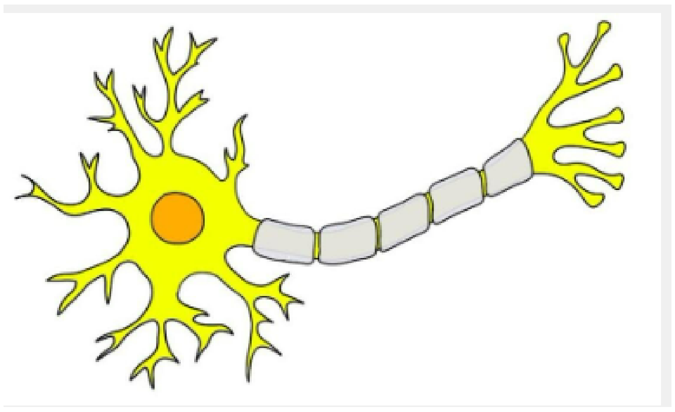

A Comparative Study of Classical Machine Learning Algorithms and Neural Networks

Name: Tshana Tumelo Konaite

June 9, 2025

Contents

List of Figures

List of abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
CNN	Convolutional Neural Network

CHAPTER 1

Introduction

1.1 Introduction to Machine Learning

Since the evolution of mankind, humans have been using various kinds of technologies to accomplish specific tasks more simply. It is safe to say AI has been the term of the last 5 years, and it has emerged as a powerful tool in many fields, including image and speech recognition, natural language processing, and even medicine. Machine learning and deep learning are two of the most revolutionary technologies in the field of artificial intelligence. They have become increasingly popular in recent years due to their ability to make predictions, analyze large datasets, and provide insights that were previously impossible to obtain. **Machine Learning** is an application of Artificial Intelligence (AI) that allows systems to learn and improve from experience without being explicitly programmed. Generally, machine Learning based analysis is dynamic and improves its learning algorithms as more data is introduced.

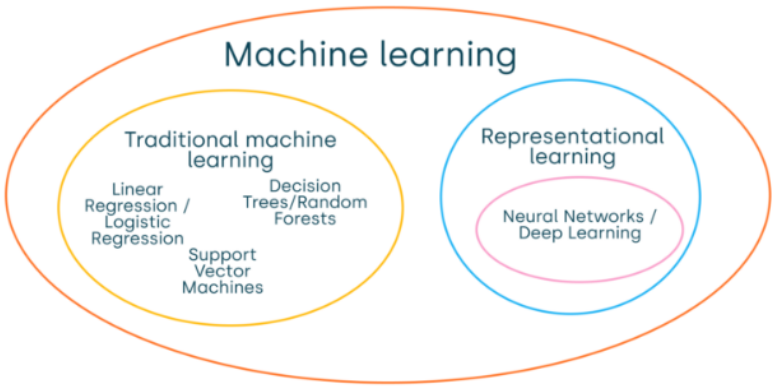
Machine learning
Traditional machine learning
Linear Regression / Logistic Regression
Decision Trees/Random Forests
Support Vector Machines
Representational learning
Neural Networks / Deep Learning

Figure 1.1: Machine Learning Methods.

Figure (1.2) is a summary of Machine Learning and its subfields. They can be classified into Traditional ML models like Linear regression models as well as Deep learning models like the Artificial Neural Network.

1.2 Supervised and Unsupervised Learning

In general, the efficacy of the machine learning model depends on the nature, characteristics, and pattern of data as well as the performance of the learning algorithm. For these reasons, a diverse array of ML algorithms (such as supervised, unsupervised, semi - supervised and reinforcement learning) has been developed to cover the wide variety of data across different ML problems.

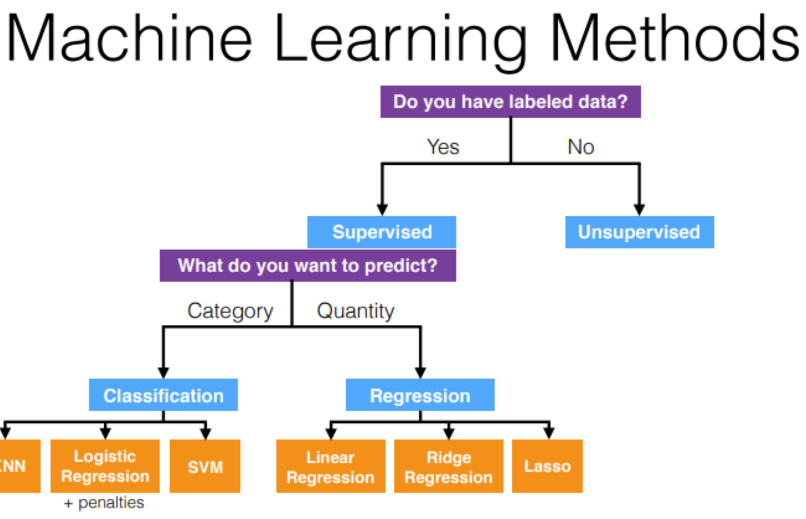

Figure 1.2: Supervised vs Unsupervised

1.2.1 Supervised Learning

In supervised learning, the algorithm is presented with data features $x_i \in X$ and labels $y_i \in Y$ and it must approximate a function $f : X \to Y$ such that $f(x_i) = y_i \; \forall i$. In essence, the goal is to *fit* f through the dataset so that it can be applied to predict labels y for new unseen data X_{new}. The process in which the algorithm approximates a function f is called *training*, and the process in which an algorithm generates predictions on unseen, unlabeled data is called *testing*.

In a nutshell, supervised models require a target variable that we are trying to train the model to predict.

Training, Testing, and Validation

Given a dataset of features, X, and an associated label, y, a machine learning algorithm f, can be trained to make predictions on some new unseen data X_{new}. A common approach is to split the dataset X into X_{train} and X_{test} . Now the subset X_{train} is used to train the model, and X_{test} is used to evaluate the performance of the model. Given that the dataset is large enough, we may push the split of training and testing data to 50/50. If the dataset is limited, a 60/40 or even a 80/20. We split the data to avoid *overfitting* during the training.

Overfitting is when an algorithm has a significantl superior predictive performance on the data on which it was trained than on some new, unseen data. This is usually caused by using a model that is too powerful or flexible for the amount of data or training the model on a small dataset.

Another common approach to train and evaluation is to use validation, or cross-validation. These methods address some challenges that arise when using a single test set, particularly on a small dataset. **Validation** is the process of splitting the dataset into three subsets: **training**, **validation**, and **test**.

- The **training set** is used to fit the model.
- The **validation set** is used to tune *hyperparameters* and monitor the model's performance during training, helping to prevent overfitting.
- The **test set** is used *only once*, after model selection and tuning, to evaluate the final model's performance on previously unseen data.

K-fold cross-validation is a technique used to evaluate the performance of a machine learning model by dividing the dataset into k equal-sized subsets (folds). Choosing a a large k value helps avoid overfitting but this may increase the computational expenses during training. Once the value for k has been selected, k model training and evaluation steps are performed iteratively.

1.2.2 Unsupervised Learning

In unsupervised learning, the algorithm is presented with only data points $x_i \in X$ and the goal is to uncover hidden patterns about the data X. Here, we typically try to approximate a different kind of function depending on the specific task. For instance, the function could be a mapping $f : X \to C$, where C is a set of cluster identifiers, but the true labels or class definitions are unknown. This type of learning is useful for the below tasks:

1. Clustering
2. Density Estimation
3. Dimensionality reduction

The learning algorithms allow one to perform more complex processing tasks compared to supervised learning. This type of learning tends to be computationally expensinve on large datasets since there are no labels present.

1.3 Classical Machine Learning Algorithms

In this section, we will introduce some of the common Machine learning algorithms used for classification and regression. The regression models we will be examining are Multiple Linear Regression, Support Vector Regression, and Random Forest Regression.

Supervised learning is classified into two categories of algorithms. Classification: A classification problem occurs when the output variable is a category, such as 'Red' or 'Blue' or 'Disease' or 'No disease'. Regression: A regression problem is when the output variable is a real value, such as “Dollars” or “Weight”.

1.3.1 Classification

Logistic Regression

Random Forest

Support Vector Machines (SVM)

Support Vector Machines (SVM) are powerful and versatile models used for classification and regression. They work by finding the best boundary (called a hyperplane) that separates data points of different classes or predicts continuous values. In $p-$dimensional space, a hyperplane is a $p-1$-dimensional affine subspace. In a $2D$, a hyperplane is a flat $1D$ subspace or a straight line.

K-Means Clustering

Random Forest Regression

Random forest is a decision tree-based method for classification or regression problems. These trees are often referred to as CART-Classification and Regression Trees. This type of learning is called *ensemble learning.*

A **decision-tree** algorithm can be thought of as a set of sequential nodes

Polynomial Regression

Support Vector Regression

1.4 Introduction to Deep Learning

Deep learning draws inspiration from one of nature's most powerful computational organs, the brain. We start by studying the structure of a neuron to understand how we can try to create an artificial neuron. The upper image in Figure (1.2) is the structure of a neuron.

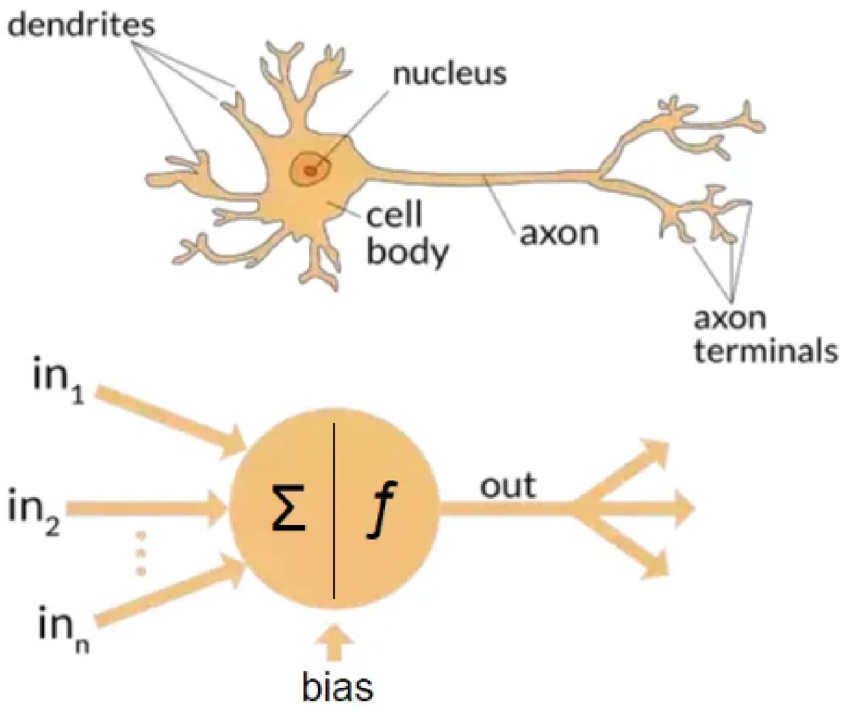

Figure 1.3: Machine Learning Methods.

The neuron has a body, which is the main part. Attached to the top of the body are dendrites and an axon at the bottom, with some terminals at the end. Alone, a neuron is not capable of much, but if you have a network of them, they become very powerful.

Neurons connect through dendrites that act as receivers of signals to neurons and axons that act as transmitters of the signal to neurons. The dendrites are connected to the body via synapses.

CHAPTER 2

The Artificial Neural Network

2.1 Fundamental concepts of the ANN

We now move into the architecture of the artificial neuron. The lower image in Figure (1.2) is an artificial neuron. It is made up of the neuron, which is also referred to as a node. This node serves as our body. We also have some inputs, which serve as our dendrites, as well as an output signal, which serves as an axon. The inputs are often called the input layer. In analogy to the human brain, the input layer can be thought of as your senses, but in the context of artificial neurons, they are the independent variables. Note that these independent variables can be considered as a single row in your database with a corresponding single output in the case of regression and classification. We often either standardize the inputs to ensure that they have a mean of zero and a variance of one, or normalize them to get the range of the values to be between zero and one. The output can either be a continuous value like the price of an item, a binary value like yes/no, or a categorical variable. The synapses are replaced by weights in the artificial neuron. The weights play a very big role in how the AN learns, as we will see later on in the coming sections. Inside the neuron, some computations occur, usually starting with the weighted sum

$$\sum_{i=1}^{m} w_i x_i \tag{2.1}$$

and then applying an activation function $\phi(x)$, resulting in

$$\phi\left(\sum_{i=1}^{m} w_i x_i\right). \tag{2.2}$$

The node then passes on the signal as an output to the next node in line.

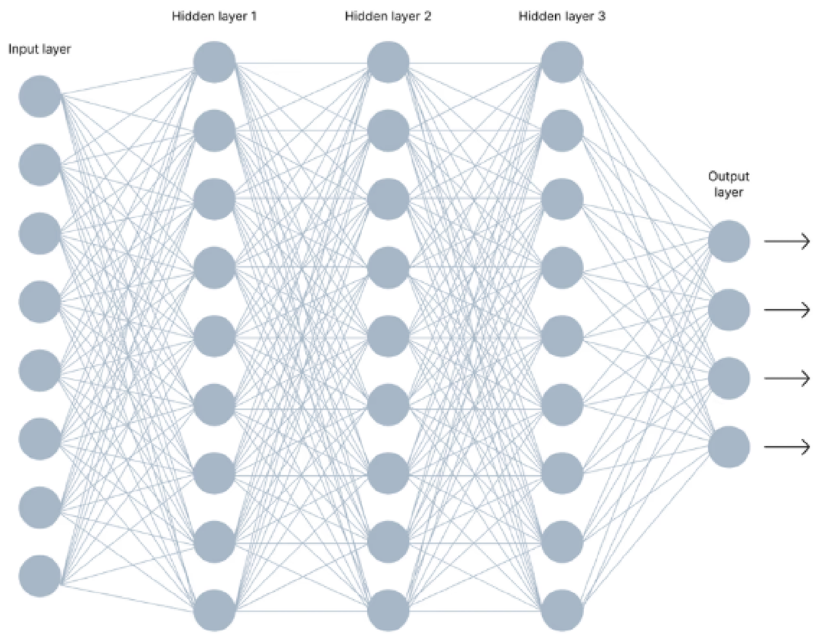

Figure 2.1: Neural Network.

In Figure 2.1, we see a neural network that is composed of interconnected neurons. Each of them is characterized by its weight, bias, and activation function. The network comprises three main layers: the input layer, the hidden layer as well and the Output layer. The main function of the input layer is to take in the raw data from our observations, and the number of neurons in this layer is determined by the dimensions of the input data. There are no computations in this layer whatsoever. The network can have zero or more hidden layers, where all the computations are performed, and then the results are transferred to the output layer. The output layer delivers the final value as a result. The number of neurons in the output layer depends on the number of possible outputs the network is designed to produce.

2.2 Activation functions in Deep learning

Activation functions are the functions that you use in a node to get its output. An Activation Function decides whether a neuron should be activated or not. This means that it will decide whether the input of the neuron to the network is important or not in the prediction process using simpler mathematical operations. The primary role of the Activation Function is to transform the summed weighted input from the node into an output value to be fed to the next hidden layer or as output.

There are mainly four activation functions (**Threshold, sigmoid, tanh, and ReLU**) used in neural networks in deep learning. These are also called squashing functions as their domain is the set of all real numbers, and the output is often within a certain range. We will also see various advantages and disadvantages of different activation functions. These activation functions help in achieving non-linearity in deep learning models. Without this non-linearity feature, a neural network would behave like a linear regression model, no matter how many layers it has. Non-linearity means that the relationship between input and output is not a straight line. In simple terms, the output does not change proportionally with the input.

The first one we look at is the **Threshold Function**.

Threshold Activation Function

The Threshold function is a basic function that outputs either 0 or 1 depending on its given inputs.

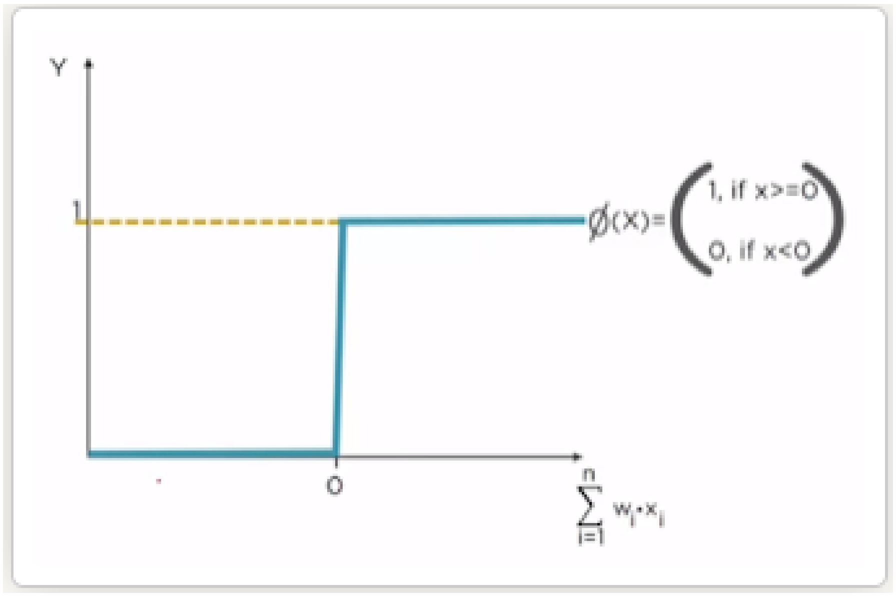

Figure 2.2: Threshold function

With this activation function, we notice that it is a very rigid function where the outputs suddenly change from 0 to 1 when the threshold is passed. This function can be extended to a smoother function to be more applicable to most real-world problems. This extension is known as the **Sigmoid Function**.

Sigmoid Activation Function

The graph of the sigmoid function is an S-shaped, smooth curve as shown in the figure below.

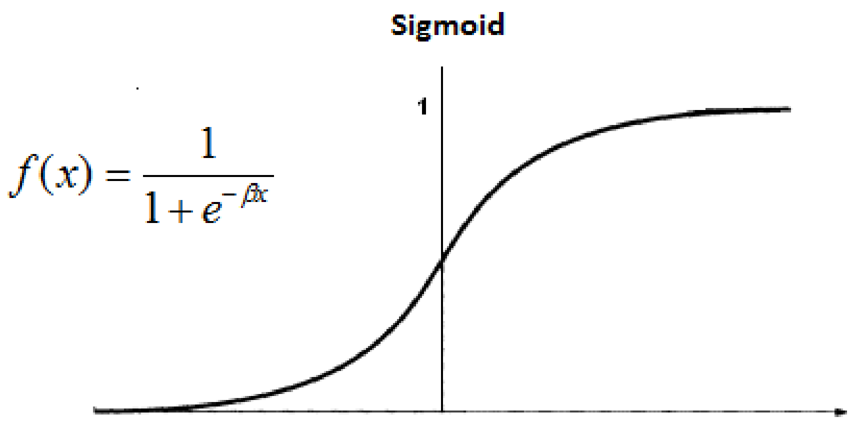

Figure 2.3: Threshold function

Unlike the Threshold function, this one does not have the kinks in its curve. This function is very useful in the output layer, especially when trying to predict probabilities. Instead of binary values, the outputs here are probability values. So the sigmoid function curve is smooth, continuous, and differentiable everywhere in its domain.

Hyperbolic Tangent(tanh) Activation Function

It is similar to the Sigmoid Activation Function, the only difference is that it outputs values in the range of -1 to 1.

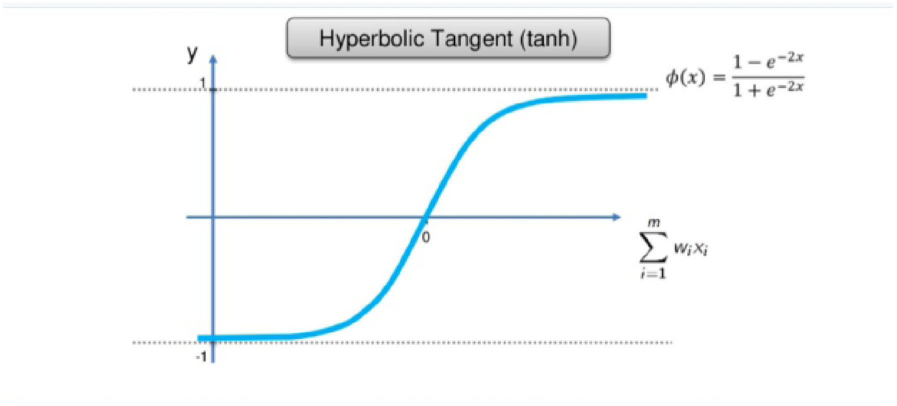

Figure 2.4: Tanh function

Rectified Linear Unit Activation Function

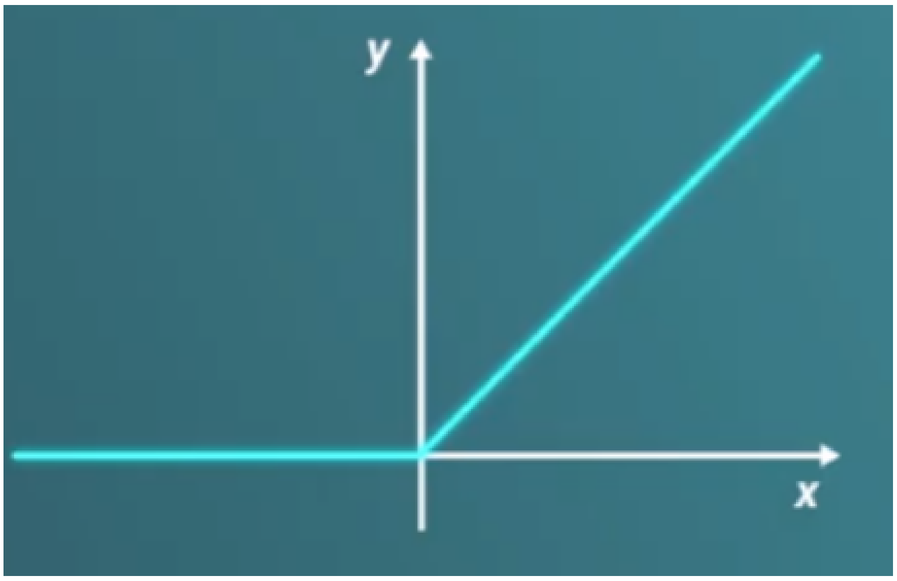

Figure 2.5: ReLu function

2.3 How do ANNs work

In the feedforward neural network, we have two main phases: the feedforward and backpropagation.

2.4 Feed-forward

In the **feedforward phase**, the input is ingested into the network through the input nodes and then propagates through the network. These nodes have associated weights. The weights w associated with each node are values in the $\mathbb{R}$ space and are important for helping the model learn the importance of a particular node in predicting an output. The bigger the weight w is, the stronger its influence on the output.

At the beginning of the training, random small weights are assigned to the nodes.

At each hidden layer, the activation function is applied, which introduces non-linearity into the model. This process continues to the output layer, and a prediction $\hat{y}$ is made. This prediction $\hat{y}$ is then compared to the actual value y, and the cost function C is then calculated. This is usually given by the formula

$$C = \frac{1}{2}(\hat{y} - y)^2 \tag{2.3}$$

Our main goal is to reduce the value of C. Since the only variables that we can control in the training of the model are the weight w_i, we have to adjust them such that the combination of weights returns the minimum cost C. This leads us to the backpropagation phase.

2.5 Backpropagation

In the Feedforward phase, an error C was calculated using 2.3. This error is then propagated back through the network to the input layer. This error is then used to compute the gradient of the cost function C with respect to each weight w_i.

Backpropagation is a method used to train neural networks. Its goal is to improve the accuracy of the model's predictions.

2.6 The Gradient Descent Methods

The gradient descent method is an optimization algorithm that is used to find the weights that minimize the cost function C. As the name suggests, there are two components required in this algorithm:

- Gradient: The first-order derivative or slope of a curve
- Descent: Movement to a lower point.

The gradient descent method is a first-order optimization algorithm because it uses only the first derivative (i.e. the gradient) of the objective function to guide the search for a minimum. This cost function can be considered as a way of punishing our model for the wrong output with respect to the current weight parameters. Thus by minimizing the cost function C, we can find the best the optimal weights that yield the best model performance. This can be considered as an optimization problem of the form

$$\begin{aligned} &\min_{x} f(x) \\ &\text{s.t } x \in \mathbb{R}^n \end{aligned} \tag{2.4}$$

In this case, a point $x_\star \in \mathbb{R}^n$ solves the optimization problem only $\nabla f(x_\star) = 0$.

The most common cost functions are

- Mean error
- Mean squared error
- Mean Absolute error

Mean absolute error

The mean Absolute Error defined as the average of the absolute differences between predicted and actual values is given by the formula:

$$MAE - \frac{1}{n}\sum_{i=1}^{n} |(y_i - \hat{y}_i)| \tag{2.5}$$

Mean squared error

The Mean Squared Error, defined as the average of the squared differences between predicted and actual values, is given by the formula:

$$MSE - \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.6}$$

Let n be the number of parameters. We begin by exploring the simple case where $n = 1$. We can plot an arbitrary cost function $C(w_n)$, as a function of w. In this base case, it is easy to identify the optimal point. However, calculating C for every possible parameter value would be computationally expensive. Alternatively, we start with an initial guess and iteratively refine it. Since we are guessing, we do not know what the cost function C looks like for the simple case $n = 1$. Fortunately, we constructed our cost function so that it was continuous and differentiable. Thus we also know the instantaneous slope cost function at our current position. For n-dimensional w, we can calculate the gradient which represents the instantaneous slope of the function with respect to each parameter w_i. We can find the minimum point of C by iteratively taking steps in the opposite direction of the instantaneous slope

Next, we evaluate the cost function and its gradient at our new parameter value and then iteratively repeat this process of moving in the opposite direction of the gradient and then reevaluating the cost function and its gradient. Generally, the process of gradient descent can be represented as

$$w_i := w_i + \Delta w_i \tag{2.7}$$

where

$$\Delta w_i = -\alpha \frac{\partial C(w)}{\partial w_i} \tag{2.8}$$

Δw_i is the amount in which we move along the gradient, which is usually called the **step size** and it is controlled by a learning rate α. The learning rate is a tuning parameter that determines the step size at each iteration of gradient descent. It determines the speed at which we move down the slope. The size of the steps is important to ensure that there is a balance between time and accuracy. If α is too large, we could miss the minimum point completely. This can lead to inaccurate results. If α is too small, the optimization process could take a lot of time and thus be computationally expensive.

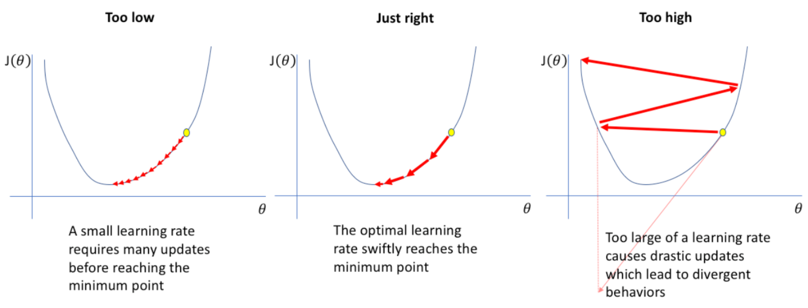
Too low
Just right
Too high
$J(\theta)$
$J(\theta)$
$J(\theta)$
θ
θ
θ
A small learning rate requires many updates before reaching the minimum point
The optimal learning rate swiftly reaches the minimum point
Too large of a learning rate causes drastic updates which lead to divergent behaviors

Figure 2.6: Learning rate

To navigate the cost function, we require the navigation direction and the **step size** Δw for the navigation. The gradient is used to determine the direction. We aim to find the negative gradient because it indicates that the slope is decreasing. A decreasing slope implies that we are moving downward. In the case of a multivariate cost function, we can move along the cost function in any number of directions. The gradient of the cost function corresponds with the direction of greatest descent.

In the gradient descent method, the slope of the current parameter reaches zero as we reach a minimum point and the parameter value will stop updating. At first glance, this might seem like a good result as it would mean that our function is convergent and implies that we can stop the iterative process. It poses a problem if the gradient descent method finds a local minimum. There are several techniques that help avoid the problem of reaching a local minimum which may not be a global minimum, like using a momentum term or developing cost functions which are proven to have a convex space, ensuring that we only have one optimum, the global minimum.

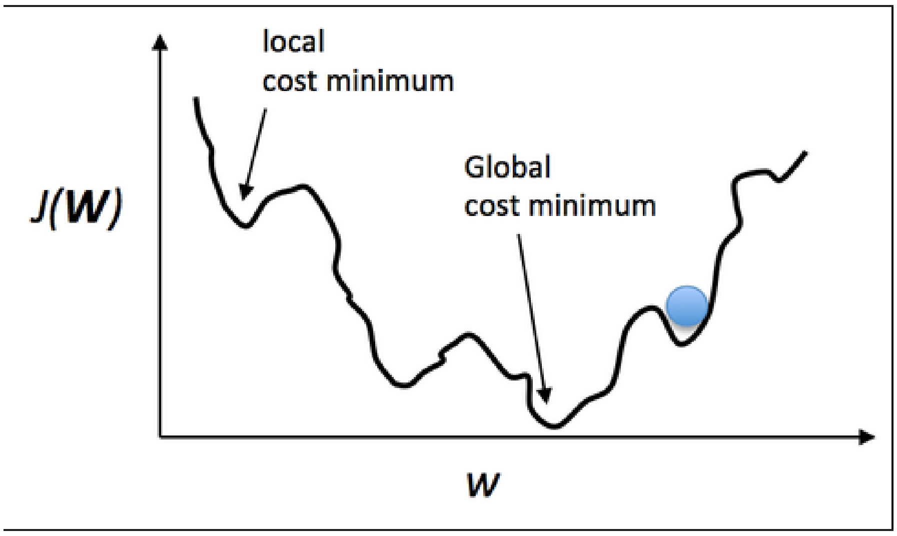

Figure 2.7: Local minimum

When training a neural network, we often use a large dataset to adjust the weights iteratively. The observations individually go through the feedforward phase, and then the total cost C is computed, and then the weights are updated. . Multiple epochs are run until the optimal C is computed.

This process is known as **batch gradient descent** training since the cost function is computed once all the observations have been fed forward into the network. For large datasets, this can be computationally expensive. An alternative to this method is **mini batch gradient descent**. Our training data is split into mini batches which can be processed separately. When each mini batch is processed, we compute the cost function relative to the data from the current mini batch and update our parameters accordingly. Then we continue to do this iterating over all of the mini batches until we have went through the whole dataset. This is often referred to as an **epoch**

The mini batch gradient descent allows us to improve our model parameters at each mini-batch iteration. Because now we are updating our parameters on a narrower view of the data, we introduce a degree of variance into our optimization process. As much as we will generally follow the direction towards the global minimum, we are no longer guaranteed that each step will bring us closer to the optimal parameter value.

It is also possible to set a mini batch size equal to one where we perform an update on a single training observation. This is known as **stochastic gradient descent**.

CHAPTER 3

Results

CHAPTER 4

Codes

```python
##Importing the libraries
import numpy as np
import pandas as pd
import tensorflow as tf

## Part 1 - Data Preprocessing
#Importing the dataset
dataset = pd.read_csv('Churn_Modelling.csv')
X = dataset.iloc[:, 3:-1].values
y = dataset.iloc[:, -1].values

##Encoding categorical data
#Label Encoding the "Gender" column

from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
X[:,2] = le.fit_transform(X[:,2])
```

```

# One Hot Encoding the "Geography" column

from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [1])],
    remainder='passthrough')
X = np.array(ct.fit_transform(X))

##Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
    0.2, random_state = 1)

## Feature Scaling
# Feature Scaling is absolutely imperative whenever you are building a
    Neural Network

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

## Part 2 - Building the ANN
#Initializing the ANN
ann = tf.keras.models.Sequential()

#Adding the input layer and the first hidden layer

ann.add(tf.keras.layers.Dense(units=6, activation='relu'))

# Adding the second hidden layer
ann.add(tf.keras.layers.Dense(units=6, activation='relu'))
```

```python

# Adding the output layer
ann.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
```

Bibliography