

Practical Mathematics

for

AI and Deep Learning

A Concise yet In-Depth Guide on Fundamentals of Computer Vision,
NLP, Complex Deep Neural Networks and Machine Learning



Tamoghna Ghosh

Shravan Kumar Belagal Math

bpb

Practical Mathematics

— for —

AI and Deep Learning

A Concise yet In-Depth Guide on Fundamentals of Computer Vision,
NLP, Complex Deep Neural Networks and Machine Learning



Tamoghna Ghosh
Shravan Kumar Belagal Math



Practical Mathematics for AI and Deep Learning

*A Concise yet In-Depth Guide on
Fundamentals
of Computer Vision, NLP, Complex Deep
Neural
Networks and Machine Learning*

**Tamoghna Ghosh
Shravan Kumar Belagal Math**



www.bpbonline.com

FIRST EDITION 2023

Copyright © BPB Publications, India

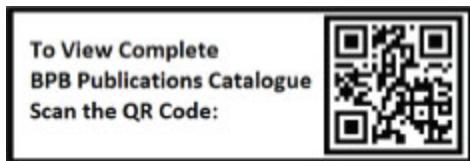
ISBN: 978-93-5551-194-2

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.



www.bpbonline.com

Dedicated to

From Tamoghna

My grandfather, late Mr. Dukari Roychoudhury who introduced me to a great Mathematics teacher and always guided me with his wise words.

My beloved grandmother, late Mrs. Renuka Roychoudhury.

From Shravan

Parents Basavaraj & Sulakshana, who supported me in all phases of life and continue to do so.

Brother Chethan, whose thinking always is in sync with mine, rarely we disagree on any topic.

Charming spouse Sudha, who has always encouraged me to achieve greater heights.

Adorable kids Anvika & Anirudh, whose presence keeps us cheerful.

About the Authors

Tamoghna is an AI Software Solutions Engineer in Client Computing Group at Intel and has 15 years of work experience. He has a master's in computer science from Indian Statistical Institute and a master's in mathematics from Calcutta University. He has 4 US patents, 3 IEEE papers and has also authored book on Transfer learning.

Shravan is currently an AI Engineer at Intel's Client Computing Group with 11 years of working experience. He had Master of Engineering degree from Indian Institute of Science, Computer Science and Automation department. He has been granted with 4 US patents. His interest lies in application of AI algorithms to solve real world problems.

About the Reviewer

Koushik Bhattacharyya is an accomplished Software Professional, who after completing M.Sc Pure Mathematics from Burdwan University and M.Tech in Computer Science (Gold Medalist) from Indian Statistical Institute, Kolkata, worked for technology giants like NVIDIA, AMD, Toshiba and Intel. He has more than 18 years of experience in software development with Architectures and Lead roles in diverse domains and technologies, including Medical Image Processing, Computer Vision, Machine Learning, Deep Learning, GPGPU and more. Koushik has also authored a book viz. OpenCL Programming by Example. His present interests include AR/VR and Blockchain.

Acknowledgements

There are a few people we want to thank for the continued and ongoing support they have given us during the writing of this book. We would like to thank our managers for continuously encouraging us for writing the book. Also, without the support of our family members could have never completed this book.

We are grateful to the technical reviewer of the book Mr. Koushik Bhattacharyya who is a AI Software Solutions Engineer in Intel Habana Labs division. His diligent reviews helped us correct few unintended mistakes. He also suggested some subtopics to be rephrased for better clarity.

Our gratitude also goes to the team at BPB Publication (including Surbhi, Shali, Lubna) for being supportive enough to provide us quite a long time to finish the first part of the book and also allow us to make some late changes to the content of the book.

Preface

The goal of *Artificial Intelligence* is to design algorithms that can perform: “data based automated decision-making under uncertainty”. To understand the theory of automated decision-making, a descent knowledge in the following mathematical concepts is essential: (1) Linear algebra (2) Vector calculus (3) Probability (4) Statistics. This book covers in depth these fundamental mathematical concepts. Each of these have a very vast literature of its own. So, we always wonder where to start and how far to go. In this book we have tried to put together the most essential topics from all these four areas of Mathematics. We have avoided detailed proofs wherever possible and tried to explain more intuitively these concepts. As new advancements are being made almost every day in the field of AI, it’s hard to keep oneself updated by constant study of latest research publications. However, with a strong mathematical foundation provided by this book, the learning curve will appear much less steep.

This book takes a practical approach for introducing the mathematical theory. It provides code or pseudocode in python for most of the mathematical concepts discussed, enabling the readers to use these concepts in their projects wherever applicable. For example, computation of gradient of a function of several variables is introduced mathematically and then corresponding code is also given both in naive python, numpy and tensorflow to clarify the concepts. This book also covers the application of the mathematical theory in building various AI algorithms. Also, this book discusses about a majority of popular neural network architectures. The readers should be able to reuse these building blocks for custom neural network architecture engineering.

This book is divided into twelve chapters. The first six chapters are theory oriented, and we strongly suggest the readers to read them in order as there are many interdependencies in these chapters. The remaining chapters are applications of these concepts and hence can be read in any order.

Chapter 1 Overview of AI: Chapter provides a high-level overview of Artificial Intelligence and its subcomponents. The common terminologies like model, data, parameters of models, dependent and independent

variables and model evaluation metrics will be explained in this chapter and will be referenced repeatedly in later chapters.

Chapter 2 Linear Algebra: Covers most topics of Linear Algebra with examples that finds its application in AI. Well thought figures in the chapter helps reader to understand the concept with clarity. This chapter will discuss about representing the real-world data in numeric form called vectors and introduce the required mathematical tools to process vectors.

Chapter 3 Vector Calculus: Chapter discuss differentiation and integration of vectors. The concept of tensors is also introduced in this chapter along with basic tensor algebra and tensor calculus. Moreover, this chapter provides basic optimization topics for function of several variables and functions over tensor.

Chapter 4 Basic Statistics and Probability Theory: This chapter covers introductory concepts of statistics like collecting, organizing, analyzing of data for the purpose of effective decision-making. Real world data has various sources of uncertainty. To quantify this uncertainty in data, probability theory is introduced.

Chapter 5 Statistical Inference and Applications: Statistical inference covers the techniques of decision making under uncertainty. In machine learning uncertainty can arise from noisy data, incomplete information about the problem domain etc. This chapter covers the core concepts of statistical inference and its application to linear models in ML like linear regression, curvilinear regression, and logistic regression.

Chapter 6 Neural Networks: Most of latest the AI algorithms are based on neural networks. This chapter introduces neural networks in general. Also, the fundamental back propagation algorithm is explained in details including the application of tensor calculus to compute layer wise derivatives if the network.

Chapter 7 Clustering: In few domains, data will be unlabeled. In these scenarios, task would be to find natural groups among data samples. Each identified group has unique characteristics which are learnt by algorithms. Learning will help in assigning new data samples to the existing groups based on its characteristics. This chapter will discuss about these algorithms that identifies natural groups.

Chapter 8 Dimensionality Reduction: In most cases, real-world data sample is of more than three dimensions. Higher dimensional data will result in data sparsity which in turn decreases accuracy of learning algorithms. Also, visualization of data whose dimensions are greater than three is not possible. This chapter will discuss algorithms that would be used in reducing dimensions of the data.

Chapter 9 Computer Vision: This chapter provides some theoretical background for the state-of-the-art AI models in computer vision. A specialized neural network architecture called convolution neural network or CNNs used of such models, is explained in details. Variations of the CNN architectures are used for different types of vision tasks. The motivation behind these architectures and how to train these networks is covered and references are provided for the model and code of these architectures.

Chapter 10 Sequence Learning Models: In few domains, data is sequential. Audio clips, video clips, time-series data are few examples of sequential data. Here, prediction of the future output will depend on previous data history. This chapter will discuss about algorithms which would help in learning and predicting based on sequential ordered data.

Chapter 11 Natural Language Processing: Natural Language has been important communication tool among humans and has grown in complexity which our brain can comprehend. This chapter will discuss about algorithms that would learn to understand natural language, represent natural language in concise human readable form.

Chapter 12 Generative Models: Generative modeling is a branch of AI which involves automatically discovering and learning the regularities or patterns in input data in such a way that the model can be used to generate new examples that plausibly could have been drawn from the original dataset. This chapter covers various generative modelling techniques like variational autoencoders, different types of generative adversarial nets (GAN).

Code Bundle and Coloured Images

Please follow the link to download the *Code Bundle* and the *Coloured Images* of the book:

<https://rebrand.ly/5m25ppc>

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/Practical-Mathematics-for-AI-and-Deep-Learning>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at <https://github.com/bpbpublications>. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book

customer, you are entitled to a discount on the eBook copy. Get in touch with us at: business@bpbonline.com for more details.

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

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit www.bpbonline.com. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

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

For more information about BPB, please visit www.bpbonline.com.

Table of Contents

1. Overview of AI

Structure

Objectives

AI systems

Machine Learning

How are ML Models created?

Data types

Learning From data

Types of ML algorithm

Unsupervised learning

Reinforcement learning

Supervised learning

Metrics for evaluating classification model

Metrics for evaluating regression model

Deep learning

Dataset preparation

Application of AI

Role of Mathematics in AI

Conclusion

2. Linear Algebra

Structure

Objectives

Linear equations

Solving system of equations analytically

Infinitely many solutions

Inconsistent system

Introducing matrix

Augmented matrix

Pseudocode forward substitution

Pseudocode back substitution

Basic matrix operations

Euclidean space

Vectors and basic properties

Representing vector

Norm

Direction

Scalar multiplication

Addition/subtraction of vectors

Distance between vectors

Dot product and orthogonality

Linear Combination of Vectors

Dimension and basis of the space

Orthogonal and orthonormal basis

Natural orthonormal basis of \mathbb{R}^n

Subspaces

Dimension of subspace

Hyperplanes and Halfspaces

Defining vector space

Vector spaces

Normed vector space

Norm of real numbers

l_p Norm

Maximum norm

Matrix norm

Inner product

Application on real dataset

K-nearest neighbor

Representing vectors in matrix

Matrix rank

Matrices types

Identity matrix

Symmetric matrix

Skew symmetric matrix

Invertible matrices

Properties of Matrix Inverse

Permutation matrix

Orthogonal matrix

Matrices in ML problem formulation

Feature/data matrix

One hot encoding

Distance matrix

Gram matrix

Covariance matrix

Correlation matrix

Jacobian and Hessian matrix

Subspaces of matrix and orthogonality

Null space

Orthogonality among subspaces

Determinant

Inverse of Matrix

Orthonormalization

Applications of Orthonormalization

Linear transformation

Matrix associated with linear map

Composition of linear transformation

Eigenvalues and vectors

Eigen properties

Geometric analysis

Existence of zero eigenvalue

Eigen properties of symmetric matrices

Positive definite

Matrix decomposition

LU decomposition

By-product of Gauss-Jordan elimination

QR decomposition

Eigen decomposition

Real symmetric matrix

Singular value decomposition

Conclusion

Points to remember

Further Reading

3. Vector Calculus

Structure

Objectives

Analysis of real functions

Limit of a function

Continuous functions

Derivative of a function

Higher Order derivatives

Taylor series expansion

Scalar and vector fields

Limits and continuity

Derivative of scalar fields w.r.t. vector

Directional derivative and partial derivatives

Total derivative

Geometry of gradient vector

Derivative of vector fields w.r.t. vector

Chain rule for derivatives of vector fields

Matrix form of the chain rule

Tensors

Einstein notation

Dot product of tensors

Tensor calculus

Total derivative of tensor

Mathematical optimization

Maxima, minima, and saddle point

Descent methods

Function optimization with constraints: Lagrange multipliers

Optimization with inequality constraints

The Lagrange dual function

Convex functions

Properties of convex functions

Convex optimization

Karush-Kuhn-Tucker conditions (KKT)

Conclusion

Points to remember

Further readings

4. Basic Statistics and Probability Theory

Structure

Objectives

Basic statistics

Measures of central tendency

Mean

Median

Mode

Partition Values

Measures of dispersion

Range

Interquartile Range

Mean deviation

Standard deviation

Coefficients of dispersion

Moments

Skewness and kurtosis

Correlation

Probability and odds

Random experiment

Events as sets

Conditional probability

Independent Events

Conditional independence

Total probability theorem

Bayes theorem

Bayesian Decision Theory

Random variable

Discrete probability distributions

Bernoulli and categorical distribution

Binomial distribution

Poisson distribution

Continuous probability distributions

Cumulative Probability Distribution Function (C.D.F)

Uniform distribution

Gaussian distribution or normal distribution

Exponential Distribution

Mathematical expectation of a random variable

Joint Probability Distributions

Transformation of a random variable

Multivariate distributions

Multinomial distribution

Multivariate gaussian distribution

Information theory

Entropy

Relative entropy or KL divergence

Mutual information

Decision tree

Conclusion

Points to remember

Further reading

5. Statistical Inference and Applications

Structure

Objectives

Large Sample Theory

Sample statistics

Sampling from known distributions

Hypothesis testing

Statistical inference

Estimator properties

Minimum Variance Unbiased (M.V.U) estimators

Likelihood function

Cramer-Rao inequality

Method of Maximum Likelihood Estimation (MLE)

Bias-variance decomposition of estimator

Applications – Formulating ML problems as statistical inferencing

Data distribution

Classification

Naive Bayes classifier

Regression

Linear and curvilinear regression

Estimating model parameters

Iterative estimation of model parameters

Overfitting and underfitting

Bias variance trade-off

Logistic Regression

[Multiclass logistic regression](#)
[Poisson regression](#)
[Interpretability of linear models](#)

[Conclusion](#)
[Points to remember](#)
[Further Reading](#)

6. Neural Networks

[Structure](#)
[Objectives](#)
[Artificial neuron: An adaptive basis function](#)
[Feed Forward neural network](#)
[Training neural network](#)
[Stochastic Gradient Descent](#)
[Computing error derivatives](#)
[Backpropagation algorithm](#)
[Challenges of training neural networks](#)
[Modifications of SGD](#)
[Momentum methods](#)
[Adaptive learning rate](#)
[Bias-variance trade-off in neural networks](#)
[Regularization of neural nets](#)
[Sensitivity of neural networks to small perturbations](#)
[Neural Network Architectures](#)
[Conclusion](#)
[Points to remember](#)
[Further Reading](#)

7. Clustering

[Structure](#)
[Objectives](#)
[Forming clusters](#)
[Distance and similarity](#)
[Cluster quality](#)
[Internal evaluation](#)
[Davies-Bouldin indicator](#)
[Dunn indicator](#)

[Silhouette coefficient](#)

[External evaluation](#)

[Rand index](#)

[F-measure](#)

[Fowlkes–Mallows index](#)

[Jaccard index](#)

[Clustering algorithms](#)

[Partition-based clustering](#)

[K-means](#)

[K-medoids](#)

[Density-based clustering](#)

[DBSCAN](#)

[Distribution-based clustering](#)

[Gaussian Mixture Model](#)

[Hierarchical-based clustering](#)

[Agglomerative clustering](#)

[Distance between clusters](#)

[BIRCH](#)

[Graph-based clustering](#)

[Fuzzy theory-based clustering](#)

[Fuzzy c-means](#)

[Conclusion](#)

[References](#)

[8. Dimensionality Reduction](#)

[Structure](#)

[Objectives](#)

[Reducing dimensionality](#)

[Principal Component Analysis](#)

[Loading Iris dataset](#)

[Calculating covariance matrix](#)

[Decomposition of covariance matrix](#)

[Reducing with principal components](#)

[Variance retention](#)

[When to use PCA](#)

[Autoencoder](#)

[Iris autoencoder](#)

[t-SNE](#)
[Choosing \$\sigma_i\$](#)
[PCA vs t-SNE](#)
[t-SNE on Iris Dataset](#)
[Conclusion](#)
[Further reading](#)
[References](#)

[9. Computer Vision](#)

[Structure](#)
[Objectives](#)
[Digital Image Formation](#)
 [Capture the light](#)
 [Sampling and quantization](#)
[Pixels](#)
 [Accessing pixels](#)
[Spatial filtering](#)
 [Geometric spatial transformation](#)
 [Neighbor pixel operation](#)
 [Convolution properties](#)
 [Separable kernels](#)
 [Convolution with separable kernels](#)
 [Gaussian kernel](#)
 [Discrete approximation of Gaussian function](#)
 [Application of Gaussian filter](#)
 [Image derivative-based kernels](#)
 [Laplacian kernel – Second order derivative](#)
 [Sobel kernel: First order derivative](#)
 [Non-linear filters](#)
[Learning filters](#)
[Convolution Neural Networks](#)
 [Convolution layer](#)
 [Pooling layer](#)
 [Spatially separable convolution](#)
 [Depthwise separable convolution](#)
 [Depthwise convolution](#)
 [Pointwise convolution](#)

Optimization
Upsampling: Transposed convolution
Development of CNN
AlexNet
TensorFlow Model
Counting trainable parameters
Inception
VGG
ResNet
Xception
Application of CNN models
Image classification
Object detection
R-CNN – Regions with CNN features
YOLO – You Only Look Once
Image segmentation
U-Net
Summary
Further reading
Points to remember
References

10. Sequence Learning Models

Structure
Objectives
Time series models
Decomposition of time series
Differencing
Time series forecasting
OLS model
Exponential smoothing
Autoregressive Integrated Moving Average
Probabilistic sequence models
Markov chain
Hidden Markov model
Recurrent neural networks
Training RNN

Long Short-Term Memory (LSTM)
Gated Recurrent Unit (GRU)
Stacked LSTM/RNN

Generative models for sequence

Handwriting generation
Mixture Density Network

Sequence classification

Bi-directional RNN

Sequence to Sequence

Connectionist Temporal Classification
Training CTC network: Maximum likelihood
DP formulation for CTC loss
Inferencing from CTC network

Encoder-Decoder architecture
Attention mechanism

Key-value-query formulation of attention
Language translation model
Speech recognition model

Self-attention and transformers

Computing self-attention
Transformer architecture

Conclusion

Points to remember

Further Reading

11. Natural Language Processing

Structure
Objectives
Natural language

Syntactic structure of language
Parts of Speech (POS)
Phrases
Clause
Sentence
Document and Text corpus

Semantic structure of language

Wordnet

Text preprocessing

Models for text

Bag of Words (BoW) model

Vector Space Model

Count based or Boolean

Term Frequency (TF)-Inverted Document Frequency (IDF)

Latent Semantic Indexing (LSI) model

Probabilistic models of text

Topic models

Probabilistic generative models: Latent Dirichlet allocation

Neural language models

Contextual models

ELMo model

BERT

Position encoding

Pre-training BERT

Input representation for pre-training tasks of BERT

WordPiece tokenization

ERNIE

Generative Pre-Training by OpenAI

Conclusion

Points to remember

Further reading

12. Generative Models

Structure

Objectives

A simple generative model

Variational Autoencoders (VAE)

Generative Adversarial Nets

Equilibrium state for GAN training

Implementing GAN

GAN training challenges

Solutions for mitigating GAN training issues

Wasserstein GAN (WGAN)

Some properties of EM distance

WGAN training

[Ensuring Lipschitz Constraint in Discriminator](#)
[Conditional GAN \(cGAN\)](#)
[Cycle GAN \(CycleGAN\)](#)
[Autoregressive generative models](#)
[Applying generative models](#)
[Conclusion](#)
[Points to remember](#)
[Further Reading](#)

[Index](#)

CHAPTER 1

Overview of AI

From the age of civilization, humans are making machines to reduce physical labor. Today, the world is full of machines. Machines cultivate and harvest our crops, make our houses, fly our planes, assemble our cars, control traffic, cook and pack food, entertain us, and even take care of us when we are sick. Machines have not only replaced physical labor but have also exponentially increased human capability. However, a majority of these machines work by following a set of predefined steps required to complete a task successfully. Computing machines and algorithms are at the core of these big and small machine. Algorithms help us formally define the steps to be executed by a machine, and the computer hardware execute these steps in sequence to complete the given task. With advancement of computing capability, new algorithms to solve more complex problems have evolved.

For the past few decades, we have been trying to build intelligent machines that can think and take decisions. Since then, machines are taking over more and more tasks from us. They began to control other machines for us. On this path of evolution, we have strived to impart human intelligence like reasoning, creativity, analyzing, problem solving ability, and natural language understanding to computers. The field of algorithms that strive to impart human intelligence to machines is called **Artificial Intelligence (AI)**.

Structure

In this chapter, we will cover the following topics:

- AI Systems
- Categories of AI Algorithms
- Applications of AI
- Role of Mathematics in AI

Objectives

This chapter gives a high-level overview of AI and its various components. You will be able to learn about common terminologies like model, data, parameters of models, and dependent and independent variables in this chapter, which will be referenced repeatedly in the subsequent chapters. Lastly, we will cover why mathematics is important for understanding AI.

AI systems

AI is a multidisciplinary field of research with a goal to create technology that can enable machines to function like humans. Human mind consists of memories, intellect, thoughts (emotions), and a sense of identity. Human intellect is the discriminative faculty of the mind that determines whether an action is right or wrong. The sense organs present the current situation someone is in, to their intellect. Then, intellect consults the memory, past experiences, present thoughts, and emotions and decides the action. The actions can be speaking, running, smiling, crying, fighting and so on. So, for a machine to function like a human, it should have all these capabilities. Well, machines may not have emotions to influence their decisions! But machines must learn from past experiences, and these experiences must influence their decisions. At first, a machine should have the sense organs by which it can digitally map and record our physical world. Then, it must have the ability to learn from the mistakes it makes.

AI systems are classified by their ability to imitate human behavior. The classification is as follows:

- **Artificial narrow intelligence (ANI or narrow AI)** refers to a computer's ability to perform a single task extremely well. This is the only type of AI that exists in reality. For example, voice assistants like *Siri*, computer playing chess, flying aeroplanes, recommending products and online content as per our interest. These machines don't think, and they also don't have emotions like humans.
- **Artificial general intelligence (AGI or strong AI)** is when a computer program can perform any intellectual task exactly like a human, that is, machines exhibit human intelligence. They can reason, represent knowledge including common sense, plan, learn, and converse in natural language. The general AI does not exist in reality today, but the idea is depicted in many sci-fi movies like *Interstellar*. Also, there are many theoretical frameworks and models proposed for AGI. Alan Turing, who first posed the question in 1950, 'can machines think?' also suggested a

test to evaluate this. *Turing Test*: A machine and a human converse with a second human who cannot see or know with whom they are conversing. This second human should evaluate and conclude who is human and who is machine. If the machine can fool the human evaluator, it means AGI is achieved.

- **Artificial super intelligence (ASI)** is an AI system that surpasses human intellect, that is, machines having greater problem solving and decision-making capabilities that are far superior to human beings. This is hypothetical AI. For such machines to be of any use for life, they must be ethical, must understand human emotions, and must be self-aware. Self-awareness is required for safety, effectiveness, trustworthiness, transparency or explainability of decision-making. Self-awareness also allows for faster reactions and adaptations to changes in dynamic environments. This means machines must have some level of consciousness like humans or possibly, much higher levels of consciousness than humans!

In this book, we will limit our discussions to ANI only. [*Figure 1.1*](#) depicts various forms of ANI at a very high level. Each of these is a huge independent literature in its own. Refer to the following figure:

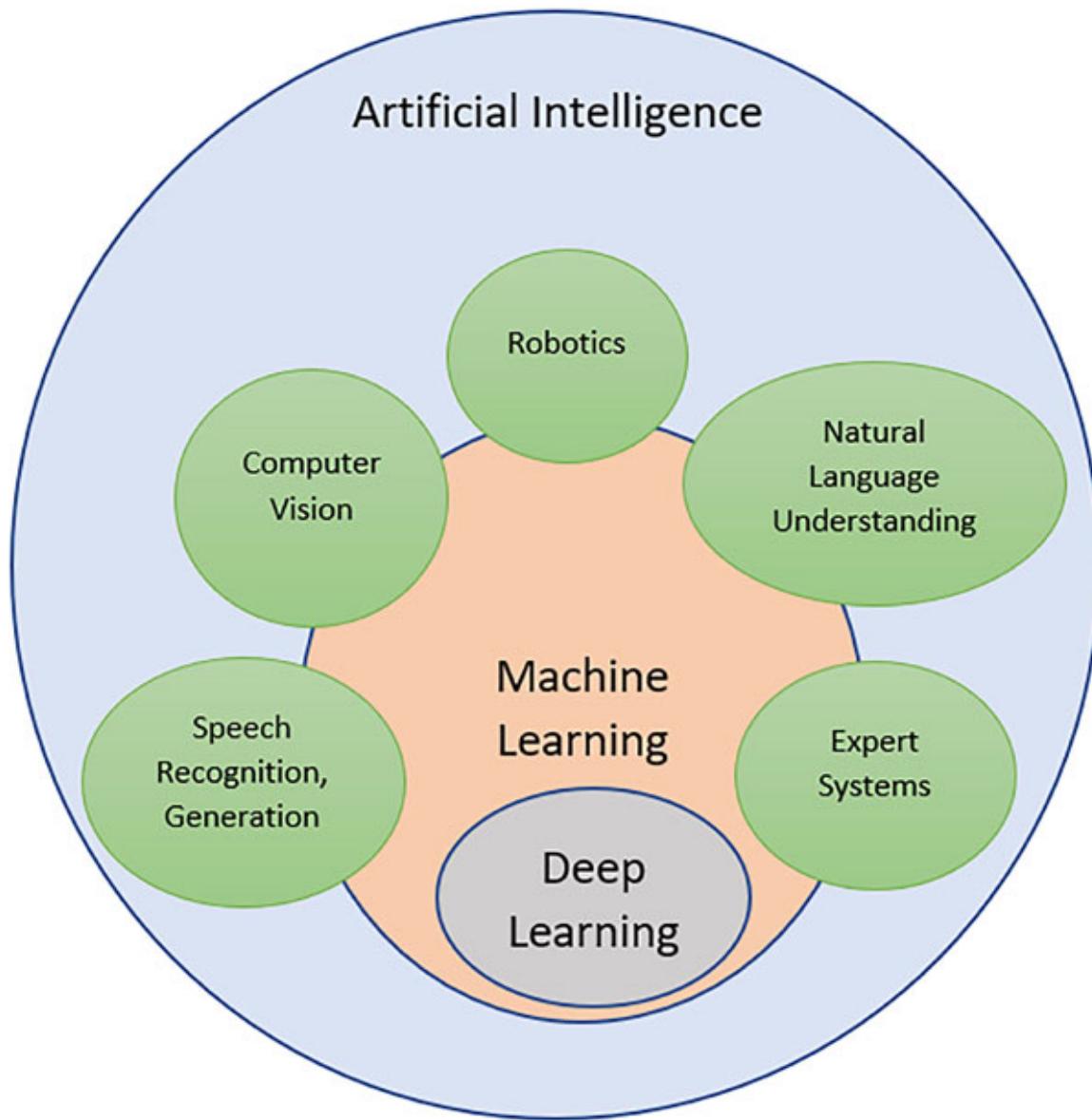


Figure 1.1: Subcomponents of AI present today

The computer vision literature gives eyes to the AI system. The speech processing literature helps the system to listen by acting as its ears. Robotics uses these artificial senses and gives limbs to the AI system, like robotic arms that can perform highly sophisticated tasks like microscopic surgery. Analyzing natural language spoken by human and understanding it helps the system to converse with humans in any natural language. An **expert system** is a computer system emulating the decision-making ability of a human expert. The first expert system was built in 1970s, which had two main parts: a *knowledge base* that represented facts and rules, and *inference engine* that applies the rules to the known facts to deduce new facts. All these together are

essential parts of a future AGI system. **Machine learning (ML)** is a broad class of algorithms that is used to build these components of an AI system.

Machine Learning

Machine Learning (ML) includes the study, design, and development of algorithms to give computers the capability to learn from data instead of requiring explicit programming of hard-coded rules. The process of discovering an algorithm involves manual analysis of input-output examples and deriving set of rules and steps such that given the input, we can always find the output by following these steps. This may be easy for certain class of problems, but when the number of possible rules is very high, it becomes almost impossible to figure out all possible rules to find a robust algorithm. [Figure 1.2](#) shows the difference of classical algorithm development vs ML:

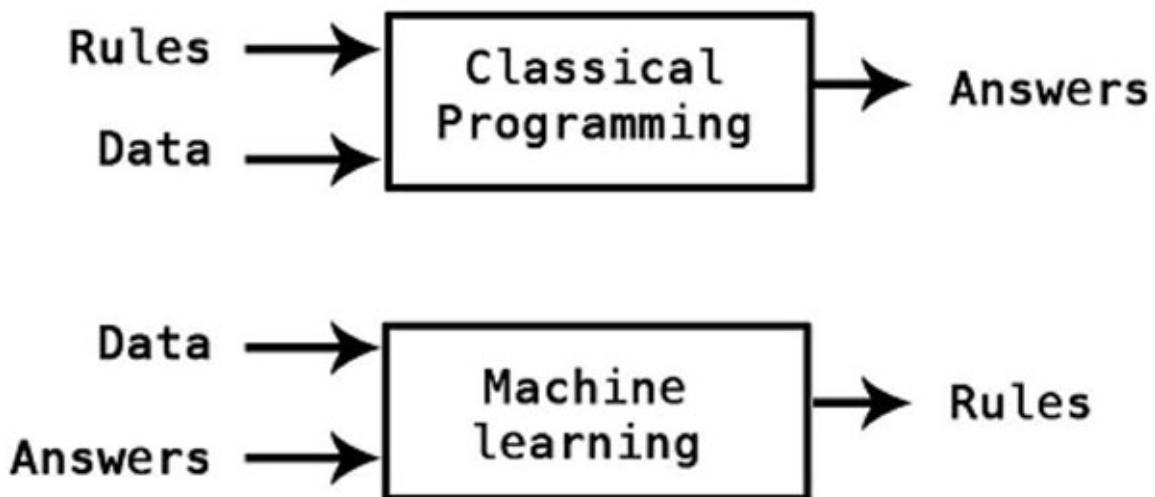


Figure 1.2: Difference b/w classical programming and machine learning

To understand this better, let us consider the problem of classifying flowers. Consider a huge basket of flowers, and assume that there are three categories of flowers, as shown in [Figure 1.3](#):

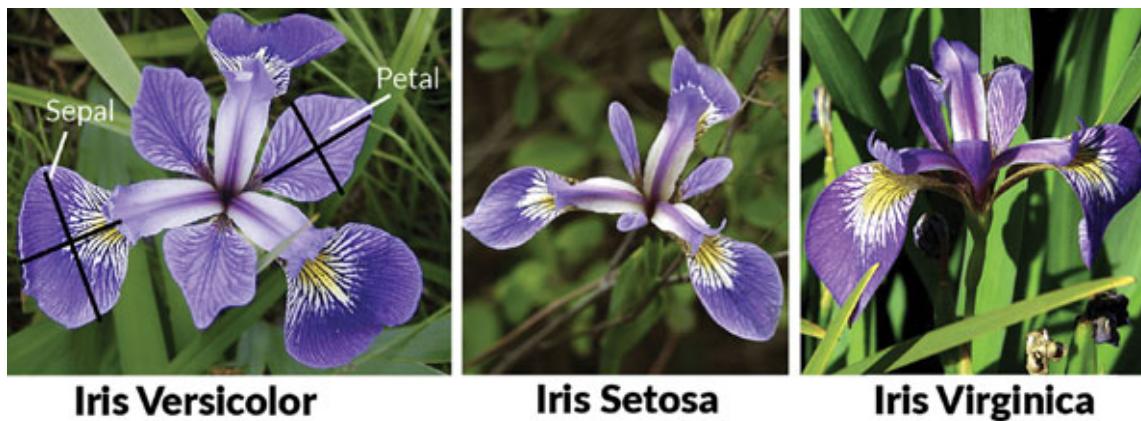


Figure 1.3: Three species of Iris flower

Flowers vary in size, colour, texture, and shape. We want to build an algorithm than can classify the flowers to a type. The first step is to select the properties or features of flowers that will be useful to identify flower species. Once selected, these features are represented with a numerical value that an algorithm can take as input. Here, we have considered sepal length, sepal width and petal length, petal width as the four features. [Figure 1.4](#) shows a sample of five flowers and their corresponding features and categories:

Sample Observations	Features				Class/Target type
	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	
Independent Variables	7.1	3.0	5.9	2.1	virginica
	6.2	2.9	4.3	1.3	versicolor
	5.6	3.0	4.1	1.3	versicolor
	5.4	3.4	1.7	0.2	setosa
	5.0	3.6	1.4	0.2	setosa
Dependent Variable					

Figure 1.4: Iris flower's features values and its species/category

Once features are identified, we must define set of rules to classify a flower. Here, our output or *target* variable is the category of flower. This is also called *independent variable* and the features are termed as *dependent variables*. [Figure 1.5](#) is an example of rules using only the first two features. The range of values of sepal length and petal length differ. We can apply some data transformations such that all the feature values in the dataset to are mapped a common scale, without distorting differences in the ranges of values. This is called *data normalization*. The feature values are normalized or scaled such

that they are centred around zero; that's why we see negative and zero values in the axes. The original data set has all positive quantities as all the four flower features are length or width. Refer to the following figure:

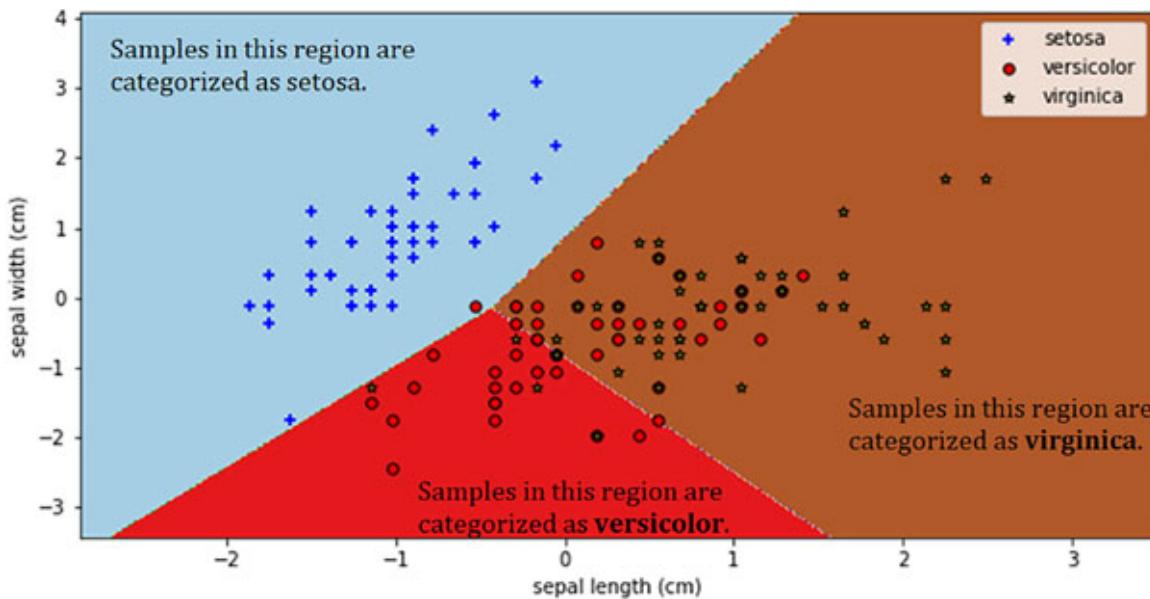


Figure 1.5: Identifying the Iris species using first two features

Simple rules to classify the flowers into the respective categories by observing only two features are as below:

- If the normalized sepal length < 0 and normalized sepal width > 0 , then it's setosa.
- If the sepal length and width fall in the lower triangular region, it's versicolor with a high chance. This triangular region can be defined by three straight line equations.
- Otherwise, its virginica.

This collection of rules or a mathematical function representing these rules that helps to identify flower type is called a *model*.

Manually deriving rules or a mathematical function is time-consuming task. Machine learning algorithms try to automate this process by learning rules or decision boundaries or a mathematical function that takes a flower's numeric representation as input and outputs the possible category of the flower.

Does selection of features impact the classification accuracy of the learned model? Yes, classification accuracy depends on selected features of flowers. We see in the preceding example that, as we have chosen only two features, we

are unable to properly distinguish the two classes: versicolor and virginica. In the next section, we will discuss a step-by-step process for designing an ML-based model.

How are ML Models created?

Building a ML model is an iterative process. It starts with understanding the business problem and then collecting data related to the problem domain. Then, this data is processed, cleaned, and prepared for modelling. This is depicted in [Figure 1.6](#):

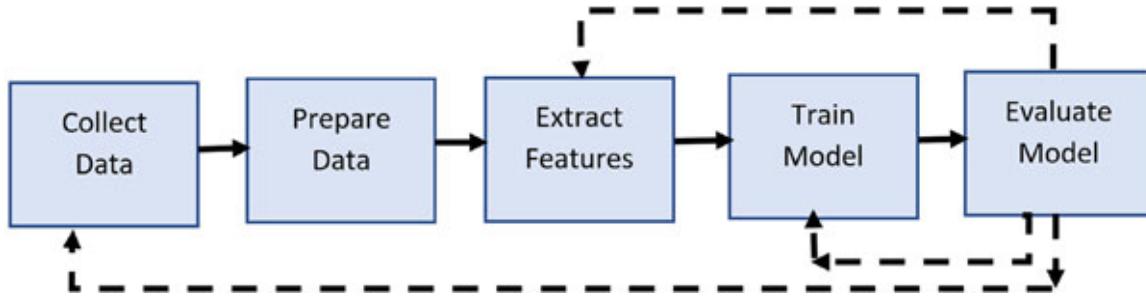


Figure 1.6: Building ML model

Following are the various steps for building ML model:

1. **Data collection:** The process of collecting observations or data related to the problem is called **data collection**. In Iris flower subtype classification, we must measure petals and sepal's length and width for each flower in the collected sample. Will these features of the flower and number of sample flowers sufficient for solving the problem? The answer is, we can't initially say if these features or the number of collected samples will be sufficient. Human domain expert in flower identification can provide useful information about sufficiency of the selected features. How many samples of flowers we need to collect? This will not be clear initially. After the model building and initial analysis, we can revisit this question. A rule of thumb is, if we have more features for each sample, we must collect more data samples.
2. **Data preparation:** In this step, the observations are analyzed to check whether there are any missing values, any error in data collection like an abnormal value of observation, and if so, those must be corrected or removed from dataset.
3. **Feature extraction/selection:** The features of the cleaned data are further analyzed for obvious intercorrelations. It may happen that some features

are very highly correlated, and using any one of these related features is sufficient to solve the problem. There may be features that are not important at all for the problem. These are statistical checks that are discussed in detail in [Chapter 5, Statistical Inference and Applications](#). After this step, a few features are selected. Sometimes, we may have to derive a new feature from the collected features. For example, we may use a logarithm function to transform a feature value and use the log value as the feature. This step is also called *feature engineering*.

4. **Train model:** Choosing a mathematical function that accepts selected features and outputs desired result: *Model selection*. Generally, these mathematical functions are *parametric*, that is, their functional forms are fixed, but changing the parameters will change the function. In the Iris flower classification shown in [Figure 1.5](#), the parametric function is a straight line in two dimensions having the general equation: $ax + by + c = 0$, where a, b, c are parameters. In fact, we have 3 such lines; hence, we need 9 parameters to define the model function. For one set of fixed values of these parameters, we have a decision boundary or rules that we call model. The process of finding these parameter values that provides results near to the expected (or ground truth) is called *model training*.
5. **Model evaluation:** Various metrics are designed to access the quality of the models created in the previous step. These metrics are different for different types of ML algorithm. These are discussed in the following section on ML model types.
6. If it's found that the model quality is not acceptable, then various ways of improving the models are tried. This may involve choosing a different functional form, like using a quadratic function $ax^2 + bx + cxy + d = 0$ as the model, and its parameters are again estimated. If we are still unsuccessful at finding a good model, we may have to go back to the data preparation stage and design more features or may have to go back further to collect more relevant data and features to solve the problem. These are depicted as the dotted lines going backward in [Figure 1.6](#).

There are various types of ML algorithm for solving different types of problems. All these algorithms are built iteratively by learning from data. The data or observations about a problem domain is the starting point of ML algorithms, and then these algorithms are iteratively improved by taking feedback from the data. We will first discuss briefly the different types of data and then the different types of ML algorithm.

Data types

Data is the starting point for solving any problem in AI. Data can be broadly categorized into two types: structured and unstructured. Structured data is tabular data where we have certain predefined features or attributes, that is, the columns are defined in the table. The rows in the table contain values of these attributes. Unstructured data is information that is not arranged according to a pre-defined data model or schema, and therefore, cannot be put in a tabular form. In [Figure 1.7](#), these two categories are further split into different subcategories with examples:

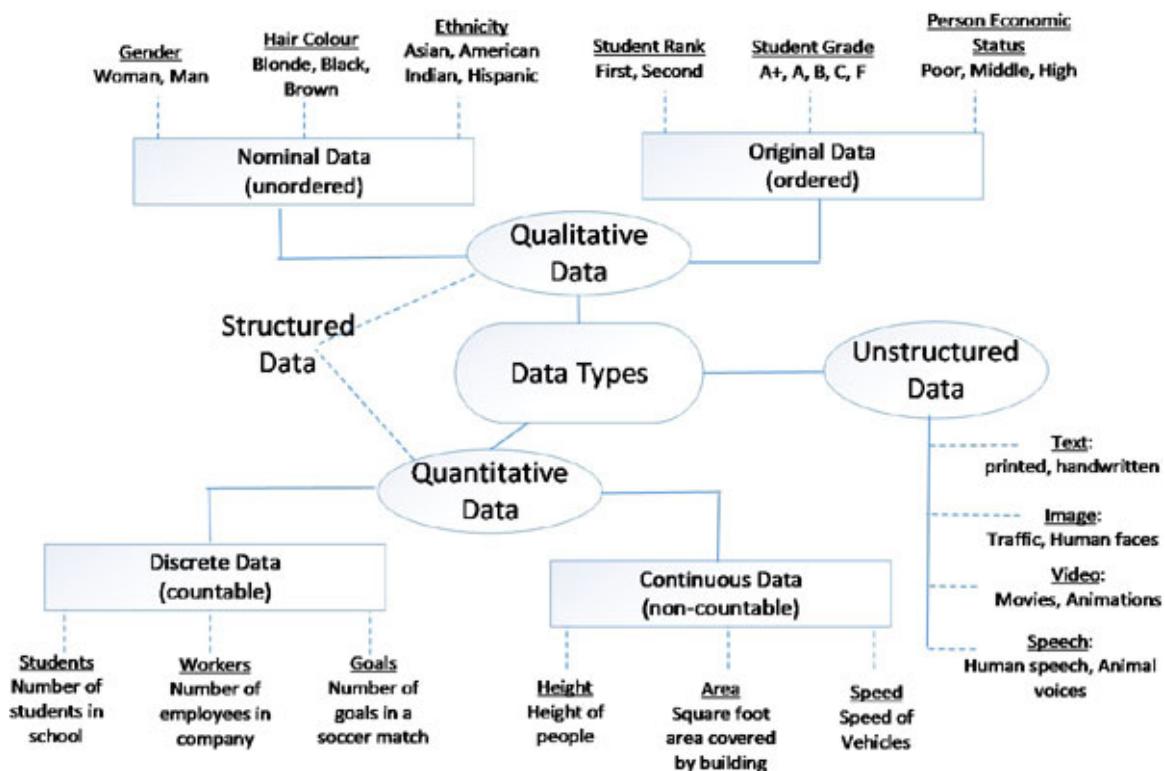


Figure 1.7: Data types

All data types must be converted to numerical form before feeding them into machine learning model. This is done in the feature extraction phase of ML model building.

Learning From data

For algorithms to learn from data, there needs to be feedback about the rules or logic learnt by the algorithms. Based on the feedback, algorithms will learn better representation of the data to achieve desired output. Different algorithms

are required for different degrees of feedback obtained. Next, we will discuss the various types of these algorithms.

Types of ML algorithm

We can categorize machine learning model types based on the level of the feedback that algorithms receive during its learning phase. This is depicted in [Figure 1.8](#). Let's discuss these three types of algorithms in further detail. Refer to the following figure:

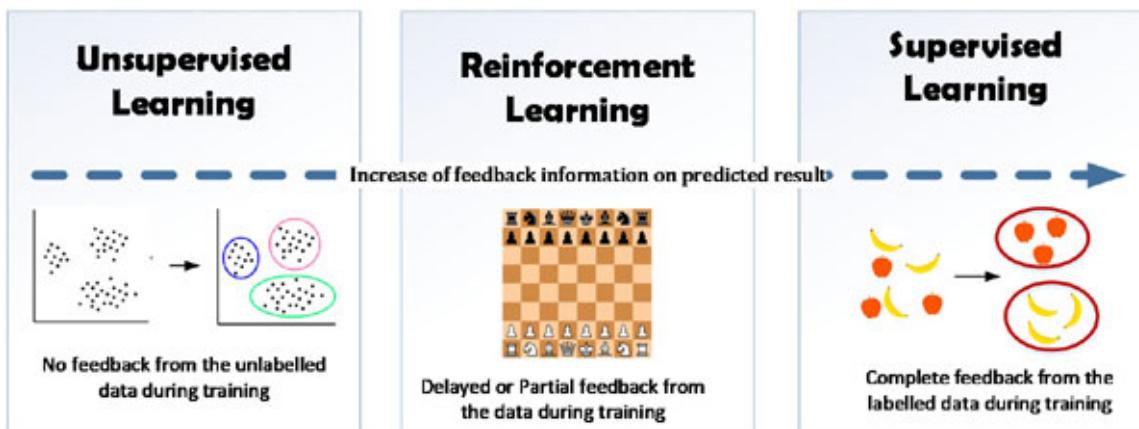


Figure 1.8: Model types based on feedback during learning

Unsupervised learning

Unsupervised learning is about identifying unknown patterns/groups from the given unlabeled data. Here, classes of data samples or total number of classes or desired output for each data sample is not part of the data; this kind of data is called *unlabeled data*. Two popular techniques that fall under this category are clustering and dimensionality reduction.

Clustering is about automatically discovering natural groups/clusters in the unlabeled data so that the degree of similarity between samples of the same cluster and the degree of dissimilarity between samples of different clusters are maximized. Similarity and dissimilarity criteria can vary based on the problem statement. The similarity between data points is defined using a distance function. There are also various ways of evaluating the quality of clusters formed, which are discussed in further detail in the [Chapter 7, Clustering](#).

Dimensionality reduction is transformation of data from high-dimensional space to low-dimensional space, such that data represented in lower-dimensional space retains the properties of original data to achieve the required

task. [Figure 1.9](#) shows a simple example of dimensionality reduction. Here, we have a two-dimensional data distributed, represented with axis f_1 & f_2 . We need two dimensions to represent the data. Suppose we now rotate the axis along the line of distribution; we can then represent the points using one axis e_1 by projecting these points on e_1 . Here, the scattering of the data was along the e_1 direction, and we have rotated the axes. We can reduce dimensionality of the data using this principle and some mathematical tools from linear algebra. Refer to the following figure:

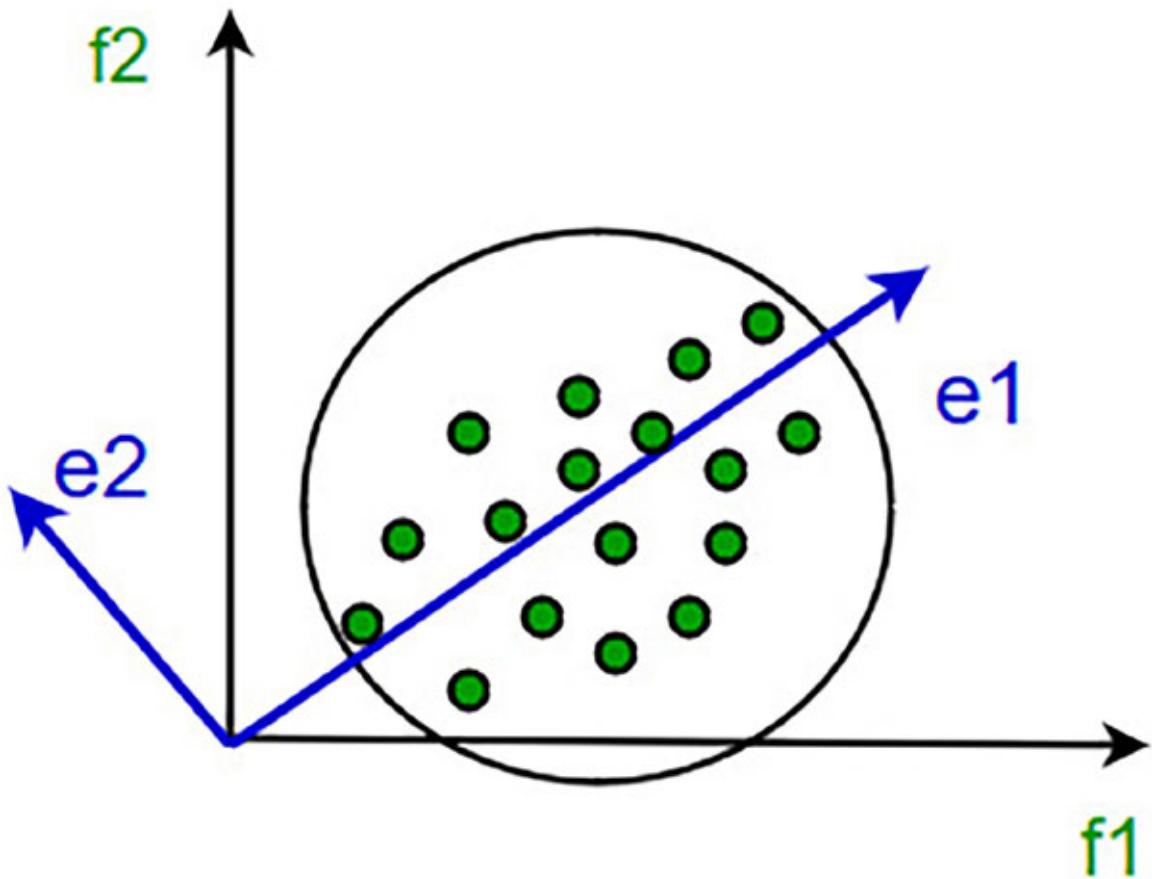


Figure 1.9: Finding optimal number of new axes for the given data

Reducing dimensionality is beneficial as algorithms will overcome sparse data representation and *curse of dimensionality*. The phrase *Curse of dimensionality* is coined by Richard E Bellman, and it refers to various challenges that arise when analyzing data in high-dimensional spaces. As dimensionality increases, volume of the space increases exponentially, which make existing data sparse. For algorithms to work reliably, we need to increase the data exponentially. Choosing right features and converting data to lower-dimension space plays an

important role in the success of machine learning algorithms. Due to this, dimensionality reduction is often used as an intermediate step for various machine learning algorithms. We will discuss these techniques and its applications in greater detail in [Chapter 8, Dimensionality Reduction](#).

Reinforcement learning

There exist many situations where there is partial feedback or the feedback is delayed. Consider the game of chess where the objective of the task is to win the match. There do not exist feedback about every move. Feedback is delayed to the end of the game. There do exist partial feedback during the game when a piece is captured. Capturing opponent's piece is positive sign but doesn't guarantee the win. When rewards or feedback is received from the game/environment, it must be recorded, and the path taken to reach the present state must be rewarding accordingly. This approach of utilizing partial or delayed rewards/feedback to learn actions for various situations/states is called **Reinforcement Learning (RL)**. The objective of the RL algorithm is to find optimal action for each state that would result in maximum cumulative long-term reward.

[Figure 1.10](#) shows an example of a simple RL problem: A robot trying to walk as long as possible without falling: The robot can be in three states: Fallen state, Standing state, or Moving state. The robot can perform only two actions: moving the legs slowly, as depicted in [Figure 1.10](#) with light-colored arrows, and moving the legs aggressively/fast, as depicted in [Figure 1.10](#) with dark-colored arrows. Given that the robot is in any of these three states, the dark arrows show what happens with slow action, and the light arrows show what happens with the aggressive action. The number over these arrows shows the partial feedback or reward on taking the action. These rewards are given by the environment where the robot is walking. The ultimate goal of the robot is to learn a strategy or policy such that it can walk for very long time, that is, to discover the best possible action (slow or fast moving) at each state so as to maximize the cumulative future reward. Refer to the following figure:

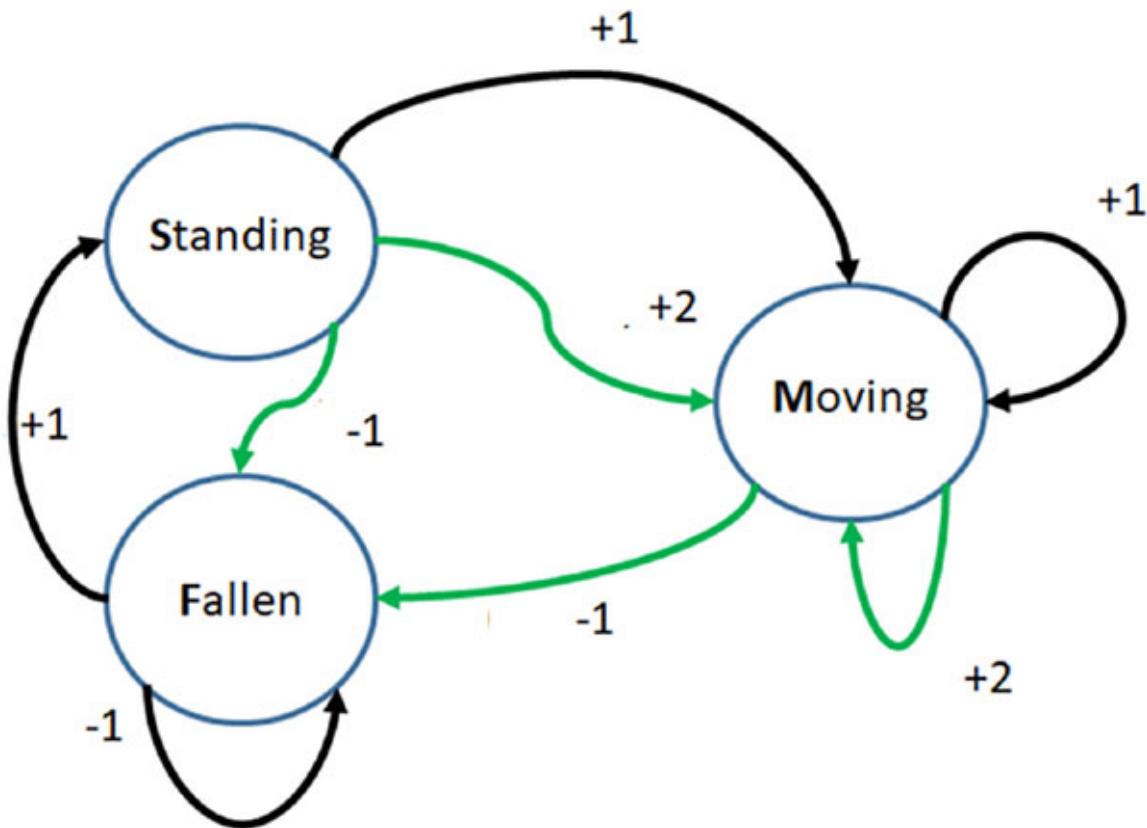


Figure 1.10: State transition experienced by the robot

Figure 1.10 represents the environment the robot is walking as a state transition diagram. One important point to note here is that this environment is not deterministic, that is, taking a fixed action at a given state may either lead the robot to any of the possible states reachable from the given state. Suppose the robot is in fallen state; it may be able to stand by moving its legs slowly or may remain fallen. The chance of landing to another state may vary and is an inherent property of the environment. The objective of RL is to learn the best possible strategy under these uncertain conditions. *Figure 1.11* shows examples of 3 possible policies π_1, π_2, π_3 the RL algorithm can learn. Policies are outputs of RL algorithm. Refer to the following figure:

s	$\pi_1(s)$
fallen	Slow
Standing	Fast
Moving	Fast

s	$\pi_2(s)$
fallen	Slow
standing	Slow
moving	slow

s	$\pi_3(s)$
fallen	Slow
standing	Slow
moving	fast

Figure 1.11: RL policies

Following are the different components of a RL problem:

- **Agent:** This is the component that makes the decision of what action to take; it is the robot's decision-making algorithm in the previous example.
- **Environment:** This is the thing agent interacts with, comprising everything outside the agent. The floor area on which the robot can move along with external factors like wind adds uncertainty to the outcome of action. States the robot is in are associated with the body of the robot. So, the body of robot is also part of the environment.
- **State:** This is the current condition of the environment, for example, whether the robot is fallen or standing or moving.
- **Action:** This is the move taken by the agent. In previous example, there are two possible actions at every stage: slow moving and fast moving.
- **Policy:** Defines the agent's way of behaving at a given time and state. It's a mapping from perceived states of the environment to actions to be taken when in those states. This is the output of the RL algorithm.

If number of states are few and transition probabilities are known, then there exist dynamic programming-based algorithms like *policy iteration*, *q-learning* to learn the policy. For large state space, function approximators are used to learn the policy.

In industry, RL-based robots are used to automate various tasks. One example is AI agents by DeepMind to cool Google data centers, which led to a 40% reduction in energy spending. RL algorithms can learn policies from medical diagnosis of patients and then can act as a virtual doctor where patients can receive treatment from policies learned by RL systems. RL is also being used for stock trading.

Next, let's look at another class of ML algorithms where complete feedback is provided from the data. Here, the data used to build the model is called training data. Each instance of the training data has one or more target features, which act as feedback to the training algorithm.

Supervised learning

Supervised learning is about learning parameters of the function based on the labelled data. In labelled data, desired output for each data sample is provided. Output desired for each data sample can be either categorical data representing

a class label for the data instance or real number (continuous variable) indicating some measurement. If desired output represents class number, then it is called *classification*. If desired output represents continuous variable, then it is called *regression*. Identifying the type of Iris flower discussed before is an example of supervised classification where the target label are the three classes of flowers. An example of regression would be predicting the price of a house based on its location, square foot area, and so on. There are various types of supervised learning algorithms, which we will cover in this book. We will be first providing the mathematical tools required to understand the theory behind these algorithms and then introduce these algorithms along with applications to solve various ANI tasks. Various metrics are defined to evaluate the quality of the learned model for regression or classification. Let's first discuss the classification metrices.

Metrices for evaluating classification model

We will consider an example of 10 predictions for the flower classification problem ([Figure 1.3](#)) to illustrate these metrices. The predictions are made using the model shown in [Figure 1.5](#) by checking which region the point falls. [Table 1.1](#) shows a sample prediction output of a model built on two sepal features, and the true output is depicted in the target column:

	sepal length (cm)	sepal width (cm)	prediction	target
0	0.192454015	2.08478395	setosa	setosa
1	1.132206284	-1.72578699	virginica	virsicolor
2	-0.959849197	2.173531324	setosa	setosa
3	2.952024909	2.138220415	virginica	virsicolor
4	-0.505463006	-2.149987293	virsicolor	virginica
5	0.80187062	0.622172986	virginica	virginica
6	-0.958066983	-2.170298289	virsicolor	virsicolor
7	0.877714008	0.053590407	virginica	virginica
8	-4.388166428	-0.23903155	virsicolor	setosa
9	-1.419429199	-0.686692025	setosa	setosa

Table 1.1: Prediction by a model on test data

We will first define some terms and then define the metrics using those. We will apply these terms and metrics on the output of a model captured in [Table 1.1](#):

- **True Positive (TP):** If the model predicts target class A as A, then the case is called True Positive. In previous table, there are four actual samples from class setosa, and the model has predicted three as setosa. So, the TP count for this class is three.
- **False Negative (FN):** If the model predicts the class A as *not* A (any class other than A) then it is called False Negative. For setosa class here, we have one FN count.
- **False Positive (FP):** If the model predicts *not* A (any class other than A) as A, then it is called False Positive. Considering the versicolor class, we see sample numbers 4 and 8 are predicted as versicolor but are actually not of that type. So, for versicolor, the FP count is 2. However, for setosa class, there is no FP.
- **True Negative (TN):** If the model correctly predicts the class *not* A as *not* A, then it is called True Negative. For the setosa class again, not setosa means all the 6 samples whose true labels are not setosa. we see none of them are predicted as setosa. so, TN count for setosa is 6.

Following are the metrics for evaluating a classification model:

Following are the metrics for evaluating a classification model:

- **Classification accuracy:** Fraction of predicted labels matching exactly with true target labels. Here we have 6 rows out of 10 where we find exact match and hence $accuracy = \frac{6}{10}$.
- **Class-wise accuracy:** Ratio of number of correct predictions for a target class to the total number of actual labels for the target class:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

For setosa, $TP = 3$, $TN = 6$, $FP = 0$, $FN = 1$ and hence, $accuracy = \frac{9}{10}$

For versicolor, $TP = 1$, $TN = 5$, $FP = 2$, $FN = 2$ and hence, $accuracy = \frac{6}{10}$

For virginica, $TP = 2$, $TN = 5$, $FP = 2$, $FN = 1$ and hence, accuracy = $\frac{7}{10}$

- **Precision:** The ratio of TP count for a class A to total number of predicted labels A by the model.

$$precision = \frac{TP}{TP + FP}$$

- **Recall:** The ratio of TP count to the total actual positive count for the class. This is also known as **True Positive Rate (TPR)** or **Sensitivity**:

$$recall = \frac{TP}{TP + FN}$$

- **F1 score:** The harmonic mean of recall and precision is called F1-score. It provides a balanced score for precision and recall. The F1 will be high only when both precision and recall are high. Generally, increasing recall by modifying the prediction algorithm will decrease precision and vice versa. This is called precision/recall trade-off. Using the Python Scikit `metrics.classification_report` function, we can calculate the F1 score, precision, recall and accuracy together; the output is captured in [Figure 1.12](#):

	precision	recall	f1-score	support
setosa	1.00	0.75	0.86	4
virginica	0.50	0.67	0.57	3
versicolor	0.33	0.33	0.33	3
accuracy			0.60	10
macro avg	0.61	0.58	0.59	10
weighted avg	0.65	0.60	0.61	10

Figure 1.12: Classification report

- **Confusion matrix:** Consider a $n \times n$ matrix (where n is the number of targets) with rows representing an actual class and columns representing a predicted class. The row sum of this matrix will be equal to the support or number of true class labels for each class. The diagonal element will show the TP count the (i, j) the entry of the matrix, where $i \neq j$ represents number of misclassifications of the i^{th} class as j^{th} class. Confusion matrix for the example is captured in [Figure 1.13](#):

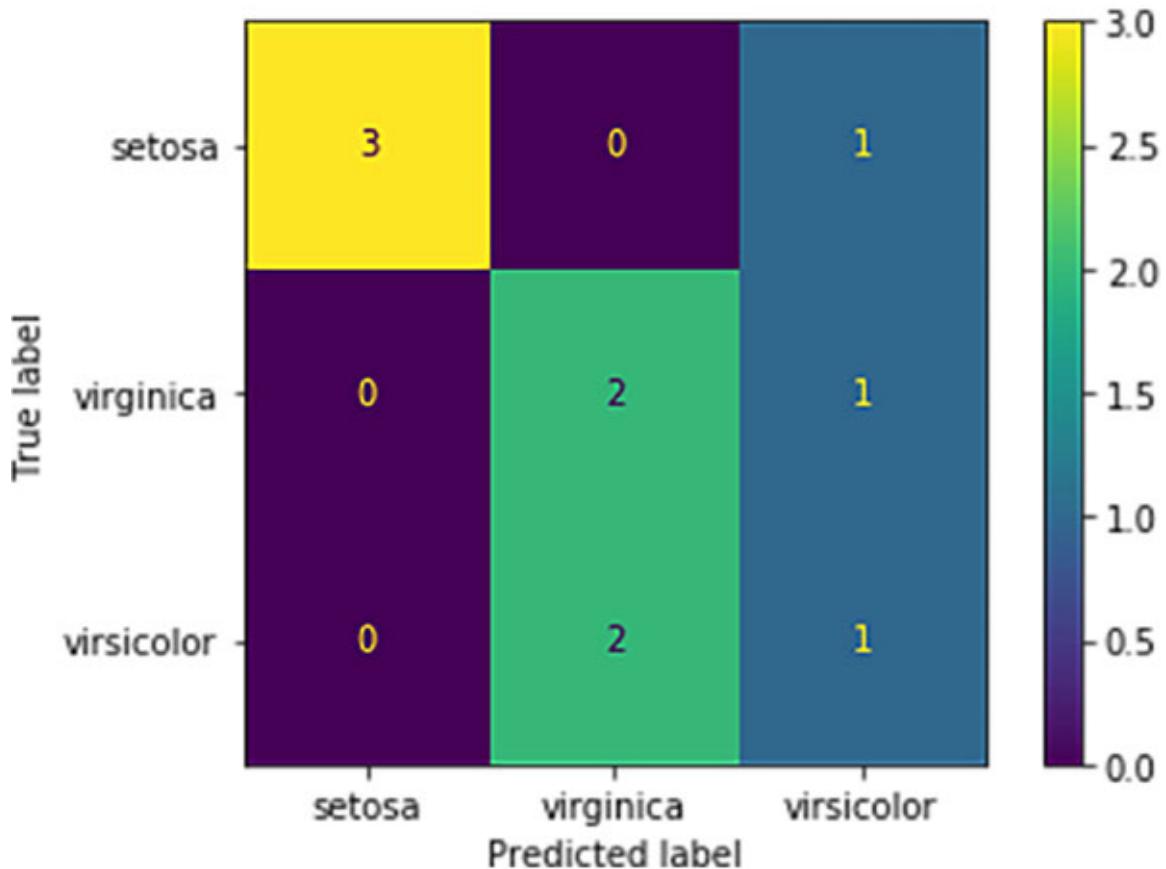


Figure 1.13: Confusion matrix

The best desired confusion matrix is one which has large diagonal elements and small entries in the rest of the matrix.

Based on the classification problem we are solving, some of these metrics may have more importance than others. For example, if we are detecting whether a transaction is fraudulent, it's more important to detect a fraud. We need high recall in this case at the cost of precision. As most of the models output some score for a prediction, these adjustments in predictions can be done by putting some thresholds. For this fraud detection case, suppose our model outputs a score between $[0, 1]$. We may predict a transaction as fraud even if $\text{score} > 0.3$ and non-fraud otherwise. Thus, increasing recall and compromising on precision. Varying the prediction thresholds, we can come up with the following metrics and get the best out of our model:

- **Specificity or True Negative Rate (TNR):** The ratio of number of negative classes, that is, *not A*, which are correctly being classified as *not A*.

$$TNR = \frac{TN}{FP + TN}$$

- **False Positive Rate (FPR):** The ratio of number of negative classes, that is, *not A*, which are inaccurately being classified as A.

$$FPR = \frac{FP}{FP + TN}, \text{ thus } FPR = 1 - TNR$$

We can also compare two different prediction models using these rates. For that, we need another metric called **Receiver Operating Characteristics (ROC)**.

- **Receiver Operating Characteristic (ROC) curve:** ROC plots the **True Positive Rate (TPR)** vs **False Positive Rate (FPR)**, as shown in [Figure 1.14](#). The area under the curve is used as a measure. For a perfect classifier, the area under the ROC curve is 1, and hence, the closer the area under the ROC curve is to 1, the better the classifier. It's generally used to compare two different prediction models. Refer to the following figure:

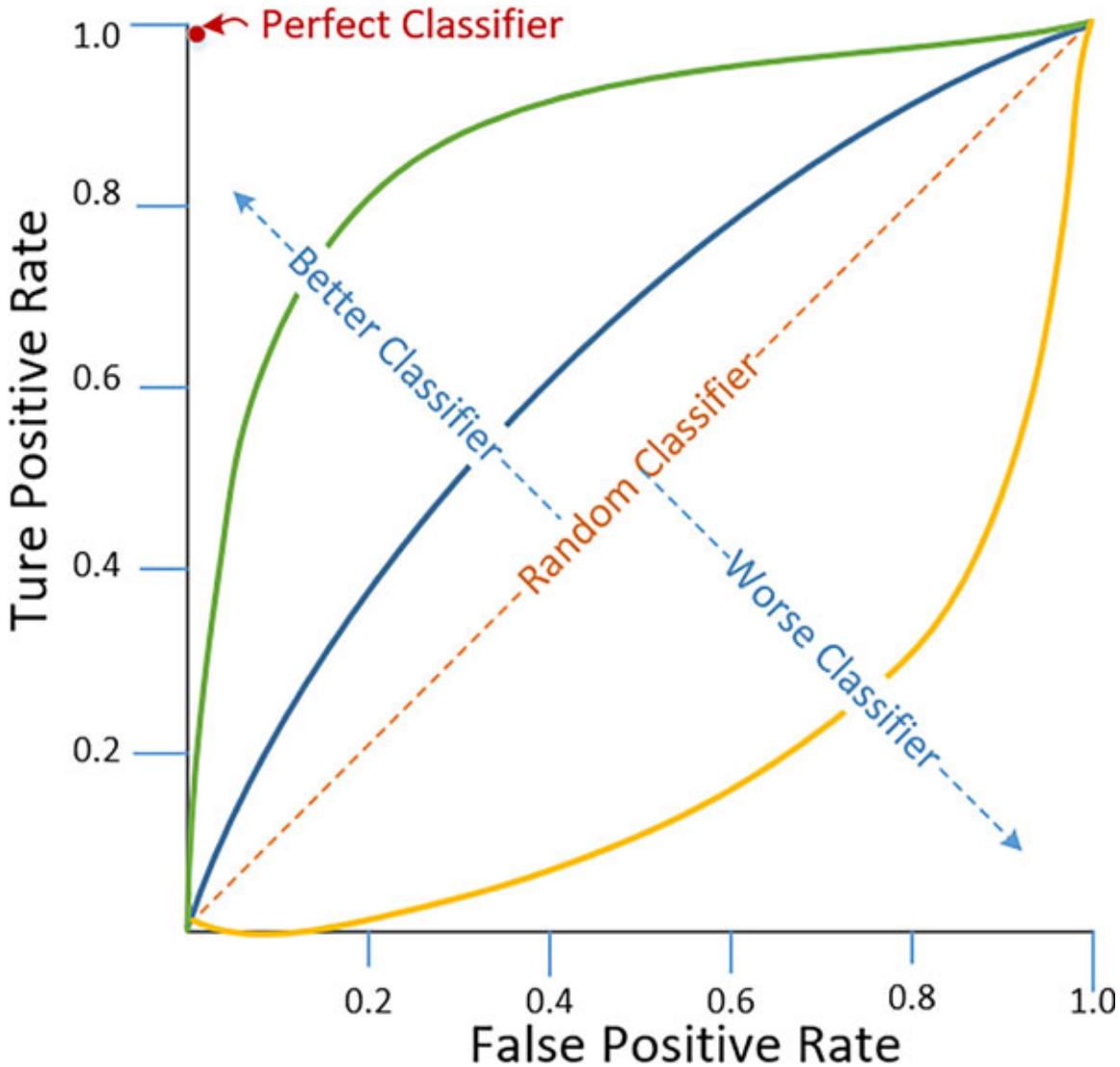


Figure 1.14: ROC curve

Next, let's look at the metric used for regression tasks.

Metrics for evaluating regression model

For discussing the regression metrics, let's take a simple linear regression example. Suppose we want to predict the weight gain based upon calories consumed only, and we have a sample data collected as shown in [Table 1.2](#):

id	calories	weight_gain
0	1489	5.167585591
1	1446	6.172757721
2	1222	6.38994428

3	1141	3.915110902
4	206	4.047348025
5	1247	3.285284391
6	1338	4.404260107
7	196	3.160958623
8	213	6.701951781
9	738	3.64042916

Table 1.2: Calories intake and resulting weight gain

Here, we have only one independent variable, which is the calories consumed (x). We have plotted this data in [Figure 1.15](#). Suppose our mathematical model for regression is a straight line $y = 0.0004 x + 4.2$, as shown in [Figure 1.15](#). Then, for calorie consumed = 1222 (2nd sample above), the predicted weight gain is 4.7 but the actual weight gain is 6.38 kg. Refer to the following figure:

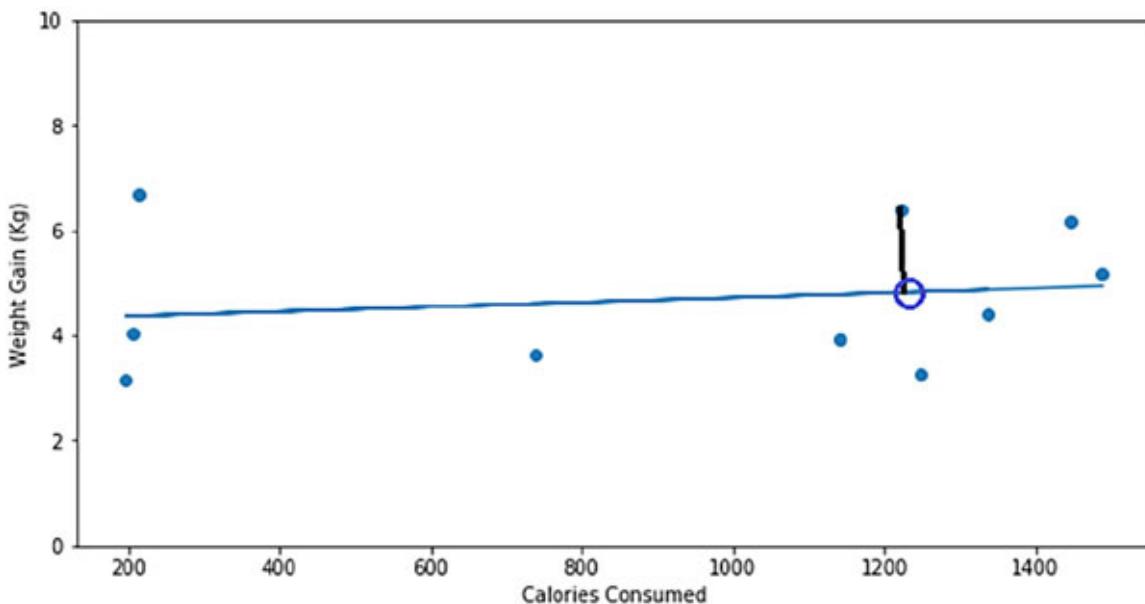


Figure 1.15: Plot of calories and resulting weight gain

Following are the metrics for evaluating a regression model:

- **Mean Absolute Error (MAE):** MAE is a very simple metric that calculates the average absolute difference between actual and predicted values. In the previous example, the predicted value by our model is point on the line corresponding to each value of the calories in the x-axis; thus:

$$MAE = \frac{1}{10} (|5.18 - 4.79| + |6.17 - 4.78| + \dots + |3.64 - 4.49|) = 1.075$$

This indicates that, on an average, the weight gain estimate by our model above is going to have an error of ± 1.075 kg error. This error has the same unit as the target variable.

- **Mean Squared Error (MSE):** MSE finds the average squared difference between actual and predicted value.

$$MSE = \frac{1}{10} ((5.18 - 4.79)^2 + (6.17 - 4.78)^2 + \dots + (3.64 - 4.49)^2) = 1.536$$

The squared error is more for points far away from the predicted value compared to MAE. But the error is now a squared quantity and does not have the same unit as the predicted value.

- **Root Mean Squared Error (RMSE):** RMSE is a simple square root of mean squared error. This has the same unit as the target.

There are few other metrics for measuring regression like R-squared and adjusted R-squared for measuring regression error. We will be revisiting these metrics in the subsequent chapters.

For comparing various models of regression, there are few statistical measures. Models are scored both on their performance on the training dataset and based on the number of model parameters or the complexity of the model.

- **Akaike Information Criterion (AIC):** AIC penalizes models that use more parameters.

$$AIC = 2k - 2\ln(L)$$

k is the number of model parameters. L log of the probability that the model could have produced your observed target values. Lower the AIC, better is the model. Calculation of these log probabilities will be discussed in the later chapters.

- **Bayesian Information Criterion (BIC):** Another similar metric that also takes the number of examples into consideration for scoring the models is called BIC. Lower BIC values indicate better models. We will provide the mathematical formula for this later as it requires some more theoretical foundations of regressions to be introduced.

AIC, BIC can be also calculated for classification models and compare them.

In all the above types of ML algorithms, supervised, unsupervised and RL, one important step is feature engineering. This is a manual step that involves handcrafting features from the observations using domain knowledge. To understand the complexity of this step, let's take another example of feature engineering for a slightly complex classification problem: face recognition. Given a query face image and a database of known faces, the task is to find the closest match of the query image with images in the database. The first logical step to solve this problem is to extract features from face images and represent the faces in the database numerically. The query image can be also converted to a set of numeric observations, and then we can compare query image observations with numeric representation of all the images in the database. In order to come up with this representation of the image, we have used domain knowledge – what are the most distinguishing features of a face: eyes, eyebrows, nose, jawline, mouth, and relative distance between these. Then, we have to design algorithms to find these points from a face image.

Thus, we see that the feature engineering step is the most time consuming and difficult in ML. Is there a way we can automate the feature engineering process? A subclass of ML algorithms discussed in following section addresses this.

Deep learning

Deep learning is a subfield of machine learning, where a hierarchical representation of the data is created. Higher levels of the hierarchy are formed by the composition of lower-level representations. More importantly, this hierarchy of representation is learned automatically from data by completely automating feature engineering. Automatically learning features at multiple levels of abstraction allows a system to learn complex representations of the input to the output directly from data, without depending on human-crafted features. Models used in deep learning are generically called **neural networks**.

Neural networks consist of small computation units called neurons (inspired by the biological neurons in human brain), which are basically parametric functions of the input. The output of a neuron is a single real number. Thus, having N neurons, we can get a set of N real numbers or set of N features. Changing the parameter values gives different feature vectors for the same input. For the face recognition example, we can design a neural network which takes a raw digital image as input. The input image is a $n \times n$ array or matrix of pixels. We define a parametric function that computes the weighted average of

every set of consecutive 3×3 pixels in the image and outputs a single value. The weights, used in computing the weighted average, are the parameters of the neuron. These parameters are learned from data. We can have many such neurons with different sets of weights and thus have a layer of neurons representing various image features like edges, color, and texture. Putting multiple hierarchy of layers like this, we can have a network of neurons called deep neural network. The depth of the network is defined by the number of layers of neurons. A comparison between deep learning's approach and classic machine learning's approach is depicted in [Figure 1.16](#):

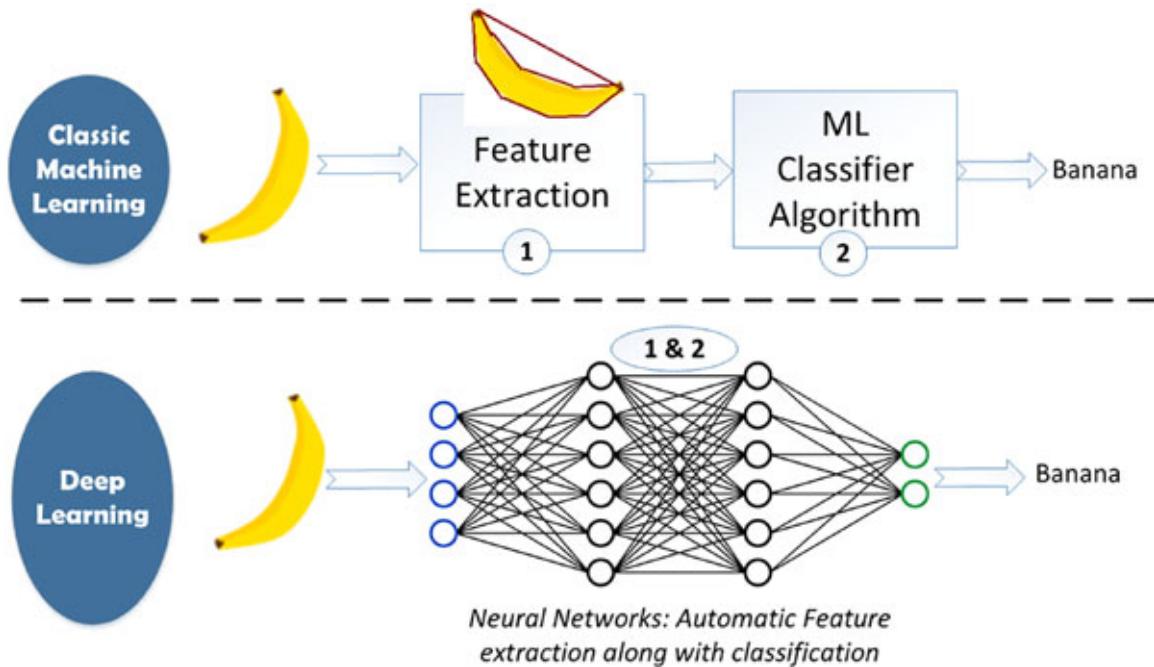


Figure 1.16: Deep learning vs classic machine learning

We will discuss neural networks in greater detail in [Chapter 6, Neural Networks](#), but most of the mathematical tools required to understand the theory of neural networks is covered in [Chapters 2 to 5](#). We will cover various applications of neural networks [Chapter 9](#) onward. The success of neural networks depends on the availability of large volumes of data and immense computing power of present day.

Dataset preparation

Neural networks need large volumes of data for computing features automatically. How much data is sufficient for the algorithm to learn? The rule

of thumb is that the dataset size must increase with an increase in learnable parameters and dimensions of the data.

Tip: We must make sure that samples in the dataset are not repeated or the number of samples of a category is higher as compared to others. This will push a model to learn better representation for the skewed category/samples, leading to lower performance for other categories or samples.

In practice, the entire dataset is not used for training the neural network model. After cleaning of the data, it is divided into three sets: training, validation, and test. Dividing should be such that variation of the data is captured in all three sets. The neural network learning algorithm and many other machine learning algorithm is an iterative algorithm.

Most learning algorithms generally start with a random initialization of parameters and iteratively improve the parameter values by taking feedback from training data. As learning algorithm learns parameter values during training phase, it needs to validate whether it is moving in the right direction. For this, validation dataset is used. After few iterations of learning parameters from the training data, partially trained model is run on validation set with recently run parameters. Performance on the validation set gives direction for the model to search for better parameter's values. The need for model validation is to restrict the model to only work on the training examples and fail miserably on any data outside training examples. Such a model is of no use, and it's called *overfitted* model. The performance evaluation of the model on validation data makes sure that the model is learning general patterns in the data and not memorising the training examples.

Another scenario can also arise. We see that the model is not even able to learn the training data well, and thus, the performance on validation is also not improving. Such a scenario is called an *underfitted* model. This generally indicates that our model needs more parameters or more capacity to learn the patterns in the data. After completion of training, trained model is evaluated on test set, and these numbers are reported as model performance.

Note: The test set is never used in training or validation. The model performance must always be reported on the test set.

For reasonable size dataset, we can split the dataset into training: 80%, and test: 20%. Out of the training set, 5% can be used as validation dataset. If

dataset size if over million samples, then we can split the dataset into training: 98% and test: 2%. The validation set can be 2% of the training examples. Divided sets should reflect similar patterns (statistical distribution) when analysed. Skewed data towards any pattern or class in any of the sets would lead to degradation of the learning algorithm's performance.

Tip: *To obtain similar statistical distribution or patterns among all three sets, we can randomly shuffle the dataset and select the samples for each set. If it is classification dataset, then make sure that samples from each class are proportionally represented in each set.*

While selecting the validation set out of the training set, we can take either a fixed validation set or randomly take out few examples from the training set in each training iteration and use these examples as validation. The latter technique is called *cross-validation* and is considered more robust in situations when the dataset size is small. Few popular cross-validation strategies are mentioned below:

- **K-fold cross validation:** Training samples are randomly partitioned into k equal-sized sets. In an iteration of training, one set is selected as validation set and remaining $k-1$ sets are considered for training. This is repeated k times where a set is considered as validation set only once. These k results are then averaged to produce single estimation. k can be any value, usually $k=10$ which is depicted in [Figure 1.17](#). In the figure, represents the cost or error associated with the iteration. 'E' represents single estimation obtained by averaging all 's. Refer to the following figure:

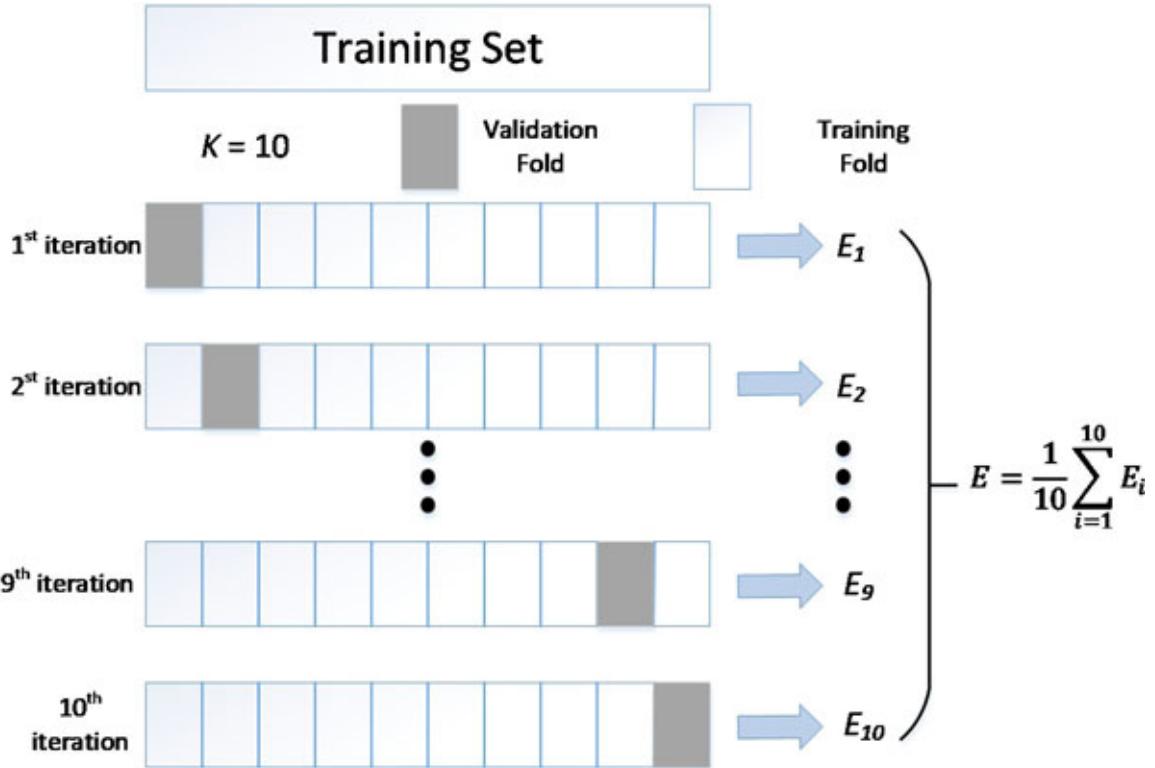


Figure 1.17: 10-fold cross-validation

- **Leave-p-out cross validation:** Out of n training set samples, this method uses p observations as validation set and remaining $n-p$ observations as training set for one iteration of training. This is repeated on all possible C_p^n sets. p can be any value. The most popular value is $p=1$, which is called Leave-one-out cross validation.
- **Repeated random sub-sampling method:** This method is also known as Monte-Carlo cross-validation. Here, sample set is randomly split into training and validation set. Split set is used for one iteration of training. For each iteration of training, sample set is randomly split every time. Results are then averaged to produce single estimation. Number of iterations will not depend on sample set size. In this method, it may happen that a few samples may never be selected for validation set, and a few samples may end up being selected more than once.

In many situations, the dataset is not exhaustive enough to capture all variations of the real data. This leads to high performance on training and cross-validation dataset and does good even on test set, but it will perform poorly when deployed in a real environment. We should collect more samples that would reflect statistical distribution of real data.

Data augmentation is one of the techniques to make a dataset robust. *Data augmentation* technique adds more samples to the dataset by imparting slight modification to the existing dataset or synthesize new samples from the existing dataset. Modification or synthesis should be performed such a way that the label of original sample and its corresponding modified or synthesized sample should remain the same.

Techniques to augment the data depends on the nature of the data and desired output.

Consider dataset of images to recognize dog or cat. For this dataset, we can apply rotation, translation, shear, flipping techniques on the existing images. Do note that, these techniques don't change the label from original sample to transformed sample. Image containing cat will still be recognized as cat after these transformations. Few of these image augmentation techniques will be discussed in [Chapter 9, Computer Vision](#).

Application of AI

AI is being used across industries for better decision-making, increasing efficiency, and eliminating repetitive work. AI is augmenting human capacity in all fields, including healthcare, education, agriculture, automobile, finance, gamming, ecommerce, fashion design, and advanced scientific research like space exploration and particle physics. [Figure 1.18](#) depicts one application in each of these fields:

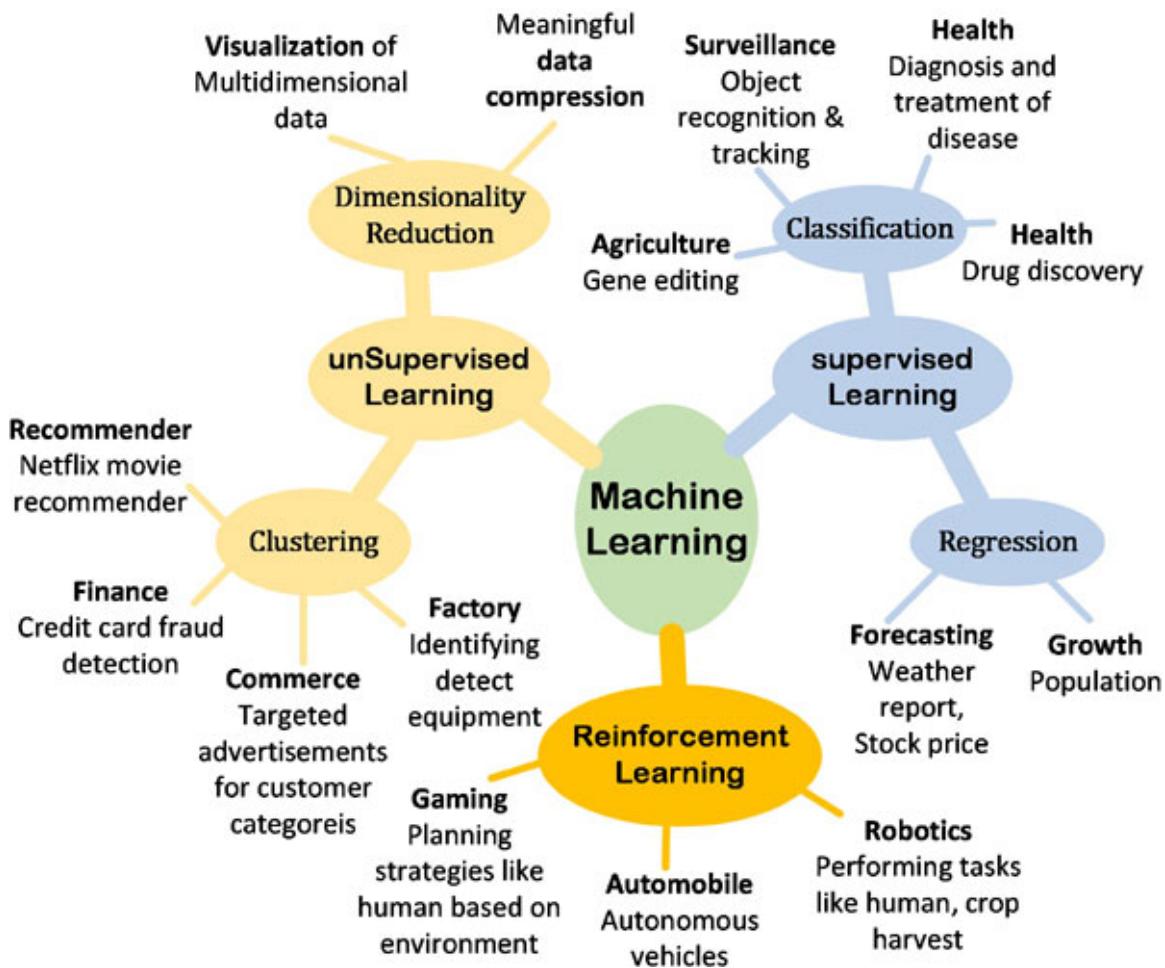


Figure 1.18: Applications of ML

All these applications of AI fall under the category of ANI or specialized AI. These AI systems rely largely on human-generated data and excel at mimicking human behaviour on well-known tasks. They also incorporate human biases as the bias is in the training data itself. These systems lack robustness, that is, the ability to perform consistently under changing circumstances. Moreover, these systems often have the problem of explainability, that is, we are unable to explain why a decision is taken by the system under a given circumstance. These problems open up new frontiers for research, the ultimate goal being AGI, which experts agree is far in the future.

Role of Mathematics in AI

The goal of AI is to design algorithms that can perform *data-based automated decision-making under uncertainty*. Data is the starting point, and this data is always insufficient. It's never possible to capture all possible scenarios in any

dataset, and if we can, then there is no need for AI. We don't need AI for writing an algorithm that can compute the sum of any two numbers, as we know all possible scenarios that can come and have rules for all of them. Insufficiency in data is a primary source of *uncertainty*, that is, working with imperfect or incomplete information.

Other sources of uncertainty are noise in data, errors while collecting data, and assumptions made while modelling. We can represent this uncertainty qualitatively with the mathematical theory of probability and statistics. Probability provides the foundation and tools for quantifying, handling, and harnessing uncertainty. Statistics deals with the methods of collecting, presenting, analysing, interpreting, and inferencing from data. Data is represented numerically as a point in high-dimensional space called vector space. However, beyond three dimensions, we cannot visualize data; thus, every observation collected is an abstract numerical object. Linear algebra provides us with all the tools to operate with these abstract objects called vectors and also define concepts of similarity, distance, and angle between these vectors.

With all these tools, we are equipped to mathematically define decision-making, which is required to automate decision-making from data, that is, to achieve the final goal of AI. These decisions can be of two types: discrete or continuous. Discrete decisions are like classification or deciding an action in a RL scenario, and continuous decisions are like regression.

Mathematically, discrete decisions can be represented as a way of partitioning the high dimensional space where the data points lie and assigning a category to each partition. Continuous decisions, on the other hand, are some functions mapping a point in high dimensional space to a real number. In both cases, a set of parametric mathematical functions must be found that can output the best possible decisions. To do this, we need tools for function optimization in high-dimensional space, and this is given by the theory of vector calculus. These four mathematical tools, i.e., Linear algebra, Vector calculus, Probability, and Statistics, are the four pillars of AI, depicted in [Figure 1.19](#). Each of these topics are vast, and it is not necessary to gain completer mastery on these topics to understand the theory of AI. In this book, we have presented the essential concepts from these topics required to get a good in-depth understanding of AI. Refer to the following figure:

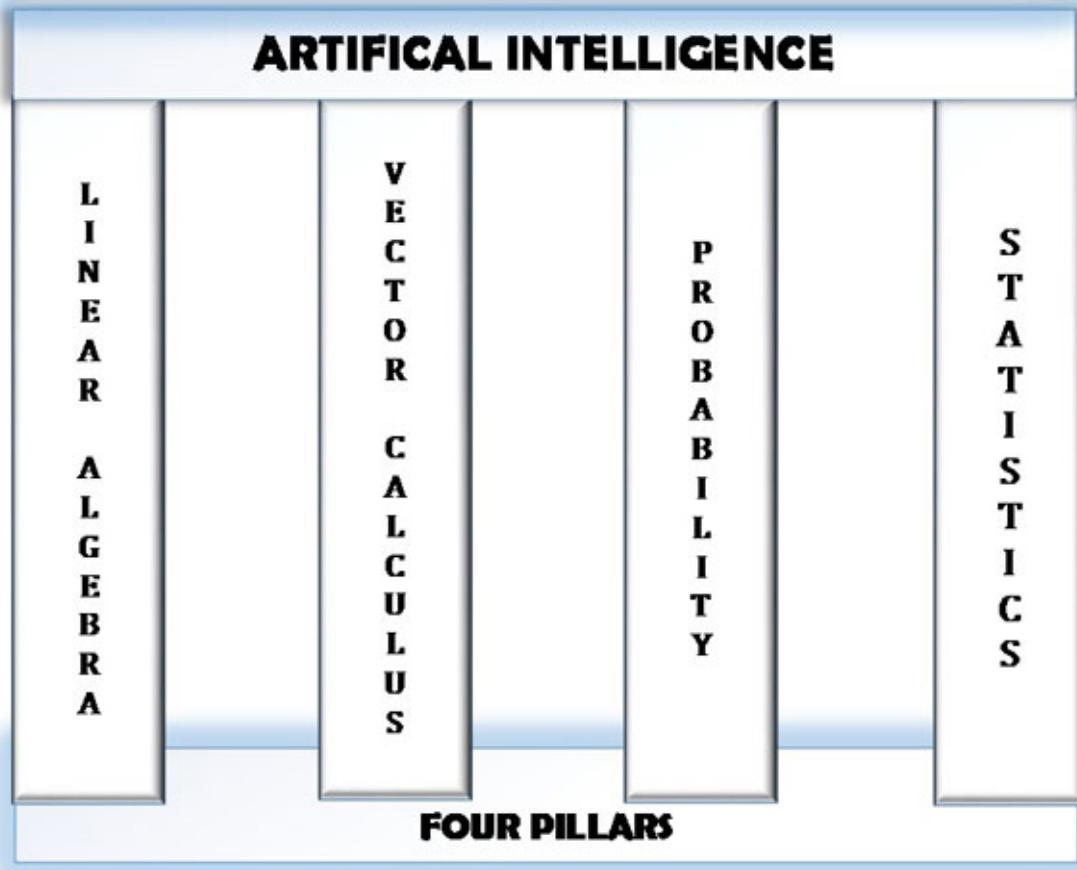


Figure 1.19: Pillars of AI

As the era of AI is still in the initial stages, there is a huge scope for all of us to contribute to this field. These mathematical tools are the foundation of the technology that is already in use and all that is yet to come. Having a deeper understanding of these mathematical basics will help the reader become a successful contributor to the next generation AI technologies and appropriately use the existing technology.

Conclusion

In this chapter, we provided a high-level overview of AI and discussed various types of algorithms and the challenges in AI. The next four chapters will be on the four foundational mathematical pillars of AI. [Chapter 6, Neural Networks](#)

will discuss about deep learning – the core of most of the state-of-the-art ANI components. After that, various ANI topics will be covered in the remaining chapters. These chapters will be based on the theory discussed in first six chapters. We strongly encourage the reader to go through [chapters 1-6](#) first, and the remaining chapters can be read in any order.

CHAPTER 2

Linear Algebra

Linear algebra is a branch of mathematics dealing with vectors and linear functions on vectors. A vector is a representation of an abstract object as a mathematical entity. As we can add, subtract, and multiply numbers, we can do similar operations with two or more vectors to get a new vector. For example, a digital image can be represented as a vector of pixels. Let's take two digital images shot from the same camera position. The first is of a lady in front of her house, and the next is of a car with the same background as the lady's image. If we add these two images, we get a new image with the lady along with a car in front of her house. Many such image operations can be represented with vector operations and transformation of vectors.

In machine learning, representation of an abstract object as a vector is the first challenge, called *feature engineering*. Traditionally, this process used to be completely manual and was time-consuming. Deep learning partially automates this task. Internally, deep learning uses the power of linear algebra, vector calculus, and optimisation techniques to achieve this. We will cover a small example of linear neural network in this chapter itself once we introduce linear transformations of vectors.

Linear algebra equips us with mathematical tools to represent abstract problem statements from various domains in a crisp, organized, and formal notation. The simplest mathematical representation of a problem is possibly done with a linear equation. Often, we need more than one linear equation to represent a problem, and we call it a system of linear equations. We will start with system of linear equations and see how vector representation help study solutions of system of equations with large number of unknowns.

Structure

The following topics of linear algebra will be discussed in this chapter:

- Linear equation

- Matrices
- Euclidean space
- Vector spaces
- Linear transformation
- Eigen values and eigen vectors
- Matrix decomposition

Objectives

After studying this chapter, you should be able to learn the basics of Linear algebra that are essential for the development of AI algorithms. The chapter contains code and examples that will help readers apply the concepts on real data.

Linear equations

A linear equation with two variables x, y represents a straight line: $y = mx + c$, where m is called the slope and c is called the intercept of the line. Here, slope m controls the angle ($slope = \tan(angle)$) the line makes with the y-axis, and the intercept c tells us where the line intersects the y-axis. The intercept represents the value of y when $x = 0$. We encounter such equations in our day-to-day life. For example, we use the following linear equation to convert temperature measured in Celsius(C) scale to Fahrenheit(F):

$$F = \frac{9}{5}C + 32,$$

Refer to [Figure 2.1](#):

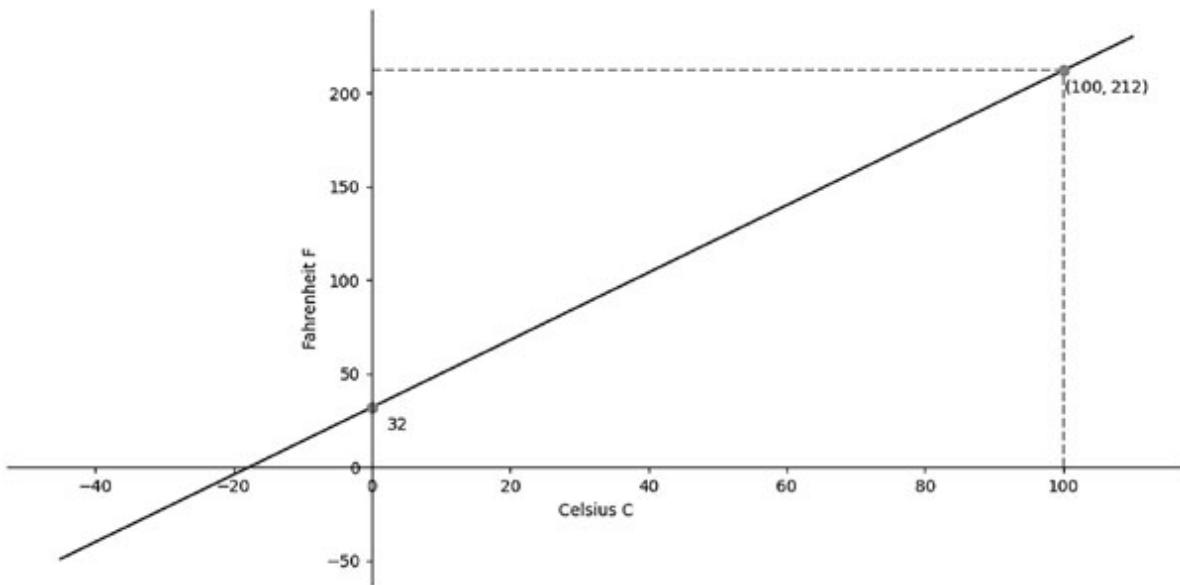


Figure 2.1: Line represents mapping between Celsius and Fahrenheit

Equations of first order are called linear equations. Rewriting the linear equation in the form $9C - 5F + 160 = 0$ results in general form $ax + by + c = 0$. This is called the general form of linear equation as we can easily extend this for more than two variables as $a_1x_1 + a_2x_2 + \dots + a_nx_n + a_0 = 0$. The constant a_i is called the coefficient and a_0 is called intercept of the equation. A linear equation in three variables ($a_i \neq 0; 1 \leq i \leq 3$) geometrically represents a hyperplane or plane in three dimensions. The idea of hyperplanes can be extended to n -dimensions.

When more than one linear equation is represented in n -dimensional space, it's interesting to analyse common points or intersection points of these hyperplanes represented by these equations. These common points lie on all the hyperplanes simultaneously and are known as *solutions* of the **system of linear equations**.

Now, let's formulate a simple problem as system of linear equations and find a solution.

Consider a situation where a group of friends plan to visit a shopping mall. They plan to spend time on movie, bowling, and play station. With difference of opinion on where to start, they get divided into three groups. After spending time in the mall, they all gather at one place for discussion.

The first group $G1$ mentions that they spent ₹1500 on 1 *bowling alleys*, 1 *play stations* and 1 *movie tickets*. The second group $G2$ spent ₹4400 on 3

bowling alleys, 4 *play stations* and 2 *movie tickets*. The third group G3 spent ₹6500 on 5 *bowling alleys*, 3 *play stations* and 5 *movie tickets*. With this information, can the price of a bowling alley, play station, and movie ticket be derived? One can represent preceding data in equations format as follows:

$$G1: 1 \text{ bowling alleys} + 1 \text{ play stations} + 1 \text{ movie tickets} = ₹1500$$

$$G2: 3 \text{ bowling alleys} + 4 \text{ play stations} + 2 \text{ movie tickets} = ₹4400$$

$$G3: 5 \text{ bowling alleys} + 3 \text{ play stations} + 5 \text{ movie tickets} = ₹6500$$

Representing system of linear equations by replacing *bowling alleys* with b , *play stations* with p and *movie tickets* with m , one obtains 3 linear equations:

$$e_1: 1b + 1p + 1m = 1500$$

$$e_2: 3b + 4p + 2m = 4400$$

$$e_3: 5b + 3p + 5m = 6500$$

Let's visualize geometrical representation of these equations in 3D with x-axis for *bowling alley*, y-axis for *play station*, and z-axis for *movie ticket*. Each equation will be a plane in 3D space. Consider equation $1b + 1p + 1m = 1500$ for representing in 3D space. Variable b & p can be free running with positive values (cost of bowling, play station & movie ticket should be positive), but value of m will be assigned by using equation e_1 as $m = 1500 - b - p$. These values representing equations will span a plane in 3D, as shown in [Figure 2.2 \(Left\)](#):

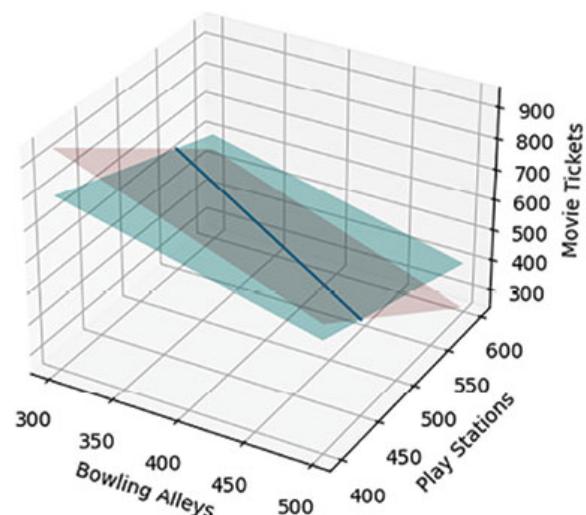
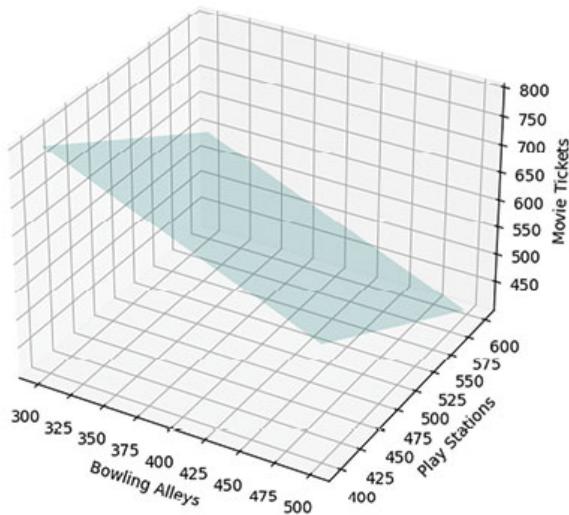


Figure 2.2: (Left) A plane representing equation $1b+1p+1m=1500$ (Right) Add plane representing $3b+4p+2m=4400$ to left figure, two planes intersect along a line

Similarly, a plane representing equation e_2 is plotted in 3D. In this case, planes representing e_1 & e_2 intersect along the line in 3D, whose line can be represented with equations $b + 2p = 1400$ & $m = p + 100$, as shown in [Figure 2.2](#) (Right). Intersecting line can be plotted by making b as free running variable, and other dependent variables are assigned with $p = (100 - b)/2$ & $m = p + 100$. This signifies that all points that lie on the intersection line will also lie on both the planes.

In the same way, a plane representing equation e_3 could be plotted in 3D, as shown in [Figure 2.3](#). These three planes fortunately intersect at a unique point $p(400,500,600)$. This signifies that the intersection point p lie on all three planes representing each of the three equations. Solution for these set of equations is $(400,500,600)$, which signifies cost of a *bowling alley*(b) is ₹ 400, *play station*(p) is ₹ 500 and *movie ticket*(m) is ₹600. The solution obtained for system of equations is through geometric way.

In this example, each equation representing a plane intersected at a unique point. This is the case of *unique solution*. This need not be the situation always. If planes intersect along a single line or hyperplane, all points that lie on the intersecting line or hyperplane are solutions to the equations representing these planes. This provides the case of *infinitely many solutions* or *infinite solutions*. There can be another situation where planes do not intersect at any point, providing the case of *no solution* or *inconsistent systems*. A system of linear equations that do not have solutions is called an *inconsistent systems*. Further discussions will revolve around these categories of solutions and analytical method to obtain these solutions. Refer to the following figure:

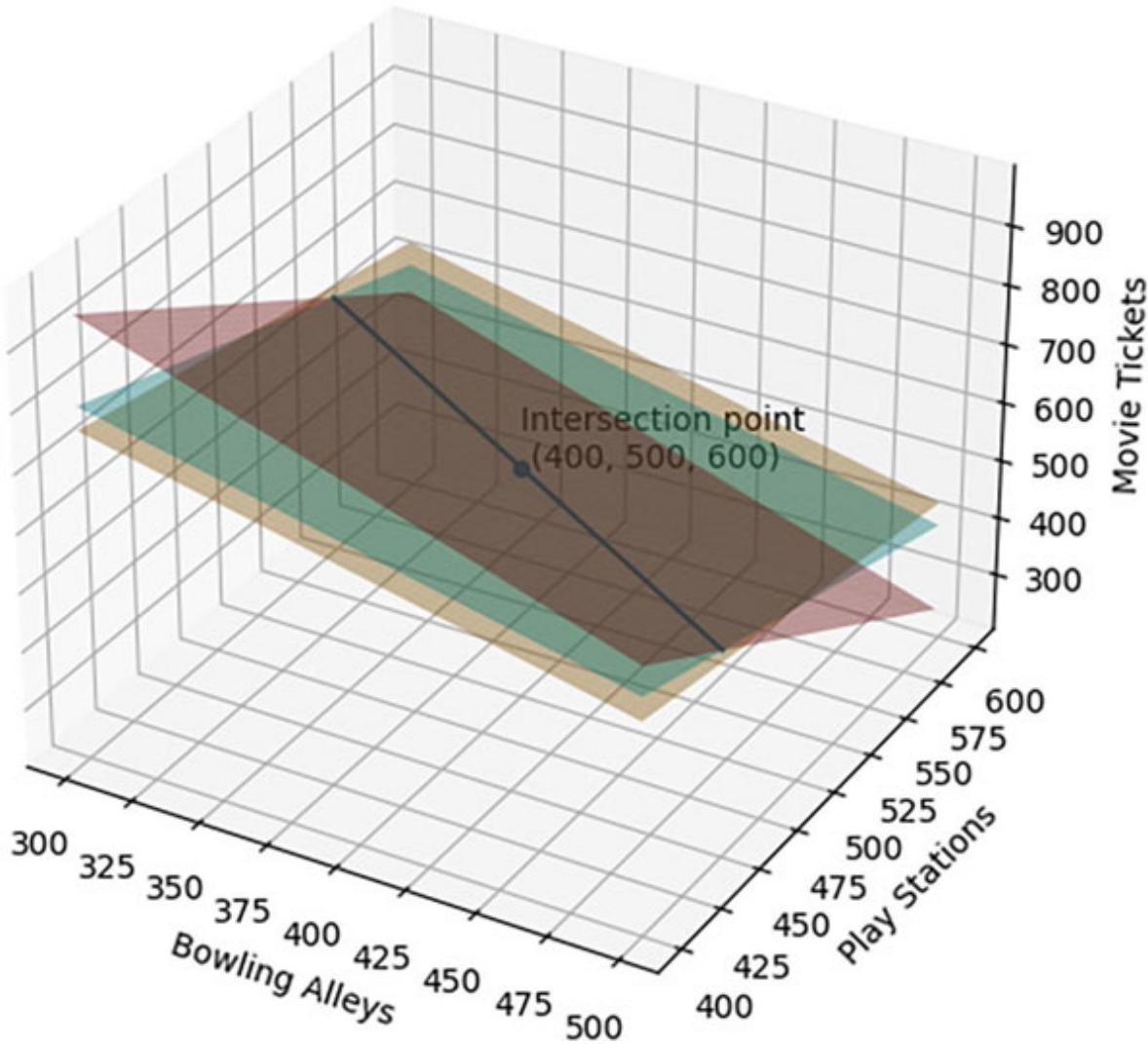


Figure 2.3: Intersection of three planes representing three equations at a unique point (400,500,600)

Solving system of equations analytically

Obtaining solution to the equations geometrically is not the preferred way due to difficulty in visualizing the solution. Consider a linear equation of four or more variables; visualizing hyperplanes formed by these equations is not possible. Let us discuss a generic approach to find the solution for system of equations. Analytically, one can perform series of valid operations on these equations to obtain the same solution. These operations include multiplication and addition.

Multiplication: Multiplying the same non-zero real number on the left-hand side (LHS) and the right-hand side (RHS) of the equation doesn't alter the

equality of the equation. For example, equation e_1 multiplied by non-zero real value 3, operation represented as $3e_1 \rightarrow e_1$ says that equation is multiplied by e_1 on both sides and the resulting equation is called e_1 :

$$\text{Equation } e_1: 1b + 1p + 1m = 1500$$

$$\text{Multiplication: } 3^*e_1 \rightarrow e_1$$

$$\text{Resulting Equation } e_1 : 3b + 3p + 3m = 4500$$

Addition: Adding the same real number on the left-hand side (LHS) and the right-hand side (RHS) of the equation doesn't alter the equality of the equation. One can also add equality equations as LHS are RHS are equal. For example, add e_2 to e_1 , and the resulting equation will be assigned to e_1 , operation represented as $e_1 + e_2 \rightarrow e_1$:

$$\text{Equations } e_1 : 1b + 1p + 1m = 1500, e_2 : 3b + 4p + 2m = 4400$$

$$\text{Addition: } e_1 + e_2 \rightarrow e_1$$

$$\text{Resulting equation } e_1 : 4b + 5p + 3m = 5900$$

To obtain solution for the equations, let's perform series of multiplication and addition on these three systems of linear equation. Solution obtained through geometric and analytic methods are same.:

$$\begin{array}{l} e_1: 1b + 1p + 1m = 1500 \\ e_2: 3b + 4p + 2m = 4400 \\ e_3: 5b + 3p + 5m = 6500 \end{array} \xrightarrow{\begin{array}{l} e_2+(-3)*e_1 \rightarrow e_2 \\ e_3+(-5)*e_1 \rightarrow e_3 \end{array}} \begin{array}{l} e_1: 1b + 1p + 1m = 1500 \\ e_2: 0b + 1p - 1m = -100 \\ e_3: 0b - 2p + 0m = -1000 \end{array} \xrightarrow{\begin{array}{l} e_1+(-1)*e_2 \rightarrow e_1 \\ e_3+2*e_2 \rightarrow e_3 \end{array}}$$

$$\begin{array}{l} e_1: 1b + 0p + 2m = 1600 \\ e_2: 0b + 1p - 1m = -100 \\ e_3: 0b + 0p - 2m = -1200 \end{array} \xrightarrow{(-1/2)*e_3 \rightarrow e_3} \begin{array}{l} e_1: 1b + 0p + 2m = 1600 \\ e_2: 0b + 1p - 1m = -100 \\ e_3: 0b + 0p + 1m = 600 \end{array} \xrightarrow{\begin{array}{l} e_1+(-2)*e_3 \rightarrow e_1 \\ e_2+e_3 \rightarrow e_2 \end{array}}$$

$$\begin{array}{l} e_1: 1b + 0p + 0m = 400 \\ e_2: 0b + 1p + 0m = 500 \\ e_3: 0b + 0p + 1m = 600 \end{array} \xrightarrow{\text{yields}} b = 400, p = 500, m = 600$$

Note: What if friends had divided themselves into two groups to play three games? In this case, a unique solution doesn't exist as one would

require at least three linear equations for three unknowns. This case might result in either infinitely many solutions or no solution.

Infinitely many solutions

Consider modification to preceding example with the following set of equations:

$$e_1: 1b + 1p + 1m = 1500$$

$$e_2: 3b + 4p + 2m = 4400$$

$$e_3: 2b + 2p + 2m = 3000$$

Let's plot planes representing equations e_1 & e_3 in 3D for geometric analysis. One can visualize that these two planes overlap each other, as shown in [Figure 2.4 \(Left\)](#). In other words, these two planes are the same but are represented with different equations. If planes are the same, then can one obtain equation e_3 from e_1 ? The answer is yes, equation e_3 can be obtained from e_1 analytically by multiplication as $2 * e_1 \equiv e_3$. To find the solution to these equations, plot plane representing e_2 . As equations e_1 & e_3 are the same, let's use only e_1 for plotting with e_2 , as shown in [Figure 2.4 \(Right\)](#). One can visualize that planes representing e_1 & e_2 intersect along the line whose equation is $b + 2p = 1400$ & $m = p + 100$. Solution to this example of system of equations is $b = -2m + 1500$ & $p = m - 100$, where m is free running variable whose value will be in range (100,750) so that the value of b & p can be positive. Refer to the following figure:

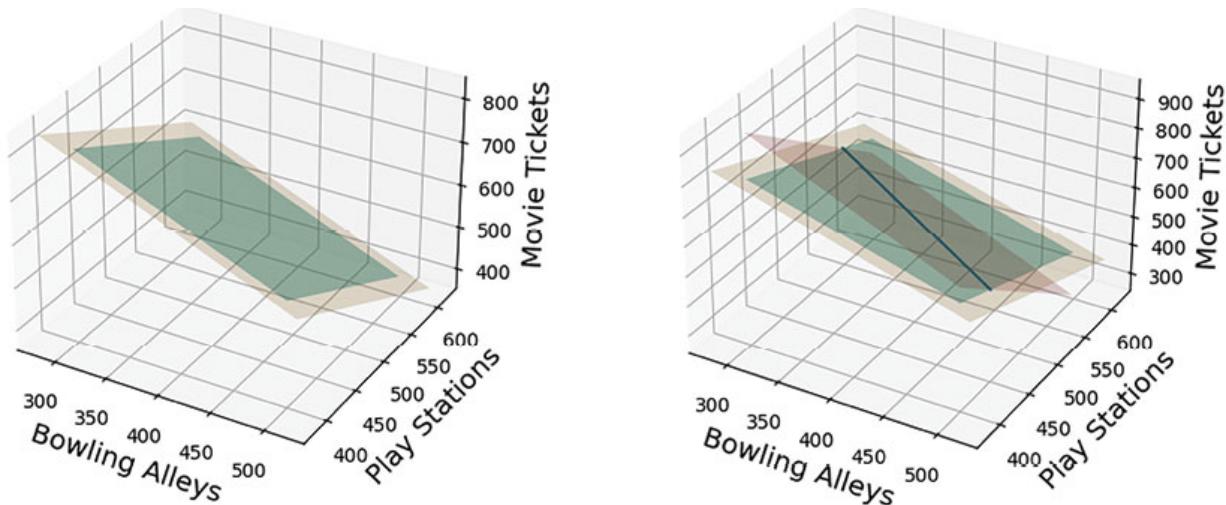


Figure 2.4: (Left) Overlapping of two planes representing two equations. Points that form both planes are same (Right) Three planes intersecting along a line. All points of the line lie on all three planes.

Apply multiplication and addition to the system of equations for finding the solution:

$$\begin{array}{l}
 e_1: 1b + 1p + 1m = 1500 \\
 e_2: 3b + 4p + 2m = 4400 \\
 e_3: 2b + 2p + 2m = 3000
 \end{array}
 \xrightarrow[e_3+(-2)*e_1 \rightarrow e_3]{e_2+(-3)*e_1 \rightarrow e_2}
 \begin{array}{l}
 e_1: 1b + 1p + 1m = 1500 \\
 e_2: 0b + 1p - 1m = -100 \\
 e_3: 0b + 0p + 0m = 0
 \end{array}
 \xrightarrow[e_1+(-1)*e_2 \rightarrow e_1]{}$$

$$\begin{array}{l}
 e_1: 1b + 0p + 2m = 1600 \\
 e_2: 0b + 1p - 1m = -100 \\
 e_3: 0b + 0p + 0m = 0
 \end{array}
 \text{ yields } b + 2m = 1600, p - m = -100$$

As cost of *bowling alley*(b), *play station*(p) and *movie ticket*(m) are positive; this restricts the range of *movie ticket* to $m > 100$. Variables b and p can have various positive values based on equation $b + 2m = 1600$, $p - m = -100$ with $m > 100$ & $m < 750$. Choosing any value of m between this range can get b and p . Hence, there exists *infinitely many solutions* for this system of equations.

Inconsistent system

There is no guarantee that a solution exists for all variations of system of linear equations. Consider modifying the preceding example with the following system of equations:

$$\begin{aligned}
 e_1: 1b + 1p + 1m &= 1500 \\
 e_2: 3b + 4p + 2m &= 4400 \\
 e_3: 2b + 2p + 2m &= 2000
 \end{aligned}$$

To visualize this geometrically, let's plot planes representing e_1 & e_2 . Planes representing these two equations are parallel and never intersect, as shown in [Figure 2.5 \(Left\)](#). Now, add a plane representing e_3 . One can see, in [Figure 2.5 \(Right\)](#), that all three planes do not intersect at a common point. Plane representing e_2 intersects planes representing e_1 & e_3 but at different points. Hence, no solution exists for these equations. These equations form *inconsistent system*. Refer to the following figure:

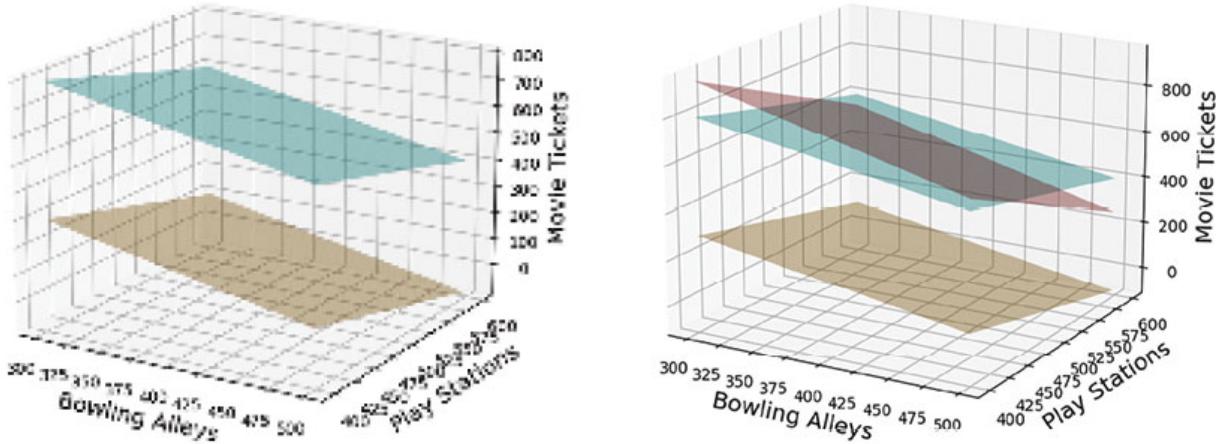


Figure 2.5: (Left) Two planes that are parallel and never intersect **(Right)** Plane representing intersects other two parallel planes but not at any common point

Apply multiplication and addition to the system of equations for finding solution:

$$\begin{array}{l}
 \begin{array}{l}
 e_1: 1b + 1p + 1m = 1500 \\
 e_2: 3b + 4p + 2m = 4400 \\
 e_3: 2b + 2p + 2m = 2000
 \end{array}
 \xrightarrow{\begin{array}{l} e_2+(-3)*e_1\rightarrow e_2 \\ e_3+(-2)*e_1\rightarrow e_3 \end{array}}
 \begin{array}{l}
 e_1: 1b + 1p + 1m = 1500 \\
 e_2: 0b + 1p - 1m = -100 \\
 e_3: 0b + 0p + 0m = -1000
 \end{array}
 \xrightarrow{e_1+(-1)*e_2\rightarrow e_1}
 \end{array}$$

$$\begin{array}{l}
 e_1: 1b + 0p + 2m = 1600 \\
 e_2: 0b + 1p - 1m = -100 \\
 e_3: 0b + 0p + 0m = -1000
 \end{array}
 \xrightarrow{\text{yields}}
 \text{no solution as } 0b + 0p + 0m \neq -1000; b, p, m \in \mathbb{R}$$

Tip: It is not always necessary to solve the equations to know whether there exists unique or infinite or no solution.

So far, our discussion revolved around solving linear equations of three variables analytically and geometrically. As number of variables and equations increase, it becomes difficult to solve them using any of these methods. Also, it's hard to automate these operations unless we have an approach that generalizes to linear equations with very large number of variables and equations. One needs a succinct representation of linear equations to deal with very large number of equations and variables, and this is provided by *matrices*.

Introducing matrix

Matrix is a rectangular array of numbers for which operations such as addition and multiplication are defined. The horizontal and vertical lines of entries in a matrix are called rows and columns, respectively. The size of a matrix is defined by the number of *rows* and *columns* that it contains. A matrix with m *rows* and n *columns* is called $m \times n$ *matrix*, or m -by- n *matrix*, while m and n are called its dimensions. Each entry is indexed with row and column numbers as a_{xy} , where x represents row number and y is column number.

$$A_{m \times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

A matrix with the same number of rows and columns; $m = n$ is called a *square matrix*, represented as A_m . A matrix whose entries are only real numbers is called *real matrix*. Most of the matrix operations in this book will be concentrating on real matrices.

Augmented matrix

While representing equations in matrix form, each equation takes one row of the matrix. Coefficient of variables from the equation is represented left side of the row, followed by the vertical line and then the RHS of equation, which is a real number, is specified after the vertical line. Similarly, all equations are represented, where each row of the matrix represents one equation. This matrix form of representing system of linear equations is called *augmented*

matrix. Augmented matrix representation of example with equations $e_1: 1b + 1p + 1m = 1500$, $e_2: 3b + 4p + 2m = 4400$ & $e_3: 5b + 3p + 5m = 6500$ is:

$$A = \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1500 \\ 3 & 4 & 2 & 4400 \\ 5 & 3 & 5 & 6500 \end{array} \right]$$

For solving the system of equations analytically, the operations performed on equations previously are applied on rows of the augmented matrix as each row represents one equation. There are three types of elementary row operations that may be performed on the rows of a matrix. These row operations do not change the solution of the underlying system of linear equations. These elementary operations will help to represent augmented matrix in a form that will facilitate finding a solution to the system of linear equations. Rows r_1 , r_2 and r_3 represent first, second and third rows of the matrix, respectively. A few elementary operations performed on augmented matrix that do not alter the solution are explained below:

- Swap two rows:

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 1500 \\ 3 & 4 & 2 & 4400 \\ 5 & 3 & 5 & 6500 \end{array} \right] \xrightarrow{\text{swap } r_1 \& r_2} \left[\begin{array}{ccc|c} 3 & 4 & 2 & 4400 \\ 1 & 1 & 1 & 1500 \\ 5 & 3 & 5 & 6500 \end{array} \right]$$

- Multiply a row by a non-zero real number:

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 1500 \\ 3 & 4 & 2 & 4400 \\ 5 & 3 & 5 & 6500 \end{array} \right] \xrightarrow{2*r_1 \rightarrow r_1} \left[\begin{array}{ccc|c} 2 & 2 & 2 & 3000 \\ 3 & 4 & 2 & 4400 \\ 5 & 3 & 5 & 6500 \end{array} \right]$$

- Add to one row a scalar multiple of another

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 1500 \\ 3 & 4 & 2 & 4400 \\ 5 & 3 & 5 & 6500 \end{array} \right] \xrightarrow{r_3 - 2*r_1 \rightarrow r_3} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1500 \\ 1 & 2 & 0 & 1400 \\ 5 & 3 & 5 & 6500 \end{array} \right]$$

Let's reconsider the example over which series of multiplication and addition steps were performed previously that resulted in unique solution. From those series of steps, consider a system of linear equations that resulted

after the application of two sets of operations to be represented in matrix format:

$$\begin{array}{l} e_1: 1b + 0p + 2m = 1600 \\ e_2: 0b + 1p - 1m = -100 \\ e_3: 0b + 0p - 2m = -1200 \end{array} \xrightarrow{\text{matrix representation}} \left[\begin{array}{ccc|c} 1 & 0 & 2 & 1600 \\ 0 & 1 & -1 & -100 \\ 0 & 0 & -2 & -1200 \end{array} \right]$$

The resulting matrix is said to be in row echelon form. Matrix is said to be in *row echelon* form if:

- All rows that consist of only zero values are at the bottom of the matrix
- The leading coefficient (leftmost non-zero entry) of a non-zero row is always to the right of the leading coefficient of the row above it:

$$\left[\begin{array}{cccc} a_1 & a_2 & 0 & a_3 \\ 0 & 0 & a_4 & 0 \\ 0 & 0 & 0 & a_5 \\ 0 & 0 & 0 & 0 \end{array} \right] \text{ where } a_i \neq 0, a_i \in \mathbb{R}$$

The final set of equations obtained before the solution is stated below, along with matrix representation. This form of matrix is called reduced row echelon form:

$$\begin{array}{l} e_1: 1b + 0p + 0m = 400 \\ e_2: 0b + 1p + 0m = 500 \\ e_3: 0b + 0p + 1m = 600 \end{array} \xrightarrow{\text{matrix representation}} \left[\begin{array}{ccc|c} 1 & 0 & 0 & 400 \\ 0 & 1 & 0 & 500 \\ 0 & 0 & 1 & 600 \end{array} \right]$$

Matrix is said to be in *reduced row echelon* form (also called row canonical form) if:

- Matrix is in row echelon form
- Leading entry in all non-zero rows is 1
- Each column containing leading 1 has zeros in all other entries

$$\left[\begin{array}{cccc} 1 & a_2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array} \right] \text{ where } a_2 \neq 0, a_2 \in \mathbb{R}$$

Let's apply row operations on the augmented matrix to obtain row echelon form matrix. Application of row operations on the matrix to convert it to row echelon form is called *forward substitution*, where r_x row is used to modify row r_y such that $x < y$:

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 1500 \\ 3 & 4 & 2 & 4400 \\ 5 & 3 & 5 & 6500 \end{array} \right] \xrightarrow{\substack{r_2 + (-3)*r_1 \rightarrow r_2 \\ r_3 + (-5)*r_1 \rightarrow r_3}} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1500 \\ 0 & 1 & -1 & -100 \\ 0 & -2 & 0 & -1000 \end{array} \right] \xrightarrow{r_3 + 2*r_2 \rightarrow r_3} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1500 \\ 0 & 1 & -1 & -100 \\ 0 & 0 & -2 & -1200 \end{array} \right]$$

Further reduction of rows can be performed through row operation where r_x row is used to modify row r_y such that $x > y$ to obtain matrix in reduced row echelon form. This operation is called *back substitution*:

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 1500 \\ 0 & 1 & -1 & -100 \\ 0 & 0 & -2 & -1200 \end{array} \right] \xrightarrow{\substack{r_1 - r_2 \rightarrow r_1 \\ (-1/2)*r_3 \rightarrow r_3}} \left[\begin{array}{ccc|c} 1 & 0 & 2 & 1600 \\ 0 & 1 & -1 & -100 \\ 0 & 0 & 1 & 600 \end{array} \right] \xrightarrow{\substack{r_1 - 2*r_3 \rightarrow r_1 \\ r_2 + r_3 \rightarrow r_2}} \left[\begin{array}{ccc|c} 1 & 0 & 0 & 400 \\ 0 & 1 & 0 & 500 \\ 0 & 0 & 1 & 600 \end{array} \right]$$

The first column of the augmented matrix represented coefficient of variable b , so the first row of matrix gives the value of *bowling alley(b)* as ₹400. Similarly, from the second and third rows, the value of *play station(p)* is ₹500 and that of *movie ticket(m)* is ₹600. This process of using row operations on a matrix to obtain row echelon form and then further reducing it to reduced row echelon form is called *Gauss-Jordan Elimination method*.

Pseudocode forward substitution

Consider matrix $A_{m \times n}$, whose i^{th} row and j^{th} column entry is accessed through $A[i, j]$. The following code provides pseudocode for forward substitution. This stage would result in upper triangular matrix.

1. for $\text{row_idx} = 1$ to m do
2. $\text{pivot_row} = \text{select_pivot_from_ref}(\text{row_idx})$
3. $\text{swap_rows}(\text{pivot_row}, \text{row_idx})$
4. $\text{normalize_row}(\text{row_idx})$
5. for $\text{tr_row} = \text{row_idx} + 1$ to m do
6. for $\text{tr_col} = \text{row_idx}$ to n do
7. $A[\text{tr_row}, \text{tr_col}] = A[\text{tr_row}, \text{tr_col}] - A[\text{row_idx}, \text{tr_col}] * A[\text{tr_row}, \text{row_idx}]$
8. end for
9. end for

10. end for

Pseudocode back substitution

Apply back substitution on upper triangular matrix to obtain a diagonal matrix. The following code provides steps for back substitution:

```
1. for row_idx = m to 1 do
2.   normalize_row(row_idx)
3.   for tr_row = row_idx - 1 to 1 do
4.     for tr_col = row_idx to n do
5.       A[tr_row, tr_col] = A[tr_row, tr_col]-
A[row_idx, tr_col]*A[tr_row, row_idx]
6.   end for
7. end for
8. end for
```

We now have an algorithm to solve any system of linear equations with a large number of variables using a sequence of matrix operations. Similar to operations like add, subtract, multiply, and inverse on real numbers, one can define operations on a whole matrix as matrix can be viewed as abstract numerical object.

Basic matrix operations

Few basic operations other than row operations that can be defined are matrix addition, scalar multiplication, transposition, and matrix multiplication.

Addition of two matrices can be performed if their dimensions are equal. The sum of A_{mxn} and B_{mxn} , denoted $A + B$, is computed by adding corresponding elements of matrices A and B :

$$A_{mxn} + B_{mxn} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \cdots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \cdots & a_{2n} + b_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \cdots & a_{mn} + b_{mn} \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 3 & 1 \\ 2 & 4 & 3 \\ 4 & 5 & 1 \end{bmatrix}; B = \begin{bmatrix} 3 & 1 & 2 \\ 4 & 1 & 1 \\ 2 & 0 & 1 \end{bmatrix}; A + B = \begin{bmatrix} 1+3 & 3+1 & 1+2 \\ 2+4 & 4+1 & 3+1 \\ 4+2 & 5+0 & 1+1 \end{bmatrix} = \begin{bmatrix} 4 & 4 & 3 \\ 6 & 5 & 4 \\ 6 & 5 & 2 \end{bmatrix}$$

Product of scalar value k and matrix A_{mxn} is obtained by multiplying every entry of with scalar value .

$$kA_{mxn} = \begin{bmatrix} k * a_{11} & k * a_{12} & \cdots & k * a_{1n} \\ k * a_{21} & k * a_{22} & \cdots & k * a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ k * a_{m1} & k * a_{m2} & \cdots & k * a_{mn} \end{bmatrix}$$

$$kA = 3 \begin{bmatrix} 1 & 3 & 1 \\ 2 & 4 & 3 \\ 4 & 5 & 1 \end{bmatrix} = \begin{bmatrix} 3 * 1 & 3 * 3 & 3 * 1 \\ 3 * 2 & 3 * 4 & 3 * 3 \\ 3 * 4 & 3 * 5 & 3 * 1 \end{bmatrix} = \begin{bmatrix} 3 & 9 & 3 \\ 6 & 12 & 9 \\ 12 & 15 & 3 \end{bmatrix}$$

Transpose of matrix A_{mxn} is obtained by turning rows into columns (or columns into rows). Matrix A_{mxn} after transpose would produce matrix with mn dimensions, denoted as A^T .

$$A_{mxn} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}; A^T = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \vdots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 3 & 1 \\ 2 & 4 & 3 \\ 4 & 5 & 1 \end{bmatrix}; A^T = \begin{bmatrix} 1 & 2 & 4 \\ 3 & 4 & 5 \\ 1 & 3 & 1 \end{bmatrix}$$

Properties of transpose are:

- $(AB)^T = B^T A^T$
- $(A + B)^T = A^T + B^T$

Multiplication of two matrices $A_{m \times n}$ & $B_{m \times n}$ is defined if and only if $n = p$ (that is, number of columns of left matrix is same as number of rows of right matrix). Elements of product matrix is defined as:

$$[AB]_{i,j} = a_{i,1}b_{1,j} + a_{i,2}b_{2,j} + \cdots + a_{i,n}b_{n,j} = \sum_{r=1}^n a_{i,r}b_{r,j} \text{ where } 1 \leq i \leq m; 1 \leq j \leq q$$

$$\begin{aligned} AB &= \begin{bmatrix} 1 & 3 & 1 \\ 2 & 4 & 3 \\ 4 & 5 & 1 \end{bmatrix} \begin{bmatrix} 3 & 1 & 2 \\ 4 & 1 & 1 \\ 2 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 * \hat{3} + 3 * \hat{4} + 1 * \hat{2} & 1 * \hat{1} + 3 * \hat{1} + 1 * \hat{0} & 1 * \hat{2} + 3 * \hat{1} + 1 * \hat{1} \\ 2 * \hat{3} + 4 * \hat{4} + 3 * \hat{2} & 2 * \hat{1} + 4 * \hat{1} + 3 * \hat{0} & 2 * \hat{2} + 4 * \hat{1} + 3 * \hat{1} \\ 4 * \hat{3} + 5 * \hat{4} + 1 * \hat{2} & 4 * \hat{1} + 5 * \hat{1} + 1 * \hat{0} & 4 * \hat{2} + 5 * \hat{1} + 1 * \hat{1} \end{bmatrix} = \begin{bmatrix} 17 & 4 & 6 \\ 28 & 6 & 19 \\ 34 & 9 & 14 \end{bmatrix} \end{aligned}$$

Matrix multiplication can also be interpreted in the following way. The first column of the resultant matrix is obtained by linear combination of columns of matrix A with weights for each column is first column of matrix B . Similarly, other columns of resultant matrix can be obtained with the second and third columns for matrix B .

$$\left[3 * \begin{bmatrix} 1 \\ 2 \\ 4 \end{bmatrix} + 4 * \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} + 2 * \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix} \quad 1 * \begin{bmatrix} 1 \\ 2 \\ 4 \end{bmatrix} + 1 * \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} + 0 * \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix} \quad 2 * \begin{bmatrix} 1 \\ 2 \\ 4 \end{bmatrix} + 1 * \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} + 1 * \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix} \right]$$

Properties of multiplication: Consider dimensions of matrices A , B & C such that multiplication of matrices is defined.

- **Associativity:** $(AB)C = A(BC)$
- **Distributivity:** Satisfies both left and right distributivity:

$$(A + B)C = AC + BC \text{ & } C(A + B) = CA + CB$$

- **Non-commutative:** $AB \neq BA$. Matrix multiplication operation of AB and BA is defined only if columns count of A = rows count of B and rows count of A = columns count of B

Trace of a square matrix A_m is defined as sum of diagonal elements of a matrix.

$$\text{trace}(A) = \sum_{i=1}^m a_{i,i}$$

Properties of trace: A, B, C are matrices of appropriate dimensions that multiplication or addition is defined.

- $\text{trace}(A) = \text{trace}(A^T)$
- $\text{trace}(AB) = \text{trace}(BA)$
- $\text{trace}(A + B) = \text{trace}(A) + \text{trace}(B)$
- $\text{trace}(ABC) = \text{trace}(BCA) = \text{trace}(CAB)$

Matrices enabled us to solve set of linear equations with n unknowns. It's not possible to visualize the solution space of these system of equations in n -dimensions. Analysis of problems with three unknown variables was comparatively simple as it restricted us to three-dimensional space. We need to generalize the concepts of three-dimensional to n -dimensional space where n is positive integer. To represent a point in three-dimensional space, one requires three values that are represented on each of the dimension axis. Similarly, to represent a point p in n -dimensional space \mathbb{R}^n , one would require ordered n -tuple (x_1, x_2, \dots, x_n) where $x_i \in \mathbb{R}$ and each x_i would correspond to a value along a dimension. A set of these all ordered n -tuples form *Euclidean n-space*. Every element of this set is represented by ordered n -tuple that are called *vectors*.

Euclidean space

Euclidean space was introduced by Greek mathematician Euclid to abstract physical space around us. A linear equation in n -dimensional Euclidean space represents a hyperplane - a generalization of plane from three-dimensions to n -dimensions. Solutions to these equations are points or vectors in n -dimensional space. This section will introduce notion of length, distance, rotation, translation, angle among these n -dimensional vectors in n -dimensional Euclidean space.

Vectors and basic properties

Vector word is derived from Latin and means ‘*carrier*’. Vectors were first introduced in geometry to represent both magnitude and direction. Rules are defined for interaction of these vectors, which are mostly interpreted from understanding of the universe.

Representing vector

Vectors connects two points, it represents both magnitude and direction in space, denoted by small letter with arrow above as \vec{v} or small letter with bold as v . Vector representation in the space is a directed line from initial point to end point. If the initial point is **0** (*origin*), then the vector is called *positional vector*. Any point $P(x_1, x_2, \dots, x_n)$ in \mathbb{R}^n is represented by a positional vector ending at P .

In *Figure 2.6 (Left)*, vectors and points are plotted in 2-dimensional space \mathbb{R}^2 . Vector v connects start point $O(0, 0)$ and end point $A(3, 4)$. To obtain vector in cartesian coordinates, perform end point’s value minus the start point’s value for each dimension. Vector u would be written as $u = (3 - 0, 4 - 0) = (3, 4)$. Similarly, vector v that connects start point $A(2, 1)$ and end point $B(5, 5)$ will result in $v = (5 - 2, 5 - 1) = (3, 4)$. Refer to the following figure:

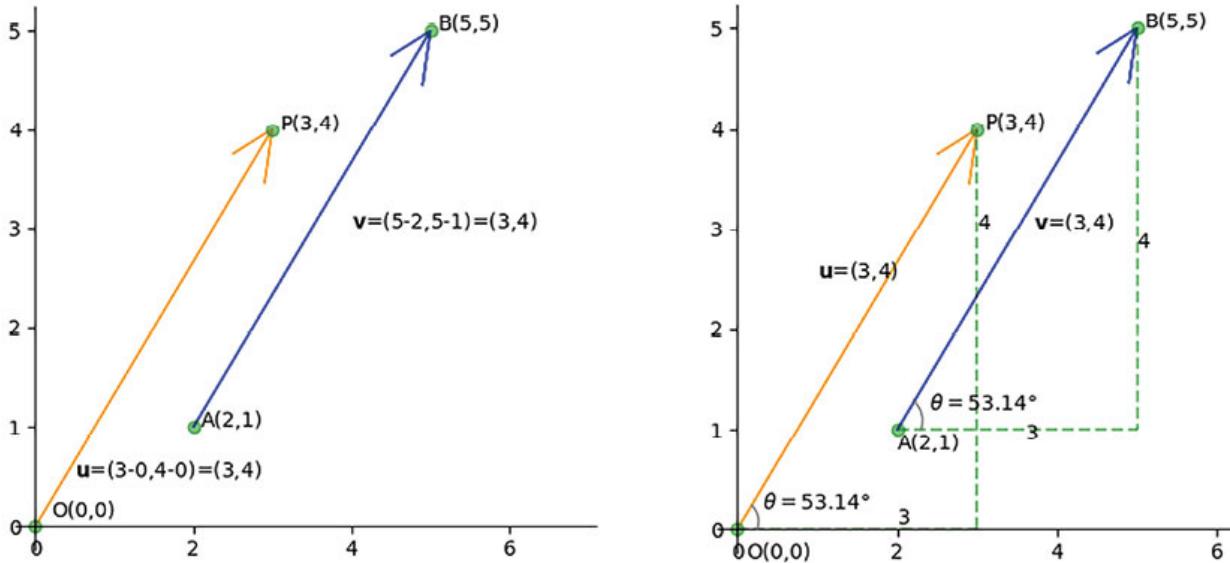


Figure 2.6: (Left) Representation of vectors that connect two points (Right) Angle induced by vector with axis

Note: Vectors in [Figure 2.6](#) are the same. Two vectors are said be equal if their magnitude and direction are the same.

Norm

Norm of a vector represents length of the vector. Also called *magnitude* of a vector can be obtained through use of Pythagoras theorem in Euclidean space. From [Figure 2.5 \(Right\)](#), length of \mathbf{v} , denoted as $\|\mathbf{v}\|$ is calculated as the consequence of the Pythagoras theorem.

$$\text{Length of hypotenuse} = \|\mathbf{v}\| = \sqrt{3^2 + 4^2} = 5$$

Similarly, norm of $\mathbf{u} = (3,4)$ is $\|\mathbf{u}\| = \sqrt{3^2 + 4^2} = 5$. Generalizing to \mathbb{R}^n , norm or length of a vector $\mathbf{v} = (a_1, a_2, \dots, a_n)$ in \mathbb{R}^n is obtained by

$$\|\mathbf{v}\| = \sqrt{a_1^2 + a_2^2 + \dots + a_n^2}.$$

Any vector \mathbf{w} is called *Unit vector* if its magnitude is 1, that is $\|\mathbf{w}\| = 1$.

In real line, we can define neighboring points of a real number a as the open interval $(a - \epsilon, a + \epsilon)$ that is, set of all real numbers within a distance from a or $x \in R : |x - a| < \epsilon$, where $\epsilon > 0$ is chosen arbitrarily small. This concept of neighbourhood of a point can be extended to higher dimensions. In two dimensions, this open interval becomes a circle of radius ϵ with centre at point $a = (x_c, y_c)$ and is defined as the set of all two dimensional points $x = (x_1, x_2)$ within the circle $(x_1 - x_c)^2 + (x_2 - y_c)^2 = \epsilon^2$ as

$$x \in R^2 : \|x - a\| < \epsilon$$

This is called *Euclidean Ball* in R^2 . In R^3 , the Euclidean ball represents a sphere or radius centered at a . In n -dimension, we call this n -dimensional **Euclidean ball**:

$$\mathbf{B}(a; r) = \{x \in R^n : \|x - a\| < \epsilon\}, \epsilon > 0$$

Direction

Direction of a vector is the angle it makes with respect to the axes of the space. In two-dimensional space, vector will make angle with both x and y axes. Generally, in n -dimensional space, vector will make angle with every n

axes of the space. Consider vector $v = (3, 4)$ from the [Figure 2.6 \(Right\)](#) that connects points from (2, 1) to (5, 5). Draw lines parallel to x and y axes passing through start point and end point of the vector respectively to form right-angled triangle with the vector as hypotenuse, as shown in [Figure 2.6](#). To find angle θ between v & x -axis, use trigonometry formula $\tan(\theta)$. In this right-angled triangle, length of the side parallel to x -axis is x -axis component of the vector, and similarly for y -axis. From [Figure 2.6](#), $\theta = \arctan(4/3) = 53.13^\circ$. Similarly, for vector u , angle with x -axis, $\theta = 53.13^\circ$.

Tip: General way to obtain angle between vectors is through dot product in Euclidean space. Dot product is explained in the next section.

In the [Figure 2.6](#), as both vectors v and u have the same magnitude and direction, they are equal, that is, $u \equiv v$.

Note: Two vectors are said to be equal if they have same magnitude and direction. This gives rise to an important notion that vectors' starting point doesn't matter, what matters is their magnitude and direction.

Scalar multiplication

Scalar multiplication is changing the magnitude or length of a vector by multiplying with real number k (called scalar) without altering its direction. Vector when multiplied by non-zero scalar value k changes its magnitude by factor of k . Multiplying with positive scalar value keeps the direction same, and the direction of the vector is flipped by an angle of 180° with a negative scalar value. Multiplying vector $v = (a_1, a_2, \dots, a_n)$ with scalar value k will result in $w = kv = k(a_1, a_2, \dots, a_n) = (k * a_1, k * a_2, \dots, k * a_n)$. In [Figure 2.7 \(left\)](#), multiply vector $u = (3, 3)$ with scalar value 2 & -1. The resulting vectors will be $w = 2u = 2 * (3, 3) = (2 * 3, 2 * 3) = (6, 6)$ and $v = (-1)u = -1 * (3, 3) = (-1 * 3, -1 * 3) = (-3, -3)$. Refer to the following figure:

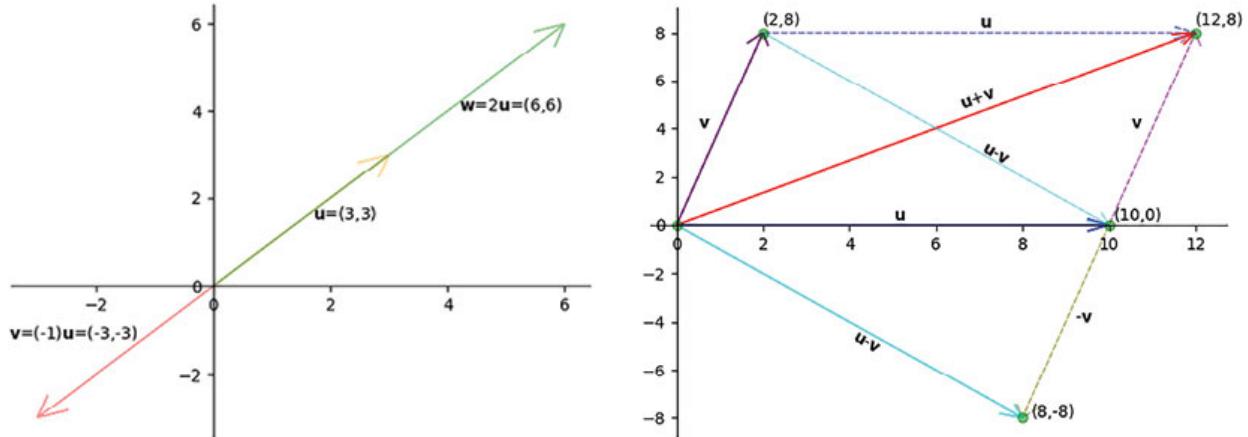


Figure 2.7: (Left) Scaling of vector (Right) Addition and Subtraction of vectors

Addition/subtraction of vectors

Addition (subtraction) of two vectors is performed by adding (subtracting) their respective components. Addition of two vectors $\mathbf{u} = (a_1, a_2, \dots, a_n)$ & $\mathbf{v} = (b_1, b_2, \dots, b_n)$ will result in:

$$\mathbf{u} + \mathbf{v} = (a_1, a_2, \dots, a_n) + (b_1, b_2, \dots, b_n) = (a_1 + b_1, a_2 + b_2, \dots, a_n + b_n)$$

Similarly, for subtraction:

$$\mathbf{u} - \mathbf{v} = (a_1, a_2, \dots, a_n) + (b_1, b_2, \dots, b_n) = (a_1 - b_1, a_2 - b_2, \dots, a_n - b_n)$$

In [Figure 2.7](#) (Right), vectors $\mathbf{u} = (10, 0)$ and $\mathbf{v} = (2, 8)$. Addition of these vectors is performed by adding the respective components, resulting in $\mathbf{u} + \mathbf{v} = (10 + 2, 0 + 8) = (12, 8)$. Subtraction of vectors is also performed by subtracting the respective components of vectors, resulting in $\mathbf{u} - \mathbf{v} = (10 - 2, 0 - 8) = (8, -8)$.

Distance between vectors

Distance between two vectors are obtained with norm over subtract operation. Distance between two vectors $\mathbf{u} = (a_1, a_2, \dots, a_n)$ & $\mathbf{v} = (b_1, b_2, \dots, b_n)$ is defined as:

$$\begin{aligned} \text{distance}(\mathbf{u}, \mathbf{v}) &= \|\mathbf{u} - \mathbf{v}\| = \|\mathbf{v} - \mathbf{u}\| = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \dots + (a_n - b_n)^2} \\ &= \sqrt{(b_1 - a_1)^2 + (b_2 - a_2)^2 + \dots + (b_n - a_n)^2} \end{aligned}$$

In [Figure 2.7](#), distance between vectors is calculated as follows:

$$\text{distance}(\mathbf{u}, \mathbf{v}) = \|\mathbf{u} - \mathbf{v}\| = \|\mathbf{v} - \mathbf{u}\| = \sqrt{8^2 + (-8)^2} = 8\sqrt{2} = 11.3$$

Dot product and orthogonality

Dot product of two vectors $\mathbf{u} = (a_1, a_2, \dots, a_n)$ & $\mathbf{v} = (b_1, b_2, \dots, b_n)$ in n dimensions denoted as $\mathbf{u} \cdot \mathbf{v}$, is defined as

$$\mathbf{u} \cdot \mathbf{v} = a_1 * b_1 + a_2 * b_2 + \dots + a_n * b_n = \|\mathbf{u}\| \|\mathbf{v}\| \cos \theta$$

where $\|\mathbf{u}\|$ & $\|\mathbf{v}\|$ denotes norm of the vectors, and θ is angle between the vectors \mathbf{u} & \mathbf{v} in n -dimensional space.

Let's analyse dot product of vectors in 2D, as shown in [Figure 2.8](#). $\|\mathbf{v}\| \cos \theta$ is magnitude of vector \mathbf{v} along the direction of vector \mathbf{u} . Dot product of vectors would be $\mathbf{u} \cdot \mathbf{v} = (10, 2) \cdot (6, 6) = 10 * 6 + 2 * 6 = 72$. One can calculate using the angle between vectors and length of the vectors as $\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\| \|\mathbf{v}\| \cos \theta = 10.2 * 8.49 * \cos(33.7^\circ) = 72$. Refer to the following figure:

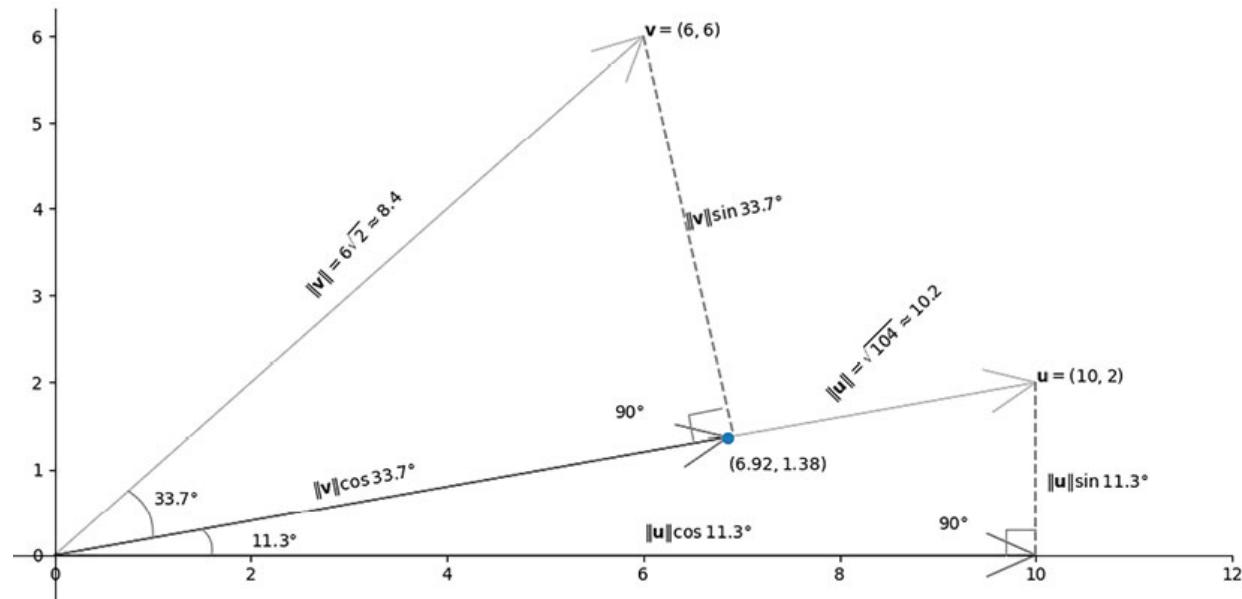


Figure 2.8: Projection of one vector over another in two-dimensional space

Vectors are said to *orthogonal* if their dot product is 0. In other words, vectors are orthogonal if the angle between them is $\theta = 90^\circ$. When θ is 90° , $\cos(\theta) = 0$ this validates dot product to 0. Any vector along x -axis is orthogonal to any vector along y -axis due to 90° angle between them. Vectors are said to be *Orthonormal* if they are orthogonal to each other and

their norm is 1. In other words, vectors \mathbf{u} & \mathbf{v} are said to be orthonormal if $\mathbf{u} \cdot \mathbf{v} = 0$ & $\|\mathbf{u}\| = \|\mathbf{v}\| = 1$.

Linear Combination of Vectors

Span of a vector \mathbf{u} is set of all possible vectors that can be obtained by performing scalar multiplication on \mathbf{u} . *Span* of more than one vector is set of all possible vectors that can be obtained by performing scalar multiplication and addition on all those vectors. From [Figure 2.9](#), span of vector $\mathbf{u} = (3,1)$ is all vectors along the line that passes through \mathbf{u} and origin:

$$\text{span}(\mathbf{u}) = k\mathbf{u} \text{ where } k \in \mathbb{R}$$

Span of two vectors $\mathbf{u} = (3,1)$ & $\mathbf{v} = (1,3)$ is calculated as:

$$\text{span}(\mathbf{u}, \mathbf{v}) = k_1 \mathbf{u} + k_2 \mathbf{v} \text{ where } k_1, k_2 \in \mathbb{R}$$

Similarly, one can generalize span of n vectors as:

$\text{span}(v_1, v_2, \dots, v_n) = k_1 v_1 + k_2 v_2 + \dots + k_n v_n$ where $k_i \in \mathbb{R}$. Refer to the following figure:

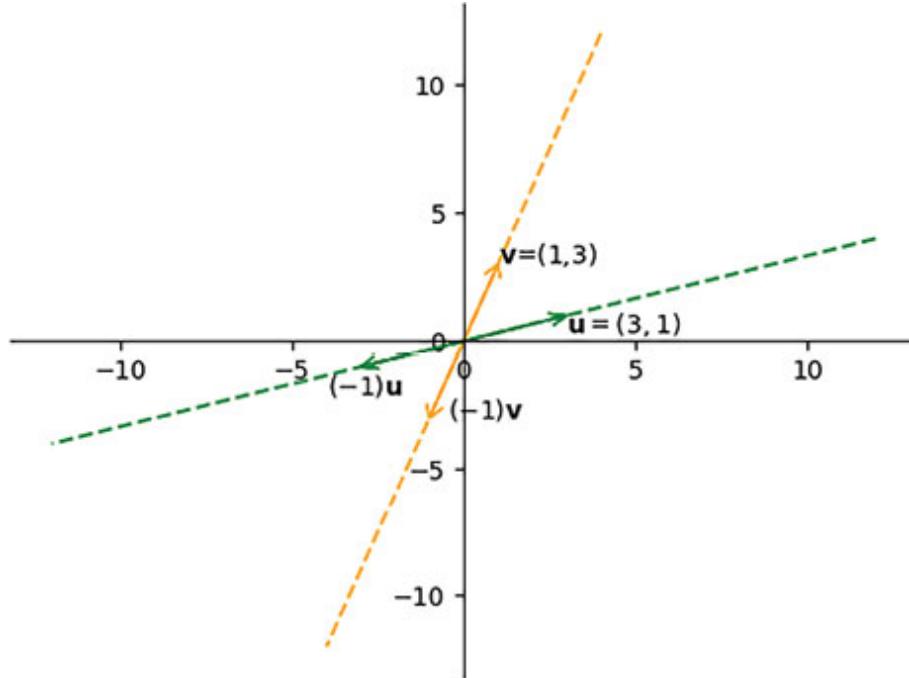


Figure 2.9: Vector spanning in two-dimensional space

One can express a vector \mathbf{u} using multiplication and addition operation on the vectors $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$. Operation of expressing a vector from scalar

multiplication and addition operation of the vectors is called *linear combination*.

$$\mathbf{u} = k_1 \mathbf{v}_1 + k_2 \mathbf{v}_2 + \cdots + k_n \mathbf{v}_n \text{ where } k_i \in \mathbb{R}$$

If one can express the vector \mathbf{u} as linear combination of vector $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$, then vector \mathbf{u} is said to be in span of vectors $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$. Set of vectors $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$ are said to be *independent* if there doesn't exist a vector in the set that can be expressed as linear combination of other vectors from the set. In other words, if there is a vector in the set that can be expressed as linear combination of other vectors in the set, then the set of vectors are called *linearly dependent vectors*.

Consider a vector set $A = \{(3,1), (1,3)\}$ in \mathbb{R}^2 from [Figure 2.9](#). Is the vector set A independent? Yes, there doesn't exist any vector that can be expressed as linear combination of other vectors. Now, add vector $(5,6)$ to the set, $B = \{(3,1), (1,3), (5,6)\}$. Is the vector set B independent? No, vector $(5,6)$ can be expressed as linear combination of other vectors $(3,1), (1,3)$ from the set as

$$(5,6) = \frac{9}{8}(3,1) + \frac{13}{8}(1,3) = \left(\frac{27}{8}, \frac{9}{8}\right) + \left(\frac{13}{8}, \frac{39}{8}\right) = (5,6)$$

What is span of vector set A ? Vector set A can span all vectors of \mathbb{R}^2 . Adding any vector from \mathbb{R}^2 would make this set dependent.

Tip: How many numbers of independent vectors are required to span a Euclidean space of \mathbb{R}^n ? Answer is n .

A linear combination of vectors $k_1 \mathbf{v}_1 + k_2 \mathbf{v}_2 + \dots + k_n \mathbf{v}_n$ is called a *convex combination* if $\sum k_i = 1$ and $0 \leq k_i \leq 1$. The set of all convex combination is called the *convex hull* of the set of vectors $\mathbf{v}_i : i = 1, \dots, n$. As a particular example, every convex combination of two points lies on the line segment between the points. In n -dimension, a line segment between vectors x and y is represented as $ax + (1 - a)y : a \in [0,1]$. A set is *convex* if it contains all convex combinations of its points. The convex hull of a given set of points is identical to the set of all their convex combinations.

Dimension and basis of the space

Dimension of a space is count of the independent vectors that spans all vectors of the space. In previous example, vectors $\mathbf{u} = (3, 1)$ & $\mathbf{v} = (1, 3)$ spanned all points in \mathbb{R}^2 and are linearly independent, so the dimension of \mathbb{R}^2 is . In generic way, dimension of Euclidean space of \mathbb{R}^n is n . If dimensions of a space are finite, then it is called *finite-dimensional space*. If dimensions of a space are infinite, then it is called *infinite-dimensional space*.

Set of independent vectors that span all vectors of the space are called *basis vectors* of that space. For the previous example of \mathbb{R}^2 , *basis* = $\{(3,1), (1,3)\}$ as they spanned all points in \mathbb{R}^2 and are independent set. Another example of basis vectors for \mathbb{R}^2 is $\{(1,0), (0,1)\}$.

Orthogonal and orthonormal basis

Basis vectors are said to be *orthogonal basis* if every basis vector is orthogonal to other basis vectors. Vectors are said to be orthogonal if their dot product is 0 or angle between them is 90° . Consider two sets of basis vectors in \mathbb{R}^2 $\{(3,1),(1,3)\}$ and $\{(-1,2),(2,1)\}$. Dot product of first set, $(3,1) \cdot (1,3) = 3 * 1 + 1 * 3 = 9$, so these basis vectors are not orthogonal. Dot product of second set, $(-1,2) \cdot (2,1) = -1 * 2 + 2 * 1 = 0$, so these basis vectors are called orthogonal basis.

Basis vectors of a space are said to be *orthonormal basis* if they are orthogonal basis and are unit vectors (norm of all vectors is 1). Vectors $(1,0)$ & $(0,1)$ form orthogonal basis of space \mathbb{R}^2 and their norm is $\|(1,0)\| = \sqrt{1^2 + 0^2} = 1$, $\|(0,1)\| = \sqrt{0^2 + 1^2} = 1$. This set of vectors $\{(1,0), (0,1)\}$ form orthonormal basis of \mathbb{R}^2 .

Natural orthonormal basis of \mathbb{R}^n

Orthonormal basis for Euclidean space of \mathbb{R}^n would be:

$$\{(1, 0, 0, \dots, 0, 0), (0, 1, 0, \dots, 0, 0), \dots, (0, 0, 0, \dots, 1, 0) (0, 0, 0, \dots, 0, 1)\}$$

where first basis vector will have 1 in first position and zero in all others; similarly, for k^{th} basis, vector will have 1 in k^{th} position and zero in all others. Total number of basis vectors for space of \mathbb{R}^n will be in n , which is the dimension of \mathbb{R}^n . These basis vectors are orthogonal to each other, and their norm is 1. These orthonormal basis vectors are widely used in Euclidean space. For all examples in this section, these orthonormal bases

were used to plot vectors of the space. Value used for the vector representation $\mathbf{u} = (a_1, a_2, \dots, a_{n-1}, a_n)$ were scalar multiplies of the respective orthonormal basis required for expressing \mathbf{u} as linear combination of orthonormal basis vectors:

$$\mathbf{u} = a_1 * (1,0,0, \dots, 0,0) + a_2 * (0,1,0, \dots, 0,0) + \dots + a_{n-1} * (0,0,0, \dots, 1,0) \\ + a_n * (0,0,0, \dots, 0,1)$$

Subspaces

A non-empty subset of Euclidean space $S \subseteq \mathbb{R}^n$ is called *subspace* if S is closed under linear combinations. In other words, non-empty subset of Euclidean space is called *subspace* if all vectors spanned by any subset of S belongs to S . Subspace can also be called vector space (explained later). Formally, a subset of Euclidean space forms *subspace* if:

$$\forall \mathbf{v}_1, \mathbf{v}_2 \in S, \alpha \mathbf{v}_1 + \beta \mathbf{v}_2 = \mathbf{v}_3 \text{ then } \mathbf{v}_3 \in S \text{ where } \alpha, \beta \in \mathbb{R}$$

Example of trivial subspace: Consider Euclidean space \mathbb{R}^n . Only zero vector $\mathbf{o} = (0,0,\dots,0)$ of this space can be subspace. This subspace of only zero vector is called trivial subspace.

Example of a line in \mathbb{R}^2 : Consider any line ($-\infty$ to ∞) that passes through origin. All vectors on the line are closed under addition and scalar multiplication. So, any line that passes through the origin is subspace.

Dimension of subspace

Count of independent vectors that spans all vectors of the subspace with their linear combination is the *dimension of that subspace*.

Example of a line in \mathbb{R}^n : Any line that passes through the origin is subspace. How many independent vectors are required to span all the vectors of the line? Only one vector along the direction of the line is sufficient to span all the vectors of the line. So, dimension of line subspace is 1.

Note: Dimensions of Euclidean space \mathbb{R}^n is n .

Hyperplanes and Halfspaces

A *hyperplane* in \mathbb{R}^n is set of all vectors $\mathbf{x} = (x_1, x_2, \dots, x_n)$ that satisfies the equation $a_1 x_1 + a_2 x_2 + \dots + a_n x_n = b$ where $b \in \mathbb{R}$, $\exists a_i$ such that $a_i \neq 0$

Dimension of the hyperplane in \mathbb{R}^n is $n - 1$. Hyperplane becomes subspace when $b = 0$, that is, when hyperplane is passing through origin. Let's represent the coefficients of hyperplane equation by the vector $\mathbf{a} = (a_1, a_2, \dots, a_n)$. We can write the linear equation representing the hyperplane by using dot product as $\mathbf{a}^T \cdot \mathbf{x} = b$.

This geometric interpretation can be clearly understood by expressing the hyperplane in the form $\mathbf{x} : \mathbf{a}^T \cdot (\mathbf{x} - \mathbf{x}_0) = b$, where \mathbf{x}_0 is any point in the hyperplane, and hence, $\mathbf{a}^T \cdot \mathbf{x}_0 = b$. For any arbitrary vector \mathbf{x} on the hyperplane, $(\mathbf{x} - \mathbf{x}_0)$ is a vector along the hyperplane. Hence, $\mathbf{a}^T \cdot (\mathbf{x} - \mathbf{x}_0) = 0$ implies vector \mathbf{a} must be perpendicular to $(\mathbf{x} - \mathbf{x}_0)$. Thus, vector \mathbf{a} is perpendicular to the hyperplane or is a *normal vector to the hyperplane*.

The hyperplane $H = \{\mathbf{x} : \mathbf{a}^T \cdot \mathbf{x} = b\}$ can be interpreted as the set of points with a constant dot product b to a given vector \mathbf{a} , or as a hyperplane with normal vector \mathbf{a} ; the constant $b \in \mathbb{R}$ determines the offset of the hyperplane from the origin. The hyperplane divides Euclidian space \mathbb{R}^n into two half-spaces. One satisfies the inequality $\mathbf{a}^T \cdot \mathbf{x} \geq b$ and is denoted by $H_+ = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{a}^T \cdot \mathbf{x} \geq b\}$, and the another one satisfies $\mathbf{a}^T \cdot \mathbf{x} < b$, denoted by $H_- = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{a}^T \cdot \mathbf{x} < b\}$. These half-spaces are called *positive half-space* and *negative half-space*, respectively. Half-spaces in two-dimensional space are captured in [Figure 2.10](#):

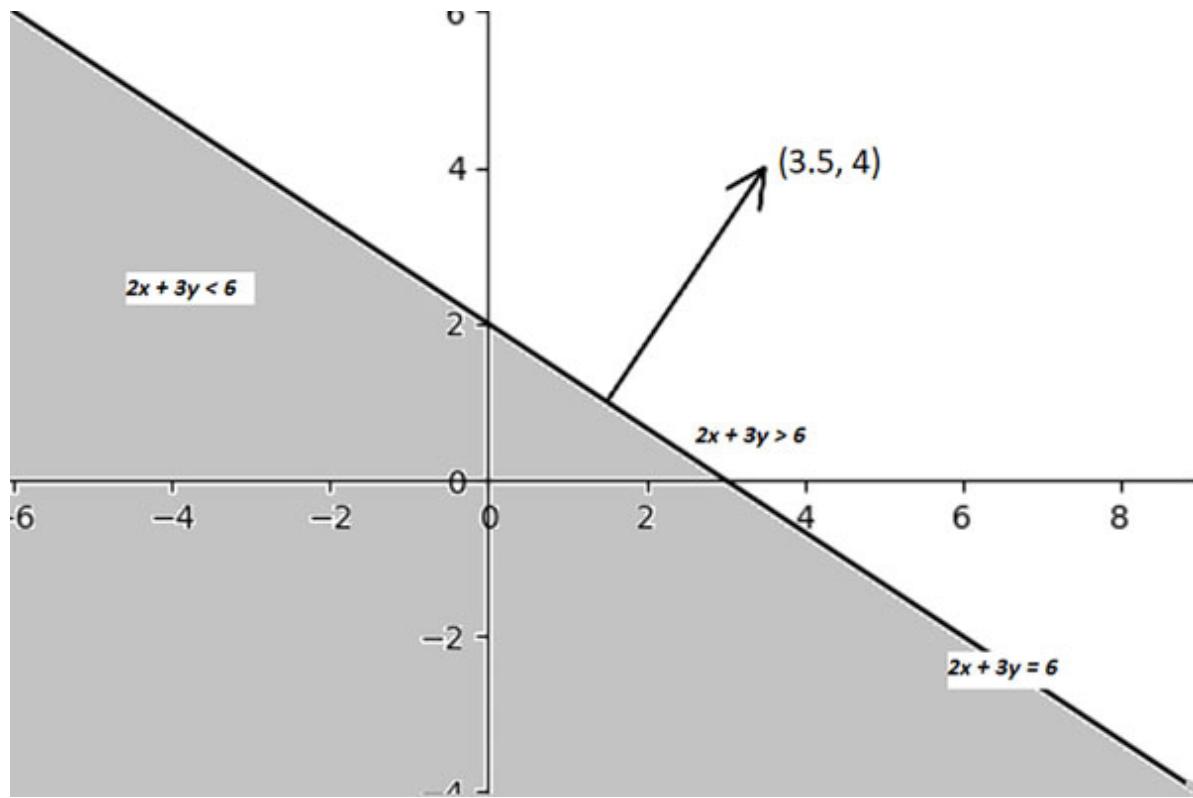


Figure 2.10: Positive half-space and negative half-space in two-dimensional space

In Euclidean space, the vectors were n -tuples of real numbers. One can generalize this concept of vector by replacing these n -tuples with any abstract object that exhibits the same properties as n -tuples. For example, consider set of all mxn real or complex matrixes. Properties like scalar multiplication and addition are well-defined even for these matrices. These matrices can be considered as vectors and define a set of basis matrices that spans the set of all mxn matrices with orthonormal basis. We can also define inner product of two matrices. So, these set of matrices can form a space of own. Let's now formally introduce vector space by abstracting out the properties enjoyed by Euclidean space.

Defining vector space

A *space* is a mathematical structure in which mathematical objects are represented using points, and the mathematical structure defines relations between points of the space. It is these relationships that define the nature of the space.

Before mathematically defining the vector space, let us understand the cartesian product operator (\times) and the map operator (\rightarrow). Cartesian product of two sets A and B , denoted as $A \times B$, is set of all ordered pairs (a, b) where $a \in A$ and $b \in B$. Another way to define cartesian product on $A & B$ is $A \times B = \{(a,b)|a \in A \text{ and } b \in B\}$. Map operator maps every element of a set to one element of another set b , denoted as $A \rightarrow B$.

Vector spaces

A *vector space* over real or complex numbers denoted by \mathbb{F} (= \mathbb{R} for real numbers or = \mathbb{C} for complex numbers) is a set of vectors V , along with two operations addition and scalar multiplication, that satisfy eight axioms. Let's formally define addition and scalar operation on vectors:

- Addition or vector addition (closure over addition) defined as $+ : V \times V \rightarrow V$, inputs any two vectors $v \in V$ and $w \in V$ (vectors are represented by small bold letters, like \mathbf{u} or arrow over small letter like \vec{u}) and outputs a third vector $v \in V$ written as $\mathbf{u} = v + w$ where \mathbf{u} is called the sum of v & w vectors.
- Scalar multiplication (closure over scalar multiplication) defined as $\cdot : \mathbb{F} \times V \rightarrow V$, takes any scalar $a \in \mathbb{F}$ and any vector $v \in V$ and outputs vector $\mathbf{u} = av$ where $\mathbf{u} \in V$.

Let's discuss the eight axioms that must be followed by addition and scalar multiplication operation. Consider vectors $\mathbf{u}, \mathbf{v}, \mathbf{w} \in V$ and scalars $a, b \in \mathbb{F}$:

- **Addition** associativity:

$$\mathbf{u} + (\mathbf{v} + \mathbf{w}) = (\mathbf{u} + \mathbf{v}) + \mathbf{w} \quad \forall \mathbf{u}, \mathbf{v}, \mathbf{w} \in V$$
- **Addition commutativity:** $\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u} \quad \forall \mathbf{u}, \mathbf{v} \in V$
- **Addition identity vector:** $\exists \mathbf{o} \in V$ (called zero vector) such that $\mathbf{v} + \mathbf{o} = \mathbf{v} \quad \forall \mathbf{v} \in V$
- **Addition inverse vector:** $\forall \mathbf{v} \in V \exists -\mathbf{v} \in V$ (called additive inverse) such that $\mathbf{v} + (-\mathbf{v}) = \mathbf{o}$ where \mathbf{o} is the identity element of addition
- **Compatibility of scalar and field multiplication:** $a(b\mathbf{v}) = (ab)\mathbf{v} \quad \forall a, b \in \mathbb{F} \quad \forall \mathbf{v} \in V$
- **Scalar multiplication identity element:** $1\mathbf{v} = \mathbf{v}$ where 1 denotes multiplicative identity in \mathbb{F}

- **Scalar multiplication distributivity over vector addition:** $a(\mathbf{u} + \mathbf{v}) = au + av \quad \forall a \in \mathbb{F} \quad \forall u, v \in V$
- **Scalar multiplication distributivity over scalar (field) addition:** $(a + b)\mathbf{u} = au + bu \quad \forall a, b \in \mathbb{F} \quad \forall u \in V$

Note: *Scalar multiplication should not be confused with the scalar product (also called inner product or dot product).*

Example of Euclidean space \mathbb{R}^n is vector space: Consider $\mathbf{u} = (a_1, a_2, \dots, a_n) \in \mathbb{R}^n$ & $\mathbf{v} = (b_1, b_2, \dots, b_n) \in \mathbb{R}^n$ of Euclidean space, where addition is defined as $\mathbf{u} + \mathbf{v} = (a_1 + b_1, a_2 + b_2, \dots, a_n + b_n)$ and scalar multiplication for $s \in \mathbb{F}$ is defined as $s\mathbf{u} = (sa_1, sa_2, \dots, sa_n)$. These two operations follow closure property over addition and scalar multiplication. They also follow eight axioms with zero vector $\mathbf{o} = (0, 0, \dots, 0)$ and additive inverse $-\mathbf{u} = (-a_1, -a_2, \dots, -a_n)$. So, Euclidean space of \mathbb{R}^n is vector space.

Example of real numbers: Set of real numbers, $a \in \mathbb{R}$ with standard multiplication and addition. Zero vector will be $\mathbf{o} = 0$ and additive inverse will be $-a$.

Example of matrices: Set of all matrices $A_{m \times n}$ forms vector space where addition and scalar multiplication of matrices are as defined in section ‘*Basic Matrix Operations*’. Each $A_{m \times n}$ is considered vector in this vector space. Verify that these two operations meet all eight axioms.

Example of l^1 sequence space: The l^1 sequence space is defined to be the set of all sequences whose series is absolutely convergent that is, $\sum_{i=1}^{\infty} |a_i| < \infty$. Addition of two sequences is element-wise addition of the corresponding elements of the sequences and scalar multiplication is element-wise multiplication by a scalar. l^1 sequence space is a vector space where each sequence is a vector. Addition of absolutely convergent series is convergent, and scalar multiplication of convergent series is still convergent. Verifying if these operations hold all eight axioms is left to the reader. This vector space is not finite dimensional space. The norm of a sequence can be defined as the finite sum $\|\{a_n\}\|_1 = \sum_{i=1}^{\infty} |a_i| < \infty$, which is always finite due to absolute convergence. So, l^1 sequence space is an infinite dimensional normed linear space. The infinite basis set for this is the set of all infinite

sequences i^{th} whose element is 1 and all other elements of this sequence is 0. This is like a generalization of the standard basis for \mathbb{R}^n .

Example: Set of all polynomials with real coefficients forms an infinite dimensional vector space where each polynomial is a vector. The basis of this vector space is the set of polynomials with one term: $x^n : n = 0, 1, 2, \dots$. Every polynomial can be represented as a finite linear combination of polynomials from this basis set.

Example: $L_2[a, b]$: The set of all real-valued functions square integrable in the interval $[a, b]$, that is, set of functions $f : [a, b] \rightarrow \mathbb{R}$, such that forms a vector space. This set of functions forms a vector space. We define addition of functions f and g as a square integrable function $h = f + g$, where $h(x) = f(x) + g(x)$ and scalar multiplication of function is defined as $cf(x)$.

Vector spaces define addition of vectors and scalar multiplication but doesn't define the length of a vector, distance between vectors, or angles among vectors. To have meaning to these parameters among vectors, additional structure over vector space are to be defined like Normed vector space and Inner product space.

Normed vector space

Normed vector space is vector space over which norm is defined. Norm represents “*length*” of a vector that can be any abstract object like a matrix, a function, or a sequence. We already defined norm in Euclidean space called Euclidean norm. Now, let's generalize that concept to arbitrary vector spaces. *Norm* of vector is denoted as $\|v\|$, which must have the following properties:

- Non-negative $\forall v \in V, \|v\| \geq 0$
- Positive norm value for non-zero vector $\|v\| = 0 \Leftrightarrow v = o$ where o is additive identity vector (zero vector) in vector space
-
- **Triangle inequality holds:**
- Properties of norm
- Parallelogram law

A norm can be turned into a distance metric, using:

This is called the *metric induced by the norm*. In general, a distance metric is defined as a function, which takes two vectors and outputs the distance between them. Distance must be always non-negative between two points. Distance from a to b should be same as the distance from b to a; this is called symmetry property of distance. Another important property of distance is triangle inequality, which states that if we have three vectors $\mathbf{u}, \mathbf{v}, \mathbf{w}$, then in the triangle formed by these vectors, the sum of distances of any two sides must be greater than third side. We can write these properties in terms of norm as follows: $\|\mathbf{u} - \mathbf{v}\| \geq 0$

- $\|\mathbf{u} - \mathbf{v}\| = 0$ if and only if $\mathbf{u} = \mathbf{v}$
- Symmetry, $\|\mathbf{u} - \mathbf{v}\| = \|(-1)(\mathbf{v} - \mathbf{u})\| = \|\mathbf{v} - \mathbf{u}\|$,
- Triangle inequality, $\|\mathbf{u} - \mathbf{v}\| \leq \|\mathbf{u} - \mathbf{w}\| + \|\mathbf{w} - \mathbf{v}\|$

Norm of real numbers

The set of real numbers also form a trivial vector space. Norm of a real number can be defined as follows. Does this definition follow all four properties of norm? Verification task is left to the reader as an exercise.

Euclidean space is a normed linear space endowed with Euclidean norm. There are other norms defined in Euclidean spaces called l^p norms.

l^p Norm

l^p -norm of \mathbf{x} is defined as , where & .

For $p=1$ this reduces to:

Any norm can be used to define the distance between two vectors as norms introduce a distance metric given by $d(\mathbf{u}, \mathbf{v}) = \|\mathbf{u} - \mathbf{v}\|$.

The, l^1 introduces the *Manhattan distance*.

Streets of Manhattan city are in grid layout. To move from one point to another, a person must drive along these roads that form grid layout. Distance calculated between points where movement is possible only in grid format is called *Manhattan distance*.

Maximum norm

When $p \rightarrow \infty$ for l^p -norm, it provides *l-norm* or *maximum norm* of vector \mathbf{u} and is defined as:

Verify that l^p norm follows all four properties of the norm definition.

Note: Minkowski Distance between vectors $\mathbf{u} = (a_1, a_2, \dots, a_n)$ & $\mathbf{v} = (b_1, b_2, \dots, b_n)$ is defined for $p \in \mathbb{Z}$ (integer) below. The formula is the same as p -norm for $p \geq 1$.

$$dist(\mathbf{u}, \mathbf{v}) = (|a_1 - b_1|^p + |a_2 - b_2|^p + \dots + |a_n - b_n|^p)^{1/p}$$

For $p \geq 1$, it follows all four properties of the distance metric. When $p < 1$, it fails in triangle inequality property

$$dist((0,0), (1,1)) > dist((0,0), (0,1)) + dist((0,1), (1,1))$$

Let sequence space be denoted as l^p and this set is defined as

The norm where $1 \leq p \leq \infty$, can be defined on the l^p sequence space by extending the finite sum of p^{th} power terms to an infinite sum of p^{th} power series. So, the *sequence space l^p is a normed linear space*, where the norm $\|\cdot\|_p$ is defined as:

Similarly, for function space L^p , the norm $\|\cdot\|_p$ of a function $f \in L^p$ is defined as:

Matrix norm

Norms are also defined for matrices as matrices can also be treated as vectors in vector space. There exist various matrix norms. Among these, we will discuss important matrix norms.

Like L^p norm for vectors, $L_{p,q}$ norm for matrices can be defined. $L_{p,q}$ norm for matrix $A_{m \times n}$ is defined as follows:

Frobenius norm: Keeping values $p = q = 2$ for matrix norm $\|A\|_{p,q}$, one obtains $L_{2,2}$ norm, which is also called *Frobenius norm* or *Hilbert-Schmidt norm*.

Example: An image can be represented as a matrix of pixel values. We can compare how close two images are using Forbenius norm.

Norm ball: Just like the Euclidean norm defines a neighborhood of a point using Euclidean ball, any norm in an abstract vector space can define a norm ball:

So, in the function space L^p , we define the neighborhood of a function f as:

that is, set of all functions h that closely approximate the function f .

Norm defines the length of a vector, the distance between vectors and few properties of distances between vectors, but it does not define angle between abstract vectors. Let's discuss Inner product spaces that provide meaning to angle between vectors in vector space.

Inner product

An inner product is a generalization of the concept of dot product in Euclidean space, which is denoted by $\langle \cdot, \cdot \rangle$ and satisfies four properties for $\langle \cdot, \cdot \rangle$.

- Distribution
- Linearity
- Symmetry or Commutativity
- **Positive definiteness:** If \mathbf{o} is identity element of addition in vector space V , then $\langle \mathbf{v}, \mathbf{v} \rangle = 0$ iff $\mathbf{v} = \mathbf{o}$ else $\langle \mathbf{v}, \mathbf{v} \rangle > 0$

An inner product naturally induces an associated norm (length) of a vector $\mathbf{u} \in V$ and is defined as $\|\mathbf{u}\|$. A normed linear space is called an **inner product space** if the norm is introduced by an inner product. With definition of norm one can define distance between vectors $\mathbf{u}, \mathbf{v} \in V$ as $distance(\mathbf{u}, \mathbf{v}) = \|\mathbf{u} - \mathbf{v}\|$. Along with this, there are a few other properties that result from the definition of norm in inner product space:

- Cauchy–Schwarz inequality
- Orthogonality: Two vectors $\mathbf{u}, \mathbf{v} \in V$ are orthogonal if $\langle \mathbf{u}, \mathbf{v} \rangle = 0$. Geometrically, orthogonal means perpendicular.
- Pythagorean theorem

Example of Euclidean space \mathbb{R}^n : Consider $\mathbf{u} = (a_1, a_2, \dots, a_n) \in \mathbb{R}^n$ & $\mathbf{v} = (b_1, b_2, \dots, b_n) \in \mathbb{R}^n$ of Euclidean space where dot product is defined as $\mathbf{u} \cdot \mathbf{v} = a_1 * b_1 + a_2 * b_2 + \dots + a_n * b_n$. One can verify that dot product follows all four properties of inner product map. So, Euclidean space of \mathbb{R}^n is inner product vector space with dot product as inner product; $\langle \mathbf{u}, \mathbf{v} \rangle = \mathbf{u} \cdot \mathbf{v}$.

Example of real numbers: Set of real numbers $a, b \in \mathbb{R}$ with standard multiplication as inner product $\langle a, b \rangle = a * b$.

Application on real dataset

Roland Fisher in his 1938 paper published a data set consisting of 50 samples from each of the three species of Iris flower (setosa, virginica, and

versicolor). Each sample of the flower was measured for four features in centimeters: length and width of sepals and petals. Let's represent flowers data in vectors using TensorFlow framework; setosa, virginica, and versicolor represent labels 0, 1, and 2. As each flower is represented by four features, vectors representing the flower will belong to \mathbb{R}^4 . The following code provides steps to load iris data. Few samples from the data set are then copied to the TensorFlow array for further analysis.

```

1. import tensorflow as tf
2. import tensorflow_datasets as tfds
3. import numpy as np
4. iris_ds = tfds.load('iris', split='train') # load Iris
dataset
5. SAMPLE_SIZE = 6 # consider few samples from the dataset
6. iris_ex = iris_ds.take(SAMPLE_SIZE) # consider first few
samples from the dataset
7. # Tensor array to store dataset samples
8. tf_arr = tf.TensorArray(tf.float32, size=0,
dynamic_size=True, clear_after_read=False)
9. index = 0
10. for sample in iris_ex:
11.     feat = sample["features"]
12.     label = sample["label"]
13.     tf_arr = tf_arr.write(index, feat) # Add sample to
array
14.     index = index + 1
15.     vec_norm = tf.norm(feat) # Calculate norm of the
vector
16.     tf.print("label=", label, ' vector v', index, '=', feat, " Euclidean Norm=", vec_norm, sep="")

```

Code outputs vector's values, label and Euclidean norm, which is captured in [Figure 2.11](#):

Figure 2.11: Few vectors from Iris Dataset

Note: How is the angle between two vectors defined in higher dimensions? Two independent vectors span a plane in higher

dimensions. Plane forms a subspace, and its dimension would be 2. The angle between the vectors in the subspace plane is defined.

Let's calculate the distance and angle between these sample vectors stored in TensorFlow array. The following code provides steps to calculate the Euclidean distance:

```

1. for ref_index in range(0, SAMPLE_SIZE):
2.     vec_ref = tf_arr.read(ref_index)
3.     for arr_index in range(0, SAMPLE_SIZE):
4.         vec = tf_arr.read(arr_index)
5.         vec_sub = tf.math.subtract(vec_ref, vec)
6.         distance = tf.math.reduce_euclidean_norm(vec_sub)

```

Output of the code is captured in [Figure 2.12 \(Left\)](#). From captured distance matrix data, we can conclude that v_1 is close to v_5 , v_2 to v_4 and v_3 to v_6 . We can notice that vectors belonging to same class are closer. Refer to the following figure:

Figure 2.12: (Left) Distance between vectors (Right) Angle between vectors

The following code provides steps to calculate angle between vectors. The output of the code is captured in [Figure 2.12 \(Right\)](#). From the captured angle matrix data, we can conclude that v_1 is close to v_5 , v_4 to v_3 & v_3 and v_4 to v_6 .

```

1. for ref_index in range(0, SAMPLE_SIZE):
2.     vec_ref = tf_arr.read(ref_index)
3.     vec_ref_eu_nr = tf.math.reduce_euclidean_norm(vec_ref)
4.     for arr_index in range(0, SAMPLE_SIZE):
5.         vec = tf_arr.read(arr_index)
6.         vec_eu_nr = tf.math.reduce_euclidean_norm(vec)
7.         dot_prod = tf.math.reduce_sum(vec_ref * vec)
8.         angle_d = tf.math.acos(dot_prod/(vec_eu_nr *
vec_ref_eu_nr))

```

Visualizing these vectors in \mathbb{R}^4 is not possible as we can comprehend till three dimensions only. Let's represent 100 vectors of the data set as two-dimensional vectors using t-SNE algorithm. The t-SNE algorithm is a non-linear dimensionality reduction technique well-suited for embedding high-

dimensional data in low-dimensional space of two or three dimensions for visualization. More discussion on t-SNE will be available in [chapter 8 Dimensionality Reduction](#). The output of the mapping is shown in [Figure 2.13](#). As you can see in [Figure 2.13](#), setosa class can be differentiated easily while other two classes are closer:

Figure 2.13: Visualizing vectors from high dimension in two dimensions

Iris dataset also has the class labels or flower names associated with each of the 150 observations. If we are given the measurement vector or observation of a new flower belonging to any of these three flower categories, can we tell what is the correct type of the new flower? Using distance metric induced by norm alone, we can build a simple classifier for the Iris dataset. One such algorithms that works on the principle of distance between vectors is K-Nearest Neighbour. It uses the entire dataset to derive a conclusion about the possible category of the new observation.

K-nearest neighbor

K-nearest neighbour (KNN) algorithm works on the principle of distance between vectors. Mostly, Euclidean distance is used to calculate the distance. In this algorithm, the distance of the new observation vector is computed with every vector in the dataset. The class of the new observation vector is decided based on its nearest neighbours in the given dataset whose classes are already known. The neighbours are defined as the best K nearest vectors in the given dataset with respect to the chosen distance metric. The number of nearest neighbours K is a parameter of the algorithm and is determined empirically. [Figure 2.14](#) shows the classification of the new sample based on values of K. For K=3, new sample would get classified as *Class B*, but for K=7, new sample would get classified as *Class A*. Refer to the following figure:

Figure 2.14: Classifying the sample based on K-nearest neighbors

The following code explains main logic of KNN algorithm. Out of 150 samples of Iris dataset, first 120 samples are considered for training and the rest are for testing. We can consider the remaining 30 samples in the test set

as new unseen observations. Each sample from test set is assigned label based on maximum repetition of labels among K -nearest neighbours from the training set. The following code runs for a range of K values in steps of 3. For, two samples are mispredicted, and for, one sample is mispredicted:

```
1. # Load Iris dataset of 150 samples into train and test test
2. iris_train = tfds.load('iris', split='train[:120]')
3. iris_test = tfds.load('iris', split='train[120:]')
4. # Extract features and labels of iris samples
5. iris_train_feat, iris_train_label =
get_features_labels(iris_train)
6. iris_test_feat, iris_test_label =
get_features_labels(iris_test)
7. # Assign labels to test samples based on k-nearest
neighbours
8. for k_val in range(3, 34, 3):
9.     test_knn_labels = np.zeros((0), dtype=np.int)
10.    outer_index = 0
11.    # Calculate Euclidean distance between a test vector &
all training vectors
12.    # Select k-nearest neighbors based on the distance
13.    for test_feat_samp in iris_test_feat:
14.        eu_dis = np.zeros((0), dtype=np.float)
15.        inner_index = 0
16.        for train_feat_samp in iris_train_feat:
17.            euclidean_distance =
np.linalg.norm(train_feat_samp - test_feat_samp)
18.            eu_dis = np.insert(eu_dis, inner_index,
euclidean_distance, axis=0)
19.            inner_index = inner_index + 1 # Move to next
train vector
20.            sorted_index = np.argsort(eu_dis) # Sort based on
euclidean distance
21.            # labels of k-shortest distance
22.            nearest_k_labels =
iris_train_label[sorted_index[0:k_val]]
23.            (labels, count) = np.unique(nearest_k_labels,
return_counts=True)
```

```

24.      # Assign maximum repeated label
25.      test_samp_knn_pred_label = labels[np.argmax(count)]
26.      test_knn_labels = np.insert(
27.          test_knn_labels, outer_index,
test_samp_knn_pred_label, axis=0)
28.      outer_index = outer_index + 1 # Move to next test
vector
29.      correct_prediction = np.sum(test_knn_labels ==
iris_test_label)
30.      wrong_prediction = iris_test_label.shape[0] -
correct_prediction
31.
32.      print("k-val:", k_val, "\nPD:", test_knn_labels,
"\nGT:", iris_test_label)
33.      print("correct prediction", correct_prediction, ",wrong
prediction", wrong_prediction)

```

This section discussed vectors, their interaction in vector space, and its properties, along with the formal definition of Euclidean space. Next, let's look at representing these vectors in matrix format.

Representing vectors in matrix

Vectors can be represented in other format that is beneficial in matrix operations. They can be represented either with column matrices or row matrices. In matrix $A_{m \times n}$, each column of the matrix can be treated as vector in \mathbb{R}^m space, and each row can be treated as vector in \mathbb{R}^n space.

Let's represent vectors as columns in a matrix. Let's represent each vector in matrix format separately, and then combine these vectors as columns of a rectangular matrix:

Representing the vectors as columns:

Space spanned by all columns of a matrix is called *column space* of A , denoted as $C(A)$. Similarly, space spanned by all rows of a matrix is called

row space of A , denoted as $C(A^T)$. These two spaces $C(A)$ & $C(A^T)$ associated with a matrix A are subspaces of \mathbb{R}^2 & \mathbb{R}^3 .

What would be dimensions of these two subspaces?

Dimension of column subspace $C(A)$ of A would be equal to the number of linearly independent column vectors of A . Similarly, dimension of row subspace $C(A^T)$ of A would be equal to the number of linearly independent row vectors of A . In the preceding example, column set and row set of A has two independent vectors each.

With dimensions of these two subspaces of matrix, one can define the rank of a matrix. This property of the matrix is widely used.

Note: Usually, vectors are represented as columns, but when they are represented as vectors, the transpose operator ‘T’ is used. Representing the vectors $v_1 = (1,2)$, $v_2 = (3,1)$ & $v_3 = (2,5) \in \mathbb{R}^2$ as rows provides

Matrix rank

Rank of a matrix is defined as the minimum of the dimensions of its column subspace and row subspace.

Rank of A is 2 as dimension of column, and row subspace is 2. Rank of A^T is 2 as dimension of column, and row subspace is 2. Here is another example of a matrix whose rank is 1.

Matrices types

Matrix is said to be *real matrix* if all entries of the matirx are real numbers. Matrix is called *square matrix* if the number of rows and columns are equal. *Diagonal* of square matrix runs from the top-left corner to the bottom-right corner of the matrix. All entries of *Diagonal Matrix* (D) outside main diagonal are zero. All entries of *upper triangular matrix* (U) below the

diagonal are zero. All entries of *lower triangular matrix* (L) above the diagonal are zero. Here are a few examples:

Identity matrix

Identity matrix represented as I_n is $n \times n$ matrix whose diagonal elements are equal to 1, and all other entries are zeros. Any square matrix S_n multiplied by identity matrix I_n of the same dimensions results in the same matrix S_n ; due to this, I_n is called identity matrix. I_n is an identity element for matrix multiplication.

Symmetric matrix

Symmetric matrix is a square matrix that is equal to its transpose $A = A^T$. In other words, matrix A_m is symmetric if $a_{xy} = a_{yx}$, where $1 \leq x, y \leq m$. All identity matrices are symmetric. The distance between vectors shown in [Figure 2.9](#) is symmetric matrix.

Properties of symmetric matrices A, B :

- $A + B$ & $A - B$ results in symmetric matrix
- A^n is symmetric matrix for $n \in \mathbb{N}$
- A^{-1} is symmetric

Skew symmetric matrix

Skew-symmetric matrix is a square matrix that is equal to the negative of its transpose $A = -A^T$. In other words, square matrix A_m is skew-symmetric if $a_{xy} = -a_{yx}$, where $1 \leq x, y \leq m$. By definition, all diagonal elements of skew-symmetric matrices are zero. All identity matrices are not skew-symmetric as diagonal elements are non-zero. Here are a few examples:

Properties of skew-symmetric matrices A_m , B_m :

- kA_m is symmetric where $k \in \mathbb{R}$
- $A + B$ is skew-symmetric
- $A + I$ is always invertible

Invertible matrices

Invertible / Non-Singular / Non-Degenerate Matrix A_n is a square matrix, and there exists a matrix B_n such that:

$$AB = BA = I_n$$

where I_n is identity matrix and multiplication used is ordinary matrix multiplication. In this case, matrix B is uniquely determined by A , represented as A^{-1} . Square matrix that is not invertible is called *singular* or *degenerate matrix*.

Question: Under what conditions does the inverse of a matrix exists?

Invertible matrix theorem states that real square matrix A_n is invertible if and only if any one of the following conditions hold:

- A_n is row-equivalent to identity matrix I_n
- A_n is column-equivalent to identity matrix I_n
- A_n has n pivot positions (a pivot position in a matrix is a location that corresponds to a leading 1 in the reduced echelon form of the matrix)
- $\text{rank}(A_n) = n$, that is, rank of A_n is full rank
- Equation $Ax = 0$ has only the trivial solution $x = 0$, zero vector
- Equation $Ax = b$ has exactly one solution for $\forall b \in \mathbb{R}^n$
- Columns of A are linearly independent
- Column subspace $C(A) = \mathbb{R}^n$
- Columns of A form basis of \mathbb{R}^n
- $\exists B_n$ such that $AB = BA = I_n$
- Transpose of matrix A^T is invertible

Properties of Matrix Inverse

The following inverse properties hold for non-singular matrices A , B & A_i :

-
-
-
- , where
- are inverse of each other
- A^{-1} is unique
- Only solution for is

Examples of matrices and their inverse

Verify the independence of rows and columns of invertible matrices. Note that inverse of first and third matrices below are their own. These kinds of matrices are called *involutory matrix*, that is, $A^2 = I$.

Example: The following matrices don't have inverses as row/columns of the matrix are dependent:

has dependent column set,
has dependent column set,

Permutation matrix

Permutation matrix is a square matrix that only one entry of 1 in each of rows and columns, and other entries are 0. Every permutation matrix can be obtained by shuffling/permuting the rows of identity matrix of the same dimension. Obtain permutation matrix to shuffle rows of 4×4 matrix, where r_x represents row x of matrix A to r_2, r_1, r_4, r_3 . Row permutation matrix can be obtained by moving the row vector of identity matrix. If r_2 of A should be moved to the first row, and then move r_2 of identity matrix to the first row.

Similarly, one can permute columns of a matrix, but column permuted identity matrix should be multiplied to the right of matrix A .

Orthogonal matrix

Orthogonal matrix or *orthonormal matrix* is a real square matrix whose set of columns and rows form orthonormal sets. For any orthonormal matrix Q_m , columns vector set C and rows vector set R are orthonormal sets.

Let's represent matrix Q_m in terms of row vectors. Transpose of this matrix Q_m^T can be stated in a simple way. Multiplying orthonormal matrix with its transpose as $Q_m Q_m^T$ can be stated using dot product.

is now number

where is multiplication of row vector with column vector resulting in scalar value. Row and column vector set are orthogonal unit vectors.

Obtaining identity matrix indicates inverse of orthonormal matrix is its transpose. Here are a few properties of orthogonal matrices Q_m :

- Inverse of the matrix is its transpose
- Product of two orthogonal matrices will be orthogonal
- Identity matrices I_n are orthogonal
- Transpose of orthogonal matrix is orthogonal

These properties of orthogonal matrices are helpful in decomposing the matrix and mapping vectors from one space to another, which will be further discussed in matrix decomposition and linear transformation.

Matrices in ML problem formulation

While formulating ML problem, properties of vectors are usually captured in matrix format. This section will discuss those matrices.

Feature/data matrix

Matrix representing samples of a dataset is called *Feature or data matrix*. The samples of a dataset are represented as row vectors and columns of the matrix will represent features of each sample. This compact representation of dataset in matrix format helps in the implementation of ML algorithms.

Consider Iris dataset from previous section of this chapter. Iris dataset contains samples of three types of Iris flowers: setosa, virginica, versicolor. Each flower is captured with four parameters called features: petal length, petal width, sepal length, and sepal width. The features of the first five samples for the Iris dataset are represented below as vectors:

Representing these vectors of sample data as rows of a matrix is called *feature matrix* or *data matrix*. Classes of these samples, setosa, virginica, and versicolor are numbered with values 0, 1, and 2, respectively. Classes of the samples it belongs to are represented in *target vector y* , where each entry of the vector is the class of the corresponding sample in feature matrix. Due to this, the number of rows of feature matrix and target vector is equal to number of samples. Feature matrix and target vector of these sample vectors are captured as follows:

One hot encoding

In the previous example, classes of Iris dataset were assigned numbers 0, 1, and 2. For a new sample, ML algorithm must predict one of these numbers associated with the class. As algorithms only understand numbers, they might misinterpret when there is a higher number allocated to versicolor class as compared to other classes. To avoid the misinterpretation by algorithms, a class of samples can be represented with unit vector that has zero entries for other classes, except for the true class whose entry will be 1. This unit vector $u \in \mathbb{R}^m$, where m is total number of classes. These unit vectors that indicate the classes of each sample are represented as rows of

the target matrix. This form of representation of output vector is called *one hot encoding*. One hot encoded vector would have only one element as 1 (that is, hot), and the remaining entries are zeros. Let's rewrite vector from the previous example as 5x3 one-hot encoded matrix E .

One hot encoded matrix would turn out to be sparse. To reduce sparseness, we can reduce its dimensionality to obtain full matrix called *embedding matrix* that would be further discussed in [chapter 11 Natural Language Processing](#).

Distance matrix

Distance matrix captures the distance between vectors where each column and row represent a vector whose element is distance from vector a_{ij} to . The distance between vectors is symmetric function, which implies that the distance from v_i to v_j is the same as the distance from v_j to v_i . This makes distance matrix symmetric.

Consider the previous example of Iris dataset. Euclidean distance between the five vectors is captured as follows:

Gram matrix

Gram matrix represents inner product between vectors, where element of the matrix indicates inner product of vectors v_i & v_j that is, $a_{ij} = \langle v_i, v_j \rangle$. Gram matrix is symmetric as inner product function is symmetric.

Consider the previous sample of Iris dataset. Dot product (which is inner product) between vectors (rounded to integer) is captured as follows:

Covariance matrix

Covariance matrix captures covariance (joint variability) between random vectors. Each entry of the matrix denotes covariance between two random vectors. As covariance is symmetric function, covariance matrix will be

symmetric. Covariance matrix is also known as *auto-covariance* or *dispersion* or *variance* or *variance-covariance* matrix. A detailed discussion on covariance will be covered in [chapter 4 Basic Statistics and Probability Theory](#).

[Correlation matrix](#)

Correlation Matrix captures statistical relationship between random vectors, called correlation coefficient. As correlation coefficient is symmetric function, correlation matrix is symmetric. A detailed discussion on correlation will be covered in [chapter 4 Basic Statistics and Probability Theory](#)..

[Jacobian and Hessian matrix](#)

Jacobian and Hessian matrix captures first order and second order partial derivates of functions of several variables, respectively. Detailed discussed of these matrices will be done in [Chapter 3 Vector Calculus](#).

Two subspaces of a matrix were discussed previously, and these subspaces dimensions were used to define the rank of the matrix. Let's discuss more subspaces of a matrix and their relationship.

[Subspaces of matrix and orthogonality](#)

Column space of a matrix $A_{m \times n}$ is the span of its column vectors, denoted as $C(A_{m \times n})$. *Row space* of the matrix is the span of its row vectors, denoted as $C(A^T_{m \times n})$. Let's interpret \mathbf{Ax} , where vector $\mathbf{x} \in \mathbb{R}^n$ linear combination of the columns of the matrix with corresponding weights from \mathbf{x} .

To express any vector $\mathbf{b} \in \mathbb{R}^m$ in terms of \mathbf{Ax} , that is, $\mathbf{Ax} = \mathbf{b}$, vector \mathbf{b} should be in column space of matrix A , that is, $\mathbf{b} \in C(A)$. Solution doesn't exist if $\mathbf{b} \notin C(A)$.

How does solution for $\mathbf{Ax} = \mathbf{b}$ behave when $\text{rank}(A) = m$?

- If $\text{rank}(A) = m$, then column space will span all vectors of \mathbb{R}^m , that is, $C(A) = \mathbb{R}^m$, and if $m < n$, then there will exist more than one solution

for $\forall \mathbf{b} \in \mathbb{R}^m$.

- If $\text{rank}(A) = m$, then column space will span all vectors of \mathbb{R}^m , that is, $C(A) = \mathbb{R}^m$ and if $m = n$, then there will exist unique solution for $\forall \mathbf{b} \in \mathbb{R}^m$.

How does solution for $\mathbf{Ax} = \mathbf{b}$ behave when $\text{rank}(A) < m$?

- If $\text{rank}(A) < m$, then column vectors do not span all vectors of \mathbb{R}^m , that is, $C(A) \subset \mathbb{R}^m$ and if $\text{rank}(A) < n$ and $\mathbf{b} \in C(A)$, then there will exist more than one solution.
- If $\text{rank}(A) < m$, then column vectors do not span all vectors of \mathbb{R}^m , that is, $C(A) \subset \mathbb{R}^m$ and if $\text{rank}(A) = n$ and $\mathbf{b} \notin C(A)$ then there will exist a unique solution.

Null space

How does the solution behave when \mathbf{b} is zero vector?

All possible values of $\mathbf{x} \in \mathbb{R}^n$ those result in $\mathbf{Ax} = \mathbf{o}$, where \mathbf{o} is zero vector in \mathbb{R}^m , form the *null space* of the matrix A , denoted as $N(A)$. Null space is a vector space with usual definition of addition and scalar multiplication (verification is left to the reader as exercise). Let's express \mathbf{Ax} as dot product, with \mathbf{r}_i representing row i of matrix A . Dot product of \mathbf{x} with all rows \mathbf{r}_k should result in 0 value. Indicating \mathbf{x} should be orthogonal to all rows of A .

This implies that null space $N(A)$ (subspace of \mathbb{R}^n) is orthogonal to row space of A (subspace of \mathbb{R}^m). If one wants to find solution to \mathbb{R}^n , then search should be restricted to subspace of $\mathbf{Ax} = \mathbf{o}$, which is orthogonal to row space of A denoted as .

Similarly, one can analyze *left null space* of matrix A as $\mathbf{x}^T A = \mathbf{o}^T$, where \mathbf{o} is zero vector in \mathbb{R}^n & $\mathbf{x} \in \mathbb{R}^m$. This is equivalent to expressing $A^T \mathbf{x} = \mathbf{o}$. *Left null space* of is all possible solutions to $\mathbf{x} \in \mathbb{R}^m$ that result in $A^T \mathbf{x} = \mathbf{o}$, represented as $N(A^T)$. $N(A^T)$ is subspace of \mathbb{R}^m will be orthogonal to $C(A)$ column space A .

Orthogonality among subspaces

For any given real matrix $A_{m \times n}$, there exist four fundamental subspaces whose properties are captured as follows:

- Column space $C(A) \subseteq \mathbb{R}^m$
- Row space $C(A^T) \subseteq \mathbb{R}^n$
- Null space $N(A) \subseteq \mathbb{R}^n$
- Left null space $N(A^T) \subseteq \mathbb{R}^m$
- Column space of A is orthogonal to left null space $N(A^T) \perp C(A)$
- Row space A is orthogonal to null space $N(A^T) \perp N(A)$

Example: Obtain vectors belonging to each of the four subspaces of the matrix $A_{2 \times 3}$ and verify these subspaces properties. Column set of this matrix is dependent, and $\text{rank}(A) = 2 = m$, implies solution exists for $\forall b \in \mathbb{R}^2$. As $m < n$, more than one solution exists for $x = [c_1 \ c_2 \ c_3]^T$, and the number of free running variables would be $n - m = 3 - 2 = 1$. So, consider one the three variables as free and assign a scalar value $k \in \mathbb{R}$, $c_3 = k$:

Case 1: Solution would be:

Case 2: Solution for $Ax = o$ would be:

Now, consider the transpose of the previous matrix A^T whose dimensions would be 3×2 . Column set of the matrix is independent set, so $\text{rank}(A^T) = n = 2 < m$ implies that solution for $A^T x = b$ doesn't exist for $\forall b \in \mathbb{R}^3$.

Case 3: As $\text{rank}(A^T) = n$, solution is unique if it exists.

Solution for

Let's consider the solution for another value of b .

Solution for b doesn't exist as

Case 4: Let's consider the solution for $A^T x = \mathbf{0}$. As $\text{rank}(A^T) = n$; there exists a unique solution, and the solution is zero vector.

Let's summarize all four cases. Solutions obtained in these cases belong to different subspaces of matrix as explained below.

- **Case 1:**
- **Case 2:** For $k = 1$ solution becomes
- **Case 3:**
- **Case 4:** Solution is

Verify the orthogonality concepts $N(A^T) \perp C(A)$ & $C(A^T) \perp N(A)$ from the obtained vectors belonging to each of the subspace of matrix .

Determinant

Determinant of a square matrix A is a scalar value obtained from the entries of the square matrix denoted as $\det(A)$ or $|A|$. *Determinant* of $n \times n$ matrix A through Laplace expansion can be calculated using any fixed row number i and is defined as:

Where j is column number, $a_{i,j}$ is element in i^{th} row and j^{th} column of A , and $M_{i,j}$ called *minor*, is determinant of submatrix obtained by removing i^{th} row and j^{th} column of A . Term $C_{ij} = (-1)^{i+j} M_{ij}$ is called *Cofactor* of $a_{i,j}$ in matrix A . This operation is applied in recursion till the dimension of M reduces to 1×1 .

Inverse of Matrix

Cofactor matrix C of a matrix A_n consists of cofactors C_{ij} of elements a_{ij} .

Inverse of non-singular matrix A_n is defined using determinant and cofactor matrix as:

Example: Determinant of 2×2 matrix can be calculated using preceding formula. Let's fix row $i = 1$ then:

Verify that the determinants of the preceding matrix obtained by fixing row number $i = 2$ and $i = 1$ are the same?

Example: Determinant of 3×3 be can constructed using formula obtained from 2×2 matrix. Fix row $i = 1$:

Example: Find inverse of a matrix A :

Recursively, one can obtain the determinant for any $n \times n$ matrix. Determinants of any $n \times n$ square matrices and real scalar value $k \in \mathbb{R}$ have these properties:

- Determinant of identity matrix is 1, $\det(I_n) = 1$
- Determinant remains the same, but its sign (+/-) changes when two rows (two columns) are swapped
- Determinant doesn't change if scalar multiple of one row (column) is added to another row (column)
- Determinant is multiplied by k if row or column is multiplied by k
- $\det(kA) = k^n \det(A)$
- $\det(A) = 0$ iff A is non-invertible or has dependent rows (columns) or its $\text{rank}(A) < n$. In other words, $\det(A) \neq 0$ iff A is invertible
- If matrix is triangular, then its determinant is equal to product of diagonal elements
- $\det(AB) = \det(A) * \det(B)$
- $\det(A^T) = \det(A)$

Orthonormalization

Orthogonalization is the process of finding a set of orthogonal vectors that span a subspace. Consider independent vector set $\{v_1, v_2, \dots, v_k\}$ that spans subspace of inner product space in \mathbb{R}^n , where $k \leq n$. Orthogonalization is process of obtaining orthogonal vector set $\{w_1, w_2, \dots, w_k\}$ that spans same subspace as $\{v_1, v_2, \dots, v_k\}$. Further reduction on the derived orthogonal set can be performed to obtain orthonormal vector set $\{u_1, u_2, \dots, u_k\}$, where length/norm of all vectors is 1 and spans the same subspace as $\{v_1, v_2, \dots, v_k\}$. The process of obtaining orthonormal set for a given subspace is called *Orthonormalization*. *Gram-Schmidt* process is one of the orthonormalization techniques that derives orthonormal vectors through projection in an inner product space.

Understanding the projection of one vector over other through dot product will give better insights into the Gram-Schmidt process. As discussed in Euclidean space, dot product between two vectors $v_1 \cdot v_2$ provides the product of the first vector and the projected length of the second along the first vector. To obtain the ratio for the projection length with respect to the length of v_1 , divide it by the dot product of the first vector with itself (projection of the first vector with itself):

The ratio obtained is scalar; to obtain direction, multiply the ratio with v_1 as v_2 is projected along the direction of v_1 . Projection of vector v_2 over v_1 along the direction of v_1 can be defined as follows:

$\text{proj}_{v_1}(v_2)$ is a vector that is part of v_2 along the direction of v_1 ; the remaining part of v_2 is $v_2 - \text{proj}_{v_1}(v_2)$, which will be orthogonal to v_1 . With this, we have obtained two orthogonal vectors: $w_1 = v_1$ and $w_2 = v_2 - \text{proj}_{v_1}(v_2)$. To verify orthogonality, let's take dot product:

Geometrically, let's visualize the algorithm in two-dimensional form with two independent vectors, as shown in [Figure 2.15](#):

Figure 2.15: Projection of a vector along the direction of another vector through inner product

To obtain the third orthogonal vector w_3 from v_3 of the independent set, subtract the projection on w_1 and w_2 from v_3 , as follows:

Similarly, any i^{th} orthogonal vector is obtained as follows:

The obtained vector set $\{w_1, w_2, \dots, w_k\}$ is orthogonal but their length is not. To obtain vectors with unit length, divide the vector by their length:

Example: Consider a matrix A for orthonormalization with column vectors $\{v_1, v_2, v_3\}$:

One can verify that $\{w_1, w_2, w_3\}$ are orthogonal through dot product (approx. to three decimal places). Converting these vectors to unit vectors, one obtains:

Express these orthonormal vectors as matrix Q and verify if $Q^T Q = I$ (round off to 3 decimal places):

Tip: Simple modification of Gram-Schmidt is proposed to obtain i^{th} orthogonal vector for stable calculation. Instead of projecting vector on orthogonal vectors at once, it is performed in a chain. The resultant vector after projection over one orthogonal vector is used for projection on the next orthogonal vector.

Applications of Orthonormalization

Expressing a vector with respect to orthogonal vectors has advantages as changes to the vector along any orthogonal vector doesn't impact other orthogonal vectors. From a system perspective, one can consider these orthogonal vectors to be knobs, and tuning these knobs can be considered equivalent to changing independent factors of a system.

Consider the example of television set that has knobs to control brightness, contrast, length, and the width of frames. Tuning these knobs changes only one property of the frame and leaves the other properties unaffected. What if there existed a knob that would change brightness, contrast, length, and width of a frame simultaneously by a small percentage each. Will you be comfortable using this knob to achieve the desired frame quality? Undoubtedly not.

The same concept can be applied for tuning process of ML algorithms. Tuneable parameters of the algorithm must behave orthogonally; otherwise, analysing the behaviour of individual parameters would be difficult.

Orthonormalization is an important step to express a matrix as product of matrices that includes orthonormal matrix, which will be discussed in the later sections of this chapter. Next, let's discuss transforming a vector from one subspace to another.

Linear transformation

Linear transformation, also called *Linear map* or *Linear mapping*, is a mapping of vectors from linear vector subspace $V \subseteq \mathbb{R}^n$ to linear vector subspace $W \subseteq \mathbb{R}^m$, both over the same scalar field \mathbb{R} , denoted as $T : V \rightarrow W$. Additionally, it must preserve vector addition and scalar multiplication operations. To state formally, map $T : V \rightarrow W$ is linear if $\forall u, v \in V$ and scalar $s \in \mathbb{R}$ ensure that the following two conditions are satisfied:

- $T(u + v) = T(u) + T(v)$ preserving addition operation
- $T(su) = sT(u)$ preserving scalar multiplication

In other words, it doesn't matter whether linear map is applied before or after the operations of addition and scalar multiplication.

Example: Derivative operator and integral operator $\int dx$ of a function of single variable is linear transformation as both addition and scalar multiplication is preserved:

Example: Expectation of random variable is linear transformation:

Addition: $E[X + Y] = E[X] + E[Y]$; Scalar Mult: $E[2X] = 2E[X]$

Matrix associated with linear map

Linear transformations from finite dimensional vector space to another can be expressed by *Transformation Matrix* $A_{m \times n}$ with respect to the given basis vectors. How can you determine transformation matrix for the given linear transformation $T(\cdot)$?

Let subspace $V \in \mathbb{R}^m$ be spanned by basis vectors $B = [\mathbf{b}_1 \ \mathbf{b}_2 \dots \ \mathbf{b}_n]$ where $b_i \in \mathbb{R}^m$ (here, basis is ordered as it is required to define the coordinates of an element), vector $\mathbf{u} \in V$ expressed as the linear combination of its basis vectors with coefficients as $[\mathbf{u}]_B = [c_1 \ c_2 \ \dots \ c_n]^T$ (these are called coordinates of \mathbf{u} w.r.t. ordered basis B).

Apply the transformation function $T(\cdot)$ on \mathbf{u} with basis :

This indicates that applying transformation function $T(\cdot)$ on the vector \mathbf{u} is equivalent to the multiplication of A and $[\mathbf{u}]_B$, where $A = [T(\mathbf{b}_1) \ T(\mathbf{b}_2) \ \dots \ T(\mathbf{b}_n)]$ & $[\mathbf{u}]_B$ are coefficients required to express vector as linear combination of basis vectors $B = ([\mathbf{b}_1 \ \mathbf{b}_2 \ \dots \ \mathbf{b}_n])$.

One can view the multiplication $Av = \mathbf{u}$ of a matrix $A_{m \times n}$ and a vector $v \in \mathbb{R}^n$, which results in vector $\mathbf{u} \in \mathbb{R}^m$ through transformation. Matrix can be viewed as mapping (linear map) vector v of n -dimension subspace that is expressed as weights of linear combination of columns of matrix A to a vector \mathbf{u} of m -dimension.

Example: Transformation function $T(x)$ is the function that rotates a two-dimensional vector by $\theta = 90^\circ$:

Given

Let $\mathbf{u} = [14 \ 7]^T$ be expressed as linear combination of three vectors $\mathbf{b}_1 = [1 \ 2]^T$, $\mathbf{b}_2 = [2 \ 1]^T$ & $\mathbf{b}_3 = [3 \ 1]^T$ with coefficients $[\mathbf{u}]_B = [1 \ 2 \ 3]^T$ where $B = [\mathbf{b}_1 \ \mathbf{b}_2 \ \mathbf{b}_3]$.

For given transformation function, transformation matrix with respect to B is obtained by applying transformation function on each column of B .

obtain

Apply transformation matrix A on vector $[\mathbf{u}]_B$, which is equivalent to applying the transformation function $T(\cdot)$ on vector \mathbf{u} :

As transformation function rotates vector by 90° , vectors $\mathbf{u} = [14 \ 7]^T$ and linearly mapped vector $\mathbf{v} = [-7 \ 14]^T$ should be orthogonal. One can verify their orthogonality through dot product.

Tip: Transformation function $T(x) = kx$ scales a vector by factor k and can be represented by matrix kI_n .

Composition of linear transformation

Composition of linear transformations is a linear transformation. Let $T : V \rightarrow W$ be a linear transformation and $S : W \rightarrow X$ be another linear transformation, as depicted in [Figure 2.16](#). Then, the composition of the linear transformation is a linear transformation that maps $\mathbf{u} \in V$ directly to X by $S(T(\mathbf{u}))$. The important fact is that the matrix of the composite transformation is equal to the product of the matrices of the two original maps. Refer to the following figure:

[Figure 2.16: Composition of linear transformation](#)

Example of linear neural network: Consider a simple network of nodes, as shown in [Figure 2.17](#). Circles in the figure are called nodes. Nodes at level 1 are labelled n_{11} , n_{12} , and n_{13} , and at level 2, they are n_{21} and n_{22} . Output is provided through node n_{31} . The line connecting the first node n_{11} of layer 1 to the first node of layer 2 multiplies input n_{21} by weight w_{11}^1 . The line connecting the second node n_{12} of layer 1 to the first node n_{21} of layer 2 multiplies input x_2 by weight w_{21}^1 . Similarly, all weights are labelled based on levels and the nodes they are connecting. Refer to the following figure:

Figure 2.17: Simple neural network

Input to n_{21} from n_{11} would be $x_1 * w_{12}^1$. All inputs from nodes to n_{21} and n_{22} are summed as:

This can be written in matrix form as:

We call the vector \mathbf{n} the *hidden layer* of the network. So, the hidden layer \mathbf{n} is obtained by applying a linear transformation defined by matrix W . This transformation converts the three-dimensional input vector to a two-dimensional vector \mathbf{n} . Again, final output can be written as a linear transformation of \mathbf{n} to a real number.

Whole of the neural network above can be regarded as a composition of two linear transformations. As a composition of two linear transformations is also a linear transformation, we can represent this input output relation $x \rightarrow y$ as a single linear transformation $W^T x = y$, where $W = W^1 W^2$, that is, putting a linear hidden layer is redundant. This suggests that any number of linear layers added to the network doesn't change linearity of the network, and the whole network can be expressed as a single transformation matrix W . That's the reason a non-linear function is applied after every linear transformation of the input space while constructing deep neural network

(discussed in [chapter 7 Neural Networks](#)) with multiple layers. We will encounter many diverse network topologies in the later chapters.

Eigenvalues and vectors

In vector spaces, there might exist certain special non-zero vectors. Square transformation matrix A corresponding to linear transformation function $T(\cdot)$ when applied on these special non-zero vectors \mathbf{e} , results in changing the vector by a real scalar factor λ as $T(\mathbf{e}) = A\mathbf{e} = \lambda\mathbf{e}$. These vectors are called *Eigen vectors*, and the corresponding scaling factor λ by which eigen vector scales are called *Eigen values* of square matrix A .

$A\mathbf{e} = \lambda\mathbf{e}$ equation can be stated as $(A - \lambda I)\mathbf{e} = \mathbf{0}$ where \mathbf{e} is non-zero vector. This will help you to obtain the eigenvalues and vectors. Non-zero eigen vector \mathbf{e} will exist if and only if matrix $B = (A - \lambda I)$ has dependent columns/rows, or null space of should have non-zero vector. In other words, a non-zero solution for \mathbf{e} will exist if matrix B is non-invertible.

Any non-invertible matrix B will have its determinant as zero, that is, $|A - \lambda I| = 0$. This equation is called *Characteristic equation*, which is a polynomial function of the variable λ and degree of this polynomial is n (order of matrix A). Polynomial function is called *Characteristic polynomial* of A . Characteristic polynomial of degree n can be factored into product of n linear terms as $|A - \lambda I| = (\lambda_1 - \lambda)(\lambda_2 - \lambda) \dots (\lambda_n - \lambda)$, where λ_i 's are roots of the polynomial and are called eigenvalues of the matrix A . Eigenvalues of a matrix is not always unique; there can be repetitions. Eigenvalues of real matrix can also be complex numbers.

Once eigen values are calculated, the corresponding eigen vector are obtained using . The number of eigen vectors corresponding to one eigenvalue are infinite, and they can be represented concisely using scalar multiplication of vector (see the following examples).

Also, every square matrix A satisfies its own characteristics polynomial. This is known as *Cayley–Hamilton theorem*. So, if we substitute A in place of λ in the above-mentioned characteristic equation, we get a matrix equation equating to zero matrix.

Will eigen vectors corresponding to distinct eigenvalues always be independent?

Consider distinct eigenvalues $\lambda_1 \neq \lambda_2$ and corresponding eigen vectors as e_1, e_2 of matrix A. If $e_1 = ke_2$ then $Ae_1 = Ake_2 \Rightarrow \lambda_1 e_1 = \lambda_2 ke_2 \Rightarrow \lambda_1 = \lambda_2$, which leads to contradiction. So, eigen vectors corresponding to distinct eigenvalues will always be independent.

Eigen properties

Let's discuss a few important eigen properties:

- Eigen vectors corresponding to distinct eigenvalues are linearly independent.
- Trace of a matrix is equal to the sum of all its eigenvalues.
- Determinant of a matrix is equal to the product of all its eigenvalues.
- Eigenvalues of A^k , where k is positive integer, are λ_i^k , where λ_i is eigenvalue of A.
- If A is invertible, then eigenvalues of A^{-1} are $1/\lambda_i$, where λ_i is the eigenvalue of A.

Example: Consider 2x2 matrix and obtain its eigen values by solving its characteristic polynomial:

Use the obtained eigenvalues in $(A - \lambda I)e = 0$ to obtain corresponding eigen vectors. Select one among possible eigen vectors for $\lambda_1 = 4$ as $e_1 = [1 \ 1]^T$, as follows:

Select one among possible eigen vectors for $\lambda_2 = 2$ as $e_2 = [1 \ -1]^T$, as follows:

Verify if transformation on eigen vectors results in scaling of eigen vector by the corresponding eigenvalue:

, one obtains

, one obtains

Applying transformation on non-eigenvector $v = [1 \ 2]^T$, will it be scaled by $k \in \mathbb{R}$?

Example: Consider matrix . This matrix has repeating eigenvalues. Will repeating eigenvalues have independent eigen vectors?

As there are two free running variables, we can have two eigen vectors that are independent, that is, $e_1 = [1 \ 0]^T$ & $e_2 = [0 \ 1]^T$.

Example: Consider matrix . This matrix has repeating eigenvalues. Will repeating eigenvalues have independent eigen vectors?

Roots of $|A - \lambda I|$ will be $\lambda_1 = 1$, $\lambda_2 = -2$, $\lambda_3 = -2$. For $\lambda_1 = 1$, one obtains:

Applying Gauss-Jordan elimination method on this matrix will simplify the solution for $(A - \lambda I)\mathbf{e} = \mathbf{0}$.

For repeating eigenvalues $\lambda_2 = -2$, apply Gauss-Jordan elimination method on $A - (-2)I$. One obtains:

The number of free running variable is only one. So, two eigen vectors corresponding to the eigen value $\lambda = -2$ that are independent do not exist. Eigen vector would be $e_2 = [-3\alpha \ \alpha \ 0]^T$.

Geometric analysis

Analysing transformation on eigen vectors geometrically provides good insights for understanding the concept. Consider transformation matrix from the previous example. Eigenvalues and vectors of this matrix are:

For each eigenvalue, let's plot two eigen vectors with values in solid lines, as shown in [Figure 2.18](#):

Plot non-eigen vectors with values in dashed lines:

Refer to the following figure:

Figure 2.18: Representation of eigen vectors and non-eigen vectors

After transformation, eigen vectors denoted with thick black line keep their direction with length changed by factor of its eigen value and non-eigen vectors denoted with dotted grey line don't maintain their direction, as shown in [Figure 2.19](#):

Figure 2.19: Representation of eigen vectors and non-eigen vectors after transformation

Existence of zero eigenvalue

Under what conditions will a square matrix have zero eigenvalue?

Definition of eigen vector e states that it is a non-zero vector that changes by a scalar factor λ when transformation A is applied. The corresponding scalar factor is called its eigenvalue. From definition:

As e is non-zero vector, $N(A)$ should have non-zero vectors, which is possible only when $\text{rank}(A_n) < n \Rightarrow$ matrix has at least one dependent column/row. Repetition of eigenvalue $\lambda = 0$ will happen if dimension of null subspace is greater than zero, that is, $\text{dimension}(N(A)) > 0$. Also, the number of times zero eigenvalue repeats will be equal to $\text{dimension}(N(A))$.

Eigen properties of symmetric matrices

Symmetric matrices are important due the following properties of eigenvalues and eigenvectors:

- Eigenvalues of real symmetric matrices are always real.

- There exist exactly n eigen values (need not be distinct) for symmetric matrix A_n .
- There exist n eigen vectors corresponding to each eigenvalue that are mutually orthogonal.

Example: In the preceding example, eigenvalues and vectors of a symmetric matrix resulted in:

Verify if eigen values corresponding to different eigenvalues are orthogonal.

Example: Consider another real symmetric matrix that has repeating eigenvalues. Two orthogonal eigen vectors can be obtained for this repeating eigen value. One can verify the orthogonality of eigen vectors.

Positive definite

Quadratic form is a function Q from \mathbb{R}^n to \mathbb{R} defined with respect to a symmetric matrix A as:

Consider the example of symmetric matrix ; the corresponding quadratic form in \mathbb{R}^2 is given by:

Note that $Q(\mathbf{x})$ is a scalar or a real number for real vector \mathbf{x} . Real symmetric matrix A is called *positive-definite* if the real number $Q(\mathbf{x}) = \mathbf{x}^T A \mathbf{x}$ is positive for every non-zero real column vector \mathbf{x} . A positive-definite matrix is denoted by the notation $A \succcurlyeq 0$.

Let \mathbf{v} be any eigen vector (it is always non-zero) of the positive-definite matrix A corresponding to the eigenvalue λ . Then, we have $A\mathbf{v} = \lambda\mathbf{v}$. Hence, $Q(\mathbf{v}) = \mathbf{v}^T (\lambda\mathbf{v}) = \lambda\mathbf{v}^T \mathbf{v} = \lambda\|\mathbf{v}\|^2 > 0$. Here, denotes the Euclidean norm and $\|\mathbf{v}\|^2 > 0$ always. Therefore, $\lambda > 0$ that is, the eigenvalues of a positive definite matrix are all positive.

If $A \succcurlyeq 0$, then set $\varepsilon = \{\mathbf{x} : Q(\mathbf{x}) < 1\}$ is called an *ellipsoid* in \mathbb{R}^n . An *ellipsoid* is a generalization of ellipse in n -dimensions. A two-dimensional ellipse has

two axes: one is called *major axis* and the other the *minor axis*, as shown in [Figure 2.20](#):

Figure 2.20: Ellipsoid

Eigenvalues λ_i and corresponding eigenvectors \mathbf{q}_i of matrix $A \geq \mathbf{0}$ determine the direction and length of semiaxes s_i of ellipsoid ε as:

If we arrange the eigen vectors $\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_n$ corresponding to eigenvalues in decreasing order as $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n$, the value of $\mathbf{x}^T A \mathbf{x}$ is large in the direction of \mathbf{q}_1 . Hence, ellipsoid is thin and elongated in the direction of \mathbf{q}_1 . This is the direction of major axis. In direction of \mathbf{q}_n , the value of $\mathbf{x}^T A \mathbf{x}$ is small, so ellipsoid is fat in direction of \mathbf{q}_n . We will encounter these ellipsoids again in [Chapter 4 Basic Statistics and Probability Theory](#) while introducing multivariate Gaussian distribution.

Using quadratic form, we can also define another norm for a matrix. For any $m \times n$ matrix A , $A^T A$ is always a square symmetric matrix. We can calculate the eigen values of $A^T A$. Let λ be any eigen value of $A^T A$; then, we have $A^T A \mathbf{x} = \lambda \mathbf{x}$ for the corresponding eigen vector \mathbf{x} . The *spectral norm* of a matrix A is defined as:

We have:

Hence:

Therefore, . The *spectral norm* of a matrix A is the largest *singular value* of A . *Singular value* of matrix A is defined as the square root of non-zero eigenvalue of $A^T A$ matrix.

Now, any vector \mathbf{v} can be also viewed as a matrix with one column. Hence, we can define spectral norm for any vector. But $\mathbf{v}^T \mathbf{v}$ is the dot product, and hence, is a scalar. The eigenvalue of a scalar or 1×1 matrix is the scalar itself.

So, we can write spectral norm of vector v as , which is actually the Euclidean norm of the vector v . Using spectral norm definition, we can write:

So, if we consider the linear transformation defined by the map $T:x \rightarrow Ax$, the length of the mapped vector Ax is, at most, λ_1 times the length of the input vector x , where λ_1 is the maximum singular value of the matrix A .

Matrix decomposition

The process of expressing a matrix as product of matrices is called *matrix decomposition* or *matrix factorization*. There exist many methods to decompose the matrix. Each method finds use in particular class of problems. The main advantage of matrix decomposition is to express a matrix in a form that helps to solve the problem.

Computing inverse of a matrix, if exists, might be compute intensive. But when expressed as a product of matrices, it could be done with comparatively less compute resources. Let say, matrix A can be expressed as the product of any three matrices P , Q , R whose inverse calculation is simple. So, decomposing a matrix into appropriate matrices can result in lesser compute:

Similarly, calculating the determinant of a matrix becomes easier when a matrix is expressed as the product of any three matrices P , Q , R , whose determinants are easier to calculate.

LU decomposition

LU decomposition is process of expressing the square matrix as the product of the lower and upper triangular matrices $A = LU$:

From the product, $a_{11} = l_{11} * u_{11}$, there arise many possible values for l_{11} & u_{11} . To obtain consistency in decomposition, the diagonals of lower triangular are assigned 1. In the following example, the value of $l_{xx} = 1$. With fixed value of l_{xx} , we can find other values as:

Use obtained values, $u_{11} = 1$ to solve $l_{21} u_{11} = 3$ we get $l_{21} = 3$

Use obtained values, $u_{12} = 3$, $l_{21} = 3$ to solve $l_{21} u_{12} + l_{22} u_{22} = 1$ we get $u_{22} = -8$

Similarly, one can obtain values for all entries of L & U matrices.

$$\begin{bmatrix} 1 & 3 & 1 \\ 3 & 1 & 2 \\ 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} =$$

$$\begin{bmatrix} l_{11}u_{11} & l_{11}u_{12} & l_{11}u_{13} \\ l_{21}u_{11} & l_{21}u_{12} + l_{22}u_{22} & l_{21}u_{13} + l_{22}u_{23} \\ l_{31}u_{11} & l_{31}u_{12} + l_{32}u_{22} & l_{31}u_{13} + l_{32}u_{23} + l_{33}u_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 1 & 1/8 & 1 \end{bmatrix} \begin{bmatrix} 1 & 3 & 1 \\ 0 & -8 & -1 \\ 0 & 0 & 17/8 \end{bmatrix}$$

What happens when the value of $a_{11} = 0$?

When $a_{11} = 0$, $a_{11} = l_{11} * u_{11} \Rightarrow l_{11} = 1$, $u_{11} = 0$, but this situation is impossible if $\text{rank}(A) = n$ as assigning $u_{11} = 0$ will lead to $\text{rank}(U) < n$ that leads to $\text{rank}(A) < n$. This situation can be avoided if one shuffles the rows of matrix A. Shuffling or permutation of the rows can be performed through permutation matrix P , as discussed in *Matrices* section. Permutated rows of matrix A will be decomposed as $PA = LU$. Every square matrix can be decomposed into this form:

$$PA = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 3 & 1 \\ 3 & 1 & 2 \\ 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 0 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 0 & -3/5 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -5 & -7 \\ 0 & 0 & -16/5 \end{bmatrix}$$

One can assign all diagonal entries of U matrix with 1 by introducing the diagonal matrix D that has 0 entries in non-diagonal elements between the L and U matrices.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 0 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 0 & -3/5 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -5 & 0 \\ 0 & 0 & -16/5 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & -7 \\ 0 & 0 & 1 \end{bmatrix} = LDU$$

By-product of Gauss-Jordan elimination

LU decomposition can be viewed as by-product of Gauss-Jordan elimination. Elimination method consists of two stages forward & back substitution. Representing forward substitution in matrix format provides *L* matrix.

Forward substitution: We can represent each i^{th} step of forward substitution method by elementary matrix E_i . *Elementary matrix* differs from identity matrix by one single elementary row operation. Suppose the row operation to be conducted on a 3×3 matrix A is $2 * r_2 + r_3 \rightarrow r_3$, then elementary matrix E representing this operation is:

$$E = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix}$$

Multiplying E on the left side of matrix A is equivalent to performing row operation of $2 * r_2 + r_3 \rightarrow r_3$ on A .

E_1 is the matrix representing the first elementary row operation to be conducted on A , E_2 will represent the second, and E_i will represent the i^{th} row operation. All steps of forward substitution (row operations) can be represented as a single matrix $E = E_\alpha \dots E_2 E_1$. The resultant matrix of forward substitution is upper triangular matrix U .

$$EA = E_\alpha \dots E_2 E_1 A = U \Rightarrow A = E^{-1}U = LU \Rightarrow L = E^{-1}$$

Example: Consider the matrix A . In forward substitution, a total of three operations are performed to obtain row echelon matrix. Operation on step 1: $(-3) * r_1 + r_2 \rightarrow r_2$ & $(-5) * r_1 + r_3 \rightarrow r_3$ and on step 2: $2 * r_2 + r_3 \rightarrow r_3$. Operation of step is represented with matrix E_i .

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 3 & 4 & 2 \\ 5 & 3 & 5 \end{bmatrix}; E_1 = \begin{bmatrix} 1 & 0 & 0 \\ -3 & 1 & 0 \\ -5 & 0 & 1 \end{bmatrix}; E_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix}; E = E_2E_1$$

$$EA = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ -3 & 1 & 0 \\ -5 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 3 & 4 & 2 \\ 5 & 3 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -3 & 1 & 0 \\ -11 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 3 & 4 & 2 \\ 5 & 3 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \\ 0 & 0 & -2 \end{bmatrix} = U$$

$$L = E^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 5 & -2 & 1 \end{bmatrix}$$

QR decomposition

QR decomposition is about decomposing a matrix $A_{m \times n}$, where $m \geq n$ & $\text{rank}(A) = n$ into the product of two matrices as $A = QR$, where Q is orthogonal matrix and R is upper triangular matrix. Orthogonal matrices have properties like $Q^{-1} = Q^T$ that are helpful in solving certain classes of problems. Matrix Q can be obtained from A through one of the various orthonormalization algorithms called Gram-Schmidt. Let's represent A & Q with column vectors as follows:

$$A = [\mathbf{c}_1 \quad \mathbf{c}_2 \quad \cdots \quad \mathbf{c}_n]; Q = [\mathbf{q}_1 \quad \mathbf{q}_2 \quad \cdots \quad \mathbf{q}_n]$$

Matrix R ($A = QR \Rightarrow Q^{-1} A = Q^{-1} QR = R$) can be expressed as the dot product of columns of Q & A :

$$R = Q^T A = \begin{bmatrix} \mathbf{q}_1^T \\ \mathbf{q}_2^T \\ \vdots \\ \mathbf{q}_n^T \end{bmatrix} [\mathbf{c}_1 \quad \mathbf{c}_2 \quad \cdots \quad \mathbf{c}_n] = \begin{bmatrix} \mathbf{q}_1 \cdot \mathbf{c}_1 & \mathbf{q}_1 \cdot \mathbf{c}_2 & \cdots & \mathbf{q}_1 \cdot \mathbf{c}_n \\ \mathbf{q}_2 \cdot \mathbf{c}_1 & \mathbf{q}_2 \cdot \mathbf{c}_2 & \cdots & \mathbf{q}_2 \cdot \mathbf{c}_n \\ \vdots & \vdots & & \vdots \\ \mathbf{q}_n \cdot \mathbf{c}_1 & \mathbf{q}_n \cdot \mathbf{c}_2 & \cdots & \mathbf{q}_n \cdot \mathbf{c}_n \end{bmatrix}$$

Example: Consider the example of a matrix A and obtain orthonormal matrix Q through Gram-Schmidt process. Verify if $A = QR$ (round to 2 decimal places):

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \end{bmatrix}; Q = \begin{bmatrix} 0.267 & 0.524 & 0.808 \\ 0.801 & -0.586 & 0.115 \\ 0.534 & 0.617 & -0.576 \end{bmatrix}; R = Q^T A = \begin{bmatrix} 3.74 & 2.94 & 2.94 \\ 0 & 2.31 & 1.02 \\ 0 & 0 & 2.08 \end{bmatrix}$$

Example: Consider rectangular matrix $A_{4 \times 3}$ whose rank is 3. Obtain orthonormal matrix Q through Gram-Schmidt process:

$$A = \begin{bmatrix} 1 & -1 & 4 \\ 1 & 4 & -2 \\ 1 & 4 & 2 \\ 1 & -1 & 0 \end{bmatrix}; Q = \begin{bmatrix} 0.5 & -0.5 & 0.5 \\ 0.5 & 0.5 & -0.5 \\ 0.5 & 0.5 & 0.5 \\ 0.5 & -0.5 & -0.5 \end{bmatrix}; R = Q^T A = \begin{bmatrix} 2 & 3 & 2 \\ 0 & 5 & -2 \\ 0 & 0 & 4 \end{bmatrix}$$

Eigen decomposition

Consider a square matrix $S_{n \times n}$ that has n linearly independent eigenvectors e_1, e_2, \dots, e_n corresponding to eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_n$. Let $Q = [e_1 \ e_2 \ \dots \ e_n]$, then:

$$SQ = [Se_1 \ Se_2 \ \dots \ Se_n] = [\lambda_1 e_1 \ \lambda_2 e_2 \ \dots \ \lambda_n e_n] = Q\Lambda$$

$$\text{where } \Lambda = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}$$

Λ is diagonal matrix with corresponding eigenvalues. Multiply by Q^{-1} on the right side of equation $SQ = Q\Lambda$. This provides $SQQ^{-1} = Q\Lambda Q^{-1} \Rightarrow S = Q\Lambda Q^{-1}$.

Eigen decomposition helps in reducing complexity while solving many problems, like finding inverse of a matrix if it exists or power of a matrix S^{-1} as:

$$S^{-1} = Q\Lambda^{-1}Q^{-1} \quad \& \quad S^2 = Q\Lambda Q^{-1}Q\Lambda Q^{-1} = Q\Lambda^2 Q^{-1}$$

Real symmetric matrix

As discussed in the eigen value section, real square symmetric matrix has n eigen real values and corresponding eigenvectors are mutually orthogonal. This property helps to obtain interesting format of eigen decomposition as $S = Q\Lambda Q^{-1} = Q\Lambda Q^T$ as Q is orthogonal matrix $Q^{-1} = Q^T$.

Example: Eigenvalues of matrix $A = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$ are $\lambda_1 = 2$, $\lambda_2 = 4$ and corresponding unit eigen vectors will be $e_1 = [1/\sqrt{2} \ -1/\sqrt{2}]^T$, $e_2 = [1/\sqrt{2} \ 1/\sqrt{2}]^T$. Verify if the following decomposition is valid:

$$\begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ -1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix} \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix}$$

Singular value decomposition

Singular value decomposition is generalization of Eigen decomposition to rectangular matrices. Eigenvalue λ and corresponding eigen vector e of a square matrix S followed $Se = \lambda e$. As $S_{n \times n}$ is square matrix, the multiplied vector and resultant vector belong to \mathbb{R}^n , and they are the same (all in the same dimensional subspace). Consider a rectangular matrix $A_{m \times n}$, when multiplied by vector $v \in \mathbb{R}^n$, results in vector $u \in \mathbb{R}^m$. As the multiplied and resultant vectors belong to different dimensional subspace, one needs to map in both ways using A (maps $\mathbb{R}^n \rightarrow \mathbb{R}^m$) and A^T (maps $\mathbb{R}^m \rightarrow \mathbb{R}^n$).

$$Av = \sigma u \quad \& \quad A^T u = \sigma v$$

where non-negative scalar value $\sigma \geq 0$ is called *singular value* of A , u is *left-singular* vector of σ and v is *right-singular* vector of σ .

Decompose rectangular matrix into product of three matrices:

$$A_{m \times n} = U_{m \times m} \Sigma_{m \times n} V_{n \times n}^T$$

where $U_{m \times m} = [u_1 \ u_2 \ \cdots \ u_m]$, $V_{n \times n} = [v_1 \ v_2 \ \cdots \ v_n]$, Σ is diagonal matrix with $\sigma_i \geq 0$ values such that $\sigma_i \geq \sigma_{i+1}$, $Av_i = \sigma_i u_i$, U & V are orthogonal matrices.

How can we find these singular-values and corresponding left-singular and right-singular vectors of a matrix that are orthonormal in their respective subspaces?

Any matrix multiplied by its transpose results in symmetric matrix. This property will help in providing the relation between singular-values and eigenvalues. U & V are orthogonal matrices implying $V^T V = I$ & $U^T U = I$. As Σ is diagonal matrix, $\Sigma^T = \Sigma$:

$$AA^T = U\Sigma V^T(U\Sigma V^T)^T = U\Sigma V^T V\Sigma^T U^T = U\Sigma^2 U^T$$

$$A^T A = (U\Sigma V^T)^T U\Sigma V^T = V\Sigma^T U^T U\Sigma V^T = V\Sigma^2 V^T$$

Symmetric matrix $AA^T = U\Sigma^2 U^T$ can be mapped to eigen decomposition of symmetric matrix $S = Q\Lambda Q^T$, where both U & Q are orthogonal matrix and Σ^2 & Λ are diagonal matrices. This implies that σ_i^2 are eigenvalues of symmetric matrix $[AA^T]_{m \times m}$, columns of U (left-singular vectors \mathbf{u}_i) are orthogonal eigenvectors corresponding to eigenvalues of symmetric matrix AA^T .

Similar analysis holds on symmetric matrix $A^T A = V\Sigma^2 V^T$. σ_i^2 are eigenvalues of symmetric matrix $[A^T A]_{n \times n}$, columns of V (right-singular vectors \mathbf{v}_i) are orthogonal eigenvectors corresponding to eigenvalues of symmetric matrix $A^T A$.

Example: Decompose square matrix $A = \begin{bmatrix} 5 & 5 \\ -1 & 7 \end{bmatrix}$. Calculate the eigenvalues and eigen vectors of $A^T A$.

$$A^T A = \begin{bmatrix} 26 & 18 \\ 18 & 74 \end{bmatrix}$$

$$\lambda_1 = 20 = \sigma_1^2 \text{ & } \mathbf{e}_1 = \mathbf{v}_1 = \begin{bmatrix} -3/\sqrt{10} \\ 1/\sqrt{10} \end{bmatrix}; \lambda_2 = 80 = \sigma_2^2 \text{ & } \mathbf{e}_2 = \mathbf{v}_2 = \begin{bmatrix} 1/\sqrt{10} \\ 3/\sqrt{10} \end{bmatrix}$$

$$V = [\mathbf{v}_1 \quad \mathbf{v}_2] = \begin{bmatrix} -3/\sqrt{10} & 1/\sqrt{10} \\ 1/\sqrt{10} & 3/\sqrt{10} \end{bmatrix}; \Sigma = \begin{bmatrix} 2\sqrt{5} & 0 \\ 0 & 4\sqrt{5} \end{bmatrix}$$

$$AV = U\Sigma \Rightarrow AV = \begin{bmatrix} -\sqrt{10} & 2\sqrt{10} \\ \sqrt{10} & 2\sqrt{10} \end{bmatrix}$$

Obtained unit orthogonal vectors \mathbf{v}_i from $A^T A$. To obtain U , use $\mathbf{u}_i = 1/\sigma_i A\mathbf{v}_i$. There is no restriction to first obtain \mathbf{v}_i . One can first obtain \mathbf{u}_i from AA^T and derive $\mathbf{v}_i = 1/\sigma_i A^T \mathbf{u}_i$.

$$\begin{bmatrix} -\sqrt{10} & 2\sqrt{10} \\ \sqrt{10} & 2\sqrt{10} \end{bmatrix} = \begin{bmatrix} -1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix} \begin{bmatrix} 2\sqrt{5} & 0 \\ 0 & 4\sqrt{5} \end{bmatrix}; U = \begin{bmatrix} -1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix}$$

$$A = \begin{bmatrix} 5 & 5 \\ -1 & 7 \end{bmatrix} = U\Sigma V^T = \begin{bmatrix} -1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix} \overset{\text{Verity}}{\begin{bmatrix} 2\sqrt{5} & 0 \\ 0 & 4\sqrt{5} \end{bmatrix}} \begin{bmatrix} -3/\sqrt{10} & 1/\sqrt{10} \\ 1/\sqrt{10} & 3/\sqrt{10} \end{bmatrix}?$$

Example: Consider a rectangular matrix $A = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{bmatrix}$. In the previous example of a square matrix, one could start with either $A^T A$ or $A A^T$. But in case of a rectangular matrix, choose the one that results in more columns/rows. If rectangular matrix $A_{m \times n}$ has $m < n$, then choose $A^T A$ otherwise choose $A A^T$. Calculate the eigenvalues and vectors of $A^T A$:

$$A^T A = \begin{bmatrix} 10 & 5 & 9 \\ 5 & 5 & 8 \\ 9 & 8 & 13 \end{bmatrix}; \lambda_1 = 25 = \sigma_1^2, \mathbf{e}_1 = \begin{bmatrix} 4/\sqrt{5} \\ 3/\sqrt{5} \\ 5/\sqrt{5} \end{bmatrix} = \mathbf{v}_1$$

$$\lambda_2 = 3 = \sigma_2^2, \mathbf{e}_2 = \begin{bmatrix} -2/\sqrt{6} \\ 1/\sqrt{6} \\ 1/\sqrt{6} \end{bmatrix} = \mathbf{v}_2; \lambda_3 = 0 = \sigma_3^2, \mathbf{e}_3 = \begin{bmatrix} -1/\sqrt{3} \\ -7/\sqrt{3} \\ 5/\sqrt{3} \end{bmatrix} = \mathbf{v}_3$$

One obtains:

$$\Sigma = \begin{bmatrix} \sqrt{\sigma_1^2} & 0 & 0 \\ 0 & \sqrt{\sigma_2^2} & 0 \end{bmatrix} = \begin{bmatrix} 5 & 0 & 0 \\ 0 & \sqrt{3} & 0 \end{bmatrix}, V = \begin{bmatrix} 4/\sqrt{5} & -2/\sqrt{6} & -1/\sqrt{3} \\ 3/\sqrt{5} & 1/\sqrt{6} & -7/\sqrt{3} \\ 5/\sqrt{5} & 1/\sqrt{6} & 5/\sqrt{3} \end{bmatrix}$$

Use $\mathbf{u}_i = 1/\sigma_i A \mathbf{v}_i$ or $A V = U \Sigma$:

$$A\mathbf{v}_1 = \begin{bmatrix} 5/\sqrt{2} \\ 5/\sqrt{2} \end{bmatrix} \Rightarrow \mathbf{u}_1 = \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}; A\mathbf{v}_2 = \begin{bmatrix} 3/\sqrt{6} \\ -3/\sqrt{6} \end{bmatrix} \Rightarrow \mathbf{u}_2 = \begin{bmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{bmatrix}$$

$$U = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix}$$

Verify if $A = U\Sigma V^T$.

SVD is applied in various domains, like data compression and recommendation systems. We will be using eigen value decomposition and SVD in various concepts in the subsequent chapters. These topics will include dimensionality reduction, semantic representation of text, or latent semantic indexing.

Conclusion

The discussion of linear algebra in this chapter started with introduction of system of linear equations and its solutions with real examples. Representing large data with linear equations has its limitations. For better representation, we introduced vectors and matrices. We discussed the types of spaces and properties whose elements are vectors. Matrices and their properties were discussed, which helped analyze solutions to system of linear equations through matrix subspaces. Next, we discussed linear transformation and various matrix decomposition techniques. Matrix decomposition techniques help in reducing dimensionality of the data, which will be discussed in [Chapter 8, Dimensionality Reduction](#). The concepts discussed in this chapter will lay the foundation for the AI algorithms discussion.

In the next [chapter 3, Vector Calculus](#), we will discuss differentiation and integration of vectors and their optimization.

Points to remember

- Real-world objects can be represented mathematically as vectors in vector space. Properties of real-world environment in which the objects interact will help us to define the properties of the vector space.

- Matrices help us to represent vector space concisely.
- Selecting appropriate basis for a vector space makes some mathematical operations simple.
- Matrix multiplication with a vector is viewed as linear transformation of a vector from one vector space to another. This is important step for a few ML algorithms as it reduces dimensionality of the data, which helps in reducing the effects of *Curse of Dimensionality*.
- Decomposing a matrix is important step for various ML algorithms.

Further Reading

Linear algebra is an old branch of mathematics, so one can find numerous resources through books, online courses, and webpages, as follows:

- Introduction to Linear Algebra book by Prof. Gilbert Strang provides information on concepts and its applicability. Video lecture series by the professor is also available (<http://web.mit.edu/18.06/www/videos.shtml>).
- YouTube channel provides explanation for a few concepts (<https://www.youtube.com/c/MathTheBeautiful>).
- For mathematical definitions, one can refer to <https://www.wolframalpha.com/>.

CHAPTER 3

Vector Calculus

Vector calculus is the study of vector fields and scalar fields. A scalar field is a mapping that assigns a scalar or a real number to every point in a vector space. A vector field assigns a new vector to each point in a vector space. Vector calculus includes computation of derivatives of scalar and vector fields and integrals over scalar fields. These are the essential tools required in defining optimisation theory and parameter estimation in probability theory, which are the foundational pillars of ML/AI. The theory of vector calculus is a generalization of the calculus of functions of single variable to functions of several variables.

We will first introduce the fundamentals of real analysis, that is, real valued functions that are easy to visualize. We will elaborate on the differentiability concepts as they are the basis of all function optimisation theory discussed later. Then, we will see how these concepts are generalized for functions of several variables: scalar and vector fields.

Structure

In this chapter, we will cover the following topics:

- Fundamentals of real analysis
- Scalar and vector fields
- Tensors and tensor fields
- Total derivative, partial derivative, derivatives with respect to tensors
- Introduction to function optimization
- Convex functions, Lagrange multipliers
- Gradient descent algorithm

Objectives

This chapter introduces fundamental mathematical background required for understanding deeper concepts in ML. For readers who are fresh college graduates, most of these concepts may be a refresher for them. The main goal of this chapter is to introduce the mathematical optimization theory and algorithms to solve mathematical optimization problems in general.

Analysis of real functions

Real analysis is the area of mathematics dealing with real numbers and the properties of real-valued functions and sequences. In this section, we will cover a few topics of real analysis, specifically, real functions that are relevant for ML/AI. We will start with the real line definition.

The *rational numbers* are numbers that can be represented exactly by a ratio of any two integers. There exist numbers that cannot be represented in this form, for example, there is no rational number whose square is 2, that is, $\sqrt{2}$ is an *irrational number*. The set of all rational and irrational numbers is called the set of *real numbers* and is denoted by \mathbb{R} . Geometrically, the set of points can be represented by a line with no beginning and no end. There exists a real number that is smaller than the smallest number you can think of. Also, there exists a real number that is greater than the greatest number we can think. The set is dense, that is, between any two real numbers a and b however close one can think of, there exists another real number c such that $a < c < b$. The set of integers is not dense, as there are no integers between two consecutive integers.

Definition: Let $a, b \in \mathbb{R}$, $a < b$. An *open interval* I is a subset of \mathbb{R} represented by $I = (a, b) = \{x \in \mathbb{R} : a < x < b\}$. The entire real line is denoted by the open interval $(-\infty, +\infty)$. Similarly, a *closed and bounded interval* is defined by the set by $I = [a, b] = \{x \in \mathbb{R} : a \leq x \leq b\}$.

Definition of (ϵ -neighbourhood): A symmetric open interval around the point $x_0 \in \mathbb{R}$: $(x_0 - \epsilon, x_0 + \epsilon)$, $\epsilon > 0$ is called a ϵ -neighbourhood at x_0 for a given ϵ . Choosing ϵ arbitrarily small positive number, we can get points that are very close to x_0 and then study the function behaviour in the close locality of x_0 . Formally, ϵ -neighbourhood is represented as a set of all real number x such that $|x - x_0| < \epsilon$. Refer to the following figure:

Figure 3.1: Neighborhood in real line

Definition of interior point: Let . A point is said to be interior point of S if there exists a neighbourhood , such that . If each point of S is an interior point, then the set S is called *open set*. Finite intersection and arbitrary union of open sets is open.

Definition of upper and lower bounds, sup and inf of subset of : A real number M is called an *upper bound* for if $x \leq M$ for all $x \in S$. The set S is said to be *bounded above* if it has an upper bound. Similarly, we can define *lower bound*. A real number α is called the least upper bound (supremum/sup) of S if (i) α is an upper bound for S and (ii) there does not exist an upper bound for S that is strictly smaller than α . If supremum exists, is unique, and is denoted by *sup S*. The greatest lower bound (or infimum or inf) is defined analogously and denoted by *inf S*.

Note: The largest element of a set is called maximum element. Sup S is not same as Maximum(S). In fact, every set (bounded above) may not even have a maximum. For example, $S = \{1 - , n = 1, 2, \dots\}$ has no maximum element. 1 is an upper bound of the set, but $1 \notin S$. However, Sup S = 1.

Completeness axiom: Any nonempty subset of that is bounded above has a least upper bound or supremum. Similarly, any nonempty subset of bounded from below has an infimum.

Now, let's define a real function. The idea of function is to mathematically represent how a varying quantity depends on another quantity. For example, the position of a planet is a function of time.

Definition of real valued function of single variable: A function f of a real variable is a mapping that assigns a real number $f(x)$ to each real number x in the domain of the function. The *domain of the function* is the subset of the real line where the function is defined to be valid and is denoted by , where is the domain of function.

Example: $f(x) = \sqrt{x}$; the domain D of this function is $D = ^+$. Each positive real number is mapped to its square root by this function.

Definition: (*Composition of functions*) Let and be a function. Let be a function on E , where . Then, for each , $f(x) \in E$, and therefore, $g(f(x)) \in$.

The function such that $h(x) = g(f(x))$, $x \in D$. Then, h is said to be the composite function of f, g and the function h is denoted as $g \circ f$.

Limit of a function

Let $f(x) = \frac{1}{x-1}$.

This function is not defined at $x=1$ as the denominator vanishes at $x=1$. But as we go closer to 1, what value does the function take? This is answered by the limit of the function f as $x \rightarrow 1$, represented as $\lim_{x \rightarrow 1} f(x)$.

Definition: (*Limit of a function* at a given point a) L is called the limit of a function at a given point if, for any chosen δ -neighbourhood of L , (choose however small), there exists a ϵ -neighbourhood of a such that for all x in ϵ -neighbourhood of a , $f(x)$ is in δ -neighbourhood of L that is for all x , $|x - a| < \epsilon \Rightarrow |f(x) - L| < \delta$ and we write the following:

$$\lim_{x \rightarrow a} f(x) = L$$

Now, as for any point x in neighbourhood of 1 and $x \neq 1$, we can rewrite:

as

Let $L = 2$. Applying the preceding definition, if we choose a $\delta > 0$, then for any x such that $|x - 1| < \delta$, we have $|f(x) - L| = |x + 1 - 2| = |x - 1| \leq \delta$. Therefore, $\lim_{x \rightarrow 1} f(x) = 2$. Refer to the following figure:

Figure 3.2: Plot of

Let's take another example:

This function is not defined at $x=0$. From the plot, it seems that the limit of the function at $x=0$ may be $L=0$. Choosing a small neighbourhood of $L=0$:

Taking, $\epsilon = \delta$ we have:

Continuous functions

A function $f(x)$ is said to be *continuous at a point x_0* in the domain of f if it maps points close to x_0 in the domain of f to close by points in the range.

To understand this better, let's first consider an example for non-continuous functions that maps close by points to far off points.

, the sign functions.

Take two points $a > 0, b < 0$ in the open interval $(-\epsilon/2, \epsilon/2)$, $\epsilon > 0$. Choosing ϵ arbitrarily small, we have $|a-b| \leq |a|+|b| = \epsilon$ but $f(a) = 1$ and $f(b) = -1$. So, $f(a)$ and $f(b)$ are very far off, even though are very close. Hence, f is not continuous at the point 0.

A *function is said to be continuous* if it's continuous at every point in its domain, that is, it always maps nearby points to nearby values. We can represent this formally using the neighbourhood.

Definition (Continuous function) Let $f: D \rightarrow \mathbb{R}$, where $D \subseteq \mathbb{R}$, and suppose that $c \in D$. Then, f is continuous at c if for every chosen ϵ -neighbourhood of c , there exists a δ -neighbourhood of $f(c)$ such that: Any x in ϵ -neighbourhood of c is mapped to some point $f(x)$ in δ -neighbourhood of $f(c)$, that is, $|x - c| < \epsilon$ and $x \in D \Rightarrow |f(x) - f(c)| < \delta$.

This is same as the limit definition if we write $L = f(c)$. So, we say a function is continuous at c if $f(x) \rightarrow f(c)$. For example, the function $f(x) = |x|$, $x \in \mathbb{R}$ is continuous function on since for any

A function is called *continuous* if it's continuous at every point of its domain.

Following are a few important theorems for continuous functions that we will state without proof:

Bolzano theorem: Let I be a closed and bounded interval and f be continuous on I . If $f(a)$ and $f(b)$ are of opposite signs, then there exists at least one point c in I such that $f(c) = 0$. A more generalized result is given by the following theorem:

Intermediate value theorem: Let I be a closed and bounded interval and f be continuous on I . If $f(a) < L < f(b)$, then f attains every value in the interval $(f(a), L)$ at least once in I .

Figure 3.3 explains these theorems:

Figure 3.3: Theorems for continuous functions. (**Left**) Here $u \in (f(a), f(b))$. We found a point $c \in [a, b]$ such that $f(c) = u$. (**Right**) $f(a) < 0$ and $f(b) > 0$ so we found a point $c \in [a, b]$ such that $f(c) = 0$

Derivative of a function

Derivative of a function at a point represents the rate of change of the function at that point. The process of finding the derivative is called *differentiation*.

Definition: Right-derivative of a function at where is a point in the domain of the function is defined as the rate at which the function changes in the right proximity of the point . The *rate of change* is the ratio of the change in function value for a small change in the value of towards the right side of in the axis, say , where $h(>0)$ is small. The rate of change is given by:

This ratio represents the slope of the secant line, as shown in [Figure 3.4](#):

Figure 3.4: Derivatives and tangent

As h becomes very small, this secant line tends to become a tangent line at x_0 . The slope of this tangent line to the graph of the function at the point , $f()$ is the right derivative of the function f at and is formally written as:

Let's call this as is in the right side of . Similarly, we can define the left derivative as:

for any point is the left side of in the axis.

The *function f is differentiable* at if and only if f has both a right-hand derivative and a left-hand derivative at , and these derivatives are equal. This means the plot of the function is smooth in proximity of , that is, if you take a very small segment of the function in the proximity of , you can

approximate it by a small line segment. That is the reason why differentiable functions are also called smooth functions. Now, if the function is differentiable, we can rewrite derivative as a symmetric difference quotient:

Example: (Power function) The derivative of the power function can be calculated using the general derivative equation:

Using the definition of derivative, we can calculate the derivatives of most of the common mathematical functions like trigonometric, logarithmic, exponential. However, in practice, we will encounter many functions that are made up of sums, products, and composition of these functions, like $f(x) = \log(2 + \sin(x))$. To compute the derivative of such complicated functions, we use the following rules of derivatives:

1. Derivative is linear operator:
2. Product rule of derivative:
3. Chain rule of derivative (for function compositions):

Example: The sigmoid function is defined as:

Let $y = \frac{1}{1 + e^{-x}}$. Therefore, $\frac{dy}{dx}$. Applying chain rule:

Therefore,

Example: The \tanh function is defined as:

Applying product rule and chain rule of derivatives the derivative of \tanh can be found as follows:

These derivatives are computed for any arbitrary point in the domain of f , and we can visualize their plots in [Figure 3.5](#):

Figure 3.5: Derivatives as a function

Next, we state few useful theorems for differentiable functions.

Theorem: A differentiable function is continuous but not conversely.

Example: Let $f(x) = |x|$. At $x=0$, f is continuous at

and

As f is not differentiable at $x=0$. Differentiability implies smoothness. At $x=0$, there is a sharp corner in the plot. Change of direction by 90 degrees. Hence, geometrically, we can see the non-differentiability of f at $x=0$, so the continuity of a function does not always guarantee differentiability.

Mean value theorem (Lagrange): Let f be continuous on $[a, b]$ and differentiable at every point in (a, b) . Then, there exists at least one point c such that:

Rolle's theorem: Let f be continuous on $[a, b]$ and differentiable at every point in (a, b) . If $f(a) = f(b)$, then there exists at least one point c such that $f'(c) = 0$.

[Figure 3.6](#) depicts these two theorems pictorially:

Figure 3.6: Theorems on derivative:(Left) The slope of the secant line from $f(a)$ to $f(b)$ is given by $f'(c)$ and the tangent at c is parallel to this secant line. (Right) As $f(a)=f(b)$ the function must turn at some point and at that point, the derivative is 0.

Higher Order derivatives

The derivative of a function f is itself a function. We represent it by $f'(x)$. Let $f'(x) = g(x)$. Then, $(g(x))'$. So, we can compute the derivative of the derivative function:

This is called the *second order derivative*. Hence, for this function, we have . Derivative function can be computed any number of times to obtain n^{th} order derivative. This is depicted with the following notation:

Refer to the following figure:

Figure 3.7: Higher order derivatives

There are many applications of higher order derivatives like approximating functions and finding maximum or minimum values attained by a bounded function. We will briefly look at the Taylor series expansion of a function using higher order derivatives.

Taylor series expansion

If a real valued function satisfies the following conditions:

- The function should be differentiable any number of times
- The function should be defined at the given point a

Then,

This is called the *Taylor series expansion* of the function f at the point a . It's an infinite series, that is, the number of terms in this series is infinite and is represented more formally with summation notation, as follows:

Now, if we take x sufficiently close to the point a , then the higher degree terms of the polynomial for $n > 2$ become negligibly small and can be ignored.

In general, for any point x , we can approximate using a finite number of terms. Suppose is differentiable n times in the neighbourhood of a ; we can consider the Taylor polynomial with n terms only. There will be an error in this approximation proportional to the distance of x from a and is given by , where c is some point between a and x . This is called *Lagrange form of*

remainder. The second order Taylor's polynomial with Lagrange form of remainder is given by:

Example: and let . For x in neighborhood of 0:

Or

Here, if x is sufficiently close to $x=0$, even the Taylor series expansion truncated to 1 term approximates the sin function well. This is a linear approximation of sin function near $x=0$ with the Taylor polynomial $TS_1(x) = x$, as shown in [Figure 3.8](#). As we include more terms, we see that the Taylor series polynomial coincides more with $\sin(x)$, that is, we get an improved approximation. Refer to the following figure:

Figure 3.8: Function approximation locally (Taylor series)

Note: If we have a differentiable function f , calculating the first derivative of a function $f'(a)$ at a point $x = a$, we can approximate the function close to "a" by a line $y = f(a) + f'(a)$. This approximation is valid and very close to the point.

For example, if we take the function $|x|$. At any non-zero point, the function represents a straight line. However, at 0, where its not differentiable, we cannot approximate the function by a like however small interval around zero we take.

So far, we have covered a few basic concepts for studying the functions of a single real variable. In ML, we will mostly encounter function of several variables. These concepts of real analysis can be extended to the functions of several variables. In fact, this way of approximating a differentiable function locally by a line will be useful in defining the derivative of functions of several variables. In the following sections, we will study functions of several variables.

Scalar and vector fields

In the previous chapter, we studied linear transformations , from one linear vector space V to another vector space W . Let both of these vector spaces be finite dimensional. Let the dimension of the domain space V is $\dim(V) = n$ and dimension of range space W be $\dim(W) = m$. We call T a real valued function or a scalar field or a vector field based on n and m . Here, T need not be a linear function.

Scalar field represents functions that maps a point in n -dimensional space to a real number. For example, if at each point of the atmosphere, we assign a real number $f(a)$ representing the temperature at a , the function f is a scalar field. As we move from point ‘ a ’ to nearby point in space, the scalar field will vary. In ML, scalar fields arise while we train models. A model can be represented as a scalar field on the parameter space. It maps each possible parameter vector to total error value (a real number) w.r.t the given data points.

Let’s first see how points close to the vector a look like. In real line, points near number ‘ a ’ are points in the \in neighbourhood of a More formally, a set of all real numbers x such that . For vectors, this generalizes to the concept of open ball: Set of all vectors x in V such that : is called an *open ball around a* . The open ball around a represents points that are within a sphere of radius \in . A two-dimensional open ball is shown by the shaded region in [Figure 3.9](#):

Figure 3.9: Open ball

Limits and continuity

The concepts of limit and continuity can be easily extended to scalar and vector fields. Function with domain . Let and . Then, means the limit:

This is the usual limit for real valued function as is a real valued function. Similarly, function f is said to be continuous at a if f is defined at a and . f is said to be continuous on the set S if its continuous at every point in S .

Theorem: For vector valued function, the function is continuous if each component is continuous. For example, . Each component is continuous, and hence, f is continuous.

So, we see that the definitions are straightforward extensions of those in the real valued functions in one-dimensional case. However, extending concept of derivative at a point for scalar fields requires some more work.

Derivative of scalar fields w.r.t. vector

Generally, the manner in which a field changes depends on the direction in which we move away from \mathbf{a} . Let's take the temperature scalar field example. Starting from \mathbf{a} , the temperature increases moving towards the heat source, and it decreases as we move away from it. The rate of change or derivative of a scalar field is defined only in a fixed direction. Starting from the same point \mathbf{a} and going in a different direction, there may be a different rate of change.

Directional derivative and partial derivatives

Let's first choose a direction represented by a vector \mathbf{u} and then compute rate of change in that direction. Let \mathbf{u} be a unit vector. To compute rate of change in the direction of \mathbf{u} , let's choose an arbitrary point \mathbf{x} close to \mathbf{a} . Let \mathbf{u} be a unit vector, that is, $\|\mathbf{u}\| = 1$. For any real number h , the vector $\mathbf{a} + h\mathbf{u}$ represents all points on the line parallel to vector \mathbf{u} . If we choose h sufficiently small, then we can find a vector $\mathbf{y} = \mathbf{a} + h\mathbf{u}$ that is within the - ball around and lies on the line parallel to direction vector \mathbf{u} , as shown in [Figure 3.10](#):

Figure 3.10: Directional derivatives

Similar to the case of single variable, we can now write the derivative of a scalar field formally as:

This is called the **directional derivative** of scalar field f at \mathbf{a} in the direction of \mathbf{u} . In particular, if $\mathbf{u} = \mathbf{e}_k$ (the k^{th} unit coordinate vector $\mathbf{e}_k = (0, 0, \dots, 1, \dots, 0)$, with 1 at the k^{th} coordinate only), the directional derivative $f'(\mathbf{a}, \mathbf{e}_k)$

is called the ***partial derivative*** with respect to e_k and is denoted by symbols as follows:

Calculating partial derivatives is fairly simpler compared to arbitrary directional derivatives because only the rate of change along one of the coordinate axes is being measured. So, we can treat all other coordinate variables as fixed as we move parallel to only one axis. Hence, we can use all known formulas and rules for derivatives of single variable to compute partial derivatives.

Example: Let

Real valued function with one variable differentiability implies continuity at that point. For scalar fields, does the existence of all directional derivatives at a point imply continuity at that point? The following scalar field shows an exception:

If we approach $(0,0)$ along any line through origin other than x axis represented by a vector $\mathbf{u} = (a, b)$ where .

So, all the directional derivatives exist for f . Now, if we choose , we have . So, there are infinite points in the \in ball around $(0,0)$ where f takes the value $\frac{1}{2}$, but $f(0,0)$ is 0. Hence, f is not continuous at origin as close by points near origin are not mapped to close values. So, *even the existence of all the directional derivatives does not imply continuity of the scalar field at given point*. We need a better generalization of derivative for functions of several variables.

Total derivative

We have seen that for real valued function we can approximate f locally at a point by a line using first order Taylor series expansion. This shows a way of extending the concept of differentiability to the higher-dimensional case using linear function.

Definition (Differentiable Scalar Field): We say that f is differentiable at ' a ' if there exists a linear transformation such that , where for some $r > 0$ and ϵ is the error in the approximation and is of smaller order than . The linear transformation is called the total derivative of f at a . Here, ϵ is a real number. Note that the *derivative is a linear transformation* and not a number. We will see how to calculate this linear transformation . Suppose the total derivative of a scalar field f exists. Then:

For any point , If we choose , where h is chosen small enough such that . Then:

Dividing both sides by h , we have,

Taking the limit , we have:

Note: The linear transformation maps any to the directional derivative vector f at the point a along the direction of x . If we choose as the k th unit basis vector, that is, the one hot vector with k th entry as 1, then f is the partial derivative.

We can define as follows for all points such that x is close to a , that is,

where ' \cdot ' represents inner product.

is called **gradient of scalar field f** at ' a '.

Note: Thus, is the projection of x along the direction of the gradient vector at ' a '.

With this new definition of total derivative, we can now say that if a scalar field f is differentiable at ' a ', then its continuous at ' a '.

For any vector x close to a , that is, . If f is differentiable at a :

The last part of the inequality is using Cauchy-Swartz inequality for norms. Here, and hence, . Therefore:

Taking the limit on both sides, we have:

This proves that scalar field f is differentiable at ‘ a ’, then its continuous at ‘ a ’.

So, we can use the gradient of a scalar field to compute the total derivative of the scalar field. Here are some examples of gradient computations for some commonly used functions:

-
-
-

Geometry of gradient vector

Let f be a scalar field defined on a set . Consider the points where has a constant value say . We denote this by . The set is called *level set*. In , we call this a *level surface*. In ML, level surfaces occur very often. We will see surfaces of constant probability density and error surfaces with constant error contours.

Let’s first see how we can define curves in higher dimension as a parametric function. For example, a circle in 3D parallel to horizontal xy plane can be written as $(r\cos(\theta), r\sin(\theta), c)$, where c is the constant height of the circle above horizontal xy plane and r is the radius of the circle.

Let represent any curve on the level surface, where are real valued functions of the parameter. Then, . Using chain rule, (Dot product of perpendicular vectors is zero, refer to Linear Algebra Chapter). For the curve represents the tangent vector. Hence, the gradient vector at any point on the curve is perpendicular to the tangent vector at that point. Refer to the following figure:

Figure 3.11: Gradient vector normal to the surface

Derivative of vector fields w.r.t. vector

A vector field can be represented as where each component is a scalar field. Hence, we can generalize the concept of derivative to vector fields easily.

A vector field is differentiable at if there exists a linear transformation such that , where for some 0 and . The linear transformation is called the *total derivative* of f at \mathbf{a} . Since is a linear transformation from finite dimensional space of dimension n to finite dimensional space of dimension m , it can be represented by a matrix such that any point can be transformed to a point , by matrix multiplication with .

Now, for each component scalar field, we have the total derivative defined using gradient vector as ; thus, we can write:

This matrix is called the **Jacobian matrix** of f at \mathbf{a} . Hence, the total derivative of the vector field f is represented by the matrix product .

Example: Suppose and f is a vector field, where is a real valued and differentiable function. In f , the function g is applied component-wise. Then, , for all i , with i^{th} entry only non-zero being equal to and all other entries is zero.

Example: Let be a linear map defined by matrix. The i^{th} component of is given by , so . Hence:

Chain rule for derivatives of vector fields

Let f and g be vector fields such that the composition $h = f \circ g$ is defined in a neighbourhood of a point \mathbf{a} . Assume that g is differentiable at \mathbf{a} , with total derivative . Let and assume that f is differentiable at \mathbf{b} , with total derivative . Then, is differentiable at \mathbf{a} , and the total derivative $\mathbf{h}'(\mathbf{a})$ is given by the following:

that is, the composition of linear transformations .

Matrix form of the chain rule

We can rewrite the chain rule in terms of the Jacobian matrices. Since composition of linear transformations corresponds to multiplication of their matrices, representing

and we have .

Example: Suppose g is a linear transformation , where and W is a constant matrix and , where is sigmoid function. We define:

Hence, , whose i^{th} entry is only non-zero.

We want to find the derivative

$$g'(\mathbf{x}) = W \quad (\text{This is discussed in above example})$$

Example: Let's consider the linear map again ; now, is not a fixed matrix but is a constant vector. We want to compute . We can think of W as a vector of dimension , and thus, the Jacobian must be of dimension . The entire derivative is a three-dimensional array or a three-dimensional matrix. We call this a *tensor*. Tensors are generalization of matrices and are represented using n -dimensional arrays. Vectors and matrices are also tensors. A vector is a one-dimensional or first order tensor, a matrix is a two-dimensional or second order tensor, and a scalar is a zero-order tensor.

Let's compute the derivative of one component of f , that is, f_i w.r.t. one component of tensor W , say . For example, let's compute partial derivative of f_3 w.r.t W_{56} . Expanding f_3 , we have . This expression is independent of W_{56} , and hence, 0. However, for any component in the third column of W , . We can write in general:

Hence, the tensor is a sparse three-dimensional tensor.

While applying chain rule in practice, for example, when we apply it for differentiating neural networks, we can encounter such higher-order tensors. We will briefly digress from the current topic of matrix form of chain rule and introduce basics of tensor algebra in the next section. Thereafter, we will revisit chain rule and introduce the more generic form of it using tensors.

Tensors

A tensor can be represented as a multidimensional array. Tensors are extensions of vectors and matrices (which are one- or two-dimensional arrays) to n dimensional arrays. The individual elements in these n -dimensional arrays are called the *components* of the tensor. [*Figure 3.12*](#) depicts tensors of up to four dimensions. In the previous section, we saw how tensors naturally occur while computing derivatives of the functions of several variables.

Tensors are heavily used in physics as they provide a concise mathematical framework for formulating and solving physics problems. Tensors are represented using *index notation or indicial notation*. For example, a 3-dimensional vector a can be represented as follows:

Similarly, a matrix can be represented in index notation as follows:

The number of indices used to represent the tensor is called the *rank or order or dimension of the tensor*.

We can represent the scalar multiplication of a matrix in index notation as follows:

A matrix A and vector (column) v multiplication can be denoted in index notation as follows:

Here, the index j is repeated in the right-hand side of the expression and indices summation over the j^{th} index. This is also called *Einstein summation*

notation, which is discussed in the next section. Refer to the following figure:

Figure 3.12: Visual representation of tensors, a stack of 1-D tensors make a 2-D tensor. A stack of $n=4$, 2-D tensors make a 3-D tensor and a stack of $m=5$ 3-D tensors make a 4-D tensor and so on.

Einstein notation

Performing tensor operation for high-dimension tensors becomes very cumbersome and are hard to code. Einstein notation was introduced by Albert Einstein in 1916 in Physics for compact representation of summation over a set of indexed terms in a formula. All the tensor operations discussed earlier can be written in terms of Einstein sum. Moreover, using Einstein notation, many common multi-dimensional, linear algebraic array operations can be represented in a simple fashion. Code written using Einstein summation is highly readable and compact.

Einstein sum takes arguments in two parts: equation string and tensors on which the operation is performed. An example of equation string for matrix transpose operation is . Here, each of the small letters denote a dimension of the tensor. All indices on the left side of arrow are indices of input tensor, and those on the right side of arrow are indices of output tensor. For multiple input tensors, we can separate the indices by commas. For example, matrix multiplication is represented by the equation string . The indices that are missing on the right side of arrow are the axes over which the summations are performed, that is, the elements of j^{th} row of left matrix are multiplied to the corresponding elements of the k^{th} column of the right matrix and summed. The dot products and outer products discussed earlier can be computed easily using Einstein summation.

NumPy documentation (refer to *Further Reading [9]*) has many examples of Einstein summation. [Table 3.1](#) lists are a few useful notations:

Notation	Description
	a, b are 2 vectors or 1-D tensors of same size, and we want to compute elementwise multiplication, which is another vector C.
	A is a matrix, and we want to compute the trace of the matrix, that is, the sum of all diagonal elements.

	A and B are matrices, and we want to compute the matrix multiplication.
	We have a 3-D tensor A and want to compute the sum along the third axis.
	Outer product (of vectors), discussed in the following sections.
	Inner product of vectors, discussed in the following sections.
	Transpose of a matrix is a matrix with swapped rows/columns, that is, swapped axes or a reordering of axes. For tensors, any permutation of the axes list can be defined as a transpose of the tensor. Here, we compute transpose of a 4D tensor A by reordering axes 0,1,2,3 as: 1->3, 0->1, 3->2, 2->0.

Table 3.1: Einstein notation examples

The **outer product** \otimes of two one-dimensional tensors is a two-dimensional tensor represented by:

In general, outer product of two tensors of order m and n will yield a tensor of order $m+n$. In the preceding example, $m=1$ and $n=1$. So, the order of this outer product is $1+1=2$. This operation is non-commutative, that is, $\mathbf{u} \otimes \mathbf{v} \neq \mathbf{v} \otimes \mathbf{u}$.

Since \mathbf{u} is a second order tensor or a matrix, we can view it as a linear transformation \mathbf{T} , which transforms a vector \mathbf{w} to \mathbf{Tw} .

The length of the new vector is $m \times n$ times the length of \mathbf{u} , and the new vector has the same direction as \mathbf{u} .

Example: The 3×3 identity matrix is a second order tensor, and we can rewrite it in terms of outer product: $\mathbf{e}_1 \otimes \mathbf{e}_1 + \mathbf{e}_2 \otimes \mathbf{e}_2 + \mathbf{e}_3 \otimes \mathbf{e}_3$. Here, $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ represents the standard basis vectors or one-hot vectors. In Einstein notation $\mathbf{e}_1 \otimes \mathbf{e}_2$. The outer product $\mathbf{e}_1 \otimes \mathbf{e}_2$ is a second order tensor with only the $(1,1)$ element as 1 and all other elements as 0.

The following code shows this:

```
1. e2 = np.array([0.,1.,0.,0.])
2. e3 = np.array([0.,0.,1.])
3. np.tensordot(e2, e3, axes=0) #passing axes=0 computes outer
product
```

This will output a second order tensor with (2,3) element as 1. Therefore, the set of all such possible outer products forms a standard basis for any real matrix of size 4×3 .

Any vector can be represented as a linear combination of basis vectors. Can we do the same for tensors? A second order tensor \mathbf{T} of size 3 is a 3×3 matrix and has 9 elements represented as .

Representing this in index notation, we have: . We can similarly define a third order tensor , where each is a first order tensor. This tensor \mathbf{T} can transform a vector \mathbf{v} as follows to a second order tensor:

A tensor is called a **special tensor** if it can be represented as a product of finite first order tensors. Suppose \mathbf{A} and \mathbf{B} are two special tensors of order m and n formed by the tensor product of m vectors and n vectors , that is:

Applying \mathbf{A} on a vector \mathbf{v} , we have:

This is a tensor of order $m - 1$.

The dot product ($\mathbf{A} \cdot \mathbf{B}$) between these special tensors \mathbf{A} and \mathbf{B} can be computed as follows:

-
-
-
-

We see that only if , then for these special tensors.

Dot product of tensors

For computing the dot product (or inner product) of two vectors, we would have done . Can we extend the definition of **dot product** for tensors? Dot product of second order tensor with a first order tensor is the same as the

matrix multiplication of a matrix and a column vector. The order of the output will be a vector, that is, the order of the output is . Dot product of the first order tensor with a second order tensor is obtained by computing the matrix multiplication of the matrix transpose with the column vector, and we again obtain a first order tensor.

Multiplying two matrices of matching order will give another matrix, that is, tensor of order $2 = 2 + 2 - 2$. Hence, the tensor dot product of two second order tensors is another second order tensor. Hence, order of output of dot product of two tensors of order is a tensor of order $p + q - 2$. This notion of dot product of tensors can be generalized further to high order tensors: doing a dot product over the specified dimensions, keeping all the other dimensions fixed.

Suppose we have two tensors of order respectively. Suppose we want to compute the dot product along k -axes where . Then, we must specify which of the k axes of we want the dot product to be computed and the output tensor will have order: . This is implemented in *NumPy* and other deep learning frameworks like *TensorFlow* and *PyTorch* as the “*Tensordot*” operation. The axes on which the dot product is applied are called *contracted axes*. The shape of the output tensor consists of the non-contracted axes of the first tensor, followed by the non-contracted axes of the second. The following code shows how to compute tensor dot using *NumPy*:

```
1. import numpy as np
2. A = np.arange(24).reshape(4,2,3)
3. B = np.ones(12).reshape(2,3,2)
4. k=2 #num contraction axes
5. order_output = len(A.shape)+len(B.shape)-2*k
6. # Dot product of last k(=2) axes of A and first k axes of B
7. out1 = np.tensordot(A, B, axes=[[1, 2],[0, 1]])
8.
9. # what is happening inside tensordot
10. out2 = np.zeros(A.shape[:-k]+B.shape[k:])
11. for p in range(4):
12.     for s in range(2):
13.         for q in range(2):
14.             for r in range(3):
15.                 out2[p, s]+=A[p, q, r]*B[q, r, s]
```

```
16. np.equal(out1, out2)
```

Figure 3.13 shows the input output tensors. Lines 10-15 show how to calculate the dot product along k chosen axes using nested loops. This is very slow computation but helps us understand clearly what is actually happening inside the *Tensordot* operation. However, the actual implementation of Tensordot is not with nested loops but is optimized and gives us a much faster way to compute.

Figure 3.13: Output of preceding code

Now, let's check how good our generalization of the dot product is. We know the dot product of a vector a with itself is the norm squared . If we take a tensor of order 2 that is a matrix, is the tensordot equal to the Frobinius norm squared of the matrix? The following code checks for this, and it's true:

```
1. M = np.arange(12).reshape((3, 4))
2. out = np.tensordot(M, M, axes=([0,1], [0,1]))
3. fob_norm = np.array(np.linalg.norm(M)**2)
4. print(fob_norm==out)
```

Now that we have generalized all the properties and operations on vectors for tensors, we can also extend the concept of vector fields to tensors and calculus of vectors are generalised to the calculus of higher-order tensors.

Tensor calculus

The gradient of a scalar field can be written as follows:

(in index notation)

The gradient of a vector field , that is, the Jacobian is defined to be the second order tensor:

Like vector fields, a tensor field assigns a tensor to each point of the space. The gradient of a second order tensor field T is defined in a manner analogous to that of the gradient of a vector.

Properties of gradients are as follows:

-
-
- , where φ is a scalar field

Total derivative of tensor

For a tensor valued function of a scalar , we can define the derivative as follows:

This turns out to be a tensor whose components are the derivatives :

Let f be a scalar valued function of a second order tensor T . We can write in index notation. Hence, we have:

Let's consider more general form, that is, a tensor valued function of a tensor. For example, the derivative of a second order tensor A with respect to another second order tensor B is given by the fourth order tensor:

Using this, we can compute in the example above where f where w are first order tensors and W is a second order tensor.

Why ? This can be easily seen by expanding the matrix multiplication with vector x . Only will have non-zero derivative w.r.t the jth row elements of matrix, that is, as . This explains why is chosen and .

We can validate this with the implementation of gradient in any of the libraries, like TensorFlow. We will create a random tensor W and a vector x , as shown in the following code XX. The **GradientTape** function is used to compute the Jacobian.

```
1. import tensorflow as tf  
2. W = tf.random.uniform(shape=[4, 4])
```

```

3. x = tf.expand_dims(tf.Variable([1., 2., 3., 4]), axis =1)
4.
5. with tf.GradientTape() as tape:
6.     tape.watch(w)
7.     y = tf.matmul(w,x)
8. dy_dw = tape.jacobian(y, w)

```

Now, let's implement the gradient that we found in the preceding theory and check whether they match.

```

1. e1,e2,e3,e4 = np.eye(4)
2. x=np.array([1.,2.,3.,4.])
3. grad = None
4. for k, ek in enumerate([e1,e2,e3,e4]):
5.     for j, ej in enumerate([e1,e2,e3,e4]):
6.         tmp = x[k]*np.tensordot(ej, ej, axes=0)
7.         if grad is not None:
8.             grad+= np.tensordot(tmp, ek, axes=0)
9.         else:
10.            grad = np.tensordot(tmp, ek, axes=0)

```

We can see that the output in both the implementations is the same third order tensor, as shown in [Figure 3.14](#):

Figure 3.14: Output of the preceding code

Note: We observe that the gradient of a scalar or 0-order tensor is a first order tensor, and the gradient of a vector field or a first order tensor is a second order tensor. In general, the gradient operator always adds the orders of the two tensors.

Example: The trace operator on a matrix is a scalar field:

(in index notation)

Following similar steps, we can derive the following matrix derivatives:

Chain rule for tensors: Suppose g is a tensor valued function and f is a scalar valued function of a tensor . Then, .

Example: Suppose is a linear transformation , where and W is a second order tensor and , being the sigmoid function. Let's define . We want to compute . In neural network training, we will encounter similar operations.

, by chain rule

Here, is a diagonal matrix or a second order tensor :

Hence, we have:

Here, we can use the property of outer product and simplify the product as follows:

This is a third order tensor, being gradient of first order w.r.t. second, its order is $(1 + 2 = 3)$. This is implemented in the following code:

```
1. W = tf.random.uniform(shape=[4, 4])
2. y = tf.matmul(W, x)
3. y1=tf.sigmoid(y)*(1-tf.sigmoid(y)).numpy()
4. grad = None
5. for j, ej in enumerate([e1, e2, e3, e4]):
6.     for k, ek in enumerate([e1, e2, e3, e4]):
7.         tmp = np.tensordot(ej, ek, axes=0)
8.         if grad is not None:
9.             grad+= np.tensordot(x[k]*y1[j]*ej, tmp, axes=0)
10.        else:
11.            grad = np.tensordot(x[k]*y1[j]*ej, tmp, axes=0)
12. grad
13.
```

The shape of grad is $4 \times 4 \times 4$. Again, using the Tensorflow **GradientTape** function to compute the Jacobian, we have:

1. with `tf.GradientTape()` as tape:

```

2.    tape.watch(w)
3.    y = tf.sigmoid(tf.matmul(w, x))
4. dy_dw = tape.jacobian(y, w)

```

Following is the output of the preceding code:

Figure 3.15

The shape of the Jacobian of the output with respect to the weight matrix is those two shapes concatenated together.

Mathematical optimization

Finding the minimum or maximum value taken by a real valued function is called *function optimization*. For example, the function $f(x) = |x|$ attains its minimum value at $x = 0$. The temperature on a surface can be represented as a scalar field and point at which the temperature is maximum of the surface is the location of the heat source. Problems in probability theory and ML can be represented as functions of several variables. In ML, we are trying to find an approximating function that maps input examples to output examples. The problem of finding a good approximating function can be framed as a function optimization. These are parametrized functions, and methods of function optimization are used to find the best possible values of these parameter by minimizing the error of approximation.

Maxima, minima, and saddle point

Definition (Global and Local minimum): A scalar field is said to have a *global* (or *absolute*) *minimum* at a point c of a set if:

for all

The function f is said to have a *local* (or *relative*) *minimum* at c if the preceding inequality holds only in a ϵ -ball around c and not for all . We can define global and local maximum similarly. A vector that is either a relative maximum or a relative minimum of f is called an *optimum* or *extremum* of f .

If f has an extremum at an interior point and is differentiable there, then the gradient of f at that point must be zero, that is, . However, the converse is

not true. If f is differentiable at a point \mathbf{a} , it's called a *stationary point* of f . There may be points where the gradient is zero, but it need not be an extremum point, and such stationary points are called *saddle points*. In any close neighbourhood of a saddle point \mathbf{a} , we will find points such that and .

Example: This represents a surface (a hyperbolic paraboloid). Near the origin, this surface looks like a horse saddle, as shown in [Figure 3.16](#). The gradient vector at zero is at origin. However, in close neighbourhood of the origin, we can find points from 1st and 3rd quadrant where f is positive. Also, close to the origin, there are points from the 2nd and 4th quadrant where x, y is of opposite signs, f is negative. So, the origin is a saddle point of this function. Refer to the following figure:

[Figure 3.16: Saddle point, \(This figure is adopted from T.M Apostol \[1\] \[chapter 9\]\(#\)\)](#)

Example: , the origin is a saddle point as depicted in [Figure 3.17](#).

We can use second order Taylor's series expansion to figure out the nature of the stationary point, that is, whether it's a maxima or minima or saddle point. The eigen values of the Hessian matrix can give us clear idea about the nature of stationary point. This is stated in the following theorem. Refer to the following figure as well:

[Figure 3.17: Saddle point, \(This figure is adopted from T.M Apostol \[1\] \[chapter 9\]\(#\)\)](#)

Theorem: Let f be a scalar field with continuous second order partial derivatives in an n-ball . Let denote the hessian matrix at a stationary point \mathbf{a} . Then, we have the following:

- If all eigen values of are positive, then f has a relative minimum at \mathbf{a}
- If all eigen values of are negative, then has a relative maximum at \mathbf{a}
- If has both positive and negative eigen values, then \mathbf{a} is a saddle point of

Example: . Here, at origin $\mathbf{a} = (0, 0)$. So, origin \mathbf{a} is a stationary point of f . This is a positive definite, and hence, is a relative minimum.

Let f be a scalar field with continuous second order partial derivatives in an n -ball ; then, using the Taylors formula for real functions, we can derive the second order Taylor's formula for scalar fields. We will give a high-level flow of the proof of this here.

Let f for . Then, . Using second order Taylors formula with Lagrange form of remainder for the real function g , we have:

Here, g is a function of the function, where r . Applying the chain rule of derivatives, we have:

In particular,

Applying chain rule once more on this, we get:

Substituting these in the Taylor expansion, we have:

We define an error term by the following equation:

and use it to reformulate the preceding equation in terms of this :

Here, . This is the Taylor expansion for scalar fields.

At any stationary point \mathbf{a} , we have . Hence:

Since the Hessian is a real symmetric matrix, the quadratic form is positive definite if and only if all its eigen values are positive and negative definite if all eigen values are negative.

Suppose all eigen values of are positive and . We choose then, . Clearly, , are the eigen values of . The quadratic form is as follows:

Now, , using the definition of limit for any chosen positive there exists a such that . Multiplying both sides of this inequality by , we get . Choosing the arbitrary number , we have .

Hence, a is a relative minimum. Similarly, we can prove the statements of the theorem for maximum and saddle point.

Example: Locate and classify the stationary points for the surface

We have and . At a stationary point, both the partial derivatives should vanish. Clearly, origin is a stationary point.

Now, and . This contradicts . Thus $(0, 0)$ is the only stationary point.

Now, let's discuss an iterative algorithm that can be used to minimize any differentiable function.

Descent methods

In practice, the functions of several variables are encountered, we will have no idea where in the domain of the function it will attain its optimum value. In fact, we may never be able to find the exact minimum . Our target is to reach as close as possible to the minimum, . So, we must start at some arbitrary point and then take small steps in the direction in which the function value decreases:

Here, is the **step size** in the direction of the search direction . Our target is to choose and such that . These sequence of points (should converge to the minimum).

Using first order Taylor series expansion, let and be a point very close to a , that is, . Let :

$$, \text{ where}$$

Here, represents the directional derivative in the direction of the change vector . If this term is negative, we can reduce the value of f at . So, v must

make an acute angle with the negative gradient . We call such a direction as *descent direction*. The best case is when the angle between the vectors and is zero or the vectors are parallel, that is, if for some . This is called the *direction of steepest descent*. The direction of the negative gradient at a point is the direction of steepest descent.

Hence, we can define an iterative algorithm for minimizing a function as follows:

- Given a starting point
- Repeat:
 - Choose a step size
 - Update

Stopping criteria for the preceding iterative algorithm is generally based on norm of the gradient. As we are finding a stationary point iteratively, the algorithm can converge when the gradient at the point is close to 0, that is, where and is small. is a pre-defined fixed number indicating acceptable level of error. This algorithm is called *gradient decent algorithm*. Now, we also need a way of choosing the step size. The step size should not be too short or too long. The following example shows the effect of step size.

Example: Let's take a real valued function . Being a function on real line, we have only two possible directions ; we can choose the descent direction as . Note that this decent direction is also parallel to the negative gradient direction, as . We will use two different step sizes (1) which is a big step size, and (2) which is a monotonically decreasing small step size. Now, starting with initial point and following the preceding update equation , we see that with (1), the function value eventually oscillates between -1 and 1 and with step size (2), the decrease in function value becomes very small and almost converges at 1. The following code shows an implementation of this:

```

1. def step(k, type=1):
2.     if type ==1:
3.         return 2+3/np.math.pow(2, k+1)
4.     elif type==2:
5.         return 1/np.math.pow(2, k+1)

```

```

6.     else: return 1
7.
8. def update(x, k, type=1):
9.     return x+np.sign(-x)*step(k, type)
10.
11. x = 2
12. for k in range(20):
13.     x = update(x, k, type=1)
14.     print(k, x, x**2)
15.     k=k+1

```

[Figure 3.18](#) shows the output with step type (1) and (2):

Figure 3.18: Gradient descent (Left) very long step (Right) very small steps

One approach to find appropriate step size is *exact line search*, where the function is minimized along the line , that is:

This is a valid method but is not cost effective. So, inexact line search methods are used where we are looking for a step size that reduces f enough. One such method is called *backtracking line search*. This depends on two parameters . Given a decent direction , we start with unit step size and then reduce the step size by a factor of b until some stopping criteria is reached. The first order Taylor's formula can give us a stopping criterion. By first order Taylors approximation,

So, in the successive step, we reduce the step size by a constant factor (this restricts the steps size from being too small), and we see that f is reduced by at least a fixed fraction of the reduction promised by first order Taylor's formula. This also restricts the step size from being too long. This condition is called **Armijo condition**.

There are other heuristics that are also used for step size selection in gradient descent.

Constant step size: Choose one fixed value .

Variable step size: 3 or 4 values of step are chosen in each iteration, and whichever gives the best reduction of function value is chosen.

- **Golden search:** A range between two values is used and divided into sections.
- **Example: (Rosenbrock function):** . This function has a global minimum at the point where . The global minimum is located inside a long, narrow, parabolic shaped flat valley, as shown in [Figure 3.19](#) for 10. The gradient of the function is given by:

Refer to the following figure:

Figure 3.19: Rosenbrock function with gradient descent

Following are some advantages and disadvantages of steepest descent:

- Converges to a local minimum from any starting point
- Convergence can be very slow sometimes

Intuitively, it may seem that the method of steepest descent is the best direction for minimizing a function. But this is not true! A more general search direction is defined as a solution to a system of linear equations , where B is a positive definite matrix. For the solution to be a valid search direction, it must satisfy . This holds true because of the positive definiteness of . A particular case for this is choosing B to be the Hessian at . If the Hessian is positive definite at . This is called **Newton's method**. The main disadvantage of Newton's method is the cost associated with finding the inverse of the Hessian and ensuring that the Hessian inverse matrix is positive definite. [Figure 3.20](#) shows the convergence with Newton's method:

Figure 3.20: Rosenbrock function with Newton's method

Function optimization with constraints: Lagrange multipliers

A constraint is a limit placed on the values of a variable, that is, the solution to the optimization problem is restricted to a subset of the domain of the function. For example, we may have a problem in economics where we want to maximize the utility function subject to the constraint . If the form of the constraint is complicated, then solving constrained optimization is very hard. However, for some simpler forms of constraints, for example, the constraint represents a curve or a surface, then they can be solved by the method of Lagrange multipliers.

Suppose a scalar field has a relative extremum when its subject to the constraints , where , then there exists scalars such that:

These scalars are called *Lagrange multipliers*; one multiplier is there for each constraint. We assume that the scalar field f and the constraints are all differentiable functions. Let's understand this geometrically.

Let represent a scalar field temperature function in three-dimensional space. We want to find the maximum value of the temperature along a curve C. We can represent C as the intersection of two surfaces and . So, we must solve the following constrained optimization problem:

subject to the constraints and

The gradient vectors and must be normal to the respective surfaces, as shown in the [Figure 3.21](#). At the extremum point, the gradient vector is normal to the curve C. Suppose C is represented by the vector values function:

and let represent the temperature along the curve. The maximum value of the temperature is attained at or . Hence, must be perpendicular to the tangent to the curve, that is, its normal to the curve.

Now, since all the three vectors , , are normal to the curve at the extremum point, they all must lie on same plane. So, if , are **linearly independent**, we can write . Refer to the following figure:

Figure 3.21: Constraint gradient and gradient of objective

With the use of Lagrange multipliers, we can convert a constrained optimization problem to an unconstrained problem. This unconstrained problem has m more variables to optimize: the multipliers . We can minimize the function:

This technique will be widely used in many constrained optimization problems that we will encounter in ML. One of them is regularization of ML models, which we will discuss in the next chapter.

Optimization with inequality constraints

We can write the optimization problem in general form as follows:

Here, we have only equality constraints, and for this, we can apply Lagrange multiplier trick. But if we have inequality constraints like , we can convert this inequality constraints to equality constraints by introduction of extra variable called *slack variables*.

Suppose denotes n slack variables corresponding to each inequality constraint. These are positive quantities such that . So now, we convert the inequality constraints to equality constraints and again use Lagrange multipliers to convert the problem to unconstrained optimisation problem. Additionally, slack variable are positive.

where .

The Lagrange dual function

The **Lagrange dual function L** is defined as the minimum value of the Lagrangian for any given value of the multipliers .

Suppose is a feasible solution of the problem, that is, and ; then:

Hence, for the optimal feasible solution :

By definition of infimum:

Also, by definition of supremum, we have the following:

Therefore, we have a new optimization problem:

This is called the *dual problem* associated with the original constrained optimization problem, which we call *primal problem*. We call the *Lagrange dual function*.

The optimum value of both the primal and dual problems must always satisfy the following condition:

This is called *weak duality*. Suppose the equality above holds. Then, its termed as *strong duality*. This happens when the objective function and constraints are of certain form that is, they are *convex functions*. In the following section, we will discuss convex functions and optimization of convex functions. Many optimization problems encountered in ML can be formulated as convex optimization problems.

Convex functions

We have already discussed convex sets in the previous chapter. A scalar field is called a convex function if the domain of f is a convex set and for any two points and we have the following:

We define f as *concave* if $-f$ is *convex*.

Examples:

- for any real a is convex, is concave.
- Every norm in \mathbb{R}^n is a convex function because of the triangle inequality obeyed by any valid norm:
- The max function in \mathbb{R}^n is convex.
- Geometric mean function,
- Negative entropy function in \mathbb{R}^n is convex.
- The function x^{-1} is concave.

Properties of convex functions

Following are a few useful properties of convex functions:

First order conditions: Suppose f is differentiable scalar field; then, f is convex if and only if the first order Taylor approximation of f underestimates f . that is, for any two point :

This is a very important property of convex functions and is easy to prove as well. Suppose f is convex and let ; then, for any two points in domain of f , we have the following by convexity property:

Taking the limit as , the ratio in the right side above shows the directional derivative of f in the direction of , and hence, can be written as . This proves the first order condition. We can also derive the converse, that is, convexity of f from the first order condition. Interested readers may refer to [4] for the proof.

Note: If , then we have for convex function. Hence, x is a global minimum of the function. So, any stationary point is a global minimum for convex functions. This property makes the convex optimization problem a special class.

Second order conditions: Suppose f is twice differentiable, and its Hessian or the second derivative exists. Then, f is convex if and only if its Hessian is positive semidefinite, that is, , for all x in the domain of f . To view this

geometrically, we can think f represents a surface; then, the surface must have positive curvature given its convex.

Tip: *The first and second order conditions hold good for concave functions as well with the following modifications:*

- Any stationary point is a global maximum
- f is concave if and only if, for all x in domain of f

Example: The quadratic function . We have and . Hence, by second order condition, f is convex if .

Example: The least square objective function . We have is convex for any A since is a real symmetric matrix, and hence, is positive definite.

Restriction of a convex function to a line: A function is convex if and only if the function , where is convex in t , where is in domain of f and t is a real number. **Example:** The function .

Here are the eigen values of

Jensen's inequality: If f is convex, then for ,

Convex optimization

A convex optimization problem is of the following form:

Here, the objective function f and the inequality constraints are convex and the equality constraints are linear or affine:

A fundamental property of convex optimization problems is ***any locally optimal solution is also (globally) optimal***. The gradient descent algorithm discussed above can be proved to converge to the global minima for unconstraint convex optimization problem, given strong convexity assumption, that is, there exists positive constants m, M such that .

Note: The condition number of the hessian is bounded by m/M , and the number of iterations N required for gradient descent algorithm to converge is bounded above by m/M . We have the relation .

For constraint convex optimization, we can use the Lagrange multiplier trick to convert to unconstraint optimization problem. There, the strong duality holds under certain basic assumptions on the inequality constraints called Slater's condition.

Slater's condition: There exists an x in domain of f such that the strict inequality holds true: and . Such a point is also called a *strict feasible solution*.

Note: The Slater's theorems states that if slater conditions hold, then the convex optimization will have strong duality, that is, the maximum value of Lagrange dual will be equal to the minimum value of the objective: . This property can give us a stopping criterion for any iterative optimization algorithm. If we iteratively reach a point such that , then we can guarantee that we are close to the optimal solution, that is, .

The strong duality also gives us certain necessary and sufficient conditions that can help solve the optimization problem analytically. These are discussed in the next section.

Karush-Kuhn-Tucker conditions (KKT)

Suppose strong duality holds for an optimization problem. Let the problem be convex optimization where Slater's condition holds:

Then, and are the primal and dual solutions if and only if and satisfy following conditions:

- , (primal feasibility condition)
- , (primal feasibility condition)
- , (dual feasibility condition)
- (complementary slackness condition)

- The point minimizes , so we must have the following *stationarity condition*:

These conditions are called **Karush-Kuhn-Tucker (KKT)** conditions.

Note: For any optimization problem (convex or non-convex) with differentiable objective function and constraint functions, such that strong duality holds, any pair of primal and dual optimal points must satisfy the KKT conditions.

These KKT conditions play an important role for ML models like **Support Vector Machines (SVM)**. SVM formulates the ML classification problem as a convex optimization and uses Lagrange duality and KKT conditions to solve the optimization problem. We will see applications of Lagrange multipliers in solving many optimization problems in the probability theory chapter as well.

Conclusion

In this chapter, we covered the differential calculus for functions of vectors and tensors. We used them to introduce the optimization theory of functions of vectors and tensors. We will be using these concepts throughout the rest of the book, in almost all the algorithms we discuss. In the next chapter, we will discuss another important pillar on which the theory of AI relies – the probability theory that quantifies the uncertainty naturally arising in AI problems and provides us mathematical tools to deal with the uncertainty.

Points to remember

- Any differentiable function of single variable can be approximated locally by a line using first order Taylor series expansion.
- For functions several variables, we must compute directional derivatives because the function values changes based on the direction in which we move away from a point. The directional derivatives along the axes are called *partial derivatives*.

- Existence of all the directional derivatives does not imply continuity of the scalar field at given point.
- A function is differentiable at a point if total derivative exists at that point, which means that the function can be approximated locally by a linear transformation at that point.

Further readings

- Tom IN. Apostol. CALCULUS. VOLUME II. Multi Variable Calculus
- Convex Optimization / Stephen Boyd & Lieven Vandenberghe
- KKT Conditions: <https://www.cs.cmu.edu/~ggordon/10725-F12/slides/16-kkt.pdf>
- Convexity Properties: https://www.princeton.edu/~aaa/Public/Teaching/ORF523/S16/ORF523_S16_Lec7_gh.pdf
- Gradient Descent: https://people.maths.ox.ac.uk/hauser/hauser_lecture2.pdf
- Tensor Calculus: <https://cedar.buffalo.edu/~srihari/CSE676/6.5.2%20Chain%20Rule.pdf>
- <http://homepages.engineering.auckland.ac.nz/~pkel015/SolidMechanicsBooks/Part III/Chapter 1 Vectors Tensors/Vectors Tensors 15 Tensor Calculus 2.pdf>
- <http://cs231n.stanford.edu/vecDerivs.pdf>
- <https://numpy.org/doc/stable/reference/generated/numpy.einsum.html>
- <https://www.tensors.net/p-tutorial-1>

CHAPTER 4

Basic Statistics and Probability Theory

The probability theory provides us the mathematical tools to quantify the uncertainty of decision-making where we have incomplete information. In machine learning, uncertainty naturally arises from noisy data or incomplete information about the problem domain. With such given uncertainties, we must come up with a predictive model that can estimate the outcome of an unseen event. There are a range of ML models that are completely built on probabilistic settings, like Bayesian nets, generative models, linear and logistic regression. Most objective functions used in training deep neural net models are derived using a probabilistic framework. However, even before we attempt to quantize the uncertainty in information, we must collect and organize information associated with a given problem. Thereafter, we can draw conclusions and make inferences or predictions based on these data. The science that deals with the collection, organization, analysis, interpretation, and inference of numerical facts or data by applying mathematical theories of probability is known as statistics. Statistics has a wide range of applications in almost every field of study, like medicine, economics, sociology, psychology and astronomy. However, the laws of statistics are true on aggregate of facts and cannot be applied for single observation, that is, statistics does not study individuals. Unlike the laws of physics, the laws of statistics are not exact but are approximate.

Structure

In this chapter, we will cover the following topics:

- Basic statistics
- Probability theory
- Introduction to Bayesian decision theory
- Random variable and probability density function
- Expectation, correlation, and covariance

- Information theory

Objectives

Probability theory is one of the foundational pillars of ML and AI, which are one of the applications of inferential statistics. This is the preparatory chapter for understanding inferential statistics and probabilistic interpretation of ML and AI problems covered in the next chapter.

Basic statistics

The science of collecting, organizing, analyzing, and inferencing from data for the purpose of effective decision-making is called statistics. Statistics has following two major branches:

- **Descriptive statistics:** Collecting and organizing data
- **Inferential statistics:** Drawing conclusions from data

Here, *data* can be defined as a collection of facts or information from which conclusions can be drawn. Data can be of various types:

- **Qualitative (Categorical):**
 - **Nominal:** Unordered categorical data like color, gender, location, ethnicity, marital status, weather.
 - **Ordinal:** Ordered categorical data for example, level of education, economic status, designation in a corporate ladder, knowledge level in a technology.
- **Quantitative (Numerical):**
 - **Discrete:** Integral values that can be finite or countably infinite, like counts of vehicles at a traffic crossing
 - **Continuous:** Any real value within a defined range, which can be infinite as well; for example, room temperature, blood pressure, height of a person, and stock price
- **Unstructured:** Text, image, audio, video

Quantitative data can exhibit certain general characteristics. The very first step to study such quantitative data is to compute the *frequency distribution*

that graphically represents the number of observations taking a particular value or number of observations in a certain interval. We may observe some larger frequencies for certain values or for certain range of values, that is, a tendency to concentrate at certain values. This is known as *central tendency*. There are various ways to compute this central value, which are collectively known as *measures of central tendency*. The data about the measure of central tendency can vary, and this measure of deviation is called a *measure of dispersion*. The data may show a symmetrical distribution about the central value or show asymmetry. The metrics to measure these degrees of symmetry is called the *measure of skewness*. Also, data may sometimes show a peak at certain central values, and the degree of sharpness of the peak can be measured by *measures of kurtosis*.

Qualitative data can also show central tendency, which is nothing but the most popular category. There are measures of dispersion for qualitative data as well. Unstructured data needs to be converted to structured data, which can be numeric or categorical, before being analyzed further. Text can be viewed as a bag of words where each word is a category. Digital image is a matrix of numeric pixel values.

Measures of central tendency

There are three types of central tendency measures, which are discussed in detail below.

Mean

For a set of n observations,

- **Arithmetic Mean** of S denoted by \bar{x} and defined as $\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$. We call the quantity $x_i - \bar{x}$ as the *deviation* of the point x_i from the point \bar{x} . Two simple but useful properties of deviation are as follows:
 - Sum of all deviations about mean is zero or $\sum_{i=1}^n (x_i - \bar{x}) = 0$.
 - The sum square of deviations is minimum when deviations are computed about the arithmetic mean.

Proof of 1 is trivial, and we can use differential calculus for 2.

- **Geometric Mean** of S is denoted by G and is defined as:

For G to be well defined, we must have all $x_i \neq 0$. G gives comparatively more importance to smaller values and is robust to small fluctuations or noise in data.

- **Harmonic Mean** of S is denoted by H and is defined as

For H to be well defined. Harmonic mean gives greater importance to small numbers. Like geometric mean G , H is also not affected by small fluctuations of data.

Among these three means, the arithmetic mean is greater than the geometric mean, and the geometric mean is greater than the harmonic mean.

Median

The **median m** of a dataset is a value that divides the data set in two equal parts such that number of points more than m is the same as the number of points less than m . So, if we have an odd number of points, we can sort the points and take the middle value m from ordered points with k points on either side of m . If n is even, then there are two middle points, and we can take the arithmetic mean of the two. Median is very robust to outliers or extreme values, which may be introduced due to experimental error or mistakes while noting observations or some rare anomaly. Arithmetic mean of data will move toward such extreme values, but median will be robust and does not change in the presence of a few outliers.

Mode

The value that occurs most frequently in a set of observations is called the mode. In case of discrete data, we can compute the frequency or number of times each discrete value occurs in the data and value with maximum frequency is the mode. Mode can be more than one as there may be more than one value with maximum frequency. Mode of continuous data can be found by bucketing the continuous data into equi-spaced interval and computing the frequency of each interval. The interval with maximum frequency contains the mode, and we call this the modal interval. Now, we

also need to find one value in the maximum frequency interval to represent the mode. This can be taken to be the midpoint of the interval. A more precise way to compute mode is given by the following formula:

Where f_m is the frequency of the interval containing mode and f_{m-1} and f_{m+1} are the frequencies of the intervals preceding the modal class and following the model class, respectively. l is the lower bound of the modal interval, and h is the interval width.

Let's take the example of Iris dataset that we already discussed in [Chapter 2, Linear Algebra](#), to discuss the various central tendency measures. This dataset consists of five attributes of flowers: four numerical features about the flowers and one categorical attribute that depicts the type of flower. [Figure 4.1](#) shows the histogram plot of the four continuous attributes. We have 50 samples from each flower category. The mean, median, and mode are marked with vertical lines and indicated in legends. For sepal length and sepal width, we observe a predominant frequency for one interval, and the mode can be defined to be the midpoint of that interval. For the petal length and width, we clearly see two peaks in the histogram plot, and this indicates a multimodal distribution. These attributes have two modes. Unlike mean and median, mode is ill-defined. There can more than one mode. The following code shows how to load Iris dataset and compute the central tendencies:

```
1. from sklearn import datasets
2. import numpy as np
3. import pandas as pd
4. iris = datasets.load_iris()
5. #compute mean, median using numpy
6. print(np.mean(iris.data[:, 0]), np.median(iris.data[:, 0]))
7. #loading into a pandas dataframe
8. df = pd.DataFrame(iris.data)
9. df.columns = iris.feature_names
10. #computing mean, median with pandas
11. print(df['sepal length (cm)'].median(), df['sepal length (cm)'].mean())
```

In the code, **pd.mode()** will give you the modes of each of the attributes, but this implementation only computes the attribute value with highest frequency. For *sepal length*, it outputs the mode as 5.0 because in the data, this value occurs highest number of times (10 times). The value 6.3 occurs 9 times, but it's evident from the histogram of **sepal length** that most of the values occur between 5.5 and 6. Thus, it's better to avoid computing the mode of continuous data only based on frequency of occurrence and use the formula mentioned above to compute the true mode or at least plot a histogram and take the midpoint of the high frequency bins as the modes. Refer to the following figure:

Figure 4.1: histogram of feature values for iris data

Slicing the data based on the modal intervals gives us some interesting insights, as depicted in the following code:

```
1. df['flower']=np.apply_along_axis(lambda x:  
iris.target_names[x], 0, iris.target)  
2. #upper bound of first modal interval is 2  
3. df[df['petal length (cm)'] < 2].flower.value_counts()  
4. #setosa      50  
5.  
6. #lower bound to second modal interval is 4.5  
7. df[df['petal length (cm)'] > 4.5].flower.value_counts()  
8. #virginica    49  
9. #versicolor   14  
10.  
11. # restricting to modal interval for petal width  
12. df[ (df['petal width (cm)'] >=1.0) & (df['petal width  
(cm)'] <=1.5)].flower.value_counts()  
13. #versicolor   45  
14. #virginica    3
```

Line 3 and 7, 8, 12, and 13 show the output. This indicates that the attribute petal length alone can be used to distinguish two classes of flowers *setosa* and *virginica* easily. The second modal interval of the attribute petal width can be used to identify the *versicolor* class of flowers from all other

flowers. Using these insights from the mode of the attributes, we can design a simple rule-based classification model, as follows:

- petal length $< 2.0 \Rightarrow$ setosa and petal length $> 4.5 \Rightarrow$ verginica
- $1.0 \leq$ petal width $\leq 1.5 \Rightarrow$ versicolor

Partition Values

In the preceding example, the mode was used to partition the data into subsets, and we saw how we can use these partitions. The median also partitions the data in two equal parts such that number of observations greater than median equals the number of observations less than or equal to median. We can generalize these to any number of partitions. The partition values or points that divide the dataset into four equal parts are called *quartiles*, and they are denoted by Q1, Q2, and Q3. Each partition contains $\frac{1}{4}$ of the total number of points in the dataset. These values can be obtained by sorting all the observations in ascending order and then finding values such that Q1 exceeds 25% and is less than 75% of the observations. Q2 exceeds 50% of observations, that is, it coincides with the median. Q3 is a point that has 75% of observation less than itself and 25% more than itself. The quartiles are pictorially shown in a *box plot* in [Figure 4.2](#):

*Figure 4.2: Histogram of feature values for iris data (min, max shown are **not** outliers)*

The following is the code to compute the quartiles using pandas and to plot boxplot for any set of observations:

```
1. df.boxplot(figsize=(10, 5))
```

The output plot is shown in [Figure 4.3](#):

Figure 4.3: Output of the code given above; the circles shown in sepal width plot are outliers

Extending this concept of quartiles to more partitions, we have deciles and percentiles. Nine points divide the dataset into 10 parts called *deciles*; similarly, 99 points divide the set into 100 parts called *percentiles*. Percentiles are used for reporting examination scores.

All these central tendencies, viz mean, median and mode, are suitable for different types of data and frequency distribution of data. Mode is the best central tendency measure if the data is qualitative. For continuous numerical data, median is used if the frequency distribution of data is skewed. Any of the three measures can be used for symmetrical data distributions as they will mostly coincide or be very close.

Measures of dispersion

The measures of scatteredness or spread of data is called **dispersion**. Suppose the central tendency of two sets of data is the same. For example, time to commute to school from home by public transport and by private school van is measured for 2 months. Both the modes of transport show the same mean, median and mode of 30 minutes. The distribution of data is symmetric, so all the three measures coincide. Then, we cannot decide which mode of transport is better using central tendency alone. This is where the measures of dispersion can help. In the following sections, we will discuss various dispersion measures.

Range

The range is the difference between the extreme values of the observations. The box plot in [Figure 4.2](#) marks the min and max values as well. **Range** = Max-Min. This is a crude measure of dispersion and is very sensitive to anomalies or outliers in data. This is simple to compute, but it does not consider all the data observations.

Interquartile Range

The **Interquartile Range (IQR)** is given by $(Q_3 - Q_1)$. This is a better measure than range as it uses 50% of the data and excludes anomalies. This measure is used to get rid of the outliers in data by a simple rule of thumb due to John Tukey. *Tukey's rule* says that the outliers are values more than 1.5 times the interquartile range from the quartiles, that is, the values that are either below $Q_1 - 1.5 \text{IQR}$ or above $Q_3 + 1.5 \text{IQR}$ are considered outliers. The sepal width box plot shows these outliers as circles in [Figure 4.3](#).

Mean deviation

Mean deviation (MD) of a set of observations is given by average of the absolute deviations from a central tendency measure A, that is, . Mean deviation is based on all observations and is better than quartile. It can be proved that MD is minimum when A is taken to be the median.

Standard deviation

The square root of the sum squared mean deviations is called **standard deviation** and is denoted by . Standard deviation is considered the best among all measures of dispersion. The square of standard deviation is called the *variance*.

Coefficients of dispersion

If we want to compare variability of two series that differ widely in their means or are measured in different units, we calculate the **Coefficient of Dispersion (C.D)** for each series to compare them. C.D is dimensionless or unit free. This is like a normalized measure of dispersion. For each of the four measures of dispersions discussed earlier, we have separate coefficients of dispersion:

- C.D for range:
- C.D for IQR:
- C.D for mean deviation:
- C.D for standard deviation: ,and is called **Coefficient of Variation (C.V)**

For comparing the variability of two series, C.V is computed, and the series with greater value of C.V is said to have more variability. From the distribution plots of the four attributes in Iris dataset, it's evident that *sepal length* and *petal width* have quite a difference in mean values. However, the standard deviations of these two series are 0.83 and 0.76, respectively, which says the variability of *sepal length* is more than variability of *petal width*. This is contradictory to the histogram plots. *Sepal length* appears to have much less variability. Let's compute C.V for all the series using pandas, as shown in the following code:

```

1. std = df.std() ; cv = 100*std/df.mean()
2. df1 = pd.concat([pd.DataFrame(std).T,
pd.DataFrame(cv).T]).reset_index()
3. df1.index=[['SD', 'CV']]

```

This produces the following output:

Figure 4.4: Comparing CV and SD

We can see that CV for petal width is very high compared to CV for sepal length. This example shows the importance of CV for comparing variability.

Moments

The r^{th} **moments** of a data set about a point A a given by . Clearly, the arithmetic mean is the 1st moment at $A = 0$, and the variance is the 2nd moment about the arithmetic mean. Various moments give us idea about the shape of the distribution. The 3rd and 4th moments are used to define skewness and kurtosis, respectively, which are discussed in the following sections. Higher moments tend to be less robust.

Skewness and kurtosis

Lack of symmetry in the data distribution is called **skewness**. This is measured by the deviation of the given data distribution with a symmetric distribution. A distribution is called asymmetric when $\text{mean} \neq \text{mode} \neq \text{median}$. Skewness is measured by the difference mean-median or mean – mode.

Kurtosis gives us an idea of how sharp or peaked the frequency curve of the data is. It's measured by the ratio of moments . This quantity can be interpreted as follows:

- $\beta_2 = 3$ means a normal frequency curve, neither too sharp not too flat
- $\beta_2 > 3$ means a sharply peaked around the mean
- $\beta_2 < 3$ means a flat and not peaked about the mean

[Figure 4.5](#) shows how skewness and kurtosis define the shape of the distribution. The data distribution is **positively skewed** if $\text{mean} > \text{median} > \text{mode}$. Similarly, we can define negative skew as illustrated in [Figure 4.5](#):

Figure 4.5: Skewness and kurtosis of data distribution

The measures we discussed so far can give us detailed insights into individual attributes or features of observations. This is collectively known as *univariate data analysis*. In practice, the relation among different variables can also give us interesting insights. For example, in study of household expenditure and price or demand of commodities, there is a high chance that these two can increase or decrease together, that is, they covary. The pairwise analysis of various measurements associated with an experiment is called *bivariate analysis*. In the next section, we will discuss how to visualize these relationships and how to measure them.

Correlation

If change in one variable affects the change in other variables, the variables are **correlated**. The correlation is called positive or direct if increase (or decrease) of one variable causes increase (or decrease) in the other. On the other hand, the correlation is called negative if the increase in one leads to a decrease in the other variable and vice versa. Correlation can be detected pictorially with a scatterplot of two variables plotted along the x and y axes. For example, we can take the *petal width* in y-axis and any other attributes, like sepal length and width in x-axis, and plot a scatterplot as shown in [Figure 4.6](#).

We can see that the rightmost plot of petal length versus petal width shows a dense scattering about a nearly 45-degree angled line through origin. Also, this correlation holds for all categories of flowers. For the other plots with sepal measurements, we see less density and more of uniform spread. Refer to the following figure:

Figure 4.6: Bivariate analysis of iris

To generate pairwise bivariate plots for all the four variables, we can use the seaborn library, as shown here:

```
1. import seaborn as sns  
2. sns.set_style("whitegrid")  
3. sns.pairplot(df, hue="flower", size=3) #df of iris data  
defined above  
4. plt.show()
```

Karl Pearson's coefficient of correlation: The degree of **linear relation** between two variables is measured using correlation coefficient. Let take two series of data points and another series , where and are two measurements related to the same individual or event i. The sample **covariance** $\text{cov}(x,y)$ between these two series is the defined as an average of the product deviations . Here, denote the sample means. The correlation coefficient between the two variables is denoted by and is defined as follows:

If can be easily proved that . If the two series are identical, that is, then and if , we have .

Probability and odds

Probability and odds are two different ways of quantifying the uncertainty associated with an event. Let the event be “*winning of a player in a game*”. Odds are the ratios of a player’s chances of losing to their chances of winning. Odds of 3 to 1 for a player means there are 3 chances of losing and only 1 chance of winning. Here, the probability of winning of the player is the ratio of the number of times won to the total number of games, that is:

Now, the probability of losing is similarly defined as the ratio of the number of times the game is lost to the total number of games:

Also, the ratio of these two probabilities is the *odds*:

Therefore, odds express relative probabilities, generally called odds in favour. The odds in favour of an event is the ratio of the probability that the event will happen to the probability that the event will not happen.

Random experiment

An experiment is an activity that produces an outcome. There may be different number of possible outcomes. For example, rolling a die is an experiment with six possible outcomes. An experiment is called **random experiment** if it has more than one possible outcome, and it's not possible to predict the outcome in advance, that is, before the experiment is performed. We know the outcome of a dice roll only after rolling it!

Events as sets

An *event* can be considered as an outcome of a *random experiment*. For the event “*winning of a player in a game*” the experiment is the “*player plays a game*”. There are two possible outcomes of this experiment: the player wins/loses. We can represent these as a set of two events: {‘player wins’, ‘player loses’}. This is the set of all possible events associated with this experiment and is called *sample space*. The singleton events or elementary events {‘player wins’} & {‘player loses’} cannot happen together and are *mutually exclusive events*. The probability of an event is denoted by P.

Suppose n trials of an experiment are performed. The probability of a desired event E is the ratio of trials that result in E, denoted by , to the number of trials performed:

Here, $P(\text{player wins}) = 0.25$ and $P(\text{player loses}) = 0.75$. The probability of these two events are unequal, that is, they are not *equally likely* to happen.

Let’s take another random experiment: “*Rolling a fair dice*”. Here, the sample space is the set $S = \{1,2,3,4,5,6\}$. All the six outcomes are equally likely as it’s a fair dice, and hence, we can write for $i = 1,2,\dots,6$. The event “*roll an even*” is the set $E = \{2,4,6\}$ and the event “*roll an odd*” is the set $O = \{1,3,5\}$. These two events are equally likely: . Also, E and O are mutually exclusive: . E and S are called disjoint events: Also, ; hence, we call E and

S exhaustive events, that is, there is no outcome possible outside the union of these event sets. For mutually exclusive and exhaustive events, we have:

Let's take another event "multiple of 3" represented by . Here, . As shown in [Figure 4.7](#), T has intersection with both E and O. T and O can happen together, and the probability of that is written as ; similarly, . Refer to the following figure:

Figure 4.7: Event set example

Now, the probability of occurrence of either of the events E or T or both, that is, the outcome is either an even number or a multiple of three is represented as:

. Here, unlike mutually exhaustive events:

Basic probability identities

If A and B are two events, then:

- $P(\text{not } A) = P(A') = 1 - P(A)$
- $P(A \cap B)' = P(A' \cup B')$, De Morgan's Law for Probability
- $P(A \cup B)' = P(A' \cap B')$, De Morgan's Law for Probability
- $P(A \cup B) = P(A) + P(B) - P(A \cap B)$
- $P(A \text{ and not } B) = P(A \cap B') = P(A) - P(A \cap B)$
- $P(\text{exactly one of } A, B) = P((A \text{ and Not } B) \text{ or } (B \text{ and not } A)) = P((A \cap B') \cup (A' \cap B)) = P(A) + P(B) - 2P(A \cap B)$

Conditional probability

For the example in [Figure 4.7](#), suppose it's given that the event T has already occurred that is, someone tells that the dice is showing either 3 or 6. Now, if we ask the chances of event E having occurred, it can occur only if the output is 6. So, the number of possible outcomes of event E reduces

from three to one, given another event T has already occurred, and thus, the probability of occurrence of E changes to $1/2$. We can write: .

Definition: The probability of occurrence of an event A, given that B has already occurred, is called *conditional probability*. It is denoted by $P(A|B)$:

Let's take another example of rolling a pair of dice together; we observe the outcome as a 2-dimensional vector representing two face values. The sample space contains $6 \times 6 = 36$ possible outcomes. Let A denote the event “sum of the two face values is 7” and B denote the event “at least one of the face values is 2”, as shown in [Figure 4.8.](#) and . Knowing that B has occurred reduces the possible outcomes for A to 2, that is, . Therefore, the conditional probability of A, assuming that B has occurred, . Now, let's consider that A is given, and we have to calculate $P(B|A)$. We have . So, we saw that the two conditionals $P(A|B)$ and $P(B|A)$ are not the same.

Using the conditional probability definition, we can always write:

This is called the *product rule of probability*. We will be extensively applying this rule in many topics. This can be generalized for a set of n events by repeatedly applying this rule:

Equivalently,

So, we can factor the probability of intersection of three events in any of the $3! = 3 \times 2 \times 1$ ways.

Independent Events

If occurrence or non-occurrence of A does not affect occurrence or non-occurrence of B, then A and B are called *independent events*.

and

Therefore,

Refer to the following figure:

Figure 4.8: Sample space for pair of dice and event sets example

In general, for N independent events, the probability that all the events happen is the product of the N probabilities that the individual events happen.

Let E denote the event that the “*first die shows even number*”; then, . Let O denote the event that the “*second die shows odd number*”; then, . The probability of occurrence of both E and O, that is, . Hence, E and O are independent events.

Note: *Mutually exclusive events are not the same as independent events. Events A and B are mutually exclusive, which means $P(A) > 0$, $P(B) > 0$ and $P(A \cap B) = 0$. However, independence of A and B means $P(A \cap B) = P(A)P(B) > 0$.*

Here, E and O in the experiment are independent but are not mutually exclusive.

Conditional independence

Again, taking the pair of dice example, let A denote the event “*first die show 1*” and B denote the event that the “*second die shows 2*”; then, A and B are clearly independent:

Now, let C be the event that the sum of two outcomes is S. Then, A and B are not independent anymore, given C. So, we call A and B conditionally dependent:

Given one of the outcomes and fixing the sum of two outcomes = S, we can have only one possible result: $A|C = \{(1, S-1)\}$ and $B|C = \{(S-2, 2\}\}$. Hence:

Therefore, $P(A|C)P(B|C) \neq P(A \cap B|C)$.

Two events A and B are called conditionally independent, given event C, if:

$$P(A \cap B|C) = P(A|C)P(B|C)$$

Note: Independence of events does not imply conditional independence, as we saw in the example here. Also, conditional independence does not imply independence of events.

Total probability theorem

Let's consider a set of three mutually exclusive and exhaustive events: and H_3 . So, H_i 's are pairwise disjoint, and the union of all of them gives the entire sample space U.

. Let A be any event. A will have intersection with at least one of the H_i 's as these H_i 's are mutually and exhaustive, as shown in [Figure 4.9](#).

As H_i 's are mutually exclusive, and are also mutually exclusive.

So,

Refer to the following figure:

Figure 4.9: Total probability theorem

For a set of n mutually exclusive and exhaustive events :

Bayes theorem

Let A and B be two events such that $P(B) > 0$; then, using product rule of probability, we have:

This is called Bayes rule. As you can see, this is nothing but an alternate way to write the product rule, but this rule has numerous applications in the theory of machine learning.

Here, $P(A)$ represents the *prior* belief about occurrence of event A. Let B be another related event. Knowing about the occurrence of event B gives some more information about event A, and we call the conditional $P(A|B)$ *posterior* probability.

Let's consider the following example of a medical test for a specific disease:

- A = Event that a person from the population has the disease.
- B = Event a person tests positive (irrespective of whether or not they are sick).
- Given that the test sensitivity is 95%, the test may fail to detect disease in a real sick person for 5% cases. The test has 5% *false negative rate*, that is, $P(B|A) = 0.95$.
- Given test specificity is 97%, that is, the test result is positive for 3% of healthy people without the disease or test has 3% *false positive rate*, that is, $P(B|\text{not } A) \text{ or } P(B|A') = 0.03$.

Survey data shows that the fraction of population having this disease is about 20 in a 1000. This gives us the prior probability of the disease, that is, $P(A) = 20/1000 = 0.02$.

Now, out of these 20 sick people $20 \times 0.95 = 19$ will test positive with the medical test. Also, out of $1000 - 20 = 980$ healthy people, 3% will test positive, that is, 29.4 people will test positive. So, in total, we have $19 + 29 = 48$ testing positive with this test out of 1000. This gives the unconditional probability of testing positive as $P(B) = 0.048$.

Now, given that a person gets positive result, the chance that they are actually sick is $= 0.395$ that is, 39.5 %. This is a simple application of Bayes' rule.

Note: $P(B)$ can also be calculated using the total probability theorem. We have two mutually exclusive and exhaustive events A and A' (complement of A), and by total probability theorem, we have $P(B) = P(B|A) P(A) + P(B|A') P(A') = 0.95 \times 0.02 + 0.03 \times 0.98 = 0.0484$.

Bayes theorem can be generalized for set of n events also. Let U be the sample space. Let be *mutually exclusive and exhaustive events* such that . Let B be any event in U such that $P(B) > 0$. Then:

Bayesian Decision Theory

Bayesian Decision Theory is a fundamental statistical approach to classification problem. It is based on Bayes theorem and measures the risk of assigning an input to a given class. Let's try to understand this with the help of a simple example inspired from the book on pattern recognition by Duda and Hart (*Further reading [4]*). In a fish packing plant, incoming fish on a conveyor belt are sorted to different boxes manually based on type of fish. Suppose there are two species that they ship: “*sea salmon*” or “*sea bass*”. They want to automate this using a robotic arm, which can pick the fish from belt and put it in the appropriate box. The first step for this is to build a classifier that can judge what type of fish is on the belt.

Let A be the event that the fish is “*sea salmon*”; then, A’ represents “*sea bass*”. We assume that the class *prior probability* for finding salmon is known. This may be based on the percent of availability during a particular season or area of fishing and so on.

- **Case 1:** There is no information available other than the class priors. Then, we decide “*salmon*” if , otherwise we decide “*sea bass*”.
- **Case 2:** We got a sensor that can measure lightness of the fish. Also, we know the distribution of lightness for both the fish, as shown in [Figure 4.10](#), which is obtained by measuring lightness from samples of manually sorted fish. From this, we also have the conditional probability of lightness values given the fish category, that is, the probability of lightness value(L) in range , given that fish is “*sea bass*” is . Similarly, given that fish is “*salmon*”, and $P(L \geq 6|A) = (6 + 7 + 7 + 15)/50 = 0.7$.

Now, using Bayes rule we can compute the reverse conditionals, that is, given the lightness value from the sensor, what's the probability of a particular fish type.

The denominator is the same for both the classes; hence, we can predict the class with higher numerator value as the most likely class. Refer to the following figure:

Figure 4.10: Histogram of lightness measure computed from 50 samples of each fish type

So, we can build a simple Bayesian classifier using the Bayes' rule:

if , then fish is salmon else sea bass.

Or , then fish is salmon else sea bass.

Here, θ is the ratio of the prior probabilities or odds against finding salmon, which is given, to , which is the *likelihood ratio* of lightness measures.

Table 4.1: Different Decision Boundaries for lightness measures based on the prior probability ratio ; note how the classification changes based on the prior values

Using this equation, we can now compute decision boundary and derive a rule based on lightness values alone to decide the fish type, as shown in the [Table 4.1](#), for various values of the given prior ratios.

The classifier must be designed to perform well over a range of prior probability values such that the worst overall misclassification risk for any value of the priors is as small as possible. The decision boundary must be chosen such that the maximum possible overall risk of misclassification is minimized. Such a decision boundary is called *Minimax Bayesian Risk* solution.

Random variable

We have represented the outcomes of a random experiment as event sets. These events are abstract objects. To use any of the mathematical tools, we must map these abstract objects to some numbers. A **Random Variable (R.V.)** maps these arbitrary events to real numbers or real vectors. For example, the outcomes of a coin toss are represented as the set $\{H, T\}$. We can define a mapping $X : \{H, T\} \rightarrow R$, as follows, $X(H) = 1$, $X(T) = -1$. Based on the range of values an r.v. takes, it can be categorized as discrete or continuous, as shown in [Table 4.2](#). Refer to the following table:

Discrete random variable	Continuous random variable
Takes distinct values, which are countable (need not be finite) like set of all integers. For example, the r.v. representing coin toss output.	Takes all possible values within an interval, that is, a continuum of values. For example, exact body temperature can be anywhere from 93 F to 105 F, including fractional values like 98.1366421 F.

Table 4.2: Random variable definition

How is a r.v. different from a simple variable? For a simple variable, we cannot say how likely is it that the variable takes a given value or how likely is its value to falls in each range because there is no probability associated with the variables. For r.v., there is an associated probability measure that tells how likely it is that the variable takes certain value or certain range of values. This extra information about the variable distinguishes an r.v. from a simple variable, and it's represented as another function called the **Probability Mass Function (P.M.F)** for a discrete random variable and **Probability Density Function (P.D.F)** for a continuous random variable.

Let the random experiment be tossing a coin three times. Let X denote the count of heads from three-coin tosses. Clearly, X can take values from this set $\{0,1,2,3\}$. From all possible eight outcomes, we can compute the probability of each outcome, as shown in [Figure 4.11](#). For example, there are three outcomes with $X=1$ total head viz $\{HTT, THT, TTH\}$, and hence, . For all $X = 0, \dots, 3$, we can compute the probabilities, and we get a discrete probability density function represented by

Note: and

For any function of a random variable to be a probability density function, it should satisfy these two properties. The density function $f(x)$ is analogous to distribution of unit mass of powder along a line. So, adding up all mass should become 1. Also, we cannot have negative mass on this line; we can have at most unit mass and at least zero mass.

Discrete probability distributions

The probability distribution associated with a discrete r.v is called a discrete probability distribution. A discrete random variable taking only Boolean values 0 or 1 is called a *binary random variable* and one which takes one

out of K discrete values is called a *categorical random variable*. Let's now study a few important discrete distributions that we will be repeatedly using in the subsequent chapters. The p.m.f $f(x)$ of any discrete random variable X has the following two properties:

and

For discrete r.v.

Refer to the following figure:

Figure 4.11: Probability distribution for 3-coin toss experiment

Bernoulli and categorical distribution

The most basic of all discrete random variables is the *Bernoulli*. X is said to have a Bernoulli distribution if $X = 1$ occurs with probability p and $X = 0$ occurs with probability $1 - p$. Here, p is generally called the probability of success. This is like a biased coin toss experiment where head occurs with chance p and tail with chance $1 - p$.

Generalized Bernoulli distribution (Categorical): In Bernoulli experiment, we have only two possible outcomes. We can generalize this to K possible outcomes. For example, we are given an urn of four different colored balls red, blue, green, and orange. Let $X=0,3$ represent the four colors, respectively. We have to find the probability of drawing a given colored ball, where each ball is drawn at random with replacement. Such a r.v is called categorial random variable. In general, a categorical variable X can take one of K possible values $\{1,2,\dots,K\}$ with probability p_i , where $i = 1, 2, \dots, K$. Sample from categorical distribution can be represented as a one-hot-encoded vector of dimension K . In the urn example, a blue ball can be represented as $(0,1,0,0)$, and a green ball can be represented as $(0,0,1,0)$. Let \mathbf{x} denote one-hot-encoded sample from a categorical distribution. Then, the mass function of vector \mathbf{x} is defined by:

Binomial distribution

Consider a sequence of n independent Bernoulli trials with probability of success = p , which is constant for each trial. In n trials, if there are x success and $n-x$ failures, and the probability of x success is given by . However, out of n consecutive trials any x trials can be success, and there are possibilities for that. Then, the probability of x success is given by . The probability distribution of the number of successes is called binomial probability distribution. If X be binomially distributed: , where n and p are the parameters of the distribution.

Figure 4.12 shows binomial distribution histogram plots for various values of n and p . We see if $p=.5$, we get a symmetric distribution, otherwise its either left or right side skewed. Refer to the following figure:

Figure 4.12: Binomial Distribution plots for various values of the parameters n, p

Poisson distribution

All distributions we studied so far have finite and known set of possible outcomes. Now, if we take the number of calls received every minute in a call center or the number of vehicles at a traffic signal at a time of day, it's not possible to predefine any max possible value. To define the distribution of such count-based random variables, Poisson distribution is used. A discrete random variable X is said to be Poisson distributed with parameter $\lambda > 0$ if it has a probability mass function given by:

Continuous probability distributions

Probability distribution of a continuous random variable X is called continuous probability distribution. The p.d.f $f(x)$ of a continuous random variable has the following properties:

and for all x

As X can take infinitely many values, the probability of X taking on any one specific value is zero. This can be explained with the following example. Suppose a species of bacteria typically lives 4 to 6 hours. What is the probability that a bacterium lives exactly 5 hours? A lot of bacteria live for approximately 5 hours, but there is negligible chance that any given bacterium dies at exactly 5.000000000... hours. The probability of X is given by the integral:

However, the probability of X assuming any fixed value, that is, $P(X=a) = 0$. Now, let's now study a few important continuous distributions useful for understanding any ML algorithms.

Note: For continuous probability density $f(x) \neq P(X = x)$, $f(x)$ represents the density of the probability mass around the point $X = x$. The probability of the continuous random variable X taking any exact value x is zero. However, for discrete distributions, $f(x) = P(X = x)$. Also, $f(x)$ can be greater than one for some values of x for continuous distributions. For discrete distributions, however, $f(x) \leq 1$ always as it represents a probability.

Example: The diameter of an electric cable manufactured from a factory can be assumed to be a continuous random variable X . Suppose the density function of X is given by:

First, let's verify that $f(x)$ is a density function. Clearly, for $x < 0$ and $x > 1$,

Now, we can also find the probability of producing a wire of length more than $2/3$ units, as follows:

Here, $f(x) = 3x^2$. So, density of a continuous distribution can be greater than 1. However, the integral of the p.d.f over certain interval represents a probability and is always less than or equal to one.

Cumulative Probability Distribution Function (C.D.F)

As the name suggests, this function gives the cumulative probability of X . This function gives probability, and hence, $F(x)$, that is, the total probability of X taking all possible value should add up to 1. Also, $F(\infty) = 1$. Being a cumulative function, $F(x)$ is always monotone increasing:

if

Using $F(x)$, we can define probability of X in an interval as follows:

Another very important property of c.d.f is as follows:

c.d.f can be also defined for discrete distributions with integral replaced by sum.

Uniform distribution

Let X be a random variable that can take any real value in the closed interval $[a, b]$. X will be called uniform distributed if it takes all values with equal probability $\frac{1}{b-a}$. So, we have $F(x) = \frac{x-a}{b-a}$, for all $x \in [a, b]$. As, $f(x) = \frac{1}{b-a}$ is probability density function, $f(x) \geq 0$. This implies $\int_a^b f(x) dx = 1$.

Therefore:

c.d.f of uniform distribution is $F(x) = \frac{x-a}{b-a}$. You can find it plotted in [Figure 4.13](#):

Figure 4.13: (Left) Uniform Density function (Right) Uniform cumulative distribution function

Gaussian distribution or normal distribution

A continuous random variable can take infinitely many values. However, most of the naturally occurring continuous random variables, like heights of people, blood pressure, students' scores in a test, and the exact dimensions

of an object produced by a machine, are observed to have bell shaped distribution when plotted as a frequency distribution, that is, if we plot a sample of values taken by these random variables, we get the “bell shaped” histogram shown in [Figure 4.14](#):

Figure 4.14: Gaussian Distribution

Normal distribution also arises naturally if we take the distribution of the sum of large number of any random variable. Let X be uniformly distributed in the interval $[0, 1]$ and $Y = X_1 + X_2 + \dots + X_n$. Then, as n increases, [Figure 4.15](#) shows that the distribution of values taken by the sum Y becomes closer to the bell curve. This property of Gaussian is formally stated as the *central limit theorem*. Refer to the following figure:

Figure 4.15: (Left) Uniform Density function (Right) Uniform cumulative distribution function

Now, let's look at the probabilistic interpretation of the bell curve in the preceding figure. This bell curve in [Figure 4.14](#) has its peak at $x = 75$. The chance of finding a value of the random variable decreases as we move away from this center point in both the directions. So, the probability is inversely related to the distance of x from the center point. In fact, the chance of observing a value x of the random variable X decreases *exponentially* as we go away from the center point. Let's denote the center point by μ . The normalized distance of any value x from the center point μ is, where $\sigma > 0$ is a measure of scatter or dispersion of the data around the center point μ .

The probability density function for Gaussian distribution is given by:

Here, $\frac{1}{\sigma\sqrt{2\pi}}$ is a normalization term required to make this a probability density function that satisfies $\int_{-\infty}^{\infty} f(x) dx = 1$. This density function has two parameters: **center** location and scale. It's denoted by $f(x)$.

Relation of continuous Gaussian distribution to discrete binomial distribution

Let X be a random variable with distribution $B(n, p)$. If n is large enough, is a good approximation for $B(n, p)$, where $\mu = np$ and . The normal distribution is generally considered to be a decent approximation for the binomial distribution when .

The dispersion can be used to divide the area under a normal curve, starting from the center location. The normal density plot shows how likely it is to find a value within a specific distance from the center location . This is called the *Empirical rule or three sigma rule*.

- Approximately 68% of the data will fall within the interval
- Approximately 95% of the data will fall within the interval
- Approximately 97.5% of the data will fall within the interval

[Figure 4.16](#) shows the empirical rule for normal distribution:

Figure 4.16: Empirical rule for normal distribution

If , then is a *standard normal variate* that is, . This is proved in a later example. The standard normal probability density function is denoted by .

The c.d.f of is denoted by . However, evaluating this integral is not very straightforward. One way is to expand the exponent term as a power series and then integrate each term of series and compute the infinite series sum. There are precomputed probability distribution tables available for standard normal distribution, which give for equispaced values of z , as shown in [Figure 4.17](#). Using this table, we can compute the probability in the interval for standard normal distribution . The entry in the table depicts the probability for row label r and column label c . For example, to compute probability , we take and and get $P(Z \leq 0.36) = 0.64058$. Refer to the following figure:

Figure 4.17: Cumulative Distribution

Using the Python “**scipy.stats**” library, we can compute the c.d.f for many continuous distribution functions, as show in the following snippet:

```
1. from scipy.stats import norm
```

```
2. print("P(Z<0.36) = ", norm.cdf(0.36))
3. print("P(1.5<Z<3.2) = ", norm.cdf(3.2)-norm.cdf(1.5))
```

Example: Quality of a product produced by a machine is measured on a scale of 0-100. If an old machine outputs 58% products in the quality range less than 75, 38% products are between quality range of [75 to 80] and only 4% were above 80. Assuming that the quality metric for each machine is distributed normally, can we find the mean quality metric and the variance of the quality metric?

Let the random variable X denote the quality index of the products. Let μ denote the mean and σ^2 denote the variance of the distribution and Z be a standard normal distributed, given μ , σ^2 and $P(75 \leq X \leq 80) = 0.38$. Referring to the standard normal cumulative probability table discussed in [Figure 4.17](#), we can rewrite these probability equations in terms of standard normal variable Z , as follows:

Here, we have used the inverse of the c.d.f function called *percent point function*: $ppf(0.58) = \Phi^{-1}(0.58) = 0.20$. *Percent point function* takes probability as input and computes the corresponding x for the cumulative distribution function. This is also implemented in “**scipy.stats**” as **norm.ppf()** function:

Solving the previous two linear equations, we get $\mu = 75$ and $\sigma^2 = 25$.

Standard normal distribution being symmetric around the origin, we have $P(X < 0) = P(X > 0) = \frac{1}{2}$. This is also evident from the c.d.f plot for normal distribution showing $\Phi(x) = 0.5$.

Exponential Distribution

The exponential distribution is another popularly used continuous distribution. It is often used to represent the time elapsed between events.

The density function of a continuous random variable X following exponential distribution is denoted by $X \sim Exp(\lambda)$, $\lambda > 0$ and is defined as follows:

Example: Airline tickets are booked in advance, and generally, the number of days ahead we book a ticket depends on factors like whether it's planned travel or unplanned. Is it during holiday season? Are the prices going to shoot up. The number of days ahead travelers purchase their airline tickets is observed to follow an exponential distribution with an average of 15 days.

Mathematical expectation of a random variable

Probability distribution of a random variable X tells us the likelihood of X taking a specific value or falling in a specific interval. In practice, a more easily interpretable information will be the average value taken by the random variable. For example, a company may be interested in the average profit they are going to make on a new product being launched. A pediatric doctor may be interested in the average height and average weight of a 5-year-old. The average value of a random variable is also termed as the **mathematical expectation** of the random variable denoted by $E[X]$.

The expected value of a discrete random variable is the weighted average of all possible values given by:

For continuous random variables, we must replace sum by integral and get the definition:

Example: Let X denote the amount of time (minutes) a person must wait for an elevator in a high-rise building. Here, X is a continuous random variable with the following distribution:

The expected value of X is given by the following:

So, the expected wait time for the elevator in the building is 1 minute. Readers can plot this density function and observe that the maximum

density is also at $x = 1$ in this case.

We can calculate the expectation of the random variables with known probability distributions discussed earlier. This is shown in the following table:

Distribution(X)	Mean or $E[X]$	Variance $E[(X - E[X])^2]$
$X \sim Bernoulli(p)$	p	$p(1 - p)$
$X \sim Binomial(n; p)$	np	$np(1 - p)$
$X \sim Poisson(\lambda)$	λ	λ
$X \sim N(\mu, \sigma^2)$	μ	σ^2
$X \sim Exponential(\lambda)$	$1/\lambda$	$1/\lambda^2$
$X \sim Uniform(a, b)$	$1/2(a + b)$	$1/12(b - a)^2$

Table 4.3: Mean and variance of few commonly used distributions

We can also define the expected value of a function of a random variable $g(X)$ as follows:

For , the expectation of $g(X)$ is called the r^{th} moment about the mean of X . In particular, if $r = 2$, then the expectation of $g(X)$ is called *variance* of the random variable and is often denoted by σ^2 .

Here are a few properties of expectations:

- Expectation is a linear operator. Hence, the expectation of a linear combination of random variables is given by the following:
 -
 - for any constant a.
- The expectation of a product of mutually independent random variables is as follows:

-
-

- **Cauchy-Schwartz inequality:** If X and Y are two random variables, then , where equality holds if and only if for some real .
- **Jenson's Inequality:** Let g be a convex function of the random variable X , then .

Joint Probability Distributions

Till now, we have discussed probability distributions for a single random variable. In real life, we are often encounter several random variables that are correlated. Here are a few examples:

1. In ecology, one species may be prey of another, and hence, the number of predators will be related to the number of prey. If we are modeling the counts of the prey and predator as two random variables X_1 and X_2 , there must be some dependency between them and the probability of these random variables assuming some pair of values. So, it makes sense to model the joint probability of the variables (X_1, X_2) .
2. Suppose we are studying about different families in a locality. We can represent statistics like household income, number of family members, highest education in family as different random variables X_1 , X_2 , and X_3 . There are high chances that these variables are dependent on each other, and we study the distribution of them jointly.

Let X , Y be two discrete random variables such that X takes n distinct values and Y takes m distinct values . The joint probability mass of X and Y is defined as . For example, let X denote the length and Y denote the width in millimeters of a plastic cover manufactured in a packaging unit of factory. If we round off X , Y to the nearest integer, they take a discrete set of values. Let $X \in \{200, 201, 202, 203\}$ and $Y \in \{300, 301\}$. Refer to the following table:

Length X

Table 4.4: Joint distribution example

The sum of all the probabilities in the table is 1.0. Given a joint probability distribution for X and Y, the individual probability distribution for X or for Y can be easily derived from the joint distribution.

$P(X = 202) = P(X = 202|Y = 300) + P(X = 202|Y = 301) = 0.33$ = column sum of the previous joint probability matrix, for 3rd column where $X = 202$.

$P(Y = 301) = P(Y = 301|X = 200) + P(Y = 301|Y = 201) + P(Y = 301|X = 202) + P(Y = 301|Y = 203) = 0.17 + 0.20 + 0.26 = 0.63$ = row sum of above joint probability matrix, for 2nd row, where $Y = 301$.

The probability distribution $P(X)$ appears in the column sums and the probability distribution $P(Y)$ appears in the row sums, as shown in [Table 4.4](#). As they appear in the margin of the table, these are terms as the *marginal probability distribution* of the joint distributions.

Joint probability density function in two variables X and Y denoted by , is called *bivariate probability density function*.

The marginal densities are given by the following:

These definitions of joint probability and marginals can be extended to n dimensions and are called *multivariate probability density function*. To find the marginals along any dimension, we must add the probability along $n - 1$ remaining dimensions. For example, marginal along the x-axis for a three-dimensional distribution is:

For continuous random variables also, we can define the joint distribution and the corresponding marginals by replacing the sums with integrals. Integral in two dimensions represents integral over a region in two dimensions, and it gives the area of the region. For continuous distribution, we have:

Probability of X, Y falling into a region \mathbf{R} is given by:

The marginal probability densities are given as:

and

Note: The conditional density function for $Y|X = x$ can be defined as:

For mutually independent random variables X, Y the joint distribution $f_{xy}(x,y)$ can be represented as the product of the marginals:

$$f_{xy}(x, y) = f_x(x) f_y(y)$$

We can represent n jointly distributed random variables as a random vector . This random variable X is vector valued and assumes values in the Euclidean space . The multivariate density of this random vector is represented by or simply as , where x is a vector in .

Expectation of jointly distributed variables

We have defined the mathematical expectation of a single variable in the previous sections. For two jointly distributed random variables X and Y , we can define the expectation of a function as follows:

, if X, Y are continuous

And

The joint central moment (r^{th} central moment of X and s^{th} central moment of Y) is denoted by and is defined as follows:

3. If , we call it covariance and is denoted by $\text{COV}(X, Y)$. So:

4. If , the central joint moment reduces to variance of X , that is,

5. If , the central joint moment reduces to variance of Y, that is,

The $COV (X,Y)$ can be equivalently written as $COV (X,Y) = E[XY] - E[X]E[Y]$. The proof of this is simple. $COV (X,Y) = E[(XY - XE[Y] - YE[X] + E[X]E[Y])] = E[XY] - E(X)E[Y] - E[Y]E[X] + E[X]E[Y] = E[XY] - E[X]E[Y]$.

Example: Let's consider two random variables X and Y having joint density function:

The density is plotted in [Figure 4.18](#), where each line represents contour of constant density parallel to the line . The lighter shades represent high density regions. As we move away from the origin, the density decreases parallel to the line . (However, the white space between lines doesn't not represent high density regions. The density varies parallel to the lines there as well.) Refer to the following figure:

Figure 4.18: Bivariate Density Contours (the diagonal lines represent fixed density)

Let's now calculate the marginal probability density functions:

We can also calculate the conditional distributions and :

Knowing the marginal distributions, we can also calculate the expectations of the individual variables:

Similarly, . Now, we can calculate the variance of the two variables as follows:

To compute the covariance of the two variables, we need :

Hence, we can now calculate the *covariance* using the following:

Theorem: Two independent random variables are uncorrelated. X, Y are independent implies . However, the converse is not true, that is, *two uncorrelated variables need not be independent.*

Example: , . Here .

However, $E[XY] = \times [-4 \times 16 - 3 \times 9 - 1 \times 1 + 1 \times 1 + 3 \times 9 + 4 \times 16] = 0$ and $E[X] = 0$.

Therefore, $COV(X,Y) = 0$, and hence, $\rho_{XY} = 0$, but X, Y are not independent as $Y = X^2$.

Transformation of a random variable

Let X be a random variable and $g(\cdot)$ be a function; then, $Y = g(X)$ is also a random variable. Here, are random vectors, and hence, g must be vector field. We can represent g as such that:

Where, for all i are continuous differentiable functions.

If represents the joint density function of X , then the joint distribution function of Y is given by . Here, J denotes the Jacobian matrix of the random variables, and represents the determinant of the Jacobian, and hence, is a polynomial.

If are real valued random variables, then the Jacobian simplifies to simple derivate; we write:

Example: Let , where X is a random variable distributed as . We have:

Hence, the density function of the random variable Y is given by

Example: Let and , that is, ; we have . The distribution of Z is

Multivariate distributions

Now, let's look at a few important multivariate distributions that will be used in defining many ML models in the later chapters. We will also look at the marginal distributions for some of these multivariate distributions.

Multinomial distribution

Consider a sequence of n independent categorical trials with categorical probability vector p , where $\sum p_i = 1$. which are constants for each trial. For example, in the ball and the urn experiment, an urn contains many balls of K different colors. The color of the ball drawn is categorically distributed, and the sequence of n such trials with replacement will be multinomially distributed. In n trials, if x_i represents the count of category i obtained, then we must have $\sum x_i = n$. These different categories can be obtained in $\binom{n}{x_1, x_2, \dots, x_K}$ ways, and the probability of obtaining this category distribution is $\prod p_i^{x_i}$. Let X be multinomially distributed $M(n, p)$. Here, X is a vector: (x_1, x_2, \dots, x_K) and we write $P(X = x) = \prod p_i^{x_i}$.

, where

Text data can be modelled as a multinomial distribution over the words. Assuming that the text is a collection of words, given a set of text documents, we can define a vocabulary set of K distinct words occurring in the entire document collection. Based on the number of times a word occurs in each document, we can represent each document as a categorical variable.

Multivariate gaussian distribution

For a vector x , the multivariate Gaussian distribution takes the following form:

where μ is d -dimensional mean vector, Σ is a $d \times d$ covariance matrix, and $\det(\Sigma)$ is the determinant of Σ . This is denoted by the notation $\mathcal{N}(x | \mu, \Sigma)$.

We will encounter this form of Gaussian distribution in several topics across the book. So, we must understand this in detail. First, we will start with the geometric interpretation of this density function, which will give us

an idea of how Gaussian distributed vectors are placed relative to each other in . Also, we will prove that $p(x)$ indeed represents a density, that is:

Representing as a vector \mathbf{dx} , we have a succinct representation of the normalization equation, that is,

The power in exponent term is a quadratic form since , being the covariance matrix, can be taken to be symmetric; hence, its inverse is also symmetric. Here, Δ is called the *Mahalanobis distance* from to x , and it reduces to the Euclidean distance when Σ is the identity matrix.

Since represents a quadratic form, so , represents the surfaces of constant probability density because the rest of the terms in the expression of are independent of x . Let's consider two-dimensional Gaussian distributed random variable diagonal covariance matrix and . Diagonal covariance matrix means that the variables are mutually uncorrelated but have different dispersion or spread. We have The quadratic form simplifies to:

So, represents an ellipse with major axis of length along the axis and minor axis along of length along the axis, as shown in [Figure 4.19 \(left\)](#). If is not a diagonal matrix, that is, the variables are correlated, then we will have a tilted ellipse, as shown in [Figure 4.19 \(middle and right\)](#).

Anywhere on the ellipse for a given c , we have the same fixed probability density . Refer to the following figure:

Figure 4.19: Elliptical contours of constant density for various covariance matrices

In [Figure 4.19](#), the point on ellipse along x-axis is and point along y-axis is . The Euclidian distance of these points from the origin are and , but we have the same probability density at these points given by . This is in contrary with Gaussian distribution of single variable, which is symmetric around the center , and hence, points that are not equidistant from mean have a different probability density. Now, instead of Euclidean distance, if we consider Mahalanobis distance, then both the points and are at the same distance from the center . The Mahalanobis distance measures the distance relative to the center or centroid of the distribution.

Mahalanobis distance is commonly used to find *multivariate anomalies or outliers*, which indicates unusual combinations of two or more variables. For example, it's quite common to find a 6 feet tall woman weighing more than 180 pounds, but it's very rare to find a 4 feet tall woman who weighs that much.

For single variable x , the Mahalanobis distance of x from the center is given by . This is also known as the *standard scalar* for any single variable x . Given a data sample, we can approximate by the sample mean and sample standard deviation, respectively. Hence, the Mahalanobis distance for a single variable is the number of standard deviations; a sample observation x is away from the sample mean. Standard scalar is very useful and is applied to data attributes before model building.

Note: The elliptical shape of this distribution is guaranteed if the covariance matrix $\Sigma > 0$, positive definite; so, if the eigen values of are all positive, then the quadratic form Δ^2 is guaranteed to represent an ellipsoid.

Now, let's see whether the multivariate normal density function is normalized. As Σ is a real symmetric matrix, the eigen values of Σ are all real. Let λ be the eigenvalues of Σ and \mathbf{v} denote the corresponding eigen vector. Then, we have the following eigen value equation:

Also, **eigenvectors of real symmetric matrices are orthogonal**. Hence:

(where I_{ij} represents the $(i,j)^{\text{th}}$ element of identity matrix)

Also, Σ is diagonalizable and can be written as , where Λ is a diagonal matrix of d -dimension with the eigenvalues of Σ as the diagonal entries and U being an orthogonal matrix with column vectors . Now, we can write matrix Σ as a sum of d matrices whose only one diagonal entry is non-zero and equal to λ_i and all other entries of the matrix are zero. So, we have:

Here, \mathbf{e}_i denotes the standard coordinate basis vectors or one hot vectors with only i^{th} non-zero entry as 1. Therefore, we can write matrix Σ as follows by multiplying the preceding equation by $\mathbf{e}_i \mathbf{e}_i^T$ from both sides:

Similarly,

Substituting this in the Mahalanobis distance expression, we have:

Taking as the simple dot product , we have:

We can interpret as coordinates of point x in a new coordinate system defined by the orthonormal vectors whose origin is] shifted to and axes rotated to align along the orthogonal eigen vectors. Let vector , then . This is depicted in [*Figure 4.20*](#) for 2-dimentional Gaussian distribution:

Figure 4.20: Elliptical contour of constant density, and major and minor axis are defined by the eigenvalues

For the Gaussian distribution to be well defined, it is necessary that all the eigenvalues , otherwise the distribution cannot be properly normalized. In that case, the axes length of the ellipsoid is .

As we have done a coordinate change, we must compute the Jacobian matrix

Matrix U being orthonormal and . Hence, matrix Therefore,

Here:

Now, we are all set to check the normalization condition of multivariate Gaussian. We will reduce to a product of d Gaussian density functions of single variable with zero mean and variance .

This shows that multivariate normal density function is normalized. This proof gives us a detailed understanding of the structure of multivariate Gaussian, which will be used in various models later.

Information theory

Any information can be broadly thought of as the resolution of uncertainty. Knowing that someone has passed an examination may not be very surprising, but knowing that someone topped the examination is quite surprising. So, information can be viewed as the ‘degree of surprise’. Information is more valuable when it’s about an unlikely event. Thus, information content is associated with the inverse of the probability of an event. Let’s denote the information about an event A by a function $H(A)$. If A and B are two unrelated events, the information gain from observing both should be the sum of the separate information gained from each of them. that is, $H(A,B) = H(A) + H(B)$. Also, as A, B are unrelated or independent, we have $P(A,B) = P(A)P(B)$. Hence, the log function is a good candidate to relate probability to information gain. Also, H must be inversely related to P; hence, we define information gained from observing an event A as follows:

The negative sign here ensures that information content is always positive or zero, the base of the logarithm is arbitrary, and the choice of base gives a unit of the information measure. If measured with base e with natural logarithm, the information is measured in *nats*. If measured with base 2, the information is measured in terms of binary bits.

Entropy

Let X be any discrete random variable following a probability distribution $p(x)$. A sender wants to communicate values of X to a receiver. The average information transmitted can be computed by taking the expectation of information sent, that is:

The average amount of information $H[X]$ needed to specify the state of a random variable X is called the *entropy* of the random variable. For a discrete distribution, computation of entropy is simple. Let X be a discrete r.v taking integer values 1 to 5 with probability $p = \{0.1, 0.6, 0.05, 0.05, 0.2\}$. Then, taking logarithm with base 2, we get the entropy in bits as $H = -(1 \times \log_2.1 + .6 \times \log_2.6 + 2 \times .05 \times \log_2.05 + .1 \times \log_2.1 + .2 \times \log_2.2) = 1.67$.

In [*Figure 4.20*](#) the leftmost distribution is sharply peaked at $x=2$, and the entropy of this distribution is the lowest as compared to the middle and right distributions, which are spread evenly across more values. The rightmost distribution is uniform with maximum entropy. We can prove that the maximum entropy configuration is achieved by uniform distribution for a discrete r.v., taking M possible discrete values with probabilities $p = \{p_1, p_2, \dots, p_M\}$, where .

We can think of H as a scalar field , (introduced in [*Chapter 3, Vector Calculus*](#)) which transforms and M -tuple of probabilities p to the corresponding entropy and maximize H using Lagrange multiplier to enforce the probability normalization constraint , as follows:

Refer to the following figure:

Figure 4.21: Elliptical contour of constant

This shows that the *maximum entropy* configuration for discrete distribution is the *uniform distribution*.

Similarly, entropy for continuous distributions can be defined as follows:

This is called *differential entropy* or continuous entropy. Differential entropy can be negative, unlike discrete entropy.

Now, let's find the maximum entropy configuration for continuous distributions. For a continuous distribution with given mean , the maximum

entropy configuration is attained by exponential distribution . If the second moment or variance of the distribution is also specified, then the maximum entropy configuration is normal distribution . These can be also proved using the same Lagrange multiplier trick.

Entropy can be defined for joint distribution of variables X, Y in the same way using the joint probability distribution :

Using product rule of probability, . So, we can write:

Or,

Since, , we can write as:

Here, is called the *conditional entropy*, that is, the remaining entropy of Y, given that X has taken a specific value x.

Relative entropy or KL divergence

Suppose a sender wants to communicate values of a random variable X to a receiver. X follows some unknown distribution $p(x)$ that we have approximated with $q(x)$. If $q(x)$ is used to construct a coding scheme for transmitting values of x to a receiver, the average additional amount of information (in nats) required to encode X is given by the difference:

This is denoted by the notation and is called relative entropy or *Kullback-Leibler divergence, or KL divergence*. This also gives a measure of difference between two probability distributions $p(x)$ and $q(x)$ over the same random variable X. However, it's not a symmetric measure, that is, . So, we cannot call it a distance function.

KL divergence is always positive, , where equality holds if and only if $p(x) = q(x)$. The proof of this follows from the convexity of -log function.

We know that for a convex function f , given two points a, b , we have the following for

Jenson's inequality generalizes this to n points: Given a convex function f and n points in its domain:

Let $p(x)$ represent any discrete distribution of discrete random variable X ; then, . Replacing by in the Jenson's inequality, we have:

For continuous random variables also, the previous inequality holds, and we can write:

Now, $-\log(x)$ is a convex function. So, we have:

Or,

Therefore, , since

We will be using this property of KL divergence in many theories later, like for variational inference in [Chapter 12: Generative Models](#). KL divergence is not symmetric, that is, . **Jensen-Shannon (JS) divergence** is a symmetrized and smoothed version of the KL divergence defined by:

This is used for measuring the similarity between two probability distributions.

Mutual information

Let's consider the joint distribution of two random variables X, Y denoted by $p(x, y)$. If X, Y are independent we can write . If the variables are not independent, then we can measure how close they are to independent

variables by considering KL-divergence between p and q . This is called mutual information between the variables X and Y denoted by

Since KL divergence is non-negative, we see that where equality holds if and only if, x and y are independent that is, p and q are independent. Also, it's trivial to see that . that is, mutual information is symmetric.

We can re-write mutual information in terms of conditional entropy as follows:

This relation can be derived using the definition of KL-divergence and product rule of probability. Following are some properties of which follows from the conditional entropy-based definition above.

- 6.
7. are independent
8. mutual information is symmetric

Using these concepts of entropy and information gain, we can come up with a simple algorithm for classification and regression called decision trees.

Decision tree

A decision tree is a simple but powerful supervised machine learning algorithm used for solving both classifications and regression problems. Decision tree training algorithm recursively partitions the data set into smaller and smaller subsets using certain criteria based on information gain or mutual information. Let's illustrate this with a simple example of how a fruit can be categorized as orange or lemon given height and width of a fruit measured, as shown in [Figure 4.22 \(rightmost\)](#):

Figure 4.22: decision tree example

We have some sample data collected for a set of fruits, that is, their height and widths as plotted in [Figure 4.22 \(leftmost\)](#). The tree in the middle shows a decision tree where every rectangular node is a decision node. The

first decision node is at the root where the entire data set is present. The check whether the fruit $width > 7 \text{ cm}$ or not splits the data set vertically into two parts. Based on the height of the fruit, the two vertical splits of the data are split horizontally at $height = 6 \text{ cm}$ for the left vertical split and at $height = 10 \text{ cm}$ for the right vertical split. Finally, we get four splits of the data, and only one class of fruit is predominant in each of the four splits. This tree representation of the data set is called *decision tree*. The final nodes are called leaf nodes of the tree. Leaf node represents a class. The class label of the majority data points in the leaf node is the class represented by leaf node.

Now, given the height and width of a new fruit outside this data set, we can predict whether it's orange or lemon using the path it follows from root to leaf. The majority class of fruit in the leaf node will be the predicted label for the new fruit. For example, in the left-bottom partition of the data set shown in [Figure 4.22](#), there are only two instances of lemon, and rest are all oranges. So, if a new fruit falls in this data partition, as its $width \leq 8$ and $height \leq 7$, we can classify it as orange as most of the instances falling here from training data are oranges, and only two are lemons.

The decision tree shown earlier can be built by visual inspection of the data set. How do we generalize this for larger datasets? There are two primary questions we need to answer:

1. Attribute selection: Which attribute or feature shall we split the dataset?
2. Where to split for a given attribute?

For the classification problem, with categorical target variable, we can calculate how much “information” an attribute gives us about the class. Let’s represent the target by the random variable and the attributes fruit height, and width as the random variable . We can calculate the information content or entropy of a set containing oranges and lemons.

$$H[Y] = -p(y = \text{orange}) \log(p(y = \text{orange})) - p(y = \text{lemon}) \log(p(y = \text{lemon}))$$

[Figure 4.23](#) show the entropy calculation for different proportions of fruits:

Figure 4.23: High purity implies lower entropy

If we take the entire data set of fruits, we will have an impure node as it has all classes in possibly equal proportions. Next, we need to measure the reduction of this impurity in our target Y, given additional information or attributes X. We can use mutual information or information gain for this: $I[Y,X] = H[Y] - H[Y|X]$. We have seen how to calculate $H[Y]$. Now, let's compute $H[Y|X]$ for each attribute height and width. Here, height and width are continuous variables, and we must find a suitable point x to split the data set into two parts $height \leq x$ and $height > x$. For discrete attribute, we can evaluate entropy for every discrete value of X. Algorithms like **Iterative Dichotomiser 3 (ID3)** and its improvements. like C4.5, CART are used to construct decision trees.

We will take a sample data set of 20 observations of lemons and 20 oranges and compute the information gain for few split values, as shown in [Figure 4.24](#):

Figure 4.24: Entropy computations for building decision tree

We have shown calculation for only three split values, but it calculated for all distinct values of continuous attributes.

Entropy of target, given split height of, is calculated using weighted average:

The entire data set has 9 lemons and 11 oranges. Hence, the entropy of whole data set:

The information gain for the split point $height = 6.31158$ is given by:

Among the split points considered, the maximum information gain is obtained for a split point of width = 8.2. So, the best attribute to choose is width, and the split point for this attribute is 8.2. So, width attribute

becomes our first attribute to split the data set into two parts and create two nodes in the tree.

We can now use the Python **sklearn DecisionTreeClassifier** module to build decision tree and visualize the tree using **graphviz**, as shown in [Figure 4.24](#). Here's the sample code:

```
1. """Building synthetic dataset"""
2. from np.random import multivariate_normal
3. np.random.seed(62)
4. lemon1 = multivariate_normal(mean=(7, 10), cov=[[1, .5],
[.5, 1]], size=20)
5. lemon2 = multivariate_normal(mean=(6, 8), cov=[[1, .5],
[.5, 1]], size=20)
6.
7. orange1 = multivariate_normal(mean=(10, 8), cov=[[1, 0.5],
[0.5, 1]], size=20)
8. orange2 = multivariate_normal(mean=(4.5, 4.5), cov=[[1, 0.5],
[0.5, 1]], size=20)
9. orange = np.concatenate([orange1, orange2])
10. lemon = np.concatenate([lemon1, lemon2])
11.
12. df = pd.DataFrame(np.concatenate([lemon, orange]))
13. df.columns = ['height', 'width']
14. df["fruit"] = ['lemon']*len(lemon)+['orange']*len(orange)
15.
16. """Training DT Classifier"""
17. from sklearn.tree import DecisionTreeClassifier
18. from sklearn.tree import export_graphviz
19. from sklearn.externals.six import StringIO
20. from IPython.display import Image
21. import pydotplus
22. clf = DecisionTreeClassifier(criterion = 'entropy',
min_samples_leaf=10)
23. clf.fit(df[["height", "width"]], df["fruit"])
24.
25. dot_data = StringIO()
26. export_graphviz(clf, out_file=dot_data,
filled=True, rounded=True,
```

```

28.         feature_names = df.columns.values[:-1],
29.         class_names=['lemon', 'orange'])
30. graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
31. Image(graph.create_png())

```

[Figure 4.24](#) shows the decision tree plot output from previous code. Here, the tree is built on the entire dataset. However, we should follow all the standard model training steps, like splitting training data into train-test-validation, and then fine-tune model hyperparameters using validation test and report classification accuracy using train step. Refer to the following figure:

Figure 4.25: Visualizing a decision tree

Decision trees can be used for both regression and classification tasks. They provide very fast inferencing and are lightweight as compared to many other ML models.

Decision trees are prone to overfitting, and it happens if the tree is allowed to grow without any restriction to very high depth. It tries to fit to every single training data point. We need to prune the tree using hyperparameters like the following:

- **max_depth**: Maximum depth of the tree
- **min_impurity_decrease**: Node will be split if this split induces a decrease of the impurity greater than or equal to given value
- **min_samples_leaf**: Minimum number of samples required to be at a leaf node

Here, we have used **min_samples_leaf** as the only metric for pruning the tree.

Conclusion

In this chapter, we discussed descriptive statistics and fundamentals of probability theory. We covered measures of central tendencies, dispersion, and correlation for univariate and bivariate data analysis. We also discussed random variables, distributions, and expected values of functions of random variable. Additionally, we introduced the concept of entropy and discussed

its application to entropy-based decision tree construction. These topics will prove to be a rigid foundation for the advanced topics that we will be learning in this book.

In the next chapter, we will begin with inferential statistics. We will see how to use probability concepts to formulate machine learning problems like classification, regression, and clustering as a probability problem. Then, we will cover how inferential statistics, hypothesis testing and parameter estimation techniques, can help us solve these problems.

Points to remember

- Correlation measures the degree of linear relation between two random variables. Random variables X and Y are independent , but X , Y are independent.
- Probability of a random variable taking value a , , the probability density at a , if X is continuous. Probability density at a point can be greater than one. The area under density curve for univariate density is 1.
- If X is continuous random variable, .
- The normal distribution is generally considered to be a decent approximation of the binomial distribution $B(n, p)$ when .
- If are independent random variables, then their joint density can be written as the product of the marginal densities: .
- Mathematical expectation is a linear operator.

Further reading

- The Elements of Statistical Learning: Data Mining, Inference, and Prediction. New York: Springer. Hastie, Trevor, Robert, Tibshirani and J. H. Friedman
- Fundamentals of mathematical statistics, S C Gupta, V K Kapoor Publisher: New Delhi: Sultan Chand & Sons
- Christopher Bishop, Pattern Recognition and Machine Learning, [Chapter 1](#) and [2](#)

- Pattern Classification by David G. Stork, Peter E. Hart, and Richard O. Duda
- Decision Tree:
https://saiconference.com/Downloads/SpecialIssueNo10/Paper_3-A_comparative_study_of_decision_tree_ID3_and_C4.5.pdf
- <https://www.math.arizona.edu/~rsims/ma464/standardnormaltable.pdf>
- William Feller: An Introduction to Probability Theory and Its Applications

CHAPTER 5

Statistical Inference and Applications

In statistics, *population* is the entire set of items or individuals that we want to study and draw some conclusions about. The number of individuals in a population is generally very large or infinite, so a study is often restricted to few *samples* drawn from it. **Statistical Inference** is the method of making propositions about a population from sample data drawn from the population. In this process, some assumptions are made about the population, and then a statistical model is built based on those assumptions. Making statistical inferences about the parameters of a probability distribution, assumed by the population, is called parameter estimation. In **Machine Learning (ML)**, the training data can be viewed as a sample from some unknown population. We make certain assumptions about the population and build a statistical model that is capable of making correct inferences on both the training data as well as unseen data, that is, on newer samples that are not part of training data. The process of building the model is termed as *training* or *learning*, and using the model for prediction is referred to as *inference*.

Structure

In this chapter, we will cover the following topics:

- Large Sample Theory
- Statistical Inference and Parameter estimation
- A good estimator and how to find it
- Formulating ML problems in probabilistic terms
- **Linear models:** Linear and Curvilinear regression
- **Generalized Linear models:** Logistic, Poisson Regression
- Interpreting linear models

Objectives

After going through this chapter, you will be able to connect to the statistical foundations on which the well-known ML models stand. There are several probabilistic assumptions behind each model, and the error functions that we optimize to train a model is also derived from the probability theory. Even as we move on to the next chapter on neural networks, we will see how the concepts learned in this chapter can be applied and extended for deep neural nets on which most of the AI relies today. The interpretation of the learned models, for example, the coefficients of a linear regression model, is based on statistical hypothesis testing.

Large Sample Theory

In statistics, a *population* is the exhaustive set of events associated with a given experiment. In the Iris example, the set of all possible measurements of the flower categories is the entire population. The dataset of 150 observations that we have is a finite subset or a sample from the entire population of flowers. Here, the entire population is infinite, and it's not possible to analyze the entire population of data to derive conclusions. Hence, we need to work with finite samples of data. A good sample should ideally represent all the characteristics of the population. Then, we can accurately calculate the population characteristics by only analyzing the sample characteristics. There are many techniques for proper sampling from a population. The two popularly used in ML are *random sampling* and *stratified sampling*:

- **Random sampling:** Samples are chosen at random such that each unit in the population has an equal chance of being selected. In case of a finite population of size N , if we decide to choose samples of size k , we can take any subset of k data points from the population. There are $M = \binom{N}{k}$ total possible combinations of k data points. Here, we are assuming *sampling with replacement*, that is, an event or observation is selected at random from the population. Before drawing the second sample, the first sample is returned to the population. This makes the probability of selection of each item equally likely. For random

sampling, each of these M samples have equal probability of being selected.

- **Stratified sampling:** Here, the population is divided into small homogeneous groups called strata, and then random sampling is applied on each of the stratum. This technique accurately reflects the population being studied, especially when the population is diverse. For example, a survey is being conducted for an education institute that offers science, mathematics, and humanities courses. The number of humanities and mathematics students in the college are almost double the number of science students. A random sample of size 10 will most likely contain 4 students from mathematics, 4 students from humanities, and 2 from science. However, the survey should give equal importance to all streams and must choose an equal number of students from each stream. Here, stratified sampling can be used to randomly choose equal number of students from each stream who can participate in the survey.

Sample statistics

The population characteristics quantified the population constants like mean μ and standard deviation σ , and the statistical constants of the population are called *parameters of the population* and the corresponding measures computed from samples of size k , like sample mean and sample s.d. are called *statistic*. A statistic, denoted by t , is a function of the sample values. In case of a finite population of size N , for each sample of size k , the statistic t can be evaluated. Let's denote a sample of size k by \mathbf{x} , M being the total number of samples. Evaluating the statistic t for each sample \mathbf{x} , we have a new set of M values t_1, t_2, \dots, t_M . Thus, the statistic t defines a random variable T , and we call its distribution the *sampling distribution of the statistic*.

We can now compute the expectation of T :

The **Standard Deviation (SD)** of the sampling distribution of T is called the **Standard Error (S.E.)**. The S.E. plays very important role in large sample theory. If T is any statistic, then for large samples:

The standard error for a few test statistics is listed here. Here, σ is the population variance, and k is the sample size. The SE can be reduced by increasing the sample size. Refer to the following table:

Statistic	Standard Error
Sample Mean ()	
Sample S.D. (s)	
Sample Median	

Table 5.1: Standard Error

Note: The SD of a sample measures the amount of variability or dispersion of the sample relative to its mean, while the SE of the sample mean statistic measures how far the sample mean (average) of the data is likely to be from the true population mean. The SE is always smaller than the SD.

For defining sampling distributions, we need to take multiple samples and estimate the statistics for each of the samples. In many practical scenarios, we may not always have large amounts of data to take several samples.

Tip: If the sample size is very small, the normality assumption does not hold, and we have to find the exact sampling distribution, followed by the statistic t. In this case, the Student's t-distribution is used in place of the normal distribution if we have small samples. t-distribution has heavier tails, that is, it can produce values that fall far from mean. For a large sample size, t-distribution looks like the normal distribution. The parameter of t-distribution is dependent on the sample size and is called the degree of freedom. A small sample of n observations from a normal distribution is said to be t-distributed with $v = n - 1$ degrees of freedom. Refer to [4] in the Further Reading section for more on t-distribution.

Now, let's look at another aspect of sampling. Suppose the probability distribution of a population is known, that is, the parameters of the

distribution are known; can we create some synthetic sample data points from this population?

Sampling from known distributions

Given a distribution with known population parameters, we can generate samples from this distribution. For example, given a categorical distribution with categories having probability , we want to generate a sample from this distribution. A sample of size one most likely will be category as it's of the highest probability. A sample of size two will possibly be or or .

The CDF of the categorical distribution is . A pseudo-random number generator will give a random number from uniform distribution. We can partition this interval into consecutive intervals of lengths given by CDF, that is, . Now, if r belongs to k^{th} partition, we can output k^{th} category as the sample. As the length of the 2^{nd} partition is the highest, the chances of r falling there is more, and we will generate more samples from there. [Figure 5.1](#) shows histograms of three different random sample, each of size 20, from this categorical distribution:

Figure 5.1: Samples from known distribution (Categorical)

For any general distribution the algorithm, to do this is called *inverse transform sampling*. Here, we first sample from a uniform distribution using a pseudo random number generator, and then using the CDF of target distribution, we can find the value that matches the same quantile in target distribution. This is implemented in the **numpy.random** package for most of the known distributions.

Hypothesis testing

In statistics, the assumptions made about a population characteristic based on samples from the population is called *hypotheses*. An example of a hypothesis is: “sample mean is the same as the population mean”. So, if we know the population to be normally distributed with some unknown parameters , then the hypothesis is . We need a method to check whether a hypothesis is statistically significant and valid for the entire population or just for the sample chosen.

Let's consider an example of an electric bulbs manufacturing company that has invented a new manufacturing process. This process is expected to produce bulbs with higher life span. An expert team of scientists reviewed the process and concluded that this new process may be as good as the old one. So, the expert's hypothesis is that the mean life span of new process is equal to the average old life span . In fact, a few scientists from the expert committee believed the new process to be worse than the previous one, based on theoretical analysis. For the company to take a decision about the adaptation of the new process, a statistical hypothesis testing needs to be performed. For the old process, the population mean μ is known and for the new process, we have observed the lifespan of good enough sample; we take the average as the sample mean.

Following are the steps for testing of hypothesis:

1. **Define null hypothesis (H_0):** An assumption that is tested for possible rejection; for example: .
2. **Define alternate hypothesis (H_1):** Any hypothesis complementary to null hypothesis is called an alternate hypothesis. Example (called *one-tailed test*) or (called *two-tailed test*).
3. **Choose the level of significance (α):** We may commit two types of error in hypothesis testing:
 - **Type I error:** Rejection of when it is true
 - **Type II error:** Accepting when it is wrong, that is,

Let and

The probability of type I error is also known as the *level of significance*. This must be chosen in advance based on the amount of risk we are allowed to take. Typical values of are 1%, 5% that is, 0.01, 0.05.

4. **Choose test statistic:** Based on the hypothesis, a corresponding statistic (T) must be chosen. For a large sample:
5. Given a level of significance , we can define a test of significance.
A critical value of the statistic Z at significance level is defined as for two-tailed test. Here, , implies . This is depicted in the shaded regions

in [Figure 5.2](#). So, we have . By symmetry, . For testing alternate hypothesis with one tail, we will have only one shaded region, either left or right, based on the type of test, as shown in [Figure 5.2](#); the shaded region is called *critical region or rejection region*:

[Figure 5.2: Critical region for hypothesis testing](#)

6. Conclusion: The computed value of Z is compared with the significance value at a given level of significance . If the value of Z falls in the defined critical region, the null hypothesis is *rejected* with confidence . Choosing , we can say that the null hypothesis (that is, the claim of expert committee here) is false and can be rejected with 99% confidence.

Now, let's look at a few examples on hypothesis testing where we follow the mentioned steps to validate the statistical hypothesis.

Example: Testing whether a die is fair or unbiased (that is, all six face values are equally likely to occur). A die is thrown 9000 times, and the number of times values 5 or 6 is observed is 3600. Is the die unbiased?

For an unbiased die, the probability of each face is . We can take the event of getting 5 or 6 as success and denote it by S . We have . Getting any value other than 5 or 6 is considered a failure. So, we have a binomial distributed variable with trials, the probability of success , and the number of successes observed is 3600. Here, the null hypothesis is that is , and the alternate hypothesis is . Let random variable X denote the count of success. For binomial distribution, we have mean as and variance as :

Hence, for large n

Therefore,

Since choosing the significance level , and we know this by the empirical rule for normal distribution. Hence, falls inside the critical region or rejection region. So, we can reject the null hypothesis with confidence of 99%. Hence, the die is not a fair die.

[Statistical inference](#)

Let us consider a random variable X with probability density f . Here, θ represents the set of parameters of the distribution or more formally, θ represents a parameter vector. For example, in case of normal distribution, $\theta = (\mu, \sigma^2)$. The set of all possible values of the parameters is called the *parameter space*. Thus, the parameter space defines a family of probability distributions:

Now, let's consider a random sample of size n from a population, with probability density function f . Can we estimate the parameter vector θ as a function of sample values, that is, can we define a statistic t that can approximate a population parameter? Let vector \mathbf{x} , then we want to find sample statistic:

such that the distribution of t is concentrated around the true value of the population parameter θ_0 . Here, $\hat{\theta}$ depicts the estimator function for statistic t . In this case, the statistic $\hat{\theta}$ is called an *estimator of the population parameter*.

Estimator properties

Let's now discuss what properties of a statistic makes it a good estimator:

- **Unbiasedness:** An estimator $\hat{\theta}$ is called an *unbiased* estimator of the population parameter θ_0 if $E(\hat{\theta}) = \theta_0$. Suppose, for an unknown population with probability distribution f , we are given the expectations: $E(X) = \mu$ and $E(X^2) = \mu^2 + \sigma^2$. The sample point x_1, x_2, \dots, x_n can be viewed as a set of n **independent and identically distributed (i.i.d)** random variables X_1, X_2, \dots, X_n , assuming values x_1, x_2, \dots, x_n each. Then, each X_i has mean μ and variance σ^2 . For the statistic sample mean $\bar{X} = \frac{1}{n} \sum_{i=1}^n X_i$, we have $E(\bar{X}) = \mu$.

Hence, the sample mean is an *unbiased* estimate of the population mean. The sample expectation of variance is as follows:

Reference to the proof is given in the *Further reading section [3]*. Hence, *sample variance is not an unbiased estimate of population variance*.

- **Consistency:** For any estimator $\hat{\theta}$, we can define a sequence of variables as follows: $\hat{\theta}_n$. Here, $\hat{\theta}_n$ is dependent on the number n of sample values taken. We call $\hat{\theta}$ a consistent estimator of parameter value if this sequence of values converge in probability to the true parameter value. We denote this as $\hat{\theta} \xrightarrow{P} \theta$, where the operator \xrightarrow{P} indicates convergence in probability, that is, for any arbitrarily small $\epsilon > 0$:

Clearly, the sample mean is a *consistent* estimator of the population mean.

- **Efficiency:** There may exist multiple unbiased and consistent estimators for a population parameter. So, there is a need for some further criterion to choose the best estimator. The one with lower variance is called an *efficient* estimator because it will tend to have values that are concentrated more closely around the correct value of the parameter, and hence, our estimate will be closer to the actual value. For normal distribution, because of the symmetry of the bell curve, the sample median is also an unbiased estimate of the population mean μ . Now, which one should you choose as a better estimator? Here, we can compare the variance or square of standard error of both the estimators. Now, it can be proved that for sample median M_d , $\text{Var}(M_d) > \text{Var}(\bar{X})$. Hence, the sample mean is a more efficient estimator of μ .
- **Sufficiency:** An estimator is called sufficient if it contains all information in the sample regarding the population parameter being estimated. This property of an estimator is out of scope of further discussion in this book.

Minimum Variance Unbiased (M.V.U) estimators

If a statistic T is unbiased estimator of a population parameter μ and T has the smallest variance among the class of all unbiased estimates, then T is called the minimum variance unbiased estimator of μ . Formally:

-
-

The MVU is unique. To check whether an unbiased estimator is MVU, **Cramer-Rao inequality** is used. This inequality provides a lower bound

for the variance of an unbiased estimator of a parameter. The quantity is called the *bias* of the estimator in general. For unbiased estimator, bias is zero.

Likelihood function

Let X be a random variable following a probability distribution . A given random sample from this probability distribution can be viewed as a set of n *i.i.d* random variables assuming values and each . We define the joint density function of these random variables as the *likelihood function* L :

The joint density function is expressed as a product because the random variables are independent. Clearly, likelihood function is *always positive*, . Applying logarithm to both sides converts the product to a sum of logs, that is, . This is called the *log likelihood*.

Cramer-Rao inequality

If T is an unbiased estimate of a function of the population parameter , then:

So, this provides a lower bound to the variance of an unbiased estimator. Here, the quantity in the denominator is called the *Fisher information on the parameter θ* contained in the sample and is denoted by . The higher the Fisher information value, the more information there is in the sample about the parameter.

If we want to estimate parameter , the function can be taken as the identity function , and hence, . So, Cramer-Rao inequality takes the following form:

In other words, the precision (that is, inverse of variance) to which we can estimate θ is limited by the Fisher information.

We will prove the Cramer-Rao inequality for univariate distribution with a single parameter under certain regularity assumptions. Understanding this proof is important as it shows us a way to find a MVU estimator.

Regularity assumptions:

- Parameter space is open interval in R
- The partial derivative exists
- exists and is positive for all
- Differentiation under the integral sign is possible for the p.d.f function

Proof of Cramer-Rao inequality: Since L is a joint p.d.f of sample where , differentiating w.r.t and using regularity conditions, we get:

Let be an unbiased estimator of . Therefore, .

Differentiating both sides w.r.t and using regularity condition to differentiate the integral:

Here, T and are two real random variables that take a fixed real value for any given sample. The covariance of any two r.v. X, Y is . So, here:

The correlation . Using Cauchy-Swartz inequality for expectations, we have shown that , if and only if X and Y are linearly dependent.

Also, , by definition of .

Therefore, . Here:

Substituting this variance in above inequality and rearranging, we arrive at the Cramer-Rao inequality:

Now, we will see when this minimum variance bound is attained by an unbiased estimator. Rearranging the previous equation, we have

Here:

This is in the form of Cauchy-Swartz inequality for expectation and there we have seen that this reduces to equality if and only if random variable X is linearly dependent on Y . Hence, the equality will happen if and only if t is linearly dependent on y , that is:

$t = \theta + \sum_{i=1}^n a_i y_i$ is a constant

Here, θ may depend on y but t is independent of the samples. In this case, the unbiased estimator T is called a **Minimum Variance Bound (MVB)** estimator.

Example: Suppose y_1, y_2, \dots, y_n is a random sample from a normal distribution with known zero mean and unknown scale parameter σ . An MVB for σ is

As, $t = \sqrt{\frac{1}{n} \sum_{i=1}^n y_i^2}$ statistic T is an unbiased estimate of σ . Now, let's see if T can attain the Cramer-Rao lower bound.

and likelihood of the sample is as follows:

Thus, t is linearly dependent on y because t is independent of the samples. Hence, t is an MVB for σ . Similarly, we can prove that the MVB estimator mean of normal distribution is sample mean, given that the population scale parameter is known. We suggest that you try this out.

So far, we have discussed how to check the quality of the estimator and choose the best, but the estimator, being any arbitrary function of the samples, can take any functional form. How can you define an estimator? Is it based on intuition, or there is a procedure to find a possible estimator? This is answered in the following section.

Method of Maximum Likelihood Estimation (MLE)

The principle of maximum likelihood consists of finding a value of unknown parameter such that , for all possible values of . This defines an optimization problem of optimizing a scalar field , where denotes the parameter space.

Since , and is a non-decreasing function of L , L and $\log L$ attain their optimum values at the same point. Hence, we can maximize $\log L$ instead of maximizing L . Using differential calculus, we can find the optimal value by solving . To make sure this equation indeed gives us the maximum value of likelihood, we must do the second derivative test , or the Hessian matrix of must be negative definite for vector valued parameter.

Example: Let's find the MLE estimate of the parameter for Poisson distribution.

Suppose is a sample from the population and the likelihood function is given as follows:

Also, , since all samples from Poisson distribution are positive, making always positive. The variance of the estimator can be obtained by the Cramer-Rao minimum variance bound:

Also, is linearly dependent on , and hence, the equality in Cramer-Rao inequality should hold, that is, .

Note: A more general result as follows: If a given population with density and MVB estimator T exists, the solution to likelihood equation is T .

Also, MLE estimates are consistent and efficient. We will just state the following two theorems regarding that:

Theorem (Cramer-Rao): Given a sample of size n , as the likelihood equation has a solution that converges in probability to the true value of the parameter .

This theorem states that the MLE estimates are consistent.

Theorem: *If MLE estimates exist, then they are the most efficient in the class of estimators.*

Example: MLE estimates for parameters of univariate normal distribution are given as sample from the univariate normal distribution . The is given by

We know that the estimate of is not unbiased. Hence, **MLE estimates need not always be unbiased estimates.**

Example: MLE estimates for multivariate Gaussian distribution. Suppose we have sampled from a multivariate Gaussian distribution. We want to estimate the parameters of the distribution using MLE. Here is a vector and is a matrix. Hence, we need to compute the partial derivatives of log likelihood w.r.t a vector and w.r.t a matrix. The derivation is easy to understand with the background of vector calculus discussed in [Chapter 3](#). Interested readers may refer to *Further reading [5]*.

Note: The MLE estimates are as follows:

There are other methods of estimation like method of least squares, method of moments, Bayesian parameter estimation (MAP). We will briefly talk about Bayesian estimation where we estimate the parameters with the help of the Bayes' Rule.

We have to find the value for that maximizes the posterior probability ; this is called the **Maximum Posteriori (MAP)** estimate for the parameter .

The denominator $P(X)$ is ignored because it has no direct functional dependence on the parameters with respect to which we want the right-

hand side to be maximized. As with the MLE, we can take the logarithm of the posteriors and have:

So, the only difference between MLE and MAP is that the latter allows us to inject into the estimation calculation of prior beliefs regarding the parameters. Now, if we use the simplest prior in the MAP estimation, that is, uniform prior, we assign equal weights everywhere, on all possible values of the , that is, . In this case, the MAP estimate is , which is the same as the MLE.

Bias-variance decomposition of estimator

We have defined the bias of an estimator as . Treating statistic T as a random variable, we can compute the mean-squared error (MSE) of T in estimating parameter as: , where represents the value of the statistic for ith sample.

We can easily prove that . The proof is as follows:

Hence, . This is called the bias variance decomposition, that is, both *bias* and *variance* contribute to MSE.

Applications – Formulating ML problems as statistical inferencing

In this section, we will discuss how different ML problems can be formulated in terms of probability theory. Then, we will see that learning a model is nothing but estimating parameters of a probability distribution using the techniques we have learned so far.

Data distribution

Suppose we are given a data set with N records. Each record has m attributes or features, that is, each data sample is a vector. We can view any data point as a random vector. Each feature can be viewed as a random variable taking values from the i^{th} attribute column of the data set. There are chances that many of these attributes are *not* mutually independent. Hence, we can represent the data set as a sample taken from a multivariate joint probability distribution of the random vector.

Generally, after the exploratory data analysis stage in practical machine learning, the raw observations are transformed to new engineered features. This can be viewed as a vector field mapping, where are fixed non-linear functions of the input variables and are called *basis functions*. Note that here, d may be less or greater than the raw data dimension k . s can be chosen such that the feature s is mutually uncorrelated. Here are a few examples of basis functions:

- Suppose raw data vector is one-dimensional. We can define feature mapping as , that is, , called polynomial basis functions or radial basis functions:

Here, we are mapping a single feature x to a d -dimensional feature vectors whose components are not (linearly) correlated.

- Let data vector be three-dimensional . We can define , and have a two-dimensional feature map . Here, is a polynomial basis function also known as interaction term.

Classification

In a classification problem, an input vector or feature vector must be assigned to one of K classes, denoted by . We are given a dataset consisting of n pairs of examples , called training data. Using this, we need to come up with a function f that can map . The simplest form of such a function is a lookup table. But is that useful? Can it map unseen , (not in the training data) to a class label?

Assuming that the data vectors come from some probability distribution, we can represent the data set by a random variable . Depending on whether constitutes categorical or continuous attributes, follows either discrete or

continuous probability distribution . Representing the target class by a categorical random variable which takes one of the K possible values, will have a categorical distribution . If we take the subset of training data from any particular class , then the probability of data vector can be represented by the conditional probability and we have the *class-conditional* probability density function . This is also known as *likelihood* of with respect to . Here, both the density functions f and p are unknown. The target distribution g can be easily estimated from the probability definition. Taking the ratio of the number of examples from the given class to the total number of examples n in the training dataset. This is called the *class prior* probability. The classification problem can be defined as estimating the probability . This is called the *posterior probability of class* . If we know this probability for all k , then we can predict the class label associated with as follows:

Using Bayes, theorem we can write:

The denominator is fixed for all the K classes; hence, we can write this as:

So, we have *posterior probability of class* $C_k \propto \text{likelihoood} \times \text{class prior}$

This formulation of classification problem as a probability model is called *Bayesian classifier*.

Now, let's look at one simple classifier based on this interpretation of the pattern classification problem named Naive Bayes classifier.

Naive Bayes classifier

Suppose the data has m features, that is, , and these features are mutually independent, given that x is from class . Formally:

Substituting this in the Bayes equation, we have:

Hence, the probability of a new data \mathbf{x} belonging to class k is given by:

This is Naïve Bayes classifier.

Here, the *prior* probability estimation is easy, as discussed earlier. We can also assume all classes to be equally likely, that is, . For estimation of , we must figure out the distribution of each of the attributes . If is discrete, we can assume multinomial distribution or categorical distribution. If s is continuous, we can assume Gaussian distribution.

For continuous attributes, the mean and variance of the attributes can be estimated for each class . Let's call them and , respectively. Hence, we now have:

Note: We can also discretize continuous attribute by properly binning the feature values. For discrete attributes, suppose we take to be categorical distribution. Suppose the discrete attribute has T categories in total and represents the number of times category t appears in the samples from class k , and let represent total number of samples from class k . Hence, we have:

Now, it may happen that in some class , all the T categories for are not present. This could be because of the limited samples that we have taken. So, it will be inappropriate to assume that category t for the attribute cannot appear in class k . Let be the total number of categories of the attribute . Then, we can rewrite:

Here, is a smoothing parameter, which takes care of the missing category in the samples from any class, where by assigning some non-zero value.

Regression

In a regression problem, an input vector must be assigned to a real number y or a real vector \mathbf{y} . We are given a data set consisting of n pairs of examples , called training data. Using this, we need to come up with a function f that can map . Here, y is continuous. Let's first understand this with a simple example where the input vector has only one attribute and target y is real. Let represent height of a father and y represent height of his adult son. We have a data set of size $N = 30$ consisting of pairs , and we have plotted it in [Figure 5.3](#) (left):

Figure 5.3: Simple Regression (left) height of adult son vs father's height (right) Curve fitting data

We see from the plot that the son's height can be modeled as a deterministic function f of father's height plus some random noise, that is:

In [Figure 5.3](#)(right), we can see that, at father's height x , $f(x)$ underestimates the son's height, and we can add some positive quantity to get the corresponding adult son's height y . For some values of x , $f(x)$ may overestimate the son's height, and we may have to subtract some variable quantity . So, the noise ϵ can be both positive and negative. Also, the random noise is assumed to be symmetrically distributed or centered along the curve $f(x)$. So, one natural choice for the distribution of the random noise is Gaussian distribution with mean zero and some fixed variance , that is, . So, y is a random normal variable shifted by $f(x)$. Hence, we can write y :

The next step is to understand how to find $f(x)$. We can assume f to be a parametrized function of the input, say $f(x)$ is a polynomial of degree 2, where are the parameters. We can represent the parameters as a vector and we can denote f as . So, we now have .

So, we have regression formulated as a probability model parameter estimation.

Here, the parameter vector w is unknown, and we can estimate it using MLE technique that we will discuss in greater detail in the linear model section. We will see that the MLE estimation technique boils down to

minimizing sum squared error function. This formulation of regression as a probability model can be generalized for vector input \mathbf{x} and target vector \mathbf{y} .

Linear and curvilinear regression

Given a training dataset , the goal is to find a *linear* function f such that , for all i , that is, approximates . This f will be called a linear regression model. So, f takes the form , where is called the *bias*. All these s are represented collectively as a vector

Now, the training data set is a sample of size N from a large population. A good model is one that will generalize well for most of the population and not only for this training sample. We are going to make a few assumptions about the population:

- **Linearity:** The relationship between and the mean of y is linear. As shown in the following figure, the y values are scattered around a mean line. ‘Regression’ means stepping backward, towards the average. The function f models the average of the target y .
- **Homoscedasticity:** As shown in [Figure 5.4](#), the scattering of the target y values about the mean line is constant and is not a function of x , that is, the variance of *residual or error* $|f(x) - y|$ is the same for any value of x .
- **Normality of errors:** For any fixed value of x , the target y is normally distributed, that is, the residuals are normally distributed.
- **No autocorrelation of errors:** There should be no correlation between the residual (error) terms. Presence of autocorrelation may drastically reduce the accuracy of the model.
- **No multicollinearity:** The components of the data vector x , that is, the independent variables, should not be correlated. As shown in [Figure 5.4](#), the regression model can be viewed as a distribution or equivalently . Refer to the following figure:

Figure 5.4: Normality assumptions and constant variance around line

Let’s see whether this model satisfies the assumptions mentioned earlier. Based on assumption 1, can be taken as a linear function of x , that is, .

Here, ϵ is an unknown constant representing the constant scattering as in assumption 2. The residue or error term ϵ , satisfying assumption 3. Assumption 5 expects all the components of the vector to be uncorrelated. We can ensure this by some feature engineering, that is, using fixed basis functions discussed earlier, to transform the data, and we obtain , where f .

f is a $(M+1)$ dimensional vector of features with β_0 is required to estimate the bias parameter β_0 . This is depicted in [Figure 5.5](#). Now, the location parameter of the normal distribution μ is represented as a function of vector f here. So, β represents the parameters of the model that we can estimate using MLE. Refer to the following figure:

Figure 5.5: Fixed basis function model

Estimating model parameters

First, let's write the likelihood expression for the probability model. The log likelihood function is given by .

The weights β are independent of n , and we can take it out of the summation, that is,

Let's write these M equations in matrix form using the design matrix:

we can write the r.h.s as ϵ , and the l.h.s. as $(X\beta)$. Hence, we have the system of linear equations $(X\beta = \epsilon)$. The coefficient matrix (X) is square matrix, so we can solve this as:

Note: The bias parameter can be interpreted using the first equation . We have set $\beta_0 = 0$. Therefore, $\beta_0 = \bar{y}$, that is, $\bar{y} = \beta_0 + \beta_1 x_1 + \dots + \beta_M x_M$. Hence,

The bias is the difference between the mean of the target values occurring in entire dataset and the weighted sum of the averages of the

basis function values. Geometrically, is the intercept of the line with the y-axis, that is, for single variable case, it represents c in equation , the point where the regression line crosses the y-axis.

Iterative estimation of model parameters

The term in the log likelihood expression represents the total sum-squared-error (SSE), where is the error term contributed from data point . Dividing by N, we get the MSE or mean squared error. Thus, for fixed , the log L depends on the SSE alone. Also, maximizing is the same as minimizing , that is, minimizing SSE. So, the weight vectors w can be estimated by iterative optimization method like stochastic gradient decent. For very large datasets, directly finding the solution for regression requires computation of , and this is a very costly computation. Here, an iterative method like gradient descent is preferred. As discussed in [Chapter 3, Vector Calculus](#), in gradient descent, we initialize the weights with random values denoted by and iteratively update the weight as in the direction of negative gradient: . Here, is the learning rate, and represents the gradient of the error term, that is:

Example: Let's take the iris data again. We saw in a previous chapter that the features *petal length* and *petal width* are correlated ([Figure 5.6](#)). Given the petal width, can we predict the petal length in cms?

Here, we are not doing any further feature engineering; thus, to apply the linear regression model, we will take , the identity mapping. First, shuffle the data and split it into two parts: train (120 data points) and test (30 data points). Then, we train a linear regression model and plot ([Figure 5.6 \(Left\)](#)) the learned regression line, as shown in the following code:

Figure 5.6: Regression with single dependent variable

Using the identity basis function will impose severe limitations on the model.

Example: (curvilinear regression) We have the two-dimensional synthetic data generated by the process , as shown in [Figure 5.6 \(middle\)](#), with a line,

we will have a very bad model. This is where the basis functions can help. Taking polynomial basis functions, we can map the single variable and then train a linear model, [Figure 5.6](#) (right). The learned model is . This is a non-linear function of x but its *linear in the coefficients*, so we still call it a linear model. It's also known as *curvilinear regression* or *polynomial regression* model. The following code provides the process of data generation and model fitting:

```
1. from sklearn.preprocessing import PolynomialFeatures
2. from sklearn.linear_model import LinearRegression
3. import matplotlib.pyplot as plt
4.
5. def sin(n):
6.     x = np.linspace(0, 1, n)
7.     noise = np.random.normal(loc=0, scale=0.3, size=n)
8.     y = np.array([np.math.sin(2*np.pi*a) for a in
x.tolist()]) + noise
9.     return x, y
10.
11. X, y = sin(100)
12. polynomial_features= PolynomialFeatures(degree=3)
13. features =
polynomial_features.fit_transform(np.expand_dims(X, axis=1))
14. model= LinearRegression().fit(features, y)
15. line=model.predict(features)
16. plt.scatter(X, y)
17. plt.plot(X, line, color='black')
```

In the experiment, will we get a better model if we choose to model using higher-order polynomial instead of a degree three polynomial? This is illustrated in [Figure 5.7](#):

Figure 5.7: Overfitting and underfitting

Overfitting and underfitting

We must check how well our machine learning model generalizes the new data and whether it's able to learn the inherent pattern in the training data.

The poor generalization performance is termed as *overfitting*. This happens when the model learns the pattern as well as the noise in the training data. The inability to learn the pattern in training data because of some assumptions made by the model is termed as *underfitting*. These are responsible for the poor performances of the machine learning algorithms.

The model with 1-degree polynomial is a poor fit, one with 3-degree polynomial is a good fit, and a model with a high-order polynomial is an overfit.

Let's now look at the coefficients of the fitted model with nine-degree polynomial shown in the first row in the following table:

Table 5.2: Coefficients of regression model

The magnitude of the coefficients, specially of the higher order, is huge. This means that the model has done some fine adjustments to the higher order terms so that it can twist and turn and make a perfect fit polynomial for the training data. So, we may alleviate this problem by restricting the magnitude of the coefficients. This is achieved by putting a constraint like to the SSE minimization, as follows. Using the Lagrange multiplier trick, we add a regularization term to the SSE to control over-fitting so that the error function to be minimized takes the following form:

We can choose $\|\cdot\|$ to be Euclidean norm or $\|\cdot\|_1$, and in that case, we still have a closed form solution: . This choice of regularizer is known as *weight decay* in machine learning literature as in sequential learning algorithms; it leads the weights to take small values close to zero. This is also known as *Ridge regression* in statistics literature. The parameter λ is a hyperparameter that must be carefully chosen. The following code shows Ridge regression for 9-degree polynomial where we choose $\lambda = 0.01$. Very small values of λ allow the model to become finely tuned to the noise on each individual data point, nullifying the effect of regularization.

1. `from sklearn.linear_model import Lasso, Ridge`
2. `reg = Ridge(alpha=0.01).fit(features, y)`
3. `line=reg.predict(features)`

If we choose to be norm, then we call the regression *Lasso regression*. It has the property by which some of the coefficients are driven to zero; if we choose to be sufficiently large, this leads to a sparse model in which the corresponding basis functions become redundant.

Note: Over-fitting is an unfortunate property of MLE and can be mitigated by regularization or with Bayesian parameter estimation. In Bayesian estimation, as we consider the prior beliefs into the optimization, it naturally includes the regularization terms. Interested readers may refer to book reference in further reading [2] [Chapter 3](#) section 3.3.

Bias variance trade-off

As we saw in case of the synthetic data, the perfect model is . In general, we can assume that the regression function is an estimation of an unknown function , such that: . Representing our linear regression model as , the expected squared error at a point x is: Here, being a function of sample values can be viewed as a statistic, and it estimated the real model parameter . Using the bias-variance decomposition discussed earlier, we can rewrite this as the sum of and , that is:

Here, is the estimate of the sample statistic The estimate of w is based on some data set D . So, we need multiple datasets to compute these expectations. For, the synthetic *sin* dataset, we can generate multiple data sets and observe the relation between bias and variance as we change the model complexity by controlling the regularization parameter . [Figure 5.8](#) depicts this with the in the :

Figure 5.8: Bias variance trade-off as a function of model capacity

We have generated 200 datasets of size 25 and fit separate ridge regularized model with various regularization parameter . We have used polynomial basis functions of degree 11 as the base model. As we vary , we get models with varying capacity because higher values of will make the coefficients of higher order terms negligible, and hence, we end up having simpler

models, which may underfit the data. Similarly, in the extreme left, we have very low values of λ , which can allow large coefficients to higher order terms, making it overfit. From the model capacity plot in [Figure 5.8](#), we can see that the minimum value of λ occurs around $\ln \lambda = -4.5$, which is close to the value that gives the minimum error on the test data. Following is the code for this bias variance trade-off. The sin function used here is defined in the code section before [Figure 5.7](#).

```

1. datasets = [sin(25) for i in range(200)]
2. d = 11; bias_vars = {};
3. for lam in np.linspace(0.001, 1.1, 100):
4.     preds = []; biases = []; variances = []; sses = [];
5.     for X, y in datasets:
6.         polynomial_features= PolynomialFeatures(degree=d)
7.         features = polynomial_features.fit_transform(
8.                         np.expand_dims(X,
axis=1))
9.         reg = Ridge(alpha=lam).fit(features, y)
10.        line=reg.predict(features)
11.        preds.append(line)
12.        sses.append(np.mean(np.square(line-
np.sin(2*np.pi*X))))
13.        E_y = np.mean(np.array(preds), axis=0)
14.        bias_square = np.mean(np.square(E_y -
np.sin(2*np.pi*X)))
15.        variance = np.mean(np.square(line - E_y))
16.        tot = bias_square + variance
17.        test_error = np.mean(sses)
18.        bias_vars[np.math.log(lam)]={'$bias^2$+variance':tot,
19.                                    '$bias^2$':bias_square, 'variance': variance,
20.                                    'test_error':test_error}
21. pd.DataFrame(bias_vars).transpose().plot()

```

In practice, we have only seen the single observed data set. Then, how can we use this bias variance trade-off to choose the optimal hyperparameter λ . We can use k-fold cross-validation or leave-one-out validation to choose the best possible experimentally.

Logistic Regression

For classification problem, the dependent variable is discrete, and hence, normality assumption in **Linear Models (LM)** does not hold true. So, we cannot directly apply linear regression model for classification. If we can transform the dependent variable such that it takes continuous values, we can also validate the normality assumption after that. Suppose our dependent variable is binary, that is, we have a two-class classification problem. The class labels can be written as . We can view this as a probability: when class label is 1, , and for However, probability values are bound to lie in interval , and thus, cannot be assumed to be normally distributed. We know that odds can take any real positive value and are related to probability by: odds in favor of class 1 . Now, or $\log_e \text{Odds}$ or logit can take any real value in . Using this transformation, we have our linear model:

Note: The function is called the sigmoid function, which is S-shaped, and its range is (0,1). We represent the posterior probability of class as a sigmoid of linear combination of dependent variables.

To visualize this, we will consider a single dependent variable example for binary classification by taking a subset of iris dataset (**virginica** and **versicolor**) classes only as target and **petal length** as the only dependent variable to predict the two classes. The following code shows these data preparation steps on the iris data frame we created in the previous chapter:

```
1. df_sample = df[((df['flower']=='virginica')|  
(df['flower']=='versicolor '))][['petal length (cm)',  
'flower']]  
2. #convert to binary labels  
3. df_sample['y']=df_sample['flower'].apply(lambda x: 1.0 if  
x=='virginica' else 0.0)  
4. df_sample.plot.scatter(x="petal length (cm)", y='y',  
marker='*')
```

[Figure 5.9](#) (left) shows the data plot, and [Figure 5.9](#) (right) shows how the y values are scattered about an S-shaped sigmoid curve:

Figure 5.9: Logistic regression

The spread of the y values around the sigmoid is always < 1 . Thus, now we have our normality assumption satisfied, and the constant variance is also satisfied to a great extent with the variance bound we have.

We can now find the coefficients of the linear equation by minimizing the following:

Sum squared error: . However, this is hard to optimize as it is a *non-convex* function. We can also estimate the parameters by our standard MLE techniques. Here, the being binary can be viewed as a Bernoulli distributed random variable.

Hence, the log likelihood takes the following form:

The is called the *binary cross entropy* error or loss function. We will need derivative of sigmoid function:

to maximize .

Setting:

The quantity represents probability that belongs to class 1, and we will denote it by from now. In vector form, we can write the gradient of as follows:

Where \mathbf{X} is the $N \times M$ design matrix, whose n^{th} row is given by \mathbf{x}_n and \mathbf{y}_n represents the vector of probability predictions .

This is not a set of linear equations such that we can solve it by inverting coefficient matrix and get a closed form solution as we got for linear regression. The sigmoid function makes it non-linear equation. However, the function can be minimized by sequential method. Also, the function here is concave, as we shall see shortly, and hence, it has a unique minimum. Starting with a random set of weights denoted by \mathbf{w} , we iteratively update the weight as follows:

Here, α is the learning rate. This has a very similar form as in case of linear regression.

Note: We can check the convexity of $\log L(\mathbf{w})$ by checking whether its Hessian is positive definite. The Hessian can be written as Refer to the further reading section [8], where \mathbf{H} is a diagonal matrix with elements $\frac{\partial^2 \log L}{\partial w_i^2}$, since $\frac{\partial^2 \log L}{\partial w_i \partial w_j} = 0$ for $i \neq j$. As each element of the diagonal matrix is positive, the Hessian is positive definite. The $\log L$ can be minimized by an efficient iterative technique based on the Hessian called Newton-Raphson and the updated formula is .

Multiclass logistic regression

Logistic regression can be extended for multiclass classification by building $K-1$ binary classifiers, each of which separate points in k^{th} class from the points not in that class. This is known as **One-vs-Rest (OvR)** scheme. For this, we can take one class as pivot class, say the largest class label K and model log of odds of being in class k vs being in class K as follows:

Exponentiating both sides:

Since , adding all equations, we get:

An alternative formulation is by assuming multinomial distribution for the class variable. Using Bayes, theorem we can write:

Where . This normalized exponential denoted by , is also known as the *softmax function*, as we can view this as a smoothed ‘max’ function: if , then and .

Comparing with logistic regression where we modelled the log of odds as a linear function, we represent . Here, represents the weights corresponding to j^{th} class.

Now, we can represent class label for each data point as 1-of-K representation or one hot encoded representing a realization of multinomial random variable with K categories. The target vector is a matrix , and the likelihood is given by:

where . The function is called the **categorical-cross-entropy** error or loss function. The derivative (Jacobian) of softmax is given by , where represents th entry of the identity matrix. Using this, we can compute the gradient of the with respect to one of the weight vectors :

Poisson regression

Poisson regression is like logistic regression, except that the dependent variable is an observed count that follows the Poisson distribution. Thus, the possible values of Y are the non-negative integers: 0, 1, 2, 3, and so on. Example application of Poisson regression is study of counts of bacteria related to various environmental conditions and dilutions. Clearly, the dependent variable breaks the normality condition, and hence, we need to apply some transformation. Using log transform, we can model the logarithm of the mean of dependent variable using a linear model.

Suppose we have a sample of n observations , which can be treated as independent Poisson random variables , with , and we can model the log of the mean as a linear function of the dependent variables . Then, we have the following model:

We can use MLE to estimate the parameters . The log likelihood of data is given by:

The gradient of takes similar form as logistic regression:

Also, $\log L$ is a convex function and can be optimized using gradient decent or by other gradient-based methods, like *Newton-Raphson*.

In Poisson distribution, we have So, we are assuming equi-dispersion. That is, the mean and variance are equal: . But in practice, it's very common to see *overdispersion* that is, . The **Negative Binomial (NegBin)** Model can accommodate over- and under-dispersion at the cost of an additional parameter.

Note: In all the four mentioned variations of the linear models linear, logistic, Poisson, Neg-binomial, we can see one common pattern:

The response variable y follows an exponential family (ExpFam) of distributions (for example, binomial, Poisson, multinomial, normal)

A linear model relates the expectation of the response variable via a link function g such and that .

This representation is known as Generalized Linear Model (GLM). The Python statsmodel glm package, mentioned in Further Reading [9], provides implementation of all these variants of linear model.

Interpretability of linear models

Linear models are easy to interpret, which makes them very popular. We can justify why the model works and get deeper insights into hidden patterns in data that can seed thoughts for further improvement of model by either feature engineering or exploring newer data sources. We will start with the simplest linear model with identity basis function as the line: Here, represents the predicted value of target variable, and represents the true value of the target. The prediction error is . Here, denote the MLE

estimates of the coefficients. Clearly, , are all random variables being function of the target random variable y . By model assumption, , and hence, . Substituting , in the MLE estimate , we get:

The coefficient can be computed as earlier:

Now:

The coefficient represents the slope of the line, that is, by what factor the target y changes for a unit change in the value of predictor.

Note: This interpretation as slope can be extended for multiple predictors as well. For more than one predictor, we can view the weights as a factor by which the target will change for a unit change in the predictor, holding all other predictors constant. This interpretation of the coefficients can help us understand which predictors affect the target and how. However, this relation should not be considered a cause-and-effect relationship as it indicates correlations, and correlations do not imply causation.

These coefficients are random estimates; there is a chance that they are not accurate and may mislead us. To safeguard against such risk, we can employ hypothesis testing (or test of significance of the coefficient). We test the hypothesis and check whether the coefficient . Formally, we write it like this in terms of null and alternate hypothesis:

against the alternative hypothesis

or or

Since are normally distributed, is normally distributed with mean 0 if is true, and thus, is t-distributed with degrees of freedom associated with the sample variance of , that is, computed earlier.

Following the standard steps for hypothesis testing, we first choose the level of significance and perform a two tailed t-test for testing the null hypothesis against . The critical value of Student's t for the two-tailed

alternative hypothesis places probability $\alpha/2$ in each tail of the distribution. The probability of falling in the critical region is called *p-value*. *Small p-value indicates, we cannot reject the null hypothesis, and hence, the coefficient is not significant.* We can also compute the confidence interval of the coefficients. This is more informative as they reflect the precision of the estimates. Testing against the alternate hypothesis the 95% confidence intervals for and are given by: and , respectively.

We can use the Python stats model for computing these confidence intervals and test the significance of each of the coefficients, as shown in the following code. Here, we have used the petal width vs petal length, as shown in [*Figure 5.9 \(Left\)*](#).

```
1. import statsmodels.formula.api as smf  
2. model = smf.ols(formula="petal_length ~ petal_width",  
data=df)  
3. results = model.fit()  
4. print(results.summary())
```

Refer to the following figure:

Figure 5.10: Regression Results

This statistical analysis of coefficients done so far for regression with a single variable can be easily extended to multiple variables using matrix algebra and quadratic forms.

Interested readers may refer to [*Chapter 4, Analysis of Variance*](#), from the book [7].

In the result in [*Figure 5.10*](#), we see scores called and F-Statistic. These help us access the overall goodness of model fit. Let's see what they mean. It can be easily proved that (see *Further reading [6]*):

Total **Sum of Squares (SST)** = **Sum of Squared Due to Regression (SSR)** + **Sum of Squares of Errors (SSE)**. In this proof, we use the normality of errors assumption we had for linear regression that gives the mean of error terms . Here, SST is the total variance of the target variable irrespective of the model. The part of that variance is explained by the model SSR and part is unexplained by model that is the squared error SSE. Dividing the

equation by SST , we have $1 = \frac{SSE}{SST}$. We define R^2 as the *coefficient of determination*, which indicated the proportion of the total variance that is explained by the model. Clearly, R^2 values between 0 and 1. Greater values of R^2 indicated better model fit.

Note: This equation $SST=SSR+SSE$ is perfectly valid for even more than one predictor variables. But as we add more predictors to the model, we will see the R^2 value monotonically increasing. This does not always mean we are getting a better fit with more predictors as every predictor may not have an impact on the target. Hence, we use adjusted R^2 as the number of predictors increase.

The adjusted R^2 tells you the percentage of variation explained by only the **independent variables** that actually affect the dependent variable.

The significance of the coefficients can be measured for other variations of the linear models, like logistic regression.

Tip: For logistic regression model, this R^2 statistic does not make sense, as its based-on ratio of variances explained. McFadden's pseudo- R^2 is defined as $R^2_{pseudo} = 1 - \frac{\ln L}{\ln L_{null}}$. Here, L denoted the maximized likelihood and L_{null} denotes the intercept only model. Intuitively, we can understand this measure as follows. If the model has no predictive ability, the likelihood value for the model will not be much greater than the null model likelihood. Therefore, the ratio of the two log-likelihoods will be close to 1, and R^2_{pseudo} will be close to zero.

Conclusion

In this chapter, we discussed the fundamentals of statistical inference. We covered sample statistic, hypothesis testing, parameter estimation techniques and then various applications of these in ML. We discussed various ML models in a probabilistic setting, and we understood that the error functions we minimize to train these ML models are derived from these probabilistic settings. We also introduced fixed basis function model, which gives a generic structure to all the regression, classification models.

In the next chapter, this structure will be extended to neural networks. The same error functions derived in this chapter will be used here, after several applications of models, based on deep neural networks.

Points to remember

- A statistic T is a function of samples from a population and is generally used to estimate a population parameter from the sample values. We can view a prediction model as a statistic and an estimator of the true population behavior. The training data can be viewed as a sample that can be used to estimate the true population behavior.
- **Bias-variance decomposition:** The **Mean-Squared Error (MSE)** of T in estimating parameter can be decomposed as:
- **Bias-variance tradeoff:** High bias model indicated our model is oversimplified and is underfitting and thus prediction from these models have high variance. Similarly, low bias implies overfitting and also prediction from this model will have low variance.
- If MVU exists for a statistic, then the MLE procedure will give that estimator.
- MLE estimates are consistent and efficient, but need not be unbiased.
- MLE estimates are prone to overfitting, and this can be mitigated by Bayesian estimation with MAP or with regularization techniques.
- Linear models discussed here should not be visualized only as lines or planes. Remember, linear means linear coefficients, and by using non-linear basis functions like polynomial or radial basis functions, we can represent very complex multivariable non-linear functions (*fixed basis function models*).
- The probabilistic view of linear, logistic and Poisson regression helps us reduce the classification and regression problem as a convex optimization problem that can be solved by iterative gradient-based optimization methods.
- The interpretability of linear models makes them more useful for solving business problems. Testing of the statistical hypothesis for whether the coefficient is actually zero helps analyze the significance of the coefficients. Lower p-value indicates low chances of rejecting

the hypothesis that the coefficient is zero, and hence, the corresponding feature must be an important feature.

Further Reading

- *Fundamentals of mathematical statistics*, S C Gupta, V K Kapoor
Publisher: New Delhi: Sultan Chand & Sons.
- *Chris Bishop, Pattern Recognition and Machine Learning*, [Chapter 3](#) and [4](#).
- Unbiased Estimator: https://dawenl.github.io/files/mle_biased.pdf.
- Student's t-distribution: https://en.wikipedia.org/wiki/Student%27s_t-distribution.
- MLE for multivariate Gaussian: <https://people.eecs.berkeley.edu/~jordan/courses/260-spring10/other-readings/chapter13.pdf>.
- https://web.njit.edu/~wguo/Math644_2012/Math644_Chapter%20_1_part4.pdf
- *Applied Regression Analysis: A Research Tool* by Rawlings, John O., Pantula, Sastry G., Dickey, David A
- Hessian of log likelihood of logistic regression is positive definite: <https://www.cs.mcgill.ca/~dprecup/courses/ML/Lectures/ml-lecture05.pdf>
- <https://www.statsmodels.org/stable/glm.html>

CHAPTER 6

Neural Networks

In the previous chapter, we discussed that with a clever choice of basis functions and using linear models alone, we can solve a wide range of ML problems. So, it appears that linear basis function models constitute a general-purpose framework for solving ML problems. However, there are certain limitations:

1. These non-linear basis functions need to be defined before training, that is, we must carefully perform feature engineering, which is a time-consuming, manual effort and demands sound knowledge of the domain.
2. The number of basis functions grow rapidly, often exponentially with data dimension, that is, the curse of dimensionality (discussed in Overview of AI chapter) problem arises. Hence, we need to look for alternative models.

Neural network models are inspired by the way biological neural systems in human brain processes information. Neural networks solve these problems by choosing fixed but adaptive basis functions. These are parametrized non-linear basis functions whose parameters can be learned from the data during training of the neural network model. This is called the training of the network. Also, we can create a hierarchy of these parametrized basis functions. Each level in the hierarchy is called a layer of the network. A layer consists of fixed number of basis functions, which takes the output of the previous layer as input. The arrangement of these layers and connections among them in a particular form is called the *architecture* or *topology of the neural network*. We will see how the knowledge of the problem domain can be easily incorporated in the neural network architecture through choices in number of layers, units per layer, connections between layers, and so on. Now, the challenge of careful feature engineering is transformed to network architectural engineering. For training these networks, the most popular method is gradient descent – the back-propagation algorithm. However, for

training deep networks, that is, networks with a large number of layers, simple gradient descent may not give us the best solution, so many modifications of that algorithm are suggested, which we will discuss in this chapter.

Structure

In this chapter, we will cover the following topics:

- **Single neuron:** An adaptive basis function
- Multiple stacked layers or hierarchy of neurons
- **Training hierarchy of neurons:** Back propagation algorithm
- **Basic neural network architectures:** DNN, CNN, RNN, Transformers, Autoencoders

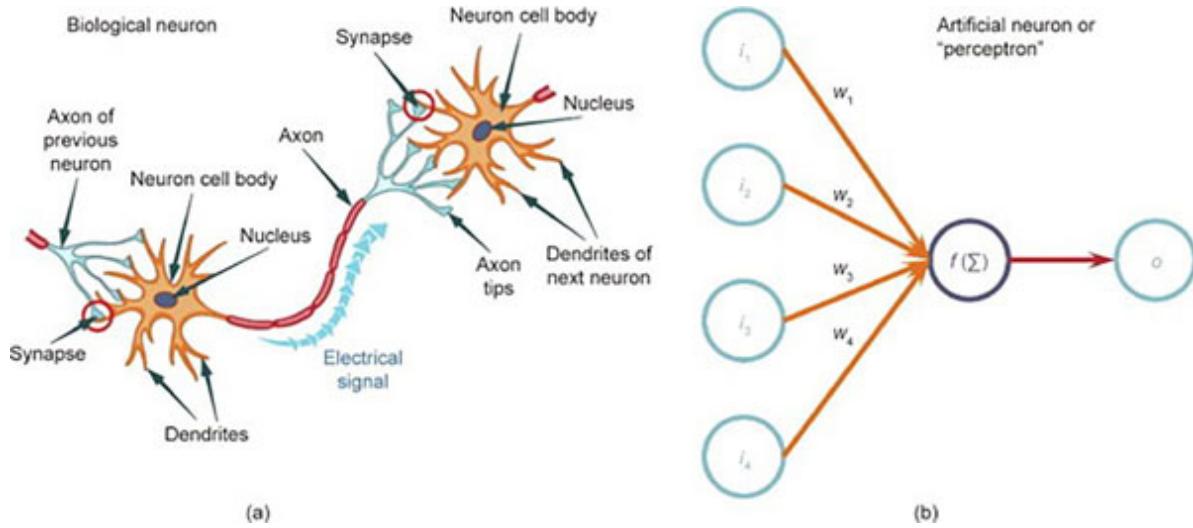
Objectives

After studying this chapter, you will be familiar with the fundamental concepts behind deep neural networks and state-of-the-art AI models. These concepts will be applied in the subsequent chapters, where specific deep neural network models will be discussed for solving AI problems like speech recognition, handwriting recognition, language translation, image classification and generation.

Artificial neuron: An adaptive basis function

The fundamental building block of a neural network is called a *neuron*. This term is borrowed from biology, where a neuron is a nerve cell that is the basic building block of the nervous system. A single neuron may be connected to many other neurons such that the information transmitted is consumed by these connected neurons. The biological neuron consists of three main parts: *dendrites* (the receivers), the cell body, and *axon* (the transmitter). Neurons communicate with one another at junctions called *synapses*, where one neuron sends a message to a target neuron, that is, another neuron cell. These are chemical messengers or ions. An artificial neuron closely mimics this structure comprising of a set of dendrite-like connections, each taking an input and multiplying it by a (synaptic) weight (weight indicates strength of synaptic connection) associated with that edge.

These weights are learnt in the learning phase. These weighted inputs are summed up after going through a *summation unit*. The result is subsequently fed to an *activation unit* whose output is then transmitted to the outside via an axon-like projection. The structure of a single biological and artificial neuron is depicted in [Figure 6.1](#):



[Figure 6.1: Biological neuron and artificial neuron](#)

Given an input vector $\mathbf{x} = (x_1, x_2, \dots, x_n) \in R^n$, the computation in an artificial neuron can be written in the form of a basis function $\phi(\mathbf{x})$, as follows:

$$\phi(\mathbf{x}) = f\left(\sum_{i=1}^n w_i x_i\right) = f(\mathbf{w}^T \mathbf{x})$$

Here, $f: R \rightarrow S \subseteq R$ is a non-linear activation function. [Table 6.1](#) shows some common activation functions. Also refer to the following figure:

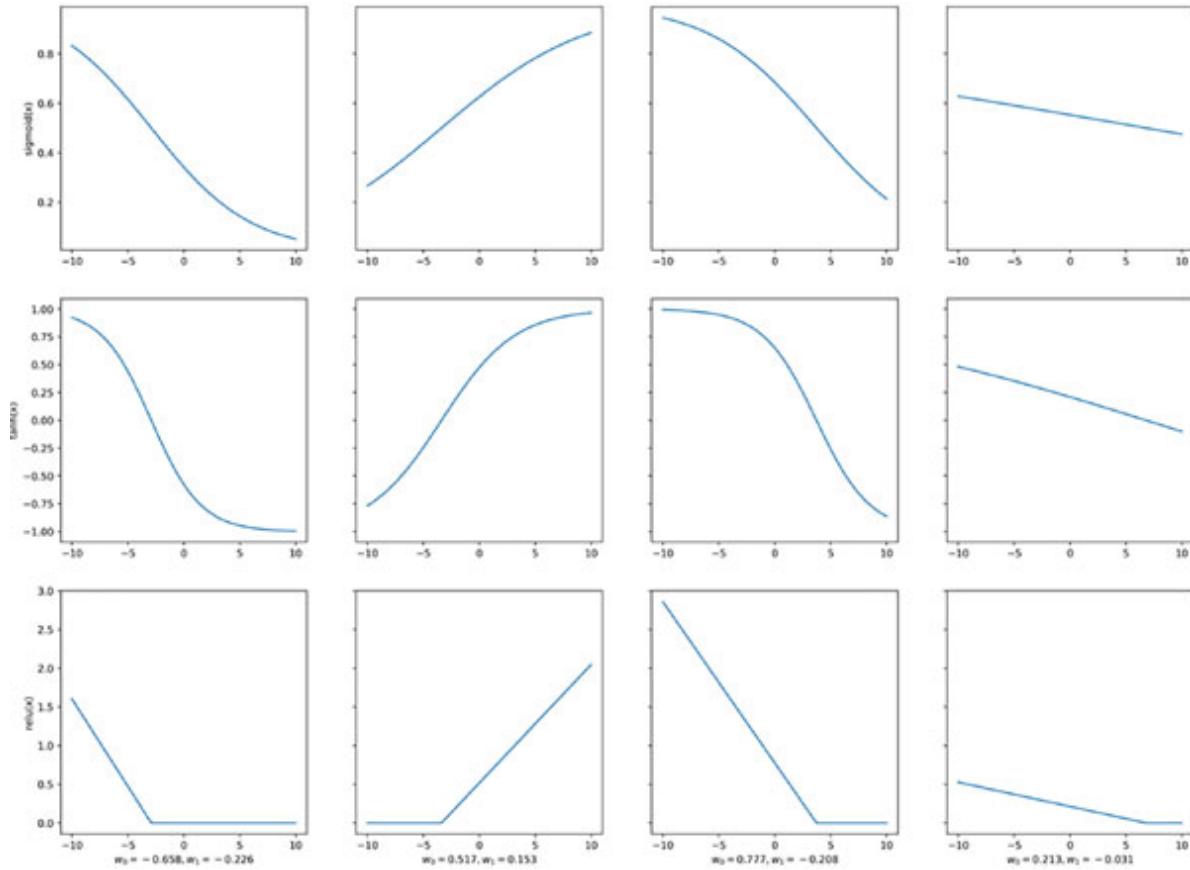


Figure 6.2: Different basis functions created by various activations: first row shows various sigmoid activations, second row shows various tanh activations, and the last row shows relu activations

Function Name (f)	Range (S)	Derivative $f'(z) = \frac{df(z)}{dz}$
$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$[-1, 1]$	$1 - (\tanh(z))^2$
$\text{sigmoid}(z) = \frac{1}{1 + e^{-z}}$	$[0, 1]$	$\text{sigmoid}(z)(1 - \text{sigmoid}(z))$
$\text{ReLU}(z) = \max(0, z)$	$[0, \infty)$	$\begin{cases} 0 & \text{if } z \leq 0 \\ z & \text{if } z > 0 \end{cases}$
$\text{exponential}(z) = e^z$	$[0, \infty)$	$\text{exponential}(z)$
$\text{LeakyReLU}(z) = \begin{cases} z & \text{if } z > 0 \\ cz & \text{if } z \leq 0; c \text{ is constant} \end{cases}$		$\begin{cases} c & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$
$\text{identity}(z) = z$	$(-\infty, +\infty)$	1

Table 6.1: Activation functions

For a fixed set of weights \mathbf{w} , this basis function $\phi(x)$ is a fixed function, like the one we have in linear basis function model, studied in the previous chapter. Now, let's see how we can get a family of basis functions by varying these weights. For example, we take a single variable input $x \in \mathbb{R}$ and make an adaptive basis function with two weights $w_0, w_1 : \phi(x) = f(w_0 + w_1x)$. [Figure 6.2](#) shows how setting the weight parameters randomly gives each basis function a distinct shape.

Feed Forward neural network

The fixed basis functions ([Figure 5.5](#) from [Chapter 5: Statistical Inference and Applications](#)) can be replaced with these adaptive basis functions in the linear basis function model discussed for regression or classification (logistic regression), and the resulting model is called *feedforward neural network* (shown in [Figure 6.3](#)):

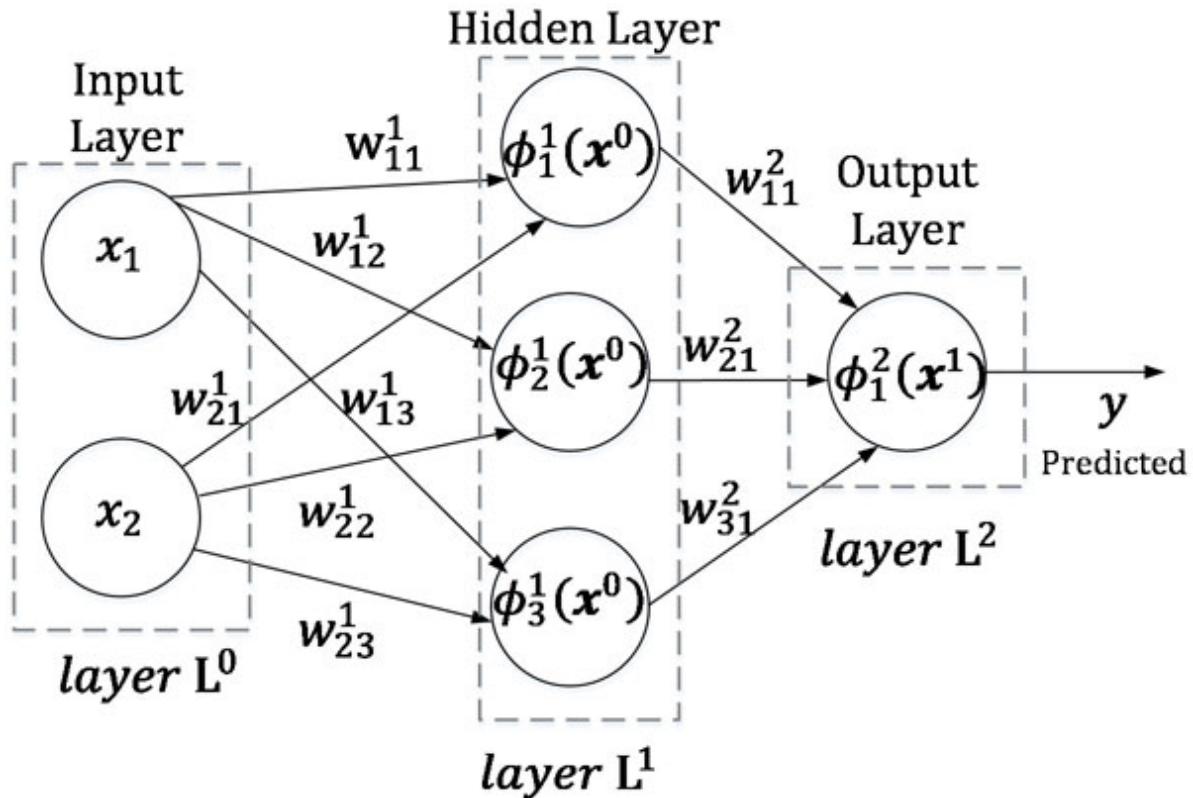


Figure 6.3: Neural network with single output neuron and one hidden layer; the weights connecting i^{th} neuron of layer j^{th} to neuron of layer $(l+1)$ is denoted by $w_{ij}^{(l+1)}$

The set of three adaptive basis functions transforming the input vector $x \in \mathbb{R}^n$ to $\phi(x) = (\phi_1(x), \phi_2(x), \phi_3(x)) \in \mathbb{R}^3$ is called a *layer* in the neural network. The input vector is called *input layer*, assuming it's a basis function layer with identity basis function, and the final output neuron(s) is called *output layer*. The layers between the input and output layers are called *hidden layers*. We can have more than one hidden layer. Numbering the layers of the network starting from the input layer numbered (0), the hidden layer numbered (1), and the output layer numbered (2) in [Figure 6.3](#). The weights connecting i^{th} neuron of layer (l) to j^{th} neuron of layer ($l + 1$) is denoted by $w_{ij}^{(l+1)}$. The set of all weights connecting layer (l) to layer ($l + 1$) is denoted by matrix $W^{(l+1)} = (w_{ij}^{l+1})$.

We can view these layers as set of *vector fields* as $L^l: \mathbb{R}^r \rightarrow \mathbb{R}^s$, where r denotes number of nodes in layer $l - 1$, s denotes number of nodes in layer l , and L^l transforms the vector $\in \mathbb{R}^r$ to a vector $\in \mathbb{R}^s$. The entire neural network in [Figure 6.3](#) can now be expressed as a composition of a set of vector fields:

$$x = L^0(x) \rightarrow L^1(L^0(x); W^{(1)}) \rightarrow L^2(L^1(L^0(x); W^{(1)}); W^{(2)}) = \hat{y}$$

So, $\hat{y} = L^2 \circ L^1 \circ L^0(x)$, where \circ denotes function composition

Here, L^0 , L^1 , and L^2 represent the input, hidden, and output layers, respectively. We have seen in [Chapter 3:Vector Calculus](#) in section chain rule for derivatives of vector fields, how to differentiate such composition of functions. We will be using this in the following section.

Initializing all the weights and holding the weights connecting layer (0) to (1): $W^{(1)}$ as fixed, we have a fixed basis function network as before. The fixed basis function model is also a neural network with a single layer.

The number of hidden layers can be many based on the complexity of the problem we want to solve. Also, the number of neurons per layer may vary. [Figure 6.4](#) depicts a generic multi-layered feed forward network with a single output node. Each layer's adaptive non-linear basis functions are denoted by $\phi^l(x^{l-1}) = (\phi_1^l(x^{l-1}), \phi_2^l(x^{l-1}), \dots, \phi_{n^l}^l(x^{l-1}))$. Refer to the following figure:

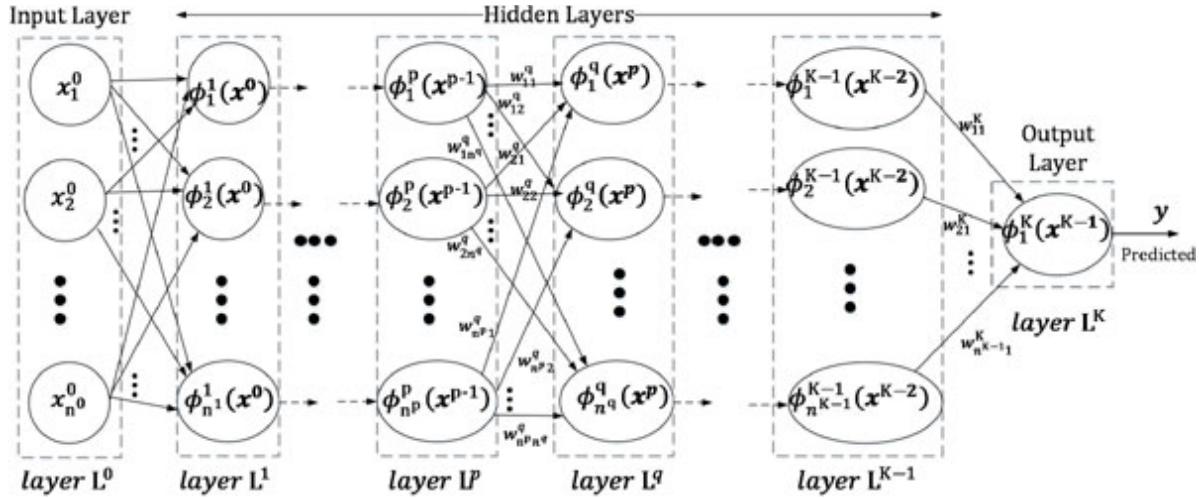


Figure 6.4: Neural network with single output neuron and many hidden layers. Here, $p & q$ represents layers where $q = p + 1$, n^p denotes number of nodes in layer p , neural network has total of $K+1$ layers which includes one input layer, one output layer and $K-1$ hidden layers, and x^l denotes l^{th} layer output vector/tensor. Here, only one output node is shown, but in general, there can be many output nodes.

Training neural network

For finding the weights in a fixed basis function network, we could use linear algebra and arrive at a closed form solution analytically. Moreover, the error functions E we arrived at, like sum-squared error or cross entropy error for classification, were convex functions with unique optimum that can be arrived at by gradient based iterative optimization methods as well.

With adaptive basis functions, the error function E is now a function of the weights associated with the basis functions (in hidden layers also). This is because the predicted output \hat{y} is now a function of hidden layer weights. The error E being some function of target y and prediction \hat{y} is also a function of hidden layer weights. We want to find a set of weights w that minimizes the chosen error function $E(w)$. Here, w represents all the weights of the network. This task is called *training* of the neural network. [Figure 6.4](#) depicts geometrically an error function for two-dimensional weight space. For any value of the weight vector $w = (w_1, w_2) \in R^2$, we can plot the error value in the z-axis and view the error function as a surface over the weight space.

The goal is to find a vector w such that $E(w)$ takes its minimum value. However, the error surface is typically non-convex because of having highly non-linear dependence on the weights and bias parameters, so there can be

many local minima points in weight space at which the gradient almost vanishes.

In [Figure 6.5](#), w_A is a local minimum. For training a neural network, it may not be always necessary to find the global minimum, but what really matters is to find a sufficiently good solution. Even if a global minima is actually found during training, we may not know it. We have already discussed how gradient descent algorithm uses the derivatives of a function to find a minimum in [Chapter 3: Vector Calculus](#), section: descent methods . However, gradient descent, in general, has often been regarded as slow or unreliable. Previously, the application of gradient descent to non-convex optimization problems was regarded as foolhardy or unprincipled. Today, nearly all neural networks are trained by one very important algorithm: **Stochastic Gradient Descent (SGD)** which is an extension of the gradient descent algorithm. Refer to the following figure:

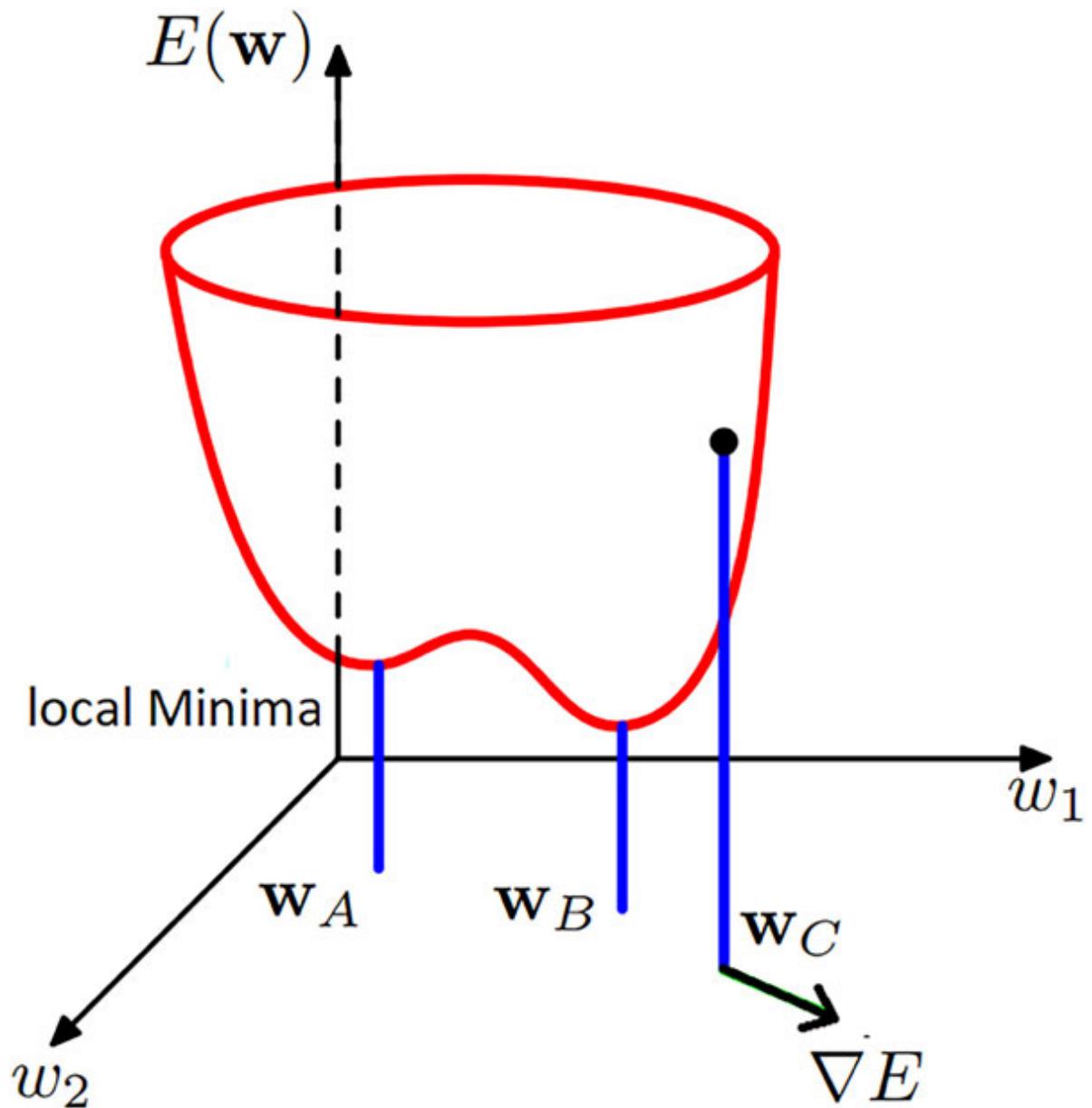


Figure 6.5: Geometrical view of the error function $E(\mathbf{w})$ for two-dimensional weight space, that is, $\mathbf{w} \in \mathbb{R}^2$ as a surface sitting on weight space. Point \mathbf{C} is a local minimum and \mathbf{w}_A is the global minimum. At any point \mathbf{w}_B , the local gradient of the error surface is given by the vector ∇E .

Stochastic Gradient Descent

We have seen in the [Chapter 3: Vector Calculus](#) the formulation of the error function for regression/classification problems. These functions being some form of log-likelihood, decompose as a sum over N training examples of some per-example loss function:

$$E = E(w) = \frac{1}{N} \sum_{n=1}^N E_n(w)$$

In the gradient decent algorithm, we initialize the weights randomly and keep updating the weights based on gradient information. Hence, for these additive loss functions, we need to compute the gradient value at each training example. As the training set size grows to billions of examples, the time to take a single gradient step becomes prohibitively long. The intuition behind SGD is that the gradient is an expectation. The expectation may be approximately estimated using a small set of samples. Specifically, on each step of the algorithm, we can sample a mini batch of examples drawn uniformly from the training set. For simplicity, assume that the mini batch size is 1. Now, at every step t , a random sample point $n \sim Uniform(1, N)$ is chosen, and we update the weights via the following:

$$w^{(t+1)} = w^{(t)} - \eta \nabla E_n$$

The expectation of the gradient is given by the following:

$$E_{n \sim U}[\nabla E_n] = \nabla E_{n \sim U}[E_n].$$

We can interchange the gradient and expectation if E_n is a smooth or continuously differentiable function. For a neural network, this assumption is valid as all the layer functions are continuously differentiable given that we are using differentiable nonlinearities in each layer. Now, the probability of choosing n is $P(n) = \frac{1}{N}$. Thus, from the expectation definition, we have:

$$E_{n \sim U}[E_n] = \sum_{n=1}^N P(n) E_n(w) = \sum_{n=1}^N \frac{1}{N} E_n(w) = \frac{1}{N} \sum_{n=1}^N E_n(w) = E(w)$$

$$\text{Hence, } E_{n \sim U}[\nabla E_n] = \nabla E_{n \sim U}[E_n] = \nabla E(w).$$

Let's try to apply SGD for training the two-layer neural network described in [Figure 6.3](#) where the predicted output is a scalar \hat{y} .

Computing error derivatives

We want to compute the gradient , where is the weight tensor representing all N layer weights of the networks. Any error function E depends on the predicted output (and given target y that is constant), and thus using *chain rule of derivatives*, we can write:

As , we can write as . The partial derivative of the error E w.r.t any layer output tensor is denoted by . There is a recursive relation between and because of chain rule of derivatives (using matrix form of chain rule for vector fields as E depends on the composition of the layers) that is:

where are Jacobian matrices

We can write and a is the activation function of L^{l+1} . Therefore:

Like any other recursion relation, the base case must be defined from where the recursion begins. Here, the recursion begins at the last layer. We can think of the error function E as a layer , and thus, define the base:

Now, coming back to the error derivative, we can represent them in terms of :

w.r.t. the weight vector of the corresponding layer.

Example: If is a sigmoid layer with n output neurons and m inputs, we can write the layer function as . Here, x is the layer input vector.

So, will be given by the following product of Jacobians:

Here, represents the derivative of a first order tensor with a second order tensor. This derivative will be a third order tensor, as discussed in [Chapter 3: Vector Calculus](#) in section tensor calculus. We can represent this with tensor outer product operation and Einstein summation notation, as follows:

The derivative of sigmoid function $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. Therefore:

$$\therefore \frac{\partial L^l}{\partial W^{(l)}} = (\sigma(y_i)(1 - \sigma(y_i))e_j \otimes e_j) \cdot (y_k e_j \otimes e_j \otimes e_k)$$

Using the property of outer product: , the previous dot product simplifies to the following:

Note: The derivative of error function E w.r.t the corresponding layer weights can be written as a product of gradient message coming from the next layer, that is, and the derivative of the activation function of the layer w.r.t the weight vector . Here, is a layer specific Jacobian and does not depend on any other layers. Thus, depends on derivative messages from the layers above and not from the layers below.

For computing the error derivatives with SGD, we need to compute the derivatives of the error function over subsets (or mini batch) of training examples and then compute the average error over the mini batch. In the previous computation, we computed the derivative for single example. A mini batch can be represented as a matrix whose each row represents one training example of the batch. So, x is now a second order tensor, and the corresponding layer output will also be also a second order tensor.

Therefore, the partial derivatives will change as follows:

Now:

Hence:

Using the property of outer product: , the previous dot product simplifies to the following:

We know that the dot product as it's the Frobenius norm of the matrix with only the element as 1 and all other elements as 0:

Let's implement the same, and then we can compare it with the derivative computed by any deep learning library. In the following code, we will first implement sigmoid layer using TensorFlow:

```
1. import tensorflow as tf
2. batch_size =2
3. tf.random.set_seed(42)
4. #create a Sigmoid layer with 4 output neurons
5. layer = tf.keras.layers.Dense(4, activation=tf.nn.sigmoid)
6. #create a input tensor 3 neurons and batch size = batch_size
7. x = tf.random.normal([batch_size, 3])
8. z = layer(x)
```

The output tensor `z` is of shape (2, 4). Using the previous formula for the layer derivative, let's compute the derivative of layer w.r.t the layer weights, which can be accessed by the `layer.kernel` property. Layer weights are initialized at random. The following code depicts this:

```

15.     v = tf.tensordot( eb,u, axes=0)
16.     if jacobian is None:
17.         jacobian = tf.tensordot(v, tmp, axes=0)
18.     else:
19.         jacobian += tf.tensordot(v, tmp, axes=0)
20. print(jacobian)

```

The computation is just one step using TensorFlow gradient tape, as shown here:

```

1. with tf.GradientTape(persistent=True) as tape:
2.     z = layer(x)
3. jacobian = tape.jacobian(z, layer.kernel)
4. print(jacobian)

```

[Figure 6.6](#) shows the output obtained from either of the two implementations of layer derivative, that is, layer output w.r.t the layer weight :

Figure 6.6: Layer derivative tensor of shape (2,4,3,4)

Note: The layer kernel matrix is of shape (3, 4). The shape of the derivative tensor or Jacobian of the output with respect to the kernel is those two shapes concatenated together.

We have shown how to compute the derivatives of each layer and a way to recursively compute the error for the entire network. We will now apply this technique for computing error derivatives for specific error functions, like sum-squared error and binary cross entropy.

[Backpropagation algorithm](#)

Let's take the three-layer neural network that we introduced before. In the linear regression section of the previous chapter, we learned that E is mean-squared error given by . Here, \hat{y} is the previous layer (L^1) output. Also, the error function will be computed for a mini batch of samples of size . So, we can write E in terms of layer function as follows:

- and thus,

-
-

This is the same gradient we obtained for linear regression, the only difference being that was a fixed function there.

Next, let's compute .

- Since *is a linear layer*, here is a weight matrix connecting n neurons in layer to m neurons in layer ; therefore,
-
- $\frac{\partial E}{\partial w^{(1)}} = \delta^{(1)} \frac{\partial L^1}{\partial w^{(1)}}$

Assuming that *tanh* activation is used in the layer

This expression can be derived in the same way we derived the derivative for sigmoid layer, only replacing the sigmoid derivative with the tanh derivative.

We now have all the required gradients to compute the gradient step needed for gradient decent algorithm: .

Now, we will implement these using TensorFlow. First, we create the model using functional API, as shown in the following code:

```

1. inp_vector = tf.keras.layers.Input(shape=(3,), name='input')
2. x = tf.keras.layers.Dense(units=4, activation='tanh',
3.                             name='hidden', use_bias=False)
   (inp_vector)
4. x = tf.keras.layers.Dense(units=1, activation='linear',
5.                             name='output', use_bias=False)(x)
6. model = tf.keras.Model(inputs =[inp_vector], outputs=x)
7. print(model.summary())

```

Now, using gradient tape as before, we can directly compute the gradient of the loss function w.r.t both the weight tensors (**model.trainable_variables**) in the two layers. This is shown in the following code:

```

1. batch_size=5
2. X = tf.random.uniform([batch_size, 3])
3. Y = tf.random.uniform([batch_size, 1])

```

```

4.
5. def mse(y_true, y_pred):
6.     return 1/2*tf.reduce_mean(tf.square(y_true-y_pred))
7.
8. with tf.GradientTape(persistent=True) as tape:
9.     y_pred = model(X)
10.    loss = mse(Y, y_pred)
11. gradient = tape.gradient(loss, model.trainable_variables)

```

Now, let's validate that we get the same result by following the steps discussed earlier. [Table 6.2](#) shows the step-by-step computation of error derivatives for all layers:

Table 6.2: Error derivative computation for all layers (back propagating error derivative)

This is called **backpropagation algorithm**. The name is given as we pass the messages backward.

Note: The Einstein sums used in the previous table represent tensor dot products between Jacobians computed in various backward gradient computation steps. For computing $\frac{\partial L^1}{\partial w^{(1)}}$, we have not used the tensor outer product formula that we derived; we used the TensorFlow gradient tape API to keep the code shorter. We have already shown how to implement the tensor dot product formulation for sigmoid activation, and this can be implemented by following the same steps for tanh activation.

We have discussed and implemented the algorithm for only three-layer network so far, but this can be easily extended to any number of layers. We can train very deep neural networks with this algorithm. For any pair of non-linear hidden layers let's look at the steps for computing . Let there be K layers in the network:

1. Choose a random sample (or mini-batch) of training examples.
2. Perform forward computation through the network and compute error E.
3. Compute the δ for the output layer, .
4. Compute the error derivative w.r.t layer weights .

5. Go back one layer .
6. Use recursive relation to compute .
7. Repeat steps 4 to 6 until the input layer is reached.

The deep learning frameworks like TensorFlow uses a computation graph to remember the variable dependencies in the network and thus the order of forward computation. The same order is traversed backward to compute the derivative. This process is also known as *automatic differentiation*.

Challenges of training neural networks

Back propagation algorithm can be used to train any deep neural network with a large number of hidden layers and many nodes per layer. However, in practice, there are many challenges we need to address first. Following are a few lists of challenges and how they can be mitigated for successful training.

Slow training with SGD

As more layers are added to the network most of the time, we can find that the learning is very slow by observing the rate of decrease of the loss function as the training progresses. SGD is best suited for convex optimization problems, where we have a convex error surface and unique global minima. In these problems, taking small steps toward the negative gradient direction makes sure we reach the global minima. In the previous sections, we have discussed the complexity of the error surface for deep non-linear neural networks. Along with that, there are a few other challenges related to the complexity of the error surface:

- **Ill-conditioning:** The *condition number* of a matrix is the ratio of the largest singular value to the smallest singular value. An ill-conditioned matrix is one with high condition number. This indicates that a few rows of the matrix are heavily correlated with each other. If we have an ill-conditioned data set, then the error surface defined by that is relatively flat in one or more directions and strongly curved in other directions. This leads to very slow convergence of SGD. This can be identified by monitoring the square of the gradient norm.
- **Cliffs and exploding gradients:** For very deep neural networks with highly non-linear activations, the error surfaces may consist of extremely steep, regions resembling cliffs, as shown in [Figure 6.7](#).

Moving in the direction of a negative gradient by the SGD algorithm can move the weights far off by taking a big jump off the cliff. This will mislead the algorithm and will go away from the minima at a point when we are very close to reaching it. One way to mitigate this risk is by clipping the norm of the gradient, that is, there is a maximum allowed value of the gradient norm. Gradients are restricted from blowing up by rescaling them as if . [Figure 6.7](#) depicts this for a hypothetical two-dimensional error surface with a cliff close to minima:

Figure 6.7: Cliff in error surface and gradient clipping in SGD

- **Weight initialization:** To begin the training, the layer weights must be initialized. This choice of initial points impacts the convergence of the algorithm. Also, various layers in the network must have different initializations to break symmetry, otherwise all layers will get the same gradient update and end up learning same function. In general, biases are initialized with 0 and weights are initialized with random numbers. These random numbers should not be very large because that will cause very high value to be passed to the activation functions like sigmoid, which take a value 1 and have very low gradient for higher input values. So, the learning becomes very low. The weights initialized with low values get mapped to 0 by activation functions and face similar flat gradient issue as earlier. There are various heuristics to initialize the weights like *Xavier Initialization*, *He Initialization*. You are advised to refer to *Further Readings [1], [2]*.

Modifications of SGD

Error functions have regions of high curvature and small but consistent gradients. This is due to the ill conditioning of the Hessian matrix and variance in the SGD, and the learning may slow down a lot in such regions. Following are two categories of techniques for improving SGD.

Momentum methods

The momentum algorithm accumulates the **Exponentially Weighted Moving Average (EWMA)** of previous gradients and makes a move in that

direction instead of the local gradient direction suggested by SGD. The exponential weighting is controlled by parameter $\alpha \in [0,1)$ for exponential weighting, that is, how quickly the effect of the previous gradient decays. The momentum method damps the oscillations in directions of high curvature by combining gradients of opposite signs.

Adaptive learning rate

We saw that the same learning rate is applied to all parameter updates for SGD and momentum methods. Adaptive gradient descent algorithms, such as AdaGrad, AdaDelta, RMSprop and Adam, provide alternatives to classical SGD by keeping per parameter learning rates:

- **AdaGrad:** It adapts the learning rate for each connection by scaling them inversely proportional to the square root of all previous gradients' sum-squared values. Thus, larger gradient changes are made in the gently sloped direction of the error surface. This may lead to shrinking of some learning rates drastically.
- **RMSProp (Root Mean Squared Propagation):** RMSProp modifies the AdaGrad algorithm by taking the EWMA of previous squared gradients. The moving average parameter controls the length and scale of the moving average. This is one of the most successful algorithms for deep neural network training.
- **Adam: Adaptive Moments (Adam)** takes the best of both momentum-based and adaptive-learning-rate algorithms and combines them. Here, the momentum algorithm is applied to rescaled gradients computed by RMSprop.

Bias-variance trade-off in neural networks

The bias variance tradeoff that we studied for general ML models also exists for neural networks. During iterative training of the model, the validation error is slightly more than the training error. If the gap between the test error and the validation error increases over iterations, it's a case of *overfitting*, that is, high bias and low variance. If the training error stops decreasing after a few iterations, we can conclude that the model is *underfitting*, that is, high variance and low bias.

Underfitting can be mitigated by increasing the *model capacity*, that is, the number of layers or the nodes per layers or the activation functions used in a layer (these are the hyperparameters defining network structure). Overfitting is handled by various regularization techniques.

[Figure 6.8](#) shows this trade-off as a function of model capacity for neural networks and compares neural network complexity in terms of layers and number of neurons with the classical ML models:

Figure 6.8: Overfitting and underfitting in neural networks

Generally, a validation data set is used to compute the prediction error of the models. This can help us choose the best model capacity or network structure related hyper parameters for the given problem.

[Regularization of neural nets](#)

Various strategies are developed to avoid overfitting and reduce generalization errors while training neural networks. These strategies are collectively known as **regularization**. Following are a few popular techniques for regularization:

- **Weight-decay:** In the previous chapter, we discussed that one effect of overfitting is directly related to explosion of the weights of the model, and this can be mitigated by adding weight penalty terms like l^1 , l^2 weight constraints.
- **Dropout:** The output of a fraction of nodes from a layer chosen randomly are masked by setting their output to zero during the training. It's equivalent to removing a fraction of nodes from a layer and creating a new neural network with fewer nodes. This can be compared to model-averaging method (ensemble learning), where many models are created by changing the number of active nodes at various layers of the base model on which dropout is applied. This is a computationally inexpensive but powerful method of regularizing deep neural networks.
[Figure 6.9](#) shows that the third node in the hidden layer is masked. All of its output connection weights are set to zero. Refer to the following figure:

Figure 6.9: Dropout: Showing the dropped node while training

- **Weight sharing:** Using the same set of weights in different layers in the network, we have fewer parameters to optimize. RNN (discussed in the [Chapter 10: Sequence to Sequence Models](#)) and CNN (discussed in [Chapter 9: Computer Vision](#)) use weight sharing.
- **Batch normalization:** Standard scaling of inputs has shown improvements in the model performance. Batch normalization applies the same trick to the hidden layers. It normalizes the previous layer's activations by subtracting the mini-batch mean, μ , of activations and dividing by the mini-batch standard deviation, σ . During inferencing, μ and σ are replaced by an average over all the values collected during training.

Sensitivity of neural networks to small perturbations

Deep neural networks are often found to be very sensitive to small well-chosen perturbations. A well-chosen small perturbation of an input image can mislead a neural network, resulting in significant decrease in its classification accuracy. One metric to assess the robustness of neural networks to small perturbations is the Lipschitz constant. A vector valued function is called *Lipschitz continuous* if there exists a constant L such that for all x, y in the domain D , $|f(x) - f(y)| \leq L|x - y|$. The smallest L for which this inequality holds is called *Lipchitz constant*. Neural network can be seen as a vector valued function. Lower value of L shows a more robust neural net model. However, the exact computation of the Lipschitz constant of neural networks is NP-hard, as proved in [4].

It can be proved that the Lipschitz constant is the largest singular value of the weight matrix of the layer for linear and convolutional layers. While training neural networks, we can keep optimizing the Lipchitz constant as well. This is called *Lipchitz regularization*. Some techniques of enforcing Lipchitz regularization are discussed in the GANs chapter. Interested reader may refer to Further Reading [5].

Neural Network Architectures

Architecture refers to the overall structure of the neural network, like the number of layers, the number of units in each layer, connections between layers, and so on. Modular deep learning frameworks, such as Caffe, Torch, and TensorFlow, have revolutionized complex neural network architecture designs. However, these designs are backed by problem domain knowledge and are not just random guesses or trial and error. A neural network solving a classification task in computer vision domain, like image segmentation or object detection, does not use a simple multi-layered neural network. We know in computer vision domain image filters are commonly used for feature extraction. These are basically some fixed convolution operations applied on the images. Inspired by this, the architecture design for solving almost all computer vision problems used **Convolution Neural Networks (CNN)**, which mainly consist of a sequence of filter-learning and filter processing. Following are a list of different architectures that we will discuss in the later chapters, along with some domain knowledge required to understand the models:

- **Autoencoder Architecture:** Used for dimensionality reduction, and popular for generative modelling
- **Generative Adversarial Network (GAN)** used for generative modelling that we have discussed in greater detail in [Chapter 12: Generative Models](#).
- **Convolutional Neural Network (CNN):** Mostly used for computer vision problems and image processing, and also used for natural language processing; it will be discussed in [Chapter 9: Computer Vision](#).
- **Recurrent Neural Nets (RNN):** Used for sequential data. We will discuss these in the [Chapter 10: Sequence to Sequence Models](#).
- **Transformers:** The state-of-the-art text analysis models like BERT are based on transformers. These are also being applied for other tasks like handwriting recognition and speech recognition. We have discussed this in the [Chapter 11: Natural Language Processing](#).
- **Siamese neural network:** This architecture contains two identical subnetworks that have the same configuration, with the same parameters and weights. It is used to find the similarity of the inputs by comparing their feature vectors.

There are many architectures that combine these basic architectures and build a new architecture. For example, a combination of CNN and RNN can be used to build a cursive handwriting recognizer that reads cursive handwriting from an image using CNN and then converts it into the corresponding text using RNN. The entire architecture is trained end to end using a data set consisting of handwriting images and corresponding text pair.

Conclusion

In this chapter, we discussed the basic concepts of neural networks. We covered the fundamental back propagation algorithm in detail and how it's implemented in the automatic differentiation framework available in most deep learning frameworks. We also discussed the challenges of training neural networks with SGD and ways to mitigate it at the high level. These concepts will be revisited in the following chapters on the applications of neural networks to solve specific problems.

In the next chapter, we will introduce one more important topic, that is, unsupervised clustering, and then the following chapters will mostly be applications of the concepts learned so far for solving various AI problems.

Points to remember

- A node in a neural network can be viewed as an adaptive basis function.
Even with fixed non-linear activations changing the weights, we get various basis functions.
- Each layer in the neural network can be viewed as a vector fields or tensor field that maps an input tensor/vector to the next layer's input tensor/vector. These vector fields must be differentiable, that is, the activation function used must be differentiable for the network to be trainable using back propagation algorithm. The entire network can be viewed as a composition of a finite sequence of vector fields.
- Using automatic differentiation technique discussed here, we can train any arbitrary neural network architecture, provided we use functions and operations that are differentiable.

- Overfitting and under fitting tradeoff are also faced by neural networks, and these are mitigated by proper regularization techniques and adjusting model capacity.

Further Reading

- http://cs231n.stanford.edu/slides/2018/cs231n_2018_ds02.pdf
- <http://cs231n.stanford.edu/vecDerivs.pdf>
- <https://www.jeremyjordan.me/neural-networks-training/>
- <https://papers.nips.cc/paper/2018/file/d54e99a6c03704e95e6965532dec148b-Paper.pdf>
- https://mi.nemzetilabor.hu/sites/default/files/2020-12/milab_lipreg.pdf
- *Deep Learning Book: Ian Goodfellow and Yoshua Bengio and Aaron Courville*

CHAPTER 7

Clustering

Clustering is about automatically discovering natural groups/clusters in unlabeled data such that the degree of similarity between samples of the same cluster and the degree of dissimilarity between samples of different clusters is maximized. It is one of the unsupervised learning techniques where learning is based on unlabeled data. Let us understand this with an example.

A company wants to grow its business. To grow the business, the company has decided to group customers so that offers can be fine-tuned to each group. The company doesn't have specific rules to decide the group of a customer. It wants to find groups that are natural and have similar buying patterns so that the offers can be fine-tuned to enhance the buying experience of their products. The process of finding the groups, also called clusters, by analyzing the unlabeled data to find natural patterns is called **clustering analysis**.

Data samples with similar patterns can form one group called *cluster*. Now, we can rephrase the company's project as finding clusters among customers so that the offers can be fine-tuned for each cluster to grow the business. There are other areas where clustering analysis brings value, like anomaly detection, genetics, pharmacy, and document information retrieval.

Anomaly detection includes fraud detection of financial transaction, labelling newly produced mechanical item as defective and many others. In genetics, clustering analysis can be used to identify DNA that produces similar behavioral patterns in animals. In turn, it helps to understand the evolution of living things on earth. In pharmacy, proven medicines and newly discovered drugs are clustered. One among the many newly discovered drugs is selected, which is closest to the proven drug cluster. The selected new drug is then used for next stage experimentation. This helps to pick the best suited drug for a disease with less time and resources.

There are numerous clustering algorithms with applications in various domains for analysis. In this chapter, we will discuss popular algorithms belonging to different clustering categories. Along with algorithms, we will discuss cluster evaluation methods necessary for the comparison of clustering algorithms.

Structure

In this chapter, we will cover the following topics:

- Defining cluster and approaches to form clusters
- Similarity and dissimilarity metrics
- Evaluation of clustering algorithms
- Categories of clustering algorithms
- Few popular algorithms in each category

Objectives

After going through this chapter, you will be able to understand the meaning of a cluster and the domains where clustering algorithms can be applied. You will learn about the different categories of clustering algorithms, along with a few popular algorithms under each category. You will learn about the metric to measure similarity or dissimilarity between data samples. You will also be introduced to various evaluation techniques applicable for clustering algorithms.

Forming clusters

Clustering is an unsupervised learning technique that identifies natural clusters such that the degree of similarity between samples of the same cluster and the degree of dissimilarity between samples of different clusters is maximized. Similarity and dissimilarity criteria can vary based on the problem statement and the clustering algorithm.

Grouping of data samples result in a cluster. Definition of a cluster varies based on the algorithm due to the similarity/dissimilarity metrics chosen. The objective of all algorithms remains the same: group all the samples that possess similar characteristics into one cluster and those with dissimilar

characteristics to different clusters. The approach of forming clusters by assigning data samples can be broadly classified into two categories:

- **Hard clustering:** Every data sample either belongs to one cluster or doesn't belong to any. K-means is a hard clustering algorithm where each data sample is assigned to only one cluster. Financial fraud detection uses hard clustering approaches.
- **Soft clustering:** Every data sample belongs to every cluster formed by the algorithm. Belongingness of a data sample to a cluster is represented by a numerical value. For example, numerical value can be likelihood of a data sample belonging to the cluster. Fuzzy theory-based algorithms follow soft clustering. Streaming websites suggest movies of different genre based on a customer's watched videos. Suggested videos will contain more from the genre customer likes.

Variations are possible between these extremes:

- **Strict partitioning clustering with outliers:** Every data sample belongs to a maximum of one cluster. Data samples that don't belong to any cluster are called outliers. K-medoids clustering algorithm partitions the data samples with outliers.
- **Overlapping clustering:** Data sample can belong to more than one cluster in hard way. Clustering algorithms based on fuzzy theory have overlapping clusters.
- **Hierarchical clustering:** Clusters are related to each other in hierarchically. A data sample belonging to child cluster also belongs its parent cluster. Balanced Iterative Reducing and Clustering using Hierarchies (**BIRCH**) is a popular hierarchical clustering algorithm.

We discussed ways of forming clusters from the data samples. Next, let us discuss metrics that are used to decide the belongings of a data sample to the cluster.

Distance and similarity

Metric or distance or dissimilarity function is a non-negative real valued function, which provides a notion of how far the two elements of the set

are. *Metric function* on a non-empty set X is defined as $d: X \times X \rightarrow [0, \infty)$, where the following properties must hold for $x, y, z \in X$:

- Identity of indiscernible
- Symmetry
- Triangle inequality

The main criteria to judge whether data points are similar is based on the distance between them. The distance metric is preferred for quantitative data where each sample is associated with unique numerical value. The higher the value of distance metric between data points, the farther the samples. For example: revenue in rupees, height in meters, distance between stars in light years, and age in years/months/days.

For qualitative data, which is primarily non-numerical in nature, similarity metric is commonly used. The higher the value of similarity metric between data points, the closer the samples.

For example: color of an object, texture of an object, like shiny or dull.

Popular distance and similarity metrics are summarized in [Table 7.1](#). Let two data samples be represented in vector space of d -dimensions as . Refer to the following table:

Table 7.1: Popular distance and similarity metrics and their formula

Let us understand the Minkowski distance with different values of n . For different values of n , we get different metrics, which are used in different contexts. When $n < 1$, distance function follows only the first two properties due to which it cannot be called a metric function. But when $n \geq 1$, distance function follows all three properties due to which it can be called metric function. Let us understand the behavior of Minkowski distance with three values of $n = 1, 2, \infty$ in two-dimensional space (with $d=2$). While calculating the distance between vectors, let us consider one of the vectors as origin, that is, $\mathbf{b} = \mathbf{0}$. Then, Minkowski distance from origin in two-dimensional space becomes:

Let us consider all points that are unit distance (`distance_from_origin = 1`) from the origin for various values of n . [Figure 7.1](#) shows these unit distance points from the origin for values of :

Figure 7.1: Plotting of all points that are unit distance from the center, where Minkowski distance is used with various values of n , (left) $n=1$, (middle) $n=2$, (right) $n=3$

Consider the analysis of documents where each document can be of any number of pages. Documents need not be of the same page count. One way to represent documents numerically is to count the occurrences of important words in the document. If we are interested in d number of words, then each document is represented by a vector of d -dimensions. Each word is represented with one dimension. Value for the document in a particular dimension represents the frequency of the occurrence of that word representing that dimension. Once the documents are represented with vectors, which metric do you use to find the distance between these vectors?

Let us consider two documents and two important words . Then, vectors representing these documents will belong to , as shown in [Figure 7.2](#). The ratio between the count of words for both documents are similar . The metric we use should reflect the same, saying they are closer. Refer to the following figure:

Figure 7.2: Euclidean distance and angle between two vectors

Euclidean distance between these vectors would be:

Whereas the cosine distance between these vectors would be:

From the calculation, we can conclude that cosine distance is better suited metric for comparison as compared to Euclidean distance. A detailed

discussion about representing documents in vector space is discussed in [Chapter 11, Natural Language Processing](#)

Mahalanobis distance can find the distance between a point and the distribution. It uses covariance information for calculating distance, due to which it is useful on multivariate data. It has application in anomaly detection and other fields as well. Let us consider a few samples of data points and calculate their mean and the Mahalanobis distance of samples from the mean. Before calculating the Mahalanobis distance, we need to calculate the covariance matrix, where rows/columns represent dimensions of the data points. *Code 7.1* provides the steps to calculate covariance matrix and its inverse on data samples:

```
1. import numpy as np
2. from numpy.linalg import pinv
3. cluster_samples = np.array([
4.     [10,15], [16,24], [25.,21], [33,28], [38,45], [40.,36],
[37.,20]
5. ])
6. cluster_mean = np.mean(cluster_samples, axis=0)
7. # calculating covariance matrix
8. clust_cov = np.cov(cluster_samples.T)
9. clust_cov_inv = pinv(clust_cov)
```

Code 7.1: Calculation of covariance matrix and its inverse

The preceding code would output cluster mean as and inverse of the covariance matrix as follows:

Code 7.2 provides the functions to calculate Mahalanobis and Euclidean distance:

```
1. def mahalanobis_dist_sqr(sample1, sample2, cov_matrix_inv):
2.     mean_smp_diff = sample1 - sample2
3.     return
np.dot(np.dot(mean_smp_diff.T,cov_matrix_inv),mean_smp_diff)
4. def euclidean_dist_sqr(sample1, sample2):
5.     return np.sum(np.square(np.subtract(sample1, sample2)))
```

Code 7.2: Calculation of Mahalanobis and Euclidean distance

Let us consider two samples from the cluster to calculate Mahalanobis and Euclidean distance, as shown in *Code 7.3*:

```

1. outlier_smp = cluster_samples[6]
2. cluster_smp = cluster_samples[5]
3. # cluster mean vs cluster sample
4. mhl_dist_sqr_smp1 = mahalanobis_dist_sqr(cluster_mean,
cluster_smp, clust_cov_inv)
5. eucl_dist_sqr_smp1 = euclidean_dist_sqr(cluster_mean,
cluster_smp)
6. # cluster mean vs cluster outlier
7. mhl_dist_sqr_smp2 = mahalanobis_dist_sqr(cluster_mean,
outlier_smp, clust_cov_inv)
8. eucl_dist_sqr_smp2 = euclidean_dist_sqr(cluster_mean,
outlier_smp)

```

Code 7.3: Measuring distances between two sample points

The preceding code would output the following information. [Table 7.2](#) contains the square of distances of samples from the mean:

With respect to cluster_mean	Mahalanobis distance square	Euclidean distance square
cluster_smp	1.04	214.9
outlier_smp	3.14	122.5

Table 7.2: Mahalanobis and Euclidean distance of two samples from the cluster mean

Euclidean distance was not able to find the outlier with respect to cluster distribution. In fact, Euclidean value indicates that the outlier sample is nearer to the cluster mean than the cluster sample. On the other hand, Mahalanobis distance is clearly indicating that outlier sample is far as compared to the cluster sample. These samples are captured in [Figure 7.3](#):

Figure 7.3: Plotting of data samples with special markers on cluster's mean, a data sample in cluster and outlier

Cluster quality

Challenges to evaluate these unsupervised clustering algorithms are different as compared to supervised algorithms. Clustering algorithms are

used to extract natural patterns from the unlabeled data. As these patterns are not known before, algorithms would extract patterns based on the approach or heuristics. Most times, we cannot decide whether the extracted patterns are right or wrong.

Evaluation of clustering algorithms can be categorized into internal and external. *Internal evaluation* is based on the assumption that data samples belonging to one cluster should be more similar as compared to data samples belonging to different clusters. External evaluation is performed on the labeled data set. This labeled data set is not seen by the algorithms. Most times, these kinds of labeled datasets are created by humans who are experts in the problem domain we are solving. These are also called *benchmarks*. The process of evaluating algorithms using standard labeled dataset is called *external evaluation*.

Internal evaluation

Internal evaluation techniques assign higher score to algorithms that produces clusters with high similarity within a cluster and low similarity between clusters. However, we should keep in mind that these indicators don't imply that the algorithms produce valid/invalid results.

For example: Consider that an indicator assumes natural clusters are convex in shape then this indicator would score high for algorithms that work on similar heuristics. Definitely, this indicator will score low for algorithms that do not assume convex patterns in the data.

Davies-Bouldin indicator

Let there be n clusters denoted as . Let us use centroid to represent a cluster (group of similar data samples) with a single vector. *Centroid of a cluster* is defined as arithmetic mean of all samples belonging to the cluster. Let the centroid of these clusters be denoted as . Let denote the average distance of all data samples belonging to cluster, and let denote the average distance of all data samples belonging to cluster . Let the distance metric between two centroids be denoted as . Distance function can be any one of the distance metrics. *Davies-Bouldin indicator* is defined as follows:

Dunn indicator

Dunn indicator helps us in identifying the algorithm that extracts dense and well-separated clusters. It is calculated as the ratio between minimum inter-cluster distance and the maximum intra-cluster distance. It is defined as follows:

Where measures the distance between cluster and and measures intra-cluster distance of cluster . Choice of these distance functions can vary based on the problem domain. Algorithms produce clusters with high Dunn indicator value is preferred.

Silhouette coefficient

Silhouette coefficient measure of each data sample depends on how similar the data sample is to its assigned cluster, called *cohesion*, and how dissimilar the data sample is compared to data samples belonging to other clusters, called *separation*. Cohesion and separation with respect to one data sample is shown in [Figure 7.4](#):

Figure 7.4: (left) cohesion: with in the cluster (right) separation: outside the cluster

Consider a data sample belonging to the cluster . represents the count of data samples assigned to the cluster . Then, similarity or cohesion of the data point with respect to the other data samples of the same cluster is defined as follows:

Value signifies how well it is assigned to the cluster ; a smaller value indicates better assignment. Dissimilarity or separation of the data sample with respect to the samples of the other clusters , where is defined as follows:

Value signifies the smallest mean distance/dissimilarity of the point to all other points in any other cluster . Cluster the smallest mean dissimilarity

with data sample is called the neighbor cluster for the data sample.

Using the terms of similarity and dissimilarity, silhouette value is defined as follows:

Silhouette value ranges $[-1, 1]$. A higher value indicates that the data sample is well matched to the cluster it is assigned to and more dissimilar to a neighbor cluster. A higher value is preferable for the data sample. A larger percentage of data samples having high silhouette value is preferred. This evaluation indicator can be used with any distance metric. Using the silhouette value of all N data samples, we can define **Silhouette Coefficient (SC)** as follows:

Plotting *silhouette coefficient* of data samples will help us in deciding the right number of clusters. The right number of clusters occur when the SC score of all data samples is near to the overall average SC score. For more information, refer to the link stated in *References [2]*.

External evaluation

External evaluation of the algorithms is performed on standard labeled data set. As the data set is labeled, we can identify whether the test samples are correctly clustered. After the application of any algorithm on test data, which is already labeled (ground truth), we have clusters containing this labeled data. Clusters formed by the selected algorithm may be different from the ground truth. Evaluation of the selected algorithm will be performed by comparing the clusters formed by the algorithm and the ground truth (labeled test data). Consider one cluster out of all clusters produced by the selected algorithm, and then we define the following terms:

- **True Positive - TP:** Data samples that belong to the cluster and are correctly assigned to the cluster .
- **True Negative - TN:** Data samples that belong to another cluster are rightly not assigned to .
- **False Positive - FP:** Data samples that belong to another cluster are assigned to cluster .

- **False Negative - FN:** Data samples that belong to cluster but are incorrectly assigned to another cluster.

Rand index

Rand Index (RI) computes the similarity between clusters formed from the algorithm versus the ground truth data of labeled data set. RI for each cluster is defined as follows:

F-measure

F-measure indirectly weights the terms TP, TN, FP, FN through the parameter to provide one number for each cluster or class. Two measures that used to define F-measure are Precision and Recall.

Precision (P) of a cluster is the ratio between the number of correctly assigned samples and the count of all samples assigned to the cluster, defined as follows:

Recall (R) of a cluster is the ratio between the number of correctly assigned samples and the count of all samples that rightly belong to the cluster (based on ground truth), defined as follows:

Using precision and recall, we can define *F-measure* using parameter as follows:

Fowlkes–Mallows index

Fowlkes–Mallows index (FM) computes similarity between two clusters. FM can be defined as the geometric mean of precision P and recall R . A higher value indicates that clusters are more similar. FM is defined as follows:

Jaccard index

Jaccard index provides value in range while comparing two sets. Value of 0 indicates that the sets have no common elements. Value of 1 indicates that the sets are identical. Jaccard index for comparing two sets is defined as follows:

Clustering algorithms

There exist many factors to choose the clustering algorithm for our task. Factors include data type, dataset size, data sample dimensions, scaling of algorithm based on the larger/newer data, time complexity and resource requirements of the algorithm, domain of the problem, similarity/dissimilarity comparison between data samples. Based on these factors, we can choose the clustering algorithm. There exist many types of clustering algorithms, each of which is suited for a particular context. Next, we will discuss a few important categories of clustering algorithms.

Partition-based clustering

In this clustering approach, each cluster is represented by a *central* or *centroid* vector, which need not be the data set vector. This *central* or *centroid* vector is representative of the corresponding cluster. There exist various methods to calculate this representation vector. Algorithm might remain the same, but the method to choose a vector representing a cluster might differ, resulting in different clusters and interpretation. k -means and k -medoids are the most popular clustering algorithms in partition-based clustering.

K-means

K-means [3] clustering algorithm partitions the data set samples into k (given) clusters such that it minimizes **With-In Cluster Sum of Squares (WCSS)** (which is variance). Centroid or central vector of a cluster is calculated by taking the mean of all samples belonging to the cluster, hence the name k -means.

Formally, k -means divides the samples , where and d = dimensionality of data-sample vector, into k distinct sets such that:

Where μ_i is mean of the data vectors belonging to cluster .

Finding the optimal solution is NP-hard problem, time complexity is . Most of the proposed approximate algorithms work on heuristics and converge quickly to local optimum. Popular approximate solution is k -means; the k -means algorithm follows the given steps with the given k value.

1. Initialize randomly, k data samples as centroid of k clusters .
2. Every data sample is assigned to a cluster whose centroid vector is nearest to the sample vector . In other words, is assigned to a cluster whose centroid is , where j is .
3. Calculate centroids for k clusters by taking the mean of all samples assigned to the respective cluster.
4. If there is change in any centroid of the cluster, then go to step 2, else stop.

Algorithm would output k clusters where each data sample is assigned to one cluster. The value of k is input to the algorithm. How do we know the right value of k ? There doesn't exist a correct way to find the value of k . We must try with different values of k and infer each of the output based on our objective. Elbow method helps to find right number of clusters but may not work in all situations.

Elbow method

Plot the graph with k -value on one axis and error on another. In some cases, we can notice that error decreases gradually with increase in k value and then suddenly, the rate of error diminishes. The value of k at this juncture, known as elbow, can be considered. This method to determine the value of k may produce right result in a few cases, and there is no supporting mathematical foundation stating that this method will work in all cases.

Challenges

K-means algorithm is not widely used due to the following challenges:

- The algorithm works on the assumption that all clusters are of similar sizes.
- An invalid k value will probably result in a few invalid clusters.
- It works on the assumption that clusters are spherical. Clusters of Iris flower dataset are not spherical, and this algorithm fails to find right clusters on this dataset.
- Algorithm output is influenced by presence of outlier or noise as centroid vector selection is affected by outliers.

We can change the method to calculate the centroid of a cluster. Instead of mean, we can use medoids or medians. The algorithm that uses median to calculate cluster's centroid is called k -medians, and the algorithm that uses medoids is called k -medoids. Other steps for these algorithms will remain similar as compared to k -means.

K-medoids

Medoid of the cluster is defined as the data sample of the cluster whose average dissimilarity to all the data samples within the cluster is minimum. **K-medoids** [4] chooses actual data points as centroid, and it aids better interpretability of the cluster centroids as compared to centroids k -means. Since, k -medoids minimizes a sum of pairwise dissimilarities instead of a sum of squared Euclidean distances (in case of k -means), it is more robust to noise and outliers than k -means. K -medoids can be used with arbitrary dissimilarity measures, whereas k -means works best with Euclidean distance. Here too, k -medoids problem is NP-hard to solve exactly. A few of the popular approximate algorithms are **Partitioning Around Medoids (PAM)** [5], **Clustering Large Applications (CLARA)** and **Clustering Large Applications based on Randomized Search (CLARANS)** [6].

The PAM algorithm follows these steps:

1. Randomly initialize k data sample as centroids.
2. Assign all non-centroid data samples to the closest centroid.
3. For each centroid m and for each non-centroid x :
 - a. compute the cost by temporarily swapping m & x
 - b. remember the least cost combination

4. Swap m & x that gives least cost.
5. If there is change in any of the centroid of cluster, then go to step 3, else stop.

Partition based-clustering algorithms mostly form hard clusters. The main limitation of this category of algorithms is choosing the value of k . Next, let us discuss density-based clustering that do not require k value as input and outliers need not be assigned to any cluster.

Density-based clustering

Density-based clustering algorithms assigns data samples in high-density region to the same cluster. The number of clusters is determined based on the cluster density and a threshold. Clusters with lower density can be considered outliers. Clusters output from these algorithms are often hard clusters. Popular algorithms in this category are **Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [7]**, **Ordering Points To Identify Cluster Structure (OPTICS) [8]**, **Density-based Clustering (DENCLUE) [9]**.

DBSCAN

In **DBSCAN** [7], data points are classified as core points, directly reachable points, reachable points, and outliers. Before defining these, let us define the ϵ parameter and the neighborhood. Let the ϵ -parameter (input to algorithm) that specifies ϵ -neighborhood with respect to a point p belonging to dataset D be defined as follows:

Where ϵ can be any distance metric. Let's define min_pts as a positive integer value that is input to the algorithm. We can define different types of points with respect to ϵ -neighborhood, as shown in [Figure 7.5](#). Definitions of these points are captured as follows:

- A point is called *core point* if the number of points within the distance of ϵ (including the point under consideration) from the core point is $\geq min_pts$.

- A point q is said to be *directly reachable* from p if q is within the ε distance from the core point p .
- A point q is said to be *reachable* from core point p if there is a path , where , and every is directly reachable from , where .
- Points that are not reachable from any other point are called *outliers*. Refer to the following figure:

Figure 7.5: Types of points with respect to ε -neighborhood

Each cluster will have at least one core point and at least $min_pts - 1$ number of points *directly reachable* from the core point. Reachability is not symmetric as only core points can reach non-core points.

Let us define a symmetric relation. Two points p and q are said to be *density-connected* if they are *reachable* from a common point, say o . [Figure 7.6](#) depicts density-connected points. From the definition, the common point o must be core point as reachability is defined only for the core points. With this symmetric relation, we can define the clusters formed by DBSCAN algorithm. These clusters satisfy two properties:

- All points within a cluster are mutually density-connected.
- If a point is density-reachable from any point of the cluster, then it must be part of that cluster. Refer to the following figure:

Figure 7.6: Density connected points

Pseudo code

Let us now write pseudo code:

For each non-processed point p belonging to the dataset D :

If p is a core point (as shown in [Figure 7.5](#)), then:

$$C = \{q \mid q \text{ is density-connected with } p, \\ \text{for given } \varepsilon \text{ & } min_pts \text{ where } q \in D\}$$

Mark all points belonging to as processed.

Output cluster .

Else, mark point p as outlier and processed.

Choosing parameters

Parameter $\varepsilon > 0$ is real number, `min_pts` is a positive integer, and *distance* metric is an input to the DBSCAN. Choice of these parameters impacts the end result of the DBSCAN. A few approaches for the selection of these values are discussed as follows:

- Parameter `min_pts` influences the number of points in one cluster. The higher the value, more points are assigned to the same cluster. Keeping the value large will result in addition of noise or outliers to the clusters. Low value will result in a greater number of clusters and outliers.
- The value of ε can be chosen by plotting the average of distances from every point to its `min_pts` nearest neighbors. These average `min_pts` distances from points are plotted in increasing order and the value of ε will be the “*knee*” point, where there is a sharp increase in the graph. For more information, refer to [20] in the *References* section. A lower value will result in a high number of clusters and high number of outliers, whereas a higher value will result in low number of clusters.
- The choice of *distance* function is tightly coupled with ε value. Its choice should be based on the domain problem we are solving.

Advantages

DBSCAN has a few advantages, mentioned as follows:

- The number of clusters are formed based on the input parameter. There is no requirement to input the number of clusters to the algorithm, like in k -means.
- It can find clusters of different shapes. This algorithm works even when one cluster is surrounded by another.
- It is robust to outliers and is mostly insensitive to the order of points being processed.

Limitations

A few challenges faced by applications of DBSCAN are mentioned as follows:

- Performance degrades if there exist clusters with large differences in their densities.
- Performance depends on the distance metric being used. The most common metric used is Euclidean distance. Choosing the value of ϵ based on Euclidean distance in higher dimensions is challenging.

Figure 7.7 shows clusters formed by k-means and DBSCAN. We can see that DBSCAN forms clusters with connected points. As k-means forms clusters around the centroid, we can see that one circle is divided into two clusters. Refer to the following figure:

Figure 7.7: Application of k-means and DBSCAN on two types of sample sets.

Distribution-based clustering

Distribution-based clustering is based on the assumption that if original data maps to more than one distribution, then samples belonging to the same cluster should come from one distribution. One of the popular algorithms under this category is **Gaussian Mixture Model (GMM)** [10].

Gaussian Mixture Model

In the K-means algorithm we just discussed, we have shown the cluster center using a single point (may not always be data point) and assigned each data point to the nearest center point. Suppose the groups or clusters have to overlap. Then, assigning every data point to a single cluster is not possible. For example, let's take a look at sample two-dimensional data in *Figure 7.8*. We can see the formation of two groups in the data, one of which has a lot of spread horizontally and one that has a lot of spread vertically, but both are centered at the same place. k-means will not be able to discover this pattern as it will try to find two circular non-overlapping clusters, assuming that Euclidean distance is used. Refer to the following figure:

Figure 7.8: Sample data points from two distributions

GMMs are an extension of the k-means model, in which each cluster is modeled with multivariate Gaussian distributions. So, we have to not only find the mean but also a covariance that describes their ellipsoidal shape of the clusters. The parameters of this distribution can be estimated by maximizing the likelihood of the observed data, which is done by an algorithm called **Expectation Maximization (EM)**, discussed in the following sections. This is a soft clustering technique where we assign data to each cluster with some soft probability. Moreover, with this approach of clustering, we are essentially creating a probabilistic generative model (explained in [chapter 12 Generative Models](#)) for the data. Hence, with GMM, we can even sample new data points that resemble the points in the data set; we can impute missing data values.

Let's understand the GMM probability distribution model in one dimension, as shown in the [Figure 7.9](#). Here, we have taken a mixture of three univariate Gaussian distributions with different means and variance . The probability of observing any point from this distribution is given by . Here, are called the *mixture weights*, and we have . Suppose we have 20 points from the leftmost cluster, 10 from the middle cluster, and another 20 from the rightmost cluster; then, we can take . Refer to the following figure:

Figure 7.9: Three univariate Gaussian distributions

We can interpret this joint probability distribution over x in a simple generative way. To draw a sample x from $P(x)$, we first select one of the components with discrete probability , that is, if we denote the mixture component by the discrete random variable Z , then . Components with large probability are selected more often. This is similar to choosing one cluster centre in k-means. Now, we can sample X from the corresponding Gaussian . Thus, these two distributions make a joint model over X and Z together.

The variable Z is sometimes called *latent* or *hidden variable*. The presence of the unknown value of Z helps explain the patterns in the values of X . Detailed discussion about this topic can be found in [Chapter 5, Statistical Inference and Applications](#) and [Chapter 12, Generative Models](#).

For clustering problem, we have multivariate data and hence we use a multivariate Gaussian with vector mean of length n (the same size as the number of features in a data) and an covariance matrix . Now, we need to estimate these parameters by **Maximum Likelihood Estimation (MLE)**. But direct application of MLE is hard for mixture models and hence we will use iterative algorithm called **Expectation Maximization (EM) algorithm**. EM has two steps: expectation step (E-step) and maximization step (M-step). We know how to calculate MLE parameter estimates of a Gaussian model: Let μ be the mean of the data, that is, , and the covariance estimate the mean of the matrices formed by the outer product of X minus with itself, that is, .

EM Algorithm

Let us now discuss the steps of EM algorithm:

- Initialize the means , covariances and mixing coefficients : One way to initiate the GMM is to first run k-means and choose the centres as initial estimates for . k-means also tells us which data points belong to which cluster. A good starting estimate for the is the within-cluster covariances, and the weights are the fractions of data points allocated to each cluster.
- Expectation (E-step):
 - for each data point :
 - for each cluster c:
 - Compute a measure of relative probability (called responsibilities) as follows:
- Maximization (M-step): Re-estimate the parameters using the responsibilities as follows:
 - for each cluster c:
 - Total responsibility associated with the cluster

Iterate E-step and M-step until convergence, that is, until the norm of the estimated parameters stop changing significantly. Every step of EM algorithm increases the log-likelihood (explained in [Chapter 5, Statistical Inference and Applications](#)) of our model.

Note: EM algorithm is very similar to k-means algorithm. k-means does not estimate the covariances of the clusters but only the cluster means. It assumes the clusters are circular by taking fixed Identity covariance matrices. Choosing $\Sigma_c = \in I$, where \in is a variance parameter that is shared by all the components, the responsibilities become

Taking the limit , in the denominator, the term , which is closest to zero, will vanish slowly and the responsibility for that term will approach 1. So, we get a hard assignment of data points to clusters, that is, each data point is assigned to the cluster having the closest mean. Thus, EM algorithm reduces to k-means with the fixed circular covariance assumption.

The EM algorithm can be generalized for any model with hidden variables. A probabilistic model is one in which we denote all the observed variables by X and all the hidden variables by Z . The joint distribution is governed by a set of parameters denoted . The likelihood function is given by .

Hierarchical-based clustering

Hierarchical clustering builds hierarchy of clusters from the data points. Approaches to build the hierarchy of clusters can be divided into two types: agglomerative and divisive:

- **Agglomerative** is a bottom-up approach where initially, each data point is treated as a cluster, and then clusters are merged while moving up the hierarchical structure.
- **Divisive** is a top-down approach where initially, all data points are treated as one cluster, and then clusters are split while moving down the hierarchical structure.

Finding optimal hierarchical structure is NP-hard. So, most times, merging and splitting of the clusters are decided in a greedy manner. Selection of clusters for merging and splitting is based on the distances between clusters. And the choice of the distance metric greatly influences the shape of the clusters. Popular algorithms under this category are **Balanced Iterative Reducing and Clustering Using Hierarchies (BIRCH)** [11], **Clustering Using Representatives (CURE)** [12], and Robust Clustering using links (ROCK) [13].

Agglomerative clustering

Algorithms under this category start by treating every data sample as a cluster on its own. Distances among all existing clusters are calculated. Two clusters with the shortest distance are merged to form one cluster at one level up in a hierarchical way. These two steps are performed repeatedly till we reduce clusters to the required number of clusters.

Example: Let us understand the approach with a few data samples, as shown in [*Figure 7.10*](#). Initially, every sample is the clusters, as shown in (a). Distances among all clusters are calculated, and the shortest distance is chosen for merging two clusters. As shown in (b), two clusters with the shortest distance are merged. These two steps of calculating distances among clusters and merging the shortest distance clusters is repeated until we obtain the desired number of clusters. Here, the simple Euclidean metric is used for distance calculation. Refer to the following figure:

Figure 7.10: Agglomerative hierarchical clustering: At each step, two closest clusters or data samples are merged

Distance between clusters

How do we calculate the distance between clusters? Algorithms used to calculate the distance between clusters are called linkage algorithms. Let us understand the categories of linkage algorithms that are popularly used to calculate distance between two clusters :

- **Single linkage:** Distance between two clusters is defined as the shortest distance between two data samples belonging to each cluster.

It is defined as follows:

- **Complete linkage:** Distance between two clusters is defined as the longest distance between two data samples belonging to each cluster. It is defined as follows:
- **Average linkage:** Distance between two clusters is defined as average of distances between every point in one cluster to every point of another cluster.
- **Centroid Linkage:** Distance between two clusters is defined as the distance between the centroids of the clusters.
- **Ward linkage:** Objective of this approach is to minimize the total within-cluster variance. Merging of clusters always leads to an increase in within-cluster variance. Two clusters are chosen for merging, which minimizes the increase of within-cluster variance.

The comparison of linkage methods on a sample data set is captured in [Figure 7.11](#). Single linkage and complete linkage are sensitive to outliers due to the distance calculation employed. As average linkage and centroid linkage calculate distance by considering all samples of the cluster, they are less sensitive to outliers. Refer to the following figure:

Figure 7.11: Comparison of linkage algorithms on two data sets

The main objective of all hierarchical clustering algorithm is to find two clusters for merging in the bottom-up approach and one cluster for splitting in the top-down approach. Criteria to choose the cluster varies by the algorithm, as discussed. To perform these operations on a larger data set, we need to use a data structure that works efficiently.

BIRCH

Balanced Iterative Reducing and Clustering using Hierarchies (BIRCH) [11] first generates a compact summary of the original that retains as much distribution information as possible, and then the clustering algorithm is applied on the summary instead of the original data set. Various hierarchical algorithms can use this approach on large data sets, where the complete data set doesn't fit into the available memory. With little modification, it can be adapted to non-hierarchical algorithms, like k-means.

Graph-based clustering

In the graph-based approach, data samples are represented as nodes of the graph, and edges between these nodes represent their relationship. Once data samples are represented in graph form, we can apply graph-based algorithms to obtain the desired results. Popular algorithms in this category are CLICK [14], algorithms based on minimum spanning tree [15][16], normalized cuts and image segmentation [17], and spectral clustering [18]. Graph-based clustering algorithms are commonly used in social network analysis, power grid networks, telecommunication networks, and spreading of a disease through contact.

Spectral clustering

Spectral clustering techniques make use of eigenvalues (spectrum) of the similarity matrix to perform dimensionality reduction. Any clustering algorithm can then be applied on the dimensionality reduced data set.

Before applying graph techniques, we must represent data in graph form. Data samples represent nodes, and edges represent the similarity relationship between data samples. Value of these edges will form *similarity matrix*. We must partition the graph into the required number of groups such that edges between different groups have low value (lower similarity) and edges within the group have high value (higher similarity). Graph partitioning is NP-hard problem. Approximate solution can be found by using Graph Laplacians.

Considering all data samples as enumerated, similarity matrix can be defined as a symmetric matrix A where represents similarity relationship between data samples i & j . The next step is to obtain Laplacian matrix of A . There exist various ways to define the Laplacian matrix, and each one

impacts clustering algorithms differently. One of the ways to define Laplacian matrix is as follows:

Here are a few properties of Graph Laplacian that help us to find an approximate solution quickly:

- is positive semidefinite when and A is symmetric
- Eigenvalues of are real and non-negative, and the corresponding eigen vectors form orthonormal basis
- Dimensions of null space of is equal to the number of connected components of the graph

Algorithm to partitioning the graph is as follows:

1. Create similarity matrix A .
2. Construct graph Laplacian .
3. Calculate eigenvalues and corresponding eigen vectors.
4. Pick k eigen vectors corresponding to the smallest k eigenvalues (eigenvalues of a positive definite real symmetric matrix are always real).
5. Construct projection matrix P using k eigen vectors.
6. Project the data to lower k -dimensions using .
7. Apply clustering algorithm on dimensionality reduced data.

Let us use spectral and k-means clustering algorithms implemented in Scikit-learn library on the toy dataset to understand the differences in the resulting clusters. [Figure 7.12](#) illustrates the output of k-means and spectral algorithm:

Figure 7.12: Application of k-means and spectral algorithm on three data sets

Fuzzy theory-based clustering

Fuzzy means vagueness. This occurs in scenarios where classification of an object cannot be performed deterministically. The object can belong to

more than one class with certain probability. Fuzzy set theory is an extension of classical set theory, where elements belong to a set with certain probability (called *degree of membership*). Consider classifying the people based on age into two sets: young and old. As there is no universally agreed age threshold after which a person becomes old, this is an example of fuzziness. A person with age 35 will belong to young with probability 0.6 and old with 0.4.

In fuzzy theory clustering algorithms, data sample's relationship to the cluster is replaced from discrete value {0,1} to a continuous value [0,1]. Here, each data sample can belong to more than one cluster, and the degree of belongingness to a cluster is represented by the continuous value [0,1]. One of the most popular algorithms in this category is **Fuzzy C-Mean algorithm (FCM)** [19]. Algorithms under this category output soft or overlapping clusters. Fuzzy algorithms are relatively insensitive to initial conditions.

Fuzzy c-means

Consider an n data sample set divided into m fuzzy clusters , where is centroid of the cluster. Belongingness of to the cluster is represented by , where . This value is also called *membership value*. d is a hyper-parameter controlling the fuzziness of the clusters (higher value results in higher fuzziness). *Membership value* is defined as follows:

Fuzzy centroid of the cluster is defined as follows:

The objective of FCM is to minimize the following function:

The K-means algorithm optimizes the same objective function stated before, but the membership value will be either zero or one. Degree of fuzziness is controlled by value of . As becomes either 0 or 1, FCM turns to k-means algorithm.

We have discussed different categories of clustering algorithms. There are no well-defined guidelines that talk about the best-suited algorithm for a

particular domain. We must experiment different distance metrics and clustering algorithms on our problem to meet the requirements.

Conclusion

In this chapter, we discussed distance metrics and clustering algorithms that are specifically used on unlabeled data. We also discussed various popular categories of clustering algorithms with specific algorithm in each category. There are different real scenarios where unlabeled data, or little labelled data is available. In these scenarios, we must use clustering algorithms.

In the next chapter, we will explore neural networks inspired by brain neurons, which have shown the capability to learn complex relations in data.

References

1. Xu, D. and Tian, Y., 2015. *A comprehensive survey of clustering algorithms*. Annals of Data Science, 2(2), pp.165-193.
2. 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.
3. MacQueen, J., 1967, June. *Some methods for classification and analysis of multivariate observations*. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability* (Vol. 1, No. 14, pp. 281-297). Popularly known as k-means.
4. Park, H.S. and Jun, C.H., 2009. *A simple and fast algorithm for K-medoids clustering*. Expert systems with applications, 36(2), pp.3336-3341. Popularly known as k-medoids.
5. Kaufman, L. and Rousseeuw, P.J., 1990. *Partitioning around medoids (program pam)*. *Finding groups in data: an introduction to cluster analysis*, 344, pp.68-125.
6. Schubert, E. and Rousseeuw, P.J., 2019, October. *Faster k-medoids clustering: improving the PAM, CLARA, and CLARANS algorithms*. In

International conference on similarity search and applications (pp. 171-187). Springer, Cham.

7. *Ester, M., Kriegel, H.P., Sander, J. and Xu, X.*, 1996, August. A density-based algorithm for discovering clusters in large spatial databases with noise. In kdd (Vol. 96, No. 34, pp. 226-231). Popularly known as DBSCAN.
8. *Ankerst, M., Breunig, M., Kriegel, H.P., Ng, R. and Sander, J.*, 2008. *Ordering points to identify the clustering structure*. In Proc. ACM SIGMOD (Vol. 99). Popularly known as OPTICS.
9. *Hinneburg, A. and Keim, D.A.*, 1998, August. *An efficient approach to clustering in large multimedia databases with noise*. In KDD (Vol. 98, pp. 58-65). Popularly known as DENCLUE.
10. *Rasmussen, C.E.*, 1999, November. *The infinite Gaussian mixture model*. In NIPS (Vol. 12, pp. 554-560). Popularly known as GMM.
11. *Zhang, T., Ramakrishnan, R. and Livny, M.*, 1996. *BIRCH: an efficient data clustering method for very large databases*. ACM sigmod record, 25(2), pp.103-114.
12. *Guha, S., Rastogi, R. and Shim, K.*, 1998. *CURE: An efficient clustering algorithm for large databases*. ACM Sigmod record, 27(2), pp.73-84.
13. *Guha, S., Rastogi, R. and Shim, K.*, 2000. *ROCK: A robust clustering algorithm for categorical attributes*. *Information systems*, 25(5), pp.345-366.
14. *Sharan, R. and Shamir, R.*, 2000, August. *CLICK: a clustering algorithm with applications to gene expression analysis*. In Proc Int Conf Intell Syst Mol Biol (Vol. 8, No. 307, p. 16).
15. *Jain, A.K., Murty, M.N. and Flynn, P.J.*, 1999. *Data clustering: a review*. ACM computing surveys (CSUR), 31(3), pp.264-323.
16. *Hartuv, E. and Shamir, R.*, 2000. *A clustering algorithm based on graph connectivity*. *Information processing letters*, 76(4-6), pp.175-181.
17. *Shi, J. and Malik, J.*, 2000. *Normalized cuts and image segmentation*. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8), pp.888-905.

18. Ng, A.Y., Jordan, M.I. and Weiss, Y., 2002. *On spectral clustering: Analysis and an algorithm*. In *Advances in neural information processing systems* (pp. 849-856).
19. Bezdek, J.C., Ehrlich, R. and Full, W., 1984. *FCM: The fuzzy c-means clustering algorithm*. *Computers & geosciences*, 10(2-3), pp.191-203.
20. <https://towardsdatascience.com/how-to-use-dbscan-effectively-ed212c02e62>

CHAPTER 8

Dimensionality Reduction

Every object is mathematically represented through vectors before the application of AI algorithms. Representing an object with a random vector is definitely a bad idea as it fails to capture the relationship among the data, if at all it exists. Most times, there do exist relationships between data that will be critical information for the AI algorithm to learn the required task. A critical task to perform before learning is to numerically represent the objects that capture their relationships.

With an increase in the count of attributes describing an object, dimensionality of the represented data increases. Most times, the represented data is sparse, resulting in the effects of “curse of dimensionality”, as explained in [chapter 1 Overview of AI](#). We can mitigate this effect to a certain extent by reducing the dimensionality of the data. The process of reducing the dimensionality of the data by preserving the required structure of the original data is called *dimensionality reduction*. This chapter will discuss algorithms that can be applied for reducing higher dimensions data to lower dimensions data.

Structure

In this chapter, we will cover the following topics:

- Principal Component Analysis
- Autoencoders
- t-Distributed stochastic neighbor embedding

Objectives

After going through this chapter, you will be able to understand dimensionality reduction algorithms like PCA, Autoencoder, and t-SNE. You will learn to apply these algorithms on the Iris dataset.

Reducing dimensionality

Dimensionality reduction provides various advantages. Feeding dimensionality reduced data will help AI algorithms learn faster with less computing resources and data sets. It is not just about feeding the data to AI algorithms; we may have to visualize the high-dimensional data. However, our senses restrict our visualization capabilities to the maximum of three dimensions. So, the higher-dimensional data must be reduced to three dimensions or lower for visualization through eye. While reducing the dimensions, we must preserve the relationships between the data points to the maximum extent possible. Reducing dimensionality of the data is not just for visualization purposes. Reducing dimensionality by removing correlated variables/features and the least-important variables/features results in reduction of noise, which further improves an AI model's accuracy.

Dimensionality reduction techniques to reduce dimensions of the data can be broadly classified into linear and non-linear.

- Linear reduction techniques perform linear transformation of high-dimensional data to low dimension. **Principal Component Analysis (PCA)**, **Linear Discriminant Analysis (LDA)**, and **Independent Component Analysis (ICA)** are a few examples of linear reduction techniques.
- Non-linear reduction techniques perform non-linear transformation. Non-linear dimensionality reduction algorithms include Kernel PCA, Autoencoder, t-SNE, and UMAP **Uniform manifold approximation and projection (UMAP)**.

Another way to broadly classify dimensionality reduction based on approach is feature selection and feature extraction:

- Feature selection approaches a select subset of the features or attributes that are more relevant for learning the task. Filter strategy, wrapper strategy, and embedded strategy are a few examples of feature selection approaches.
- In feature extraction or feature projection, high-dimensional data is transformed into lower-dimensional data. Technique used to transform data to lower dimensions may be linear or non-linear.

Dimensionality reduction algorithms are commonly used in signal processing, speech recognition, neuroinformatics, and bioinformatics. They are used as an intermediate step in noise reduction, data visualization, and cluster analysis. We will also come across many other applications and will discuss three-dimensionality reduction algorithms PCA, Autoencoder, and t-SNE in detail.

Principal Component Analysis

Principal Component Analysis (PCA) is a linear unsupervised dimensionality reduction technique that performs linear mapping of high-dimensional data to low-dimensional data by choosing new axes or basis vectors. Choosing appropriate basis vectors for the given data makes tasks easier.

Consider a few data samples, as shown in [Figure 8.1](#). From the left figure, we can visualize that two axes are necessary to represent the data with its original variability. Can we reduce this to one dimension? By just removing one axis out of two, we lose more information or variation in the data. To reduce data to one dimension, we must select an appropriate axis or a basis vector such that the variance of the original data is retained to the maximum extent. This is depicted in the right part of [Figure 8.1](#). One inclined basis vector is enough to capture most of the original data's variance, as shown in [Figure 8.1](#). This is how choosing appropriate axis or basis vector for the given data simplifies the given task. Refer to the following figure:

Figure 8.1: (left) plot of data samples with regular axis (right) choosing appropriate new axis for the data

Let us consider the data in n -dimensional space. We need to find maximum of n vectors called *principal components* such that all the following three conditions are satisfied:

- is unit vector and orthogonal to where .
- is the best-fitting line that minimizes the average squared distance from the data vectors to the line in the direction of .
- Principal components are numbered such that captures variance of n -dimensional data better than where .

These principal components can form an orthonormal basis for the linear transformation from n -dimensions to lower l -dimensions. Principal component analysis is the process of computing principal components and using them to linearly transform data from n -dimensions to lower l -dimensions with only the first k principal components. PCA reduces dimensions of the data by projecting each data vector on to only the first k principal components to obtain lower-dimensional data while preserving as much of the data's variation as possible.

Principal components can be computed by eigen decomposition of data covariance matrix or singular value decomposition of data matrix. We will consider the Iris dataset and apply eigen decomposition on its data covariance matrix to obtain principal components. We will consider a few of these obtained principal components to reduce the dimensionality of the Iris dataset.

Loading Iris dataset

The Iris dataset contains 50 samples from each of three species of Iris flower, namely, setosa, virginica, and versicolor. Every flower is represented by four features: sepal length, sepal width, petal length, and petal width in centimetres. Code to load the data set is captured in *Code 8.1*:

```
1. from sklearn import datasets  
2.  
3. def load_iris_data(num_rows=150):  
4.     iris = datasets.load_iris()  
5.     x = iris.data[:num_rows, :]  
6.     y = iris.target[:num_rows]  
7.     return x, y
```

Code 8.1: Loading Iris dataset

As each sample of the data is represented with four features, we will not be able to plot the samples in two dimensions. Instead, we will draw two plots with sepal's length/width and another with petal's length/width. These two plots are shown in *Figure 8.2*, where three species of Iris flower is plotted in three different colors:

Figure 8.2: (left) plotting of sepal's length and width (right) plotting of petal's length and width

Calculating covariance matrix

Now, we need to calculate covariance matrix from the loaded data x . Covariance matrix contains covariance between each pair of features. As the number of features of Iris data is four, the shape of covariance matrix would be 4×4 . Covariance between two features F and G can be calculated as follows:

Where n is the total number of observations, and x_{ij} represent values for features F and G during i observation. The Numpy library provides functions to calculate covariance matrix, as shown in *Code 8.2*, which is a continuation from *Code 8.1*:

1. `import numpy as np`
2. `x, y = load_iris_data()`
3. `# Covariance matrix`
4. `cov_mat = np.cov(x.T)`

Code 8.2: Covariance matrix calculation

Covariance matrix will always be symmetric. Output from the previous code would be as follows:

Decomposition of covariance matrix

Once covariance matrix is obtained, we need to decompose it with its eigen vectors. These eigen vectors will act as principal components for transforming data from higher to lower dimensions. *Code 8.3* provides the steps to calculate eigenvalues and eigen vectors using NumPy. Covariance matrix can then be decomposed as follows (discussed in [*chapter 2 Linear Algebra*](#)):

Code 8.3 provides the steps to verify the Eigen decomposition of covariance (continuation from *Code 8.2*):

```

1. # Eigen Decomposition of covariance matrix
2. eigen_val, eigen_vect = np.linalg.eig(cov_mat)
3. mat_p = eigen_vect
4. mat_p_inv = np.linalg.inv(mat_p)
5. # Diagonal matrix
6. mat_d = np.array([
7.     [eigen_val[0], 0, 0, 0],
8.     [0, eigen_val[1], 0, 0],
9.     [0, 0, eigen_val[2], 0],
10.    [0, 0, 0, eigen_val[3]]]
11. ])
12. # Matrix will be equal to covariance matrix
13. mat_obtained = np.matmul(np.matmul(mat_p, mat_d), mat_p_inv)

```

Code 8.3: Eigen decomposition of a symmetric square matrix

The obtained eigenvalues and corresponding Eigen vectors for the Iris dataset would be as follows:

We can verify that these eigen vectors are unit in length and are orthogonal. These eigen vectors form the orthonormal basis for the linear transformation.

Reducing with principal components

We can use all four eigen vectors for the linear transformation on the original data. However, the resulting vectors after transformation would still be in four dimensions. Instead, we can select the first two eigen vectors that captures top two maximum variations of the original data. Transformation matrix with the first two eigen vectors, also called *principal components*, would be as follows:

We can apply transformation on original data to obtain the transformed data in two dimensions, as shown in *Code 8.4* (continuation of *Code 8.3*):

1. # Select two principal components

```

2. eigen_vect0 = eigen_vect[:, 0]
3. eigen_vect1 = eigen_vect[:, 1]
4. trans_mat = np.array([eigen_vect0, eigen_vect1]).T
5. # Transform the data points
6. x_reduced = np.matmul(x, trans_mat)

```

Code 8.4: Transforming the data to lower dimensions

As the transformed data is in two dimensions, we can plot the points as shown in [*Figure 8.3*](#). With this, we have reduced the dimensionality of Iris dataset from four to two dimensions. From the figure, we can guess that simple linear separator algorithm would help us in classifying three species of Iris flow with good accuracy. Refer to the following figure:

Figure 8.3: Plotting of PCA transformed data

Variance retention

We can reduce the dimensions of the data by selecting a few principal components of data covariance matrix and performing linear transformation of original data with chosen orthonormal basis. A question would definitely arise in your mind about the number of principal components to choose. Each of the principal components retains certain variance of the original data. If all components are used, then all variance of the original data is retained. We can calculate the percentage of retention of the variance. Based on this percentage of variance retention, we can decide on the number of principal components.

Trace or sum of diagonal elements of the covariance matrix is equal to sum of its eigenvalues. The percentage retention of one eigen vector can be calculated using the following:

To calculate percentage retention of more than one Eigen vector, we must add its corresponding eigenvalues as follows:

On the Iris dataset, the first principal component would retain 92.5% of total variance. The first two principal components would retain 97.7% of total variance. Adding the third principal component would result in 99.5%. For the Iris dataset, we can comfortably reduce the dimensions from four to three by retaining 99.5% variance of the original data.

When to use PCA

AI algorithms assume that features are independent. If we want to make features independent, then PCA would be the right choice. With PCA, we will be able to reduce dimensions of the data as well if a few principal components are able to retain the variance we needed, which, of course, depends on the data.

In original data, each dimension represents a feature. These features convey meaning regarding the sample under consideration. However, when reduced through PCA, dimensions/features of the reduced data may not make sense to understand the sample. If you want to retain the meaning of each dimension, do not use PCA for dimensionality reduction.

Autoencoder

Autoencoders are non-linear unsupervised dimensionality reduction technique that learns the compact representation of the original data using neural networks. Simple autoencoder model is depicted in [*Figure 8.4*](#). Dimensions of input and output layer would always be same. The objective of the autoencoder model is to produce the output vector just like as the corresponding input vector while the data passes through hidden layer where dimensions are reduced. In [*Figure 8.4*](#), input and output dimensions are four, and there are total of three hidden layers between them. *Encoded layer* also called *bottleneck*, is the lowest dimension layer in the entire network. In this case, we have encoded the layer's dimension as two. Encoder model during training will learn compact representation of input data such that it is able to recreate the original data from the encoded data. Part of the encoder model that represents input data in compact representation through encoded layer is called *Encoder*. Part of the encoder model that creates original input data from the encoded data is called *Decoder*. Refer to the following figure:

Figure 8.4: Simple Encoder Architecture

There exists variation in the architecture. Regularized autoencoders such as sparse, denoising (creates corrupted copy of input by introducing noise) and contractive (encoded layer has lower dimensions as compared to input) are learning representation for classification tasks. Convolutional autoencoders uses convolution operation to learn convolution filters. Variational autoencoders finds its application in generative AI models. It assumes that data is generated by a model, and encoder is learning an approximation of posterior distribution. After training, sampling from the distribution, followed by decoding, will generate new data. In this section, we will implement simple sparse autoencoder for the Iris dataset.

Iris autoencoder

Let us build an autoencoder for the Iris dataset with encoded layer of two dimensions. As the Iris dataset contains four features, dimensions of input and output layer would be four. To keep it simple, let us have one hidden layer that will be the encoded layer. *Code 8.5* provides the steps to create TensorFlow autoencoder model:

```
1. from sklearn import datasets
2. from sklearn.model_selection import train_test_split
3. import tensorflow as tf
4. from tensorflow.keras import layers, losses
5. from tensorflow.keras.models import Model
6. from tensorflow.keras.initializers import RandomUniform
7.
8. class Autoencoder(Model):
9.     def __init__(self):
10.         super(Autoencoder, self).__init__()
11.         self.encoder = tf.keras.Sequential([
12.             layers.Dense(
13.                 units=2, activation='relu',
14.
kernel_initializer=RandomUniform(minval=0., maxval=1.,
15.                                         seed=10))
'
```

```

16.         ])
17.         self.decoder = tf.keras.Sequential([
18.             layers.Dense(
19.                 units=4, activation='relu',
20.
21.                 kernel_initializer=RandomUniform(minval=0., maxval=1.,
22.                                         seed=10))
23.
24.         ])
25.     def call(self, x):
26.         encoded = self.encoder(x)
27.         decoded = self.decoder(encoded)
28.         return decoded

```

Code 8.5: Simple autoencoder

We can now create an instance of this model class. Once it is created, we can train the model with the Iris dataset, as shown in *Code 8.6* (**load_iris_data()** function is defined in *Code 8.1*):

```

1. x, y = load_iris_data()
2. # Split the data into train & test
3. x_train, x_test, y_train, y_test =\
4.     train_test_split(x, y, random_state=10, test_size=.3)
5. # Training the model
6. autoencod_model = Autoencoder()
7. autoencod_model.compile(optimizer='sgd',
8.                         loss=losses.MeanSquaredError())
9. autoencod_model.fit(
10.    x_train, x_train, epochs=40, batch_size=30,
11.    validation_data=(x_test, x_test)
12. )

```

Code 8.6: Training the autoencoder with Iris dataset

After training is complete, we can pass the Iris samples and capture the corresponding values at encoded layer, as shown in *Code 8.7*:

```

1. # Obtain encoded information of the dataset
2. encoded_vect = autoencod_model.encoder(x).numpy()

```

Code 8.7: Reduced dimensionality of Iris dataset

As the encoded vector is two-dimensional, we can plot the vectors. This is shown in [*Figure 8.5*](#):

Figure 8.5: Plotting of Autoencoder reduced Iris data

There exist various algorithms to reduce dimensionality of data to two or three dimensions for visualization purposes. In the next section, we will discuss one of these algorithms called t-SNE.

t-SNE

t-Distributed Stochastic Neighbour Embedding (t-SNE) [1] is an unsupervised non-linear dimensionality reduction technique suitable for visualization of high-dimensional data in two or three dimensions. t-SNE calculates similarity measure between data in high-dimension and its corresponding points in low-dimensions, and then it optimizes these two similarities.

Let us consider a data set of n samples belonging to high dimensions space , which will be mapped to n samples of lower dimensions space where . Compute pairwise similarity in high dimensions. Then, find 's such that pairwise similarity measures in lower dimensions space are equal to their corresponding similarity measure in higher dimensions space.

Similarity between two samples is the conditional probability , that would pick as its neighbour (in higher dimension space) if neighbours were picked in proportion to their probability density under a Gaussian distribution centered at . It is defined as follows:

where is the variance of the Gaussian centered at . Methods to determine will be discussed later. As we are interested in only pairwise similarities, we can set . This pairwise similarities in high dimensions causes problem when is an outlier. For an outlier, probability value will be extremely small , due to which the location of corresponding mapped has little impact on cost function. Due to this, the mapped position of the outlier is not well determined. This problem can be overcome by defining the joint

probabilities in the high-dimensional space to be the symmetrized conditional probabilities:

This ensures that for , as a result every data sample, makes significant contribution to the cost function.

Now, let us compute similarities between mapped samples in low-dimensional space. In the high-dimensional space, we converted distances into probabilities using a Gaussian distribution. In the low-dimensional map, we can use a probability distribution that has much heavier tails than a Gaussian to convert distances into probabilities. This allows a moderate distance in the high-dimensional space to be faithfully modeled by a much larger distance in low dimensions. t-SNE uses Student t -distribution with one degree of freedom (Cauchy distribution) as the heavy-tailed distribution in the low-dimensional space. Using this distribution, the joint probabilities or pairwise similarities are calculated as follows:

Now, we have similarity measures or joint distributions in higher and lower dimensions. The next step would be to calculate a single *Kullback-Leibler divergence* between a joint probability distribution in the high-dimensional space and a joint probability distribution Q in the low-dimensional space:

Where . We must learn the parameters that optimizes this cost function. Gradient descent can be used to learn the parameters. Gradient function of this cost function would be as follows:

Choosing σ_i

Bandwidth of a Gaussian is controlled by value of . It is not likely that there is a single value of that is optimal for all samples due to variation in the data density. In dense regions, smaller value of is more appropriate than in sparser regions. The value of induces probability distribution over all data samples. This distribution has entropy that increases with . We can perform

binary search for value of σ such that resulting p produces perplexity specified by the user. Perplexity is defined as follows:

Where H is Shannon entropy of p measured in bits as follows:

Typical value of perplexity is range [5,50]. It can be interpreted as a smooth measure of an effective number of neighbors.

PCA vs t-SNE

PCA is a linear dimension reduction technique that seeks to maximize variance and preserves large pairwise distances. Data samples that are different end up far apart. This can lead to poor visualization, especially when dealing with non-linear manifold structures. Think of a manifold structure as cylinder, ball, or curve. t-SNE differs from PCA by preserving only small pairwise distances or local similarities, whereas PCA is concerned with preserving large pairwise distances to maximize variance. PCA and t-SNE preserve global and local structure of the data, respectively. Also, t-SNE has three hyperparameters, i.e. learning rate, number of steps, and perplexity, while PCA doesn't.

t-SNE on Iris Dataset

We can apply t-SNE dimensionality reduction on the Iris dataset using Scikit provided libraries. *Code 8.8* applies t-SNE on the Iris dataset (`load_iris_data()` function is defined in *Code 8.1*):

1. `from sklearn.manifold import TSNE`
2. `from sklearn import datasets`
3. `x, y = load_iris_data()`
4. `tsne = TSNE(random_state=10)`
5. `x_transformed = tsne.fit_transform(x)`

Code 8.8: t-SNE on the Iris dataset

Dimensionally reduced data is plotted in *Figure 8.6*. We can visually make out three clusters pertaining to each of the three species of Iris flower.

Figure 8.6: Plotting of t-SNE reduced Iris data

We have successfully reduced four dimensions Iris data to two dimensions data using *t*-SNE. There exists many other dimensionality algorithms that can be applied on Iris data. Choice of the algorithm should be based on the domain of the data.

Conclusion

This chapter introduced PCA, autoencoder and *t*-SNE dimensionality reduction techniques. One technique is preferred over others based on the domain or application. PCA is preferred when features are strongly correlated. For visualization of data, *t*-SNE is preferred as it captures non-linear relations among features. Autoencoder are preferred in image domain like image compression, image denoising. In next chapter we will discuss about computer vision algorithms that are used in image domain.

Further reading

Dimensionality reduction algorithms are being used in various fields, and reduction technique enhances the efficiency of many algorithms. We discussed only a few reduction techniques. Wikipedia provides really good source for further reading on dimensionality reduction algorithms. Many standard AI frameworks or libraries provide tutorial [2] and guides [3] about these reduction algorithms.

References

1. Van der Maaten, L. and Hinton, G., 2008. Visualizing data using *t*-SNE. Journal of machine learning research, 9(11)
2. Tutorial: <https://www.tensorflow.org/tutorials/generative/autoencoder>
3. https://scikit-learn.org/stable/auto_examples/manifold/plot_manifold_sphere.html#sphx-glr-auto-examples-manifold-plot-manifold-sphere-py

CHAPTER 9

Computer Vision

Living beings on Earth have evolved to understand the nature through senses. Sense organs that help humans to sense things are eyes (light reflected from the object), tongue (taste of the object), skin (touch of the object), ears (sound emitted from the object), and nose (smell emitted from the object). These senses feed information to the brain, which then processes the information for interpretation and action if necessary. Humans have tried replicating the performance of both senses and the brain.

Humans have developed cameras that work like eyes and capture information about light reflected by the objects and convert it to digital images/videos. **Computer Vision (CV)** deals with analysing the information captured by camera, like the brain. Recent advancements of algorithms in CV field are inspired by visual information processing of the brain. This chapter will discuss computer vision algorithms that help interpret digital images/videos that are primarily captured by camera.

Structure

In this chapter, we will cover the following topics:

- Digital images, pixels
- Geometric transformation
- **Filters/Kernels:** Spatial, Gaussian, Laplacian, Sobel
- Learning filters using **Convolution Neural Network (CNN)**
- Development of CNN
- Applications of CNN

Objectives

After going through this chapter, you will have clear understanding of the theoretical background behind the state-of-the-art AI models in computer

vision. Most of the CV models are CNN-based. However, they differ widely in their architecture. We will discuss the motivation behind these architectures. Understanding these topologies will help you come up with your own custom topologies that best suits your problem domain.

Digital Image Formation

Light is part of the electromagnetic spectrum that is sensed by eye, which is further divided into violet, indigo, blue, green, yellow, orange, and red. Colors perceived (by eye) by looking at the object are determined by the category of light reflected by an object. An object is perceived as white when it reflects all wavelengths of light equally. An object appearing green reflects green light and absorbs all other wavelengths of light. A black object absorbs all wavelengths of light. Now, let us understand how this information is captured in digital images.

Capture the light

Electromagnetic radiation reflected or emitted by the object is usually captured through two-dimensional array of sensors. Response of each sensor is proportional to the integral of radiation energy projected on to the surface of the sensor. Analog circuitry analyzes sensors output to produce analog signal. [Figure 9.1](#) shows capture of ellipse shaped object by analog circuitry. This analog signal is then digitized to produce digital image. Digitizing involves two processes sampling and quantization. Refer to the following figure:

Figure 9.1: Ellipse shaped object

Sampling and quantization

Output of most sensors are continuous voltage waveform whose amplitude and spatial variation is related to electromagnetic waves sensed that are reflected by the object. Two processes, sampling and quantization, are necessary to convert these analog signals to digital. Consider line AB on the captured object, as shown in [Figure 9.2\(a\)](#). Intensity of pixels along this line gradually increases and decreases as shown in [Figure 9.2\(b\)](#):

Figure 9.2: (a) A line on the object is considered (b) Intensity variation along line (c) Consider equally spaced samples along line (d) Sampled values along line

As AB line is continuous, we need to sample a few points along the line AB. The number of samples required would depend on how close you need the digital representation to be with respect to analog, as shown in [Figure 9.2\(c\)](#). The higher the number of samples, the closer would be the representation. This process of representing an image with sampled points is known as **sampling**.

Now let's represent complete ellipse with six sample points along x-axis and four along y-axis with total of 24 sample points, as shown in [Figure 9.3\(a\)](#). Each of these 24 sample points will be represented with integer intensity values in range [0,255], as shown in [Figure 9.3\(b\)](#). This is known as *quantization* of sample values. Outcome of sampling and quantization on ellipse would look like in [Figure 9.3\(c\)](#):

Figure 9.3: (a) sampling of intensity in 2D (b) Quantization of values (c) Resulting digitized image

In digitization of image, each of these cells that represented sampled value of real image are called *pixels*. In this example, ellipse image is represented using 24 *pixels* with 6 & 4 pixels along x and y axes, respectively. Count of the pixels used to represent an image is called its resolution. In this case, resolution of the ellipse image is mentioned as 6×4.

Pixels

In the digital world, images are presented using pixels. Pixel also called picture element can be understood as a minute area of illumination on a display screen. Higher count of pixels results in higher resolution of the image, and higher resolution results in better representation. Each pixel of the image will denote a value of the intensity in that position. Intensity values are represented with integer value in range [0,255]. To represent one pixel, one byte is sufficient. For grey ellipse image in the preceding example, $24=6*4$ bytes would be sufficient for digital representation of 24 sampled values. Programmatically, 2-dimensional unsigned byte array of size 6×4 is sufficient to represent this image.

Note: How many bytes are required to represent one color pixel?

Appearance of the color pixel is dependent on values of Red, Green, and Blue components. In general, range for each of these components is [0,255]. So, 3 bytes are required to represent one color pixel. This format of representation of the image is called RGB. Programmatically, 3D array of size $w \times h \times 3$) is required to represent color image of resolution $w \times h$.

Accessing pixels

Once image is represented as arrays, accessing the pixels is the same as accessing arrays. Numbering of pixels starts from the top-left corner of the image, as shown in [Figure 9.4](#). Representing an image using its spatial knowledge with use of pixels is useful for techniques that operate based on *spatial domain* knowledge.

Another important way of representation is Fourier transform of the image. This representation falls under *frequency domain* where image is represented as waves of various frequencies. Refer to the following figure:

Figure 9.4: Pixel's position

Spatial domain filtering is sufficient for a majority of tasks of computer vision. However, for many image processing and enhancement tasks, frequency domain filters have been used successfully. Algorithms developed in one domain can be successfully translated to another. In this chapter, we will focus mainly on spatial domain.

Spatial filtering

Filtering is the name used for modifying or rejecting specific components of an image through the use of mathematical operations. The process of applying a filter on an image represented in spatial domain is called **spatial filtering**. Spatial filtering modifies a pixel of an image by replacing it with function of the pixel value or its neighboring pixels values. We can broadly classify spatial operations into three broad categories:

- a. Single pixel operations

- b. Neighbor pixels operations
- c. Geometric spatial transformations

In *single pixel operation*, transformation function receives only one pixel input and outputs one value that will be used to replace the pixel under consideration. In *neighbor pixels operation*, transformation function accepts pixel under consideration along with its neighbors and outputs one values that will be used to replace the pixel under consideration. In *geometric spatial transformation*, modification is performed on spatial arrangement of pixels in an image.

Geometric spatial transformation

In **geometric spatial transformation**, transformation function accepts pixel coordinate (x, y) of original image and outputs new position coordinate (a, b) for the pixel in transformed image. Transformations in 2D that preserves points, straight lines, parallelism and planes is called *affine transformations*. *Affine transformation* preserves collinearity (that is, all points lying on a line initially still lie on that line even after transformation) and ratios of distances (e.g., the midpoint of a line segment remains the midpoint of that line segment even after transformation).

In this section, we will be concentrating on affine transformations that includes scaling, shearing, translation, and rotation. Transformation function relating to these operations can be expressed in matrix format. Each pixel of the original image is represented by (x, y) coordinate and (a, b) represents the new coordinates of the pixel in transformed image. The following transformation matrix is applied to every pixel position of the original image to obtain pixel coordinates in transformed image:

Let's perform few transformations on an image using this approach. Transformations that we will discuss are rotation, shear, scaling, and translation. Transformation matrices for these tasks will be of the form given as follows. Along with the form of the matrix there is an example for the respective form:

We will use the OpenCV library to apply transformation. Before applying the transformation, let us read the image file and convert to gray scale using the library, as shown in the following code:

```
1. import cv2 as cv
2. def read_image(path):
3.     image = cv.imread(path)
4.     # convert image to grayscale
5.     image_gray = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
6.     return image_gray
7. # IMAGE_PATH is absolute path of input image. Should be assigned.
8. image_gray = read_image(IMAGE_PATH)
```

Code 9.1: Reading image and converting to gray scale

We can use the **warpAffine()** function of OpenCV library. It accepts image and matrix of size 2×3 and applies the matrix to every pixel location of original image and outputs transformed image. Code to rotate the gray image obtained previously is captured in the following code. Output of this code is captured in [Figure 9.5\(b\)](#). Rotation operation could move pixels out of the defined view. We might need to enlarge the current view to visualize the rotated pixels. Following code provides steps to rotate an image.

```
1. import numpy as np
2. def rotate(image, angle_degree=20):
3.     (rows, cols) = image.shape[:2]
4.     radians = np.deg2rad(angle_degree)
5.     cos_val = np.cos(radians); sin_val = np.sin(radians)
6.     rotate_matrix = np.float32(
7.         [[cos_val, -1*sin_val, 0],
8.          [sin_val, cos_val, 0]])
8.     # applies transformation matrix to every pixel coordinate
9.     rotate_image = cv.warpAffine(image, rotate_matrix,
10.      (cols, rows))
10.    return rotate_image
```

```
11. rotate_img = rotate(image_gray)
12. cv.imshow("Rotated", rotate_img)
```

Code 9.2: Rotating image by 20 degrees

We can scale the image by using corresponding transformation matrix. The following code shows steps to apply scaling to the gray image. Output of this code is captured in [Figure 9.5\(d\)](#). Scaling operation changes only the aspect ratio of the original image. Following code applies scaling operation on an image.

```
1. def scale(image, scale_x=0.3, scale_y=0.8):
2.     (rows, cols) = image.shape[:2]
3.     scaled_matrix = np.float32([[scale_x, 0, 0], [0,
scale_y, 0]])
4.     scaled_img = cv.warpAffine(image, scaled_matrix, (cols,
rows))
5.     return scaled_img
6. scaled_img = scale(image_gray)
7. cv.imshow("Scaled", scaled_img)
```

Code 9.3: Scaling the image

Similarly, we can write code for application of other matrices by just modifying the transformation matrix and passing it to the **warpAffine()** function. Outputs of different transformation matrices specified before are captured in [Figure 9.5](#), along with parameter value used to create the matrix:

Figure 9.5: (a) Original gray Image (b) Rotated image (c) Sheared image (d) Scaled image (e) Translated image

Till now, we changed the position of the pixel in transformed image. Next, let us change the value of the pixel (intensity values) in the transformed image.

Neighbor pixel operation

Transformation function can consider the pixel and its neighboring pixels' intensity values to output an intensity value that will be used to replace the pixel under consideration in the transformed image. This transformation

function or the matrix representing the function is called *filter* or *kernel* or *filter kernel*. Process applying the filter or kernel to every pixel of the input image is called *filtering the image*. Filtering an image can also be viewed as extracting features from the image that helps to infer the image. Different filters offer various features of the image. If the transformation function/matrix is linear, then it is called *linear spatial filter*; otherwise, it is known as *nonlinear spatial filter*.

Consider simple *linear spatial filters* that perform sum-of-product operations on an image I with filter w . Let image I be 2-dimensional, with each pixel representing pixel intensity. Pixel of the image in position i is accessed using $I[i]$. Filter w can be of various sizes. For simplicity, let's consider 3×3 two-dimensional filter. Linear filter is applied to every pixel of the image to obtain transformed image G . *The following formula* shows the application of 3×3 kernel on a pixel at location i of an image.

Centre of the kernel aligns with the pixel under consideration. Kernel cells are numbered by keeping the origin at the center, as shown in [Figure 9.6](#). After alignment, product of respective cells values are calculated and summed. Resulting value is used for replacing the pixel value at location i . In this example, we considered kernel of size 3×3 , but kernel can be of any size. Usually, kernels with odd dimensions are chosen, like $k \times k$, where k is odd positive integer. The advantage is that after the alignment of the pixel of image under consideration and kernel origin, the number of surrounding pixels around pixel under consideration will remain equal along every direction. Refer to the following figure:

Figure 9.6: Kernel cells and part of image under consideration to change a pixel value

Tip: What happens when all the surrounding pixels are not available, like in case of border pixels?

If a pixel doesn't have surrounding pixels that maps to kernel size, then we can use a constant value instead of the missing surrounding pixels. This is called padding.

This linear operation is called *correlation* or *cross-correlation*. For a kernel of size $m \times n$ where $m = 2a + 1$ & $n = 2b + 1$, *correlation* is defined as follows:

Another important type of linear operation is called convolution. *Convolution* operation is defined as follows:

Convolution operation can be defined in terms of correlation as rotate kernel by 180° and perform correlation operation. [Figure 9.7](#) depicts the cells of kernel that get multiplied with pixels of image:

Figure 9.7: Mapping of neighboring pixels with kernel cells for linear operation

Convolution properties

Convolution operation follows commutative, associative, and distributive properties.

As convolution operation follows commutative property, during operation , either rotate kernel w or image I by 180° before performing correlation. Consider a situation where L kernels are convoluted on image I one after another. As convolution follows commutative property, instead of performing convolutions with L kernels, we can perform with one kernel w , which is obtained as .

Note: Correlation or cross-correlation follows distributive property only.

Separable kernels

Two-dimensional function is said to be *separable* if it can be expressed as the product of two one-dimensional functions as . Kernel in two-dimensions

is a matrix and if it can be expressed as outer product of two one-dimensional vectors that has dimensions of as , then it is called *separable kernel*.

Convolution with separable kernels

Let's understand computation required to perform convolution operation. Consider image I of size and kernel W of size . One convolution operation requires multiplications and additions. This operation must be applied on every pixel of the image, resulting in total of multiplications and additions.

Can the computation be reduced with separable kernels? Yes, reduction is possible through use of separable kernels. Let the kernel of size be expressed as outer product of two one-dimensional vectors that has dimensions of as . Computation required to apply kernel on complete image is . Application of another kernel would cost . In total, computation cost of applying kernels on complete image is . This provides computation advantage that can be defined as follows:

Tip: How do we determine whether kernel is separable?

Rank of kernels is always 1 due to its dimension . Product of these rank 1 kernels would always result in rank 1 (Reason: Matrix multiplication AB can be expressed as a linear combination of columns of A using weights from columns of B , refer to [Chapter 2 Linear Algebra](#)). In this case, kernel is separable if and only if its rank is 1.

Example: Consider filter of size 3×3 that can be expressed as the product two 3×1 filter. Check their ranks:

Example: Consider another filter of size 3×3 called Sobel (more details in further sections). Check their rank:

There exists different filters or kernels that help extract distinct features of the image through convolution operation. These features help in inferring

the image. Among these, popular ones are smoothing filters like Gaussian and edge detection filters like Laplacian & Sobel, which will be discussed in the following sections.

Gaussian kernel

Gaussian function in one dimension using standard deviation σ and zero mean , is expressed as follows:

Distribution of Gaussian function values around mean are symmetric. [Figure 9.8](#) shows plot of Gaussian function without normalization factor . 68% of values fall under distance from mean. Within distance of σ from mean, 95% of values are covered. account for 99.7% of values. This information is vital for designing Gaussian kernel of fixed length. Another important property is that Gaussian function is never equal to zero. This property is useful in filtering operation as higher weightage is given to nearer pixels and is symmetric in all directions. Refer to the following figure:

Figure 9.8: Distribution of values in Gaussian function

To work with image, we need Gaussian function in 2D and its discrete approximation in form of 2D matrix. As 99% of the values are covered with in distance, kernel can have values within this distance. Due to this, kernel of size (each in opposite directions of one axis) is sufficient to capture the function behavior.

In two-dimension, Gaussian function is product of Gaussian functions along each dimension. Product of Gaussian functions is Gaussian function. Kernel obtained from Gaussian function are separable. Here, x denotes distance from origin along the x . Similarly, y denotes distance from origin along y .

Discrete approximation of Gaussian function

Now, let's obtain discrete approximation of Gaussian function that can be used as kernel in convolution operation on images. To make things simple,

let us consider zero mean and standard deviation as . With these values, function becomes the following:

Consider only the variable part of the function as for writing the code. The following code shows the implementation. It accepts coordinate values from 2D and outputs variable part of Gaussian function value: Following code shows implementation of Gaussian function.

```
1. import numpy as np
2. def gaussian_fn_2d(x, y ):
3.     # Exponent part of gaussian function mean=0, sd=1
4.     exp_part = np.exp(-(np.power(x, 2.) + np.power(y, 2.)))
/ 2.)
5.     return exp_part
```

Code 9.4: Gaussian function in two dimensions

As discussed, considering matrix is sufficient. We need odd matrix, so for , we should consider 7×7 matrix. Function parameter values consider for discretization should be within 3σ distance from origin. We can consider values, as shown in [Figure 9.9\(a\)](#), for the kernel matrix. These coordinate values are used to kernel matrix, as shown in the following code:

```
1. # 3 sigma distance from origin
2. X_START = Y_START = -3
3. X_STOP = Y_STOP = 3
4. def get_gaussian_kernel():
5.     kernel_shape = (X_STOP-X_START+1, Y_STOP-Y_START+1)
6.     gaussian_sample = np.zeros(shape=kernel_shape,
dtype=float)
7.     ker_x = ker_y = 0
8.     for y_idx in range(Y_START, Y_STOP+1):
9.         for x_idx in range(X_START, X_STOP+1):
10.             gaussian_sample[ker_x][ker_y] =
gaussian_fn_2d(x_idx, y_idx)
11.             ker_y = ker_y + 1 # updating index
12.             ker_x = ker_x + 1 # Updating index
13.             ker_y = 0
14.     return gaussian_sample
```

```

15. # Obtain gaussian kernel and normalize
16. gaussain_ker = get_gaussian_kernel()
17. div_part = 2. * np.pi
18. norm_gauss_kernel = (1./div_part) * gaussain_ker

```

Code 9.5: Obtain Gaussian function value for few coordinates equidistant from origin

Obtained kernel matrix using the code is captured in [Figure 9.9\(b\)](#). Note that it is not normalized. Refer to the following figure:

Figure 9.9: (a) (x,y) coordinates considered for creating 2D kernel (b) Gaussian kernel with $\mu=0$, $\sigma=1$ without normalization

Normalization of the preceding Gaussian kernel can be performed in two ways. One approach is to divide all values of the kernel by the constant part of Gaussian function, that is, (case: $\mu=0$, $\sigma=1$). All values of the kernel may not add to $(\mu=0, \sigma=1)$ due limited kernel size. We can follow another approach, in which we can divide all values of the kernel by the sum of all elements of the kernel. The second approach is better.

We have to use normalized Gaussian kernel for convolving with images. Normalized Gaussian kernel is captured in [Figure 9.10](#). Note that the sum of all elements of normalized kernel must equal 1. Refer to the following figure:

Figure 9.10: Normalized Gaussian Kernel with $\mu=0$, $\sigma=1$

Application of Gaussian filter

We have obtained normalized Gaussian kernel of size 7×7 that Gaussian function of zero mean as and standard deviation as . Now, let's apply the kernel on gray image. Gray image can be obtained with *code 9.1*. Apply normalized Gaussian kernel on gray image, shown in [Figure 9.11](#), to obtain smoothed image:

Figure 9.11: Snapshot of a Wikipedia page in gray scale

The following code shows the application of the kernel using OpenCV library:

```
1. image_gray = read_image(IMAGE_PATH)
2. gauss_filtered_img = cv.filter2D(
3.                 image_gray, ddepth=-1,
kernel=norm_gauss_kernel)
4. cv.imshow("Gaussian Smoothed Image", gauss_filtered_img)
```

Code 9.6: Convolve normalized Gaussian kernel with every possible pixel of the image using `filter2D()`

Output of the *Code 9.6* is captured in [Figure 9.12](#). Figure is smoothed using Gaussian kernel. Refer to the following figure:

Figure 9.12: Smoothing of gray Wikipedia image with Gaussian 7×7 kernel of $\mu=0, \sigma=1$

We can apply Gaussian kernel repeatedly too. Image is blurred further with each application of Gaussian kernel.

Note: Working with gray scale images is simple as it can be represented in 2D and values represent intensity. Gaussian filter is applied to intensity of these pixels. How to apply the filter on color images, which are usually represented in 3D using Red, Green & Blue? Gaussian filter cannot be applied to the color components. Instead, image is converted to YUV format where Y is luma/intensity, U & V represents blue and red channels. Gaussian kernel will be applied on Y component only.

We saw that image blurring/smoothing is accomplished in spatial domain with the use of neighborhood pixel averaging technique (Gaussian function). There are other blurring techniques, which will be discussed later in the *Non-Linear Filters* section. Next, we will create edge detection filters using differentiation. Differentiation at a point is proportional to magnitude of intensity discontinuity. Thus, image differentiation enhances edges and other discontinuities with respect to slowly varying intensities.

Image derivative-based kernels

Let us first understand the behavior of derivatives on constant intensity, onset and end of step and ramp intensity discontinuity/ramps. [Figure 10.13](#)

has plot of pixels intensity and depicts intensity ramp and step.

In image, along one dimension, pixels neighboring to pixel x are $x+1$ & $x-1$. Using this information, basic definition of first order derivative for one-dimensional function is stated as follows:

Properties of first order derivative are as follows:

- Zero in area of constant intensity
- Non-zero at onset and end of intensity ramp/step
- Non-zero along intensity ramps

Similarly, definition of second order derivative can be stated as follows:

Properties of second order derivative are as listed here:

- Zero in on areas of constant intensity
- Non-zero at onset and end of intensity ramp/step
- Zero along intensity ramps

Figure 9.13 shows first and second order derivatives of intensity values. We can verify the properties these derivates. Derivatives provide information about constant intensity, ramp up/down, and step intensity. Edges in digital images behave like ramp intensity. Identifying these ramps will help us in finding edges that will aid in sharpening the image. Next, we will use second order derivative for image sharpening. Refer to the following figure:

Figure 9.13: Graph captures pixel intensity along 1D, table captures derivatives for each pixel

Note: Due to the property “zero in area of constant intensity”, derivative kernels must sum up to zero.

Laplacian kernel – Second order derivative

The simplest derivative operator that is also isotropic (applies equally in all direction) is *Laplacian*, and it is defined for function with two variables as

follows:

As derivatives of any order are linear, Laplacian is linear operator. We need to express the operator in discrete format for 2D images, as follows:

Discretized Laplacian can be written as follows:

Kernels that satisfy this equation is captured in [Figure 9.14](#). Kernels (a) & (b) in the figure do not consider diagonal cells. When diagonal pixels are included, we obtain kernel as depicted in (c) & (d). As these kernels must satisfy the preceding equation, take negative of the kernel that satisfy the equation is also the valid kernel. Kernels (b) & (d) represents negative counterpart of (a) & (c). Refer to the following figure:

Figure 9.14: Laplacian Kernels (a)(b) without diagonal cells (c)(d) with diagonal cells

Laplacian kernel highlights sharp intensity transition and de-emphasis slowly varying intensities areas of the image. This produces images with grey lines and other discontinuities, all superimposed on dark featureless background. Background features are recovered along with sharpening effects by adding Laplacian image to the original.

Value of will depend on kind of Laplacian kernel used. for kernel (a) & (c) and for kernel (b) & (d) of [Figure 10.14](#).

Note: *Laplacian is sensitive to noise. To counter this, image is often smoothed with Gaussian function before the application.*

Let us apply Laplacian kernel of [Figure 9.14\(c\)](#) on [Figure 9.12](#) (Gaussian smoothed image). We can use OpenCV library's *filter2D()* function for filtering the image (similar to application of Gaussian kernel). The output of this is depicted in [Figure 9.15](#). We can see that edges are clearly visible.

Tip: Why should derivative kernels be applied on smoothed image?

Derivative expects function to be smooth for good behavior. To remove sharpness (noise) of the image, one needs to apply smoothing filters before applying derivative filters.

Refer to the following figure:

Figure 9.15: Laplacian kernel of size 3×3 on Gaussian smoothed image mentioned in [Figure 9.12](#)

To obtain image along with background, we subtract the obtained image for original image. The output of this operation is captured in [Figure 9.16](#):

Figure 9.16: Laplacian image subtracted from original image

We can combine Gaussian and Laplacian filters. As convolution is associative, we can convolve Gaussian and Laplacian kernels before applying on the image. This is termed as **Laplacian of Gaussian (LoG)**. Two-dimensional *Laplacian of Gaussian* function with mean is stated as follows:

The following code implements this function with $\mu=0$ & $\sigma=1$:

```
1. def laplacian_of_gaussian_2d(x, y):  
2.     # Exponent part of gaussian function mean=0, sd=1  
3.     power_part = (np.power(x, 2.) + np.power(y, 2.)) / 2.  
4.     exp_part = np.exp(-power_part)  
5.     prod_part = (-1./np.pi) * (1. - power_part)  
6.     return prod_part * exp_part
```

Code 9.7: Laplacian of Gaussian function with $\mu=0$, $\sigma=1$ implementation

To generate LoG kernel of size 7×7 , use function **get_gaussian_kernel()** as stated before, but call **laplacian_of_gaussian_2d()** instead of **gaussian_fn_2d()** with parameter values as used in Gaussian kernel generation. Once executed, we obtain LoG kernel, as shown in [Figure 9.17](#):

Figure 9.17: LoG kernel with $\mu=0$, $\sigma=1$

Gradient kernels must sum (elementwise sum) to zero. Sum of the kernel elements in this case is -0.00817. Subtract this value from center of the kernel only. Now, the sum of kernel elements will equal zero. Value of LoG kernel elements are low when compared to values of pixels whose range is [0,255]. To have a good impact of filtering, we must scale the kernel values by constant factor. In this case, let us multiply all elements of kernel by value 10. This would provide us the kernel mentioned in [Figure 9.18](#):

Figure 9.18: Laplacian of Gaussian Kernel $\mu=0$, $\sigma=1$

Tip: Multiplying all elements of the kernel with constant value doesn't alter its properties. Larger values in kernel will have high impact on the image when filtered.

We can now apply the LoG kernel on the image. Result of LoG filtering is captured in [Figure 9.19](#). We can see that this LoG kernel has captured lines much better than previous 3×3 Laplacian kernel. Refer to the following figure:

Figure 9.19: Application of LoG 7×7 kernel on original gray image

Now, subtract this image with the original image to obtain sharpened image with background, as shown in [Figure 9.20](#):

Figure 9.20: Subtract LoG kernel filtered image with original gray image to obtain background

Sobel kernel: First order derivative

Laplacian kernel discussed so far is second order derivative that measured change of slope. While Sobel kernel is first order derivative that measures the slope and combines Gaussian smoothing like LoG.

Sobel kernel consists of two kernels that calculate approximations of derivative along horizontal and vertical axis. OpenCV library can be used to obtain Sobel filter of various sizes. [Code 9.8](#) shows the steps to obtain

Sobel filter. **getDerivKernels()** outputs two vectors of size $dim \times 1$, these two need to multiplied to obtain separable vector:

```
1. def get_sobel_kernel(dim=3):
2.     sobelx_sep = cv.getDerivKernels(1, 0, dim,
normalize=True)
3.     sobelx = np.outer(sobelx_sep[0], sobelx_sep[1])
4.     sobely_sep = cv.getDerivKernels(0, 1, dim,
normalize=True)
5.     sobely = np.outer(sobely_sep[0], sobely_sep[1])
6.     return sobelx, sobely
```

Code 9.8: Obtain Sobel filter using OpenCV library

Sobel filter of 3×3 obtained using the preceding codes is captured as follows. To obtain kernel without normalization, set *normalize* to False in the code:

We can apply these normalized filters on Wikipedia page (using OpenCV library's *filter2D()* function as before). The result of application of horizontal kernel is captured in [Figure 9.21](#); it highlights all horizontal gradient directions:

Figure 9.21: Horizontal Sobel filter applied

The result of application of vertical kernel is captured in [Figure 9.22](#); it highlights all vertical gradient directions:

Figure 9.22: Vertical Sobel filter applied

We have discussed linear filters like Gaussian and Gradient based filters in detail. Now, let us discuss few non-linear filters in brief, like average and median filters.

Non-linear filters

Median filter is non-linear filter used to remove noise (effective on ‘salt-and-pepper’ noise: it is a form of a noise caused by sharp and sudden disturbances and gets reflected as black and white pixels spread sparsely over the image) in the image. The *median filter* of size $n \times n$, when applied on a pixel, replaces it with median of the pixel and its neighboring pixels (median of $n \times n$ pixels) with center of kernel aligned with the pixel under consideration. Filter is slid throughout the image. To cover border pixels, *border replicate* padding is normally used where values are padded at the border with the nearest pixel value.

Average filter, when applied on a pixel, replaces it with average of the pixel and its neighboring pixels (average of $n \times n$ pixels) with the center of kernel aligned with the pixel under consideration. As for all other filters, this filter is applied on an image by sliding the filter over every possible pixel.

Similarly, *maximum (minimum) filters* replace pixel value with maximum (minimum) value among the $n \times n$ pixels (pixel and its neighbors) values. This non-linear filter is widely used in networks that infer images.

We discussed a few filters in spatial domain and their use in extracting various features of an image. These features help us to perform required manipulation/classify the images. Curating filters for various classification tasks is not straightforward. In fact, we may not be able to identify all filters that help in performing classification task. Due to these complications, researchers thought of learning these filters based on classification task.

Learning filters

Deep Neural Networks (DNN) perform a good job in automatically learning the features based on the classification task. Can we combine DNN and convolution operations to automatically learn filters/kernels based on the image modification/classification task? Yes, networks that combine these two operations are called **Convolution Neural Networks (CNN)**.

Note: *Can we feed images as input to DNN for image classification task? Yes, we can. Input layer dimension of DNN would equal to $w \times h \times d$ (dimension of image). For 224×224 RGB image would need $224 \times 224 \times 3 = 150,528$ input layer dimension. Parameters to learn increases further with deep layers. So, due to higher dimension, DNN may fail to learn the parameters (curse of dimensionality).*

Convolution Neural Networks

To overcome large parameter space of DNN in image processing, convolution operation that shares parameters across the input data is utilized. This parameter sharing of convolution operation, along with DNN, have provided a boost to the image inferencing algorithms, and these networks are called **Convolution Neural Networks (CNN)**. Main building blocks of CNN are convolution, pooling (subsampling), and **fully connected (FC)** layers.

CNN consists of more than one convolution and pooling layers. Usually, convolution layer is followed by one pooling layer. This combination layers repeat to extract better features. At the end of the network, there exist FC layers that helps in classification based on the features extracted using convolution and pooling layers, as depicted in [Figure 9.23](#):

Figure 9.23: High level architecture of CNN

Convolution layer

This is the core building block of CNN where convolution operation is performed. Majority of the network's computation occurs in this layer. Suppose the input to the convolution layer is of dimension $I \times J \times K$. There are n kernels of various sizes but with the same depth d to be applied in this layer, as shown in [Figure 9.24](#).

Each kernel is convolved with input block to produce one 2D *feature map*. Kernel is slid ($w \times h$ plane) from the top-left corner to the bottom-right corner on the input for convolution. Sliding of the kernel can be either one pixel or more. This is called *stride*. Number of pixels slid/skipped in a particular direction will be *stride* value in that particular direction. Stride is 1 if kernel is moved to the right or below by one pixel for the next convolution operation.

Cells ($w \times h$ plane) along the border of the input block cannot be aligned with the center of the kernel due to the absence of cells mapping to the cells of kernel. Convolution operation cannot be performed on every cell of the input block, which leads to reduced size of feature map (dimensions less

than $w \times h$). To avoid this, we can pad constant value (usually 0 value) along the border of input block such that center of the kernel can be aligned with every cell in $w \times h$ plane. This is called *padding*.

There are three popular ways in which values can be padded:

- **Valid padding:** In valid padding, convolution operation is performed only if there exists one to one mapping of the cells between input and kernel. This reduces output size as valid convolution is not possible on every cell of the input.
- **Same padding:** In the same padding, input is padded with values such that the size of the output is the same as of input ($w \times h$ plane).
- **Full padding:** In full padding, input is padded with values such that the size of the output will be more than the size of the input ($w \times h$ plane).

Generic formula to know the output size , for input size of after convolution is as follows. Kernel is of size and stride is of S. & represents left and right padding along the width, and represents top and below padding:

Refer to the following figures:

Figure 9.24: Convolution layer in CNN

Example: Consider input block of and kernel of 3×3 . There is no need to consider depth, as depth of input block and kernel will be equal. In case of valid padding, stride S=1 would output 13×8 and S=2 would output 7×4 . In case of the same padding, S=1 would output 15×10 and S=2 would output 8×5 .

Output of convolution operation (feature maps) is normally fed to activation function like ReLU (discussed in [Chapter 7 Neural Networks](#)). Output of the activation function is fed to pooling layer that acts as subsampling to further reduce computation or parameters.

Pooling layer

Pooling layer reduces spatial dimension of input features, which, in turn, helps in the reduction of trainable parameters, resources, and computing time. This layer helps in extracting dominant features that are rotational and positional invariant. Maximum and average non-linear filter is usually used in this layer.

Consider part of the image of dimension 3×3 , as shown in [Figure 9.25\(a\)](#). *Max pooling* (application of maximum filter) would output 8 and *average pooling* would output value 4.3. This window of 3×3 is then moved, like kernel sliding in convolution operation. The concept of stride is applicable here as well. The formula to calculate the output dimension of the pooling layer is the same as the convolution layer discussed earlier. Refer to the following figure:

Figure 9.25: (a) Pooling applied on 3×3 size, (b) spatially separable convolution

CNN will have many layers of convolution and pooling. At the end, output would be fed to fully connected layers. *Fully connected layers* are the same as discussed in [Chapter 7 Neural Networks](#).

Convolutions operation with parameter sharing kernels is still significant in terms of computation resources and parameters count when depth/channels input to the layer increases. The number of parameters and computation resources can be reduced with use of separable kernels discussed earlier.

Spatially separable convolution

Spatially separable convolution does not perform convolution with $n \times n$ kernel. Instead, it breaks the kernel into two kernels of size $n \times 1$ each. Spatially separable convolution then applies convolution on input with one $n \times 1$ kernel; the output of this is again convolved with the second $n \times 1$ kernel. The concept is the same as explained in separable kernels. This is depicted in [Figure 9.25\(b\)](#). As every kernel is not separable, spatially separable convolution cannot be performed with every kernel. Kernel's rank must be 1 for it to be separable.

Depthwise separable convolution

Spatial separable convolution failed to exploit the depth/channels of input data. As the number of channels increases, computation resources would substantially increase. Depthwise separable convolution provides solution.

Depthwise separable convolution breaks the operation into two stages: filtering stage or depthwise convolution, and combination stage or pointwise convolution. Let us consider the input data of dimensions . Consider the application of one kernel. The depth of this kernel must be , which is equal to input data depth/channels. Dimension of this kernel will be .

Depthwise convolution

In **depthwise convolution** stage, instead of one kernel of size, kernels of each dimension is used. Each of these kernels is applied on only one channel of the input data. Output of this stage is of depth equal to input data's channel count (depth). In this stage, *Multiplication Count* = and *Parameters Count* = . Refer to the following figure:

Figure 9.26: Depthwise separable convolution

Pointwise convolution

Pointwise convolution uses kernel of size . Due to kernel dimension of 1×1 , it is called pointwise convolution. Depth of the kernel used in this stage is equal to channel count or depth of input data. Depth of output data after this operation is 1 for one kernel. To obtain more channels in output, we should use the same number of kernels of size in this stage. In this stage, and .

Optimization

Let us understand the benefit of this convolution with respect to regular convolution. Assume that we need output with channels or depth. Regular convolution requires kernels of size . This would result in and .

To obtain output with channels or depth using depth-wise separable convolution, and

Optimization ratio for multiplications count would be as follows:

Optimization ratio for parameters count would be as follows:

Convolution and pooling layers either reduce/retain spatial dimensions (height/width) of the input data. This reduction also called *downsampling* is beneficial for classification task but is of little help in other image tasks, like object detection or localization. For these tasks, it would be beneficial if the spatial dimension of input is increased, also called *upsampling*. Next, we will discuss the type of convolution operation that will perform upsampling: *transposed convolution*.

Upsampling: Transposed convolution

To understand transposed convolution, let us take input data of dimensions $2 \times 2 \times d$, where d is the number of channels or depth and a kernel of size $2 \times 2 \times d$ whose parameters are being learned during the training phase. Every cell of the input data is multiplied with all cells of the kernel to produce an intermediate result corresponding to each cell of the input. It is then placed in enlarged dimensions with provided stride to get intermediate results. In this case, we have considered $stride=1$. To obtain the final output, these intermediate results are added as shown in [Figure 9.27](#):

Figure 9.27: Transposed convolution with input and kernel of size $2 \times 2 \times d$ with $stride=1$

Increasing kernel size or stride will result in higher dimensions of the output as compared to the input data. With $stride=1$, input size $2 \times 2 \times d$ is increased to $3 \times 3 \times d$. With the same input data and kernel, $stride=2$ would result in output of size $4 \times 4 \times d$, as shown in [Figure 9.28](#):

Figure 9.28: Transposed convolution with input and kernel of size $2 \times 2 \times d$ and $stride=2$

We discussed an upsampling approach that had parameters to be learned during the training phase. There exist upsampling approaches that do not need parameters to be learned.

We discussed building blocks of CNN. Building blocks or tricks of DNN that help in training to learn better representation will be used in CNN too. These have already been discussed in [Chapter 7 Neural Networks](#). Next, let us discuss the development of CNN architectures.

Development of CNN

First network that had most of the building blocks of today's CNN was published in 1998 by Yann LeCun called *LeNet* [3]. There were a few challenges, like availability of robust data set, compute resources due which the development of CNN models were stalled for 14 years after this submission. Data set availability issue was addressed by ImageNet dataset in 2010. It started annual competition, which is now known as **ImageNet Large Scale Visual Recognition Challenge (ILSVRC)**. As part of this challenge in 2012, Alex Krizhevsky submitted *AlexNet* [4] that won the competition by beating the runners up with a huge margin of 10.8%. This received a lot of attention from researchers in CNN models and acted as the turning point in the development of CNN models.

In the next few years, many CNN models were proposed each with a unique way to enhance performance. Important models are **Visual Geometry Group (VGG)** (2014) [6], Inception (2014) [5], ResNet (2015) [7], Xception (2015) [8]. Recent CNN models include most of the tricks or proposals from these networks, along their novelty. Let us discuss a few of these models to understand the working of CNN.

AlexNet

AlexNet [4] is a simple CNN model. This architecture would be discussed in detail to understand the complete working of CNN models. AlexNet used five convolutional layers for feature extraction, three max pooling layers for subsampling, and three fully connected layers at the end to classify based on extracted features. It introduced activation function **Rectified Linear Unit (ReLUs)** and used dropout (explained in [Chapter 7 Neural Networks](#)) for training.

Architecture of AlexNet is depicted in [Figure 9.29\(a\)](#). Input to the network is the image of dimension $224 \times 224 \times 3$. In the first layer, convolution operation occurs with 96 kernels of 11×11 with valid padding and stride of

4. Applying the formula discussed earlier, the output dimension of this layer is $54 \times 54 \times 96$.

Depth (channels) is equal to the number of kernels used. It is then fed to the ReLU activation layer (no change in dimension). Next, in the pooling layer, max-pooling kernel of 3×3 is applied with a stride of 2 and valid padding. Using the formula discussed earlier, the output of this layer would be $26 \times 26 \times 96$.

The next layer is convolution, which applies 256 kernels of 5×5 () with stride of 1 and the same padding (). Just like before, we can apply the formula to obtain output dimension. Similarly, we can analyze the layers. The output of the last pooling layer is fed to the **Fully Connected (FC)** layer of 4096 size after flattening the data. Data then passes through two more FC layers before being fed to the Softmax layer. This outputs probability of the input belonging to a class. Refer to the following figure:

Figure 9.29: (a) AlexNet Model (b) Parameters and output dimension of each layer of AlexNet in TensorFlow framework

TensorFlow Model

TensorFlow code to create AlexNet is captured in the following code, and the output of this code is captured in [Figure 9.29\(b\)](#):

```
1. from tensorflow import keras
2. alexnet = keras.models.Sequential([
3.     keras.layers.Conv2D(filters=96, kernel_size=(11,11),
4.                         strides=(4,4), activation='relu',
5.                         input_shape=(224,224,3),
padding="valid"),
6.     keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
7.     keras.layers.Conv2D(filters=256, kernel_size=(5,5),
strides=(1,1),
8.                         activation='relu', padding="same"),
9.     keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
```

```

10.      keras.layers.Conv2D(filters=384, kernel_size=(3, 3),
11.                             activation='relu',
12.                             padding="same"),
13.      keras.layers.Conv2D(filters=384, kernel_size=(3, 3),
14.                             activation='relu',
15.                             padding="same"),
16.      keras.layers.Conv2D(filters=256, kernel_size=(3, 3),
17.                             activation='relu',
18.                             padding="same"),
19.      keras.layers.MaxPool2D(pool_size=(3, 3), strides=
(2, 2)),
20.      keras.layers.Flatten(),
21.      keras.layers.Dense(4096, activation='relu'),
22.      keras.layers.Dropout(0.5),
23.      keras.layers.Dense(4096, activation='relu'),
24.      keras.layers.Dropout(0.5),
25.      keras.layers.Dense(1000, activation='softmax')
26.  ])
27. print(alexnet.summary())

```

Code 9.9: TensorFlow keras code to create AlexNet CNN model

Counting trainable parameters

The first convolution layer of the network performs convolution with 96 filters of size 11×11 . The depth of these kernels is equal to depth of input data dimension to this layer. As input dimension to the first layer is $224 \times 224 \times 3$, the depth of all kernels in the first layer would be 3. The number of trainable parameters in first layer would equal to the following:

Bias is single learnable parameter for every kernel. As there were 96 kernels in this layer, learnable parameters for bias are 96.

Activation ReLU and max pooling layer don't have trainable parameters. Trainable parameters for the next few convolution layers can be calculated

as before. We can verify the calculation with the output of TensorFlow code in [Figure 10.26b](#). Output of last max pooling layer is flattened, which outputs tensor of size 6400, which is fed to dense (fully connected) layer of size 4096. Trainable parameters for this dense layer are calculated as follows:

Note: What are the advantages of having deeper CNN networks?

Initial convolution layers learn filters to detect local patterns. Deeper convolution layers learn filters hierarchically based on previous convolution layers learning. Due to this, deeper convolution layers learn complex and large patterns. This helps to solve complex tasks.

[Inception](#)

Inception V1 [5], also called GoogLeNet, submitted from Google was the winner of the ILSVRC 2014. It achieved a top-5 error rate of 6.67%. The main hallmark of this architecture is the improved utilization of the computing resources inside the network. This was achieved by a carefully crafted design that allows for increasing the depth and width of the network while keeping the computational budget constant. To optimize quality, the architectural decisions were based on the *Hebbian principle* [9] and the intuition of multi-scale processing. It introduced an inception module that allows us to use multiple types of filter size, instead of being restricted to a single filter size, in a single layer, which will then be concatenated and passed on to the next layer. This inception module, which optimizes the convolution layer, helped the network to go deep. Inception module is depicted in [Figure 9.30](#):

Figure 9.30: Inception V1 Module (Source: [5])

Inception module reduces computation by applying dimensionality reduction and projection. 1×1 convolution helps in dimensionality reduction that removes computational bottlenecks. It improves the utilization of computing resources inside the network.

In 2015, a team submitted Inception-V3 [10] that makes convolution layer's computation much more efficient using the concept of separable kernels. Separating $n \times n$ kernel into $1 \times n$ and $n \times 1$ kernel further reduced computation.

Tip: *Data required to train model increases exponentially with an increase in the number of training parameters. How can we mitigate this requirement to certain extent? The technique to create more data from existing training data is called data augmentation. Algorithms used for data augmentation depends on the task we are solving. For example, for image data that is used to identify animals like cat/dog, we can use affine transformation like rotation, scaling, shear, and translation discussed earlier.*

VGG

VGG models are proposed by **Visual Geometry Group (VGG)** from University of Oxford. VGGNet [6] model proposed in 2014 secured the first and the second places in the localization and classification tracks, respectively, in ILSVRC 2014. Main contribution is to increase depth using an architecture with very small (3x3) convolution filters. Two 3x3 convolution layers (without spatial pooling in between) has an effective receptive field of 5x5, and three such layers have a 7x7 effective receptive field. Using only 3x3 convolution layers make it uniform architecture.

Note: *What have we gained by using a stack of three 3x3 conv. layers instead of a single 7x7 layer?*

First, we incorporate three non-linear rectification layers instead of one, which makes the decision function more discriminative. Second, we decrease the number of parameters. This approach can also be seen as imposing a regularization on the 7×7 conv. filters, forcing them to have a decomposition through the 3×3 filters (with non-linearity injected in between).

TensorFlow provides popular models (along with trained weights on ImageNet dataset) for importing in our code. *Code 10.10* depicts the steps to import VGG16 model from TensorFlow library and print its architecture.

Number ‘16’ in VGG16 represents the number of trainable layers. VGG16 model is one of the ways to express VGGNet as mentioned in [6]:

```
1. import tensorflow as tf
2. # Instantiating built in vgg16 model
3. vgg16 = tf.keras.applications.vgg16.VGG16(
4.     include_top=True, weights=None,
5.     classes=1000, classifier_activation='softmax'
6. )
7. print(vgg16.summary())
```

Code 9.10: TensorFlow keras code to import vgg16 model

ResNet

ResNet [7] from Microsoft team won ILSVRC 2015 classification, detection, localization tasks. Model introduced residual learning framework to ease the training of networks that are deeper. With the proposed framework, they were able to go much deeper than the earlier CNN networks. It reformulated the layers as learning residual functions with reference to the layer inputs instead of learning unreferenced functions. It provided comprehensive empirical evidence, showing that these residual networks are easier to optimize and can gain accuracy from considerably increased depth. On the ImageNet dataset, it evaluates residual nets with a depth of up to 152 layers, 8 times deeper than VGG nets but still having lower complexity. Refer to the following figure (source[7]):

Figure 9.31: Residual Learning Framework’s building block

Consider $H(x)$ as an underlying mapping to be fit by a few stacked layers (not necessarily the entire net), with x denoting the inputs to the first of these layers. If one hypothesizes that multiple non-linear layers can asymptotically approximate complicated functions, then it is equivalent to hypothesize that they can asymptotically approximate the residual functions, that is, $H(x) - x$ (assuming that the input and output are of the same dimensions). So, rather than expecting stacked layers to approximate $H(x)$, it explicitly let these layers approximate a residual function $F(x) := H(x) - x$. The original function thus becomes $F(x) + x$, as shown in [Figure 9.31](#). Although both forms should be able to asymptotically approximate

the desired functions (as hypothesized), the ease of learning might be different.

Xception

Xception model [8] was proposed by François Chollet, creator of Keras deep learning library, in 2016. Proposed architecture is entirely based on depthwise separable convolution layers. It is based on the following hypothesis: mapping of cross-channels correlations and spatial correlations in the feature maps of convolutional neural networks can be entirely decoupled. As this hypothesis is a stronger version of the hypothesis underlying the inception architecture, this architecture is named Xception, which stands for “*Extreme Inception*”.

The typical inception module first looks at cross-channel correlations via a set of 1x1 convolutions, mapping the input data into two or four separate spaces that are smaller than the original input space, and then maps all correlations in these smaller 3D spaces, via regular 3x3 or 5x5 convolutions. This is illustrated in [Figure 9.32 \(a\)](#):

Figure 9.32: (a) Canonical Inception V3 module (b) “extreme” version of our Inception module, with one spatial convolution per output channel of the 1x1 convolution (Source: [8])

An “extreme” version of an Inception module, based on this stronger hypothesis, would first use a 1x1 convolution to map cross-channel correlations, and it would then separately map the spatial correlations of every output channel. This is shown in [Figure 9.32\(b\)](#).

Xception model significantly outperformed Inception V3 on a larger image classification dataset. Xception architecture has the same number of parameters as Inception V3; the performance gains are not due to increased capacity but to a more efficient use of model parameters. Depth separable convolutions proposed through this model are used in other popular models like *MobileNets*[11].

Application of CNN models

CNN models have been used in a variety of tasks whose input is image or video. We will discuss some of the tasks applicable on images like

classification, object detection, and segmentation.

Image classification

Image classification is a task that attempts to comprehend an entire image. The goal is to classify the image by assigning it to a specific label. Typically, image classification refers to images in which only one object appears and is analyzed.

The first CNN-based application used for hand-written digit classification is LeNet[3]. CNN models discussed previously, like VGG, ResNet, Inception models, can be applied directly to image classification tasks. Apart from these, popular models are ShuffleNet[26], NASNet[27], and SqueezeNet[28]. These models are modifications to those mentioned earlier, and they are being used in almost all domains that deal with images like medical, sports, and medicines. For example, CNN models are applied for identifying lung infection [12] and breast cancer [13].

Object detection

The difference between **object detection** algorithms and classification algorithms is that in detection algorithms, bounding box is drawn around the object of interest to locate it within the image. It might not necessarily be just one bounding box in an object detection case; there could be many bounding boxes representing different objects of interest within the image, and it is not known how many beforehand.

A naïve approach to solve this problem would be to take different regions of interest from the image and use a CNN model to classify the presence of the object within that region. The problem with this approach is that the objects of interest might have different spatial locations within the image and different aspect ratios. Hence, one would have to select a huge number of regions, and this could computationally blow up. Researchers have proposed techniques to mitigate this cost. We will discuss some of these popular models. Sample output of object detection algorithms is shown in [Figure 9.33:](#)

Figure 9.33: Object Detection – drawing bounding boxes around all items identified in an image
(source: tensorflow.org)

R-CNN – Regions with CNN features

R-CNN [16] proposal combines two key insights: one can apply high-capacity CNN models to bottom-up region proposals to localize and segment objects and when labeled training data is scarce, supervised pre-training for an auxiliary task, followed by domain-specific fine-tuning, yields a significant performance boost.

R-CNN accepts an image as input, extracts around 2000 bottom-up region proposals, computes the features for each proposal using a CNN model, and then classifies each region using class-specific linear **Support Vector Machines (SVMs)**. Improvements over this approach are proposed as Fast R-CNN [17], Faster R-CNN[18], Mask R-CNN[19].

YOLO – You Only Look Once

Prior work on object detection repurposes classifiers to perform detection. Instead, YOLO [15] (2016) performed object detection as a regression problem to spatially separated bounding boxes and associated class probabilities. A single neural network predicts bounding boxes and class probabilities directly from full images in one evaluation. Since the whole detection pipeline is a single network, it can be optimized end-to-end directly on detection performance. Several improvement versions of this approach are proposed.

Image segmentation

Image segmentation is the process of assigning a unique label from a set of finite labels to every pixel in an image such that pixels with the same label share certain characteristics (ex: color, intensity, texture). There are two broad categories of segmentation: semantic and instance.

Semantic segmentation associates every pixel of an image with a class label (ex: person, flower, car, foreground, background), where multiple objects of the same class are treated as a single entity. **Instance segmentation** too associates every pixel of an image with a class label but

treats multiple objects of the same class as distinct individual instances. Output of segmentation algorithm is shown in [*Figure 9.34*](#):

Figure 9.34: (a) Semantic segmentation (b) Instance segmentation

Few popular models for semantic segmentation task are FCN [20], U-Net [21], and SegFast[22]. Popular models for instance segmentation task are Mast R-CNN [19] and YOLACT [23].

[U-Net](#)

U-Net [21] is primarily used for semantic segmentation. This network has inspired several networks used in various other image processing tasks, like biomedical image segmentation, dense volumetric segmentation, and image-to-image translation. This model consists of a contracting/encoding path (left side) and an expansive/decoding path (right side), as shown in [*Figure 9.35*](#). The contracting path follows the typical architecture of a convolutional network. It consists of the repeated application of two 3×3 convolutions (unpadded convolutions), each followed by a ReLU activation function and a 2×2 max pooling operation with stride 2 for downsampling. At each downsampling step, the number of feature channels is doubled. Every step in the expansive path consists of an upsampling of the feature map, followed by a 2×2 convolution that halves the number of feature channels, a concatenation with the correspondingly cropped feature map from the contracting path (dotted box in figure) and two 3×3 convolutions, each followed by a ReLU. The cropping (dotted box in figure) is necessary due to the loss of border pixels in every convolution. At the final layer, a 1×1 convolution is used to map each 64-component feature vector to the desired number of classes. Refer to the following figure (source[21]):

Figure 9.35: U-Net architecture, blue box represents multi-channel feature map, number of channels denoted on top of box and spatial dimensions on left, white box represents copied feature maps

Energy function is computed by a pixel-wise soft-max over the final feature map combined with the cross-entropy loss function. The soft-max is defined as follows:

Where \hat{y} is approximated maximum-function, y_k denotes the activation in feature channel k at the pixel position with $\hat{y}_k = 1$ for the k that has the maximum activation and 0 for all other k . Cross-entropy then penalizes at each position the deviation of using the following:

Where y is the true label of each pixel and w is a weight map that is introduced to give some pixels more importance in the training.

Summary

In this chapter, we discussed about algorithms that applies filter using convolution to process the image and various categories of filters along with examples. Next, we discussed about architectures that used both DNN and convolution operation, called CNN which helped us to learn filters automatically based on the given task. We discussed about development of CNN and few models in detail along with few applications. In next [chapter 10 Sequence Learning Models](#), we will learn from the sequence data.

Further reading

The current trend of research in image processing has eventually become investigation and experimentation of various CNN architectures. The models mentioned in this chapter are insufficient to understand the landscape of CNN models but would provide the platform to understand other models. To understand the landscape of CNN models, you can go through survey papers like [24] and [25]. Just to give a glimpse, popular CNN models are plotted in paper [25], along with learnable parameters and operations required for one inference, as shown in [Figure 9.36](#):

Figure 9.36: Ball chart reporting Top-1 accuracy vs computational complexity, size of ball represents trainable parameters of the model (Source: [25])

Points to remember

- Dimensions of images is large, due to which tasks like classification cannot be applied directly. You need to extract features from the

images before performing the task.

- Initially, features were extracted from the images with the use of filters. These filters were manually developed based on the required task. This task of identifying the filters for the required tasks is very tough.
- CNN models learnt the filters based on the given task. These models have been successful on image tasks.

References

1. “*Digital Image Processing*” 4th edition by *Rafael C Gonzalez, Richard E Woods*.
2. “*Deep Learning*” by *Ian Goodfellow, Yoshua Bengio, Aaron Courville*
3. Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, “*Gradient-based learning applied to document recognition*,” in Proceedings of the IEEE, Nov. 1998. Popularly called *LeNet*.
4. Krizhevsky, Alex and Sutskever, Ilya and Hinton, Geoffrey E, “*ImageNet Classification with Deep Convolutional Neural Networks*” in Proceedings of the NIPS 2012, vol. 25. Popularly called *AlexNet*.
5. C. Szegedy et al., “*Going deeper with convolutions*,” IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015. Popularly known as *Inception V1* or *GoogLeNet*.
6. S. Liu and W. Deng, “*Very deep convolutional neural network based image classification using small training sample size*,” 3rd IAPR Asian Conference on Pattern Recognition (ACPR), 2015. Popularly known as *VGGNet*.
7. He, K., Zhang, X., Ren, S. and Sun, J., 2016. *Deep residual learning for image recognition*. In Proceedings of the IEEE conference on computer vision and pattern recognition. Popularly known as *ResNet*.
8. Chollet, F., 2017. *Xception: Deep learning with depthwise separable convolutions*. In Proceedings of the IEEE conference on computer vision and pattern recognition. Popularly known as *Xception* model.
9. The Hebbian Learning Rule specifies how much the weight of the connection between two units should be increased or decreased in proportion to the product of their activation. The rule builds on

Hebb's 1949 learning rule, which states that the connections between two neurons might be strengthened if the neurons fire simultaneously.

10. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J. and Wojna, Z., 2016. *Rethinking the inception architecture for computer vision*. In Proceedings of the IEEE conference on computer vision and pattern recognition. Popularly known as *Inception V3*.
11. Andrew, G. and Menglong, Z., 2017. *Efficient convolutional neural networks for mobile vision applications*. Popularly known as *MobileNets*.
12. Q. Li, W. Cai, X. Wang, Y. Zhou, D. D. Feng and M. Chen, “*Medical image classification with convolutional neural network*,” 2014 13th International Conference on Control Automation Robotics & Vision (ICARCV), 2014.
13. Jiang, Y., Chen, L., Zhang, H. and Xiao, X., 2019. *Breast cancer histopathological image classification using convolutional neural networks with small SE-ResNet module*. *PloS one*, 14(3), p.e0214587.
14. Iandola, F., Moskewicz, M., Karayev, S., Girshick, R., Darrell, T. and Keutzer, K., 2014. *Densenet: Implementing efficient convnet descriptor pyramids*. Popularly known as *DenseNet*.
15. Redmon, J., Divvala, S., Girshick, R. and Farhadi, A., 2016. *You only look once: Unified, real-time object detection*. In Proceedings of the IEEE conference on computer vision and pattern recognition. Popularly known as *YOLO*.
16. Girshick, R., Donahue, J., Darrell, T. and Malik, J., 2014. *Rich feature hierarchies for accurate object detection and semantic segmentation*. In Proceedings of the IEEE conference on computer vision and pattern recognition.
17. Girshick, R., 2015. *Fast r-cnn*. In Proceedings of the IEEE international conference on computer vision.
18. Ren, S., He, K., Girshick, R. and Sun, J., 2015. *Faster r-cnn: Towards real-time object detection with region proposal networks*. Advances in neural information processing systems.
19. He, K., Gkioxari, G., Dollár, P. and Girshick, R., 2017. *Mask r-cnn*. In Proceedings of the IEEE international conference on computer vision.

20. Long, J., Shelhamer, E. and Darrell, T., 2015. *Fully convolutional networks for semantic segmentation*. In Proceedings of the IEEE conference on computer vision and pattern recognition. Popularly known as *FCN*.
21. Ronneberger, O., Fischer, P. and Brox, T., 2015, October. *U-net: Convolutional networks for biomedical image segmentation*. In International Conference on Medical image computing and computer-assisted intervention. Springer, Cham. Popularly known as *U-Net*.
22. Pal, A., Jaiswal, S., Ghosh, S., Das, N. and Nasipuri, M., 2018, December. *Segfast: A faster squeezenet based semantic image segmentation technique using depth-wise separable convolutions*. In Proceedings of the 11th Indian Conference on Computer Vision, Graphics and Image Processing (pp. 1-7). Popularly known as *SegFast*.
23. Bolya, D., Zhou, C., Xiao, F. and Lee, Y.J., 2019. *Yolact: Real-time instance segmentation*. In Proceedings of the IEEE/CVF International Conference on Computer Vision (pp. 9157-9166). Popularly known as *YOLOACT*.
24. Li, Z., Liu, F., Yang, W., Peng, S. and Zhou, J., 2021. *A survey of convolutional neural networks: analysis, applications, and prospects*. IEEE Transactions on Neural Networks and Learning Systems.
25. Bianco, S., Cadene, R., Celona, L. and Napoletano, P., 2018. *Benchmark analysis of representative deep neural network architectures*. IEEE Access.
26. Zhang, X., Zhou, X., Lin, M. and Sun, J., 2018. *Shufflenet: An extremely efficient convolutional neural network for mobile devices*. In Proceedings of the IEEE conference on computer vision and pattern recognition. Popularly known as *Shufflenet*.
27. Qin, X. and Wang, Z., 2019. *Nasnet: A neuron attention stage-by-stage net for single image deraining*. arXiv preprint arXiv:1912.03151. Popularly known as *NASNet*.
28. Iandola, F.N., Han, S., Moskewicz, M.W., Ashraf, K., Dally, W.J. and Keutzer, K., 2016. *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5 MB model size*. arXiv preprint arXiv:1602.07360. Popularly known as *SqueezeNet*.

CHAPTER 10

Sequence Learning Models

Many unstructured data types, like video, audio, and text, are sequential in nature. Video is a sequence of image frames, audio is a sequence of audio frames, and text is a sequence of words. Sequential data can also be structured in some scenarios. For example, time series: stock prices, sensor data, and weather monitoring data. These data types have time as one of the data dimensions. In sequential data, the elements of the sequence cannot be assumed to be **independent and identically distributed (i.i.d.)**. For example, a video frame image at time t is very similar to the frame at time $t+1$. Given a word in a sentence, there are limited possibilities for the next word for making the sentence meaningful. In all the models we discussed so far in this book, the i.i.d assumption over the data points was always a necessary condition to train the models. Hence, modelling sequences requires some specialised models, which can respect this dependency between successive data points in the sequence.

There are three major types of sequential modelling problems that we will discuss in this chapter:

1. Forecasting the future value of a sequence. For example, predicting stock price tomorrow.
2. Classifying an entire sequence, like genome read categorization
3. Sequence to sequence models like translating text in one language to other, speech to text, or text to speech. We will start with some classical probabilistic sequence models and then introduce **Recurrent Neural Networks (RNN)** and its variations.

Structure

In this chapter, we will cover the following topics:

- Time series models

- Probabilistic sequence models: Markov chains, HMMs
- Recurrent neural networks
- LSTM, GRU, Bi-directional RNN
- LSTM with attention
- Sequence to Sequence Models: Encoder-Decoder Architecture
- Self-attention and transformer architecture
- Applications

Objectives

This chapter gives a detailed overview of various types of sequence models and dives deep into some of the deep learning-based state-of-the-art sequence models. These deep learning models are employed for solving various AI problems and are already being used by many applications that we encounter every day, for example, speech recognition, handwriting recognition, language translation. After going through this chapter, you will be able to formulate business problems to a sequence modelling problem and employ a suitable class of sequence models to solve it.

Time series models

A time series is sequence of data points ordered in time. Here, time is the independent variable. A time series is called *stationary* if its statistical properties do not change over time, that is, it has constant mean and variance. Stationarity is one of the primary assumptions most time series models have. However, time series models encountered in practice are generally non-stationary. They are made stationary using some transformations, and then the stationary series is modeled.

A *non-stationary time series* may have trend, seasonality, and cyclical component. These are called *deterministic components*. If we can remove these components from the time series, the residual is often observed to be a stationary *stochastic component*.

Decomposition of time series

Following are the components of a time series:

- **Trend:** A long-term increase or decrease in value that might not be linear
- **Seasonal component:** Exists when a series exhibits regular fluctuations based on the season (e.g., every month/quarter/year). Seasonality is always of a fixed and known period. The “*frequency*” of a time series is defined as the number of observations before the seasonal pattern repeats. A measure observed every minute might have an hourly seasonality (frequency = 60), a daily seasonality (frequency = $24 \times 60 = 1440$), or a weekly seasonality (frequency = $24 \times 60 \times 7$).
- **Cyclical component:** Exists when data exhibit rises and falls that are not of fixed period. The average length of cycles is longer than the length of a seasonal pattern. In practice, the trend component is assumed to also include the cyclical component. Sometimes, the trend and cyclical components together are called trend-cycle.
- **Irregular component:** This is a stationary process; it's the residual time series after the removal of trend-cycle and seasonal components, corresponding to the high frequency fluctuations of the series.

Figure 10.1 shows an example time series: Airline Passengers dataset : ‘AirPassengers.csv’ [<https://www.kaggle.com/rakannimer/air-passengers>], which contains the total number (in thousands) of monthly airline passengers over a period.

Note: Cyclic behavior must not be confused with seasonal behavior. They are quite different. If the fluctuations are not of a fixed frequency, then they are cyclic; if the frequency is constant and associated with some temporal aspect, then the pattern is seasonal.

Refer to the following figure:

Figure 10.1: Airline passengers' data

We can see a linearly increasing trend and some seasonal spikes in the passengers.

Mathematically, we can represent a time series as follows:

where $\text{is a trend-cycle component, } \text{is a deterministic seasonal component, and } \text{is the irregular component. The functional form of } f \text{ can be either additive or multiplicative, that is, } \text{or } . \text{ However, taking logarithm of both sides, we get } . \text{ So, multiplicative relationship can be fit by fitting additive relationship to the logarithms of the data and then moving back to the original series by exponentiating. } \text{Figure 10.2} \text{ shows additive decomposition of a time series. We have used Python library } \texttt{statsmodel's seasonal_decompose} \text{ function to generate this plot, as shown in the following code:}$

```

1. import statsmodels.api as sm
2. import pandas as pd
3. df = pd.read_csv('AirPassengers.csv', header=0)
4. df = df.reset_index(drop=True).set_index('Month')
5. res = sm.tsa.seasonal_decompose(df, freq=12)
6. resplot = res.plot()

```

Code 10.1:

This data has monthly passenger counts, and we observe that the seasonal pattern is yearly. Hence, the frequency of this time series is 12 (line 5 in the previous code snippet).

Estimating trend: The trend of a time series can be estimated using *moving average*. The average over a specific time window computed at each point t is called the moving average. This window can be two sided or one sided. For example, considering a two-sided window of size 3, we have the moving average time series:

Taking one sided window of size 3 (taking two past time stamp values), we have:

This smoothens the time series. The window size is a parameter to the moving average time series:

- If there is no seasonal component, the trend of the time series can be estimated by simply taking any odd number moving average.

- If there is seasonal component, then the length of the moving average must be equal to the seasonal frequency.

Estimating seasonality: The seasonal factor can be extracted by subtracting the trend denoted by \hat{y}_t from the time series y_t . If the frequency of the time series is d , we must have $\sum_{t=1}^d \hat{y}_t = 0$. Seasonal factors should cancel out when added over one entire period, that is, we must have $\sum_{t=1}^d s_t = 0$. If this does not happen, then there is a way to correct it by adjusting each seasonal factor as follows:

Refer to the following figure:

Figure 10.2: Decomposition of time series of airline passenger data

Differencing

Considering the difference in consecutive values, we can create a difference time series. The differenced series is the *change* between successive observations in original series and can be written as Δy_t . The differenced series will have one value less than the original series. For example, daily stock prices timeseries is non-stationary, but the daily changes can be stationary. Thus, differencing is another way to make a non-stationary time series stationary. Sometimes, differenced data may not appear to be stationary, and differentiating the data second time will make the series stationary: $\Delta^2 y_t$. The second difference series will have two points less than the original series. Similarly, we can compute the p^{th} order difference series.

Transformations like logarithms can help to stabilize the variance of a time series. Differencing can help stabilize the mean of a time series by removing changes in the level of a time series, and therefore, eliminating (or reducing) trend and seasonality.

Time series forecasting

Once we remove the seasonal and trend component from the time series, what remains is: ϵ_t . But we still don't have a way of forecasting the value of the time series at a future point in time. If we observe some simple

curvilinear form in the trend, we can model it with a lower-degree polynomial.

OLS model

For the time series with simple trend and no seasonality, the **Ordinary Least Squares (OLS)** or method can be used to estimate a polynomial trend and use that as an estimate. In this case, the problem reduces to a curvilinear regression with single variable ‘ t ’.

We must choose the coefficients such that the prediction error between is minimized.

The two most widely used approaches on time series forecasting are *exponential smoothing* and ARIMA models. They provide complementary approaches to the problem.

Exponential smoothing

Exponential smoothing is technique of making forecasts using weighted averages of past observations, with the weights decaying exponentially as the observations get older. Hence, it's also known as **Exponential Weighted Moving Average (EWMA)**. There are three main types of exponential smoothing algorithms: (1) simple exponential smoothing, (2) double exponential smoothing or Holt, and (3) triple exponential smoothing (Holt-Winters). In simple exponential smoothing, the smallest weights are associated with the oldest observations:

Here, α is called the smoothing parameter. This method generates reliable forecasts for a wide range of time series. Simple exponential smoothing has a “flat” forecast function, that is, all forecasts take the same value: $\hat{y}_t = \hat{y}_1$. There is a recursive formula to calculate this. The recursion starts at the first-time step as follows: $\hat{y}_1 = y_1$. Here, y_1 is the first fitted value that we must estimate:

Hence, recursively, we are computing the following; this is an efficient way to compute simple EWMA:

These forecasts will only be suitable if the time series has no trend or seasonal component.

Holt (1957) extended simple exponential smoothing to capture trend in the data, thus providing a trending forecast unlike flat forecast before. This is also known as double exponential smoothing as there are two exponential smoothing equations: one for the trend , and another for the remaining series or level series :

The initial values , are estimated by minimizing the **Sum of the Squared Errors (SSE)** for the one-step training errors, that is:

Holt and Winters (1960) extended Holt's method to capture seasonality as well. Here, we have three smoothing equations: one for the level, one for the trend, and one for the seasonal component.

Autoregressive Integrated Moving Average

Autoregressive Integrated Moving Average (ARIMA) models provide another approach to time series forecasting. While exponential smoothing models are based on a description of the trend and seasonality in the data, ARIMA models aim to describe the autocorrelations in data.

An autoregressive model of order p denoted by $AR(p)$ can be written as follows:

Moving-average model assumes that the output variable depends linearly on the current and various past values of a stochastic term, like white noise.

Rather than using past values of the forecast variable in a regression, a moving average model uses past forecast errors in a regression.

This as a **MA(q) model**, a moving average model of order q. Combining time series differencing with autoregression and a moving average model, we obtain a non-seasonal ARIMA. An ARIMA(p,q,d) model is one where dth order differencing is applied, and then the differenced series is modeled as a combination of AR(p) + MA(q) model, as follows:

Clearly, p and q are two parameters of the model. To determine appropriate p, q for the data, we sometimes use the **Autocorrelation Plot (ACF)** and the closely related **Partial Autocorrelation (PACF)** plot. You can refer to the Further Reading section for more details on this [1].

Probabilistic sequence models

Given a finite sequence , what is the probability of observing s, that is, ? In general, we can write the probability of the sequence using product rule of probability as follows:

In terms of sequence, it means that the probability of observing depends on *all the previous values* of the sequence, given by the conditional . However, in practice, we observe that dependence on all previous values of sequence is not very realistic. For example, whether it will rain today may depend on cloudy weather for the last few days. However, if it was cloudy a month ago, it cannot be a strong predictor for today's rain. Hence, we need to relax this assumption of dependence on *all* previous values. The simplest form of such relaxation is given by *first order Markov chains*, where dependence on only the most recent previous value of sequence is assumed.

Markov chain

In a first order Markov chain, to predict the next observation in a sequence, the prediction distribution will depend on immediately previous observation

only. Similarly, in a second order Markov chain, each observation is assumed to be dependent on the previous two observations.

The assumption of 1st and 2nd order can be written mathematically as follows:

In 2nd order Markov chain, for the first two elements of the sequence, the assumption of dependence on the previous two values does not hold. Hence, we write the probability of a sequence following the 2nd order Markov chain assumption as follows:

Refer to the following figure:

Figure 10.3: Markov Chains

Similarly, probability of sequence following first order Markov chain is as follows:

Now, these conditionals must be defined to complete the model. We can assume that all these conditionals share the same probability distribution. It's also known as *homogeneous* Markov chain. If the observed sequence is discrete and taking K different values, then the model is like a state space model. The conditional distribution for the first order Markov chain is given by a table called the state *transition matrix* P . The i th entry of the transition matrix denotes the probability P_{ij} . Clearly, the sum of row probabilities equals one as it indicates the probability of landing to any other state given a current state. [Figure 10.4](#) is an example of four-state transition matrix:

Figure 10.4: Example Markov chains; the numbers indicate the transition probability

The arrows in [Figure 10.4](#) indicate state transitions, and the numbers on the arrows indicate transition probability. Also, it is shown in a tabular form in the right portion of [Figure 10.4](#).

For modelling continuous variables with p^{th} order Markov chain, we can use linear-Gaussian conditional distributions, that is:

We can write it as follows:

Here, each conditional is a Gaussian distribution whose mean is linear function of its parents. This is an autoregressive or AR(p) model discussed earlier.

Markov chain models, although simple to understand, looks very abstract, and hence, are not of much use in practice. We want to build a model for sequences that is not limited by the Markov assumption to any order, but it can be specified using a limited number of free parameters. We can achieve this by introducing additional hidden (latent) variables that follow a discrete Markov chain, that is, we cannot directly observe these latent variables. However, these latent states generate data that is given by some distribution conditioned on the state. We call these **hidden Markov models**.

Hidden Markov model

Let's start with an example to understand **Hidden Markov Models (HMM)**. Given an English sentence “*Will Jane spot Mary?*”, we want to find the **Parts Of Speech (POS)** of each word in the sentence. For illustration purposes, we will assume that there are three parts of speech only: Noun (NN), Verb (VB), Modal (MD). We can represent the sentence graphically, as shown in [Figure 10.5](#):

Figure 10.5: English sentence with simple parts of speech

Here, the numbers over the arrows indicate probabilities. The probability of observing the word ‘will’ is $3/4$, given the POS tag is MD. These are called the *emission probabilities*, of the words in the sentence and are shown along

the vertical arrows. The horizontal lines are representing all the transition probabilities: the probability of observing a NN after a MD is $\frac{3}{4}$ and the probability of observing a VB after a NN is 1. We can assume that the POS sequence for any sentence is a Markov chain governed by these transition probabilities. Thus, any sentence can be assumed to be coming from a generative process. First, POS are generated from an underlying hidden Markov chain, and then, a word is generated from each POS category. An HMM follows the same graphical structure. We can pictorially represent an HMM as shown in [Figure 10.6](#). Here, the latent sequence follows a discrete Markov chain. The observed variable conditioned on the latent variables is denoted as \mathbf{x} , where ϕ is a set of parameters governing the distribution and is known as *emission probabilities*. The initial latent node is special as it does not have a parent node. Refer to the following figure:

Figure 10.6: Hidden Markov model

So, the marginal distribution represented by a vector of probabilities. We can represent the transition probabilities of the latent Markov chain by \mathbf{A} , the transition matrix. So, the entire parameter set for HMM can be represented by ϕ . The joint probability distribution is given by the following:

Now, the observed data is \mathbf{x} . So, we have to marginalize the preceding distribution if we want to write the likelihood equation such that we get an equation for the observed data:

Directly optimizing this function is intractable. We can use the **Expectation Maximization (EM)** algorithm, which was discussed in detail in [Chapter 7, Clustering](#), in the GMM section, to find an efficient framework for maximizing the likelihood function.

Baum-Welch algorithm (or forward-backward algorithm) is a dynamic programming approach and a special case of EM algorithm that is used to train HMM.

In the E-step, given the observed data and the set of parameter matrices tuned before the expected hidden states are estimated.

The M-step updates formulas to tune the parameter matrices to best fit the observed data and the expected hidden states. These two steps are then iterated over and over, until the parameters converge, or until the model has reached a certain accuracy requirement.

The latent variables in HMM can have some meaningful interpretation. So, it's often useful to find the most probable sequence of hidden states for a given observed sequence. One such example we have already discussed here is the POS tagging for a sentence. Also, in speech recognition, we can find the most probable phoneme sequence for a given series of acoustic observations. We can find the most probable hidden state sequence using *Viterbi* algorithm, a dynamic programming-based algorithm. You can refer to Further Reading [3], [4] for a detailed explanation of the Viterbi algorithm.

HMMs found applications in a wide variety of sequence modelling problems. With advancement of deep learning and its ability to train models on very large data sets, recurrent neural networks started outperforming on many of the tasks that HMMs were used for.

Recurrent neural networks

Neural network can also be used for learning sequences. In feed forward neural network, two different inputs fed at different times are assumed to be independent in the sense that first input or first input's output – both don't impact the second output. So, we cannot directly use feed forward network for modeling sequences. **Recurrent Neural Network (RNN)** models the sequence prediction problem as predicting the sequence for time step , given sequence . RNN stores past time step information in a state and then uses this along with the current time step information to predict the next time step. Also, the state is updated at every time step.

Let's understand this with a simple example. Given a sequence of numbers, we want to compute the EWMA (single) and output some function of EWMA. Let's take of EWMA as the desired output. The following code does this:

1. import pandas as pd
2. import numpy as np
- 3.
4. seq = np.random.random(10)

```

5.
print(pd.DataFrame(seq).ewm(alpha=0.1, adjust=False).mean()/2)
6.
7. #Also, we can implement it as
8. S = seq[0]
9. alpha = 0.1
10. for i in range(len(seq)):
11.     S = alpha*seq[i]+(1-alpha)*S
12.     output = S/2
13.     print(output, S)

```

Code 10.2:

Here, the variable S can be thought of as a state variable keeping the processed sequence information. On the other hand, α is a parameter of the state, which roughly determines how much previous information to store in the state. The state is initialized to be equal to the first element of the sequence. The state variable is used in every time step, along with new sequence input to compute the EWMA/2.

The structure of RNN is analogous to this example. We can replace the state variable S by a tensor h , (hidden state). h is computed as a function of the input, which is also a tensor x , and the previous hidden state h_{t-1} . The parameter α can be replaced by a weight matrix W . The predicted output by the network is a function of the state h . Here, b are the bias terms that depicts the mean hidden state vectors and the mean output vector, respectively. At $t=0$, we may initialize the hidden state by zero tensor. Formally, this can be written as follows:

Here, σ and ϕ are activation functions for hidden state and output, respectively. These are the RNN layer equations, also known as *RNN cell*. A RNN layer consists of applying the RNN cell for each element of the sequence. The cell should take input and previous state vectors and output next state and next sequence element. Unlike the feedforward neural network layer, RNN cell has a feedback loop connection, as h_{t-1} is also input to the layer that is the layer output in the previous time step. We can pictorially represent this in

two forms: one with feedback loop ([Figure 10.7](#) (left)) and the other is unrolled loop ([Figure 10.7](#) (right)):

Figure 10.7: Simple RNN architecture

Training RNN

We must define a loss function for sequences to train RNN. The loss function L of all time steps is defined based on the sum over the loss at every T time step:

. Here, \hat{y}_t is the prediction at time t , and y_t is the actual value of the sequence. E is the error function. The error function E for each time step is defined based on the type of problem we want to solve. For predicting the next element of a sequence, we can compute the MSE of predicted with actual sequence at time t .

Theoretically, RNNs should be capable of learning from very long sequences, but in practice, they are limited to looking back only a few steps. The reason for this limitation lies in chain rule-based gradient update. For updating the weights corresponding to the state tensor , we must compute the error gradient:

But \hat{y}_t depends on all the previous time step hidden states. And we can write this as follows:

is a product of Jacobians:

And hence the norm:

As the sequence length increases, if $\lambda < 1$, then the product can become very small, and we call this the vanishing gradient problem for RNN training. On

the other hand, if , these products can become very large, leading to the exploding gradient problem.

To address these issues, some modifications to the RNN architecture are made using **Long Short-Term Memory (LSTM)** and **Gated Recurrent Units (GRU)**. These are called gated cells in general.

Note: The backpropagation algorithm for training RNNs needs to unroll the input by first computing the forward step for each time step. It is called back-propagation through time (BPTT). BPTT cannot be parallelized because of this temporal dependency. So, another algorithm called teacher forcing is proposed, where the model receives ground truth output y_t as input at time $t + 1$ during training time. Rather than feeding the model's own output to itself, the target values specify what the correct output would be, and this can be easily parallelized. We can compute forward step for each time step in parallel, and hence, compute the loss in parallel.

Long Short-Term Memory (LSTM)

LSTMs consist of three or four gates, including input, output, and forget gates, which decide whether to write information to the memory or hidden state. These gates are also small neural network-based functions, formulated as follows, and are learned during the training phase. The following equations define how information flows and state updating happens in a LSTM cell:

- Forget gate:
- Input gate:
- Output gate:
- Cell input activation vector:
- Cell state output:
- Hidden state vector:

Figure 10.8 depicts a single LSTM cell taking input at time t:

Figure 10.8: LSTM Cell (source [12])

For a detailed pictorial description of LSTM architecture, you may refer to Further Reading [5].

Note: *LSTMs tend to not suffer from the vanishing gradient problem; they can have the exploding gradient problem. This can be mitigated by clipping gradient norm, that is, the gradients are rescaled when norm exceeds a threshold.*

Gated Recurrent Unit (GRU)

The GRU unit combines the forget and input gates we had in LSTM into a single “update gate.” It also merges the cell state and hidden state. GRU is a simpler model with only two gates. GRU can be trained much faster. [Figure 10.9](#) shows the architecture and the cell update equations:

Figure 10.9: GRU Cell (source [5])

Stacked LSTM/RNN

As we have seen in other deep neural network models, adding more layers helps in learning complex feature representation of data; similarly, addition of LSTM layers adds levels of abstraction of input observations over time. This was first introduced for developing speech recognition model [6]. We will see in the following sections that this architecture is used in several sequence modelling tasks. Refer to the following figure:

Figure 10.10: Stacked

The sequence models discussed so far are suitable for predicting the next element of the sequence. Now, let’s look at the other two types of sequence modelling problem: sequence classification and predicting another sequence or sequence generation.

Generative models for sequence

While inferencing from RNN, given a starting state and input, a trained RNN can predict the next element of the sequence. This can be iteratively

used to generate very long sequence. Suppose we train a LSTM on a large collection of text documents where each document is fed to the LSTM as a sequence of characters. Then, given a seed state, we can generate new text that is representative of the text from the original corpus. The problem of developing a model to generalize the structure of a collection of text documents is called *language modelling*, which we will discuss in the next chapter.

A generative LSTM is not really any new architecture; it is more of a change in perspective about how the model is used and interpretation of what the model has learned from the data. Here, we will briefly discuss one generative model for handwriting generation. We suggest that you go through the paper “*Generating Sequences with Recurrent Neural Networks*” [13] for more on this, like handwriting generation and text generation. We will discuss handwriting generation in greater detail here. This model is trained on IAMOnline DB [14], which has the pen stroke handwriting data collected from various authors who were asked to write on a smart board with a stylus. The pen stroke coordinates were captured in an XML format, as shown in [Figure 10.11](#). Here, a ‘stroke’ node in XML represents a continuous curve drawn without lifting the pen. The x, y pen coordinates of each stroke data is sampled at the shown timestamp. The ground truth text is attached as a label to the pen stroke coordinates data. Refer to the following figure:

Figure 10.11: IAMOn data sample

Handwriting generation

Handwriting is considered as a sequence of coordinates: , where the first two dimensions are the coordinates of pen stroke and the third is Boolean indicating whether the pen is up or down. So, in the preceding xml, whenever a new pen stroke starts, we must take , otherwise . Sometimes, we must lift the pen from the paper to complete certain strokes, like the dot in small letter ‘i’ or the horizontal line in small letter ‘t’. Stacked three layers of LSTM is used to model the first difference series of coordinates. Also, skip connections are included for faster training and to avoid the vanishing gradient problem.

The difference series is assumed to follow Bivariate Gaussian Mixture Model (GMM) distribution. LSTM is used to predict the parameters of this GMM distribution at each time step, that is, given a coordinate , the model must be able to predict the parameters of GMM distribution from which the next coordinate comes. Also, the pen touch probability can be modelled using a Bernoulli distribution. Therefore:

Bivariate GMM distribution parameter can be represented as set of M mixture weights , M location (mean) parameters , and M covariance matrix parameter of the Gaussians :

That is, is a $6M$ dimensional vector of parameters.

Mixture Density Network

Mixture Density Networks (MDN) are a class neural network of models obtained by combining a conventional neural network with a mixture density model. The neural network outputs are used to parameterize a mixture distribution. If RNN is used as the neural network, then the output distribution is conditioned not only on the current input but also on the history of the previous input sequence. [Figure 10.12](#) shows the MDN layer architecture for a bivariate Gaussian with three mixture components. There are $6 \times 3 = 18$ nodes in the MDN layer. Here, mixture weight output nodes are normalized with a *softmax* function to ensure that they form a valid categorical distribution. The other nodes for various parameter components are passed through suitable functions to keep their values within meaningful range. The variance should always be positive, and hence, can be passed through *exponential* activation function. The covariance can be positive or negative, but we don't want it to take very large values, and hence, use *tanh* activation to limit its value in the range [-1, 1]. Refer to the following figure:

Figure 10.12: MDN Layer output

For handwriting generation using MDN, we can replace the base neural network by stacked LSTM. Let's denote the stacked LSTM outputs at time t by \hat{z} . We want to model \hat{z} . [Figure 10.12](#) gives the high-level architecture of the network. The final layer of the network is MDN, which will always output the parameters of a GMM. The coordinates of the handwriting can be obtained by sampling from the learned distribution, that is, learned GMM. For sampling, the first mixture component must be chosen based on the parameters \hat{z} . Then, we can sample the coordinates \hat{x} from the chosen bivariate mixture. Here, \hat{x} and \hat{y} are not the actual coordinates but the first order difference of the coordinates. The actual coordinates can be obtained by computing the cumulative sum of this sampled difference sequence.

To train the network, we must maximize the log likelihood of the model. We can write the log likelihood as follows:

Here, \hat{z} is the output from the MDN layer. For bivariate normal distribution, we can write the previous expression as follows:

We must minimize negative log likelihood over the entire sequence to train this neural network. We have the following:

Following is an implementation of this loss function. Here, epsilon is a small constant to prevent underflow in log function:

```

1. epsilon = 1e-8
2. def mdn_loss(real_coords, e, pi, mu1, mu2, std1, std2,
rho):
3.     xs, ys, es = tf.unstack(real_coords, axis=2)
4.     mrho = 1 - tf.square(rho)
5.     xms = (tf.expand_dims(xs, axis=2) - mu1) / std1
6.     yms = (tf.expand_dims(ys, axis=2) - mu2) / std2
7.
8.     #Calculate probability for each element of the
sequence

```

```

9.           z = tf.square(xms) + tf.square(yms) - 2. * rho *
xms * yms
10.          n = 1. / (2. * 3.14 * std1 * std2 *
tf.sqrt(mrho))
11.          n = n * tf.exp(-z / (2. * mrho))
12.          mixture_probability = tf.reduce_sum(pi * n,
axis=2)
13.
14.          #binomial finish(sequence end) probability
15.          e = tf.squeeze(e, axis = 2)
16.          ep = es * e + (1. - es) * (1. - e)
17.
18.          #-log likelihoods for sequence (sum over sequence)
19.          sequence_loss = tf.reduce_mean(
20.              -tf.math.log(mixture_probability +
epsilon) \
21.                  - tf.math.log(ep + epsilon), axis=1)
22.          loss = tf.reduce_mean(sequence_loss)
23.          return loss

```

Code 10.3:

Note: While training RNNs for generative modeling, the output and input of RNN are from the same distribution, that is, the RNN must output the next element of the input sequence. Then, the next element of sequence can be predicted by feeding the predicted output as input. However, the training becomes very slow if we do this. Hence, a different strategy called teacher forcing is used while training. This uses ground truth as input, instead of the model's predicted output from a prior time step as an input.

For generating handwriting from this model, we can start with the coordinate (0, 0, 1) as initial input to the RNN and generate the first parameter set from the GMM layer. We can sample the coordinates by first choosing one of the mixture components and then generating a sample from the corresponding bivariate Gaussian, as shown in the following code from these parameters:

```
1. def sample(e, mu1, mu2, std1, std2, rho):
```

```

2. cov = np.array([[std1 * std1, std1 * std2 * rho],
3.                 [std1 * std2 * rho, std2 * std2]]))
4. mean = np.array([mu1, mu2])
5.
6. x, y = np.random.multivariate_normal(mean, cov)
7. end = np.random.binomial(1, e)
8. return np.array([x, y, end])

```

Code 10.4:

The sampled coordinates are passed as input to the network along with the previous hidden states, and then the next time step coordinates are generated. This is repeated for several time steps. [Figure 10.13](#) shows a sample of the generated handwriting after training the model with the preceding loss function and sampling:

Figure 10.13: Sample generated handwriting

Sequence classification

Sequence classification is a predictive modeling problem, where given a sequence of observations, the task is to predict a category for the sequence. Sequence classification has a wide range of applications, such as genomic analysis, information retrieval, health informatics, finance, anomaly detection, gesture recognition, and motion recognition.

A sequence may carry a class label. For example, a time series of ECG data may come from a healthy or ill person. A DNA sequence may belong to a gene coding area or a non-coding area. A piece of text can be considered a sequence of words. A movie review is a sequence of words, and it can be positive or negative. These are examples of binary sequence classification problems.

A typical architecture for sequence classification is an RNN to encode the variable length sequence input into one fixed size vector. This acts as the feature vector of the sequence, which is passed through a series of fully connected layer, followed by a softmax layer for classification. [Figure 10.14](#) depicts this:

Figure 10.14: Typical sequence classification architecture

RNNs process the sequence in forward direction only. For some sequences, like text, it may be necessary to see words coming next to the current word to get more context. In speech recognition also, classification for phonemes can be better if we look at the future phonemes and not only the phonemes already uttered. This is in line with human speech understanding. Many times, we understand a word uttered by a speaker only after listening to the next word or may be after completion of the sentence. BRNNs, discussed in the next section, can process sequences from both directions to extract better sequence features.

Bi-directional RNN

Bi-directional RNNs (BRNN) process the sequence in both directions. Typically, two separate RNNs are used: one for forward direction and one for reverse direction. This results in a hidden state from each RNN, which are usually concatenated to form a single hidden state. The forward and backward RNNs don't interact; they can be trained in the same way as the standard RNNs. This is shown in [*Figure 10.15*](#):

Figure 10.15: Bi-directional RNN

Multiple layers of BRNNs can be stacked for more complex sequence classification tasks. BRNNs are used in many other sequence modeling tasks, like speech recognition, handwriting recognition, and sequence anomaly detection.

Sequence to Sequence

Sequence-to-Sequence modelling (Seq2Seq) is about training models to convert sequences from one domain to sequences in another domain. Following are a few examples of Seq2seq modelling:

- Handwriting recognition
- Language Translation: An English sentence to French
- Speech Recognition: Audio to text transcript
- Video captioning

- Question-answering: chat bots
- Text to speech
- Text to handwriting ink coordinates

In all these examples, input sequences and output sequences have different lengths. Although RNN models are powerful sequence learning models, the input is *unsegmented* in many practical problems like handwriting (cursive) recognition, speech recognition, and gesture recognition. For example, in cursive handwriting recognition, we have the **Sayre's paradox**: A word written in cursive cannot be recognized without being segmented and cannot be segmented without being recognized. This means most OCR systems based on character segmentation are not usable directly for cursive handwriting. [Figure 10.16](#) explains unsegmented input for handwriting recognition:

Figure 10.16: Alignment issues in sequence to sequence

So, the main problem is the *alignment of input to output*, that is, which part of handwritten image corresponds to a character or which segment of the speech signal corresponds to a character. Had there been a way to segment the input, a simple RNN-based architecture would have sufficed to build the models, but that is not the case here. The first breakthrough in this was done by Alex Graves in 2006 [7], where this problem was solved by introducing Connectionist Temporal Classification (CTC), which is an *alignment free* technique. Much later, in 2015, a novel neural network architecture was introduced by Google [8] to address this challenge.

Connectionist Temporal Classification

The idea of **Connectionist Temporal Classification (CTC)** is eradicating the need for explicit alignment, that is, to redefine the problem as an alignment-free problem with *input and output sequence having the same length*. The input is first split into equal-sized segments such that we get a sequence input from a raw audio/image. For each segment, we must have a target label. For example, consider the handwriting example shown in [Figure 10.17](#):

Figure 10.17: Alignment of handwritten image to characters

For each segment of the image, a character label must be assigned. Here, we see that the character ‘c’ is two segments long, ‘a’ is two segments long, and ‘t’ is one segment long. Suppose we had this labeling available as training data; we could easily train a RNN to model this. Removing the repeated characters from the prediction would give the final output, but what if we had a word with repeated characters, like ‘hello’. Suppose the alignment after image segmentation is [h, h, e, l, l, l, o]. Then, collapsing the repeats will produce “helo” instead of “hello”. Let’s introduce a new character called the blank token ϵ . ϵ token doesn’t correspond to any character in output and is simply removed from the output. Consider a possible alignment using ϵ , as follows: [h, h, e, ϵ , ϵ , l, l, l, ϵ , l, l, ϵ , o]. From this alignment, if we first remove repeats and then ϵ , we will get the word ‘hello’, as shown in [Figure 10.18](#). CTC can help us achieve this without creating segment-wise labeling or annotation for training a model. Refer to the following figure:

Figure 10.18: Alignment of handwritten image to characters using CTC type encoding

The alignment we saw earlier is not unique. Based on different handwriting inputs, there may be different possible alignments. The word hello can be derived from any of the following [hh ϵ ll ϵ l ϵ o, hee ϵ lll ϵ llo, hhhe ϵ ll ϵ loo, ...]. If we have a predefined length of encoding sequence T , then there are N^T possible sequences, where N denotes the number of alphabets. Hence, any encoded character output y of length T is a sample from a distribution over the set of all sequences .

With this view of the target sequence, we can use simple RNN/LSTM with softmax output layer (having N number of output nodes) to model the output distribution. Given an input handwriting or audio segmented into T parts, this model will output a probability distribution over N characters at each time step t . This is called CTC network, as shown in [Figure 10.19](#). Refer to the following figure:

Figure 10.19: CTC Network

Now, how do we train such a model? We don't have a target alignment defined so that we can evaluate cross entropy loss at each of T time steps. The target label is still the given text corresponding to a handwriting or audio.

Training CTC network: Maximum likelihood

We can choose the subset of all encoded sequences of length T , which represent the target sequence, out of all possible sequences. For example, let the target sequence be 'hello'. If we take , we have . We must maximize the probability of observing , given input sequence x , that is, :

The goal is to maximize the log probabilities of all the correct classifications in the training set, that is, minimizing the following objective function where Tr denotes the training set.

The probabilities are given by the softmax activation. For any , we can compute as the product of T probabilities given by their respective softmax activations. To compute these probabilities efficiently, we can represent it in a tabular form with $T=8$ columns, and the number of rows is the length of target sequence, that is, 'hello'. To allow blanks in the output, we modify the output sequence by interleaving empty tokens between the characters "hello" and making it "Eh Ee El El Eo". Any alignment is a path from a node in t_1 to a node in t_8 . A valid path must satisfy the following:

- We are allowed to move right or down.
- Paths can include any number of \in tokens but must also include all characters of the target string "hello".
- At least one \in must be included between the reprinted characters "ll" in any valid path.

One valid path corresponding to the alignment 'hel' is shown in [Figure 10.20](#):

Figure 10.20: The path for 'hel-lo--'

The probability of this path is as follows:

Here, denotes the probability (given by softmax activation) at the time step t for the character c. Other valid paths are “hel”, “”, and so on. Many of these paths have portions in common; for example, “hel” and “” have “hel” in common, that is, they have a common subproblem to solve. Hence, these probability computations can be further optimized by **Dynamic Programming (DP)**.

DP formulation for CTC loss

Given a labeling l , we define as forward variable:

- $\alpha_t(l)$ = total probability of observing labeling l that is:

$$\alpha_t(l) = \text{total probability of observing first } s \text{ symbols of } l \text{ till time } t.$$
- $\beta_t(l)$ = total probability of observing labeling l that ends at time t
- $\Gamma_t(l)$ = all paths corresponding to l that go through symbol $l[s]$ at time t

To allow for blanks in the output paths, we modified label sequence l with blanks added to the beginning and the end and inserted between every pair of labels. Let’s call this l' . The length of l' is . We can write the probability of l as the sum of the total probabilities of l' with and without the final blank at time T :

We will compute the forward and backward variables by CTC Forward-Backward Algorithm: the algorithm to solve the previous DP. We are not providing the details of the solving DP. You may refer to the paper [7] to know more. Most DL frameworks have APIs available to compute CTC loss. For TensorFlow, it’s “`tf.nn.ctc_loss`”. The gradient of this loss can be computed directly from the forward and backward variables and is also discussed in the paper.

Inferencing from CTC network

Inferencing from CTC network can be done in two different ways: *greedy search* and *beam search*. Given an input sequence of length T , the CTC

network will output a sequence of probability distributions of length T . The greedy decoding chooses the output token that has the maximum probability in each time step.

Instead of greedily choosing the most probable next step, as the output sequence is being decoded out of the probabilities, the beam search expands all possible next steps and keeps the k most likely and controls the number of beams or parallel searches through the sequence of probabilities. [Figure 10.21](#) shows an example beam search with beam width and for an output alphabet {a, b, }. Refer to the following figure:

Figure 10.21: Beam search diagrams [source: <https://github.com/distillpub/post--ctc/issues/4>]

Limitations of CTC are as follows:

- CTC assumes that the model outputs for a given frame are independent of the previous frames. The CTC layer is not recurrent.
- Output sequences cannot be longer than the input sequence – the output can be longer than the input for tasks like text-to-speech, or text to handwriting.

Next, we will discuss another architecture for sequence-to-sequence modelling that mitigates these limitations.

Encoder-Decoder architecture

A simple encoder decoder architecture is depicted in [Figure 10.22](#). This architecture supports different input sequence lengths for the source and target. The encoder is a recurrent neural network like LSTM or stacked LSTM, which encoded the source sequence into a fixed size vector. This is called the encoder *context vector*. This is the final hidden state of the encoder. The context vector acts as an initial hidden state of the decoder. The context vector has the responsibility of encoding all the information in each source sequence to one single vector. This is challenging for longer sequences. To solve this, a new technique called *attention mechanism* is introduced. BRNNs are also used in the encoder architecture instead of simple LSTMs. Refer to the following figure:

Figure 10.22: Encoder-Decoder architecture

Attention mechanism

Attention allows the model to focus on the relevant parts of the input sequence as required, by accessing *all the past hidden states of the encoder, instead of just the last one*.

Here, a context vector is derived, which captures relevant source-side information. Attention mechanism-based models have two broad categories: *global* [[Figure 10.23](#)] and *local* attention. This means whether the “*attention*” is placed on all source positions or on only a few source positions, that is, how the context vector is derived.

Global attention: The context vector c is the weighted sum of hidden states of the input sequence, weighted by alignment scores (a scalar):

Here, attention alignment scores are computed using one-hidden layer feed-forward network with a softmax activation, as follows:

is the decoder hidden state at time t , and is the source sequence hidden state at time step s in the source sequence.

The current target hidden state is compared to all source states using score function to derive . The *score* function outputs a scalar value. For example, the score can be a dot product of the vectors . Refer to the following figure:

Figure 10.23: Attention (source <https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html>)

Different attention mechanisms compute this score in different ways:

- **Dot product:** Multiply the hidden states of the encoder by the hidden state of the decoder at time t :
- **General:** Very similar to the dot product, but a learnable weight matrix is included to project the encoder hidden states and then

perform the dot:

- **Concat:** The decoder hidden state and encoder hidden states are concatenated before being passed through a linear layer with a tanh activation function and, finally, being multiplied by a weight matrix :

where are trainable parameters.

- **Bahdanau attention:** This is similar to the concat given earlier, except that they have used the previous hidden state of the decoder , and instead of concatenating the hidden states, they used separate weight matrices :

The weight matrices and the vector are learned during training.

- **Location based:** The scores are computed from solely the target hidden states, as follows:

Local attention: A single aligned position for the current target is predicted first. The source hidden states from a window of source time steps is used to compute a context vector . Local attention is also known as *window-based attention*. Here, D is empirically selected. Now, can be chosen in the following ways:

- **Monotonic alignment:** ; here, it assumes that the source and target sequence are monotonically aligned.
- **Predictive alignment:** , S being the length of source sequence. Because of the sigmoid function, .

We can now modify the alignment scores to give more weightage to positions near the source alignment position . We can do this using a Gaussian centered at , that is:

Here, is a fixed variance that can be empirically chosen.

Key-value-query formulation of attention

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values. In other words, the output is the context-vector. We can view the constituents of attention layer as follows:

- The ‘query’ is the last decoder hidden state till time step t.
- The ‘values’ are the encoder outputs, that is, all the hidden states of the encoder. The ‘query’ attends to all the ‘values’.
- We may represent the values by some ‘key’ and compare the query with the keys instead of directly comparing it with the values.

This is analogous to document search or retrieval. For searching a YouTube video, we pass a text query. This query text is compared to various video descriptions (keys), and the relevant videos (values) are retrieved. For the types of attention discussed so far, we have value=key. With this formulation, all the previous score functions have only two arguments, i.e., query and value:

For example, we can implement Bahdanau attention as follows:

```
1. class BahdanauAttention(tf.keras.layers.Layer):  
2.     def __init__(self, units):  
3.         super(BahdanauAttention, self).__init__()  
4.         self.W1 = tf.keras.layers.Dense(units)  
5.         self.W2 = tf.keras.layers.Dense(units)  
6.         self.va = tf.keras.layers.Dense(1)  
7.  
8.     def call(self, query, values):  
9.         #query shape: [batch_size, hidden_size],  
10.        #values shape: [batch_size, inp_seq_len,  
hidden_size]  
11.        query = tf.expand_dims(query, 1)  
12.        scores = self.va(tf.nn.tanh(self.W1(query) +  
self.W2(values)))  
13.        attention_weights = tf.nn.softmax(scores, axis=1)
```

```

14.         context_vector = attention_weights * values
15.         context_vector = tf.reduce_sum(context_vector,
axis=1)
16.         return attention_weights, context_vector

```

Code 10.5:

Now, let's see how we can use this to implement a language translator model that can translate a sentence in French to English.

Scaled dot-product attention: Using the key/value/query formulation, another attention scoring function can be defined, which is very similar to dot-product attention discussed earlier; it is called scaled dot product attention. Here, α is the scaling factor, and it is equal to the dimension of the query, key vectors:

The attention is defined as: $\alpha = \frac{1}{\sqrt{d_k}} \text{softmax}(\text{query}^T \text{key}) \text{value}$. Here the values v need not be the same as keys.

Language translation model

Given a sequence of text in a source language, say French, the task is to convert it into a target language, like English. The natural ambiguity and flexibility of human language causes the lack of a single best translation of the source text to another language. This makes the problem of language translation challenging. Classical models of language translation were either rule-based or statistical. The rule-based models were developed by linguists, and the collection of such rules was massive. Statistical approach is data driven, where a collection of pairs of source and target language were used to convert phrases or sub-sentences from source to target language. It's known as **Statistical Machine Translation (SMT)**. Neural network models when employed for this task outperformed both the previous approaches.

We can download French-English dataset, which consists of bilingual sentence pairs, from <http://www.manythings.org/anki/>.

Text can be viewed as a sequence of words. Suppose there are N words in the vocabulary of any language; we can represent them by numbers 1 to N . Thus, the source sentence is a sequence of integer indices. We must convert

these integers to one hot encoded vector. Generally, N will be very large, so converting it to one hot encoding will result in a high-dimensional and sparse vectors. To avoid this, words are converted to dense vector representation using a lookup matrix that has N dense vectors of dimension . This lookup matrix is called *embedding matrix*, and is called the *embedding dimension*. We will discuss embedding matrix in greater detail in the next chapter on NLP.

Following is the code for the encoder. This model takes a sequence of integers as input and uses the embedding layer in line 8 to convert the sequence of integers to sequence of dense vectors, which are processed by the RNN (or GRU). Refer to the following code:

```
1. class Encoder(tf.keras.Model):
2.     def __init__(self, vocab_size,
3.                  embedding_dim,
4.                  enc_units, batch_sz):
5.         super(Encoder, self).__init__()
6.         self.batch_sz = batch_sz
7.         self.enc_units = enc_units
8.         self.embedding =
tf.keras.layers.Embedding(vocab_size,
9.                           embedding_dim)
10.        self.gru = tf.keras.layers.GRU(self.enc_units,
11.                                       return_sequences=True,
12.                                       return_state=True,
13.
recurrent_activation='sigmoid')
14.        def call(self, x, hidden):
15.            x = self.embedding(x)
16.            output, state = self.gru(x, initial_state =
hidden)
17.            return output, state
18.        def initial_hidden_state(self):
19.            #Generating encoder initial states as all zeros
20.            return tf.zeros((self.batch_sz, self.enc_units))
```

Code 10.6:

Following is the decoder model that uses Bahdanau attention to look at all the source sequence hidden states with the current decoder hidden state as the query [*line 21* in the following code snippet]. The context vector found by attention is concatenated with the embedded target language token at *i*th step and passed to the decoder as the input.

```
1. #Decoder with attention
2. class Decoder(tf.keras.Model):
3.     def __init__(self, vocab_size,
4.                  embedding_dim,
5.                  dec_units, batch_sz):
6.         super(Decoder, self).__init__()
7.         self.batch_sz = batch_sz
8.         self.dec_units = dec_units
9.         self.embedding =
10.            tf.keras.layers.Embedding(vocab_size,
11.                                      embedding_dim)
12.            self.gru = tf.keras.layers.GRU(self.dec_units,
13.                                         return_sequences=True,
14.                                         return_state=True,
15.                                         recurrent_activation='sigmoid')
16.            self.fc = tf.keras.layers.Dense(vocab_size)
17.            self.attention = BahdanauAttention(dec_units)
18.            def call(self, x, hidden, enc_output):
19.                # enc_output (batch_size, max_length, hidden_size)
20.                # hidden (batch_size, hidden size)
21.                attention_weights, context_vector =
22.                  self.attention(hidden,
23.                                 enc_output)
24.                  x = self.embedding(x)
25.                  #Concatenating previous output with contx_vec
26.                  x = tf.concat([tf.expand_dims(context_vector, 1),
x], axis=-1)
27.                  output, state = self.gru(x)
```

```

27.         output = tf.reshape(output, (-1, output.shape[2]))
28.         x = self.fc(output)
29.         return x, state, attention_weights
30.
31.     def initialize_hidden_state(self):
32.         return tf.zeros(self.batch_sz, self.dec_units)

```

Code 10.7:

The output of the decoder is passed through a dense layer [*line 28* in the preceding code snippet] to predict the next decoded token. We must use a softmax layer to predict the probability of the next token. Here, we output the logits that can be converted to probabilities in the loss function, as discussed in the following code:

```

1. def loss_function(self, real, pred):
2.     loss_ =
tf.nn.sparse_softmax_cross_entropy_with_logits(
3.             labels=real, logits=pred)
4.     return tf.reduce_mean(loss_)

```

Code 10.8:

Now, let's discuss the training step that uses teacher forcing. We have used two special tokens “**<start>**” and “**<end>**” to indicate the beginning and end of a sentence in source or target language, respectively. For example, the sentence “*Comment ça va?*” in French can be converted to the sequence [**<start>**, ‘comment’, ‘ça,’va’, ‘?’, **<end>**]. We have used ‘**tf.keras.preprocessing.text.Tokenizer**’ to convert the sentences into sequences, as shown in the code snippet 10.9. We will use teacher forcing to train the network. So, the input to the decoder will be the real input from the target language, not the input predicted by model in the previous step. This is shown in *line 36*; in code 10.9. we have a loop to iterate through the entire decoder sequence. We have made sure that the length of each sequence in a batch is the same by padding the sequences with zeros.

```

1. class NMTModel:
2.     def __init__(self,
3.                  vocab_size_in,
4.                  embedding_dim_in,
5.                  vocab_size_out,

```

```
6.          embedding_dim_out,
7.          enc_units,
8.          dec_units,
9.          batch_size,
10.         inp_lang_tokenizer,
11.         targ_lang_tokenizer
12.      ) :
13.      super(NMTModel, self).__init__()
14.      self.inp_lang_tokenizer = inp_lang_tokenizer
15.      self.targ_lang_tokenizer = targ_lang_tokenizer
16.      self.batch_size = batch_size
17.      self.encoder = Encoder(vocab_size_in + 1,
18.                             embedding_dim_in, enc_units, batch_size
19. )
20.      self.decoder = Decoder(vocab_size_out + 1,
21.                             embedding_dim_out, dec_units,
batch_size)
22.      self.optimizer = tf.keras.optimizers.Adam()
23.
24. @tf.function
25. def train_step(self, inp, targ):
26.     loss = 0
27.     hidden = self.encoder.initial_hidden_state()
28.     with tf.GradientTape() as tape:
29.         enc_output, enc_hidden = self.encoder(inp,
hidden)
30.         #final encoder states are taken as initial
decoder states
31.         dec_hidden = enc_hidden
32.         #Passing '<start>' token as first token to
decoder
33.         dec_input = tf.expand_dims(
34.             [self.targ_lang_tokenizer. \
35.
word_index['<start>']] * self.batch_size, 1)
36.         for t in range(1, targ.shape[1]) :
```

```

37.          # passing enc_output to the decoder
38.          predictions, dec_hidden, _ = self.decoder(
39.              dec_input, dec_hidden, enc_output)
40.          loss += self.loss_function(targ[:, t],
41.              predictions)
42.          # using teacher forcing
43.          dec_input = tf.expand_dims(targ[:, t], 1)
44.          batch_loss = (loss / int(targ.shape[1]))
45.          variables = self.encoder.variables +
46.              self.decoder.variables
47.          gradients = tape.gradient(loss, variables)
48.          self.optimizer.apply_gradients(zip(gradients,
variables))
49.      return batch_loss

```

Code 10.9:

The training time of encoder-decoder model is very high because of the loop in the decoder step. This cannot be parallelized because of the dependency on the previous time steps. Attention-based speech recognition models like LAS, discussed as follows, were trained for about a month using GPUs to get the desired results.

Speech recognition model

Automatic Speech Recognition (ASR) or converting speech to text was traditionally approached by breaking the problem into multiple stages and solving using multiple independent models for each stage, as depicted in the [Figure 10.24](#):

Figure 10.24: Components speech recognition model

HMM-based acoustic model was used to create phonemes from audio signal features. The phone sequence was converted to word sequence or text by another decoder model.

ASR can be viewed as a sequence-to-sequence modeling problem. The input speech is a sequence of audio frames, and output is a sequence of characters depicting the transcription corresponding to the speech signal.

The speech signals can be hundreds to thousands of frames long. In the paper *Listen, Attend and Spell (LAS)* [9], pyramid structure stacked BRNN is used as the encoder architecture called *listener*, as shown in [Figure 10.25](#). This helps to reduce the length of the input sequence considerably. Each layer reduces the sequence length by a factor of 2; hence, with 3 layers, we have a factor of 8 reduction in the sequence length. Refer to the following figure:

Figure 10.25: Pyramidal recurrent encoder

Next, they have used a decoder with attention over the encoded audio. This is another RNN-based decoder that outputs a character distribution in every time step. The distribution for y_t is a function of the decoder state s_t and context vector c_t . The decoder state s_t is a function of the previous state s_{t-1} , the previously emitted character y_{t-1} , and context c_t . The context vector c_t is produced by an attention mechanism. The attention mechanism is content based: the contents of the decoder state (*query*) are matched to the contents of encoder hidden states using dot product type attention:

Where ϕ and ψ are neural networks learned during training.

Self-attention and transformers

Attending to elements of the same sequence is called **self-attention** or **intra-attention**. While modelling the input sequence with encoder, we assumed that every element of the sequence may have dependence on all previous elements. This need not be true all the time. So, while encoding the sequence also, we can use attention mechanism over the same sequence. For example, consider a sequence of words in the sentence. We will take two similar sentences:

“*The animal didn’t cross the street because it was too tired*”.

“*The animal didn’t cross the street because it was too wide*”.

In the first sentence, “it” refers to the animal and in the second, it refers to the street. Can we capture such relations and inter-dependencies within the same sequence? This is what self-attention or intra attention is all about.

[Figure 10.26](#) shows the same. Sequence to sequence tasks like machine translation have improved using self-attention.

Figure 10.26: Visualizing self-attention

Self-attention is an attention mechanism relating different positions of a single sequence to compute a representation of the sequence.

Computing self-attention

Self-attention is represented as a vector. For each input vector in an input sequence (like embedding vector in case of text sequence), the self-attention vector is computed. First, each input vector is projected to 3 separate vectors by a linear transformation, that is, multiplying by a matrix. For query and key vectors, we use matrices of dimension , where is the input dimension or embedding dimension. is the projection dimension. For value, we can use the projection matrix of dimension . We may choose for simplicity, as shown in [Figure 10.27](#), where and . The sequence length is 5. Using these projected vectors, we can compute scaled dot product attention taking each input as query and all other input's keys and values. Then, we can add all the softmax weighted value vectors for each query and get the corresponding attention vector for each element of the sequence. Refer to the following figure:

Figure 10.27: Pyramidal encoder

So, applying self-attention on a sequence, we get a sequence of the same length as the input. This sequence has the attention features encoded in its representation. This was first introduced in the paper “Attention is All you Need” (Vaswani, et al., 2017). In this paper[11], a novel architecture called **transformer** is introduced, which is a recurrence free architecture for modelling sequences. Refer to the following figure:

Figure 10.28: Multi headed attention

There, they have used multiple self-attention layers in parallel to encode different types of features in sequence. This is like the multiple filters we

find in CNN architectures. They called this the *multi-headed attention*. This computes multiple attention vectors, one for each head, and these are concatenated to form the final representation, as depicted in [Figure 10.28](#).

Transformer architecture

The transformer architecture also has an encoder and a decoder for modelling sequence to sequence tasks. Here, the encoder is a stack of transformer layers that creates a representation of the input sequence, just like all the hidden states generated by RNN-based encoder. However, the transformer is not recurrent. The decoder shares the same structure as the encoder, but it inserts another additional sub-layer, which performs multi-head attention over the output of the encoder stack. This is called encoder-decoder attention. The “Encoder-Decoder attention” layer works just like multiheaded self-attention, except that it creates its queries matrix from the layer below it and takes the keys and values matrix from the output of the encoder stack. This is shown in [Figure 10.29](#). While training, the decoder self-attention layer must attend to earlier positions in the output sequence only. This is essential as the entire output sequence will not be available while inferencing from this network; only previously predicted elements will be available. This restriction is done by masking future positions. You can refer to <https://www.tensorflow.org/text/tutorials/transformer> for more details on implementation.

Transformers are applied to various sequence modelling tasks, like speech recognition, handwriting recognition, and sequence classification. Using transformers, many natural language processing tasks have improved drastically. We will be discussing these models in the next chapter. Refer to the following figure:

Figure 10.29: Transformer Encoder-Decoder architecture [source: Further Reading [10]]

Conclusion

In this chapter, we covered various sequence models, like time series models for structured data, probabilistic models like Markov chains, HMMs, neural network models for unstructured data like RNN and LSTM, and transformer models. We also covered the applications of these models

for solving various AI problems like speech recognition, handwriting recognition, language translation, sequence generation, and sequence classification.

In the next chapter, we will see more applications of sequence models for modelling text and solving other natural language processing problems.

Points to remember

- Sequence modelling tasks are of three major types: (1) forecasting the future value of a sequence, (2) classifying an entire sequence, and (3) Sequence to sequence models.
- If our data set is small, we must restrict ourselves to simpler models, like time series models, or hidden Markov models.
- Neural models require large data sets for training. However, even with small, labeled training data set, we can pretrain the network weights by defining suitable unstructured pretraining tasks.
- Pure RNNs cannot be used for learning long-range sequence patterns; instead, we use the gated variants of RNNs like LSTM and GRU.
- RNN/LSTM model training time is more because of BPTT algorithm, and this can be addressed by using transformer architecture for sequence models, which is pure feed forward architecture and can be trained with normal BP.

Further Reading

1. Otexts.com. (2016). *Forecasting: Principles and Practice*. [online] Available at: <https://otexts.com/fpp2/>
2. Buteikis, A. (n.d.). 03 Time series with trend and seasonality components. [online] Available at: http://web.vu.lt/mif/a.buteikis/wp-content/uploads/2019/02/Lecture_03.pdf
3. HMM : Viterbi algorithm -a toy example H Start. (n.d.). [online] Available at: <https://www.cis.upenn.edu/~cis262/notes/Example-Viterbi-DNA.pdf>

4. *Baum–Welch algorithm*. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Baum%E2%80%93Welch_algorithm
5. Olah, C. (2015). *Understanding LSTM Networks -- colah's blog*. [online] Github.io. Available at: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>
6. Graves, A., Mohamed, A. and Hinton, G. (2013). *Speech Recognition with Deep Recurrent Neural Networks*. [online] arXiv.org. Available at: <https://arxiv.org/abs/1303.5778>
7. Graves, A., Ch, A., Fernández, S., Gomez, F., Schmidhuber, J. and Ch, J. (n.d.). *Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks*. [online] Available at: https://www.cs.toronto.edu/~graves/icml_2006.pdf
8. (2014). *Sequence to Sequence Learning with Neural Networks*. [online] Available at: <https://arxiv.org/pdf/1409.3215.pdf>
9. Chan, W., Jaitly, N., Le, Q. and Google Brain, V. (n.d.). *Listen, Attend and Spell*. [online] Available at: <https://arxiv.org/pdf/1508.01210.pdf>
10. Alammar, J. (n.d.). *The Illustrated Transformer*. [online] Available at: <https://jalammar.github.io/illustrated-transformer/>
11. Vaswani, A., Brain, G., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A., Kaiser, L. and Polosukhin, I. (n.d.). *Attention Is All You Need*. [online] Available at: <https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fb053c1c4a845aa-Paper.pdf>
12. Jenkins, Ian & Gee, Ludvig & Knauss, Alessia & Yin, Hang & Schroeder, Jan. (2018). *Accident Scenario Generation with Recurrent Neural Networks*. 3340-3345. 10.1109/ITSC.2018.8569661.
13. Alex Graves. *Generating Sequences With Recurrent Neural Networks*, [online] arXiv.org. Available at: <https://arxiv.org/pdf/1308.0850.pdf>
14. IAMDBOn: <https://fki.tic.heia-fr.ch/databases/iam-on-line-handwriting-database>

CHAPTER 11

Natural Language Processing

About 80% of the enterprise data is unstructured data, and majority of this data is text. This text can be in any language developed and evolved by humans, that is, any *natural language*. Natural language can be in the form of speech, handwriting, printed/typed text, or digitized text. The science of making computers capable to interpret natural language in the form of digitized text is called **Natural Language Processing (NLP)**. The other forms of natural language like speech or handwriting are converted to digitized text to apply NLP. The first step in NLP is to represent the natural language in a numeric form as a vector. Once that is done, we can apply the tools from ML and DL literature to formulate and solve various business problems. NLP has a wide range of application, for example, classifying textual documents to some predefined categories – email routing and classification, finding similarity between documents – search engines, understanding human intent -chat bots, question answering system, chat bots, summarizing large text documents. NLP literature is vast and may take several chapters to discuss various concepts in detail. In this chapter, we will give a high-level overview of text processing required for text representation and discuss various models of text.

Structure

In this chapter, we will cover the following topics:

- Structure of Natural Language
- Text Preprocessing: Stemming, Lemmatization
- Bag of words model, Vector Space Model
- Probabilistic Models of Text: LSI, LDA and applications to Information Retrieval
- Dense Representation of text: Glove, Skip-Grams, CBOW
- Contextual Models for Text ELMO, BERT

Objectives

After studying this chapter, you should be able to learn the basics of NLP. This chapter will help you in understanding the state-of-the-art papers in NLP, and you should be able to implement and modify those models for solving your business problem. We will introduce the mathematical tools and concepts required for modelling text. We will cover some statistical models of text and will also cover some state-of-the-art deep learning-based NLP models for text.

Natural language

A *natural language* is one developed and evolved by humans over ages for communicating with each other. There is a hierarchical structure in natural languages. Linguistics is a branch of science for understanding these structures in languages. Spoken language can be broken down to small set of sound or acoustic patterns called *phones*. For example, the word ‘cup’ in English language can be broken into phones /k/,/uh/,/p/. This representation of atomic units of speech are called *phonemes*. In written form of natural language, the atomic units are the *characters* or letters. *Grapheme* is a letter or group of letters that represent a single phoneme. The phonemes, in turn, are the constituents of *morphemes*, that is, minimal meaningful word segments. *Words* are comprised of one or more *morphemes*. Words are combined to form a group, expressing complete thoughts, called into *phrases*, such as noun phrases, verb phrases, adjective phrases, and prepositional phrases, which are the structural components of *sentences*, expressing complete thoughts. At still higher levels, we have various types of discourse structure, like paragraphs and sections. The structure in natural language can be broadly divided into two types (1) syntactic structure and (2) semantic structure, which we will discuss in detail in the following sections. The objective of studying these linguistic structures is to apply them for feature extraction from textual data. In this chapter, we will be discussing the English language syntax and semantics, but most of these concepts can be extended to other languages as well.

Syntactic structure of language

There are certain rules that we collectively called the *grammar* of a language, which act as a guideline of how words are combined into phrases, phrases get combined into clauses, and clauses get combined into sentences.

Parts of Speech (POS)

The number of words in any language is huge. To define any grammar, we need to first categorize the words into some small set of categories and then define rules over the categories and not for individual words. We can categorize English words into nine basic types according to its syntactic function. There are called **Parts Of Speech (POS)**: noun, pronoun, adjective, determiner, verb, adverb, preposition, conjunction, and interjection.

Phrases

A phrase is one or more words that form a meaningful fragment of a sentence/clause. There are five main types of phrases: Noun Phrase (NP), Verb Phrase (VP), Adjective Phrase, Prepositional Phrase, and Adverb phrase. In each of these, a few words are added around the main word type. NP: a single noun or a group of words built around a single noun. For example: '*The President of India*' is a NP. VP: a single verb or a group of words built around a single verb. For example, *I will be going* to France next week.

Clause

A clause is a group of words with some relation between them that usually contains a subject (a noun) and a predicate (verb with an object). The subject need not always be present in a clause. There are two main types of clauses: the main clause and the subordinate clause. The main clause is also known as an independent clause because it can be a sentence by itself. The subordinate or dependent clause depends on the main clause for its meaning.

Sentence

A sentence is a grammatical unit of one or more words that expresses an independent statement, question, request, command, exclamation, and so

on., and that typically has a subject as well as a predicate. A sentence typically begins with a capital letter and ends with appropriate punctuation based on its type. Refer to the following figure:

Figure 11.1: Grammatical structure of a sentence shown as a tree. Each leaf node in the tree represents a word in the sentence with its parent representing the word type. Then, these words/word types are groups based on some rules to form noun phrase and verb phrases and so on.

Document and Text corpus

A collection of sentences in a written, printed, or digitized text file is called a text *document*. A collection of documents is called text *corpus*. It usually consists of bodies of written text, often stored in digitized form. There are many large text corpus build over time targeted for various NLP tasks.

Semantic structure of language

The study of meaning in language is called semantics. Words with similar meanings can be linked together in a database, which can be very useful in understanding the meaning of language. In any language like English, two words with same meaning can have different spellings based on the tense, number, and so on. In languages like French, different spellings are used for adjectives referring to different genders as well. The smallest meaningful and syntactically correct unit of a language is called *lemma* or *lexeme*.

The meaning of a word may change based on the position in the sentence and the context. There are words with the same spelling or pronunciation that have different meanings, called *homonyms*. For example, in the two phrases, ‘a cricket bat’ and a ‘bat which is seen at night’, the word ‘bat’ has different meanings. The relationship between words of these types is called *homonymy*. *Capitonyms* are words that have the same spelling but different meanings when capitalized. Example, the month May and the phrase ‘may be’. *Synonyms* are words that have different spellings but have the same meaning. *Polysemes* are like homonymy. A word is polysemous if it can be used to express different meanings. The difference between these two concepts is very subtle. If you hear (or read) two words that sound (or are written) the same but are not identical in meaning, we must decide if it’s really two words (homonyms), or if it is one word used in two different

ways (polysemy). Example of polysemy is “*Good man*” vs “*Good artist*”. In the first one, the word “good” describes a moral quality, and the second one is describing skill.

Wordnet

WordNet is a lexical database of semantic relations between words, as defined earlier, in more than 200 languages with the synonyms being grouped into synsets with short definitions and usage examples. Its primary use is in automatic text analysis and artificial intelligence applications.

Text preprocessing

Extracting the preceding syntactic and semantic structures computationally from text as features of text is called text pre-processing. These structures help us convert an unstructured text document into a structured vectorized form, which can be used to train a model to solve various AI problems like text classification, text summarization, question answering, and so on. Following are a few standard text preprocessing techniques applied on any textual data for feature extraction:

- **Sentence splitting:** The process of splitting a text into sentences is also known as sentence segmentation. This is mainly rule-based approach; for example, sentence must begin with capital letter and end with certain punctuation.
- **Word tokenization:** This is the process of splitting sentences into its constituent words. Here, the punctuation characters are split and separated into independent tokens. These are also rule based, and we can specify the rules to split the tokens with regular expressions, or we can use the standard grammatical rules for splitting.
- **Text cleanup:** Commonly occurring words in text like common verbs, conjunctions, for example, { a, an, and, but, how, in, is, are, on, or, the, what, will} called *stopwords* and are removed from the word token list. Lowercasing, removing special characters, blank spaces are also a common preprocessing step.
- **Lemmatization:** The word affixes are removed to get to a base form of the word. This base form must be a semantically correct word. Generally, wordnet type database is used to lemmatize a word. For

example, all of the following words {connection, connections, connective, connected, connecting} originate from the root word ‘connect’ and is the lemma of all.

- **Stemming:** Stemming is similar to lemmatization except that the output need not be a meaningful word. This is a rule-based approach. Porter stemmer is a popular rule-based stemming algorithm.
- **Sub-word tokenization:** Segmenting the word further into small chunks that need not be any meaningful word is called sub-word tokenizing. This is required to address **Out Of Vocabulary (OOV)** words. Suppose we extracted all words from a corpus, which we call the *vocabulary* of the corpus. We build text features based on these words only and a text model also takes these features as input. Now, given new text for inferencing using the trained model, if this text has a new word that is not in our vocabulary, we call it OOV word. If we have sub word tokens, then we can split the new word into sub tokens, which are part of our vocabulary. Many state-of-the-art NLP models use this tokenization technique. We discussed one sub-word tokenization in greater detail.
- **POS tagging:** The parts of speech tags can also be used as features of text.

There are multiple text preprocessing libraries in various programming languages. In Python, we have *nltk*, *genism*, *scikit learn*. Reader is suggested to refer to [1],[2] in *Further Reading* section for detailed examples. NLTK documentations [3] also provide a lot of examples to get started.

Models for text

“*All models are wrong, but some are useful!*”- this is a famous saying in statistics, generally attributed to the statistician named George Box. This holds true in NLP as well. There are various models for text, and each model is suitable for solving specific problems in text. None of these models alone can represent all aspects of structure in text. For example, if we want to find whether an email is spam or not looking at certain key words, and key phrases typically found in spam emails may be useful. So, for this problem, modelling the text as a bag-of-words suffices. We may not

need to consider the structural details in the written text. However, if we want to analyse, whether a review about a movie or a newly launched product is positive or negative, we many need to use deeper semantic and syntactic relations in the text.

Text models can be built at all the hierarchies: document level, sentence level, paragraph level, and word level. If we want to compare or classify a collection of documents, we would prefer document level text models. For building a chatbot, we need to build sentence or phrase level model that can keep track of the context of the conversation and provide desired response sentences.

Bag of Words (BoW) model

From a given text document, we apply the standard pre-processing techniques discussed earlier, and each document is represented as an unordered list of words. processed words present in the document. None of the syntactic or semantic attributes of text are captured in this representation. This is the simplest model of text. There are various approaches to convert the unordered list of words to a numeric representation discussed as follows.

Vector Space Model

Given the BoW extracted from a document, we can create a feature vector for the document, where each feature is a word, and the feature's value should reflect the presence and importance of the word in the document. The entire document is represented as a feature vector, and each feature vector corresponds to a point in a vector space. The dimension of this vector space is V , where V is the size of the vocabulary. Here, the vocabulary is built over the entire corpus.

Count based or Boolean

Given a corpus of documents, we can represent each document as a vector. Suppose there are V unique words in the entire corpus. Then, each document will be represented as a V dimensional vector. We call V the size of the *vocabulary* of the corpus. Each word in the vocabulary represents one feature in the vector. If a word w from the vocabulary is not present in a

particular document, the corresponding feature value is zero; otherwise, the feature value is the number of times the word occurs in particular document, that is, the word frequency. [Table 11.1](#) shows an example of a corpus with two documents and their corresponding word count vector. There are seven unique words in the corpus.

Following is an example:

Table 11.1: Count vector representation of documents

The entire corpus can be thus represented as a matrix where each row corresponds to a vector representation of a document. Each column corresponds to a unique word in the corpus. This is called *Document-Term matrix*. Here is a code example from Python scikit learn:

```
1. from sklearn.feature_extraction.text import CountVectorizer  
2. corpus = ["The cat sat on the mat", "The mat was red"]  
3. vectorizer = CountVectorizer()  
4. X = vectorizer.fit_transform(corpus)  
5. print(X.toarray())
```

Code 11.1:

[**Term Frequency \(TF\)-Inverted Document Frequency \(IDF\)**](#)

The count-based or Boolean models give equal importance to all the words in the corpus. There can exist a set of words that are more frequent in a corpus and occur in almost every document. These words may not provide any important feature for a document from the corpus. Rather, the rare words that occur in very few documents may give some distinguishing features. The document frequency is the number of documents where a word is seen. The **Inverse Document Frequency** denoted by **IDF** is computed by dividing the total number of documents in our corpus by the document frequency for each term and then applying logarithmic scaling on the result. We can add 1 to the document frequency for each term to prevent making $idf = 0$ for terms that occur in only one document and also for words not in corpus. This prevents divide by zero error while computing idf . The *term frequency* is the count of the term in a document and thus we weight the term frequency by idf , and we have the tf-idf for each wordp as follows:

Note: The bag-of-words models treat each word as independent, and thus, the contextual sense of words is completely lost. This problem can be partially addressed by using phrases or n-grams along with single words and computing the term document matrix over n-grams, where n is the number of consecutive words you want to consider.

Replacing `CountVectorizer` in the preceding code by `TfidfVectorizer`, we can create tf-idf term document matrix with `sklearn` Python library.

Latent Semantic Indexing (LSI) model

The vector space models are unable to cope with two classic problems in natural languages: *synonymy* and *polysemy*. In the context of information retrieval, where we have a text query that needs to be matched with a collection of documents, the query should match documents with synonyms and not with documents with polysemes. This is impossible with vector space model alone. The document-term C matrix obtained in vector space model can be modified to obtain a low-rank approximation C_k using singular value decomposition. This yields a new representation for each document in the corpus. We can cast queries into this low-rank representation as well, and thus, compute query-document similarity scores in this low-rank representation. This process is known as **Latent Semantic Indexing (LSI)**. This low-rank approximation is distance preserving in cosine similarity sense that is, two documents which are similar in the original vector space should be also similar or close by in the lower dimensional subspace defined by LSI. Let's take a small corpus of documents, as shown in [Figure 11.2](#), to understand this better:

Figure 11.2: Term-document matrix of sample corpus

We discussed in detail in [Chapter 3, Vector Calculus](#), how to compute SVD. Using the same method, we can decompose the preceding term document matrix, as shown in [Figure 11.3](#):

Figure 11.3: Singular value decomposition

Here, we have chosen the two largest singular values only; thus, the dimension of our low-rank space is 2. The preceding matrix gives us the document representation in two-dimensional space, that is, each of the 5 documents in the corpus are represented as a two-dimensional column vector. This is known as *document embedding*. Similarly, the left matrix U represents each word as a dense 2-dimensional vector. This is called *word embedding*. Now, for a new document not in the corpus, we must have a similar representation to compare it with any of the embedded documents. The new document is mapped into its representation in the LSI space by the following transformation:

This enables us to do semantic querying on the corpus without even supplying any synonym list to the query pre-processing using. The LSI representation has leaned the similarity of words from the data give a large enough corpus. For implementation of LSI refer to Gensim API reference for LSI [4] in the *Further Reading* section.

Probabilistic models of text

A sentence S is a sequence of words . A probabilistic language model estimate: . Probability of an upcoming word can be estimated as Thus, applying the chain rule of conditional probability, the probability of observing a sentence is as follows:

In the preceding factorization, we can restrict the conditionals by the assumption that the nth word depends on only the last d words at most and not all the words. The most simplistic assumption is that the current word is conditionally independent of all other words, which reduces this model to a bag-of-words model and is called unigram model.

- **Unigram model:** . We can estimate as count of the word in the corpus / total number of words.
- **Bigram model:** Approximating factors to condition on only the previous word.

Estimating bigram probability:

These counts can be obtained by counting pairwise occurrences of the words in the entire corpus for every pair of words in the vocabulary. Bigram estimates of sentence probability for an example sentence can be calculated as follows:

$$P(\langle s \rangle \text{ I love English food } \langle e \rangle) = P(I|s) \times P(\text{love}|I) \times P(\text{English}|\text{love}) \\ \times P(\text{food}|\text{English}) \times P(\langle e \rangle|\text{food})$$

Here are special tokens indicating the start and end of sentence.

- **N-gram Model:** We can extend bigram to trigrams, 4-grams and 5-grams. Google has published their n-gram model by processing 1,024,908,267,229 words of running text and computing counts for all 1,176,470,663 five-word sequences that appear at least 40 times. For any given corpus, we can create our own n-gram model by counting occurrences of n-word phrases.

Topic models

Any text document constitutes of information about a topic like science, technology, entertainment, health and well-being, sports, business, politics and so on. Document may have more than one topic but is very unlikely to have all possible topics.

Given a corpus of documents, say news articles, we can always assign one or more topics to it. Now, these topics can be expressed as a bag of words. For example, [Figure 11.4](#) shows an example document discussing four topics: arts, budget, children, and education. Each of these topics can be associated with some words from the document. The same word may be present in two different topics. The word ‘state’ can come in two topics: state and education. Assignment of words to a topic is done best by looking at multiple documents and analysing the word frequencies across documents in various topics. This method of representing a text document as a mixture of topics is called *topic model*. These models represent the topic as a latent variable denoted by z . There are two types of topic models that we will discuss in the next two sections. Refer to the following figure:

Figure 11.4: Topics extracted from text (Source:
<https://www.jmlr.org/papers/volume3/blei03a/blei03a.pdf>)

Probabilistic generative models: Latent Dirichlet allocation

Now, with this view of text documents as a mixture of words from a few topics, we can enhance the naive bag of words model to a probabilistic generative model called **Latent Dirichlet Allocation (LDA)**. This is based on Dirichlet Distribution, hence the name LDA.

The basic idea of topic model is that documents are represented as mixtures of topics(latent), where each topic is characterized by a distribution over words. So, we can assume that *each* word in a document is generated by a two-stage sampling process:

1. Choose a topic from topic distribution.
2. Choose words from the topic based on the distribution of words describing the topic.

Now, the topic distribution must be learned from the corpus. Assuming a finite number of topics M, we can model the topic distribution follows as Multinomial Distribution (discussed in [Chapter 4, Probability Theory](#), under the *Multivariate distributions* section). Multinomial distribution for M categories will have M parameters , where . How do we find these ? We don't have topics assigned to each document such that we can just do a frequency count of topics and assign probability to each topic as a ratio of per topic count to total topics count. So, we take the Bayesian approach. We will assume a prior distribution for the parameters . This must be a continuous distribution as is a continuous variable and also, we have to restrict the samples of the distribution with the constraint . The *Dirichlet distribution* denoted by has this property, where is a M dimensional vector. This is depicted in [Figure 11.5](#) for various values of parameter .

Note: Dirichlet distribution p.d.f is given by:

Here, represents the gamma function, which is a generalization of factorial function for any positive real number. Note the similarity of the

functional form of the Dirichlet distribution with multinomial distribution. The factorials are replaced by functions.

For $M=2$ this Dirichlet distribution can be geometrically represented as a line segment from 0 to 1. The samples are , where . For $M=3$ this distribution can be represented by a triangular region in the first quadrant of the coordinate axes. As shown in [Figure 11.5](#), line AB represents . For any point inside the triangle . So, if we take then we meet the required condition . For $M=4$, we can choose points from a tetrahedron whose base is the triangle for $M=3$. For $M>4$, we have a n-dimensional simplex as the region.

Refer to the following figure:

Figure 11.5: Dirichlet distribution (source: <https://www.cs.cmu.edu/~epxing/Class/10701-08s/recitation/dirichlet.pdf>)

Now, assuming that the parameters of the required multinomial distribution follow the prior Dirichlet distribution, we can formally write the probability of observing latent topic (z) as follows:

A word from a document can be generated from the multinomial word distribution conditioned on the topic with probability:

Here, is a matrix, k being the number of topics and V being the number of words in the vocabulary where .

The basic assumption in LDA is that a document is a bag of words, and hence, words are assumed to be independent of each other. So, the joint probability distribution of the words and topics can be written as follows:

We can define the joint distribution of the topic mixture , a set of N topics z (one for each word), and a set of N words w is given by the following:

This factorization is possible because of the conditional independence of θ on w . Hence, we have this:

This generative probabilistic model can be pictorially represented as in [Figure 11.6](#):

Figure 11.6: Graphical model representation of LDA: The rectangular plates represent repetition. The outer plate represents repetition over M documents and inner plate repeats over N words in each document. Here, N is taken to be fixed but can be taken as a Poisson distributed variable as document lengths need not be fixed.

Now we must learn the two parameters of the model and from the text corpus, that is, what setting of these parameters will generate the text corpus documents with very high probability. The log likelihood of the corpus of M documents is as follows:

Here, θ must be obtained by marginalizing the joint distribution over the hidden variables and that is intractable. Hence, we have to look for alternative ways of estimation, like EM algorithm or Bayesian parameter estimation. We have discussed one Bayesian parameter estimation technique in [Chapter 5: Statistics Inference and Applications](#) called MAP where the posterior distribution of the model parameters are computed and then we try to maximize the posterior density. However, if the posterior distribution function of the parameters also takes a complicated form, optimizing those functions is hard. So, an alternative way is to find some other algorithms to explore the parameter space and find a best possible value of the parameters using the form of the posterior distribution. *Gibbs sampling* is one such algorithm for exploring the parameter space by sampling from posterior distribution. Most of the implementations of LDA use Gibbs sampling to estimate the parameters. This estimation algorithm can be run in a distributed computing framework like *Hadoop Spark* [5]. *Mallet* is a java-based topic modelling tool and *Gensim* python library has a wrapper for this.

Neural language models

A neural network language model is a language model based on neural networks. These models have been successful in creating dense vector representations of words that can be used for writing vector equations, like the following:

Adding the dense vectors associated with the words *king* and *woman* while subtracting *man* is equal to the vector associated with *queen*. This describes how a gender relationship is captured in the dense word representation. One more example is: Paris – France + Poland = Warsaw. In this case, the vector difference between *Paris* and *France* depicts the concept of capital city.

Continuous Bag-of-Words (CBOW) model: The CBOW is a neural network architecture that predicts target word (the center word) based on the source context words (surrounding words). To predict the word in a context window of n words, we must maximize the conditional probability:

This network has a softmax output layer to model this probability. The high-level architecture is shown in the [Figure 11.7 \(left\)](#):

Figure 11.7: Neural language models

Skip-gram model: The skip-gram model achieves the reverse of CBOW model (11.7 (right)). It tries to predict the source context words (surrounding words) given a target word (the center word). This becomes slightly complex since we have multiple words in our context. Simplification: breaking down each (**target, context_words**) *pair* into (**target, context**) *pairs* such that each context consists of only one word.

Contextual models

The models we discussed so far have a single representation for each word. However, in many languages, including English, the meaning of a word changes based on the context of the word. For example, the word bank may have two different meanings in the following phrases: ‘*bank of a river*’,

‘Indian Bank’. However, we will have one dense or sparse representation of this word. This is not correct. Hence, we need contextual representation of words, that is, the word representation will change based on the neighbouring words. One way to capture this information is using sequence models like RNNs or bi-directional RNNs.

ELMo model

Unlike traditional word embeddings such as *word2vec* and *GLoVe*, the *ELMo* word embedding is a function of the entire sentence containing that word. Therefore, the same word can have different word vector representation under different contexts.

Embeddings from Language Model (ELMo) word vectors are computed using a two-layer bidirectional LSTM model. The input word embedding is derived from a CNN-based character embedding model. The character representation uses 16-dimensional character embeddings and 128 convolutional filters of width three characters, a ReLU activation and max pooling. Refer to the following figure:

Figure 11.8: ELMo model

In the output of the forward and reverse passes through the sentence by the bidirectional LSTM generated two output vectors per word. These are concatenated and fed to the next bidirectional LSTM layer. The final representation is the weighted sum of the raw word vectors and the two intermediate word vectors.

BERT

Bidirectional Encoder Representations from Transformers (BERT) [6] is another contextual language model that uses a stack of transformer layers. Transformer architecture was first introduced in *Language Translation* in [Chapter 11, Sequence Modelling](#). There we had a transformer encoder and decoder. The BERT model uses only the encoder part of the transformer. The following figure shows the BERT architecture with two transformer encoder layers. The number of input tokens is the same as the

number of output tokens (N). We will call these *transformed features*. Refer to the following figure:

Figure 11.9: BERT architecture with 2 transformer encoder layers

The input is a sequence of tokens embeddings of size H . The output is a sequence of vectors of size H . Length of output sequence is the same as the input sequence. These vectors are contextual embedding of the input vectors. For BERT, generally, fixed sequence lengths are used. However, inputs can be of variable length, so inputs must be padded to make them of same length.

As the transformer architecture is not recurrent and sees the whole sequence at once, we must have a way of preserving the sequential information in text. Also, these are natural segments in text, like sentences and paragraphs. Preserving this input information is important for better modelling of text. This is done by positional encoding and segment encoding.

Position encoding

As our word embeddings are vectors to encode the position of a word we cannot directly use integers. A vector representation of integers is used to encode position. We know the simplest vector representation of any integer is the binary (6-bit) representation as shown in the following table for integers 0-7. We see the least significant bit (LSB) is alternating between consecutive bits. The next bit changes in every two numbers and the next in every four numbers and so on. A continuous counter part of binary alternating signal is sinusoidal functions . Also, we can control the frequency of how they are alternating by altering the parameter . So using sinusoidal functions we can define positional encoding for a sequence is defined as follows:

The following figure shows 128-dimensional positional encoding for a sequence of length 50. We can see here the first component of the vector (the left most vertical column) alternates between high positive and negative values very similar to the LSB in binary representation and as we

move to the right, we see the frequency of change decreases. Refer to the following figure:

Figure 11.10: BERT positional encoding Source:
https://kazemnejad.com/blog/transformer_architecture_positional_encoding/

Transformer architecture is equipped with residual connections, and this allows to pass the sequence information till the top layer of the transformer. This position encoding can be used to encode both token level positions and sentence segment positions, as shown in the following figure:

Figure 11.11: BERT Positional encoding

For training this architecture and creating these embeddings, there is no need of any labelled training data. Training BERT transformer is done in an unsupervised fashion which is called *pre-training*.

Pre-training BERT

The following [Table 11.2](#) lists two pre-training tasks: (1) **Masked Language Model (MLM)** and (2) **Next Sentence Prediction (NSP)**. Labelled training data can be prepared for these two tasks easily without the need for any human annotator.

Masked LM(MLM)	Next Sentence Prediction (NSP)
<p>For each sentence 15% of the tokens are chosen at random and –</p> <ul style="list-style-type: none">• 80% of the time tokens are replaced with the token [MASK]• 10% of the time tokens are replaced with a random token• 10% of the time tokens are left unchanged <p>The BERT loss function considers only prediction of the masked values and ignores the prediction of the non-masked words.</p>	<p>Next sentence prediction task is a binary classification task in which, given a pair of sentences, it is predicted if the second sentence is the actual next sentence of the first sentence.</p>

Table 11.2: Comparison between MLM and NSP

Input representation for pre-training tasks of BERT

As BERT takes fixed sized input, each sentence is first made to be of equal length by zero padding and the following steps are used to mark separation and end of sentences.

1. The first token of every input sequence is the special classification token – **[CLS]**. This token is used in classification tasks as an aggregate of the entire sequence representation. It is ignored in non-classification tasks.
2. For single text sentence tasks, this **[CLS]** token is followed by the WordPiece tokens and the separator token – **[SEP]**.
3. For sentence pair tasks, the WordPiece tokens of the two sentences are separated by another **[SEP]** token. This input sequence also ends with the **[SEP]** token.

WordPiece tokenization

Tokenization is fundamentally the process of breaking text into tokens. **Out of vocabulary words (OOV)** or words not included in the vocabulary, are treated as “*unknown*”. A better and modern approach to address this issue is by tokenizing text into sub word units, which in most of the spits can retain linguistic meaning. So, even though a word is unknown to the model, individual sub word tokens may retain enough information for the model to infer the meaning. WordPiece is one such algorithm. Given text, WordPiece first pre-tokenizes the text into words (by splitting on punctuation and whitespaces) and then tokenizes each word into sub word units, called wordpieces. For example, let's take the sentence as shown in the following figure:

Figure 11.12: WordPiece Source (<https://ai.googleblog.com/2021/12/a-fast-wordpiece-tokenization-system.html>).

Pre-trained BERT model is generally performed on huge data sets. After pre-training BERT model can act as a feature extractor for text and can be used to solve other NLP tasks like text classification, text summarization and so on. For these tasks generally we have smaller datasets. A few tasks specific layers can be added on the top of BERT, as shown in the following

figure and trained with task specific data. This process is called *fine-tuning*. Depending on the size of the data set, we may choose to either use the BERT feature extractor as a fixed function, which is not trainable or may train all the layers.

[Figure 11.13](#) shows various fine-tuning tasks performed by incorporating BERT with one additional output layer.

- (Top Left in [Figure 11.13](#)) Sentence pair classification where given two sentences and the task is to find either the similarity score between them or classify them as paraphrases, that is, they express the same meaning using different words. Quora question pairs and **Microsoft Research Paraphrase Corpus (MRPC)** are the two popular benchmarking datasets for these tasks, respectively.
- (Top Right in [Figure 11.13](#)) The second task is sentence classification, where the computed sentence embedding [CLS] is used as features for sentence classification.
- (Bottom Left in [Figure 11.13](#)) The third task is question answering or extractive summarization, where given a question sentence and a paragraph containing the answer, the task is to pick the best sentence boundaries from the paragraph that can answer the question.
- (Bottom Right in [Figure 11.13](#)) The fourth task is sentence token labelling, like POS tagging and named entity recognition. Refer to the following figure:

Figure 11.13: BERT fine-tuning tasks

Masking used in BERT Is not adequate for Chinese language. In English, the word serves as the semantic unit and single characters do not have any meaning. The same cannot be said for characters in Chinese: certain characters do have inherent meaning fire (火, *huǒ*), water (水, *shuǐ*), or wood (木, *mù*). The character 灵 (*líng*), for example, can either mean clever (机灵, *jīlíng*) or soul (灵魂, *línghún*), depending on its match. Baidu researchers developed other pretraining tasks that are suitable for Chinese languages. They developed masking that hides strings of characters rather than single ones and named their model **Enhanced Representation through kNowledge IntEgration (ERNIE)**.

ERNIE

Enhanced Representation through kNowledge IntEgration (ERNIE) [7] model is a transformer-based model that was designed to learn language representations enhanced by knowledge masking strategies that include entity-level masking and phrase-level masking. ERNIE outperforms Google's BERT in multiple Chinese language tasks.

Following are some other pre-training tasks specific to ERNIE:

1. **Knowledge masking task:** Random phrases are masked and named entities are masked and model is trained to predict the whole masked phrases or named entities.
2. **Capitalization prediction task:** Whether the word is capitalized or not.
3. **Token-document relation prediction task:** This task predicts whether the token in a segment appears in other segments of the original document. It can enable the ability of a model to capture the key words of the document to some extent.
4. **Sentence distance task:** Enhancement of NSP. This task is modeled as a 3-class classification problem. "0" represents that the two sentences are adjacent in the same document, "1" ->two sentences are in the same document, but not adjacent, and "2" ->two sentences are from two different documents.
5. **IR relevance task:** It is a 3-class classification task that predicts the relationship between a *query and a title*. We take the query as the first sentence and the title as the second sentence. 0-> strong relevance, 1->weak relevance, 2->no relevance.

[Figure 11.14](#) shows the comparison of BERT and ERNIE pre-training strategies. Refer to the following figure:

Figure 11.14: Masking strategies of BERT and ERNIE

Generative Pre-Training by OpenAI

Generative Pre-Training (GPT) is another transformer-based model, but it uses stack of decoder blocks from the transformer only. This is unlike

BERT, which uses the encoder blocks only and is non-auto-regressive, that is, uses only the previous tokens from the sequence to predict the next one. In the standard transformer architecture, the decoder takes a word embedding concatenated with a context vector as input. In GPT-2, the context vector is zero-initialized for the first word embedding as there is no encoder block. GPT was found to be better than BERT in many natural language understanding tasks.

Conclusion

In this chapter, we covered the linguistic structures in natural language, methods of preprocessing text using these linguistic structures, and feature extraction from textual data. We also covered various models for text like vector space models, probabilistic models, and neural language models. Using these models, we demonstrated an overview of important applications of NLP, like text classification, text similarity, and application to information retrieval, summarization, question answering, and so on.

In the next chapter, we will discuss different generative modelling techniques with application to image generation and text generation.

Points to remember

- Types of language models vector space, probabilistic, neural network
- The word embeddings generated by LSI or any of the neural language models are distance preserving, that is, words with similar meaning and occurring in similar context will have embedding vectors that are close by in the embedding space.
- Contextual language models can create different embeddings for the same word based on the context of the word.
- Neural language models do not always need large volumes of data as we have models like BERT, ELMO, and GPT-3, which can be fine-tuned with smaller data sets.

Further reading

1. Dipanjan Sarkar, Text Analytics with Python A Practitioners Guide to Natural Language Processing, APRESS
2. Bird, Steven, Edward Loper and Ewan Klein (2009), Natural Language Processing with Python. O'Reilly Media Inc.
3. NLTK Documentations (<https://www.nltk.org/>)
4. GENSIM API Reference: Latent Semantic Indexing
<https://radimrehurek.com/gensim/models/lsimodel.html>
5. Qiu, Z., Wu, B., Wang, B., Shi, C., Yu, L., Fan, W., Bifet, A., Yang, Q. and Yu, P. (2014). Collapsed Gibbs Sampling for Latent Dirichlet Allocation on Spark. JMLR: Workshop and Conference Proceedings, [online] 36, pp.17–28. Available at: <http://proceedings.mlr.press/v36/qiu14.pdf>
6. Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. [online] arXiv.org. Available at: <https://arxiv.org/abs/1810.04805>
7. Peters, M., Neumann, M., Gardner, M., Clark, C., Lee, K. and Zettlemoyer, L. (2018). Deep contextualized word representations. [online] Available at: <https://arxiv.org/pdf/1802.05365.pdf.s>

CHAPTER 12

Generative Models

Generative modelling is a methodology of training machines to mimic seen data, that is, to generate new data points that resemble seen data. We have generative capabilities; once we learn about a new type of object, we can recreate similar objects either with our drawing skills, describe it in detail with words, or build a 3D model of that object. Here, at first “*representation learning*” is performed to map the data in terms of low-dimensional features. Then, those representations can be tweaked slightly to create new data points.

Generative modelling was first applied for building classification models in supervised setting where we jointly learn the probability distribution $p(x, y)$ of input x and class label y . Naive Bayes and **Gaussian Mixture Model (GMM)** are examples of such models. Later, generative models were applied in unsupervised settings as well, for example, to model unlabelled text data using topic modelling or **Latent Dirichlet Allocation (LDA)**. Text is modelled as a mixture of hidden topics, and each topic is modelled as a categorical distribution over words. These models can be collectively categorized as Bayesian Nets or probabilistic graphical models.

Restricted Boltzmann Machine (RBM) was the first neural network-based generative model. **Variational Autoencoder (VAE)** and the **Generative Adversarial Network (GAN)** are state-of-the-art neural network-based generative models with many successful real-world applications. Both these models were published long ago, but we see many modifications of these architectures and their training methodologies till date to develop newer models with application to high fidelity image, audio, video and text generation.

Structure

In this chapter, we will cover the following topics:

- A simple generative model for two-dimensional data
- Building a generative model for complicated two-dimensional data
- Representing data distributions as transformation of known simple distributions
- Variational auto encoders
- Generative adversarial networks
- Challenges of training GANs and how to overcome them
- Other GAN-based architectures: Cycle GAN, Conditional GAN
- Real-world applications of GAN
- Autoregressive generative models

Objectives

After studying this chapter, you should be able to formulate a real-world problem as a generative modelling problem and train such models for labeled/unlabeled data. This chapter will help you in understanding elegant mathematical theory behind the generative models. The loss functions used for training generative models are different from the other loss functions we have already encountered in supervised settings. We will derive these loss functions from the theory. Additionally, we will guide you to implement these network topologies and their training methodologies. Understanding the fundamental theory behind VAE and GAN will enable the reader to understand the state-of-the-art generative models and applications.

A simple generative model

The basic idea of generative model is that data follows a probability distribution and tries to approximate underlying distribution such that we can generate new data points from the same distribution by sampling from the distribution. Now, let's see what this means with the help of a simple example. Karl Pearson collected a data set consisting of the height of fathers and their adult sons in inches. He had 1078 cases. This data set is available from [kaggle.com](https://www.kaggle.com) and is named **Pearson.txt** [refer to *Further Reading 12*]. Let's first visualize a sample from this data set in the following figure:

Figure 12.1: (**Left**) Sample of 300 points from Pearson dataset with two variables x_1 =Father's height, x_2 =Adult Son's height (**Right**) Triangular marks represent generated samples from this learned distribution.

We can assume that the dataset follows bivariate Gaussian distribution and estimate the parameters of this distribution by **Maximum Likelihood Estimation (MLE)**. As discussed in the probability chapter, for a large sample, the MLE estimate for mean of Bivariate Distribution is the sample mean and the covariance matrix is the sample covariance matrix. Let X and Y be the random variables representing father's height and son's height, respectively; then, , where μ is the bivariate mean and Σ is the 2×2 covariance matrix. The correlation coefficient () between X , Y is 0.51. By definition of covariance, we have .

, these values are generated by the code below *line 5*.

Now, we can generate new samples from this distribution by random sampling from this distribution. This can be done using NumPy library's build in multivariate normal sampling function, as shown in the following code. Also, we have plotted 50 new data samples from this distribution, along with the actual data points in [Figure 12.1](#):

```
1. import pandas as pd
2. df = pd.read_csv('Pearson.txt', sep='\t')
3. df.sample(300).plot.scatter(x='Father', y='Son' )
4. """Estimate sample mean and covariance"""
5. mu = df.mean()
6. sigma = df.cov()
7. """Generate New Samples"""
8. samples = np.random.multivariate_normal(mean=mu, cov=sigma,
size=50)
9. plt.scatter(df['Father'].values, df['Son'].values)
10. plt.scatter(samples[:, 0], samples[:, 1], marker='^',
c='red')
```

Now that we have a probabilistic model for the data, we can answer a few interesting questions, as follows:

- Given the father's height is 6 feet or 72 inches. What is the probability that the son's height is over six feet? that is,

- How likely is it that the father's height is 5 feet, and his son will grow over 6 feet tall, that is,

To answer these, we can compute the conditional distribution . For bivariate normal distribution, conditional probability distribution is a univariate normal distribution with mean 70.9 and standard deviation . Hence, we can compute the required probability by area under the pdf for $X>72$, as shown in [Figure 12.2](#):

Figure 12.2: Shaded region depicts the probability (area under curve. Note that the plot on the right also has a small shaded region showing negligible probability

In the preceding example, we assumed that the data follows bivariate normal distribution, and it was a fair guess from the plot. It may not be always possible to guess the data distribution from simple visualizations. Increase in dimensionality of the data makes such assumptions impossible. Not only with high-dimensional data, even in the 2D data set shown in [Figure 12.3](#), how do we guess the underlying distribution and estimate such probable distribution by simple parameter estimation techniques? Refer to the following figure:

Figure 12.3: A synthetic two-dimensional data set

As we still can visualize the data, it seems the data follows some circular/oval distribution. Also, there is some noise in the data. There is no such known distribution function form like this whose parameters we can estimate as earlier. However, if we generate points from a known distribution and then transform the samples using a non-linear mapping, we may be able to generate samples from this distribution.

Taking sample z from standard bivariate normal distribution with mean at (0,0) and identity covariance matrix, let's transform z as follows: (as shown in [Figure 12.4](#)). Here, the first term normalizes the sample and puts them on the unit circle around the origin. Multiplying by , we can put the point at a distance from origin. Refer to the following figure:

Figure 12.4: Transforming random samples from Gaussian distribution to the circular distribution

The second term adds some noise by adding a small vector parallel to the unit vector such that the result $f(z)$ does not lie exactly on the circle circumference. Here, we choose , as the data distribution is around circle of unit radius.

The data distribution can be even more complicated, like images or text, where the number of dimensions of data increases by many folds. Can we generate an image data set following the approach described for the circular data distribution? For that, a non-linear mapping is required, which can take a known distribution and convert it to an image. Finding such mapping manually by inspection is not easy. Here, our old friend “the universal function approximators”, that is, the neural networks can help. Variational autoencoders are a class of neural network topologies that can learn such non-linear transformation from the data.

Variational Autoencoders (VAE)

Autoencoders are neural network architectures that can learn a low-dimensional representation of the input space. VAE also does the same but in a probabilistic fashion, that is, the encoder here learns a probabilistic distribution in a latent space and not a single representation of the input to a latent space. We will introduce VAE using the preceding circular distribution example.

First, let's formally rewrite the steps we followed in the previous circular distribution example:

1. We have a vector of variables z that we can easily sample according to some known probability density function $p(z)$ over some space Z .
2. A family of deterministic function , parametrized by vector .
3. f is deterministic, but z is a random variable, and hence, represents a random variable.

We have to optimize parameter such that we can sample from a simple distribution $p(z)$ and then , with high probability resembles X . Therefore, we can represent the random variable by the conditional notation $X|Z$ because first we choose Z and then X is generated based on that. We can call Z as the *latent variable* or *hidden variables*.

By law of total probability , where we can choose the output distribution to be Gaussian: , the covariance equal to the identity matrix I times some scalar . This σ can be chosen based on the factor we use in the transformation to add noise.

Now, for any complicated data distributions also, we can estimate the model parameters by maximizing $p(x)$. Instead of finding by inspection, we let it be a neural network. Maximizing $p(x)$ is equivalent to minimizing that is, MLE estimation.

Like before, we can choose $p(z) = \dots$. We can estimate $P(X)$ approximately by taking very large sample of size N from Z say $\{\cdot\}$ and then averaging the probabilities as follows:

However, for most of the samples z from this distribution, the probability of generating a data point X is very low. To understand this better, let's go back to the circular distribution example. Take a data point X from this distribution, as shown in [Figure 12.5](#). x can only be generated by z from the shaded region in the left. For all points z outside the shaded region, $p(x|z)$ is very small. This will make the estimation process intractable as the log of these very small quantities will lead to computational overflow. Refer to the following figure:

Figure 12.5: Subset of values of z that can generate a given x

Given a data point X , is there a way we can restrict the choice of samples z ? This can be done by defining the conditional probability distribution . Now, by Bayes rule, . This is computationally hard. Let's approximate by another distribution or which we can choose to be a tractable distribution. We can define the parameters of such that they are very similar to and use them to perform approximate inference of the intractable distribution.

The Kull back–Leibler divergence or KL divergence is a qualitative measure of how one probability distribution $p(x)$ is different from another probability distribution $q(x)$. Here, we can design such that $KL(p||q)$ is minimum.

By product rule of probability, we have . Substituting this in the preceding KL divergence equation, we have the following:

Using properties of log function, $\log(AB) = \log(A) + \log(B)$ and $\log(1/A) = -\log(A)$, we have the following:

Now, we can take out the terms independent of z outside the summation as a constant.

As $q(z)$ is a probability distribution over z , . Let's represent by the function . So, we can rewrite the preceding equation as follows:

Since , and KL divergence is non-negative quantity, we can write , that is, is a lower bound of .

So, maximizing means maximizing log likelihood of data: $\log(P(X))$. Now onward, we will try to simplify this lower bound function to derive our objective function.

We can write . Refer to the following figure:

Figure 12.6: is a lower bound of $p(x)$. Maximizing means maximizing log likelihood of data: $\log(p(x))$.

Therefore:

Therefore:

Pictorially, we can represent equation (II) as shown in [Figure 12.6](#).

Here, E represents expectation of over $q(z)$. This technique of approximating intractable integrals the way we did is called Variational Bayesian method in general, and this loss function being a lower bound is known as the variational lower bound or **Evidence Lower Bound (ELBO)**.

We can assume $q(z)$ to be multivariate Gaussian distribution with mean and diagonal covariance matrix , where and are n-dimensional vectors. Suppose we find a deterministic function that transforms X to the mean and variance of the tractable distribution $q(z)$. We could have directly generated z instead of mean and variance. However, as is a deterministic function, we will always get the same fixed z for a given X . Having mean and variance will enable us to take many samples from this distribution of the latent variables z . can be represented as a neural network with the parameter as shown in [Figure 12.7](#):

Figure 12.7: Using DNN to learn latent distribution $q(z|x)$

Here, $q(z|x)$ will restrict the choice of z , and hence, we can estimate our required parameter by maximizing log likelihood $\log(p(x))$. Refer to the following figure:

Figure 12.8: VAE block diagram: here, we have changed for ease in computation, which is described later.

So, now we can jointly estimate both the parameters and of the neural net topology in [Figure 12.8](#). To train this topology, our objective function is .

We saw that has two parts. The second part of is the KL divergence, which enforces the distribution $q(z)$ to take a certain form. It represents KL-divergence between two multivariate Gaussian distributions and .

In general, KL-divergence between two k -dimensional multivariate Gaussians and is given by the following:

Here,

and , and hence,

So, all terms have sum over k , and thus, the second term in simplifies to the following:

Implementing this is simple, as shown in the following code. Let z_mean and z_log_var represent the tensors and (as shown in [Figure 12.8](#)) of dimension $k=10$, initialized randomly; then:

```

1. import tensorflow as tf
2. z_mean = tf.random.uniform([1,10])
3. z_log_var = tf.random.uniform([1,10])
4. kl_loss = tf.reduce_sum(
5.     -0.5 * (z_log_var - tf.exp(z_log_var) -
tf.square(z_mean) + 1),
6.             axis=1)
7. kl_batch_loss = tf.reduce_mean(kl_loss)
```

The first part of is an expectation . This can be approximated by taking average over samples from $p(x|z)$. Assuming that $p(x|z)$ is multivariate Gaussian. , being a neural network is completely deterministic, and hence, we can write $p(x|z)$ as $p(x|z)$. $p(x|z)$, being Gaussian, will have an exponential term , and hence, $\log(p(x|z))$ will have the squared error term . So, the first part of conceptually represents the reconstruction error, as we have seen in autoencoders. We can implement it as a pixel-wise cross entropy loss as well, as we did for normal autoencoders, as shown in the following code:

```

1. '''Following is a code will run only if valid data and
reconstruction tensors are provided from an AE'''
2. '''As sum squared'''
3. reconstruction_loss = tf.reduce_mean(
4.         tf.square(tf.norm(data-reconstruction)))
5. )
6. '''As pixel wise binary cross entropy loss'''
7. reconstruction_loss = tf.reduce_mean(
8.         tf.reduce_sum(
9.             tf.keras.losses.binary_crossentropy(data,
reconstruction),
10.                axis=(1, 2))
```

```
11.         )
12.     )
```

The first part of the network for generating the latent distribution $q(z|x)$ is called the encoder, and the second part, which generated data points from samples of z , is called the decoder. We have the encoder-decoder architecture like autoencoders as both input to encoder and output from decoder is expected to be the same X , that is, .

For training, the network shown in [Figure 12.8](#) first the forward pass is computed to create the reconstructed input and then the reconstruction loss is computed. The derivative of the reconstruction loss is back propagated.

- **Forward pass:** Input X , generate , . Then, sample vector z from and pass it forward to generate .
- **Backward pass:** Compute reconstruction error + kl-loss (as shown in the following code) and back propagate error derivatives.

To implement the sampling in the forward pass, we have to use a trick called “re-parametrization”. So, we can use any standard deep learning framework to implement this. Along with a batch of data X of size n , we will sample n noise vectors $\sim N(0,1)$ at random. During forward pass, we will compute z as , where denotes element-wise multiplication. Thus, the encoder block takes batch of X and a batch of noise vectors z as input and outputs the triplet (**z_mean** , **z_log_var** , **z**), as shown in the following code. For reparameterization, we need , but we have . So, we have to transform **z_log_var** as `tf.exp (0.5 * z_log_var)`.

```
1. def vae_encoder(latent_dim):
2.     epsilon = layers.Input(shape=latent_dim)
3.     img = layers.Input(shape=(28, 28, 1)) #For MNIST
4.
5.     x = layers.Conv2D(filters=32, kernel_size=3,
6.                       strides=2, padding='same')(img)
7.     x = layers.LeakyReLU(0.2)(x)
8.     x = layers.Conv2D(filters=64, kernel_size=3
9.                       , strides=2, padding='same')(x)
10.    x = layers.LeakyReLU(0.2)(x)
11.    x = layers.Flatten()(x)
12.    x = layers.Dense(16, activation="relu")(x)
```

```

13.     z_mean = layers.Dense(latent_dim, name="z_mean")(x)
14.     z_log_var = layers.Dense(latent_dim, name="z_log_var")
(x)
15.     z = z_mean + tf.exp(0.5 * z_log_var) * epsilon
16.     return Model(inputs=[img, epsilon], outputs=
[z_mean, z_log_var, z])

```

The variational decoders can be implemented as shown in the following code using deconvolution operators. The decoder takes samples $z \sim q(z|x)$ created by encoder and reproduces X . If we are using normalized pixel input values in $[0, 1]$ for the image data, then we can apply *sigmoid* activation at the last layer of the decoder to get normalized pixel values for reconstructed image; refer to *line 12* in the following code:

```

1. def vae_decoder(latent_dim):
2.     z = layers.Input(shape=(latent_dim,))
3.     x = layers.Dense(7 * 7 * 64, activation="relu")(z)
4.     x = layers.Reshape((7, 7, 64))(x)
5.     x = layers.Conv2DTranspose(filters=64, kernel_size=3,
6.                               strides=2, padding="same")
(x)
7.     x = layers.LeakyReLU(0.2)(x)
8.     x = layers.Conv2DTranspose(filters=32, kernel_size=3,
9.                               strides=2, padding="same")
(x)
10.    x = layers.LeakyReLU(0.2)(x)
11.    decoder_outputs = layers.Conv2DTranspose(1, 3,
12.                                              activation="sigmoid", padding="same")(x)
13.    return Model(z, decoder_outputs)

```

The decoder part of the VAE can be used as a generative model. We can choose any z from the prior distribution $p(z)$ that is unit Gaussian and then decode z to generate a sample image. Note that we do not require restricting z any more as this was intended only for training the decoder. Taking latent dimension = 2 and sampling z from the square unit grid *bottom left* $[-1, -1]$, *top right* $[1, 1]$, we can visualize the generated digits in a 2D grid, as shown in [Figure 12.9](#). Position of the digit in the grid represents the position of sample z , which was used to generate the digit.

As you can see, the sharpness of the generated images by the VAE is poor. So, if we apply this technique to other complex data set with human faces, then the obtained results will not be convincing at all. An example of face generation with VAE is available here: <https://github.com/podgorskiy/VAE>. The reason for poor generation may be that the model is unable to learn the true posterior distribution using variational inference. Refer to the following figure:

Figure 12.9: two dimensional vectors z taken from square grid with bottom left [-1,-1], top right [1,1] and transformed to a handwritten digit image using trained VAE generator.

There are modifications of VAE like **Vector Quantised Variational Autoencoder (VQ-VAE)** [1], which can generate high-quality images. Here, a discrete codebook component is added to the network. The output of the encoder network is compared to all the vectors in the codebook and the nearest code is passed to the decoder.

Generative Adversarial Nets

Generative Adversarial Nets (GANs) is another class of generative models, designed by Ian Goodfellow in June 2014. GANs when used for image generation can produce high-quality images compared to vanilla VAE. GANs do not try to make any explicit density estimation, and with conditional GANs, we can enforce the generator output desired class of data. GAN architecture consists of two components a generator and a discriminator. The generator G , which takes a random vector z from a known distribution, transforms it to a data point X . The discriminator D is a binary classifier that outputs the probability of a data point X being chosen from the real data set. If we present a fake X that is synthesized using the generator G , then D should output a very low score. Both G and D are neural networks whose weights are randomly initialized. The two networks continuously update each other, becoming more smarted than the other. In practice, they are trained one at a time. Keeping the weights of G fixed, the weights of D are updated for some number of steps. Then, the other way, the weights of D are kept fixed and G is updated for a certain number of steps.

The discriminator network D wants to assign high probability to real images, that is, it wants to maximize log likelihood i.e., . D also wants to assign low probability to the generated images. Hence, it wants to minimize or equivalently, it wants to maximize the quantity , as shown in the following figure:

Figure 12.10: GAN

For a batch of samples, discriminators objective is as follows:

E denotes expectation. On the other hand, the generator network wants to improve its weight such that the generated images $G(z)$ get high score from the discriminator. For a batch of samples from generator and real data, the generators objective is as follows:

Note: If D is fixed, then the first term in the discriminator objective is constant. Adding this constant term to the generator objective is not going to affect the optimum; hence, write the objective for generator as follows:

If we define ,

then for fixed generator G , discriminator objective is .

For fixed discriminator D , generator objective is .

And we have minimax objective .

Combining generator and discriminator objectives, we have the following minimax objective function:

This is similar to two-player minmax game in Game theory. We can take G and D as two players.

In two player minimax games, like two person zero sum game, an equilibrium state exists where no player has the incentive to change their strategy. This is known as Nash equilibrium for zero sum minimax games. For GAN training, viewed as a game, is there any such equilibrium state? Yes; here, an equilibrium state will be attained when the discriminator does not need to change its weight and the generator also becomes an expert in creating a data point very close to the real data. Let's see if such an equilibrium state can exist theoretically.

Equilibrium state for GAN training

First, observe that there exists such a theoretical equilibrium state for this GAN game. The distribution of the samples $G(z)$ is a probability distribution defined by the generator G , where . So, at the equilibrium state, we should have the following:

Next, we will see whether our defined minimax objective function can attain that equilibrium point.

Given any *fixed* generator G , the minimax objective can be written as follows:

Here, the function inside the integral has the form . The function , attains its maximum at .

For fixed G , the discriminator cost will attain its maximum at the point . Hence, the optimal discriminator is given by the following:

Now, let's look at the generator. The generator wants to minimize:

$$C(G) =$$

The max in this expression for $C(G)$ is attained at , as shown in equation IV. We can write:

The generator wants to minimize $C(G)$ in VI, that is:

Suppose we attain this equilibrium point, that is, if , then , and hence, the minimum value of $C(G)$ is as follows:

Now, let's check if we can reach this best possible value of $C(G)$ at any other state of discriminator and generator where . Expanding expectations in equation VII:

Note that both the terms in the integral are in the KL divergence form. The first term is $KL()$, and the second term is $KL()$. The KL divergence of two distributions need not be symmetric. The symmetric version of KL divergence is called *Jensen–Shannon divergence (JS)*:

To write $C(G)$ in terms of Jensen–Shannon divergence, we need to introduce a factor of half in the log above. Readjusting equation IX by adding and subtracting $\log(1/2)$ from both the KL terms:

Since and we can write:

Now, and if if . Hence, the lower bound for $C(G)$ is . We have already seen that if , this lower bound is attained. Hence, global minimum of the generator training objective function $C(G)$ is attained if and only if that is the generative model starts perfectly replicating the data distribution.

Given that the generator and discriminators are neural networks, the preceding proof also suggests an algorithm for training these networks:

- Randomly initialize the weights of the generator and discriminator networks.
- (Keep Generator fixed to obtain D*) Freeze the generator and find an optimal discriminator for the given generator.
 1. Run k number of SGD steps to optimize discriminator weights
- (With fixed discriminator D*) Update Generator weights by optimizing $C(G)$ and keeping discriminator fixed. Note that only the second part of $C(G)$ will be used, as the first part is constant for fixed D.

Implementing GAN

The generator network for images can be made using a CNN-based architecture. The first layer of the network transforms the input noise vector to a three-dimensional array of dimension (height \times width \times channels). This is like a random image input, and then the deeper layers are de-convolution layers with every layer upscaling the image. **Deep Convolutional GANs (DCGAN)** architectures consist of a CNN-based generator and a CNN-based discriminator, as shown in [Figure 12.11](#). Both generator and discriminator typically have the same number of convolution layers, like autoencoder architectures. The discriminator layers will be placed in reverse order as that of the generator like a decoder. Refer to the following figure:

Figure 12.11: DCGAN Generator architecture

Following is the code implementing the generator and discriminator corresponding to the architecture shown in [Figure 12.11](#):

```

1. def generator(noise_dim):
2.     z = layers.Input(shape=noise_dim)
3.     x = layers.Dense(units=4*4*1024)(z)
4.     x = layers.Reshape((4, 4, 1024))(x)
5.     for filter_size in [512, 256, 128, 3]:
6.         x = layers.Conv2DTranspose(filters=filter_size,
7.                               kernel_size=5, strides=2, padding='same')(x)
8.         x = layers.LeakyReLU(0.2)(x)

```

```

9.         x = layers.BatchNormalization()(x)
10.        return Model(inputs = z, outputs=x)
1. def discriminator():
2.     img = layers.Input(shape=[64, 64, 3])
3.     x = layers.Conv2D(filters=128, kernel_size=5,
strides=2,
4.                         padding='same')(img)
5.     for filter_size in [256, 512, 1024]:
6.         x = layers.Conv2D(filters=filter_size,
kernel_size=5,
7.                         strides=2, padding='same')(x)
8.         x = layers.LeakyReLU(0.2)(x)
9.         x = layers.BatchNormalization()(x)
10.        x = layers.Flatten()(x)
11.        x = layers.Dense(1)(x)
12.        return Model(inputs = img, outputs = x)

```

Since both the generator and the discriminator objective functions are defined in terms of the discriminator logits, we have cross-entropy loss function for both discriminator and generator. Cross-entropy loss requires binary class labels, which is 1 for real image and 0 for fake image from the generator. Hence, we define GAN adversarial loss in terms of cross-entropy, as shown in the following code:

```

1. cross_entropy =
tf.keras.losses.BinaryCrossentropy(from_logits=True)
2. def discriminator_loss(real_output, fake_output):
3.     real_loss = cross_entropy(tf.ones_like(real_output),
real_output)
4.     fake_loss = cross_entropy(tf.zeros_like(fake_output),
fake_output)
5.     total_loss = real_loss + fake_loss
6.     return total_loss
7.
8. def generator_loss(fake_output):
9.     return cross_entropy(tf.ones_like(fake_output),
fake_output)

```

For computing gradients required to update the parameters, we can use two gradient tapes: one for generator and another for discriminator, as they are

updated separately. The two tapes will be used to calculate the gradients separately with respect to the discriminant parameters and generator parameters, as shown in the following code:

```
1. noise_dim = 100
2. G = generator(noise_dim)
3. D = discriminator()
4. noise = tf.random.normal([BATCH_SIZE, noise_dim])
5.
6. with tf.GradientTape() as gen_tape, tf.GradientTape() as
disc_tape:
7.     generated_images = G(noise, training=True)
8.     real_output = D(images, training=True)
9.     fake_output = D(generated_images, training=True)
10.
11.    gen_loss = generator_loss(fake_output)
12.    disc_loss = discriminator_loss(real_output,
fake_output)
13.
14.    gradients_of_G = gen_tape.gradient(gen_loss,
G.trainable_variables)
15.    gradients_of_D = disc_tape.gradient(disc_loss,
D.trainable_variables)
```

Here are the results, in [Figure 12.12](#):

Figure 12.12: DCGAN Generator output on Celeba faces dataset

[GAN training challenges](#)

Although there is a theoretical justification of the convergence of GANs, attaining the equilibrium state of this minimax game is quite hard in practice. Often, GAN training suffers three types of problems:

1. Models do not converge, and training process is unstable due to non-convex objective with continuous high-dimensional parameters.
2. **Mode collapse:** The generator produces a small variety of images, but the images are of good quality, so it's able to fool the discriminator. But such a generator has no use as it fails to represent the complex

real-world data distribution. All outputs of generator race toward a single point that the discriminator currently approves as realistic image. As possible reason for this is that as we minimize with respect to the generator and then maximize with respect to the discriminator, we hold the discriminator constant such that a single region in space is the point that is most likely to be real.

3. Vanishing gradient for generator leading to slow training. With a perfect discriminator, the model we will have $D(x) = 0$ for all images from generator; hence, the gradient presented to the generator is zero, and no update of generator happens.

Solutions for mitigating GAN training issues

Following are a few techniques to mitigate GAN training issues:

- **Feature matching:** The generator can be constrained to generate data that matches some descriptive statistics of the real data like mean and variance. We train the generator with an additional objective to match the expected value of the features on an intermediate layer of the discriminator. This will prevent the overfitting of the discriminant and mitigate the vanishing gradient problem. Let $f(x)$ denote activations on an intermediate layer of the discriminator. For DCGAN, we can take any of the fully connected dense layers at the end of the network. The new term in the objective function for generator is .

Let **conv2d_2** be the name of the second convolution layer in the discriminator with 512 filters. We can use this layer as $f(x)$, as shown in the following code, and update the generator loss:

```
1. from tensorflow.keras import Model  
2.         intermediate_D      =      Model(D.inputs,  
D.get_layer('conv2d_2').output)  
3.  
4. def generator_loss(real_output, fake_output):  
5.             features_fake      =  
tf.reduce_mean(intermediate_D(fake_output))  
6.             features_real      =  
tf.reduce_mean(intermediate_D(real_output))
```

```

7.
8.         return cross_entropy(tf.ones_like(fake_output),
fake_output)
9.                     + tf.square(tf.norm(features_fake-
features_real))

```

- **Minibatch discrimination:** To avoid mode collapse, we can make sure that the discriminator model looks at multiple generated examples in combination rather than one in isolation. It's based on the idea that a random sample from the real data set will have a diverse set of images, and hence, a minibatch of real images will be diverse. So, if we somehow measure intra-minibatch similarity of images for a true random sample, we should get a very low similarity score. The samples from generator should also have these characteristics if the generator is a good one. For measuring intra-batch similarity, output $f(x)$ from an intermediate layer of the discriminator is taken and then the layer output is projected to a low-dimensional space using random projection like distance-preserving operation.

Suppose $f(x)$ is L -dimensional vector, and we choose an orthogonal projection matrix M of dimension , then is a d -dimensional vector where .

For each sample of a minibatch of n samples, we now have a low-dimensional representation . For every sample we can compute L1 distance between and any other sample that is, . Now, similarity is inverse of distance, and hence, we can apply negative exponential to get similarity values, and we can represent the similarity between two samples and as To get a score for intra-minibatch similarity of the images, we can just add up all these similarity measures and get a real number for each image .

If a batch has similar looking images, we can expect these quantities to be very high and if the minibatch is diverse, we will have a very low value for all . Now, instead of one projection matrix, we can take a set of K such low-dimensional projection matrix such that we have a vector of intra-batch similarity scores for each image instead of a single scalar . To implement this, we pre-multiply by a different matrix K of dimension .

An alternate way for compact implementation of this is to stack all the K projection matrices and create a tensor T of dimension . For example, if K=2, we can create T from 2 random matrices M1 and M2, as shown in the following code:

```

1. M1 = tf.random.uniform([L,d]); M2 =
tf.random.uniform([L,d])
2. T = tf.concat([tf.expand_dims(M1, axis=1),
3.      tf.expand_dims(M2, axis=1)], axis=1)

```

Multiplying the vector f() by the tensor T, we get a matrix of dimension , and this can be done using Einstein summation operation in tensorflow, as shown in the following code:

```

1. fx = tf.random.uniform([n,L]); T =
tf.random.uniform([L,K,d])
2. features = tf.einsum('ij,jkl->ikl',fx, T)

```

Here, K rows have the K projections of f(xi) of dimension d. For each x_j , we get , and hence, for calculating pairwise distance, we can compute the L1 distance between the corresponding rows of the resulting matrix across samples , as shown in [Figure 12.13](#):

Figure 12.13: Minibatch Discrimination

Computation of L1 distance and then the negative exponentiation is shown in the following code:

```

1. row_wise_L1 = tf.abs(
2.      tf.map_fn(lambda x: x - 2. features,
3.                  tf.expand_dims(2. features, [1])))
4.
5. sim_scores = tf.exp(-tf.reduce_sum( row_wise_L1 ),
axis=[3]))
6. sim_score_out = tf.reduce_sum( sim_scores, axis=[1])

```

Hence, for the entire batch of size n, we have a set of n vectors of dimension K. We can concatenate these vectors to the intermediate layer output f(), and we feed the result into the next layer of the discriminator. Minibatch features are computed separately for samples from the generator and the training data. Now, if the generator generates a batch of similar looking images, the discriminator can

easily catch it from the minibatch discrimination features getting very high value.

- **Other cost functions:** For VAEs, we have used KL-divergence to define the loss function, and for GANs, we have used **Jensen–Shannon (JS) divergence** to define the loss function. With a simple example, we can see that both these measures have some properties that can inhibit stable learning with gradient decent. Wasserstein distance or earth mover's distance is another metric for quantifying similarity of two probability distributions. Using this metric, we can define another class of GANs called WGAN or Wasserstein GAN that shows better learning stability. Experimentally, it's observed that WGANs always avoid mode collapse.
- **Using class Label information in both discriminator and generator:** Conditional GANs mean that both the generator and discriminator can be conditioned on some kind of auxiliary information y . Here, y can be class labels or data from other modalities. For example, y can be an image description in free text form, or structured form, or y can be an image category.

Wasserstein GAN (WGAN)

WGAN is based on Wasserstein distance or **Earth Movers' (EM)** distance between the two distributions and that we encountered in [Chapter 4: Basic Statistics and Probability Theory](#). We have seen that training GAN with the minimax objective is equivalent to minimizing the JS-divergence: WGAN reformulates the optimization problem as a minimization of the Wasserstein distance between the real and generated distributions: and . EM distance is continuous and differentiable almost everywhere (that is, the set of points where it is not differentiable is very few). So, we can train the discriminator till optimality and avoid the vanishing gradient problem.

A probability distribution can be interpreted as a piling of unit mass of earth over a region. So, two different probability distributions represent two ways of piling up unit amount of earth over the region. The EMD is the minimum cost of turning one pile into the other. The cost of moving the pile is the amount of earth moved times the distance by which it is moved. The following definition is a formal way of stating the same.

Definition (Wasserstein distance or EMD): Let \mathcal{P} represent the set of all joint distributions x, y whose marginals are $p(x)$ and $q(y)$. Then, we define:

Here, the distance moved is given by the norm $\|\cdot\|_1$, and the amount of earth to be moved is given by the joint distribution π_{xy} .

WGAN network topology is the same as GAN. We have $\pi_{xy} \in \mathcal{P}$ and let the generator function G be π_{xy} , a parametrized function or a neural net with parameter θ , which transforms a Gaussian noise vector z to a point in X . Let π_z represent the distribution of z . For a fixed z , π_{xy} can be treated as a function of the parameters θ . So, WGAN objective is to $\min_{\theta} \mathbb{E}_{z \sim \pi_z} [\mathbb{E}_{x,y \sim \pi_{xy}} [g(x, y) - g(x, G(z))]$.

Some properties of EM distance

Let P be a fixed distribution over X and g be a parametrized function with parameter θ . Let Q represents the distribution of $g(z)$. For a fixed z , Q can be treated as a function of the parameters θ .

1. If g is continuous in θ , so is EMD (P, Q) :

Let θ and θ' be two parameters that are close in the parameter space, that is, $\|\theta - \theta'\|_1 < \epsilon$, as g is continuous in θ therefore, $g(\theta)$ and $g(\theta')$ should be very close by, that is, $\|\pi_{xy} - \pi_{xg(\theta)}\|_1 \leq \epsilon$.

EMD is defined as an infimum, so it's a lower bound, and hence:

EMD is a metric, so using triangle inequality:

$\|\pi_{xy} - \pi_{xg(\theta)}\|_1 \leq \|\pi_{xy} - \pi_{xg(\theta')}\|_1 + \|\pi_{xg(\theta')} - \pi_{xg(\theta)}\|_1$, as $\|\pi_{xg(\theta')} - \pi_{xg(\theta)}\|_1 \leq \epsilon$.

2. If g is locally Lipschitz continuous and satisfies 1, then EMD (P, Q) is continuous everywhere and differentiable almost everywhere (that is, it may not be differentiable on a small subset of points only).

Lipschitz continuity is a measure of how sensitive a function is to a small variation of the inputs. A function g is said to be locally Lipschitz at a point x if there exists an open ball $B_r(x)$ and a real number K such that:

Here, K gives an upper bound on the degree of perturbation of f for small perturbations in input measures as .

Here, ϕ is a neural network. To make Lipschitz, one crude method is truncating each element of the weight matrices by weight clipping to a small closed interval, like $[-0.01, 0.01]$. However, this restricts the capacity of the network. There are other advanced methods like gradient penalty and spectral weight normalization to enforce Lipschitz constraint in neural networks.

ϕ is locally Lipschitz at x if there exists an open set U around x such that for all :

The last part of the inequality holds for Euclidean norm. Taking $\epsilon = 1$, we have:

Now,

Therefore, ϕ is locally Lipschitz. Hence, ϕ is locally Lipschitz, and by Radamacher's theorem, we know that it has to be differentiable almost everywhere.

3. **1 and 2 are not true for JS and KL:** Let us take two simple distributions shown in [*Figure 12.14*](#), P and Q , which are uniformly distributed along the vertical y-axis with $b=1$ but are shifted by a positive number a . The probability of any sample from uniform distribution in interval $[a, b]$ is $= 1$ if $b=1, a=0$. We can write P, Q as follows:

P is uniform $[0, 1]$ along y-axis. Refer to the following figure:

Figure 12.14: Samples from distributions P and Q

Here, $P \neq Q$, since P and Q as they don't have any region in common, and where P is non-zero, Q becomes zero and vice-versa.

and , as this is the distance by which we need to move the unit probability mass.

At $\theta = 0$, $KL(P, Q)=KL(Q, P)=JS(P, Q) = EMD(P,Q) = 0$. But for all $0 < \theta < 1$ only EMD varies smoothly as a function of . Both KLS and JS have infinite discontinuity and a jump discontinuity at

This shows that KL and JS don't exhibit continuity and differentiability as EMD does; hence, EMD is better suited for neural net training compared to them.

WGAN training

An equivalent formulation of EMD can be derived using linear programming formulation and Kantorovich-Rubinstein duality. You may refer to section *Further reading [13]*.

Here, means f is a K-Lipschitz continuous, that is, for a small change in the input, denoted by , the change in f is bounded by K . $f(x)$ is K -Lipschitz; then:

Now, if we can make sure that the discriminator function in a GAN is 1-Lipschitz continuous ($K=1$) for any weight set w . The generator function be is a parametrized function or a neural net with parameter θ , which transforms a Gaussian noise vector z to a point in X . Using the preceding formula, we can rewrite loss formulation as *minimize EM(p_{data}, p_g)*:

Or,

The expectation E in the preceding equation can be approximated by the mean over a sample batch of size m , and hence, the gradient of the discriminator objective is given by the following:

where will be kept fixed. The gradient for the generator is written as . The implementation of generator and discriminator loss is shown here:

```

1. def discriminator_loss_wasserstein(real_output,
fake_output):
2.     return tf.reduce_mean(D(real_output)) -
tf.reduce_mean(D(fake_output))
3.
4. def generator_loss_wasserstein(real_output, fake_output):
5.     return -tf.reduce_mean(D(fake_output))

```

Here, one important assumption is overlooked. We have to make D satisfy Lipschitz constraint. This has been discussed in [Chapter 7: Neural Networks](#), as a part of adversarial learning.

Ensuring Lipschitz Constraint in Discriminator

Deep neural nets are very sensitive to their input. For example, a carefully chosen small perturbation of input image can mislead the neural network and significantly decrease its classification accuracy. A metric to evaluate the robustness of neural networks to small perturbations is the Lipschitz constant K, which upper bounds the relationship between input perturbation and output variation. In WGAN the following two methods are used to ensure Lipchitz condition:

1. **Weight Clipping:** Clamp the weights to a fixed box, say [-0.01, 0.01], after each gradient update. This can ensure Lipschitz condition. However, it takes many more iterations and time to train. This is because weight-clipping significantly limits the capacity of the network. This can be implemented as follows:

```

1. for p in D.trainable_variables:
2.     p.assign( tf.clip_by_value(p, -0.01, +0.01) )

```

2. **Gradient Penalty (GP):** Make sure that the gradient has norm at most 1. This can be achieved by adding an error term to the WGAN objective that enforces the gradient norm to lie close to unity: , where is some point between a real and a fake sample: , where and , both chosen independently at random. This is shown in line 4 in the following code. Alternatively, we can add small noise to the real data points and create perturbed input , where , and then the gradient penalty is implemented as earlier: .

```

1. D =discriminator()
2. def gradient_penalty(real_output, fake_output):

```

```

3.     epsilon = tf.random.uniform([real_output.shape[0],
1, 1, 1], 0.0, 1.0)
4.     x_hat = epsilon * real_output + (1 - epsilon) *
fake_output
5.     with tf.GradientTape() as tape:
6.         tape.watch(x_hat)
7.         d_hat = D(x_hat)
8.     gradients = tape.gradient(d_hat, x_hat)
9.     gradnorm_sqr_reg = tf.reduce_mean((tf.norm(gradients) - 1.0) ** 2)
10.    return gradnorm_sqr_reg

```

The GP loss term is added to the discriminator loss for enforcing Lipschitz condition in the discriminator.

Conditional GAN (cGAN)

We can implement conditioning of generator/discriminator by feeding auxiliary information y into both the discriminator and generator as additional input. Here, y generally comes as some categorical data; hence, a natural choice will be to use an embedding layer to encode y and then feed the encoded label into both discriminator and generator. For example, consider text to image generation, where y is text. We can use an embedding layer to encode the text representation. However, if the number of categories is few, we can avoid using embedding layer.

Now, the discriminator D will be judging not just X but a point (x, y) from the joint distribution (X, Y) . Also, the corresponding generator G models the conditional joint probability distribution , where z is a given noise vector. Given a noise vector z , we should have $G(z) = (X, Y)$. A more interesting and useful formulation of the generator is that we input the auxiliary information y as guidance to the generator G , stating what to generate, and then the generator takes a noise vector z as input to output the data point X conditioned on the auxiliary information. For example, in face generation, if Y represents a single Boolean variable gender, then setting $y=0$ the generator will generate a female face image. Here, generator models . Refer to the following figure:

Figure 12.15: Components of CGAN

Formally, we can define the conditional discriminator and conditional generator as follows:

We have a generator function , which takes a noise data and label embedding and outputs a data point . Also, we have a discriminator function that takes a datapoint and a noise embedding to output a probability score of whether the pair (x, y) came from the real distribution $p(X,Y)$. Here, our data distribution can be represented as a joint probability distribution . The generator wants to model the conditional distribution , where auxiliary data y follows a distribution . So, to yield a point from the joint distribution (x, y) , we can first sample y from and then use generator model to generate x from , that is, we have . The generator model is also conditioned on the noise z , and hence, the generator actually estimates . However, as Y and Z are independent, . So, we have:

Thus, we can modify our minimax loss function as follows:

At training time, we need to sample images from the generator to evaluate the two players G and D. This requires sampling from noise distribution, like before, for vanilla GANs and also sampling from the auxiliary data distribution .

If we draw sample of y directly from the training examples, the generator can reach a spurious optimum where it exactly reproduces the training data given a conditional input. So, we don't get any new data point generated and our generator acts as a database for the training data reproducing each training example from the given data set. To avoid this, we can draw y from a data distribution model trained using any classical density estimation technique, like a Parzen's window estimate, using the conditional values drawn from the training data. The following is an implementation of conditional discriminator for CelebA dataset. CelebA comes with a list of 40 binary attributes for each face image. These attributes include gender. We can extract that as the class label, and we have a binary class label as

the auxiliary information. The following is the discriminator code for conditional GAN:

```
1. def discriminator ():
2.     img = layers.Input(shape=[64, 64, 3])
3.     label = layers.Input(shape=[2,]) #for two classes
4.     y = layers.Dense(64*64)(label)
5.     x = layers.LeakyReLU(0.2)(x)
6.     y = layers.Reshape((64, 64, 1))(y)
7.
8.     x = layers.concatenate([img, y])
9.
10.    x = layers.Conv2D(filters=128, kernel_size=5,
11.                      strides=2,
12.                      padding='same')(x)
13.    for filter_size in [256, 512, 1024]:
14.        x = layers.Conv2D(filters=filter_size,
15.                           kernel_size=5,
16.                           strides=2,
17.                           padding='same')(x)
18.        x = layers.LeakyReLU(0.2)(x)
19.        x = layers.BatchNormalization()(x)
20.    return Model(inputs = [img, label], outputs = x)
```

In *line 8* of the preceding code, we are adding the auxiliary information of a new channel as the fourth channel, along with three channels of the input image. The rest of the code remains the same as before for the vanilla discriminator. Similarly, for the conditional generator, we take the first convolution layer as is in the vanilla generator to create the first 3D tensor of size $4 \times 4 \times 1024$ and then append the embedding layer of auxiliary information after mapping and reshaping it to $4 \times 4 \times 1$ as another channel.

The training loop remains the same as for the vanilla GAN, except that the generator also needs fake labels as input, and the discriminator requires real label as input. The rest of the code remains the same, as shown in the following code:

```
1. def train_step(images, labels):
```

```

2.     noise = tf.random.normal([BATCH_SIZE, noise_dim])
3.
4.     with tf.GradientTape() as gen_tape, tf.GradientTape()
as disc_tape:
5.         generated_images = G(noise, training=True)
6.         fake_labels = np.random.randint(0, 2, BATCH_SIZE)
7.
8.         real_output = D([images, labels], training=True)
9.         fake_output = D([generated_images, fake_labels],
training=True)

```

Conditional GANs are also used for image-to-image translation tasks. Image-to-image translation is the task of taking images from one domain and transforming them to another image so that they have the characteristics of images from another domain. Instead of passing auxiliary data as input, we can pass image from source image to condition the generator. Also, the auxiliary information being so rich, the generator can take source image as input directly and random noise vector is not required at all. To generate a good quality image output now, the simple CNN architecture we used so far may not be sufficient. We can use architectures with higher capacity, like U-Net or Resnet-50, to create the conditional generator. Pix2Pix model is a cGAN where output generation is conditioned on an input source image, and they use U-Net architecture for generator.

Cycle GAN (CycleGAN)

CycleGAN is a GAN architecture that uses two generators and two discriminators and is primarily applied to various image-to-image translation tasks where paired image data from two domains are not available.

For example, if we define source image domain as a set of natural images. We take a hand-drawn image of a beach where we spent our holiday in childhood. How nice would it be if we could input this hand-drawn image to the generator and get a photo realistic beach image with our childhood memories! Image translation is a classical computer vision problem, and all the traditional approaches to solve this involves using paired training data set, that is, a data set that has pairwise images from two different domains. Examples of paired data sets are (1) pair of satellite image and Google map

image, and (2) facades data set that consists of 506 Building Facades and the corresponding segmentations. However, getting paired image data set is not always possible; this is where CycleGANs can be useful. Let's now formally introduce CycleGANs.

Let \mathcal{X} and \mathcal{Y} be data sets with image samples from two different domains. We can denote the data distributions as $p_{\mathcal{X}}$ and $p_{\mathcal{Y}}$. Let's define two generators G and F . Note that unlike the previously defined generators, the generators here take an image as input directly and not a random noise vector. Also, we define two adversarial discriminators D_X and D_Y , where D_X 's goal is to distinguish between images in \mathcal{X} and translated images $\{G(x)\}$, and D_Y 's aims to discriminate between images in \mathcal{Y} and $\{F(y)\}$.

So, now we have four neural networks to train: G, F, D_X, D_Y . We can define a joint objective function for training this network. Also, we have to state the steps for training these networks. As there are two separate GANs, we have two adversarial losses:

and

This objective will also have the same issues as seen in vanilla GAN adversarial loss formulation, like vanishing gradient and mode collapse. Similar to the feature matching technique we used in vanilla GANs, we can enforce the generators F and G to be consistent, that is, if $x \rightarrow G(F(x))$, then transforming x using G and then applying F on the transformed image, we should get an image very close to x . If $G(F(x)) \approx x$. Refer to the following figure:

Figure 12.16: CycleGAN

So, L_G should be minimized, that is, L_D should be minimized and similarly, L_F should be minimized. We call the total quantity **cycle consistency loss**:

Hence, the loss for CycleGAN can be written as adversarial loss + cycle consistency loss: $L = L_G + L_F + \lambda L_C$. Here, λ is a hyperparameter that controls how much weight should be given to the cycle consistency loss. For better colour preservation in the output, another loss term was proposed, which is optional and is called identity loss term. This is also a pixel wise L1 loss

term, like $\|G(x) - x\|_1$. The idea behind this is that for the generator, if we take an image from the target domain, we should leave it as is because this image already belongs to Y , and hence, no transformation is required. Ideally, and , and hence, we should minimize the term . So, our loss has one more term and one more hyperparameter m :

Here, the adversarial loss terms can be replaced by Wasserstein loss for more stable training of the cycle GAN.

Here are the steps for training cycle GANs:

1. Choose hyperparameters l, m .
2. Randomly initialize the weights of the generator and discriminator networks.
3. Freeze the generator and find an optimal discriminator for the given generator:
 - a. Take a batch of images and
 - b. Run SGD to optimize discriminator weights by computing gradient of the adversarial loss for discriminator.
4. Update Generator weights by optimizing the generator loss and keeping discriminator fixed:
 - a. For images and , compute:
 - i. Cycled x for all x in
 - ii. Cycled y for all y in
 - iii. Same x for identity loss: for all y in
 - iv. Same y for identity loss: for all x in
 - v. Compute all three loss components and then the total generator loss gradient.
5. Repeat 3 and 4 until convergence.

TIP: The following link provides an implementation of cycle GAN using U-Net generator and discriminator:

<https://www.tensorflow.org/tutorials/generative/cyclegan>

Autoregressive generative models

Most generative models discussed so far are suitable for image generation. Can we generate audio or music, speech, cursive handwriting, or literary text, like poetry? All these data types are sequential in nature. What we generate at a given instance is dependent on what we generated in the previous instant. For example, in natural language processing, a language model takes a certain set of words or characters as the input context to generate. Recurrent neural networks and their variants, like **Long Short-Term Memory (LSTM)**, are best suited for these tasks. Similarly, for speech synthesis, we can use sequence to sequence models that can generate speech from text. Traditionally, **Hidden Markov Model (HMM)** were used for modelling sequences. HMMs are generative sequential models. We have discussed some of these models in the previous chapters. Now, let's look at some more generative models for sequences.

Music generation problem has similarity with text generation. A musical “*note*” is a symbol denoting a musical sound. Notes are the building blocks of written music, and they represent the pitch and duration of a sound in musical notation. The music generator model should take both pitch and duration information as input at every timestep and generate an output note for the next timestep. This is looped back as input, and a sequence of notes generated. So, LSTMs can be used here as well, to build simple music generator model. Combining the power of RNNs to model sequence with GAN, a hybrid model was proposed called C-RNN-GAN (Continuous RNN-GAN). This is trained with adversarial loss to model the joint probability of a sequence and can generate sequences.

Let's understand how the adversarial loss is calculated for sequences. The generator network is a RNN (stacked LSTM), which takes random vector as input that is concatenated with the output of previous cell . The output of the generator cell is a note vector obtained from a fully connected dense layer output . The discriminator is a bi-directional LSTM, which takes context in both directions into account and outputs the probability of whether an input sequence to the discriminator is real or fake, that is, whether it's real music data or its generated using the generator network. This is shown in [Figure 12.17](#):

Figure 12.17: Continuous RNN GAN

Here, the generator is autoregressive model, as shown in detail in [Figure 12.18](#). The discriminator encodes the sequence input into a dense fixed length representation using stacked bidirectional RNNs, and then the fixed encoded vector is connected to a sigmoid dense layer to output the probability of the sequence being real/fake.

The adversarial loss function in this case is an extension of the normal adversarial loss to sequences. Here, the adversarial loss for each element of the sequence is computed and then averaged to get the adversarial loss for the entire sequence of arbitrary length m.

Refer to the following figure:

Figure 12.18: Details of the recurrent autoregressive generator and the recurrent discriminator in C-RNN-GAN

There are other autoregressive GAN models, like MuseGAN [2], which can generate polyphonic music.

Autoregressive generative models are also applied on images. The pixels of an image can be viewed as a sequence starting from top left and then traversing the image matrix row wise or along the diagonal. Here, each pixel probability is conditioned on the previous pixels. PixelRNN [3] is a neural network architecture that consists of 12 fast, two-dimensional LSTM layers. They use residual connections around LSTM layers for training this very deep network.

Applying generative models

Let's discuss some real-world applications of generative models:

- **Super Resolution (SR):** Image super-resolution is a process of recovering high-resolution images from low-resolution images. It has a wide range of real-world applications such as medical imaging, video conferencing, video surveillance and security. Traditionally, SR was performed by various image rescaling techniques in computer vision, like Bicubic Interpolation, Pyramid Pooling, and Wavelet

Transformation. At present, deep neural net models like FSRCNN- and GAN-based models like SRGAN[9] and ESRGAN[10] have outperformed all these traditional approaches. To train such models, we need a data set of pairs of low- and high-resolution images. We can treat the SR model as a generator, which takes a low-resolution image as input and outputs a high-resolution image. The discriminator judges whether the input image is generated image or actual.

- **Synthetic Tabular Data Generation:** Tabular/structured data is one of the most common enterprise data modalities. Mostly, such data has **Personally Identifiable Information (PII)**. The data analyst must be very careful while publishing data analysis results so that no PII is disclosed. This is crucial for staying compliant with privacy regulations. This may restrict the analyst from publishing many important insights of the data. The ability to use synthetic datasets whose distribution is the same as true enterprise data ensures that PII are not disclosed. **Tabular GAN (TGAN)** [4] generates high-quality and fully synthetic tabular data, including both discrete and continuous columns.

Text to Speech Generation: GAN-TTS [5] is a generative network for generating **Text To Speech (TTS)**. Most neural network models for TTS use an autoregressive generator that is slow at inferencing, but GAN-TTS uses a convolutional feed forward network as the generator and is very efficient at inference time. Here, an ensemble of discriminators is trained instead of one to criticize different aspects of the audio generated, and the result is a high-fidelity audio.

- **Text to Image generation:** DALL·E [11] takes a piece of text and optionally, a part of an image, and it will output an image. It either continues the image whose part is given, or it generates the image by itself. It's trained on a data set of text–image pairs. It can create plausible images for a wide variety of sentences. It has the ability to combine disparate ideas to synthesize objects, some of which are unlikely to exist in the real world. So, we can apply such models for interior designing, fashion designing. Here, VAE is used to encode the input to a discrete latent space, and then transformer-based autoregressive decoder is used to construct the output.

- **Anomaly detection:** Many practical business problems like fraud detection, intrusion detection, system failure prediction can be formulated as anomaly detection. Also, anomaly detection and elimination are crucial steps in data analysis. Anomalies are rare examples in data. The general approach to detect anomalies is to learn the normal distribution and then find a way of assigning a score to each data point. As anomalies are rare, the learned distribution will give very low score to them. GANs and VAEs also try to learn the data distribution and represent data in a latent space. So, they have been successfully employed to solve reconstruction-based anomaly detection problem. Here is a survey paper on various GAN-based anomaly detection approaches [7]. There is a very recent study on unsupervised timeseries anomaly detection using LSTM-based generator called *TagGAN* [6].
- **Image-to-Image Generation:** *ArchiGAN* [8] is a cGAN for apartment building design. It learnt topological features and space organization directly from floor plan image.

Conclusion

In this chapter, we presented the core theory of deep learning-based generative models and a few applications. In the latest applications of GANs and VAEs, we may find different architectures being tried out for the generator/decoder, which may be suitable for a particular problem domain. However, the loss functions for training such networks are still some derivatives of the known loss functions that we discussed here, like adversarial loss, reconstruction loss, ELBO, feature mapping, cycle consistency loss, and Wasserstein loss. This chapter should enable the readers to explore the state-of-the-art papers in generative modelling with cool applications and implement them for solving their own business problems.

Points to remember

- Generative models are primarily unsupervised machine learning models.

- Generative models can generate new data instances by learning underlying probability distribution of data, whereas discriminative models discriminate between different kinds of data instances by learning class conditionals.
- VAE and VQ-VAEs are easier to train compared to GANs; moreover, the encoder part of the VAE can be used as a dimensionality reduction technique for representing data in low-dimensional latent space.
- Training GAN with adversarial loss is equivalent to minimizing the JS-divergence, whereas training VAE boiled down to minimizing KL-divergence between true data distribution and generator data distribution.
- We have theoretically proved the existence of equilibrium for GAN, but achieving this equilibrium numerically while training with adversarial loss is hard. We may get stuck in many *local* Nash equilibrium state, which leads to mode collapse and unstable learning. These local equilibria are far away from global equilibrium, and hence, we should always use some of the tricks discussed here, like feature mapping, minibatch discrimination, and conditioning GANs to mitigate these challenges.
- If we have sequential data, it's better to use sequential generative models like HMM or recurrent neural network-based models, like C-RNN-GAN.

Further Reading

1. VQ-VAE: <https://arxiv.org/abs/1711.00937>
2. MuseGAN: Multi-track Sequential, <https://arxiv.org/abs/1709.06298>
3. PixelRNN :
https://slazebni.cs.illinois.edu/spring17/lec13_advanced.pdf
4. Tabular GAN (TGAN): <https://arxiv.org/pdf/1811.11264.pdf>
5. TTS-GAN: <https://arxiv.org/pdf/1909.11646.pdf>
6. TagGAN: <https://arxiv.org/pdf/2009.07769v3.pdf>
7. Anomaly Detection: <https://arxiv.org/pdf/1906.11632.pdf>

8. ArchiGAN: <https://developer.nvidia.com/blog/archigan-generative-stack-apartment-building-design/>
9. SRGAN for super resolution: <https://arxiv.org/abs/1609.04802>
10. ESRGAN : <https://arxiv.org/abs/1809.00219>
11. DALL.E: <https://openai.com/blog/dall-e/>
12. <https://www.kaggle.com/datasets/abhilash04/fathersandsonheight>
13. <https://vincentherrmann.github.io/blog/wasserstein/s>

Index

A

AdaGrad [269](#)
Adaptive Moments (Adam) [269](#)
agglomerative clustering [298](#)
 BIRCH [300](#)
 distance between clusters [299, 300](#)
Akaike Information Criterion (AIC) [20](#)
AlexNet [356](#)
application
 on real dataset [60-63](#)
Arithmetic Mean [155](#)
artificial general intelligence (AGI or strong AI) [3](#)
Artificial Intelligence (AI) [1](#)
 applications [25, 26](#)
 role of mathematics [26-28](#)
 systems [2](#)
artificial narrow intelligence (ANI or narrow AI) [2](#)
artificial neuron [252-255](#)
artificial super intelligence (ASI) [3](#)
attention mechanism [405](#)
 Bahdanau attention [406](#)
 concat [406](#)
 dot product [406](#)
 general [406](#)
 key-value-query formulation [407, 408](#)
 language translation model [409-414](#)
 local attention [407](#)
 location based [406](#)
 speech recognition model [414-416](#)
augmented matrix [38](#)
Autocorrelation Plot (ACF) [381](#)
autoencoder [315, 316](#)
automatic differentiation [267](#)
Automatic Speech Recognition (ASR) [414](#)
autoregressive generative models [486-488](#)
Autoregressive Integrated Moving Average (ARIMA) models [380](#)
average filter [348](#)
average pooling [351](#)

B

backpropagation algorithm [264-267](#)
backtracking line search [141](#)

Bag of Words (BoW) model [429](#)
Balanced Iterative Reducing and Clustering using Hierarchies (BIRCH) [300](#)
basis vectors [50](#)
Bayesian Decision Theory [173-175](#)
Bayesian Information Criterion (BIC) [20](#)
Bayes theorem [172, 173](#)
Bernoulli [177](#)
Bi-directional RNNs (BRNN) [398](#)
bigram model [433](#)
binary cross entropy error [241](#)
binary random variable [176](#)
Binomial distribution [178](#)
bivariate analysis [164](#)
Bolzano theorem [108](#)
bottleneck [315](#)

C

categorical-cross-entropy error [243](#)
categorical random variable [177](#)
Cauchy-Schwartz inequality [188](#)
Cayley–Hamilton theorem [86](#)
central or centroid vector [287](#)
central tendency [155](#)
central tendency measures [155](#)
 mean [155, 156](#)
 median [156](#)
 mode [156-159](#)
Characteristic equation [86](#)
Characteristic polynomial [86](#)
classification [13](#)
classification model
 classification accuracy [15](#)
 class-wise accuracy [15](#)
 confusion matrix [16](#)
 evaluation metrics [14, 15](#)
 F1 score [16](#)
 precision [15](#)
 recall [15](#)
Receiver Operating Characteristic (ROC) curve [17](#)
True Positive (TP) [14](#)
clustering [10, 275, 276](#)
 hard clustering [277](#)
 soft clustering [277](#)
clustering algorithms
 agglomerative clustering [298](#)
 density-based clustering [290](#)
 distribution-based clustering [293](#)
 Fuzzy theory-based clustering [302](#)
 graph-based clustering [300, 301](#)

hierarchical-based clustering [297](#)
partition-based clustering [287](#)
clustering analysis [275](#)
cluster quality [283](#)
CNN architecture [349](#)
 convolution layer [350](#), [351](#)
 depthwise convolution [353](#)
 depthwise separable convolution [352](#), [353](#)
 optimization [354](#)
 pointwise convolution [353](#)
 pooling layer [351](#), [352](#)
 spatially separable convolution [352](#)
 transposed convolution [354](#), [355](#)
CNN development [356](#)
 AlexNet [356](#), [357](#)
 inception [359](#), [360](#)
 ResNet [361](#), [362](#)
 TensorFlow Model [358](#)
 trainable parameters, counting [359](#)
 VGG [360](#), [361](#)
 Xception [362](#), [363](#)
CNN models application [363](#)
 image classification [363](#), [364](#)
Coefficient of Dispersion (C.D) [162](#)
Coefficient of Variation (C.V) [162](#)
Computer Vision (CV) [325](#)
Conditional GAN (cGAN) [480-483](#)
Connectionist Temporal Classification (CTC) [399-401](#)
 DP formulation, for CTC loss [403](#)
 inferencing from [403](#)
 limitations [404](#)
 training [401](#), [402](#)
constraint [143](#)
contextual models [439](#)
 BERT [440](#), [441](#)
 Bidirectional Encoder Representations from Transformers (BERT) [440](#)
 ELMo model [439](#), [440](#)
 ERNIE model [446](#)
 Generative Pre-Training (GPT) [447](#)
 input presentation, for pre-training BERT [443](#)
 position encoding [441](#), [442](#)
 pre-training BERT [442](#)
 WordPiece tokenization [443-445](#)
Continuous Bag-of-Words (CBOW) model [438](#)
continuous entropy [200](#)
continuous functions [107](#), [108](#)
 Bolzano theorem [108](#)
 intermediate value theorem [108](#)
continuous probability distributions [179](#), [180](#)
contracted axes [128](#)

convex combination [50](#)
convex function
 properties [147](#), [148](#)
convex functions [146](#)
Convolution Neural Networks (CNN) [272](#), [273](#), [349](#)
convolution operation [334](#)
correlation [164](#), [165](#)
correlation matrix [74](#)
covariance [165](#)
covariance matrix [74](#)
 calculating [311](#)
 decomposition [312](#), [313](#)
Cramer-Rao inequality [219](#), [220](#)
cross-validation [23](#)
Cumulative Probability Distribution Function (C.D.F) [180](#)
Cycle GAN (CycleGAN) [484-486](#)

D

data [154](#)
 learning from [9](#)
 qualitative [154](#)
 quantitative [154](#), [155](#)
data collection [7](#)
dataset preparation [22](#), [23](#)
data types [8](#), [9](#)
Davies-Bouldin indicator [283](#)
DBSCAN [290-292](#)
 advantages [292](#)
 limitations [293](#)
decision tree [203-208](#)
 leaf node [204](#)
Decoder [315](#)
Deep Convolutional GANs (DCGAN) [468](#)
deep learning [21](#), [22](#)
Deep Neural Networks (DNN) [349](#)
density-based clustering algorithms [290](#)
 DBSCAN [290-292](#)
derivative of function [109](#)
derivative of scalar fields, w.r.t. vector [116](#)
 directional derivative [116](#), [117](#)
 geometry of gradient vector [120](#)
 partial derivative [116-118](#)
 total derivative [118](#), [119](#)
derivative of vector fields, w.r.t. vector [121](#)
 chain rule [122](#)
 dot product of tensors [127-129](#)
 Einstein notation [124-127](#)
 matrix form of chain rule [122](#), [123](#)
 tensor calculus [129](#), [130](#)

tensors [123](#), [124](#)
total derivative of tensor [130-134](#)

determinant
inverse of matrix [78](#), [79](#)
of square matrix [77](#)

differential entropy [200](#)

differentiation [109](#)

digital image formation [326](#)
light, capturing [326](#)
quantization [327](#)
sampling [327](#)

dimensionality reduction [10](#), [307-309](#)

dimension of space [50](#)

dimension of subspace [52](#)

directional derivative [117](#)

discrete probability distribution [176](#), [177](#)

dispersion [161](#)

dispersion measures
coefficients of dispersion [162](#), [163](#)
Interquartile Range [161](#)
Mean deviation (MD) [161](#)
range [161](#)
standard deviation [162](#)

dissimilarity function [277](#)

distance and similarity metrics [277](#), [278](#)

distribution-based clustering [293](#)
Gaussian Mixture Model (GMM) [294-297](#)

Document-Term matrix [430](#)

dot product [48](#), [127](#)

dual problem [146](#)

Dunn indicator [284](#)

E

eigen decomposition [96](#), [97](#)
real symmetric matrix [97](#)
singular value decomposition [97](#)

eigenvalues
and vectors [86](#)

Einstein notation [124](#)
examples [125](#)

ellipsoid [91](#)

EM algorithm [296](#), [297](#)

Embeddings from Language Model (ELMo) [439](#)

EM distance
properties [476-478](#)

Encoder [315](#)

encoder decoder architecture [404](#)

Enhanced Representation through kNowledge IntEgration (ERNIE) [445](#), [446](#)

error derivatives

computing [259-264](#)
estimator properties [218, 219](#)
Euclidean ball [46](#)
Euclidean distance [282](#)
Euclidean space [44](#)
vectors [44](#)
Evidence Lower Bound (ELBO) [458](#)
Expectation Maximization (EM) [294](#)
expert system [4](#)
exponential distribution [185, 186](#)
Exponential Weighted Moving Average (EWMA) [269, 379](#)
external evaluation [285, 286](#)

F

False Negative (FN) [15](#)
False Positive (FP) [15](#)
False Positive Rate (FPR) [17](#)
feature engineering [29](#)
feature matching [472](#)
feedforward neural network [255, 256](#)
hidden layers [255](#)
input layer [255](#)
layer [255](#)
output layer [255](#)
filtering [329](#)
finite-dimensional space [50](#)
F-measure [286](#)
forming clusters [277](#)
forward substitution [40](#)
forward substitution, in matrix [95](#)
Fowlkes-Mallows index (FM) [287](#)
frequency distribution [155](#)
Frobenius norm [58](#)
full padding [350](#)
Fully Connected (FC) layer [357](#)
function optimization [134](#)
decent methods [138-142](#)
maxima [135](#)
minima [135](#)
saddle point [135-138](#)
with constraints [143-145](#)
with inequality constraints [145](#)
Fuzzy C-Mean algorithm (FCM) [303](#)
Fuzzy theory-based clustering [302, 303](#)
Fuzzy c-means [303](#)

G

Gated Recurrent Unit (GRU) [388, 390](#)

Gaussian distribution [181-185](#)
Gaussian kernel [336](#), [337](#)
 discrete approximation [337-339](#)
 Gaussian filter, applying [339](#), [340](#)
Gaussian Mixture Model (GMM) [293](#), [294](#), [449](#)
Gauss-Jordan elimination [40](#)
 by-product [95](#)
Generalized Bernoulli distribution [177](#), [178](#)
Generative Adversarial Nets (GANs) [464](#), [465](#)
 challenges, for training [471](#)
 equilibrium state [465-468](#)
 implementing [468-471](#)
 solutions, for mitigating training issues [472-475](#)
Generative Adversarial Network (GAN) [273](#), [449](#)
generative model [450-454](#)
 handwriting generation [392](#), [393](#)
 for sequence [391](#)
 Mixture Density Networks (MDN) [393-396](#)
Generative Pre-Training (GPT) [447](#)
Geometric Mean [156](#)
geometric spatial transformation [329-332](#)
GoogLeNet [359](#)
Gradient Penalty (GP) [479](#)
GradientTape function [131](#)
Gram-Schmidt process [79](#)
graph-based clustering [300](#)
 Spectral clustering [301](#), [302](#)
graphviz [206](#)

H

halfspaces [53](#)
Harmonic Mean [156](#)
Hidden Markov Models (HMM) [383](#), [384](#)
hierarchical-based clustering [297](#)
hyperplane [52](#), [53](#)
hypotheses [215](#)

I

identity matrix [67](#)
ILSVRC [356](#)
image classification [363](#), [364](#)
 image segmentation [365](#)
 instance segmentation [365](#)
 object detection [364](#)
 semantic segmentation [365](#)
U-Net, using [366](#)
image derivative-based kernels [341](#)
Laplacian kernel [342-346](#)

Sobel kernel [346-348](#)
inconsistent systems [33](#)
index notation or indicial notation [123](#)
inference [211](#)
infinite-dimensional space [50](#)
information theory [198, 199](#)
 entropy [199, 200](#)
 KL divergence [201, 202](#)
 mutual information [202, 203](#)
 relative entropy [201](#)
intermediate value theorem [108](#)
internal evaluation [283](#)
Interquartile Range (IQR) [161](#)
intra-attention [415](#)
Inverse Document Frequency (IDF) [431](#)
invertible matrices [68, 69](#)
Iris autoencoder
 building [316-318](#)
Iris dataset
 loading [310, 311](#)
Iterative Dichotomiser 3 (ID3) [205](#)

J

Jaccard index [287](#)
Jacobian and Hessian matrix [74](#)
Jacobian matrix [121](#)
Jensen-Shannon (JS) divergence [202](#)
Jensen–Shannon (JS) divergence [474](#)
Jenson's Inequality [188](#)
joined probability distributions [188-192](#)

K

Karl Pearson's coefficient of correlation [165](#)
Karush-Kuhn-Tucker (KKT) [150](#)
k-fold cross validation [24](#)
K-means clustering algorithm [288](#)
 challenges [289](#)
 elbow method [288](#)
K-medoids [289](#)
k-nearest neighbour (KNN) algorithm [63-65](#)
kurtosis [163](#)

L

Lagrange dual function [145](#)
Lagrange multipliers [144](#)
language translation model [409-414](#)
Laplacian kernel [342-346](#)

Laplacian of Gaussian (LoG) [344](#)
Large Sample Theory [212](#)
 hypothesis testing [215-217](#)
 sample statistics [213, 214](#)
 sampling, from known distributions [215](#)
Latent Dirichlet Allocation (LDA) [435-438, 449](#)
Latent Semantic Indexing (LSI) model [431-433](#)
layer.kernel property [262](#)
leave-p-out cross validation [24](#)
linear algebra [29](#)
linear combination [49, 50](#)
linear equation [30-33](#)
 inconsistent system [36, 37](#)
 infinitely many solutions [35, 36](#)
 system of linear equations, solving analytically [34](#)
linearly dependent vectors [49](#)
Linear mapping [82](#)
Linear Models (LM) [239](#)
linear transformation [82](#)
 composition [84, 85, 86](#)
 eigen properties [87, 88](#)
 eigen properties, of symmetric matrices [90, 91](#)
 eigenvalues [86](#)
 example [82](#)
 geometric analysis [89](#)
 matrix, with linear map [83, 84](#)
 positive definite [91-93](#)
 zero eigenvalue [90](#)
Long Short-Term Memory (LSTM) [388, 389](#)
lower triangular matrix [67](#)
LU decomposition [93, 94](#)

M

Machine Learning (ML) [4-6, 211](#)
Mahalanobis distance [195, 280, 281](#)
major axis [91](#)
Manhattan distance [57](#)
MA(q) model [381](#)
Markov chain [381-383](#)
mathematical expectation
 example [186](#)
 of random variable [186, 187](#)
 properties [188](#)
matrices, in ML problem formulation
 correlation matrix [74](#)
 covariance matrix [74](#)
 distance matrix [73](#)
 feature/data matirx [72](#)
 gram matrix [73](#)

Jacobian and Hessian matrix [74](#)
one hot encoding [72, 73](#)
matrix [38](#)
augmented matrix [38-40](#)
basic matrix operations [41-44](#)
identity matrix [67](#)
inverse properties [69, 70](#)
invertible matrices [68, 69](#)
null space [75](#)
orthogonality [74, 75](#)
orthogonal matrix [70, 71](#)
permutation matrix [70](#)
pseudocode back substitution [41](#)
pseudocode forward substitution [40](#)
rank [67](#)
skew-symmetric matrices [68](#)
subspaces [74](#)
symmetric matrix [67, 68](#)
types [67](#)
vectors, representing [66](#)
matrix decomposition [93](#)
eigen decomposition [96, 97](#)
LU decomposition [93, 94](#)
QR decomposition [96](#)
matrix factorization [93](#)
matrix norms [58](#)
Frobenius norm [58](#)
inner product [59, 60](#)
norm ball [59](#)
Maximum Likelihood Estimation (MLE) [451](#)
Maximum Posteriori (MAP) [225](#)
max pooling [351](#)
Mean Absolute Error (MAE) [19](#)
Mean Squared Error (MSE) [19](#)
mean value theorem (Lagrange) [112](#)
measure of dispersion [155](#)
measure of skewness [155](#)
measures of central tendency [155](#)
measures of kurtosis [155](#)
median filter [348](#)
metric function [277](#)
metric induced by the norm [56](#)
Microsoft Research Paraphrase Corpus (MRPC) [444](#)
minibatch discrimination [472](#)
Minimax Bayesian Risk solution [175](#)
Minimum Variance Bound (MVB) estimator [222](#)
Minimum Variance Unbiased (M.V.U) estimators [219](#)
bias-variance decomposition [226](#)
Cramer-Rao inequality [220-223](#)
likelihood function [220](#)

Maximum Likelihood Estimation (MLE) [223-225](#)
Minkowski distance [279, 280](#)
Mixture Density Networks (MDN) [393-396](#)
ML algorithm
 reinforcement learning [11-13](#)
 supervised learning [13, 14](#)
 types [9](#)
 unsupervised learning [10, 11](#)
ML model
 building [7](#)
 data collection [7](#)
 data preparation [7](#)
 evaluation [8](#)
 feature extraction/selection [7](#)
 training [8](#)
ML problems, formulating as statistical inferencing
 bias variance trade-off [237-239](#)
 classification [227, 228](#)
 curvilinear regression [230-232](#)
 data distribution [226, 227](#)
 linear models, interpretability [244-247](#)
 linear regression [230-232](#)
 logistic regression [239-241](#)
 model parameters, estimating [232, 233](#)
 model parameters, iterative estimation [233-235](#)
 multiclass logistic regression [242, 243](#)
 Naive Bayes classifier [228](#)
 overfitting [235, 236](#)
 Poisson regression [243, 244](#)
 regression [229, 230](#)
 underfitting [235, 236](#)
moments [163](#)
multivariate distributions [193](#)
 multinomial distribution [194](#)
 multivariate Gaussian distribution [194-198](#)
multivariate probability density function [190](#)

N

natural language [424, 425](#)
 syntactic structure [425](#)
Natural Language Processing (NLP) [423](#)
Negative Binomial (NegBin) Model [244](#)
neighbor pixel operation [332-334](#)
neural language models [438](#)
neural network [21, 22, 251](#)
 challenges of training [267, 268](#)
 sensitivity, to small perturbations [272](#)
 training [257](#)
neural network architectures [272](#)

autoencoder architecture [272](#)
Convolutional Neural Network (CNN) [273](#)
Generative Adversarial Network (GAN) [273](#)
Recurrent Neural Nets (RNN) [273](#)
Siamese neural network [273](#)
transformers [273](#)
neural network training, challenges
 bias variance tradeoff [270](#)
 regularization, of neural nets [270](#)
 SGD modifications [269](#)
 slow training, with SGD [267](#), [268](#)
neuron [252](#)
Newton's method [142](#)
N-gram model [434](#)
non-linear filters [348](#)
 average filter [348](#)
 learning [349](#)
 median filters [348](#)
normal distribution [181](#)-[185](#)
null space [75](#)

O

object detection algorithms [364](#)
 R-CNN [365](#)
 YOLO [365](#)
odds [165](#), [166](#)
one hot encoding [73](#)
One-vs-Rest (OvR) scheme [242](#)
optimization, with inequality constraints [145](#)
 convex functions [146](#), [147](#)
 convex optimization [148](#), [149](#)
 Karush-Kuhn-Tucker conditions (KKT) [149](#), [150](#)
 Lagrange dual function [145](#), [146](#)
Ordinary Least Squares (OLS) [379](#)
orthogonal basis [50](#)
orthogonality among subspaces
 among subspaces [76](#), [77](#)
orthogonal matrix [70](#), [71](#)
orthonormalization [79](#), [80](#)
 applications [81](#), [82](#)
 example [81](#)
Out of vocabulary words (OOV) [428](#), [443](#)
overfitted model [23](#)
overfitting [235](#)

P

Partial Autocorrelation (PACF) plot [381](#)
partial derivative [117](#)

partition-based clustering algorithms [287](#)
 K-means [288](#)
 K-medoids [289](#)
Partitioning Around Medoids (PAM) [289](#)
partition values [159-161](#)
Parts Of Speech (POS) [384, 425](#)
permutation matrix [70](#)
petal width [162](#)
phones [424](#)
pixels [328](#)
 accessing [328](#)
Poisson distribution [179](#)
positional vector [44](#)
position encoding [441, 442](#)
positive-definite [91](#)
positively skewed [164](#)
preprocessing techniques [428](#)
pre-training BERT [442](#)
primal problem [146](#)
Principal Component Analysis (PCA) [309, 310](#)
 using [315](#)
 versus t-SNE [321](#)
principal components [313](#)
 reducing with [313, 314](#)
probabilistic models, of text [433](#)
 Latent Dirichlet Allocation (LDA) [435-438](#)
 neural language models [438](#)
 topic models [434](#)
probabilistic sequence models [381](#)
 Hidden Markov Models (HMM) [384, 385](#)
 Markov chain [381-383](#)
probability [165, 166](#)
Probability Density Function (P.D.F) [176](#)
Probability Mass Function (P.M.F) [176](#)
probability theory [153](#)
pseudocode forward substitution [40, 41](#)
Python sklearn DecisionTreeClassifier module [206](#)

Q

QR decomposition [96](#)
quadratic form [91](#)
qualitative data [155](#)
 nominal [154](#)
 ordinal [154](#)
quantitative data [155](#)
 continous [155](#)
 discrete [154](#)
quantization [327](#)
quartiles [159](#)

R

Rand Index (RI) [286](#)
random experiment [166](#)
conditional independence [170, 171](#)
conditional probability [168, 169](#)
events as sets [166-168](#)
independent events [169, 170](#)
random sampling [212](#)
random variable
 mathematical expectation [186](#)
 transformation [193](#)
Random Variable (R.V.) [175, 176](#)
range [161](#)
R-CNN [365](#)
real analysis [104](#)
 continuous functions [107, 108](#)
 derivative of function [109-112](#)
 fundamentals [104-106](#)
 higher order derivatives [112, 113](#)
 limit of function [106, 107](#)
 Taylor series expansion [113, 114](#)
real matrix [67](#)
real symmetric matrix [97](#)
real-world applications, generative models
 anomaly detection [489](#)
 Image-to-Image Generation [490](#)
 real-world applications [488, 489](#)
 Super Resolution (SR) [488](#)
 Synthetic Tabular Data Generation [489](#)
 Text to Image generation [489](#)
 Text to Speech Generation [489](#)
Receiver Operating Characteristic (ROC) curve [17](#)
Rectified Linear Unit (ReLUs) [356](#)
Recurrent Neural Network (RNN) [273, 386](#)
 Gated Recurrent Unit (GRU) [390](#)
 Long Short-Term Memory (LSTM) [389, 390](#)
 stacked LSTM/RNN [390, 391](#)
 structure [387, 388](#)
regression [13](#)
regression model
 evaluation metrics [18-20](#)
regularization [270](#)
 batch normalization [271](#)
 dropout [271](#)
 weight-decay [271](#)
 weight sharing [271](#)
Reinforcement Learning (RL) [11-13](#)
repeated random sub-sampling method [24, 25](#)
Restricted Boltzmann Machine (RBM) [449](#)

RL problem, components
action [13](#)
agent [13](#)
environment [13](#)
policy [13](#)
state [13](#)
RMSProp [269](#)
Rolle's theorem [112](#)
Root Mean Squared Error (RMSE) [20](#)

S

same padding [350](#)
sampling [327](#)
scalar field [115](#)
 derivative, w.r.t.vector [116](#)
 limit and continuity [116](#)
scalar multiplication [46](#)
second order derivative [112](#)
self-attention [415, 416](#)
 calculating [416-418](#)
semantics [427](#)
Sensitivity [15](#)
sepal length [162](#)
separable kernels [335](#)
 convolution with [335, 336](#)
sequence classification [397](#)
Sequence-to-Sequence (Seq2Seq) modelling [398, 399](#)
 attention mechanism [405-407](#)
 CTC [399-402](#)
 encoder decoder architecture [404](#)
 transformer architecture [418](#)
Siamese neural network [273](#)
Silhouette coefficient [284, 285](#)
singular or degenerate matrix [68](#)
singular value decomposition [97-100](#)
skewness [163](#)
skew-symmetric matrices [68](#)
skip-gram model [439](#)
Sobel kernel [346-348](#)
space [53](#)
span of vector [49](#)
spatial filtering [329](#)
 convolution operation [334](#)
 geometric spatial transformation [329-332](#)
 neighbor pixel operation [332-334](#)
special tensor [127](#)
Spectral clustering [301](#)
speech recognition model [414, 415](#)
square matrix [67](#)

stacked LSTM/RNN [390](#), [391](#)
standard deviation [162](#)
standard scaler [196](#)
statistical inference [211](#), [218](#)
Statistical Machine Translation (SMT) [409](#)
statistics [153](#), [154](#)
 descriptive statistics [154](#)
 inferential statistics [154](#)
Stochastic Gradient Descent (SGD) [257-259](#)
 adaptive learning rate [269](#)
 momentum methods [269](#)
stratified sampling [213](#)
strong duality [146](#)
subspaces [51](#)
supervised learning [13](#)
Support Vector Machines (SVM) [150](#)
symmetric matrix [67](#), [68](#)
synapses [253](#)
syntactic structure of language
 clause [425](#)
 Parts of Speech (POS) [425](#)
 phrase [425](#)
 sentence [426](#)
 text corpus [426](#)
 text document [426](#)
system of linear equations [31](#)

T

Tabular GAN (TGAN) [489](#)
Taylor series expansion [113](#), [114](#)
tensor [123](#)
term frequency [431](#)
Term Frequency (TF)-Inverted Document Frequency (IDF) [431](#)
text models [429](#)
 Bag of Words (BoW) model [429](#)
 contextual models [439](#)
 Latent Semantic Indexing (LSI) model [431-433](#)
 probabilistic models [433](#)
 vector space model [429](#)
text preprocessing [427](#)
time series models [374](#), [375](#)
 ARIMA model [380](#), [381](#)
 decomposition [375-377](#)
 differencing [378](#)
 exponential smoothing [379](#), [380](#)
 forecasting [379](#)
 OLS model [379](#)
topic model [434](#)
total probability theorem [171](#), [172](#)

transformation
of random variable [193](#)
transformer [273](#), [417](#)
 architecture [418](#)
triangular matrix [67](#)
trivial subspace
 example [51](#)
True Negative (TN) [15](#)
True Positive Rate (TPR) [15](#)
t-SNE [319](#), [320](#)
 in Iris dataset [321](#), [322](#)

U

underfitted model [23](#)
underfitting [235](#)
U-Net [366](#), [367](#)
uniform distribution [181](#)
unigram model [433](#)
unique solution [33](#)
univariate data analysis [164](#)
unsupervised learning [10](#)

V

valid padding [350](#)
variance retention [314](#), [315](#)
Variational Autoencoder (VAE) [449](#), [454-463](#)
vector [29](#), [44](#)
 addition/subtraction [47](#)
 direction [46](#)
 distance between [47](#)
 dot product [48](#)
 linear combination [49](#), [50](#)
 magnitude [45](#)
 natural orthonormal basis [51](#)
 norm [45](#), [46](#)
 orthogonal basis [50](#)
 orthogonality [48](#)
 orthonormal basis [51](#)
 representing, in matrix [66](#)
 representing vector [44](#), [45](#)
 scalar multiplication [46](#), [47](#)
 span [49](#)
 Unit vector [45](#)
vector calculus [103](#)
vector field [115](#)
 limit and continuity [116](#)
vector space [54](#), [55](#)
 defining [53](#)

lp norm [57](#)
matrix norm [58](#)
maximum norm [57, 58](#)
normed vector space [56](#)
norm of real numbers [56, 57](#)
vector space model [429](#)
count based or Boolean [430](#)
TF-IDF [431](#)
Visual Geometry Group (VGG) [356-360](#)

W

Wasserstein GAN (WGAN) [475](#)
Lipschitz Constraint, ensuring in discriminator [479, 480](#)
training [478, 479](#)
weak duality [146](#)
Weight Clipping [479](#)
With-In Cluster Sum of Squares (WCSS) [288](#)
WordNet [427](#)
WordPiece tokenization [443, 444](#)

Y

YOLO [365](#)