupyter Notebook environment. At the start of the notebook, there is a shell command that starts with "!". The notebook environment can easily handle that. However, if a Python environment is preferred, comment line 2 and run the following command in the shell command window before running the code.

```
pip install -q segmentation-models-pytorch
```

```
1  # Dataset: https://www.kaggle.com/datasets/bulentsiyah/semantic-
       drone-dataset
2  !pip install -q segmentation-models-pytorch
3  import os, torch
4  import cv2 as cv
5  import numpy as np
6  import pandas as pd
7  import matplotlib.pyplot as plt
8  from torch.optim import AdamW as AW
9  import torch.nn as nn
10 from torch.utils.data import Dataset as ds
11 from torch.utils.data import DataLoader as DL
12 from torchvision import transforms as Trans
13 import torch.nn.functional as F
14 import segmentation_models_pytorch as smpy
15 from PIL import Image
16 import albumentations as ae
17 from sklearn.model_selection import train_test_split
18 from torch.optim.lr_scheduler import OneCycleLR as ocLR
19 # ---------------Read the dataset & Preprocess-------------------
20 full_path = []
21 image_folder = "../input/semantic-drone-dataset/dataset/
       semantic_drone_dataset/original_images/"
22 label_folder = "../input/semantic-drone-dataset/dataset/
       semantic_drone_dataset/label_images_semantic/"
23
24 for _, _, file_name_list in os.walk(image_folder):
25     for file_name in file_name_list:
26         full_path.append(file_name.split('.')[0])
27
28 df = pd.DataFrame({'index_id': full_path}, index = np.arange(0,
       len(full_path)))
29
30
31 Xtrain_val, Xtest = train_test_split(df['index_id'].values,
       test_size=0.2)
32 Xtrain, Xval = train_test_split(Xtrain_val, test_size=0.2)
33
34
35 # ----------------Define dataset & dataloader--------------------
```

```python
class CreateTrainDataset(ds):
    def __init__(self, image_dir, mask_dir, list_files, mean, std, transform=None, patch=False):
        self.image_dir = image_dir
        self.mask_dir = mask_dir
        self.list_files = list_files
        self.transform = transform
        self.patches = patch
        self.mean = mean
        self.std = std

    def __len__(self):
        return len(self.list_files)

    def __getitem__(self, id):
        images = cv.imread(self.image_dir + self.list_files[id] + '.jpg')
        images = cv.cvtColor(images, cv.COLOR_BGR2RGB)
        masks = cv.imread(self.mask_dir + self.list_files[id] + '.png', cv.IMREAD_GRAYSCALE)

        if self.transform is not None:
            aug = self.transform(image=images, mask=masks)
            images = Image.fromarray(aug['image'])
            masks = aug['mask']

        if self.transform is None:
            images = Image.fromarray(images)

        tr = Trans.Compose([Trans.ToTensor(), Trans.Normalize(self.mean, self.std)])
        images = tr(images)
        masks = torch.from_numpy(masks).long()

        if self.patches:
            images, masks = self.divide_into_patches(images, masks)

        return images, masks

    def divide_into_patches(self, images, masks):
        image_patches = images.unfold(1, 512, 512).unfold(2, 768, 768)
        image_patches = image_patches.contiguous().view(3, -1, 512, 768)
        image_patches = image_patches.permute(1, 0, 2, 3)
        masks_patches = masks.unfold(0, 512, 512).unfold(1, 768, 768)
        masks_patches = masks_patches.contiguous().view(-1, 512, 768)


        return image_patches, masks_patches
```

```python



# ------Transformation for train and validation dataset----------
transformation_train = ae.Compose([ae.Resize(800, 1216,
    interpolation=cv.INTER_NEAREST),
                                   ae.HorizontalFlip(),
                                   ae.VerticalFlip(),
                                   ae.GridDistortion(p=0.2),
                                   ae.RandomBrightnessContrast
    ((0,0.5),(0,0.5)),
                                   ae.GaussNoise()])

transformation_val = ae.Compose([ae.Resize(800, 1216,
    interpolation=cv.INTER_NEAREST),
                                 ae.HorizontalFlip(),
                                 ae.GridDistortion(p=0.2)])

# ---------------Dataset & dataloader creation-------------------
mean_values = [0.485, 0.456, 0.406]
std_values  = [0.229, 0.224, 0.225]
training_dataset = CreateTrainDataset(image_folder, label_folder,
     Xtrain, mean_values, std_values, transformation_train, patch
    =False)
validation_dataset = CreateTrainDataset(image_folder,
    label_folder, Xval, mean_values, std_values,
    transformation_val, patch=False)


bs= 3 #Batch Size
train_dataloader = DL(training_dataset,
                      batch_size=bs, shuffle=True)
val_dataloader = DL(validation_dataset,
                    batch_size=bs, shuffle=True)


# -----------------------Define model----------------------------
Unet_model = smpy.Unet('mobilenet_v2',
                  encoder_weights='imagenet',
                  classes=23,
                  activation=None,
                  encoder_depth=5,
                  decoder_channels=[256, 128, 64, 32, 16])



# -----------------Define training functions--------------------
def get_accuracy_score(output_tensor, mask_tensor):
    with torch.no_grad():
        output_tensor = torch.argmax(F.softmax(output_tensor, dim
    =1), dim=1)
        correct_predictions = torch.eq(output_tensor, mask_tensor
    ).int()
```

```
        accuracy_score = float(correct_predictions.sum()) / float
    (correct_predictions.numel())
    return accuracy_score

def lr_from_optimizer(optimizer):
    for param_group in optimizer.param_groups:
        return param_group['lr']

def fit(num_of_epochs, Unet_model, train_dataloader,
    val_dataloader, loss_criterion, optimizer, scheduler, patch=
    False):
    torch.cuda.empty_cache()
    training_losses = []
    testing_losses = []
    validation_accuracy = []
    training_accuracy = []
    learning_rates = []
    min_val_loss = np.inf
    decrease_counter = 1;
    no_improvement_count = 0

    Unet_model.to(device)
    for e in range(num_of_epochs):
        running_loss = 0
        accuracy_score = 0
        Unet_model.train() # Set model to training mode
        for i, data in enumerate(train_dataloader): # Iterate
    over training data loader
            image_tiles, mask_tiles = data
            if patch: # If patch-based training enabled, image
    and mask need to be flattened
                bs, num_tiles, channels, height, width =
    image_tiles.size()
                image_tiles = image_tiles.view(-1, channels,
    height, width)
                mask_tiles = mask_tiles.view(-1, height, width)

            image = image_tiles.to(device); # Forward pass image
            mask_tensor = mask_tiles.to(device);
            output_tensor = Unet_model(image)
            loss = loss_criterion(output_tensor, mask_tensor) #
    Calculate the loss
            accuracy_score += get_accuracy_score(output_tensor,
    mask_tensor) #Accuracy measurement for evaluation
            loss.backward() # Backpropagate the loss
            optimizer.step() # Update model weights
            optimizer.zero_grad() # Reset gradient
            learning_rates.append(lr_from_optimizer(optimizer)) #
     Update learning rate
            scheduler.step()
            running_loss += loss.item()

        else:
```

```
        Unet_model.eval() # Set model to evaluation mode
        testing_loss = 0
        testing_accuracy = 0
        with torch.no_grad(): # Iterate over validation data loader
            for i, data in enumerate(val_dataloader):
                image_tiles, mask_tiles = data
                if patch:
                    bs, num_tiles, channels, height, width = image_tiles.size()
                    image_tiles = image_tiles.view(-1, channels, height, width)
                    mask_tiles = mask_tiles.view(-1, height, width)

                image = image_tiles.to(device);
                mask_tensor = mask_tiles.to(device);
                output_tensor = Unet_model(image)
                testing_accuracy += get_accuracy_score(output_tensor, mask_tensor)
                loss = loss_criterion(output_tensor, mask_tensor)
                testing_loss += loss.item()

        training_losses.append(running_loss / len(train_dataloader))
        testing_losses.append(testing_loss / len(val_dataloader))

        if min_val_loss > (testing_loss / len(val_dataloader)):
            print('Loss Decreasing {:.3f} >> {:.3f} '.format(min_val_loss, (testing_loss / len(val_dataloader))))
            min_val_loss = (testing_loss / len(val_dataloader))
            decrease_counter += 1
            if decrease_counter % 5 == 0:
                print('Model saved')
                torch.save(Unet_model, 'Unet-Mobilenet_v2_acc-{:.3f}.pt'.format(testing_accuracy / len(val_dataloader)))

        if (testing_loss / len(val_dataloader)) > min_val_loss:
            no_improvement_count += 1
            min_val_loss = (testing_loss / len(val_dataloader))
            print(f'The loss has not decreased for {no_improvement_count} iterations.')
            if no_improvement_count == 50:
                print('The loss has not decreased in the last 50 iterations, so stop training.')
                break

            training_accuracy.append(accuracy_score / len(train_dataloader))
            validation_accuracy.append(testing_accuracy / len(val_dataloader))
            print("\nEpoch {}/{}:".format(e + 1, num_of_epochs),
                  "\nTrain Loss: {:.3f}".format(running_loss / len(train_dataloader)),
                  "Val Loss: {:.3f}".format(testing_loss / len(val_dataloader)),
                  "\nTrain Accuracy: {:.3f}".format(accuracy_score / len(train_dataloader)),
                  "Val Accuracy: {:.3f}".format(testing_accuracy / len(val_dataloader)) )

    train_results = {'training_losses': training_losses, 'testing_losses': testing_losses,
               'training_accuracy': training_accuracy, 'validation_accuracy': validation_accuracy}
    return train_results


# ------------------------Training loop-------------------------
max_learning_rate = 1e-3
epoch_no = 100
w_decay = 1e-4

loss_criterion = nn.CrossEntropyLoss()
optimizer = AW(Unet_model.parameters(), lr=max_learning_rate, weight_decay=w_decay)
sched = ocLR(optimizer, max_learning_rate, epochs=epoch_no,
                                            steps_per_epoch=len(train_dataloader))
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
train_results = fit(epoch_no, Unet_model, train_dataloader, val_dataloader, loss_criterion, optimizer, sched)

torch.save(Unet_model, 'Unet-Mobilenet.pt')

# ------------------------Plot results-------------------------
plt.plot(train_results['testing_losses'], label='val_loss', marker='o')
plt.plot(train_results['training_losses'], label='train_loss', marker='o')
plt.plot(train_results['training_accuracy'], label='train_accuracy', marker='P')
plt.plot(train_results['validation_accuracy'], label='val_accuracy', marker='P')
plt.title('Loss/Accuracy per epoch');
plt.ylabel('loss');
plt.xlabel('epoch')
plt.legend(), plt.grid()
plt.show()
```

```

train_results_df = pd.DataFrame(train_results, columns = ['
    training_losses','testing_losses','training_accuracy','
    validation_accuracy'])
train_results_df.to_csv("plot_data.csv", index = False)


# -------------------------Evaluation-----------------------------
class CreateTestDataset(ds):

    def __init__(self, image_dir, mask_dir, list_files, transform
    =None):
        self.image_dir = image_dir
        self.mask_dir = mask_dir
        self.list_files = list_files
        self.transform = transform

    def __len__(self):
        return len(self.list_files)

    def __getitem__(self, id):
        images = cv.imread(self.image_dir + self.list_files[id] +
     '.jpg')
        images = cv.cvtColor(images, cv.COLOR_BGR2RGB)
        masks = cv.imread(self.mask_dir + self.list_files[id] + '
    .png', cv.IMREAD_GRAYSCALE)

        if self.transform is not None:
            aug = self.transform(image=images, mask=masks)
            images = Image.fromarray(aug['image'])
            masks = aug['mask']

        if self.transform is None:
            images = Image.fromarray(images)

        masks = torch.from_numpy(masks).long()

        return images, masks


transformation_test = ae.Resize(800, 1216, interpolation=cv.
    INTER_NEAREST)
test_dataset = CreateTestDataset(image_folder, label_folder,
    Xtest, transform=transformation_test)



def predict_image_mask_acc(Unet_model, image, masks, mean=[0.485,
     0.456, 0.406], std=[0.229, 0.224, 0.225]):
    Unet_model.eval()
    tr = Trans.Compose([Trans.ToTensor(), Trans.Normalize(mean,
    std)])
    image = tr(image)
```

```python
    Unet_model.to(device);
    image = image.to(device)
    masks = masks.to(device)
    with torch.no_grad():
        image = image.unsqueeze(0)
        masks = masks.unsqueeze(0)

        output_tensor = Unet_model(image)
        acc = get_accuracy_score(output_tensor, masks)
        masked = torch.argmax(output_tensor, dim=1)
        masked = masked.cpu().squeeze(0)
    return masked, acc


def acc(Unet_model, test_dataset):
    accuracy_score = []
    for i in range(len(test_dataset)):
        images, masks = test_dataset[i]
        pred_mask, acc = predict_image_mask_acc(Unet_model,
    images, masks)
        accuracy_score.append(acc)
    return accuracy_score

t_acc = acc(Unet_model, test_dataset)

print('Test Accuracy: ', np.mean(t_acc))


# -------------Cross validate with ground truth------------------
path = '../working/fig'
if not os.path.exists(path):
    os.makedirs('../working/fig')

for n in (32, 34, 36):
    image, masks = test_dataset[n]
    pred_mask, score = predict_image_mask_acc(Unet_model, image,
    masks)

    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(12, 4))
    ax1.imshow(image)
    ax1.set_title('Picture {:d}'.format(n));
    ax1.set_axis_off()
    ax2.imshow(masks)
    ax2.set_title('Ground truth')
    ax2.set_axis_off()
    ax3.imshow(pred_mask)
    ax3.set_title('Predicted | Accuracy {:.3f}'.format(score))
    ax3.set_axis_off()
    plt.savefig('../working/fig/' + str(n) + '.png', format='png'
    , dpi=300, facecolor='white', bbox_inches='tight',
                pad_inches=0.25)
    plt.show()
```

Listing 6.3 Aerial semantic segmentation using U-Net architecture

Output of Listing 6.3:

```
Loss Decreasing inf >> 2.269

Epoch 1/100:
Train Loss: 2.786 Val Loss: 2.269
Train Accuracy:0.220 Val Accuracy:0.452
Loss Decreasing 2.269 >> 1.848

Epoch 2/100:
Train Loss: 2.298 Val Loss: 1.848
Train Accuracy:0.468 Val Accuracy:0.600
Loss Decreasing 1.848 >> 1.587

Epoch 3/100:
Train Loss: 2.012 Val Loss: 1.587
Train Accuracy:0.565 Val Accuracy:0.689
Loss Decreasing 1.587 >> 1.353
Model saved

Epoch 4/100:
Train Loss: 1.731 Val Loss: 1.353
Train Accuracy:0.633 Val Accuracy:0.702
Loss Decreasing 1.353 >> 1.217

Epoch 5/100:
Train Loss: 1.532 Val Loss: 1.217
Train Accuracy:0.652 Val Accuracy:0.709
Loss Decreasing 1.217 >> 1.076


...      ...      ...
...      ...      ...
...      ...      ...

Epoch 97/100:
Train Loss: 0.240 Val Loss: 0.304
Train Accuracy:0.924 Val Accuracy:0.908
Loss Not Decrease for 43 time

Epoch 98/100:
Train Loss: 0.238 Val Loss: 0.313
Train Accuracy:0.926 Val Accuracy:0.908
Loss Decreasing 0.313 >> 0.308

Epoch 99/100:
```

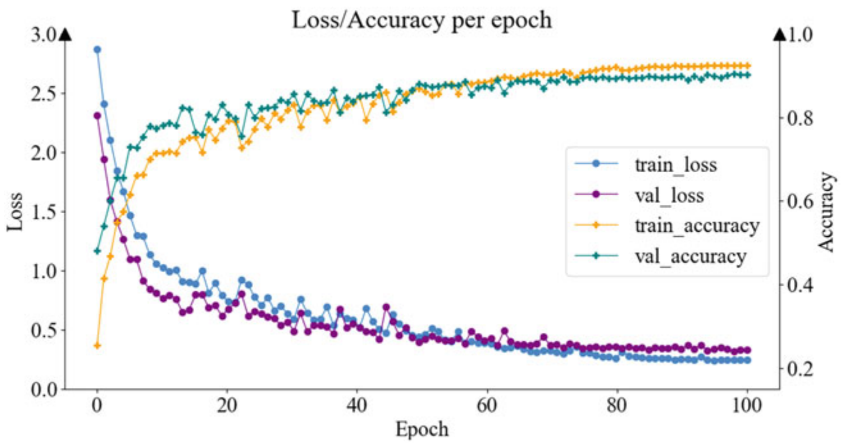


Fig. 6.7 Loss/accuracy per epoch

```
Train Loss: 0.249 Val Loss: 0.308
Train Accuracy:0.923 Val Accuracy:0.908
Loss Decreasing 0.308 >> 0.291

Epoch 100/100:
Train Loss: 0.244 Val Loss: 0.291
Train Accuracy:0.924 Val Accuracy:0.912

Test Accuracy:  0.9042500770970395
```

6.3 Robot: A Line Follower Data Predictor Using Generative Adversarial Network (GAN)

In this section, we demonstrate a semi-supervised algorithm, the *Generative Adversarial Network (GAN)* [12], which uses a generator and a discriminator to learn a given task. The generator's job is to produce the desired output with some random numbers as input. In other words, the generator learns to predict sequential data by observing random noise at the input. The algorithm uses a discriminator network to challenge the generation of the generator. Hence, the generator itself is a semi-supervised data predictor that does not use labeled data but utilizes the discriminator to learn to predict a given data sequence.

Programming Example 6.4
We have already provided a comprehensive description of the GAN algorithm in Chap. 3. Here, we will use a simple GAN architecture to fit a path according to a given equation $y = 7x^2 + 2x + 1$. One can imagine this GAN as a robot that learns to predict data points that match the non-linear equation presented above. The Python

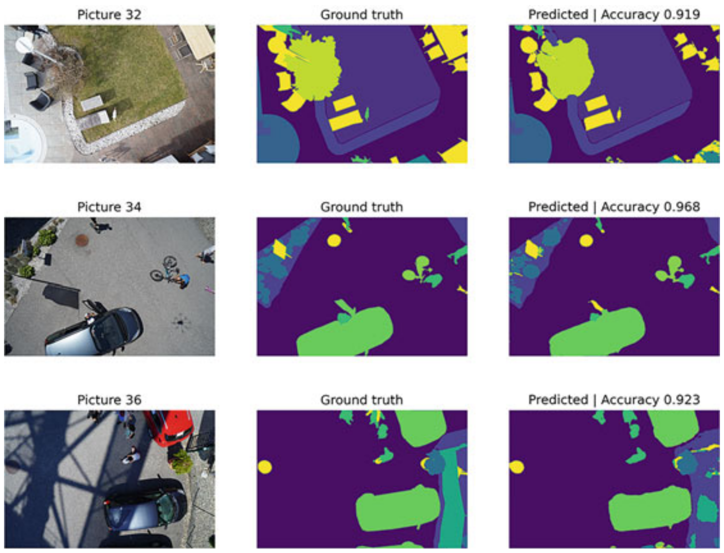

Picture 32
Ground truth
Predicted | Accuracy 0.919
Picture 34
Ground truth
Predicted | Accuracy 0.968
Picture 36
Ground truth
Predicted | Accuracy 0.923

Fig. 6.8 Cross-validate predicted results with ground truth

code to build the GAN is provided in Listing 6.4, followed by its explanation in Table 6.4. Figure 6.9 depicts the code output.

```
# ----------------------------Modules----------------------------
import torch
from torch import nn, optim
from torch.autograd import Variable
import numpy as np
from numpy.random import randn
from matplotlib import pyplot as plt
import math
import torch.nn.functional as F


# ---------------------------Variables---------------------------
batch_s = 256 # batchsize
Iterations = 5000 # iteraiton for training
plot_epoch = 4999 # final output printing epoch
learning_rate = 0.001 # learning Rate
d_step = 10 # generator training steps
g_step = 10 # discriminator training steps


# ------------------Dataset Using an Equation------------------
# We want to fit a path to the following equation:
# 7x^2 + 2x + 1

def functions(x):
    return  7*x*x + 2*x + 1

def data_generation():

    'This function generates the data of batch size = batch_s'

    data = []
    x = 20 * randn(batch_s) # random inputs

    for i in range(batch_s):
        y = functions(x[i])
        data.append([x[i], y]) # dataset

    return torch.FloatTensor(data)

def plotting(real, fake, epoch):

    'plotting the real and fake data'

    x, y = zip(*fake.tolist())
    plt.scatter(x, y, label='Generated Data')
    x, y = zip(*real.tolist())
    plt.scatter(x, y, label='Original Data')
    plt.legend(loc='upper left')
    plt.xlabel("inputs")
    plt.savefig('GAN.png', bbox_inches='tight')
    plt.show()


# -------------Defining Generator and Discriminator--------------

class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        "Generator Model 3-layer fully connected"
        self.layer1 = nn.Linear(4, 20)
        self.layer2 = nn.Linear(20, 10)
        self.layer3 = nn.Linear(10, 2)


    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        x = self.layer3(x)
        return x

generator = Generator()


class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        "Dicriminator Model 3-layer fully connected"
        self.layer1 = nn.Linear(2, 20)
        self.drop1  =  nn.Dropout(0.4)

        self.layer2 = nn.Linear(20, 10)
        self.drop2  =  nn.Dropout(0.4)

        self.layer3 = nn.Linear(10, 1)

    def forward(self, x):
        x = F.leaky_relu(self.drop1(self.layer1(x)))
        x = F.leaky_relu(self.drop2(self.layer2(x)))
        x = torch.sigmoid(self.layer3(x))
        return x

discriminator = Discriminator()


# Setting up the models
generator = Generator()
discriminator = Discriminator()



# Define Optimizer
optimizer_D = optim.Adam(discriminator.parameters(), lr=
    learning_rate)
optimizer_G = optim.Adam(generator.parameters(), lr=learning_rate
    )


# -----------------------Training Function-----------------------

def train(Iterations,optimizer_D,optimizer_G,discrimator,
    generator):

    "this function trains both generator and discriminator"
    # Set the models to training mode
    discriminator.train()
    generator.train()
    loss_f = nn.BCELoss() # BCE loss function

    for epoch in range(Iterations):
        # discriminator training
        for d_steps in range(d_step):

            data = data_generation()
            z =  torch.randn([batch_s, 4])


            # no gradient computation at this stage
            fake_data = generator(z).detach()


```

```python
            sizes = data.size()[0]
            optimizer_D.zero_grad()

            #      the real data update
            prediction_real = discriminator(data)
            d_loss = loss_f(prediction_real, torch.ones(sizes, 1).fill_(0.95))
            d_loss.backward()

            #     fake data update
            prediction_generated = discriminator(fake_data)
            loss_generated = loss_f(prediction_generated, torch.zeros(sizes, 1))
            loss_generated.backward()

            optimizer_D.step()

        for g_steps in range(g_step):
            z =  torch.randn([batch_s, 4])


            fake_data = generator(z)

            optimizer_G.zero_grad()

            #     Run the generated data through the discriminator
            prediction = discriminator(fake_data)

            #     Train the generator with the smooth target, i.e. the target is 0.95
            loss_gen = loss_f(prediction, torch.ones(sizes, 1).fill_(0.95))
            loss_gen.backward()

            optimizer_G.step()

        print("epoch:", epoch)
        print("Genrator Loss:",d_loss)
        print("Discriminator Loss:",loss_gen)

        if((epoch+1) % plot_epoch == 0):
            plotting(data, fake_data, epoch)


# ---------------------------Training---------------------------
train(Iterations,optimizer_D,optimizer_G,discriminator,generator)
```

Listing 6.4 LFR path generation using GAN

Table 6.4 Explanation of the GAN example in Listing 6.4

Line number	Description
2–10	Import PyTorch modules
13–18	Hyper-parameters
25–26	$y = 7x^2 + 2x + 1$ function based on which the robot will fit a path
28–39	Data generation function
41–51	Plotting function
57–72	Defining the generator model
75–93	Defining the discriminator model
103–105	Two optimizers for generator and discriminator
109–166	Training function declaration
170	Training and plotting output

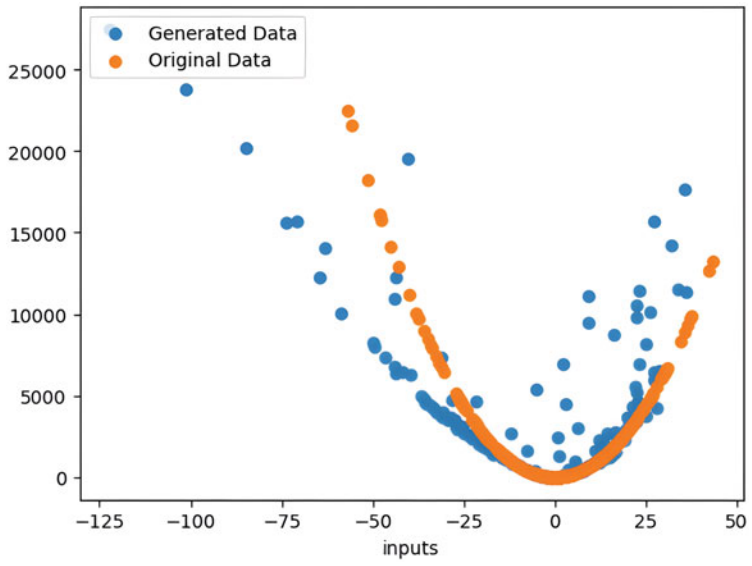


Fig. 6.9 The original line data are shown in blue, and the line following robot's predicted data is shown in orange

6.4 Conclusion

The combination of machine learning and robotics can be regarded as the most powerful technological combination of humankind up to date. Machine learning and artificial intelligence allow robots to think and intelligently interact with the environment. Machine learning enhances robots' vision, motion, sensing, and environment interaction abilities by providing them with a sense of intelligence;

thus, the robots are no longer confined to the barriers of pre-programmed activities. This chapter introduces the concept of machine learning applications in robotics through some basic theoretical concepts and real-life examples with programming examples. This chapter discusses computer vision, which spans object tracking, detection, and image segmentation and provides an example of a line follower robot using a generative adversarial network. This chapter provides the readers with basic concepts of robot learning, which will help them venture into further advanced concepts in this domain. The readers are encouraged to solve the problems, try them with different input images, and analyze the outputs. In the next chapter, we will study some state-of-the-art technologies of machine learning and artificial intelligence and their future possibilities.

6.5 Key Messages

- While interacting with the environment, robots come across a large data stream. Machine learning can utilize these data to provide intelligence to robots, making them smarter and more efficient.
- Computer vision and machine vision are integrated with the world of robotics. Robots gain the power of sight through computer vision and machine vision, which help them see and analyze various elements of their surrounding environment.
- The application of machine learning in robotics has freed the robots from their pre-programmed and confined boundaries, thus giving them a true sense of automation.

6.6 Exercise

1. Briefly explain the role of machine learning in robotics. How significant is the influence of machine learning in robotics?
2. Show some differences between machine learning and machine vision.
3. Develop a face detection model and implement it on a video you have captured. Make one video containing only one face and another containing several (3-5) faces. Does your model detect faces correctly?
4. Self-driving car works based on object tracking. Provide a high-level overview of implementing object tracking in a self-driving car environment.
5. Implement an object tracking model on the MS COCO (Microsoft Common Objects in Context) dataset [13].
6. Design a hyperbolic, parabolic, and half-circular path for a line follower robot (LFR). Train separate GAN models for the LFR to follow each of the paths. Can the LFR predict each of the paths equally well?

References

1. *Face recognition using Pytorch*. https://github.com/timesler/facenet-pytorch
2. Zhang, K., Zhang, Z., Li, Z., & Qiao, Y. (2016). Joint face detection and alignment using multitask cascaded convolutional networks. *IEEE Signal Processing Letters, 23*(10), 1499–1503.
3. *FaceNet example video*. https://github.com/timesler/facenet-pytorch/blob/master/examples/video.mp4
4. *Face tracking pipeline*. https://github.com/timesler/facenet-pytorch/blob/master/examples/face_tracking.ipynb
5. *Yolo: Real-time object detection*. https://pjreddie.com/darknet/yolov2/
6. *GTSRB - German traffic sign recognition benchmark*. https://www.kaggle.com/datasets/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign
7. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770–778).
8. *DeepSteal/TSIGN.py at main · adnansirajrakin/DeepSteal*—github.com. https://github.com/adnansirajrakin/DeepSteal/blob/main/TSIGN.py. Accessed September 07, 2023.
9. Unsplash. *Photo by Andrey Haimin on Unsplash*—unsplash.com (2020). https://unsplash.com/photos/bpPJ2RSCyPE. Accessed September 09, 2023.
10. Ronneberger, O., Fischer, P., & Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. *arXiv:1505.04597*.
11. *Semantic drone dataset* (2019). https://www.tugraz.at/index.php?id=22387
12. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2020). Generative adversarial networks. *Communications of the ACM, 63*(11), 139–144.
13. *COCO dataset*. https://cocodataset.org/#home

7 State of the Art of Machine Learning

7.1 Introduction

With technological and hardware advancements, machine learning (ML) has started influencing almost every aspect of our daily lives. The goal of ML is for machines to perform specific tasks the way humans do but without human intervention. ML has been applied to all aspects of modern life, from detecting objects to conversing with other human beings. ML has left its footprints in computer vision, natural language processing, the medical industry, music, art, fashion, and more. State-of-the-art technologies have been developed to achieve desired goals with high accuracy. Various ML models have been able to detect objects in real-time and take action accordingly, work with more complex structure data by the day, and mimic how a human communicates with another human. On the flip side, the use of ML and artificial intelligence (AI) also poses some threats. Since ML and AI have been so involved in our day-to-day prospects, privacy breaches, data loss, temperament of data, and other cybersecurity-related issues have become a concern. However, technologies and software are continuously being developed to make AI safer and mitigate the risks associated with ML and AI.

In this chapter, we present the state of the art of ML by discussing the latest advancements in ML, such as graph neural networks, EfficientNet, Inception v3, YOLO, Facebook Prophet, and ChatGPT. Next, we discuss the security challenges associated with ML/AI and their possible solutions, followed by the hardware challenges of ML/AI and its future potential. This discussion includes the concepts of quantization and weight pruning. Then, we will discuss multi-domain learning, including transfer learning and domain adaptation. Finally, we will discuss AI in greater detail, including the Turing test, AI limitations, and future possibilities.

E. Hossain, *Machine Learning Crash Course for Engineers*,
https://doi.org/10.1007/978-3-031-46990-9_7

7.2 State-of-the-Art Machine Learning

The journey of ML has been long and adorned with numerous milestones. New possibilities emerge daily with the consistent and accelerating pace of research and developments in this field. In this section, we will talk about some latest developments in ML to present the current state-of-the-art of the field of ML.

7.2.1 Graph Neural Network

In graph theory, a graph may consist of several nodes and edges. City maps, chemical compositions, protein structures, and many other types of data and information can be organized using graph structure. The nodes of a graph can represent cities, protein units, or any other object according to the problem, and the edges represent the relationship among the nodes. They are intricate data structures that can be of any size or shape. Graphs can also have dynamic structures, i.e., they do not have a fixed number of nodes and edges. No data point in a graph can be indicated using 2D or 3D coordinates, as graphs do not have any Euclidean structure.

Although a graph is an excellent data structure for representing many types of real-world data and information, its intricacy and amorphous nature make it incompatible with commonly used neural networks. For example, a convolutional neural network (CNN) is a very commonly used neural network structure that operates on static and easier-to-understand data formats like arrays, and for this reason, CNNs and other standard neural networks cannot operate on graphs. The graph neural network (GNN) is introduced to solve this data structure incompatibility issue.

GNNs apply the predictive capability of deep learning to complex data structures that represent items and their interactions as points connected by lines in a graph. To enable ML algorithms to make valuable predictions at the level of nodes (data points), edges, or complete graphs, GNNs link data points together by lines known as edges. The overall workflow of the GNN is shown in Fig. 7.1.

7.2.1.1 Applications of GNN

Some practical applications of GNN are briefly discussed below.

1. **Computer Vision:** A fundamental task in computer vision is image classification. Most models produce satisfactory results when presented with a sizable training set of classes with labels. GNN can fulfill the current goal of improving these models' performance on tasks requiring zero-shot (a classification technique where the model learns to classify objects from previously unseen classes) and few-shot learning (a classification technique where the model is pre-trained using a few training samples to generalize over new dataset). Other uses for computer vision tasks include region classification, interaction detection, and object detection. In human-computer interaction (HCI), GNNs operate as message-passing tools between humans and objects.

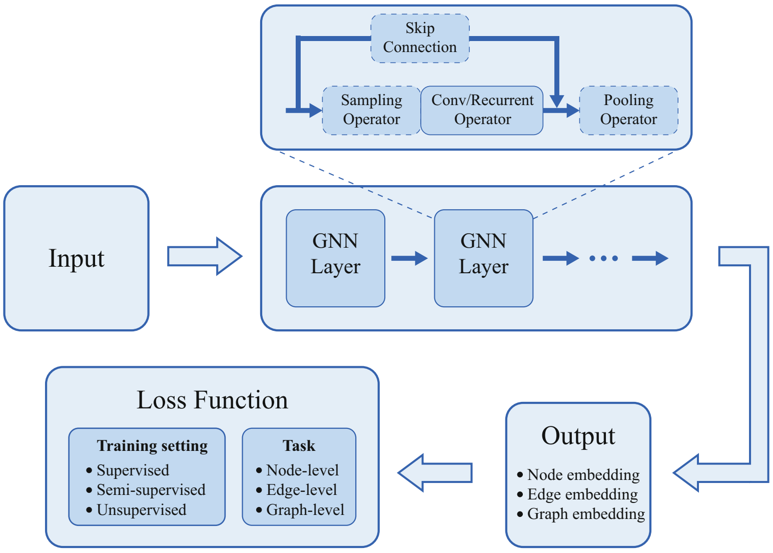


Fig. 7.1 Workflow of graph neural network

2. **Natural Language Processing (NLP):** One of the most significant tasks of GNN in NLP is text classification. Texts can be represented as graphs, and GNN uses their interrelations among the words to perform the classification task. Text classifications can be used for sentimental analysis, news classification, question and answer segments, neural machine translations, etc. In addition, using a graph to work with texts can take advantage of non-consecutive words.
3. **Molecular Biology:** AlphaFold is an AI-based software, used to predict protein structures. It has been specifically designed to solve the problem of protein folding in biology. Protein folding is the process through which it gains its biological functionality. AlphaFold can also be used to study and predict protein structures, which leads to the development of potential drugs and medicine.
4. **Social Networks:** Another significant use case of GNN is in the domain of social media. GNN uses social graphs to develop a recommendation system for social users. The friend circle or the social interaction group of a user mainly influences the prediction of this recommendation system. For example, if most of the friends on the list are from a similar background or race, the recommendation system will keep recommending content more likely to be consumed by that particular background or race. The Penn State team has been working with GNN on social media influence to decrease this bias.
5. **Particle Physics:** The Compact Muon Solenoid (CMS) detector, a general-purpose particle physics detector, at the Large Hadron Collider at CERN generates images. Researchers at Fermilab use GNN to analyze these CMS-generated images and look for objective particles for particle physics experiments.

6. **Molecular Chemistry:** Molecular fingerprinting is a method to represent a molecular structure. Researchers use GNN to study, analyze, and research the structures of different chemical compounds and structures from these molecular fingerprints. The molecular fingerprints can also be analyzed and studied to synthesize new chemical compounds that can lead to potential drug discovery.
7. **Cyber-security:** A network of computers can be represented as graphs. GNN can be used on such graphs to study and analyze anomalies. In order to detect malicious activities and identify these anomalies on individual nodes, within pathways, or at the edge level to detect lateral movement, GNNs have been utilized.
8. **Traffic Jam Prediction:** A key component of an intelligent transportation system is the ability to forecast traffic volume, speed, or road density. The traffic network can be represented as a spatial-temporal graph, with nodes representing the sensors placed on roadways, edges representing the separation between pairs of nodes, and dynamic input features representing the average traffic speed within a window for each node in a GNN-based traffic solution. Google DeepMind uses GNN to estimate the potential routes and travel time. Figure 7.2 demonstrates the workflow of GNN to predict traffic jams using Google Maps API.
9. **Combinatorial Optimization:** The goal of combinatorial optimization (CO) is to select the best object from limited options. It is the foundation for numerous crucial applications in science, engineering, finance, and logistics. The majority of CO problems are represented as graphs. In a recent project by DeepMind and Google, bounding the goal value and joint variable assignment are the two main subtasks for the MILP solver that uses graph nets. On large datasets, their neural network technique outperforms conventional algorithms in speed.

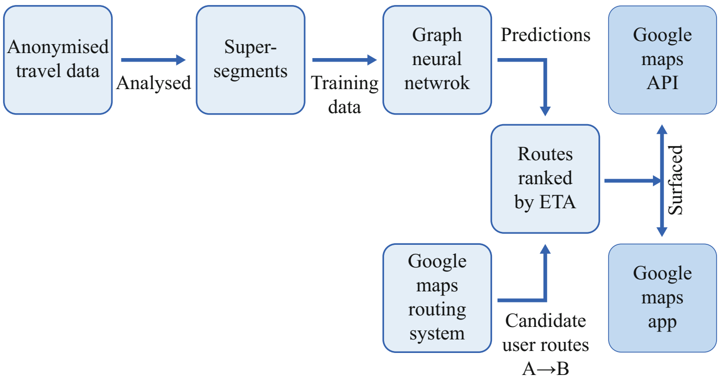


Fig. 7.2 An application of Graph Neural Network on traffic jam prediction

7.2.2 EfficientNet

Convolutional neural networks (CNNs) are often built at a fixed resource cost and then scaled up when more resources become available to attain higher accuracy. This scaling strategy has typically contributed to greater accuracy on most benchmarking datasets. However, the traditional methods of model scaling are quite sporadic. Some models have depth scaling, while others have width scaling. To achieve better results, some models merely utilize photos with a higher resolution. This method of arbitrarily scaling models necessitates human adjustment and numerous working hours, frequently yielding little to no performance increase.

EfficientNet is a type of CNN architecture that uses a *compound coefficient* method to scale up models quickly and easily. Compound scaling uniformly scales each dimension with a predetermined fixed set of scaling coefficients instead of randomly increasing width, depth, or resolution. The developers of EfficientNet created seven models in different dimensions using the scaling approach and AutoML, outperforming most convolutional neural networks' state-of-the-art accuracy while operating far more effectively.

The authors of EfficientNet thoroughly investigated the effects of each scaling strategy on the functionality and effectiveness of the model before formulating the compound scaling method. It was observed that scaling one dimension enhances model performance. Similarly, scaling all three dimensions—width, depth, and image resolution—while considering the varied resources can significantly improve the model's overall performance.

The idea behind neural networks is that larger input images require adding more layers to enhance the receptive field and adding more channels to capture more minute patterns on the larger image. Compared to other random scaling techniques, the compound scaling strategy also helped enhance the efficiency and accuracy of earlier CNN models, such as MobileNet and ResNet, by roughly 1.4% and 0.7% ImageNet accuracy, respectively.

Based on the baseline network created by the neural architecture search utilizing the AutoML MNAS framework, EfficientNet was created. The network is adjusted for optimum accuracy, but it also suffers if it requires a lot of calculation. If the network takes a long time to make predictions, it is also punished for slow inference time. Due to the increase in FLOPS, the architecture uses a mobile inverted bottleneck convolution that is larger than MobileNet V2. To create the EfficientNet family, this basic model is scaled up.

For EfficientNet, only the B0 model has been shown in the example. The weights of the pre-trained EfficientNetB0 have been downloaded. The weights have been achieved by training the model on the ImageNet dataset with 1000 classes. The next 7 models can also be used by importing the weights by simply calling EfficientNetB1—EfficientNetB7.

Table 7.1 Explanation of the code for implementing EfficientNet in Listing 7.1

Line number	Description
1–12	Importing necessary libraries
15–35	Defining a custom image classification model based on the EfficientNetB0 architecture
39–48	Defining the model training process
50–59	Visualizing the training and validation accuracy
61–77	Defining function to predict class labels for unseen image
81–82	Initialize a model and print its summary
85–87	Loading the CIFAR-10 dataset, splitting it into training and testing sets, and one-hot encoding class labels
95–96	Calculating the test accuracy of the trained model
99–100	Employing the trained model to predict the class label of unseen images

Fig. 7.3 Input image to the model in Listing 7.1. (Photo by Tom Cattini on Pexels [1])

Programming Example 7.1
Listing 7.1 demonstrates the code for image classification using EfficientNet, and Table 7.1 explains the code. Figure 7.3 shows the image of an airplane. This image is fed to the ML model as the input, and the model is expected to identify the object in the image. This code uses the EfficientNetB0 architecture to train a model on the CIFAR-10 dataset. Several additional layers are added to create a modified version of the EfficientNetB0 model, which was trained on the CIFAR-10 dataset and used to make predictions on unseen images (Figs. 7.4 and 7.5).

Input image:

```
1 # -----------------Importing Required Libraries------------------
2 import cv2
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import tensorflow as tf
6 from tensorflow.keras.optimizers import Adam
7 from tensorflow.keras.models import Model
8 from tensorflow.keras.layers import Input, Dense,
      GlobalAveragePooling2D
```

```python
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.applications import EfficientNetB0
from tensorflow.keras.applications.efficientnet import
    preprocess_input


# ------------------------Model Formation------------------------
def modified_efficientNet(image_shape=(32, 32, 3), fc_units
    =(1024, 512)):
    input_src = Input(shape=image_shape)
    base_model = EfficientNetB0(
        include_top=False,
        weights='imagenet',
        input_shape=image_shape
    )(input_src)
    x = GlobalAveragePooling2D()(base_model)

    for units in fc_units:
        x = Dense(units, activation='relu')(x)

    output = Dense(10, activation='softmax')(x)

    model = Model(input_src, output)
    optimizer = Adam(learning_rate=0.0002, beta_1=0.5)
    model.compile(loss='categorical_crossentropy', optimizer=
    optimizer, metrics=['accuracy'])
    return model

    return x_train, y_train, x_test, y_test


# -----------------------Model Training-------------------------
def train_model(model, x_train, y_train, x_test, y_test, epochs
    =50, batch_size=32):
    predicted_output = model.fit(
        x_train,
        y_train,
        validation_data=(x_test, y_test),
        batch_size=batch_size,
        epochs=epochs,
        shuffle=True,
    )
    return predicted_output

def plot_accuracy(predicted_output):
    plt.plot(predicted_output.history['accuracy'])
    plt.plot(predicted_output.history['val_accuracy'])
    plt.title('Model Accuracy')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Test'])
    plt.savefig("./results/train_test.png")
    plt.show()


# -----------------Detection for Unseen Images-------------------
def predict_custom_image(model, img_path):
    img = cv2.imread(img_path)
    img = cv2.resize(img, (32, 32))
    x = np.expand_dims(img, axis=0)
    x = preprocess_input(x)

    preds = model.predict(x)
    predicted_label_index = np.argmax(preds)
    class_labels = ["airplane", "automobile", "bird", "cat", "
    deer", "dog", "frog", "horse", "ship", "truck"]
    my_image = plt.imread(img_path)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(my_image)
    plt.title('Predicted: {}'.format(class_labels[
    predicted_label_index]))
    plt.savefig("./results/predicted.png")
    plt.show()


# -------------------Training and Evaluation---------------------
EffNet = modified_efficientNet()
print(EffNet.summary())

# Split Dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)

predicted_output = train_model(EffNet, x_train, y_train, x_test,
    y_test)

# Plot training and validation accuracy
plot_accuracy(predicted_output)

# Evaluate the model on the test dataset
test_loss, test_acc = EffNet.evaluate(x_test, y_test)
print("Test accuracy:", test_acc)


# Predict an unknown image
predict_custom_image(EffNet, './data/img.jpg')
```

Listing 7.1 Image classification using EfficientNet

Output of Listing 7.1:

```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, 32, 32, 3)]       0
 efficientnetb0 (Functional)  (None, 1, 1, 1280)       4049571

 global_average_pooling2d_2   (None, 1280)             0
 (GlobalAveragePooling2D)

 dense_6 (Dense)             (None, 2048)             2623488

 dense_7 (Dense)             (None, 1024)             2098176

 dense_8 (Dense)             (None, 10)               10250

=================================================================
Total params: 8,781,485
Trainable params: 8,739,462
Non-trainable params: 42,023
_________________________________________________________________
Epoch 1/50
1563/1563 [==============================] - 123s 52ms/step
- loss: 1.1370 - accuracy: 0.6018
- val_loss: 0.7559 - val_accuracy: 0.7405
Epoch 2/50
1563/1563 [==============================] - 80s 51ms/step
- loss: 0.7509 - accuracy: 0.7421
- val_loss: 0.6585 - val_accuracy: 0.7714

... ... ...
... ... ...
... ... ...

Epoch 48/50
1563/1563 [==============================] - 75s 48ms/step
- loss: 0.0595 - accuracy: 0.9814
- val_loss: 0.7917 - val_accuracy: 0.8398
Epoch 49/50
1563/1563 [==============================] - 75s 48ms/step
- loss: 0.0711 - accuracy: 0.9790
- val_loss: 0.7920 - val_accuracy: 0.8425
Epoch 50/50
1563/1563 [==============================] - 75s 48ms/step
- loss: 0.0643 - accuracy: 0.9807
- val_loss: 0.7206 - val_accuracy: 0.8469
```

7.2.3 Inception v3

The convolutional neural network (CNN) named Inception v3 [2] was developed as a plugin for GoogLeNet and is used *to support object detection and image analysis*. The Google Inception CNN, first presented during the ImageNet Recognition Challenge, is currently in its fourth iteration. Nevertheless, the third iteration, i.e., *Inception v3*, is still the most popular Inception architecture. Inception v3 was created to enable deeper networks without making the number of parameters unmanageable. It has a total of 42 layers and contains around 25 million parameters, which is almost 40% of the number of parameters of AlexNet.

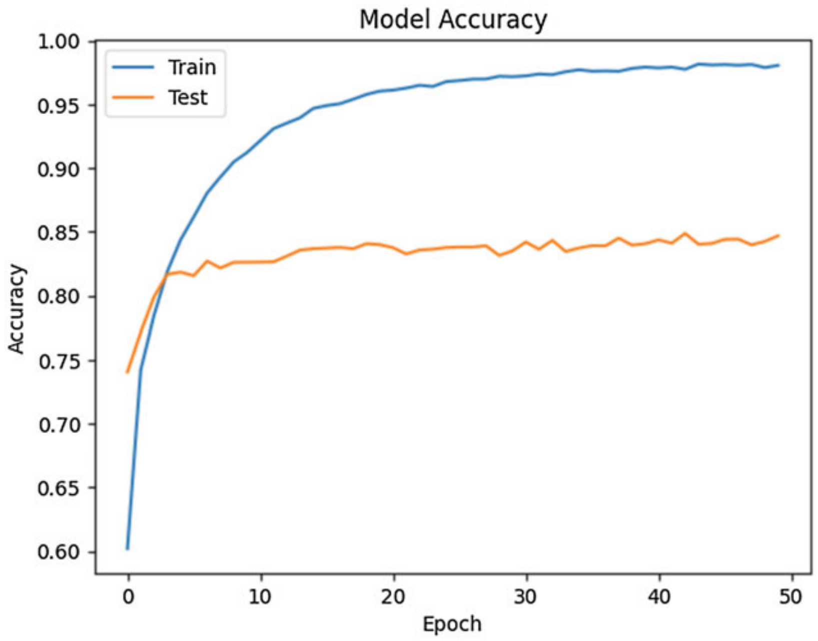
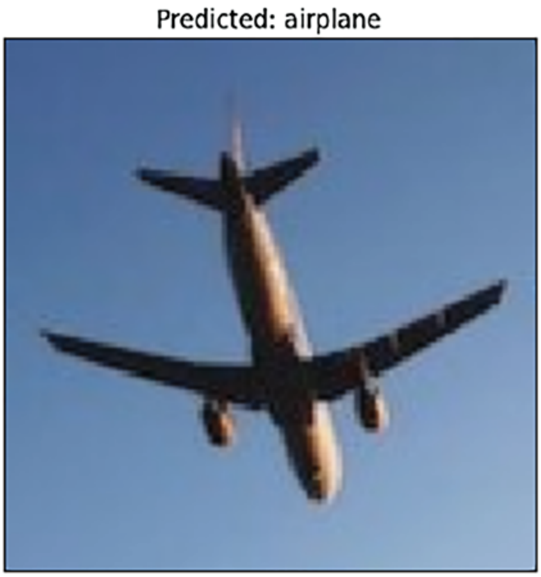


Fig. 7.4 Train and test accuracy compared with respect to epochs of Listing 7.1

Fig. 7.5 Output image with label of Listing 7.1



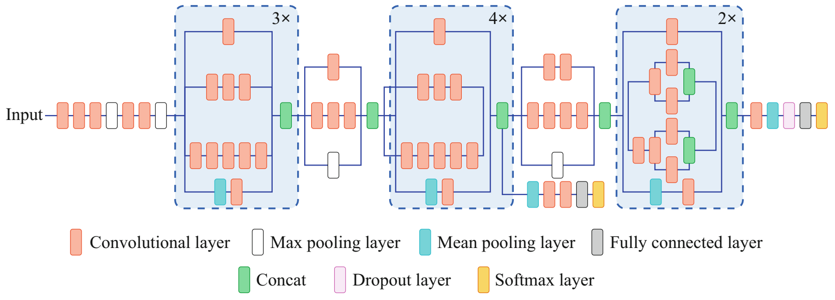


Fig. 7.6 Inception v3 architecture

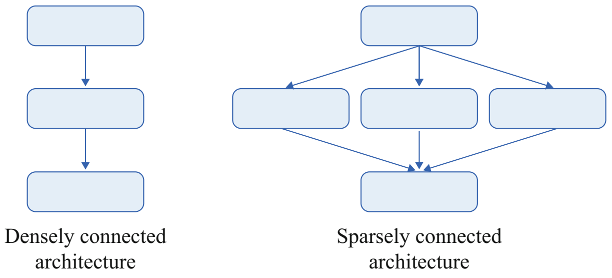

Densely connected architecture
Sparsely connected architecture

Fig. 7.7 Densely connected vs. sparsely connected network architectures

Inception aids in classifying items in computer vision, much like ImageNet can be seen as a database of categorized visual objects. Numerous applications have utilized the Inception v3 architecture, frequently using "pre-trained" data from ImageNet. Figure 7.6 shows the architecture of the Inception v3 model. The model's name is inspired by the popular internet meme, "*We need to go deeper,*" from Christopher Nolan's film Inception.

The effectiveness of a deep learning model can be improved simply by increasing the number of layers and/or neurons in each layer. However, creating more depth in the model frequently leads to complications—the model becomes more susceptible to overfitting as it gets larger, and training the larger model requires more computational resources. The low availability of training data also makes the model more susceptible to overfitting.

One way to address these issues is to switch from fully connected network structures to sparsely connected network architectures, particularly inside convolutional layers. Figure 7.7 visually compares densely connected versus sparsely connected network architectures. This is basically how the Inception model creates more depth and does not increase the number of parameters.

7.2.4 YOLO

YOLO (*You Only Look Once*) is a state-of-the-art *object detection algorithm* that can produce real-time results. Redmon et al. [3] introduced YOLO and brought a new dimension to the field of computer vision. Many improved versions of YOLO have been developed since then. Its name comes from the way YOLO works. Most object detection systems use a classifier model to recognize objects and then evaluate that insight on different locations of an image. Many systems, such as deformable parts models (DPM), use the sliding window technique for locating an object within an image. DPM systems slide a classifier along the entire image at evenly spaced locations. On the other hand, YOLO treats object detection as a single regression problem. The system looks at the whole image at a time and makes its predictions.

7.2.4.1 Features of YOLO

Some of the features that make YOLO superior to other object detection algorithms are briefly discussed below.

- **Processing Time:** YOLO has an extraordinarily fast processing time. It can process images at 45 FPS (frames per second) in real-time. Fast YOLO, a smaller version of the YOLO network, can process as many as 155 frames per second in real-time.
- **Fastest Object Detector:** YOLO is regarded as the fastest object detector and boasts the highest mean average precision (mAP) with less background error. The mAP is an evaluation metric to analyze the performance of object detectors. The mAP, weight (bits), and weight size (MB) of YOLO are compared to other fast object detectors on the *COCO dataset* in Table 7.2. The COCO dataset [4] is published by Microsoft for large-scale object detection, segmentation, and captioning.
- **Generalization Capability:** The better the generalization a model can make, the better its performance. In this regard, the latest YOLO versions have shown incredible generalization performance. YOLO can be applied to almost any domain for object detection due to its immense generalization capability.
- **Open-source System:** YOLO is an open-source system, and the online community has contributed to and developed many improved versions of it. This is

Table 7.2 Different algorithms compared with YOLO

Method	mAP (%)	W (bits)	Weight size (MB)
YOLOv3 [5]	54.64	32	237
YOLOv2 [6]	48.1	32	194
Zhang et al. [7]	48.1	16	97
YOLOv3-tiny [5]	33.1	32	33.79
Gaussian YOLOv3-tiny [8]	39.3	32	33.82
Nayak et al. [9]	28.09	8	30.48

also one of the critical reasons why YOLO is almost the first choice for object detection for many beginners.

7.2.4.2 YOLO Concepts

Some key concepts regarding YOLO are briefly discussed below:

Residual Blocks The image is divided into equal $N \times N$ grids. Each cell of the grid predicts individually if it contains any object. Therefore, each cell has to localize and predict the object class it contains. The cell containing the detected object's center becomes the center coordinate for the bounding box.

Bounding Box The system uses dimension clusters as anchor boxes to predict the bounding boxes for respective objects. It predicts four coordinates for each bounding box: t_x, t_y, t_w, t_h. The assumption is made that the cell is offset from the top left corner of the image by (c_x, c_y), and the prior width and height of the bounding box are p_w and p_h. Using these parameters, the system detects the bounding box for the object using the necessary calculations below.

$$b_x = \sigma(t_x) + c_x, \tag{7.1}$$

$$b_y = \sigma(t_y) + c_y, \tag{7.2}$$

$$b_w = p_w e^{t_w}, \tag{7.3}$$

$$b_h = p_h e^{t_h}, \tag{7.4}$$

where b_x is the x-coordinate of the center, b_y is the y-coordinate of the center, b_w is the width, b_h is the bounding box's height, and σ is the sigmoid function. The bounding box's width and height are predicted as offsets from cluster centroids. The box's center coordinates relative to the filter application's location are predicted using a sigmoid function. Figure 7.8 shows an example of the bounding box prediction by YOLOv3.

Anchor Box YOLO uses anchor boxes to localize different objects within an image. YOLO models start object detection by forming numerous bounding boxes throughout the image. These bounding boxes are known as anchor boxes. These anchor boxes are used as a baseline for bounding box prediction, and YOLO models predict a bounding box by adjusting the anchor boxes. For each anchor box, some offset from that box is predicted as a candidate box. The loss function is calculated from that candidate box based on ground truth. Then, the probability of an offset box overlapping with an object is calculated. By rewarding and penalizing the predicted boxes through object overlapping probability calculation described in the bounding box section and loss function optimization, YOLO models predict bounding boxes closest to the ground truths.

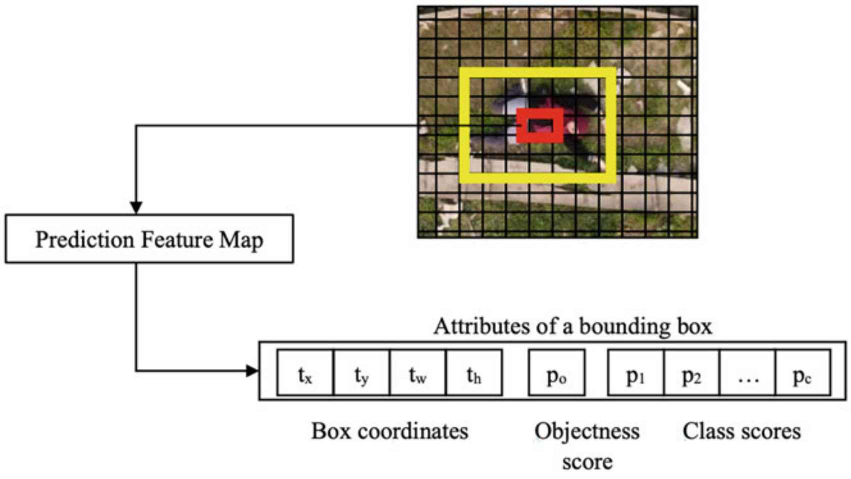


Fig. 7.8 Bounding box detection by YOLOv3

If you incompletely train a YOLO model, you will observe predicted bounding boxes all over the image. This happens due to the anchor boxes, as the model has yet to arrive at a bounding box decision from the anchor boxes.

Intersection Over Union (IoU) Intersection over union is defined as the ratio of the intersection of the ground truth box and predicted bounding box to the union of the ground truth box and predicted bounding box. It is expressed by Eq. 7.5:

$$IoU = \frac{Area\ of\ Intersection}{Area\ of\ Union}. \tag{7.5}$$

An object can have more than one boundary box predicted against it. However, not all boundary boxes will be relevant and accurate. Hence, any boundary box with an IoU score below a threshold is discarded.

Objectness Score Objectness score determines the probability of an object existing in a certain region of interest. A high objectness score for YOLO means an object is correctly detected and localized within an image, and the object is perfectly contained in a bounding box. The objectness score is calculated through this simple equation:

$$\sigma(t_o) = Pr(obj) \times IoU, \tag{7.6}$$

where $\sigma(t_o)$ is the objectness score, $Pr(obj)$ is the prediction score of an object, and IoU is the intersection over union score.

Non-Max Suppression (NMS) Often, an object may have more than one boundary box with an IoU score exceeding the threshold value. However, the goal is to have one boundary box per object. Therefore, non-max suppression is applied to keep the most accurate and relevant boundary box with the highest objectness score.

7.2.4.3 YOLO Variants

The first version of YOLO consists of 24 convolutional layers, four maxpooling layers, and two fully connected layers. Although superior to other fast object detectors, it has limitations in detecting smaller or unusually shaped objects.

The YOLOv2 [6] used Darknet-19 as its architecture, which uses batch normalization and improved the mAP by 2%. In addition, YOLOv2 incorporates the concept of anchor boxes within the convolution layers. Another improvement is that YOLOv2 uses higher resolution inputs of 448 $\times$ 448, improving the model performance to extract and learn finer details or features of different objects.

The YOLOv3 [5] uses Darknet-53 as its architecture. It is a 106 neural network consisting of upsampling networks and residual blocks. In addition, YOLOv3 uses a logistic regression model to calculate the objectness score for each bounding box.

The next version of YOLO, YOLOv4, [10] uses CSPDarknet-53 as its architecture, which is a network of 29 convolution layers and consists of 27.6 million parameters. YOLOv4 outperforms YOLOv3 by 10% in speed. Later, there were more modifications, and YOLOR, YOLOX, YOLOv5, YOLOv6, and YOLOv7 were developed.

7.2.4.4 YOLOv3 Implementation for Object Detection

An implementation of object detection has been shown in this section using the YOLOv3 variant. As discussed earlier in this section, YOLOv3 uses Darknet-53 as its architecture. For this purpose, we need to download, compile, and configure the Darknet architecture to train YOLOv3. The training should be continued for 3–4 hours. A custom dataset is used where only five classes are used.

YOLO is mostly implemented with the COCO dataset [4]. Even weights are available for the COCO dataset, which we can use to easily detect almost every daily life object. But to reduce the computational complexity, we will use a custom dataset. The dataset is created with the help of Open Images Dataset [11]. It is a publicly available dataset developed for object detection and image classification. It contains diverse images with annotated labels. The procedure involves using some shell commands that can be executed by a PowerShell, Command Prompt, or Shell, depending upon the operating system. The shell is a text-based program that helps users interact with the system. The process of creating a custom dataset is briefly described here:

1. The repository on OIDv4_ToolKit [12] was cloned.
```
git clone {h}ttps://github.com/theAIGuysCode/OIDv4_ToolKit.git
```
2. The directory was changed to the newly cloned repository.
```
cd OIDv4_ToolKit
```

3. All the required libraries were installed.
   ```
   pip install -r requirements.txt
   ```
4. The main.py file was executed with appropriate parameters.
   ```
   python main.py downloader --classes Person Laptop Mug Bicycle
   Handbag --type_csv train --limit 1000 --multiclasses 1
   ```
   Classes can be searched from Open Images Dataset website [11]. Classes with two words can be concatenated using an underscore. CSV type should be *train* in case of training or otherwise *test*. The limit sets how many images per class will be used for creating the dataset. The multiclasses parameter ensures all the images are stored in a folder.
5. Two CSV files were created before downloading the images.
6. For each class, the class names, i.e., the labels, were stored in a text file using an echo command.
   ```
   echo -e "Person\nLaptop\nMug\nBicycle\nHandbag" > classes.txt
   ```
7. A label file was created with each image.
   ```
   python convert_annotations.py
   ```

After successfully creating the dataset, an archived file was created for easy data transfer. The archive file is renamed to obj.zip in this case.

Programming Example 7.2
Listing 7.2 is an implementation of YOLOv3 for object detection, and Table 7.3 explains the code. This code is implemented using the darknet framework. It is a deep learning framework primarily designed for object detection.

The listing incorporates some shell commands. In a Python environment, the *subprocess* library can be used to run shell commands directly within a Python script. But using the IPython environment, the process can be a bit more simplified. For instance, ! does not mean anything in a Python environment. But in the IPython environment, it executes a shell command. Similarly, the % does not mean anything in Python, whereas it is defined as a magic command in the IPython environment. For these reasons, this script is executed in an IPython environment.

The code creates necessary configuration files and trains an object detection model on the custom-generated dataset. Although the code was written for the Google Colab notebook environment, it can also be executed in any other notebook environment.

```
# Framework: https://github.com/AlexeyAB/darknet

# -----------------Importing Necessary Framework-----------------
!git clone https://github.com/AlexeyAB/darknet


# -------------------Creating Location Shortcut------------------
from google.colab import drive
drive.mount('/content/gdrive/')
!ln -s /content/gdrive/My\ Drive/ /mydrive

```

```

# -----------------------Compiling Darknet-----------------------
%cd darknet
!sed -i 's/OPENCV=0/OPENCV=1/' Makefile
!sed -i 's/GPU=0/GPU=1/' Makefile
!sed -i 's/CUDNN=0/CUDNN=1/' Makefile
!make


# --------------------Creating Configuration---------------------
!cp cfg/yolov3.cfg cfg/yolov3_training.cfg
!sed -i 's/batch=1/batch=64/' cfg/yolov3_training.cfg
!sed -i 's/subdivisions=1/subdivisions=16/' cfg/yolov3_training.cfg
!sed -i 's/max_batches = 500200/max_batches = 10000/' cfg/yolov3_training.cfg
!sed -i 's/steps=400000,450000/steps=8000,9000/' cfg/yolov3_training.cfg
!sed -i '610 s@classes=80@classes=5@' cfg/yolov3_training.cfg
!sed -i '696 s@classes=80@classes=5@' cfg/yolov3_training.cfg
!sed -i '783 s@classes=80@classes=5@' cfg/yolov3_training.cfg
!sed -i '603 s@filters=255@filters=30@' cfg/yolov3_training.cfg
!sed -i '689 s@filters=255@filters=30@' cfg/yolov3_training.cfg
!sed -i '776 s@filters=255@filters=30@' cfg/yolov3_training.cfg


# ----------Create Folder on Google Drive to Save Data----------
!mkdir "/content/gdrive/MyDrive/yolov3"
!echo -e "Person\nLaptop\nMug\nBicycle\nHandbag" > data/obj.names
!echo -e 'classes= 5\ntrain  = data/train.txt\nvalid  = data/test.txt\nnames = data/obj.names\nbackup = /mydrive/yolov3' > data/obj.data


# ---------------------Copy and Unzip Dataset--------------------
!cp /mydrive/yolov3/obj.zip ../
!unzip ../obj.zip -d data/


# ----------------Generate Image and Label Paths----------------
!cp /mydrive/yolov3/generate_train.py ./
!python generate_train.py


# -------------------------Training Phase------------------------
!wget http://pjreddie.com/media/files/darknet53.conv.74
!./darknet detector train data/obj.data cfg/yolov3_custom.cfg darknet53.conv.74 -dont_show
# !./darknet detector train data/obj.data cfg/yolov3_training.cfg /mydrive/yolov3/yolov3_custom_last.weights -dont_show
```

Listing 7.2 Code for training YOLOv3 for object detection

Table 7.3 Explanation of the code for training YOLOv3 in Listing 7.2

Line number	Description
4	Cloning the darknet framework
8–10	Creating a shortcut for project folder, as it is necessary to work back and forth in the project folder and darknet folder
14–18	Editing the Makefile to enable GPU processing and compile darknet
22–24	Setting up batch number and subdivisions to 64 and 16 respectively
25	Max batch should be 20 times of the number of classes
26	Steps should be 80% and 90% of max batch
27–32	Using 5 classes in this case and filters should be 3 $\times$ (class number+5)
36	Creating a folder in the project directory
37–38	Creating obj.names and obj.data files as these are necessary for darknet to execute
37	obj.names contains the classes name or label names each in new line
38	obj.data contains some directory where other files will be stored
41	Before this step, transfer obj.zip to newly created folder in line 36
42–43	Transferring dataset to darknet directory
46–48	Generating image and labels path for darknet
52	Fetching a weight for faster execution
53	Training object detection using YOLOv3
54	In case of any failure between training process, this line can be used to resume training as line 53 stores trained weight files after each 100 steps to the backup folder mentioned in the obj.data file

The dataset was created earlier and renamed to obj.zip for convenience. The obj.zip file should be transferred to the preferred project folder location (in this case, to yolov3 folder) before running the code after line 41. The code in Listing 7.2 outputs a weight file in the project folder.

Listing 7.3 (explained in Table 7.4) utilizes the weights generated by Listing 7.2 and outputs an image file in the project folder. A sample input and output of this listing are shown in Fig. 7.9.

```
 1 # Framework: https://github.com/AlexeyAB/darknet
 2 
 3 # -----------------Importing Necessary Framework-----------------
 4 !git clone https://github.com/AlexeyAB/darknet
 5 
 6 
 7 # -------------------Creating Location Shortcut------------------
 8 from google.colab import drive
 9 drive.mount('/content/gdrive/')
10 !ln -s /content/gdrive/My\ Drive/ /mydrive
11 
12 
13 # -----------------------Compiling Darknet-----------------------
14 %cd darknet
```

```
!sed -i 's/OPENCV=0/OPENCV=1/' Makefile
!sed -i 's/GPU=0/GPU=1/' Makefile
!sed -i 's/CUDNN=0/CUDNN=1/' Makefile
!make


# --------------------Creating Configuration---------------------
!cp cfg/yolov3.cfg cfg/yolov3_training.cfg
!sed -i 's/max_batches = 500200/max_batches = 10000/' cfg/
    yolov3_training.cfg
!sed -i 's/steps=400000,450000/steps=8000,9000/' cfg/
    yolov3_training.cfg
!sed -i '610 s@classes=80@classes=5@' cfg/yolov3_training.cfg
!sed -i '696 s@classes=80@classes=5@' cfg/yolov3_training.cfg
!sed -i '783 s@classes=80@classes=5@' cfg/yolov3_training.cfg
!sed -i '603 s@filters=255@filters=30@' cfg/yolov3_training.cfg
!sed -i '689 s@filters=255@filters=30@' cfg/yolov3_training.cfg
!sed -i '776 s@filters=255@filters=30@' cfg/yolov3_training.cfg


# -----------Create Folder on Google Drive to Save Data----------
!mkdir "/content/gdrive/MyDrive/yolov3"
!echo -e "Person\nLaptop\nMug\nBicycle\nHandbag" > data/obj.names
!echo -e 'classes= 5\ntrain  = data/train.txt\nvalid  = data/test
    .txt\nnames = data/obj.names\nbackup = /mydrive/yolov3' >
    data/obj.data


# ------------------------Object Detection-----------------------
!./darknet detector test data/obj.data cfg/yolov3_training.cfg /
    mydrive/yolov3/yolov3_custom_final.weights /mydrive/yolov3/
    img9.jpg -thresh 0.3
!cp predictions.jpg /mydrive/yolov3/detection9.jpg
```

Listing 7.3 Code for object detection using YOLOv3

7.2.5 Facebook Prophet

Facebook Prophet is an algorithm for forecasting time series data. It is available as an open-source package in Python and R. Taylor and Letham [14], from the core data science team at Meta (formerly known as Facebook), introduced the Facebook Prophet algorithm in 2017. The official website for Facebook Prophet states comprehensive documentation on installing and using this package alongside different examples in different case scenarios.

This research and development aimed to provide the business industries with a potent and easily interpretable forecasting tool without understanding its depth. First, most of the existing forecasting tools are entirely automatic and leave very little space for flexibility to integrate different necessary information about the businesses. Next, most of the tools require experts with in-depth knowledge of

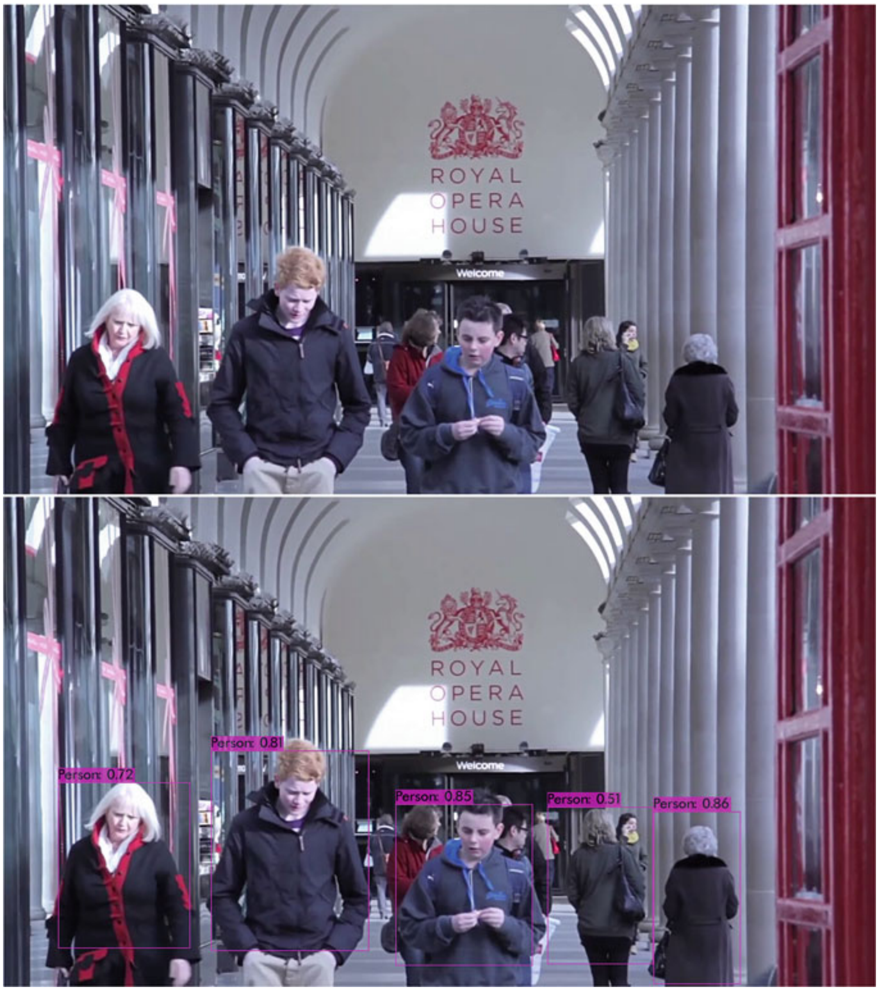


Fig. 7.9 Input (top) and output (bottom) of Listing 7.3. (Input snippet source: GitHub source [13])

the domain and the forecasting tool itself. Such requirement is demanding and expensive to meet.

7.2.5.1 Features of Facebook Prophet

The features of Facebook Prophet that make it a prominent and state-of-the-art forecasting tool are briefly discussed below.

- Facebook Prophet can work with linear and non-linear trends alongside flat trends. It allows users to select "changepoints" on the trend lines manually.
- Facebook Prophet can work with multiple seasonalities at once.

Table 7.4 Explanation of the Object Detection using YOLOv3 code in Listing 7.3

Line number	Description
4	Cloning the darknet framework
8–10	Creating a shortcut for project folder, as it is necessary to work back and forth in the project folder and darknet folder
14–18	Editing the Makefile to enable GPU processing and compile darknet
23	Max batch should be 20 times of the number of classes
24	Steps should be 80% and 90% of max batch
25–30	Using 5 classes in this case and filters should be 3 × (class number+5)
34	Creating a folder in the project directory
35–36	Creating obj.names and obj.data files as these are necessary for darknet to execute
35	obj.names contains the classes name or label names each in new line
36	obj.data contains some directory where other files will be stored
40	Execute detection using the weight file generated in Listing 7.3
41	Copy the predicted image to project directory

- It can also work with seasonalities with characteristics of irregular intervals, e.g., the FIFA World Cup.
- Facebook Prophet can capture trend changes in historical data due to a significant event, such as a product launch or hype of a product among specific populations.
- This algorithm does not require too much data preprocessing. It can handle missing data and also is unaffected by the presence of outliers.
- Facebook Prophet has intuitive parameters that are easier to tune by analysts with little to no expertise in forecasting tools.
- This algorithm can generate very accurate forecasts almost in real-time.

The Facebook Prophet forecasting tool uses a decomposable time series model, allowing users to interpret the model's decision effortlessly. The algorithm mainly uses an additive regression model with three main model components: trend, seasonality, and holidays.

7.2.6 ChatGPT

Chat Generative Pre-trained Transformer (ChatGPT) is a buzzword at present. The AI company OpenAI, headquartered in San Francisco, developed the large language model (LLM)-based chatbot named *ChatGPT* based on GPT-3.5 and GPT-4. The prototype was released on November 30, 2022 and is open to all. It resembles a chat interface, where we can input prompts, and ChatGPT provides responses in the context of those prompts. It can engage in conversational interactions and respond in a human-like manner in more than eighty languages, including Arabic, German, French, Spanish, Greek, Japanese, Chinese, Bengali, and Hindi.

LLMs can predict the next words in a train of words. So, ChatGPT uses the keywords and generates sentences by predicting words that are commonly used together. An additional layer, Reinforcement Learning with Human Feedback (RLHF), which is a combination of supervised and reinforcement learning techniques, was used to train ChatGPT to achieve the aptitude to obey instructions and provide responses acceptable to humans by using human feedback. Python, TensorFlow, and PyTorch were used to build the core structure of ChatGPT.

Massive volumes of data are used to train LLMs to precisely anticipate what word will appear next in a phrase. 300 billion words and 175 million parameters were used to train ChatGPT, which consists of 570 GB of text data from about 8 million web pages on the Internet, various articles, books, and online forum discussions (e.g., Reddit discussions). ChatGPT can remember the past dialogues in the conversation up to 3000 words and adapt its responses accordingly. But it does not remember past conversations.

In the next two sections, we will discuss some applications and limitations of ChatGPT.

7.2.6.1 Applications of ChatGPT

ChatGPT is a powerful language tool that emerged with boundless possibilities for applications. Despite its limitations, it is ubiquitously used in a wide range of applications, some of which are discussed below:

1. **Textual Applications:** ChatGPT is great for generating textual content, so much so that when ChatGPT was launched, writers felt threatened about their job security. ChatGPT can generate articles on various topics, translate text excerpts, summarize long texts, and even elaborate short prompts.
2. **Coding:** ChatGPT can be used for writing, editing, documenting, explaining, refactoring, and debugging codes. It can prove to be a valuable assistant to programmers by saving their time and also help novice programmers in coding.
3. **Education and Research:** ChatGPT provides very basic information in response to prompts. So it is a good source if someone is looking for very simplified explanations on specific topics. However, it is discouraged to trust ChatGPT on the information it provides because it can hallucinate false information. So, it is wise to double-check the information. It is also ethically wrong to plagiarize from ChatGPT for academic or research writings.
4. **Healthcare:** ChatGPT can help doctors and patients to access healthcare-related resources, such as diagnosis, medical advice, common prescriptions, etc. It can also mediate between doctors and patients in telemedicine services, booking appointments with the doctor and ensuring remote follow-up sessions.
5. **Businesses:** Businesses use ChatGPT for marketing content creation, advertising ideas, product descriptions, keyword search for search engine optimization, customer support, job descriptions, interview question preparation, creating emails, etc.

7.2.6.2 Limitations of ChatGPT

ChatGPT or other LLMs are still in their primitive phase and have some limitations. Some of the limitations of these remarkable human-like chatbots are briefly discussed below.

1. **Incorrect Information:** ChatGPT sometimes fails to offer correct answers; it hallucinates false information and confidently delivers it. When asked about the mammal to lay the largest eggs, it responded with "elephants." One reason is that the data used to train the ChatGPT was obtained from online discussion forums, some of which contained incorrect information. Although inaccurate information, the style of conveying the information is human-like. Hence, ChatGPT picked up on those patterns and sometimes responded with human-like inaccurate information. Besides, ChatGPT was built based on training data up to September 2021. So it has no knowledge of any event after that as it is not connected to the Internet.
2. **Question Phrasing:** The quality of the response frequently depends on the phrasing of the question. For example, ChatGPT may give one answer to one specific question and can give an entirely different answer if the question is rephrased differently. Sometimes it may even fail to recognize questions if phrased differently.
3. **Unresponsiveness:** ChatGPT is built to filter out and not respond to inappropriate and harmful questions and requests. Although it may be able to provide accurate responses accordingly, it has been programmed to avoid such conversations.
4. **Redundancy:** Although it has a human-like response, ChatGPT has been observed to repeatedly use the same specific phrases and answer patterns. This unnecessary overuse of phrases and patterns may have come from the biases and optimizations during the training phase.

7.3 AI/ML Security Challenges and Possible Solutions

AI and ML-related security threats mostly come from the cybersecurity side [15–17]. These security threats pose a potential challenge for different practical applications. A system with AI can be sabotaged by manipulating the AI through cyber-attacks in many ways. Therefore, proper protection against this issue has become crucial. The security challenges of AI can be summarized into three major categories, which are discussed below with their defense mechanisms:

1. **Adversarial Input Attack:** We already demonstrated the coding example of an adversarial input attack in Chap. 4. As displayed in Fig. 7.10, the adversarial input attack adds malicious imperceptible noise to the input to misclassify the image of a panda as a gibbon. Two common adversarial input attacks are projected gradient descent (PGD) [15] and fast gradient sign method (FGSM) [16]. A popular defense technique against adversarial input attacks is to train the target

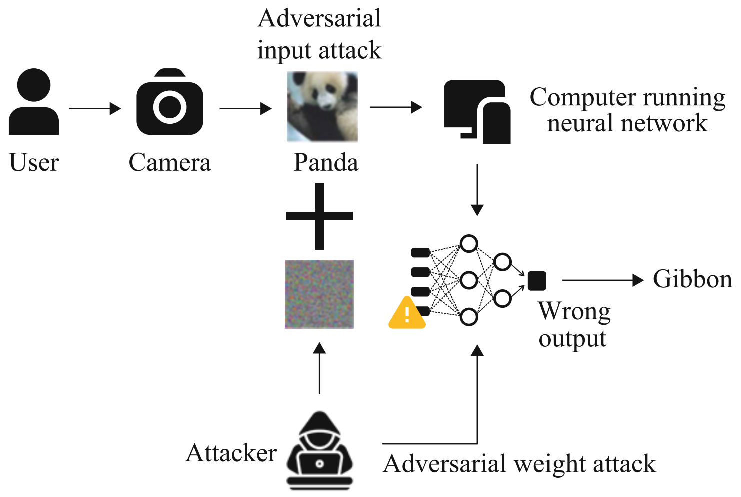


Fig. 7.10 Demonstration of adversarial input attack and weight attack

model with both adversarial and clean images, which is known as adversarial training [15].

2. **Adversarial Weight Attack:** Similar to an adversarial input attack, a popular weight attack tries to maliciously add noise to the weights of a neural network [17]. These attacks use popular fault injection techniques (such as the row-hammer [18]) to inject error into the main memory of a computer to objectively cause malfunction of the neural network running in that specific computer. For example, a popular bit-flip attack [18] on the main memory of a computer can cause a ResNet-18 [19] model to drop its accuracy to 0.1% from 69% on the ImageNet [20] dataset. Some common defenses against adversarial weight attacks are using binary weights [21] and error corrections [22].
3. **Trojan Attack:** The Trojan attack [23, 24] is a combination of both input and weight attacks. First, the attack injects malicious noise into the weights of a neural network during the training or test phase of a neural network. Later, the attacker adds a noise/sticker to the input to activate the Trojan, known as a trigger. Analysis of activation function [25] and output entropy analysis [26] are some of the existing methods to defend against these attacks.

7.4 AI/ML Hardware Challenges and Future Potential

After training a deep learning model, the real-time inference of that model often relies on hardware implementation. But there is a problem—the trained neural networks are often sized over several megabytes, e.g., Inception V3 and ResNet 50 are over 90 MB, AlexNet is over 200 MB, and VGG-16 is over 500 MB [27]. Low-power microcontrollers, smaller FPGAs, and edge devices cannot accommodate this memory requirement. Hence, running a large deep neural network on a mobile device presents a huge challenge as we need to shrink the model size by 33×. Thus, some of the critical hardware limitations of deploying AI models in smart mobile devices are:

1. Limited memory budget
2. Limited computational resources
3. Requirement for energy and power-efficient neural networks for hardware implementation

To mitigate hardware implementation-related issues, several approaches have been adopted in the literature to develop potential solutions. In the following sections, we will discuss two of them—quantization and weight pruning.

7.4.1 Quantization

In typical computer hardware, each deep learning model parameter (weights and biases) and activation is represented by a 32-bit or a 64-bit floating point representation. A 32-bit floating point precision means the values can take 2^{32} levels of values to represent any number. Thus, it is intuitive to quantize the weights into lower precision, such as an 8-bit integer, which helps reduce the model size by 4 times while decreasing the inference time significantly without sacrificing much of inference accuracy [28, 29]. This significant reduction in memory footprint also reduces power consumption, enabling low-power hardware applications. Quantization consists of two basic operations:

1. **Quantize:** Transform higher precision data into lower precision data.
2. **De-quantize:** Transform quantized lower precision data back into higher precision data.

As displayed in Fig. 7.11, the 32-bit weights are first converted into integer numbers representing the four levels of a 2-bit weight model. After that, the 2-bit weights can be mapped into the digital hardware using 00, 01, 10, and 11. The simple ways in which we can perform this quantization are discussed in the following sections.

7.4.1.1 Affine Quantization

If x is a floating point value in the range $[A, B]$, i.e., $x \in [A, B]$, then it can be mapped as $x_q \in [A_q, B_q]$ where $A_q = -2^{b-1}$ and $B_q = 2^{b-1} - 1$. Now, the affine quantization process can be defined as:

$$x_q = round(\frac{1}{s}x + z). \tag{7.7}$$

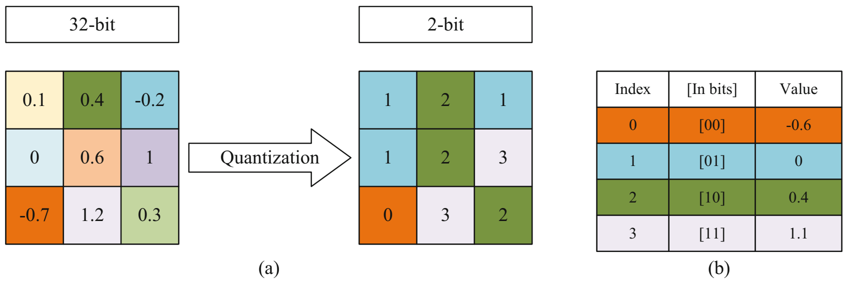


Fig. 7.11 Converting 32-bit precision weights into a 2-bit quantized representation of 00, 01, 10, and 11

And the de-quantization process will be:

$$x = s(x_q - z), \tag{7.8}$$

where s is scale and z is the zero point. s and z can be determined as follows:

$$s = \frac{B - A}{2^b - 1}; \tag{7.9}$$

$$z = -(round(A \times s)) - 2^{b-1}. \tag{7.10}$$

For 8-bit integer quantization, the values will become:

$$s = \frac{B - A}{255}; \tag{7.11}$$

$$z = -(round(A \times s)) - 128. \tag{7.12}$$

7.4.1.2 Scale Quantization

In scale quantization, the values are symmetric. The zero point z does not play any role as ($z = 0$), and thus it is absent from the equation. This quantization maps $x \in [-A, A]$ in such a way that $x_q \in [-A_q, A_q]$, where $A_q = 2^{b-1}$. Therefore, the scale quantization process is:

$$x_q = round\left(\frac{1}{s}x + 0\right) = round\left(\frac{1}{s}x\right), \tag{7.13}$$

where

$$s = \frac{A}{2^{b-1} - 1}. \tag{7.14}$$

For 8-bit integer quantization,

$$s = \frac{A}{255}. \tag{7.15}$$

In neural networks, the quantization process is applied in mainly two ways:

1. **Post-training Quantization:** In this method, the neural network is trained using normal floating point operation at first. After training the neural network, the trained parameters are then quantized. This quantized model is then deployed for performing inference. This is a simple and straightforward method, but very much susceptible to lower model accuracy as there is no way to compensate for quantization-related error because the quantization is applied after model training has commenced.
2. **Quantization-aware Training:** In quantization-aware training, quantization operations are emulated during forward propagation, but the backward propagation process is kept unchanged. By doing this, the quantization-related error accumulates in the total training loss while the optimizer function minimizes the error, hence adjusting the model parameter accordingly. This method reduces quantization-related error; thus, model accuracy reduction is compensated in most cases.

Our coding example will use scale quantization to convert a floating point number into a quantized level representation. Our quantization implementation technique will be quantization-aware training. Also, since quantization is a non-differentiable function, we will adopt the popular straight-through estimator [30] function to estimate the gradient of this quantization process during training.

7.4.2 Weight Pruning

Another popular approach to performing neural network compression is the pruning of weight connections. For example, it is empirically observed that if some of the weights below a certain threshold are totally cut off from the neural network (i.e., pruning or setting weights equal to zero), it does not hamper the performance of the neural network [28]. Similar to lasso regression, a weight penalty can be added to penalize the weights with larger weights. As a result, more weights in the neural network will have smaller values, which can be pruned to generate a compact network.

7.4.3 Implementation of Quantization and Pruning

Programming Example 7.3

We will once again work on the MNIST classification problem, where we demonstrate the efficacy of pruning and quantization in shrinking the model size of the CNN. During the training process, the weight pruning step is implemented. The percentage of weights pruned at each layer is monitored to ensure optimal performance. Listing 7.4 shows the Python code for this, and Table 7.5 explains the code.

```python
# -------------------------Torch Modules-------------------------
from __future__ import print_function
import numpy as np
import pandas as pd
import torch.nn as nn
import math
import torch.nn.functional as F
import torch
from torch.nn import init
import torch.optim as optim
from torchvision import datasets as ds
from torchvision import transforms as Trans
from torchvision import models
import torch.nn.functional as F
from torch.utils.data import DataLoader as DL
# ---------------------------Variables---------------------------
# for Normalization
mean = [0.5]
std = [0.5]

# batch size
bs =128 #Batch Size
Iterations = 20
learn_rate = 0.01

# compresion hyper-paramters
Quantized_bit = 8 # quantization bit
Lasso_penalty = 0.000001 # lasso penalty on weight
Thresholds = 0.005 # threshold


# -------Commands to download and prepare the MNIST dataset------

train_transform = Trans.Compose([
        Trans.ToTensor(),
        Trans.Normalize(mean, std)
        ])

test_transform = Trans.Compose([
        Trans.ToTensor(),
        Trans.Normalize(mean, std)
        ])


train_dataloader = DL(ds.MNIST('./mnist', train=True, download=
    True,
                       transform=train_transform),
                  batch_size=bs, shuffle=True)


test_dataloader = DL(ds.MNIST('./mnist', train=False,
                          transform=test_transform),
                 batch_size=bs, shuffle=False)


# -------------------------Defining CNN--------------------------
# Model Definition

#quantization function
class _Quantize(torch.autograd.Function):

    @staticmethod
    def forward(ctx, input, step):
        ctx.step = step.item()
        output = torch.round(input/ctx.step) ## quantized output
        return output

    @staticmethod
    def backward(ctx, grad_output):
        grad_input = grad_output.clone()/ctx.step ## Straight
    through estimator
        return grad_input, None

quantize1 = _Quantize.apply

class quantized_conv(nn.Conv2d):
    def __init__(self,nchin,nchout,kernel_size,stride,padding=0,
    bias=False):
        super().__init__(in_channels=nchin,out_channels=nchout,
    kernel_size=kernel_size, padding=padding, stride=stride, bias
    =False)
        "this function manually changes the original pytorch
    convolution function into a quantized weight convolution"

    def forward(self, input):
        self.N_bits = Quantized_bit - 1
        step = self.weight.abs().max()/((2**self.N_bits-1))

        QW = quantize1(self.weight, step)

        return F.conv2d(input, QW*step, self.bias,
                        self.stride, self.padding, self.dilation,
     self.groups)

```

```python

class quantized_linear(nn.Linear):
    def __init__(self, in_features, out_features, bias=True):
        super().__init__(in_features, out_features)
        "this function manually changes the original pytorch
    Linear function into a quantized weight with Linear value"

    def forward(self, input):

        self.N_bits = Quantized_bit - 1
        step = self.weight.abs().max()/((2**self.N_bits-1))

        QW = quantize1(self.weight, step)

        return F.linear(input, QW*step, self.bias)


class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = quantized_conv(1, 32, 3, 1)
        self.conv2 = quantized_conv(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = quantized_linear(9216, 128)
        self.fc2 = quantized_linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output


# defining CNN model
model = CNN()

## Loss function
criterion = torch.nn.CrossEntropyLoss() # pytorch's cross entropy
     loss function

# definin which paramters to train only the CNN model parameters
optimizer = torch.optim.SGD(model.parameters(),learn_rate)

```

```

## Lasso weight penalty
def lasso_p(var):
    return var.abs().sum()

# defining the training function
# Train baseline classifier on clean data
def train(model, optimizer,criterion,epoch):
    model.train() # setting up for training
    lasso_penalty = 0
    for id, (data, target) in enumerate(train_dataloader): # data contains the image and target contains the label = 0/1/2/3/4/5/6/7/8/9
        optimizer.zero_grad() # setting gradient to zero
        output = model(data) # forward
        loss = criterion(output, target) # loss computation

        ## iterating all the layers
        for name, module in model.named_modules():
            if isinstance(module, nn.Linear) or isinstance(module, nn.Conv2d):
                lasso_penalty += lasso_p(module.weight.data)  # penalty on the weight


        loss += lasso_penalty * Lasso_penalty
        loss.backward() # back propagation here pytorch will take care of it
        optimizer.step() # updating the weight values
        if id % 100 == 0:
            print('Epoch No: {} [ {:.0f}% ]   \tLoss: {:.6f}'.format(
                epoch, 100. * id / len(train_dataloader), loss.item()))



# to evaluate the model
## validation of test accuracy
def test(model, criterion, val_loader, epoch):
    model.eval()
    test_loss = 0
    correct = 0

    with torch.no_grad():
        for id, (data, target) in enumerate(val_loader):

            output = model(data)
            test_loss += criterion(output, target).item() # sum up batch loss
            pred = output.max(1, keepdim=True)[1] # get the index of the max log-probability
```

```python
            correct += pred.eq(target.view_as(pred)).sum().item() # if pred == target then correct +=1

    test_loss /= len(val_loader.dataset) # average test loss
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.4f}%)\n'.format(
        test_loss, correct, val_loader.sampler.__len__(),
        100. * correct / val_loader.sampler.__len__() ))


## training the CNN
for i in range(Iterations):
    train(model, optimizer,criterion,i)

    # pruning certain weights
    ## iterating all the layers
    layer_count = 1
    for name, module in model.named_modules():
        if isinstance(module, nn.Linear) or isinstance(module, nn.Conv2d):
            # pruning weights below a threshold = Thresholds * maximum weight value at that layer
                module.weight.data[module.weight.data.abs() < Thresholds* module.weight.data.abs().max()] = 0

                # percentage of weight pruned = no of weights equal to zero / total weights * 100
                weight_pruned =  module.weight.data.view(-1)[module.weight.data.view(-1) == 0].size()[0]/module.weight.data.view(-1).size()[0]*100
                print("Percentage of weights pruned at Layer " + str(layer_count) + ":\t" + str(weight_pruned) + "%")
                layer_count += 1

    test(model, criterion, test_dataloader, i) #Testing the the current CNN
```

Listing 7.4 Speech Recognition

Output of Listing 7.4:

```
Epoch No: 0 [ 0% ]     Loss: 2.312606
Epoch No: 0 [ 21% ]     Loss: 2.477860
Epoch No: 0 [ 43% ]     Loss: 2.347047
Epoch No: 0 [ 64% ]     Loss: 2.800500
Epoch No: 0 [ 85% ]     Loss: 3.244485
Percentage of weights pruned at Layer 1: 2.7777777777777777%
Percentage of weights pruned at Layer 2: 12.608506944444445%
Percentage of weights pruned at Layer 3: 71.74970838758681%
Percentage of weights pruned at Layer 4: 6.09375%

Test set: Average loss: 0.0030, Accuracy: 9000/10000 (90.0000%)

....................

Epoch No: 19 [ 0% ]     Loss: 0.043977
Epoch No: 19 [ 21% ]     Loss: 0.334064
Epoch No: 19 [ 43% ]     Loss: 0.487566
Epoch No: 19 [ 64% ]     Loss: 0.622565
Epoch No: 19 [ 85% ]     Loss: 0.755978
Percentage of weights pruned at Layer 1: 7.986111111111111%
Percentage of weights pruned at Layer 2: 33.75651041666667%
Percentage of weights pruned at Layer 3: 98.56906467013889%
Percentage of weights pruned at Layer 4: 14.21875%

Test set: Average loss: 0.0006, Accuracy: 9779/10000 (97.7900%)
```

Table 7.5 Explanation of the compression technique in Listing 7.4

Line number	Description
1–54	Same description as Listing 3.5
63–64	Applying quantization
69	Calculating the gradient during the backward path
74–87	Modified 8-bit quantized convolution operation
89–101	Modified 8-bit quantized linear operation
104–131	CNN model using quantization
133–192	Similar as before for training and test function
140–142	Lasso Penalty function
150	Applying the lasso penalty on the total loss function
196–205	Printing the percentage of weights pruned at each layer

Table 7.6 By applying pruning and quantization, we can shrink the CNN model in each of the four layers

	Before compression		After compression		
Layer	Pruning	Precision	Pruning	Precision	Compression ratio
1	0%	32-bit	10%	8-bit	(100/90)*(32/8) = 4.4x
2	0%	32-bit	31.5%	8-bit	(100/68.5)*(32/8) = 5.84x
3	0%	32-bit	98.4%	8-bit	(100/1.6)*(32/8) = 250x
4	0%	32-bit	14.7%	8-bit	(100/85.3)*(32/8) = 4.69x

Table 7.6 summarizes the code output. It represents the compression ratio of the pruning and quantization method. In particular, the third layer is compressed by 250 times, which would certainly fit into the on-chip memory of an FPGA device. Hence, compression techniques like pruning and quantization certainly can go a long way in overcoming the hardware limitations of deep learning deployment.

7.5 Multi-domain Learning

Deep learning (DL) models require separate datasets and task training routines. This requirement might be time and resource-consuming for some tasks at times. One way to tackle this issue is through multi-domain learning. Multi-domain learning is a technique where a single model can be adjusted and applied to minor to vastly different tasks at once. Multi-domain learning works by sharing a model's parameters among all domains while learning some domain-specific parameters for each individual case. There are several approaches to multi-domain learning. The following sections briefly discuss two major multi-domain learning approaches—transfer learning and domain adaptation.

7.5.1 Transfer Learning

We have briefly discussed transfer learning in Chap. 3. In this chapter, we will learn more about it through a programming example using Python.

Programming Example 7.4
This section presents an example of transfer learning using a pre-trained MNIST model. Our initial four-layer MNIST classifier has an accuracy of 98.92%. We took this model and fixed the first two convolution layer weights. We only train the last two linear layers to adapt to the new domain. After training the last two layers, the model adapts to a new dataset (e.g., Fashion-MNIST). To evaluate the efficacy of the transfer learning scheme, we first directly evaluated the MNIST model on the F-MNIST dataset and achieved only 8.25% accuracy. However, after fine-tuning the last two layers, we could achieve 87.93% accuracy on the F-MNIST dataset. We present the description of our code in Table 7.7 and the corresponding code in Listing 7.2.

```python
# -------------------------Torch Modules-------------------------
from __future__ import print_function
import numpy as np
import pandas as pd
import torch.nn as nn
import math, torch
import torch.nn.functional as F
from torch.optim import SGD
from torch.nn import init
import torch.optim as optim
from torchvision import datasets as ds
from torchvision import transforms as Trans
from torchvision import models
import torch.nn.functional as F
import torchvision.models as models
from torch.utils.data import DataLoader as DL
```

```
# ---------------------------Variables---------------------------
mean = [0.5] # for Normalization
std = [0.1]
# batch size
BATCH_SIZE =128
Iterations = 20
learn_rate = 0.01


# -------Commands to download and prepare the MNIST dataset------
train_transform = Trans.Compose([
        Trans.ToTensor(),
        Trans.Normalize(mean, std)
        ])

test_transform = Trans.Compose([
        Trans.ToTensor(),
        Trans.Normalize(mean, std)
        ])


train_dataloader = DL(ds.MNIST('./mnist', train=True, download=
    True,
                           transform=train_transform),
                  batch_size=BATCH_SIZE, shuffle=True)


test_dataloader = DL(ds.MNIST('./mnist', train=False,
                          transform=test_transform),
                 batch_size=BATCH_SIZE, shuffle=False)



# -------------------------Defining CNN--------------------------
# Pytorch official Example site: https://github.com/pytorch/
    examples/blob/master/mnist/main.py
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(9216, 128,bias=False)
        self.fc2 = nn.Linear(128, 10,bias=False)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
```

```python
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output


#defining CNN model
model = CNN()

## Loss function
loss_criterion = torch.nn.CrossEntropyLoss() # pytorch's cross entropy loss function

# definin which paramters to train only the CNN model parameters
optimizer = SGD(model.parameters(),learn_rate)

# defining the training function
# Train baseline classifier on clean data
def train(model, optimizer,train_dataloader,loss_criterion,epoch):
    model.train() # setting up for training
    for id, (data, target) in enumerate(train_dataloader): # data contains the image and target contains the label = 0/1/2/3/4/5/6/7/8/9
        optimizer.zero_grad() # setting gradient to zero
        output = model(data) # forward
        loss = loss_criterion(output, target) # loss computation
        loss.backward() # back propagation here pytorch will take care of it
        optimizer.step() # updating the weight values
        if id % 100 == 0:
            print('Epoch No: {} [ {:.0f}% ]   \tLoss: {:.6f}'.format(
                        epoch, 100. * id / len(train_dataloader), loss.item()))



# to evaluate the model
## validation of test accuracy
def test(model, loss_criterion, val_loader, epoch):
    model.eval()
    test_loss = 0
    correct = 0

    with torch.no_grad():
        for id, (data, target) in enumerate(val_loader):

            output = model(data)
            test_loss += loss_criterion(output, target).item() # sum up batch loss
            pred = output.max(1, keepdim=True)[1] # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item() # if pred == target then correct +=1

    test_loss /= len(val_loader.dataset) # average test loss
    print('\nTest set: Average loss: {:.4f},\n          Accuracy: {}/{} ({:.4f}%)\n'.format(
        test_loss, correct, val_loader.sampler.__len__(),
        100. * correct / val_loader.sampler.__len__() ))


## training the CNN
for i in range(Iterations):
    train(model, optimizer,train_dataloader,loss_criterion,i)
    test(model, loss_criterion, test_dataloader, i) #Testing the the current CNN

print("The model was initially trained for an MNIST dataset")
test(model, loss_criterion, test_dataloader, i)
print("The trained model has a good accuracy on MNIST")


# --------Transfer the Learning from One Domain to Another-------
# Downloading Fashion MNIST dataset
train_dataloader = DL(
        ds.FashionMNIST('./mnist', train=True, download=True,
                       transform=train_transform),
        batch_size=BATCH_SIZE, shuffle=True) # train dataset

test_dataloader = DL(
        ds.FashionMNIST('./mnist', train=False,
                         transform=test_transform),
        batch_size=BATCH_SIZE, shuffle=False) # test datase

## Testing shows the model fails on Fashion MNIST dataset
print("But fails on Fashion-Mnist dataset:")
test(model, loss_criterion, test_dataloader, i)


## No need to train the conv layer
for name,  module in model.named_modules():
    if isinstance(module, nn.Conv2d):
        module.weight.requires_grad = False
        module.bias.requires_grad = False

# Only train the last 2 fully connected layers
for name,  module in model.named_modules():
    if isinstance(module, nn.Linear):
        module.weight.requires_grad = True

```

```python
optimizer = torch.optim.SGD(model.parameters(),0.005) ##
    selecting a smaller learning rate for transfer learning

## training the CNN for F-MNIST
for i in range(10):
    train(model, optimizer,train_dataloader,loss_criterion,i)
    test(model, loss_criterion, test_dataloader, i) #Testing the
    the current CNN


print("After fine-tuning the model on the last layers the model
    recovers good accuracy on Fashion MNIST as well")
test(model, loss_criterion, test_dataloader, i)
```

Listing 7.5 Transfer Learning from a MNIST pre-trained model to a Fashion MNIST dataset

Output of Listing 7.5:

```
Epoch No: 0  [ 0% ]     Loss: 2.341227
Epoch No: 0  [ 21% ]     Loss: 0.674251
Epoch No: 0  [ 43% ]     Loss: 0.313292
Epoch No: 0  [ 64% ]     Loss: 0.324727
Epoch No: 0  [ 85% ]     Loss: 0.159849

Test set: Average loss: 0.0012,
          Accuracy: 9569/10000 (95.6900%)


... ... ... ... ...
... ... ... ... ...
... ... ... ... ...
... ... ... ... ...
... ... ... ... ...

Epoch No: 19 [ 0% ]     Loss: 0.017948
Epoch No: 19 [ 21% ]     Loss: 0.064786
Epoch No: 19 [ 43% ]     Loss: 0.068695
Epoch No: 19 [ 64% ]     Loss: 0.032498
Epoch No: 19 [ 85% ]     Loss: 0.041616

Test set: Average loss: 0.0002,
          Accuracy: 9892/10000 (98.9200%)

The model was initially trained for an MNIST dataset

Test set: Average loss: 0.0002,
          Accuracy: 9892/10000 (98.9200%)

But fails on Fashion-Mnist dataset:
```

```
Test set: Average loss: 0.0399,
          Accuracy: 825/10000 (8.2500%)
Epoch No: 0 [ 0% ]    Loss: 5.325634
Epoch No: 0 [ 21% ]    Loss: 0.943522
Epoch No: 0 [ 43% ]    Loss: 0.699631
Epoch No: 0 [ 64% ]    Loss: 0.664727
Epoch No: 0 [ 85% ]    Loss: 0.641860

Test set: Average loss: 0.0039,
          Accuracy: 8259/10000 (82.5900%)

... ... ... ... ...
... ... ... ... ...
... ... ... ... ...
... ... ... ... ...
... ... ... ... ...

Epoch No: 9 [ 0% ]    Loss: 0.425309
Epoch No: 9 [ 21% ]    Loss: 0.454595
Epoch No: 9 [ 43% ]    Loss: 0.333127
Epoch No: 9 [ 64% ]    Loss: 0.370811
Epoch No: 9 [ 85% ]    Loss: 0.331274

Test set: Average loss: 0.0026,
          Accuracy: 8793/10000 (87.9300%)

After fine-tuning the model on the last layers the
model recovers good accuracy on Fashion MNIST as well

Test set: Average loss: 0.0026,
          Accuracy: 8793/10000 (87.9300%)
```

The result shows a classifier that was trained to obtain 98.92% test accuracy on the MNIST dataset. Later, we changed the domain of handwritten digits (i.e., MNIST) to a new domain of the fashion dataset Fashion-MNIST. We replace the last layer and only train the last layer of the model to learn this new dataset. By only fine-tuning the last layer, we could achieve 87.93% test accuracy on the Fashion-MNIST dataset. In practice, even for complex tasks, it is possible to change the domain of knowledge from one dataset to a new dataset by only fine-tuning the last layer.

Table 7.7 Explanation of the transfer learning code in Listing 7.5

Line number	Description
2–14	Importing PyTorch modules
26–47	Data preparation for MNIST
50–78	CNN model Initialization
80–84	Setting up the training
86–120	Train and test functions
123–131	Training and testing the model on MNIST
134–143	Loading the second domain dataset Fashion MNIST (F-MNIST)
150–153	Setting the Convolution layer gradient to false (i.e., no training)
156–159	Only make the linear layers trainable
161–170	Retrain the last two layers on the new dataset Fashion-MNIST

7.5.2 Domain Adaptation

After training a neural network model on a certain dataset, a necessity arises to adopt the model to a newer dataset. However, the newer dataset almost always has a different distribution than the original dataset. This results in the underperformance of the deep learning model. Moreover, repeatedly training the model on newer datasets consumes much time and resources. These issues are tackled by implementing *domain adaptation* [31]. It is a modified version of transfer learning.

Instead of training the pre-trained model on the newer dataset, domain adaptation aims at utilizing learned knowledge of the pre-trained model by finding the meaningful correspondence between the source and target domain. Here, the source domain is the original dataset on which the model was trained, and the target domain is the newer dataset. Domain adaptation techniques can be unsupervised, supervised, and even semi-supervised. Domain adaptation has extensive use cases in various visual and audio applications [32].

7.6 Artificial Intelligence

Human beings perceive their surroundings and learn from their experiences. For instance, an infant does not instinctively know what an apple is and how it looks. This information about an apple (what it is, how it looks, what the varieties are, how it tastes) is something learned and stored in our infant memory. An apple is such a recognizable object that we do not even wonder how the information about an apple gets embedded into our memory. This learning process is so natural to humans that we can almost pick up on all objects we see and perceive our surroundings effortlessly. Another impressive fact about human behavior is that we learn how to respond to situations from our experiences. For example, our conversation style differs significantly based on formal or casual situations, the audience, etc.

Artificial intelligence (AI) is the field designated to build systems that act and think like humans and act and think rationally. AI systems are purposed to perform actions like humans, such as playing games, performing surgeries, performing diagnoses, driving a car, proving a theorem, and more. However, sometimes our actions are based on our conscience and tend to be more biased toward our humane side. Because human beings are sentimental creatures, prone to error, and cannot calculate every possible solution in their head very quickly, we may often make decisions that seem less rational than the other. With AI systems, they can think and act more rationally. The system can run all possible scenarios, calculate the success rate, and perform the best rational action.

AI has been used to develop technology that makes our lives easier and more efficient. Smartly designed AI systems will make smarter decisions and will be less prone to mistakes. Sometimes tasks that require sensitive calculations are better to be done by AI systems. The mundane and repetitive tasks can be all handed over to AI. Moreover, AI systems will always be faster than human beings. Although AI systems have a lot of potential for the betterment of mankind, it comes with its issues. Implementing AI systems regarding both the memory and processing units is expensive. Another drawback is that no matter how proficient AI can be, it will be a machine and needs more human-level creativity. Unlike AI, ML does not involve a machine that can replicate human intelligence. By recognizing patterns, ML seeks to train a machine how to carry out a certain task and produce reliable results.

7.6.1 The Turing Test

Now the question arises about the evaluation method and whether the built system emulates the human being. Alan Turing, a pioneer of modern computer science, proposed a test known as the *Turing test* to conclude if a built system has achieved the intelligence level of a human being. Figure 7.12 demonstrates the visualization of the Turing test.

The built system (Player A) and a human subject (Player B) are placed in a room, and another human being (the interrogator, Player C) is placed in another room, oblivious to the information regarding the players. The interrogator has to ask questions to players A and B. From the responses, player C has to guess which one

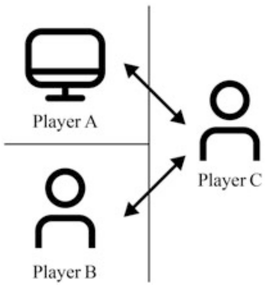


Fig. 7.12 Illustration of the Turing Test

of the players is a human and which one is a machine. If player C cannot distinguish between the human and the machine, it is concluded that the machine has achieved human intelligence.

7.6.2 Limitations of AI and Solutions

Recent advances in computing resources and efficient hardware platforms have made training and inference of AI effortless. AI is gradually stepping into the doorstep of our daily life. We are using AI in our day-to-day lives with or without being aware of it. Hence, the scientific community must raise their awareness in combating the associated limitations of the AI model. We have already discussed two major limitations of AI in previous sections. In Sect. 7.3, we discussed the emerging vulnerabilities of AI models against *adversarial attacks*. We have demonstrated how maliciously imperceptible noise can cause a target AI model to malfunction. Such a threat makes large-scale deployment of self-driving cars, medical robots, and other sophisticated applications challenging. A series of works [33, 34] have been conducted to counter this safety issue. Another major concern was the *hardware constraints* in deploying these powerful AI models in practice (discussed in Sect. 7.4). However, recent advances in AI accelerators [35] and GPU units [36] have made the path to the practical implementation of large-scale AI models much easier.

Nowadays, we often see in the news how AI predicts *racially biased results* due to biased data all over the Internet. The next major concern regarding widespread AI usage is the *lack of transparency* in AI models, their training data, and prediction outcomes. This will lead to biased and unfair predictions. The fairness problem of AI can be attributed to poor data collection quality. The sensitive features associated with input data force a model to learn racial representation and predict biased results. Recently, scientists have adopted several techniques such as reliable data collection, adversarial training, and post-processing of model and prediction to mitigate bias and improve fairness in AI models [37].

The emerging AI community treated advanced AI models as a complete black box without having knowing the underlying principles. Such a methodology has led to the emergence of adversarial attacks and fairness issues in these models despite performing exceptionally well with high accuracy. However, when AI becomes an integral part of our lives, we will have to understand the principles of these algorithms better, ensure the safety of the users, and provide a secure world controlled and powered by AI for our future generation.

7.6.3 Future Possibilities of AI

AI has the capability to diffuse into every aspect of our lives, including healthcare, education, banking, finance, transportation, military, warfare, industries, businesses, entertainment, and even household management. In fact, we cannot even imagine what AI can do within the next few years. However, as long as we keep educating ourselves on the developments of AI, we do not have to worry about AI taking over human jobs. AI is an industry itself that will require many people for its development. AI would be nothing without human intelligence. Therefore, when discussing the future possibilities of AI, we are also talking about the things for which we should be prepared to keep up with the rapid development of AI. Some noteworthy future possibilities of AI are discussed below:

1. **Automobile Industry**: Artificial intelligence is anticipated to significantly impact future car development. AI can potentially revolutionize the automobile sector in many ways, including increasing vehicle safety and enhancing driving efficiency. One of the contributions of AI will be autonomous driving vehicles, where every possible aspect of driving a vehicle will be controlled and decided by AI. The aspects may include driving, navigating the map, parking, and even managing traffic. The first and foremost concern for autonomous vehicles is the safety of passengers and pedestrians. AI systems might be used to calculate probable risks on the road beforehand by integrating sensors and cameras. In addition, AI can be used to create a network of cars—autonomous cars communicate among themselves and share necessary data. With AI in our cars, all we have to do is sit back and enjoy the ride.
2. **Smart Homes**: AI could revolutionize how we perceive and live at home. AI systems can control our homes' temperature, heating, cooling, and humidity levels. With the integration of an AI system, all levels will always be controlled and maintained at optimal levels. It can also help with electricity usage and avoid wasting energy. A smart fridge can help the owner with a menu based on the existing ingredients. It could also remind the owners to buy groceries. The whole cooking process can also be brought under AI systems, where the owner will not have to tenaciously control every step of cooking. The system can keep the food temperature at its optimum level for cooking and alert the owner when the food has been cooked. The AI system can also help keep the houses safe and sufficient, such as dealing with a power surge, gas leakage, plumbing issues, etc. AI has a lot to surprise us with our everyday lifestyles at home.
3. **Smart City**: AI has been used to design and build smart cities. The goal is to build sustainable infrastructure to collect information and data and analyze them to allocate and use resources efficiently. The residents of smart cities will have easy access to such technologies. For example, waste management would be handled proficiently and automatically by integrating AI systems and the Internet of Things (IoT). Sensors around the city will help collect and analyze the data to keep the city pollution-free. Smart cities will also contribute to keeping climate issues at bay.
4. **Finance**: A nation's economic and financial status is an unambiguous indication of its prosperity. Due to AI's impressive potential in practically every subject, it has the ability to significantly improve both the economic well-being of an individual and a nation. While determining the most effective method of managing finances, an AI system might consider many factors and be devoid

of human errors. With the integration of AI systems, equity funding, trading, and investments will be handled more efficacy.

The possibilities of AI are practically limitless, given the current pace of its research and development worldwide. The research domain of AI has taken over all branches of knowledge; it is no longer limited to the sciences. So, we need to constantly update ourselves with the growing pace of AI to keep abreast with modern technology. As AI systems become more intelligent, we should remain careful about the ethics, uses, and abuses of AI applications. We must always remember to use AI for the greater good of humankind.

7.7 Conclusion

ML and AI have come a long way and have recently begun to evolve beyond their primitive stage. AI systems are still somewhat dependent on humans, as they are susceptible to trivial errors that can be easily mitigated. ML and AI should be implemented as a complementary workforce of humans. ML and AI are quickly becoming inevitable in modern technologies, which will only continue to be more advanced and more accessible in the future. As the final chapter of this book, this chapter discusses state-of-the-art ML and AI technologies, such as graph neural network, EfficientNet, Inception network, YOLO algorithm, Facebook Prophet algorithm, and ChatGPT. Along with the potentials and challenges of emerging AI technologies, this chapter also hints to the readers about the future possibilities of ML and AI. With the information and insights gained from this chapter, readers can start their ML and AI research endeavors independently.

7.8 Key Messages

- State-of-the-art ML and AI techniques have improved and enhanced the lifestyles of mankind. In addition, these technologies have broadened the horizon to more potential innovations and inventions.
- AI systems have their security challenges. Hence, it should be used with caution. However, more research is being done to keep AI systems safe.
- As AI is becoming more advanced and the scope of AI applications is becoming broader, the hardware implementation of AI is becoming more and more crucial. The hardware implementation of AI and ML is a challenging task. Two of the possible solutions to this task, quantization and weight pruning, have been discussed in this chapter.
- AI has yet to unlock its full potential in the future. There are endless possibilities for how AI systems will be built, used, and applied.

7.9 Exercise

1. Many ML-related problems have been discussed throughout this book. Among these problems, which ones do you think can be solved using GNN? Discuss.
2. Take the Fashion MNIST dataset. Train three multiclass classification models using this dataset. Use vanilla CNN, Inception, and EfficientNet as your models' backbone. Now compare their performances.
3. Implement an object tracking model on the COCO dataset [4] using any YOLO variant. Compare this result with your result from the previous chapter (Chap. 6, Exercise 5).
4. Can Facebook Prophet be used for electrical load forecasting? Explain.
5. Train classification models on the Fashion-MNIST dataset with and without weight pruning and quantization. Point out the differences in performance.
6. Develop a traffic sign detection model using a state-of-the-art CNN architecture other than ResNet-18. Compare the result with the output from Listing 6.2. Did you manage to improve the performance?
7. What are some major security challenges regarding ML-based systems? Briefly explain.
8. Mention some hardware implementation-related issues of ML. How can we mitigate them?
9. Define quantization and classify them. How does quantization improve ML model performance?
10. Train a Fashion MNIST classifier using VGG-16 without utilizing saved weights. Then use pre-trained VGG-16 for the same problem (i.e., utilize saved weights). Comment on which method you would recommend and why.
11. Most of the basic ML and AI theories were established a long time ago. But it was not until recently that ML and AI saw these many practical uses, such as self-driving cars and advanced language models such as ChatGPT. This is mainly due to the fact that we have been able to manufacture and utilize high-performance hardware devices such as GPUs and TPUs in recent times. What do you think could happen to AI in 2040, and how would it affect the overall engineering sector? Discuss and provide insights.

References

1. Tom Cattini. (2018). Airplane in Mid Air Above Trees during Day, July 2018.
2. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2015). Rethinking the Inception Architecture for Computer Vision.
3. Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 779–788).
4. COCO dataset. https://cocodataset.org/#home
5. Redmon, J., & Farhadi, A. (2018). Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*

6. Redmon, J., & Farhadi, A. (2017). Yolo9000: Better, faster, stronger. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 7263–7271).
7. Zhang, S., Cao, J., Zhang, Q., Zhang, Q., Zhang, Y., & Wang, Y. (2020). An FPGA-based reconfigurable CNN accelerator for YOLO. In *2020 IEEE 3rd International Conference on Electronics Technology (ICET)*, May 2020. IEEE.
8. Choi, J., Chun, D., Lee, H.-J., & Kim, H. (2020). Uncertainty-based object detector for autonomous driving embedded platforms. In *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, Aug 2020. IEEE.
9. Nayak, P., Zhang, D., & Chai, S. (2019). Bit efficient quantization for deep neural networks.
10. Bochkovskiy, A., Wang, C.-Y., & Mark Liao, H.-Y. (2020). Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*.
11. Open Images V7—storage.googleapis.com. https://storage.googleapis.com/openimages/web/index.html [Accessed 26 Aug 2023].
12. The AI Guys. *OIDv4 Toolkit*. https://github.com/theAIGuysCode/OIDv4_ToolKit
13. FaceNet example video. https://github.com/timesler/facenet-pytorch/blob/master/examples/video.mp4
14. Taylor, S. J., & Letham, B. (2018). Forecasting at scale. *The American Statistician, 72*(1), 37–45.
15. Madry, A., Makelov, A., Schmidt, L., Tsipras, D., & Vladu, A. (2018). Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*.
16. Goodfellow, I. J., Shlens, J., & Szegedy, C. (2014). Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*.
17. Hong, S., Frigo, P., Kaya, Y., Giuffrida, C., & Tudor Dumitraş. (2019). Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In *28th {USENIX} Security Symposium ({USENIX} Security 19)* (pp. 497–514).
18. Yao, F., Rakin, A. S., & Fan, D. (2020). DeepHammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips. In *29th {USENIX} Security Symposium ({USENIX} Security 20)* (pp. 1463–1480).
19. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770–778).
20. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems, 25*, 1097–1105.
21. He, Z., Rakin, A. S., Li, J., Chakrabarti, C., & Fan, D. (2020). Defending and harnessing the bit-flip based adversarial weight attack. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 14095–14103).
22. Li, J., Rakin, A. S., Xiong, Y., Chang, L., He, Z., Fan, D., & Chakrabarti, C. (2020). Defending bit-flip attack through DNN weight reconstruction. In *2020 57th ACM/IEEE Design Automation Conference (DAC)* (pp. 1–6). IEEE.
23. Gu, T., Dolan-Gavitt, B., & Garg, S. (2017). BadNets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*.
24. Rakin, A. S., He, Z., & Fan, D. (2020). Tbt: Targeted neural network attack with bit Trojan. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 13198–13207).
25. Wang, B., Yao, Y., Shan, S., Li, H., Viswanath, B., Zheng, H., & Zhao, B. Y. (2019). Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *2019 IEEE Symposium on Security and Privacy (SP)* (pp. 707–723). IEEE.
26. Gao, Y., Xu, C., Wang, D., Chen, S., Ranasinghe, D. C., & Nepal, S. (2019). Strip: A defence against Trojan attacks on deep neural networks. In *Proceedings of the 35th Annual Computer Security Applications Conference* (pp. 113–125).
27. Keras Team. *Keras documentation: Keras Applications*. https://keras.io/api/applications/

28. Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M., Dally, B., et al. (2016). Deep compression and EIE: Efficient inference engine on compressed deep neural network. In *Hot Chips Symposium* (pp. 1–6).
29. Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M. W., & Keutzer, K. (2021). A Survey of Quantization Methods for Efficient Neural Network Inference. *arXiv preprint arXiv:2103.13630*.
30. Bengio, Y., Léonard, N., & Courville, A. (2013). Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*.
31. Ben-David, S., Blitzer, J., Crammer, K., & Pereira, F. (2006). Analysis of representations for domain adaptation. In B. Schölkopf, J. Platt, & T. Hoffman (Eds.), *Advances in Neural Information Processing Systems* (Vol. 19). MIT Press.
32. Farahani, A., Voghoei, S., Rasheed, K., & Arabnia, H. R. (2021). A brief review of domain adaptation. In R. Stahlbock, G. M. Weiss, M. Abou-Nasr, C.-Y. Yang, H. R. Arabnia, & L. Deligiannidis (Eds.), *Advances in Data Science and Information Engineering* (pp. 877–894). Cham: Springer International Publishing.
33. Allen, B. (2019). The role of the FDA in ensuring the safety and efficacy of artificial intelligence software and devices. *Journal of the American College of Radiology, 16*(2), 208–210.
34. Bostrom, N., & Yudkowsky, E. (2018). The ethics of artificial intelligence. In *Artificial Intelligence Safety and Security* (pp. 57–69). Chapman and Hall/CRC.
35. Yeoh, P. (2019). Artificial intelligence: Accelerator or panacea for financial crime? *Journal of Financial Crime, 26*, 634–646.
36. Jouppi, N., Young, C., Patil, N., & Patterson, D. (2018). Motivation for and evaluation of the first tensor processing unit. *IEEE Micro, 38*(3), 10–19.
37. Parikh, R. B., Teeple, S., & Navathe, A. S. (2019). Addressing bias in artificial intelligence in health care. *JAMA, 322*(24), 2377–2378.

Answer Keys to Chapter Exercises

Chapter 1

1. See Sect. 1.2.4
2. See Sect. 1.2.1
3. See Sect. 1.2.3.3
4. See Sect. 1.6.1
5. See Sect. 1.7
6. See Sects. 1.5.8.3, and 1.5.8.4
7. See Sect. 1.5.8.7. To use NumPy built-in functions:

```python
import numpy as np

marks = np.array([70, 67, 56, 90, 78, 68, 87, 89, 87,
         85, 86, 76, 75, 69, 74, 74, 84, 83, 77, 88])

mean = np.mean(marks)
max = np.max(marks)
min = np.min(marks)
std_dev = np.std(marks)

print("Mean = {}\nMax = {}\nMin = {}\nSTD = {}".format(mean,
    max, min, std_dev))

```

8. The output:

```
My favorite bands are:
Linkin Park
System Of A Down
Metallica
Evanescence
Poets of the Fall
Now the list of my favorite bands is:
Linkin Park
```

```
System Of A Down
Metallica
Imagine Dragons
Evanescence
Poets of the Fall
```

9. (a) $\begin{bmatrix} 46 & 73 \\ 102 & 191 \end{bmatrix}$

   (b) $\begin{bmatrix} 28 & 50 & 72 \\ 42 & 72 & 102 \\ 28 & 50 & 72 \end{bmatrix}$

   (c) $\begin{bmatrix} 42 & 42 \\ 54 & 60 \\ 42 & 42 \end{bmatrix}$

10. The ans:

```python
a = int(input("Dimension size: "))

matrix = []
result = [[0 for i in range(a)]
            for j in range(a)]

for i in range(a):
    arr = []
    for j in range(a):
        arr.append(int(input()))
    matrix.append(arr)

for i in range(a):
    for j in range(a):
        sum = 0
        for k in range(a):
            sum += matrix[i][k] * matrix[k][j]
        result[i][j] = sum

print(result)

```

11. See Answer for 10. Hint: Check column size for the first matrix and row size for the second matrix before multiplication
12. See Answer for 10
13. Skim through the code carefully

Chapter 2

1. See Sect. 2.10
2. See Sect. 2.2
3. See Sect. 2.3
4. See Sects. 2.5, 2.6, and 2.7
5. See Sects. 2.8 and 2.9
6. (a) MSE

$$\begin{aligned} MSE &= \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 \\ &= \frac{(1-0)^2 + (2-0.9)^2 + (3-1.4)^2 + (4-1.7)^2 + (5-7)^2}{5} \\ &= 2.812 \end{aligned}$$

(b) RMSE

$$\begin{aligned} RMSE &= \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2} \\ &= \sqrt{\frac{(1-0)^2 + (2-0.9)^2 + (3-1.4)^2 + (4-1.7)^2 + (5-7)^2}{5}} \\ &= 1.677 \end{aligned}$$

(c) MAE

$$\begin{aligned} MAE &= \frac{1}{n}\sum_{i=1}^{n}|y^{(i)} - \hat{y}^{(i)}| \\ &= \frac{|1-0| + |2-0.9| + |3-1.4| + |4-1.7| + |5-7|}{5} \\ &= 1.6 \end{aligned}$$

(d) MAPE

$$MAPE = \frac{100}{n}\sum_{i=1}^{n}\left|\frac{y^{(i)} - \hat{y}^{(i)}}{\hat{y}^{(i)}}\right|$$

$$= \frac{100}{5} \times \left(\left|\frac{1-0}{1}\right| + \left|\frac{2-0.9}{2}\right| + \left|\frac{3-1.4}{3}\right| + \left|\frac{4-1.7}{4}\right| + \left|\frac{5-7}{5}\right| \right)$$

$$= 45.167$$

7. Plug in the numbers in Eqs. 2.1 and 2.5
   Hint: MAE = 13.02, Huber Loss = 58.96
8. See Sect. 2.2.6
9. See Sect. 2.9
10. Take any two points. Calculate the Minkowski distance of these two points for h = 1 and h = 2 from Eq. 2.16. Then, calculate the Manhattan distance and Chebyshev distance of these two points
11. (a) Plug in the numbers in Eqs. 2.11, 2.14, 2.15, 2.13, and 2.16
    (b) See Listing 2.4
12. See Sect. 2.3.6
13. See Sects. 2.3.7, and 2.3.2
14. See Sect. 2.4. Plug in the respective values in Eqs. 2.23, 2.24, 2.25, and 2.26

Chapter 3

1. See Sect. 3.2.1
2. See Sect. 3.2.2.3
3. See Sect. 3.2.1.2
4. See Sect. 3.2.3.2
5. See Sect. 3.3
6. See Sect. 3.5
7. See Sect. 3.5.1
8. See Sect. 3.5.4
9. (a) See Sects. 3.4, 3.7, and 3.8
   (b) See Sects. 3.5.4 and 3.5.5
   (c) See Sect. 3.4
10. See Sect. 3.9
11. The problem can be regarded as a time series prediction problem. Therefore, study Sect. 3.6, follow the instructions stated in the problem, and provide your own solution
12. You may get help from Listing 3.2
13. See Sect. 3.4.2, Listing 3.4, Listing 3.5, and follow problem instructions
14. Follow Listing 3.10 and provide your own solution
15. Follow Listing 3.21 and provide your own solution
16. Follow Example 3.4

Chapter 4

1. See Sect. 4.2
2. See Sect. 2
3. See Sect. 4.5
4. See Sect. 4.7
5. See Sect. 4.9
6. Try modifying the model by adding or removing layers and observing the results
7. See Listing 4.2
8. See Sect. 4.4
9. See Sect. 4.5
10. See Section See Listing 4.6 and follow the instructions stated in the problem
11. See Listings 4.6 and 4.7

Chapter 5

1. (a) See Sect. 5.2
   (b) See Sect. 5.3
   (c) See Sect. 5.5
   (d) See Sect. 5.5
2. Share your own idea based on this chapter
3. See Sect. 5.2
4. See Sect. 5.2 and Listing 5.1
5. See Sect. 5.5
6. Follow listing 5.7
7. Follow Sect. 5.3.2 and provide your own solution
8. Besides this chapter, you might want to study further in this topic from other sources. Use your imagination, too

Chapter 6

1. See Sect. 6.1
2. See Sect. 6.1
3. See Listing 6.1
4. See Sect. 6.2.2.2
5. See Listing 6.1. Instead of the dataset used in the example, use the COCO dataset
6. See Listing 6.3. Instead of using the equation of a straight line, use the equations for hyperbola, parabola, and half circle

Chapter 7

1. See Sect. 7.2.1
2. See Sects. 7.2.2, 7.2.3, and use your previous CNN knowledge. Provide your own solution
3. See Listing 7.2 and follow the instructions stated in the problem
4. See Sect. 7.2.5 and draw your conclusion based on its features
5. You can take help from Listing 7.4
6. Utilize your knowledge from Sect. 7.2 and provide your own solution
7. See Sect. 7.3
8. See Sect. 7.4
9. See Sect. 7.4.1
10. Import the VGG-16 model just like any of the other state-of-the-art CNN architecture (e.g., Efficientnet, Inception, etc.). Then, follow the instructions stated in the problem
11. Read the whole chapter and try to comprehend the trend. 2040 is not so far as it seems

Index