CELL IMAGE CLASSIFICATION USING CONVOLUTIONAL NEURAL NETWORKS

A THESIS SUBMITTED TO
THE FACULTY OF ACHITECTURE AND ENGINEERING
OF
EPOKA UNIVERSITY

BY

KEVIN ÇUEDARI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR BACHELOR DEGREE
IN COMPUTER ENGINEERING

JULY, 2021

ABSTRACT

CELL IMAGE CLASSIFICATION USING CONVOLUTIONAL NEURAL NETWORKS

Kevin Çuedari
B.Sc., Department of Computer Engineering
Supervisor: Prof. Dr. ……….

Medical image processing is becoming increasingly prevalent as technological advancement has facilitated medical diagnosis of organs and tissues. Additionally, as Machine Learning and Deep Learning are increasingly becoming omnipresent in human society, it has found numerous applications in the medical field. This thesis seek to give a general overview of important ML concepts, explore the components of Convolutional Neural Networks, and most importantly construct and deploy a CNN to conduct cell image classification, as well as compare and interpret the impact preprocessing, more specifically unsharp masking, has on classification. The neural network pertaining to this thesis is based on the LeNet architecture, with certain modifications to improve compatibility with our dataset. The dataset of this study consists of 128x128 cell images divided into two classes, healthy and unhealthy. The training set consists of 20,102 images, of which 12,520 depict unhealthy cells and the remaining 7,582 showcase healthy cells. After the model is trained using this dataset, it is tested on two testing sets of entirely new examples, one containing 11339 and the other 11360 images.

Keywords: Cell images, LeNet, Preprocessing, Classification, Convolutional Neural Network

Dedicated to my family for their unrelenting support

ACKNOWLEDGEMENTS

I wish to extend my most sincere gratitude to Dr. Arban Uka for his helpful guidance, valuable advices, and continuous interest in the success of my thesis throughout its development. His experience, commitment, and relentless dedication to this field of study has personally served me as a great source of inspiration with regards to Deep Learning.

TABLE OF CONTENTS

ABSTRACT iii
ACKNOWLEDGEMENTS (optional) v
TABLE OF CONTENTS vi
LIST OF TABLES viii
LIST OF FIGURES ix
LIST OF ABBREVIATIONS x
CHAPTER 1 0
INTRODUCTION 0
CHAPTER 1 2
Literature Review 2
2.1. Types of Learning 2
2.1.1. Supervised Learning 2
2.1.2. Unsupervised Learning 12
2.1.3. Semi-Supervised Learning 16
2.2. Artificial Neural Networks 17
2.2.1. Activation Function 19
2.2.2. Loss Function 22
2.2.3. Gradient Descent 23
2.2.4. Regularization 25
2.3. Convolutional Neural Network 27
2.3.1. Convolutional Layer 28
2.3.2. Pooling Layer 29
2.3.3 Fully-Connected Layer 30
2.3.4 Normalization 31
2.3.5. Dropout 32

CHAPTER 3 33
Network Architecture & Methodology 33
3.1. Architecture 33
3.2. Dataset 34
3.3. Preprocessing 35
CHAPTER 4 36
Results... 36
4.1. Experiment 1 – No Preprocessing Applied 38
4.2. Experiment 2 – Unsharp Masking with Radius = 5, Amount = 2 40
4.3. Experiment 3 – Unsharp Masking with Radius = 1, Amount = 1 40
4.4. Experiment 4 – Unsharp Masking with Radius = 10, Amount = 1 41
4.5. Experiment 5 – Unsharp Masking with Radius = 1, Amount = 0.5 42
CHAPTER 5 43
Conclusion 43
References 44
Appendix 46

LIST OF TABLES

Table 1: List Of Activation Functions

Table 2**:** Training results with epoch 20, batchsize 64

LIST OF FIGURES

Figure 1: SVM Visualization of Binary Classification.........4
Figure 2: Kernel Trick.........5
Figure 3: KNN Visualization.........6
Figure 4: Simple Linear Regression Visualization.........8
Figure 5: Multiple Linear Regression Visualization.........9
Figure 6: Sigmoid Function......... 10
Figure 7: K-Means Clustering......... 13
Figure 8: Fuzzy Clustering......... 14
Figure 9: Hierarchical Clustering......... 15
Figure 10: Feed Forward Neural Network......... 18
Figure 11: Structure of a node with activation function depicted......... 19
Figure 12: Graphical representation of the ReLU function......... 21
Figure 13: Batch Normalization......... 24
Figure 14: Mini-Batch Gradient Descent......... 24
Figure 15: Stochastic Gradient Descent......... 25
Figure 16: Lasso (left) and Ridge Regression (right). The red contours represent the error function, while the blue areas are the constraint regions......... 26
Figure 17: Typical CNN Architecture......... 28
Figure 18: The image on the left showcases the network before dropout was applied, and the image on the right represents the result of dropout......... 32
Figure 19: Two images from our dataset representing a healthy (right) and an unhealthy (left) cell......... 34
Figure 20: Graphical interpretation of the results......... 37
Figure 21: Cell images of testSet1 and their predicted labels......... 38
Figure 22: Cell images of testSet2 and their predicted labels......... 39
Figure 23: Sharpened image (left) and original image (right). Amount = 5, Radius = 2. 40
Figure 24: Sharpened image (left) and Unsharpened image (right). Amount = 1, Radius = 1. 41
Figure 25: Sharpened image (left) and Unsharpened image (right). Amount = 1, Radius = 10. 42
Figure 26: Sharpened image (left) and Unsharpened image (right). Amount = 0.5, Radius = 1. 42

LIST OF ABBREVIATIONS

CNN	Convolutional Neural Network
ML	Machine Learning
DL	Deep Learning
SVM	Support Vector Machines
KNN	K-Nearest Neighbor

CHAPTER 1

INTRODUCTION

Medical imaging is concerned with imaging the interior of a body for clinical analysis, medical intervention, and diagnoses. Medical image analysis enables doctors and healthcare workers to provide proper treatment to patients suffering from life-threatening diseases. The deployment of Deep Learning (DL) techniques expedites the process of medical image analysis while simultaneously having the potential to produce far more accurate results, free of human error. Furthermore, Deep Learning has become increasingly omnipresent in computer vision applications such as medical image analysis, cancer detection, image segmentation etc. Additionally, it has proven to be successful in image registration, (Wu, Minjeong, Munsell, & Wang, 2015) computer-aided disease diagnosis or prognosis, (Heung-Il, Lee, & Shen, 2013) (Suk, Wee, Lee, & Shen, 2016) tissue classification, (Ronneberger, Fischer, & Brox, 2015) and cell counting, (Uka, Tare, Polisi, & Panci, 2020) and quantification of cellular shape and size on micro patterned surfaces. (Uka, Polisi, Halili, Dollinger, & Vrana, Analysis of cell behavior on micropatterned surfaces by image processing algorithms, 2017)

Cell image interpretation is of critical importance to science and medicine. Microscopy has enabled scientists, researches, and medical experts to observe the characteristics and behaviors of cells, thus allowing us to study and understand their structure and operation. Cells not only exhibit complex phenotypes and phenotypic heterogeneity, but they also exhibit different responses to different stimuli, which renders the study and comprehension of their biological modus operandi as a convoluted and difficult process. Computational methods have been developed and deployed to facilitate this process, and have shown remarkable results. Cell classification, i.e. the process of determining the category of a cell or a collection of cells, is of critical importance to medical diagnosis, personalized treatment, drug discovery, genetic screening, and disease prevention.

The focus of this paper is to use a Deep Learning binary classification model to classify cell images from a dataset of grayscale images into two different classes. The model must take an image as input, detect all cells in the image, and then accurately classify each cell into its respective class. We begin by first conducting a literature review, in which we will provide a brief, but comprehensive overview of the different learning types in ML, introductions to various ML models, with a particular focus on image classification – namely, K-Nearest Neighbor, Support Vector Machine, and Convolutional Neural Network - including each classifiers' advantages and disadvantages, as well as a review and analysis of previous studies.

CHAPTER 1

Literature Review

2.1. Types of Learning

Machine learning is a broad field of study that overlaps with, inherits ideas from, and exchanges ideas with numerous related fields such as Statistics, Data Mining, and Artificial Intelligence – just to name a few. The basis of machine learning is the construction of computer programs that automatically "learn" and improve with experience and data usage. This combination has given birth to a numerous types of learning, each with their own specific applications. For the purpose of this thesis, our focus is going to be narrowed down to the three most widely applied methods, which are Supervised Learning, Unsupervised Learning and Semi-Supervised Learning.

2.1.1. Supervised Learning

In Supervised Learning (SL), a model is trained using a training set consisting of numerous examples, and is then used to – hopefully correctly - predict the output or value of new examples. Each example in the training set is represented as an input-output pair, consisting of an input and an output vector. The Supervised Learning algorithm is going to analyze instances from the training set, infer a function from this analysis, then use this function to map new, previously unseen instances to the correct output. The accuracy with which a model trained on a specific training set is able to predict the correct output for new instances is denoted as generalization. If the complexity of the function inferred from the training phase matches the complexity of the underlying function intrinsic to the training dataset, then we are going to have a low generalization error. If the complexity of the inferred function is less than the complexity of the underlying function, we are going to experience underfitting, thus producing lower accuracy, meanwhile if the complexity of the inferred function is greater than the complexity of the underlying function, we are going to experience overfitting. In such a case, the model may also learn the noise in the data, leading to a bad fit. If overfitting occurs, the generalization error somewhat controlled by increasing the amount of training data. Generally an increase in the amount of training data, leads to a decrease of the generalization error, and increase in the complexity of the "learned" function leads to an initial, temporary decrease of the generalization error, which will promptly increase again. Thus, trade-off is apparent these three factors: The amount of training data, the generalization error, and the complexity of the "learned" function or model. Supervised Learning can be broadly classified into two predominant types: Regression and Classification. In regression, the model learns from labelled datasets to be able to produce continuous-valued output predictions for new examples inputted to the algorithm. In classification, the train model is able to correctly identify the class (or category) of the new examples (or observations). There are many Supervised Learning algorithms, but in this thesis our discussion is going to be limited to Support-Vector Machines (SVM), K-Nearest Neighbor, Linear Regression, Logistic Regression, and Naive Bayes.

2.1.1.1. Support Vector Machines

SVMs are supervised learning models used for classification and regression analysis. Relative to other machine learning methods, SVMs are very effective at recognizing subtle patterns in complex datasets. Classification usually involves partitioning data into two sets, a training set and a testing set. Each instance in the training set contains one class label and several observed variables. The training set is used to construct a model, which is later deployed to classify the test data based on the observable variables of the test set.

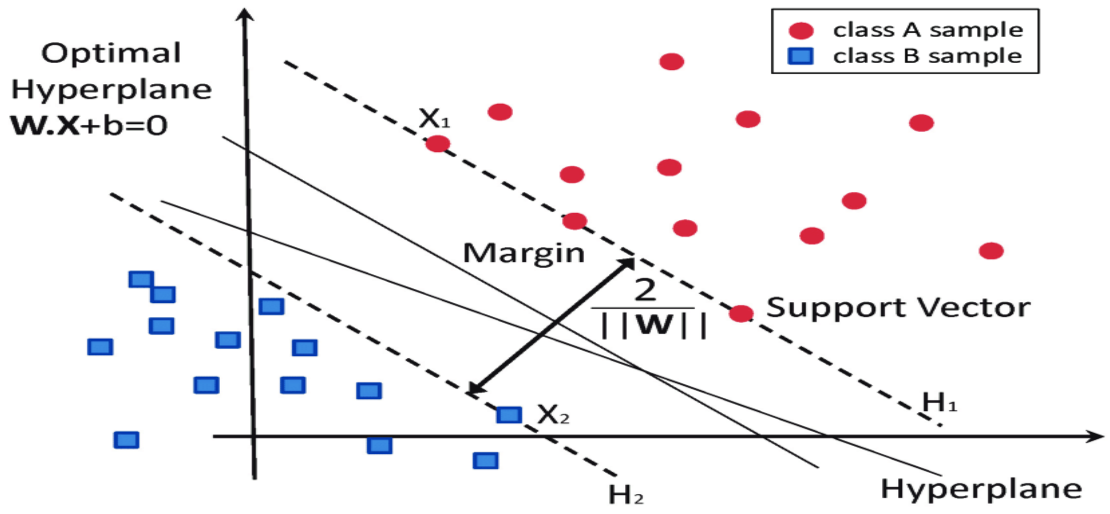

Figure 1: SVM Visualization of Binary Classification

The algorithm will output a hyperplane, which separates instances of the two classes and maximizes the margin between them. As can be observed in **Error! Reference source not found.**, the Optimal Hyperplane maximizes the functional margin between the two classes, which typically minimizes the generalization error of the classifier. (Friedman, Tibshirani, & Hastie, 2008) Samples, which fall on the margin are referred to as support vectors, thus, the Optimal Hyperplane can be defined as the mean between the two support vectors. Again, referring to figure 1, it is obvious that the samples are represented in two-dimensions, and, therefore, they can be separated by a line, for example $x_2 = a \cdot x_1 + b$. Defining the input vector $\boldsymbol{x} = (x_1,\ x_2)$, the weight vector $\boldsymbol{w^T} = (a, -1)$, and bias as $\boldsymbol{b}$ we get the general equation for the hyperplane:

$$(2.1) \qquad \boldsymbol{w^T} \cdot \boldsymbol{x} + \boldsymbol{b} = 0$$

Using equation 2.1, we define a hypothesis function denoted as h that takes as input any sample point x_i, and outputs the class of that point.

$$(2.2) \qquad h(x_i) = \begin{cases} +1 & if\ \boldsymbol{wx} + b \geq\ 0 \\ -1 & if\ \boldsymbol{w} \cdot \boldsymbol{x} + b \leq 0 \end{cases}$$

Thus, any point above the hyperplane will be classified as a member of Class +1, and any point below will be classified as a member of Class -1. It must be noted that equation 1 is not constrained to two-dimensionally representable data, but is also applicable to datasets containing data represented in N-dimensions.

To maximize the margin between the classes, we need to minimize the norm of the weight vector ||w||, which can be defined as the following minimization problem

$$(2.3) \qquad \min_{w,b} \frac{1}{2}\left||w|\right|^2 \ subject\ to\ y_i(w \cdot x + b) - 1 \geq 0, i = 1 \ldots m$$

$$(2.4) \qquad \begin{cases} x_i \cdot w + b \ \geq \ +1 \ \ if\ yi = +1 \\ x_i \cdot w + b \ \leq -1 \ \ if\ yi \ =-1 \end{cases} \ combines\ into\ y_i(w \cdot x + b) - 1 \geq 0$$

Minimizing the parameters in equation 2.3 will yield the best fit, thus the optimal solution.

Using the Kernel Trick, i.e. mapping input into higher-dimensional feature space, allow us to perform non-linear classification, which is particularly useful when the data is not linearly separable in the input space. (Schölkopf, 2000)

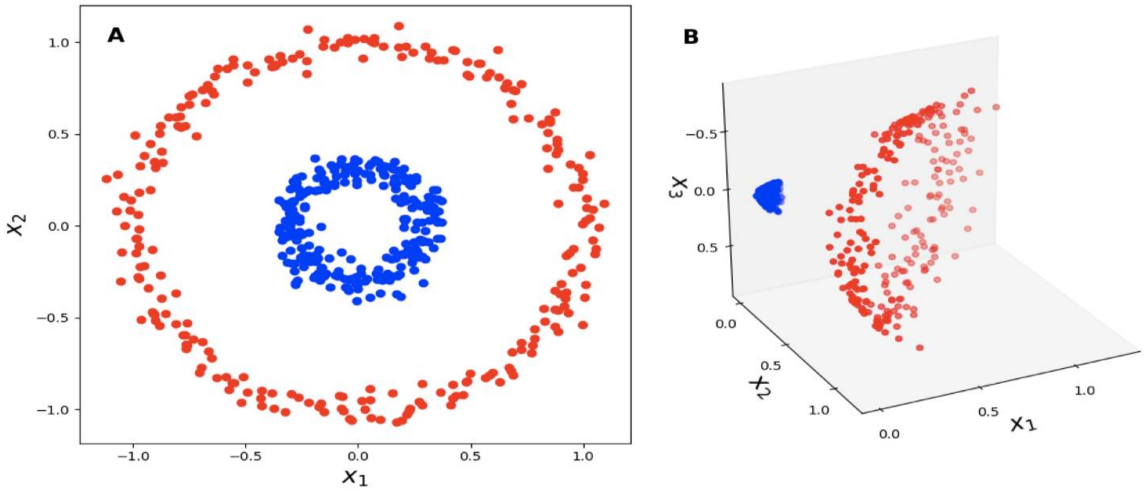

Figure 2: Kernel Trick

SVMs are particularly effective when there is a clear margin of separation between the classes and they are able to perform well in high dimensional spaces. However, SVMs are sensitive to noise and their performance degrades for large datasets.

2.1.1.2. K-Nearest Neighbor

The KNN algorithm (Altman, 1990) is a supervised algorithm primarily used in image classification, (Zhang, Li, Zong, Zhu, & Wang, 2018) but it has also found applicability in regression problems. (Song, Liang, Lu, & Zhao, 2017) The training phase consists of storing feature vectors and the labels of the training images, then an unlabeled data point is assigned the label of its k nearest neighbors in the classification phase. The principle of K-Nearest Neighbor algorithms is that data points belonging to the same class tend to be located near one another, i.e. it classifies data points on the basis of their similarity.

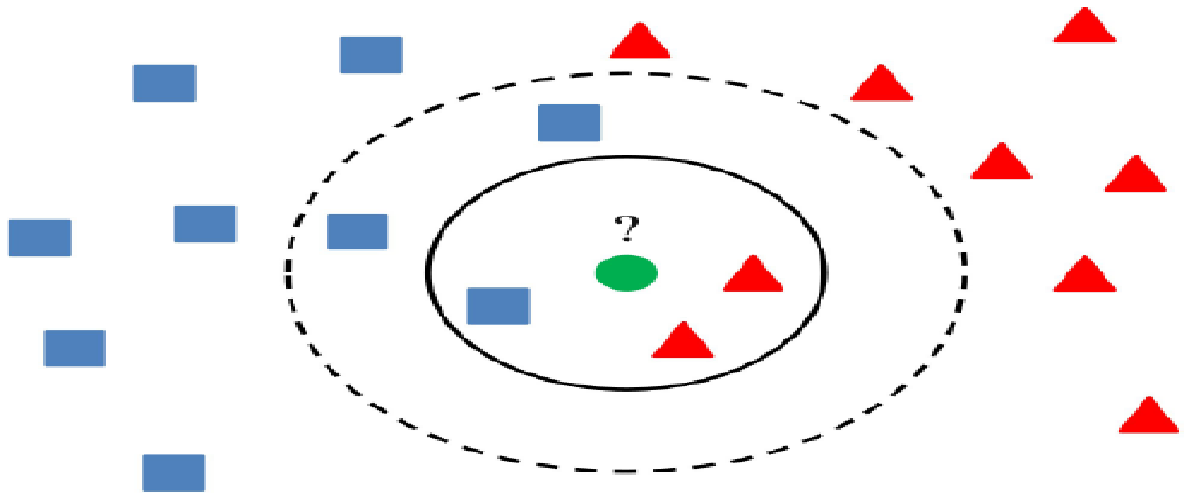

Figure 3: KNN Visualization

Additionally, the distance between the test data and each of the training data is measured to decide the final classification output of the algorithm. A distance function merely calculates the distance between elements of a set, if the result of this function is zero, then the elements are equivalent, otherwise they differ from each other. The selection of a distance metric may be influenced by the particular dataset pertaining to a certain problem. The Euclidean distance is the most commonly used distance metric with respect to KNN classifiers, however it is not always the most accurate. In medical domain datasets – including categorical, numerical, and mixed datasets – the Chi distance has been shown to be the best performer, (Hu, Huang, Ke, & Tsai, 2016) but even this may not hold true for each case. The main advantage of the KNN classifier is its simplicity and its ability to provide satisfactory results. On the other hand, selecting an appropriate k-number may be problematic, and, especially in the presence of a small subset of features, the algorithm may produce erroneous classification. (Kim, Kim, & Savarese, 2012)

2.1.1.3. Linear Regression

Linear Regression is an algorithm that produced a constant-valued output associated with a constant slope. Just like every Supervised Learning Regression method, the model "learns" from labelled datasets, and then predicts values within a continuous range, rather than classifying them into categories. Linear Regression fits linearly separable datasets virtually flawlessly, but it requires the data to be independent which is not always the case, and is particularly sensitive to outliers in the dataset. Linear Regression attempts to model the relationship between independent and dependent variables linearly, which may lead to underfitting in circumstances in which the underlying function of the dataset is of a higher complexity. Based on the input variable, we distinguish between two different types of Linear Regression, simple and multiple.

2.1.1.4. Simple Linear Regression

A Linear Regression model is referred to as a Simple Linear Regression (SLR) model when there is only one input variable. The datasets consists of two-dimensional samples with one dependent and one independent variable, from which a linear function is found to predict the dependent variable values as a function of the independent variable. Denoting the dependent variable as y and the independent variable as x, we have

$$y = \beta_0 + \beta x + \varepsilon \tag{2.5}$$

where ε represents the random error component, β0 represents the y-intercept, and β represents the regression coefficient. The intercept and the regression coefficient are the two variables the model will try to "learn" to produce as-accurate-as-possible predictions. The loss function of this model is the Mean-Squared-Error, or MSE for short, which is often used to calculate the error of the model. This procedure is done by squaring the distance of the observed values from the predicted values at each value of the independent variable and calculating the mean of each of the squared distances. Minimizing MSE will improve the accuracy of the model.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}\left[y_i - (m \cdot x_i + b)\right]^2 \tag{2.6}$$

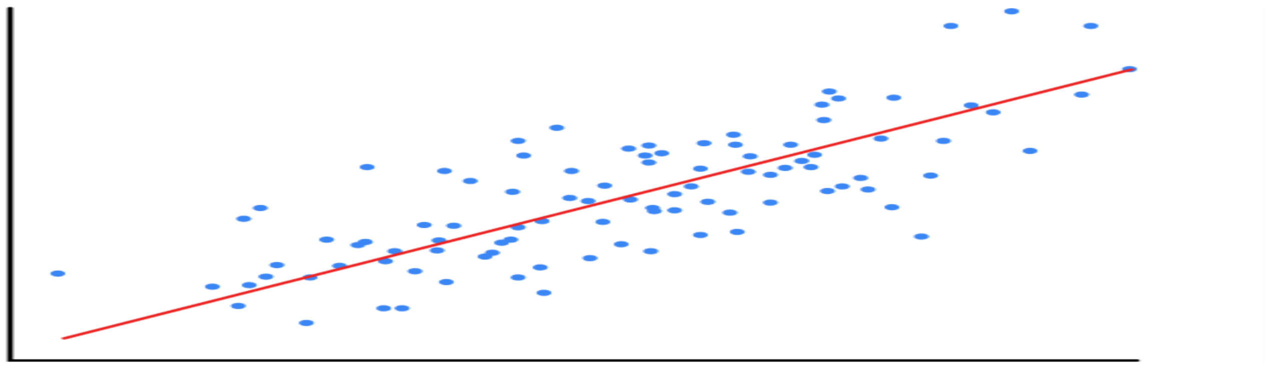

Figure 4: Simple Linear Regression Visualization

2.1.1.5. Multiple Linear Regression

A Linear Regression model is referred to as a Multiple Linear Regression (MLR) model when there is more than one input variable. The goal of the MLR is to model the relationship between the independent variables and the dependent variable. For *m* independent regressor variables we have

$$y_i = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_m x_m + \varepsilon \quad (2.7)$$

where $\boldsymbol{\varepsilon}$ represents error of the model, $\boldsymbol{\beta_0}$ represents the plane intercept and $\boldsymbol{\beta_i x_i}$ *for* $i = 1, \dots, m$ represents the product of the i^{th} regression coefficient for the i^{th} independent variable. To achieve the best accuracy, MLR will calculate, for each independent variable, the regression coefficient that yields the smallest error, the t-statistic and the associated p-value of the overall model. MSE of an MLR model is the same as the variance.

$$MSE = \sigma^2 = \frac{\sum e_i^2}{n - p - 1} \quad (2.8)$$

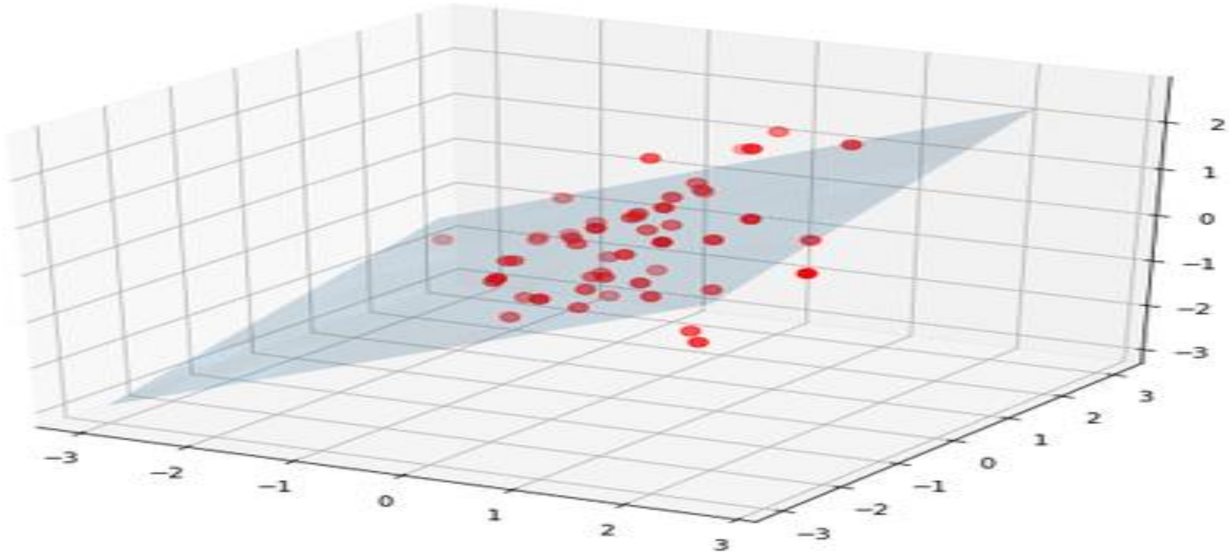

Figure 5: Multiple Linear Regression Visualization

2.1.1.6. Logistic Regression

Logistic Regression is a model deployed when the dependent variable is categorical. Strictly speaking, the logistic regression model is not a classifier, because it simply models probability of output in terms of the independent variable. It can perform binary classification if a threshold probability value is specified, with which the probabilities of the inputs are compared and, thus, each input with probability greater than the threshold is assigned to one class, and those with lesser probabilities are assigned to the other class. It is the sigmoid function $\sigma(z)$ that maps the predicted values into probabilities. An example of the Binary Classifier is the problem of categorizing cell images of a dataset into two different categories, healthy and unhealthy - represented by 0 and 1, respectively. In this case, the model will calculate a probability for each element in the dataset, the probability will be compared to the threshold, and according to the result of the comparison the image is going to be classified as either healthy or unhealthy. The logistic regression operates on the assumption of linearity between the dependent variable and the independent variables, it cannot solve non-linear problems, and it requires the independent variables to be linearly related to logit function. Also, overfitting may occur when the number of observations is less than the number of features.

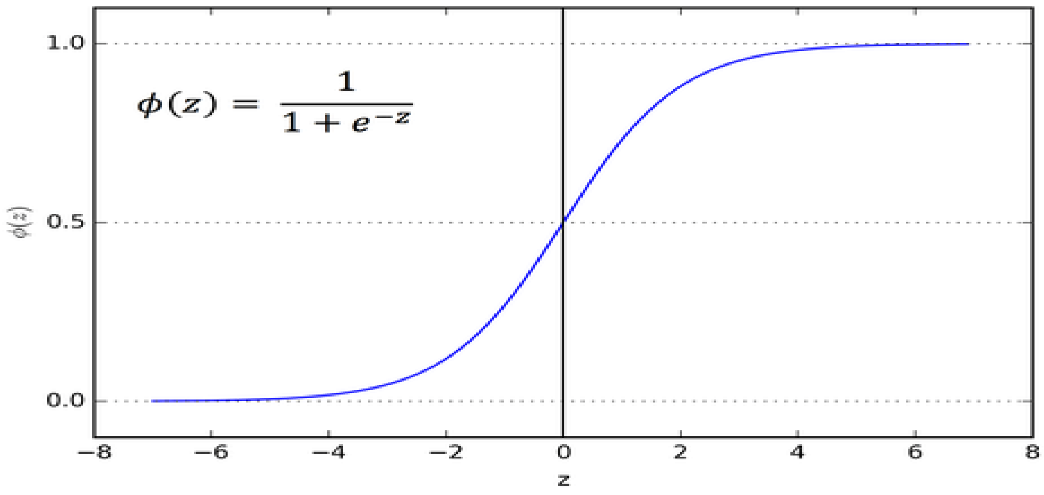

Figure 6: Sigmoid Function

The hypothesis of the model is expressed as

(2.8) $\phi(z) = \phi(\beta_0 + \beta x)$

Then applying the sigmoid function gets us

$$(2.9) \qquad h_\theta(x) = \frac{1}{1 + e^{-(\beta_0 + \beta x)}}$$

The Cross-Entropy function is defined as the loss function in logistic regression

$$\mathrm{J}(\theta) = \frac{1}{\mathrm{m}} \sum_{\mathrm{i}=1}^{\mathrm{m}} [\mathrm{Loss}(\mathrm{h}_\theta(\mathrm{x}^{(\mathrm{i})}), \mathrm{y}^{(\mathrm{i})})] \text{ s.t. } \mathrm{Loss}(\mathrm{h}_\theta(\mathrm{x}), \mathrm{y}) \begin{cases} -\log(\mathrm{h}_\theta(\mathrm{x})) & \text{if } \mathrm{y} = 1 \\ -\log(1 - \mathrm{h}_\theta(\mathrm{x})) & \text{if } \mathrm{y} = 0 \end{cases}$$

Combining equation above into one expression yields

$$(2.11) \qquad J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \cdot log(h_\theta(x)) + (1 - y^{(i)}) \cdot log(1 - h_\theta(x))$$

It is important to note that classification for *k* classes where $k > 2$ can be performed by an extension of the Binary Logistic Regression known as the Multinomial Logistic Regression.

2.1.1.7. Naïve Bayes

Naive Bayes Classifiers are a family of classifiers based on the Bayes Theorem. The fundamental assumption pertaining to this family of algorithms is that features must be independent with one another. If we define x as the vector of observable variables such that we have $x = [x_1, x_2, \dots, x_n]^T$, and C_j as a class part of $C = \{C_1, \dots, C_m\}$, and we want to calculate the probability that *x* belongs to class C_j we have

(2.12) $$P\left(C_j/x\right) = \frac{P\left(x/C_j\right) \cdot P\left(C_j\right)}{P(x)}$$

where *P(C$_j$/x)* is the posterior probability, *P(x/C$_j$)* is the class likelihood and is the conditional probability that an event belonging to C has the observed value *x*, *P(C$_j$)* is the prior probability that class C occurs, and *p(x)* is the marginal probability of *x*. Furthermore, making use of the independence assumption, i.e. features must be independent of one another, which means $P(Y, Z) = P(Y)P(Z)$, gives us

$$P(C_j/x_1, x_2, \dots, x_n) = \frac{P(x_1/C_j) \cdot P(x_2/C_j) \cdot \ldots \cdot P(x_n/C_j) \cdot P(C_j)}{P(x_1) \cdot P(x_2) \cdot \ldots \cdot P(x_n)} = \frac{P(C_j) \cdot \prod_{i=1}^{n} P(x_i/C_j)}{\prod_{i=1}^{n} P(x_i)}$$

The evidence will remain constant for a given input, thus it can be removed to define a proportional relationship between the posterior and the multiplication of the likelihood with the probability of class *C$_u$* occurring

(2.14) $$P(C_j/x_1, x_2, \dots, x_n) \propto P\left(C_j\right) \cdot \prod_{i=1}^{n} P\left(x_i/C_j\right)$$

Lastly, we need to define a classifier, such that, for all classes in the set of classes *C* we calculate the associated probabilities and output the class with the highest probability for the given input, in other words, the input is categorized as member of the class for which it has the highest probability of occurrence

(2.15) $$C_{output} = \max_{k=1,\dots,m} P(C_k) \cdot \prod_{i=1}^{n} P(x_k/C_k)$$

2.1.2. Unsupervised Learning

In contrast to Supervised Learning, in Unsupervised Learning (UL) the algorithm learns patterns from unlabeled data. The user need not supervise the model, instead, the model itself works on its own to discover patterns and data groupings. Two broad methods appear in this learning paradigm, which are Probabilistic Methods and Neural Networks. We will briefly discuss some probabilistic methods, and then introduce Neural Networks in section 2.5.

2.1.2.1. Cluster Analysis

Cluster Analysis is used to group together unlabeled and/or unclassified data that exhibit commonalities with one another. Each new data is analyzed and acted upon on the basis of the commonalities it exhibits (or lack thereof) with already-existing groups. Clustering algorithms can be generally fall into the following categories.

Exclusive clustering methods permit a data point to belong *exclusively* to one, and only one, cluster. An example of such clustering methodology is the K-means algorithm. In this algorithm, data points are assigned into K groups, where K is the number of clusters based on the distance from each group's centroid. The algorithm aims to achieve that data points inside one cluster are as as similar as possible to one another, while simultaneously maximizing the difference between clusters. The data points closest to a given centroid will be clustered under the same category. A larger K value generally corresponds to smaller groupings with more detailed and specific data inside of the groups, whereas a smaller K value corresponds to larger groupings and less specific data in the groups. K-means is an Expectation-Maximization (EM) algorithm. The E-step is assigning the data points to the closest cluster. The M-step is computing the centroid of each cluster.

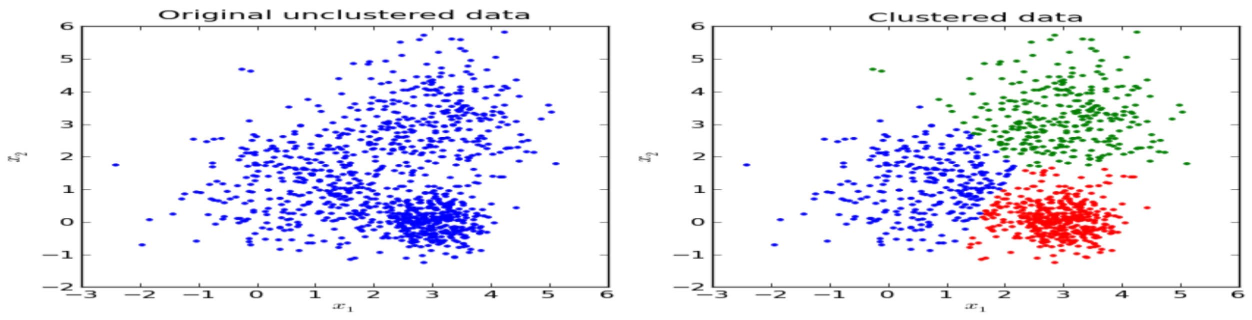

Figure 7: K-Means Clustering

In Fuzzy Clustering, data points may belong to more than one cluster. Soft or Fuzzy k-means algorithm is an example of Fuzzy Clustering.

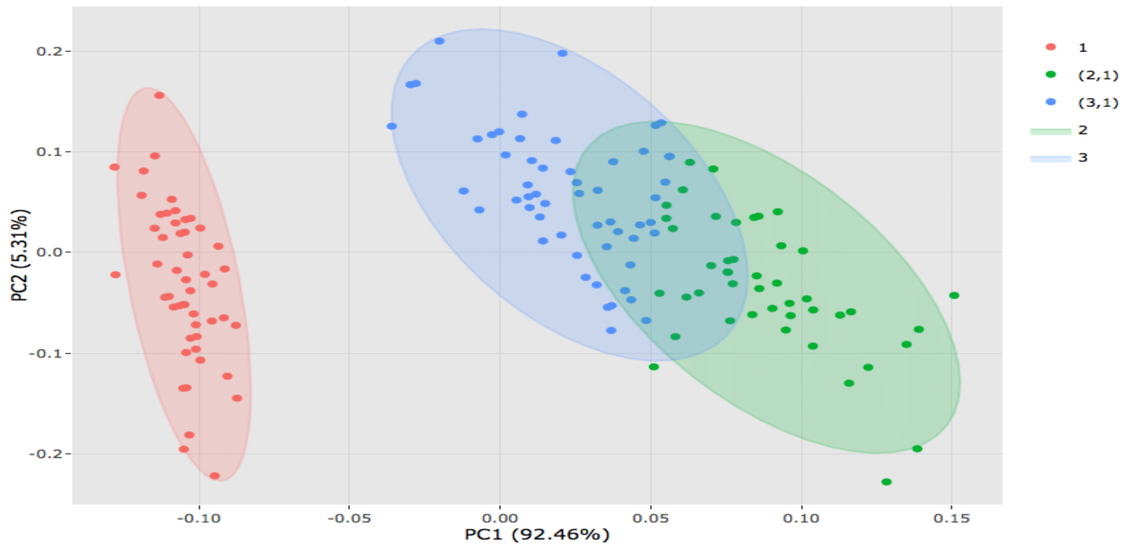

Figure 8: Fuzzy Clustering

There are two categories of Hierarchical Clustering algorithms which are known as agglomerative and divisive. The latter does not find much practical application and deployment, thus our attention will be limited to agglomerative clustering. The data points are initially isolated into separate groupings, until, through an iterative process, they are finally grouped together on the basis of their similarity. Similarity is commonly measured by six methods:

1. Average linkage: This method is defined by the mean distance between two points in each cluster

2. Ward's linkage: This method states that the distance between two clusters is defined by the increase in the sum of squared after the clusters are merged.

3. Maximum linkage: This method is defined by the maximum distance between two points in each cluster

4. Minimum linkage: This method is defined by the minimum distance between two points in each cluster

5. Centroid linkage clustering

6. Minimum energy clustering

Dendrograms, such as the one in figure 9, are typically used to represent hierarchical clustering processes.

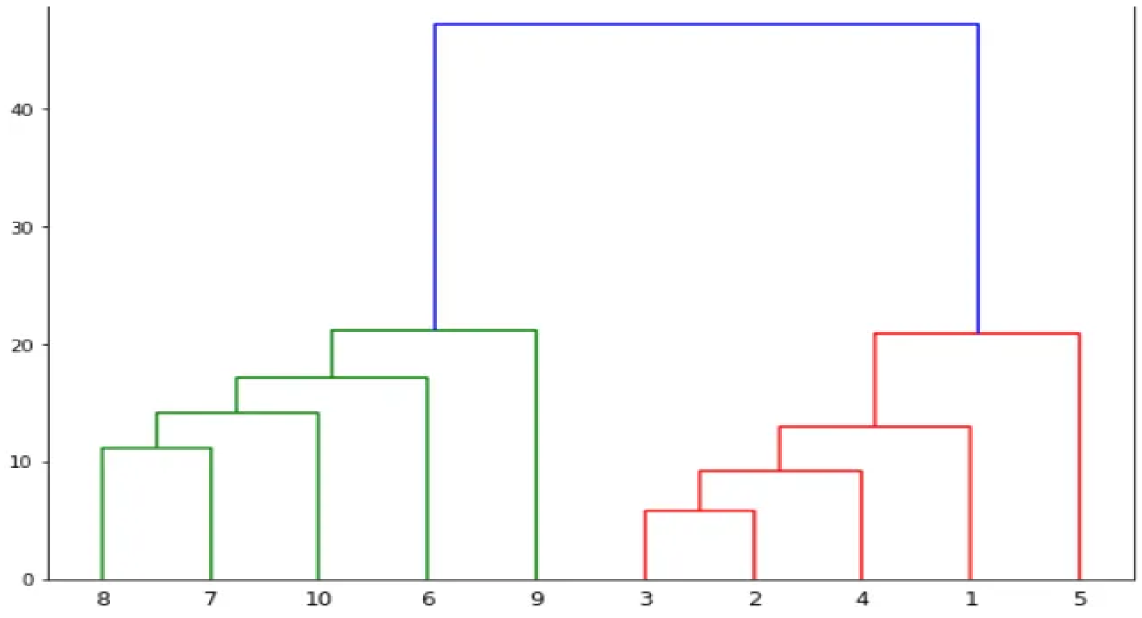

Figure 9: Hierarchical Clustering

Probabilistic models are deployed with the intent of solving density estimation problems. In this method of clustering, the data points get clustered together by the likelihood that they belong to a particular distribution. Mixture models generally consist of an unspecified number of probability distribution functions, to which a given data point can be assigned to. For example, Gaussian Mixture Models are primarily used to determine the probability distribution that a specific data points follows. To do so, the model must know, or have a way of finding out, the values of the distribution's parameters – namely, the variance and the mean. With these two parameters known, the model can directly determine the distribution which the data point follows. Usually, there is no prior knowledge of the mean and variance, thus an assumption is made that there must exist some hidden variable that can be used for clustering.

$$(2.16) \qquad p(\theta) = \sum_{i=1}^{K} p(\theta/z_i)\phi_i$$

Equation 2.16 depicts a Gaussian Mixture Model with K components, where *z* is a categorical hidden variable indicating the component *i*, ϕ_i is the prior probability of component *i*, and $p(\theta/z_i)$ is the likelihood term of the *i the* component. The likelihood of component i is expressed as the parameterized Gaussian distribution with mean μ_i and covariance matrix Σ_i.

$$(2.17) \qquad p(\theta/z_i) \sim \mathcal{N}(\mu_i, \Sigma_\mathrm{i})$$

2.1.3. Semi-Supervised Learning

Semi-Supervised Learning (SSL) is an approach to machine learning that can be thought of as the *intersection* of the two previous learning types we explored, i.e. the intersection between supervised learning and unsupervised learning. A small amount of labeled data is used in conjunction with a large amount of unlabeled data during the "learning", or training, process. In a dataset consisting of n elements $\mathcal{X} = \{x_1, \dots, x_n\}$ we distinguish between two *"sub-datasets",* one consisting of *k* labeled elements $\mathcal{X}_k = \{x_1, \dots, x_k\}$ with labels $\mathcal{Y}_k = \{y_1, \dots, y_k\}$ and the other of n-k unlabeled elements $\mathcal{X}_{n-k} = \{x_{k+1}, \dots, x_n\}$. The idea behind SSL is that an increase in learning accuracy, over UL, can be achieved by the usage of a relatively small number of labelled data in a dataset predominantly consisting of unlabeled data. Acquisition of correctly labelled data can be costly, as it may require input from human experts or vigorous experimental procedures. Building a large enough dataset to conduct proper training in SL, with such cost being indispensable, is inefficient, thus, making SSL an attractive alternative. There are two distinct learning approaches to Semi-Supervised Learning, transductive learning and inductive learning. The first approach entails extrapolating a prediction function from a set *exclusively consisting of labelled data*, and then having the model make predictions on a test set *exclusively consisting of unlabeled data*. In inductive learning, the prediction function is an output of the entire dataset X. (Chapelle, Schölkopf, & Zien, 2006)

2.2. Artificial Neural Networks

Artificial Neural Networks (ANNs), or Neural Networks (NNs), are Deep Learning models loosely based on the human brain. An ANN consists of thousands or millions of densely interconnected *artificial neurons / nodes*, with each connection facilitating the transmission of signals from one neuron to the next. Typically, NNs are organized into layers of neurons, with each neuron being *feed-forward*, meaning that data can only move through them in one direction. Each node is connected to other nodes via an edge and each edge will be associated with a number known as *weight*. Each node receives different signals from its connections, multiplies each signal with the associated weight of the respective edge, calculates the sum of the weight signals, checks if this sum exceeds a threshold value, and only if it does it passes the data to the next layer. The value of the weight can be positive, in which case the signal is amplified, or negative, in which case the signal is reduced. During the training phase, training data is fed to the input layer, undergoes numerous operations as it passes along the succeeding layers, and arrives at the output layer. It should be noted the values of the weights and thresholds are initially random, and are continuously adjusted until running the training set produces similar outputs

Let us discuss figure 10 strictly for illustration purposes of our thesis (this figure 4 is *not* an actual representation of the DL model we will construct later on). It depicts a feed-forward Neural Network organized in layers, which can also be regarded as a 4-6-5-3 network. The first layer is the input layer, which for the purposes of this thesis, will take as input a matrix of pixels pertaining to cell images. Then, we observe two hidden layers, and immediately after them we have the output layer, which will determine the result of the classification. The number of neurons in the output layer denote the exact number of classes available, in figure 4 there are 3 neurons in the output layer, thus 3 classes. The output layer must have as many neurons as the exact number of class labels available.

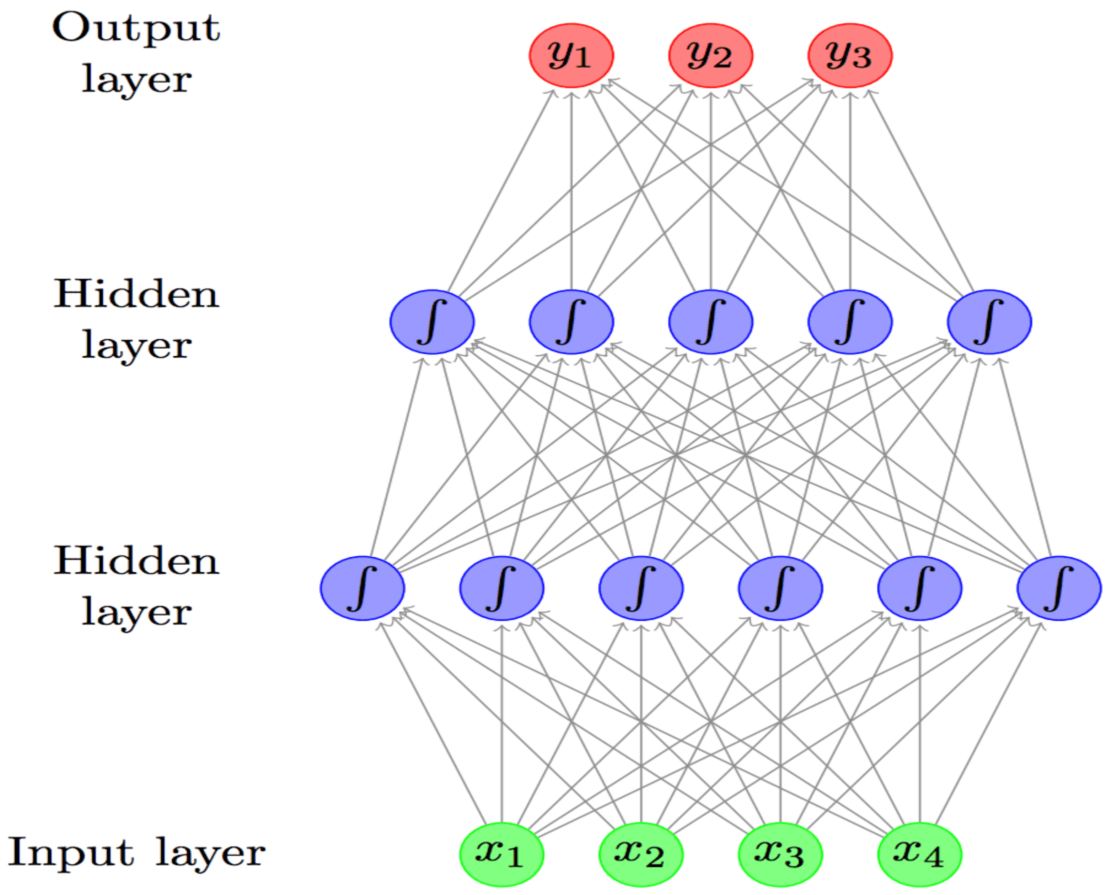

Figure 10: Feed Forward Neural Network

2.2.1. Activation Function

We previously mentioned that a node will pass data to the next layer if, and only if, a certain threshold value is exceeded, i.e. the node will activate only when the threshold value is exceeded. An activation function is added to the output end of a neural network to map the input to a range depending on the specific function used. To put it plainly, it is the activation function that defines the output of a neuron given a single input, or a set of multiple inputs. A visual representation of the activation function in a node is given in figure 10.

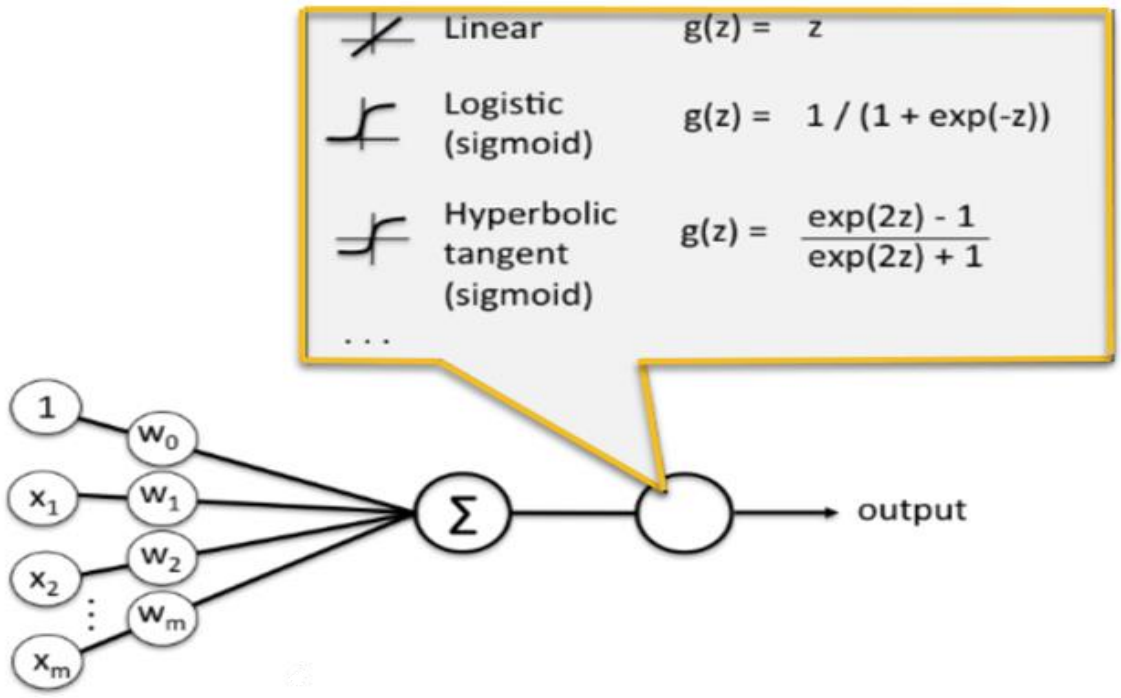

Figure 11: Structure of a node with activation function depicted

The selection of the activation function is absolutely crucial, not only because it has a huge impact on the accuracy, capability, and performance of NNs, but also because certain functions are better suited to specific situations – there is no one size fits all. For example, a step function might be an excellent choice for a binary classification problems, because the function itself has only two possible output values.

Table 1: List Of Activation Functions

Name	***Function***
Linear	$f(x) = x$
Step	$f(x) = \begin{cases} 0 \; if \; x < 0 \\ 1 \; if \; x > 0 \end{cases}$
Sigmoid	$\sigma(x) = \dfrac{1}{1 + e^{-x}}$
Hyperbolic Tangent	$tanh(x) = \dfrac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$
Rectified Linear Unit	$f(x) = max(0, x)$

The most popular and successful activation function in deep learning is the ReLU function, (Ramachandran, Zoph, & V. Le, 2017) because of its ability to produce very accurate results. (Nair & Hinton, 2010) The ReLU function is so widespread that it is commonly, and informally, referred to as the "default" activation function for Neural Networks, and it is suggested to be used whenever there is uncertainty in selecting an activation function. In contrast to other functions, ReLU usage leads to "sparse activation", a situation in which not all of the neurons in the NN are activated, it has superior gradient propagation in comparison to its peers, and it is computationally efficient due to the fact that it has only simple operations, such as addition, multiplication, and division. Compared to the Sigmoid and Hyperbolic Tangent function, it leads to faster and more effective training of NN with large and complex datasets. (Glorot, Bordes, & Bengio, Deep Sparse Rectifier Neural Networks) (Glorot & Bengio, Understanding the difficulty of training deep feedforward neural networks, 2010) However, there are some possible drawbacks that need to be mentioned. Firstly, the function is unbounded and as such can produce large values, secondly it is non-differentiable at zero, moreover it is susceptible to “dying”, (Lu, Shin, Su, & Em Karniadakis, 2020) and, lastly, there is no obtainable information from the nuance of negative outputs, because the functions considers them to be zero.

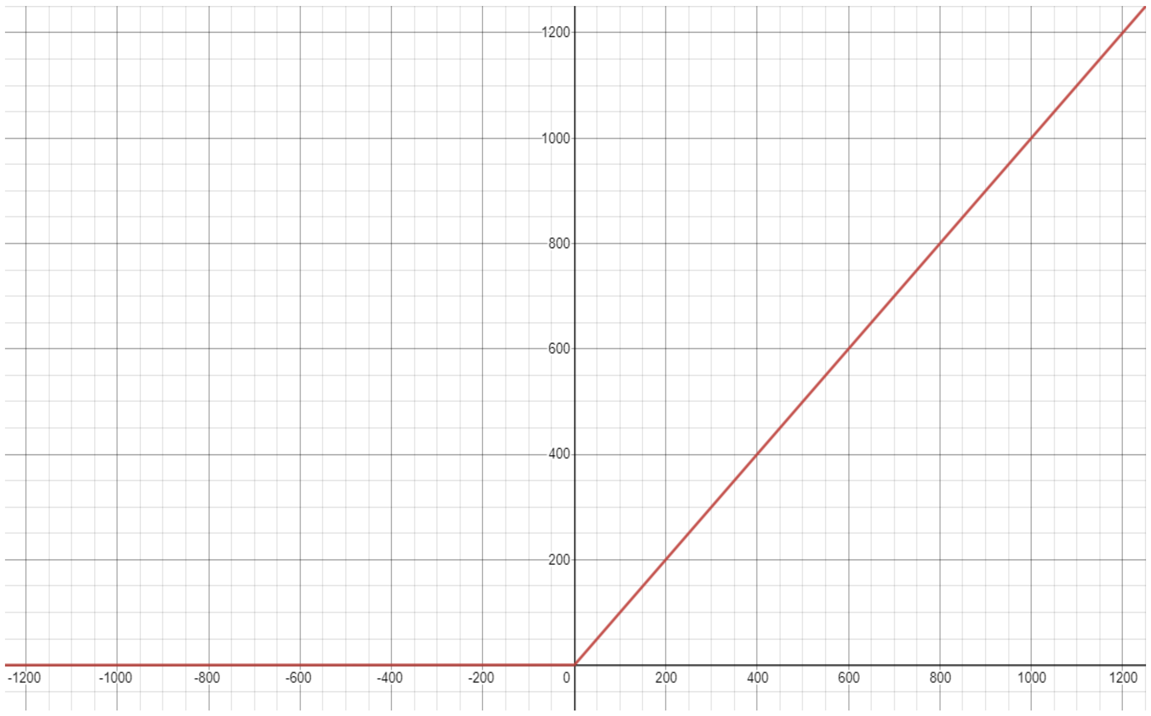

Figure 12: Graphical representation of the ReLU function

2.2.2. Loss Function

A loss function, also known as a cost function or error function, is measure of evaluating how well the ML algorithm models the dataset. If the predictions are incorrect, the value of the error function will be high, and if they are correct, the value of this function is going to be low. Due to the fact that error functions give us insight into how wrong a certain ML model is, minimizing the output of this function would yield better results. There exist a variety of such functions, each having usability in different problems types.

Regression Loss Function

Regression models produce continuous-valued output predictions given a certain input. Most Regression Error Functions estimate the error on the basis of the distance between the actual output, and the correct output. The most widely used functions of this type are:

- Mean Squared Error

(2.18) $$MSE = \frac{\sum_{i=0}^{n}(\hat{y} - y)^2}{n}$$

- Mean Absolute Error

(2.19) $$MSE = \frac{\sum_{i=0}^{n}|\hat{y} - y|}{n}$$

Binary Classification Loss Function

Binary Classification models can produce only two possible output predictions given a certain input, for example, 0 and 1. It is to be noted that two Binary Classification Error Function, namely - the hinge loss and squared hinge loss are are rarely used with ANNs, thus we only consider the Binary Cross-Entropy function in the scope of this discussion.

- Binary Cross-Entropy

(2.20) $$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \cdot log\big(h_\theta(x)\big) + \big(1 - y^{(i)}\big) \cdot \log\big(1 - h_\theta(x)\big)$$

Multiclass Classification Loss Function

- Multiclass Cross-Entropy
- Spare Multiclass Cross-Entropy
- Kullback–Leibler divergence

$$(2.21) \quad D_{KL}(P||Q) = \sum_{x \in \mathcal{X}} P(x)\log(\frac{P(x)}{Q(x)})$$

2.2.3. Gradient Descent

Gradient descent is a first-order iterative optimization technique that finds the local minimum of a differentiable function. Repeatedly taking steps in the opposite direction of the gradient of the function at the current point leads to the steepest descent. In ML, the parameters are updated by going in the opposite direction of the gradient of the loss function until we reach the point of the steepest descent. The rationale behind this procedure is that minimizing the loss function would lead to better results.

The Batch, or vanilla, gradient descent algorithm calculates the gradient of the loss function for the whole training set, which might become an impediment if the training set is large enough. However, this method allows the user to take a “hands-off” approach toward the learning rate, because once the user has selected a fixed learning rate it will suffer from decay. Furthermore, if the function is convex the algorithm is guaranteed to converge to the global minimum, and if that is not the case then it is guaranteed to converge to the local minimum. On the other hand, the training set may contain redundant elements that do not make significant contribution to the update, and as such leading to less-than-efficient usage of computational resources, as well as unnecessary added runtime.

$$(2.22) \qquad \theta = \theta - \alpha \nabla_{\theta} J(\theta$$

Equation 2.22 showcases how the parameters of the neural network get updated by gradient descent. *Θ* represents the parameters, J is the loss function, *α* is the learning rate, and $\nabla_{\Theta} J(\Theta)$ is the gradient.

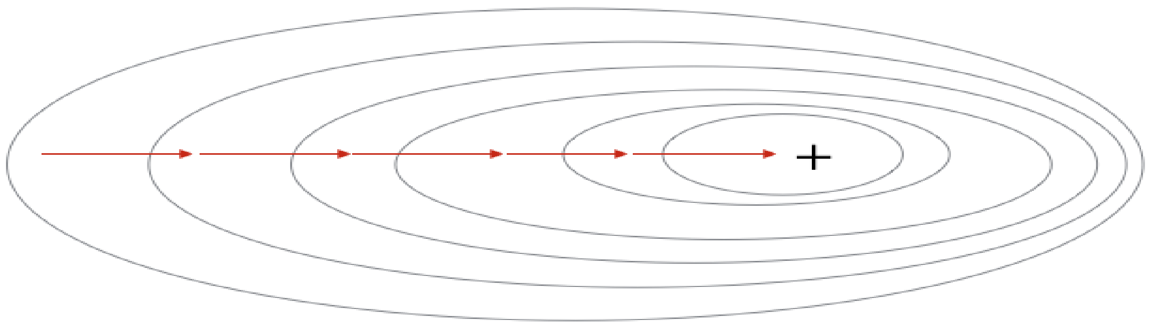

Figure 13: Batch Normalization

The Mini-batch Gradient Descent algorithm is an improvement upon Batch Gradient Descent that, instead of going over each element in the training set, opts to iterate merely over a number of elements based on the batch size *b*, resulting in faster performance, aversion of redundant examples, and additional noise, which improves the generalization error. However, noise also introduces oscillations during iterations and requires caution toward the learning-decay as the algorithm gets closer to the minimum.

(2.23) $$\theta = \theta - \alpha \nabla_{\theta} J(x^{(i,\ldots,i+batch)}, y^{(i,\ldots,i+batch)}; \theta)$$

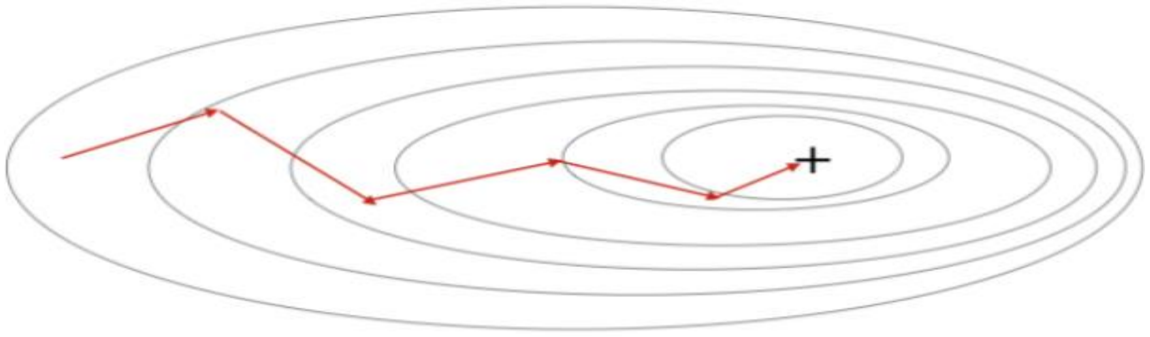

Figure 14: Mini-Batch Gradient Descent

Stochastic Gradient Descent (SGD) updates the parameters for every instance of the training set, yielding in an algorithm that shares much of its features with Mini-Batch, while introducing additional noise and resulting in a large variance.

(2.24) $\theta = \theta - \alpha \nabla_{\theta} J(x^{i}, y^{i}; \theta)$

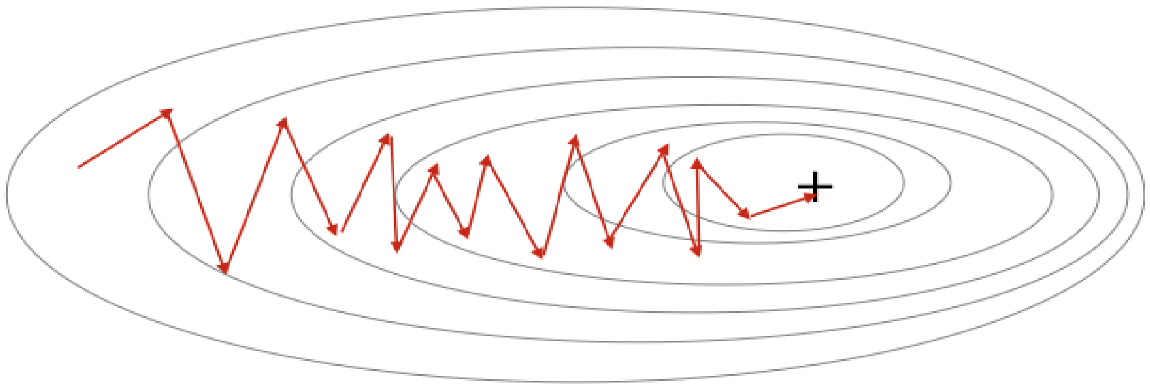

Figure 15: Stochastic Gradient Descent

2.2.4. Regularization

Regularization is a technique used to reduce error by fitting a function appropriately on the training set and avoiding the undesirable situation of overfitting. Given a certain dataset $\mathcal{D}$ with N data points, the goal in ML is to find underlying patterns, or an underlying function, in the data that can be used to predict the outcomes of new examples. A piece of data is always going to carry a certain amount of noise with it and "learning" the underlying function from such data may lead to overfitting. Let us define a generic function to represent our loss function,

(2.25) $\mathrm{LOSS} = \mathrm{Error}(y, \hat{y})$

where y and $\hat{y}$ represent the actual and the predicated output for some input. The loss function will also contain some coefficients β_{i} such that $i = 1, \dots, N$ associated with the data points that minimize the loss function. Coefficients associated with noisy data in the training set will not generalize well to new examples, thus we use regularization to zero these coefficients, decreasing the flexibility of the model, but negating overfitting.

$$\text{(2.26)} \qquad \text{LOSS} = \text{Error}(y, \hat{y}) + \lambda \sum_{i=1}^{N} \beta_i^2$$

Equation 2.26 represents a regularization technique known as Ridge Regression. The added quantity is known the shrinkage quantity, in which λ is the parameter that determines how much the coefficients get shrunk. Consider 2 parameters and let us define s is a constant that exists for each value of shrinkage factor λ, and thus we can define the constraint function $\beta_1{}^2 + \beta_2{}^2 \leq s$.

$$\text{(2.27)} \qquad \text{LOSS} = \text{Error}(y, \hat{y}) + \lambda \sum_{i=1}^{N} |\beta_i|$$

Equation 2.4.1 represents a regularization variant known as Lasso. In contrast with Ridge Regression, Lasso only penalizes the high coefficients. Similar to what we did, we can also define the constraint function for lasso $|\beta_1| + |\beta_2| \leq s$.

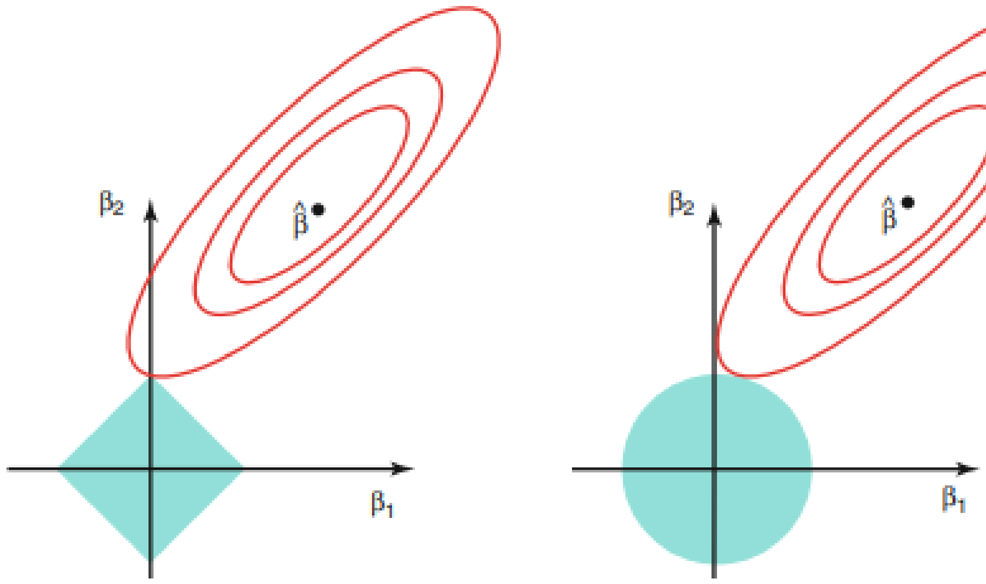

Figure 16: Lasso (left) and Ridge Regression (right). The red contours represent the error function, while the blue areas are the constraint regions

2.3. Convolutional Neural Network

Convolutional Neural Networks (CNNs) (LeCun, Bengio, & Hinton, 2015) (O'Shea & Nash, 2015) are a class of deep learning methods which dominate the field of computer vision. These methods take as input data in the form of multiple arrays, for example, an image is represented as a 2D array. Convolutional neural networks are created by placing convoluted layers and ReLU layers on top of each other, and by inserting intermediate pooling layers to keep network complexity under control. This setting is repeated until the data is reduced to a small size. Architecturally connected layers are also placed in the architecture. When CNNs are deployed in classification problems, the last layer of this type contains the classification result. Although there exists a variety of CNNs, their key components are the use of multiple layers, local connections, shared weights and pooling. The layers, essentially the building blocks of the CNN, are comprised of neurons organized in three dimensions, namely height, width, and depth. Data in a convolutional layer is organized on feature maps, which are the outputs of one filter applied to the previous layer. Each of the elements contained within this feature map retains a connection to the previous layer through a set of weights. A local sum will then be passed to the rectified linear unit (ReLU), which will apply, on each element, an activation function to the output of the activation produced by the previous layer. The convolutional layer detects conjunctions of features from the previous layer, then the pooling layer is going to merge similar features, reducing the width and height of feature maps in the network. In other words, it downsamples along the spatial dimensionality of the given input, leading to a reduction in the number of parameters within that activation. The fully-connected layers will attempt to produce class scores from the activations, which will be used for classification. Thus, the main idea is CNNs produce class scores by having the original input undergo a layer-by-layer transformation using convolutional and downsampling methods. CNNs are computationally efficient, are capable of detecting important features without human supervision, and of achieve high accuracy in classification problems. However, they require a large training set and are not invariant to rotation and scale.

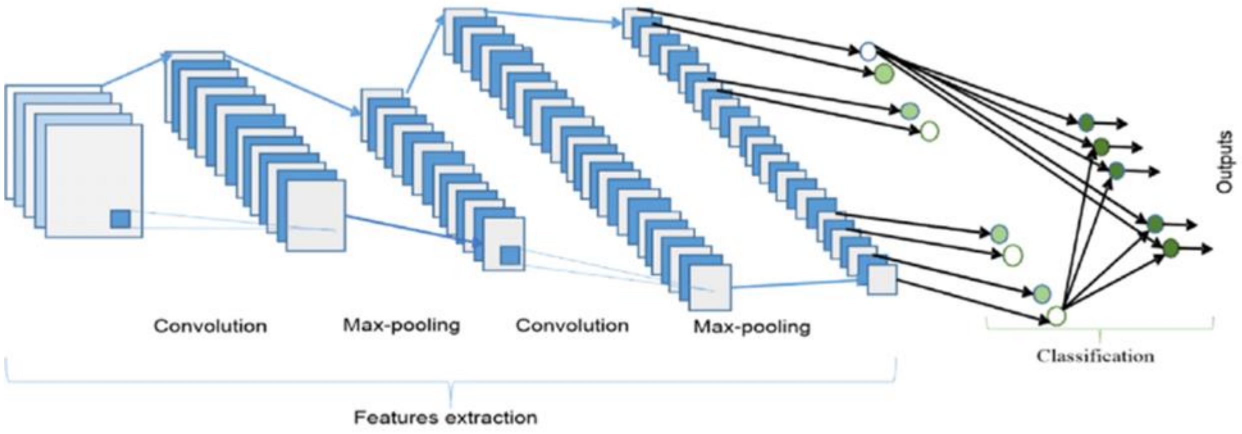

Figure 17: Typical CNN Architecture

2.3.1. Convolutional Layer

Regardless of architectural organization, the convolutional layer is the beating heart of every CNN and it is the single most important component of the network. Its principal responsibility is to extract features from a given input, such as an image. The parameters of the layer consist of a set of learnable kernels (filters) and each and every kernel has a spatially (i.e. width and height) that extends through the full depth of the input volume. During a process known as the forward pass, each kernel is convolved along the spatiality of the input volume, which leads to the computation of dot products between the entries of the kernels and the input at any position. A 2-dimensional activation map is produced for each kernel as it convolves over the width and height of the input volume. Thus, the neural network learns kernels that fire-up when they detect certain visual features of the input. The output volume of this layer is formed by stacking all kernels along the depth dimension. It is unfeasible to connect neurons to all the neurons located in the previous

previous volume, instead, each neuron establishes a connection to a local region of the input volume. The spatial extent of this connectivity for a single neuron is referred to as the receptive field of that neuron. The depth, the stride and zero-padding are the three hyperparameters that control the size of the of the output volumes.

- **The depth** of the output volume corresponds to the number kernels to be used. The moment a convolutional layers receives an input neurons along the depth may activate in presence of various features of the input.
- **The Stride** is responsible for managing the allocation of sets of neurons looking at the same region to the spatiality of the input. It is the stride that defines the shift of the kernel matrix over the image matrix, meaning a stride of 3 indicates that the filters are moved 3 pixels at a time over the image matrix.
- Zero-padding is a technique used to pad the input matrix with zeros along its borders, so that filters may be applied at these positions. It allows the size of the input to be adjusted to the specific problem requirement. This technique is used to preserve the spatiality of the input volume in the output volume, which may be necessary in order to train the network. **The size of the padding** is the third hyperparameter.

Knowing the input volume size W, the stride S, the size K of the receptive field K, and the size S of zero-padding allows us to formulate an equation to compute the value of the spatial size of the output volume V. (see equation 2.28)

$$V_{output} = \frac{W - K + 2P}{S} + 1 \tag{2.28}$$

2.3.2. Pooling Layer

In ML, pooling is a form of non-linear downsampling. There are various functions that achieve this result, with max-pooling being the most frequently used. Architecturally, the pooling layer is generally located after the convolutional layer. It progressively reduce the volume of the data, thus reducing the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence keeping overfitting under control. Maxpooling partitions the matrix input image into a set of rectangles and operates independently on each rectangle to output the maximum pixel of the corresponding rectangle. Most commonly, the pooling layer is used with filters of size 2x2 and a stride of 2, downsampling every depth slice in the input by 2 along both width and height, in which case the MAX operation would calculates a maximum over 4 pixels. The pooling layer takes as input the volume $V = WHD$, the Stride S and the spatial extent K to output the volume V_{output}. Depth remains unchanged, but the values of the width and height will be reduced according to the following equations.

$$(2.29) \qquad W_{output} = \frac{W_{input} - F}{S} - 1$$

$$(2.30) \qquad H_{output} = \frac{H_{input} - F}{S} - 1$$

2.3.3 Fully-Connected Layer

The Fully-Connected Layers appear toward the end of the convolutional neural network and have the responsibility of carrying out the final classification of the input image based on the features extracted by the previous layers. It is referred to as fully-connected, because each node in this layer has a connection to all activations in the previous layer.

2.3.4 Normalization

Batch Normalization is a technique proposed by Ioffe and Szegedy that accelerates the training of neural networks by normalizing activations and embedding the normalization in the network architecture. (Ioffe & Szegedy, 2015) Consider a mini-batch consisting of m activations $\mathcal{B} = \{x_1 \ldots x_m\}$, let the normalized values be $\widehat{x_1} \ldots \widehat{x_m}$, and the linear transformation be $y_1 \ldots y_m$, then batch normalization algorithm is represented as

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.31}$$

$$\sigma_{\mathcal{B}}{}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i - \mu_{\mathcal{B}} \tag{2.32}$$

$$\widehat{x_i} \leftarrow \frac{(x_i - \mu_{\mathcal{B}})}{\sqrt{\sigma_{\mathcal{B}}{}^2 + \varepsilon}} \tag{2.33}$$

$$y_i \leftarrow \gamma \widehat{x_i} + \beta \tag{2.34}$$

During the training phase, the mini-batch mean and the mini-batch variance (equations 2.31 and 2.32, respectively) are calculated.

Increasing the learning rates, removing Dropout, and applying other modifications speeds up training, and improves prediction accuracy. Additionally, using batch normalization reduces overfitting, decreases the model’s sensitivity to hyper parameter tuning, reduces the internal covariant shift, and lowers the reliance of gradients on the scale of the parameters.

2.3.5. Droput

Dropout is a regularization technique used to avoid overfitting in neural networks. During the training phase, nodes are dropped out of the network with probability 1 - p or kept with probability p. Only the nodes that were kept in the network participate in the training phase, thus decreasing training time and overfitting. The dropout probability is usually around 50% during the training stage, meaning that half of the nodes are ignored and only the remaining half are active.

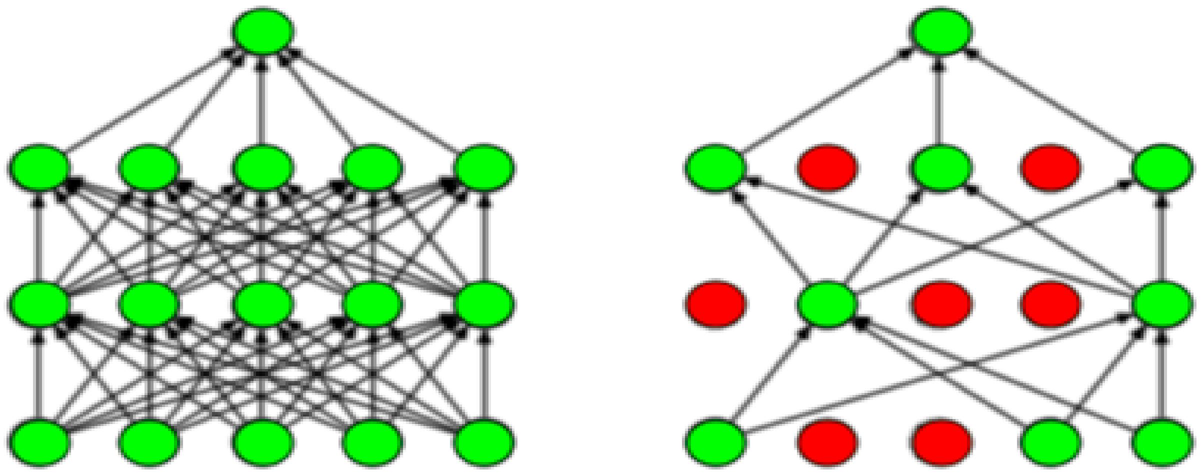

Figure 18: The image on the left showcases the network before dropout was applied, and the image on the right represents the result of dropout.

CHAPTER 3

Network Architecture & Methodology

3.1. Architecture

The purpose of this thesis is to train a Convolutional Neural Network architecture in python, which will then classify cell images into healthy or unhealthy. To build a network capable of performing such a task, we must implement each building block in the network's architecture, which are:

1. The Convolutional Layer
2. The MaxPooling2D Layer
3. The Activation Layer
4. The Dropout Layer
5. The Fully-Connected Layer

The architecture of our network is based on the LeNet architecture, which consists of 2 consecutive sets of Convolutional, RELU, and Pooling layers designed to process 32x32 images. On the other hand, our network consists of 4 consecutive sets of Convolutional, RELU, and Pooling layers each followed by a Dropout function. We added two more consecutive sets of layers to preserve the original size of the input images, which is 128x128. Each convolutional layer will consists of 50 filters of size 5x5, each activation layer will use the ReLU function, and each pooling layer will implement the maxpool function defined by a 2x2 pool size with 2x2 strides. The dropout function will be initialized with value 0.1 in the first set, 0.2 in the second set, 0.3 in third and 0.4 in the last set. Succeeding the 4th set, we have a single fully-connected layer followed by a RELU layer. The input in the fully-connected layer will be flattened and connected to 500 nodes, and each of these nodes will be associated with a ReLU function. Lastly, the softmax classifier is used to perform the classification of the input images.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{3.1}$$

Z is a vector of K real numbers, which is then normalized into a probability distribution consisting of K probabilities proportional to the exponentials of the elements of the input vector. That is, after applying softmax, each component of the input vector will be mapped to a number in the interval (0,1) interpreted as a probability and the sum of all probabilities will add up to 1.

3.2. Dataset

The dataset of this study consists of healthy and unhealthy cell images of size 128x128. The training set consists of 20,102 images, of which 12,520 depict unhealthy cells and the remaining 7,582 showcase healthy cells. The training set will be split into an 80:20 ratio, in which 80% of the images will be used to train the network, and 20%.

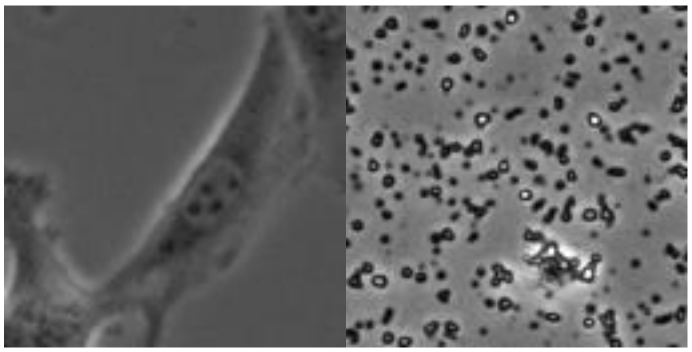

Figure 19: Two images from our dataset representing a healthy (right) and an unhealthy (left) cell

3.3. Preprocessing

We first train the network without any preprocessing to test the performance. Then we apply unsharp masking to compare the effect that preprocessing has on the model's accuracy. Unsharp masking is a linear image processing technique used to sharpen the images. It does so by using a negative blurred image to obtain a mask of the original image. The sharp details are given by the difference between the original image and the blurred version of the image. The blurring step in our implementation is a Gaussian filter.

$$sharpened = original + (original - blurred) \cdot amount \tag{3.2}$$

The value of the amount is the factor by which the details of the images will be amplified and the value of the radius affects the size of the edges to be enhanced, thus a smaller radius enhances smaller-scale detail. Larger radius values may produce halos at the edges, a detectable faint light rim around objects. Fine detail needs a smaller radius. Also, reducing the amount allows more of the radius, and vice versa.

A study using the same training dataset was able to achieve improvements on classification accuracy after applying preprocessing to the training dataset. (Uka, et al., 2020) This thesis aims to explore the effect that applying preprocessing on the testing sets has on classification accuracy of model trained on an unpreprocessed training set.

CHAPTER 4

Results

We first train our CNN on a dataset, let’s call it trainSet1, of 20,102 128x128 images, of which 12,520 depict unhealthy cells and the remaining 7,582 showcase healthy cells. During this training phase, 80% of the dataset will be used to train the network, and the remaining 20% will be used to validate it.

Table 2: Training results with epoch 20, batchsize 64

	precision	**recall**	**F1-score**	**support**
Healthy	0.98	0.93	0.95	2504
Unhealthy	0.89	0.97	0.93	1517
Accuracy			0.94	4021
Macro average	0.94	0.95	0.94	4021
Weighted average	0.95	0.94	0.94	4021

Using a batch-size of 64 and 20 epochs we achieved the results given in table 2 and a graphical interpretation of the results presenting the loss, accuracy, validation loss, and validation accuracy is given in figure 22.

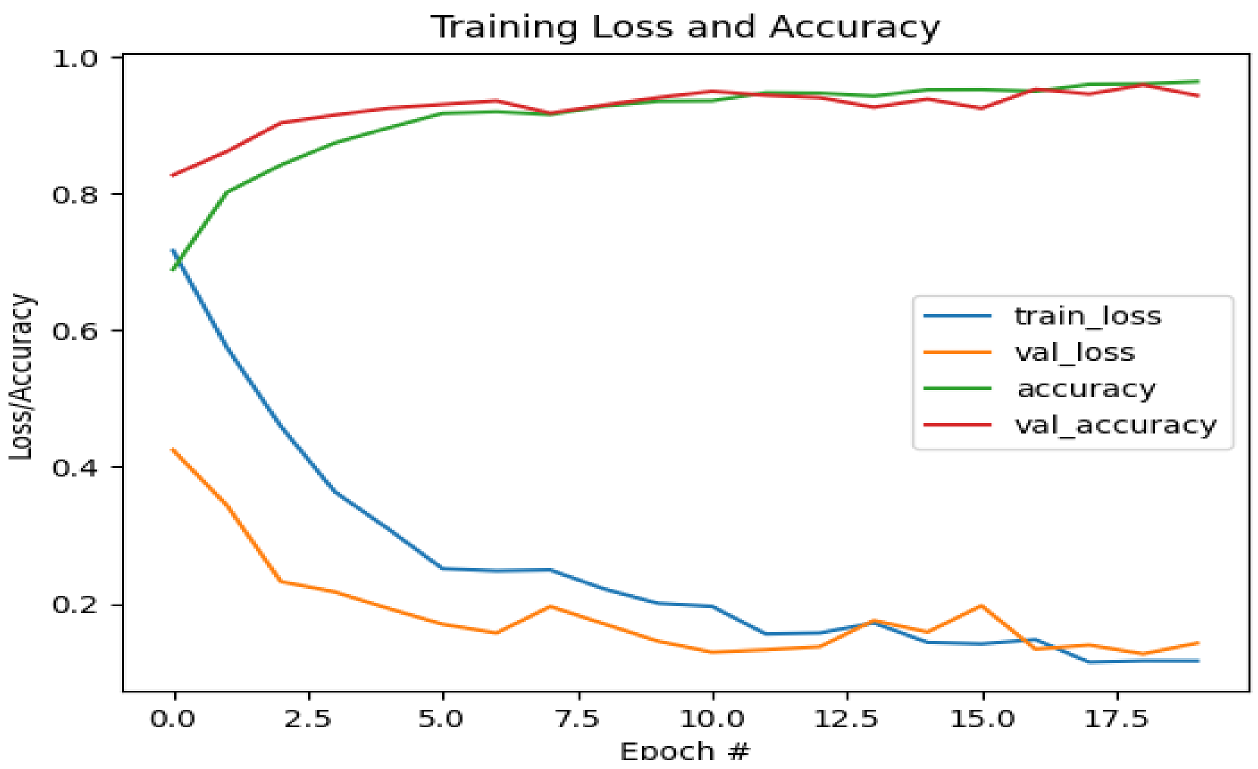

Figure 20: Graphical interpretation of the results

4.1. Experiment 1 – No Preprocessing Applied

After training this network on the training set, we test it on another set of entirely new examples, let’s call it testSet1, consisting of 11,339 cell images. We were able to achieve an accuracy of approximately 98%. The cell images as well as the predicted labels are given in figure 23.

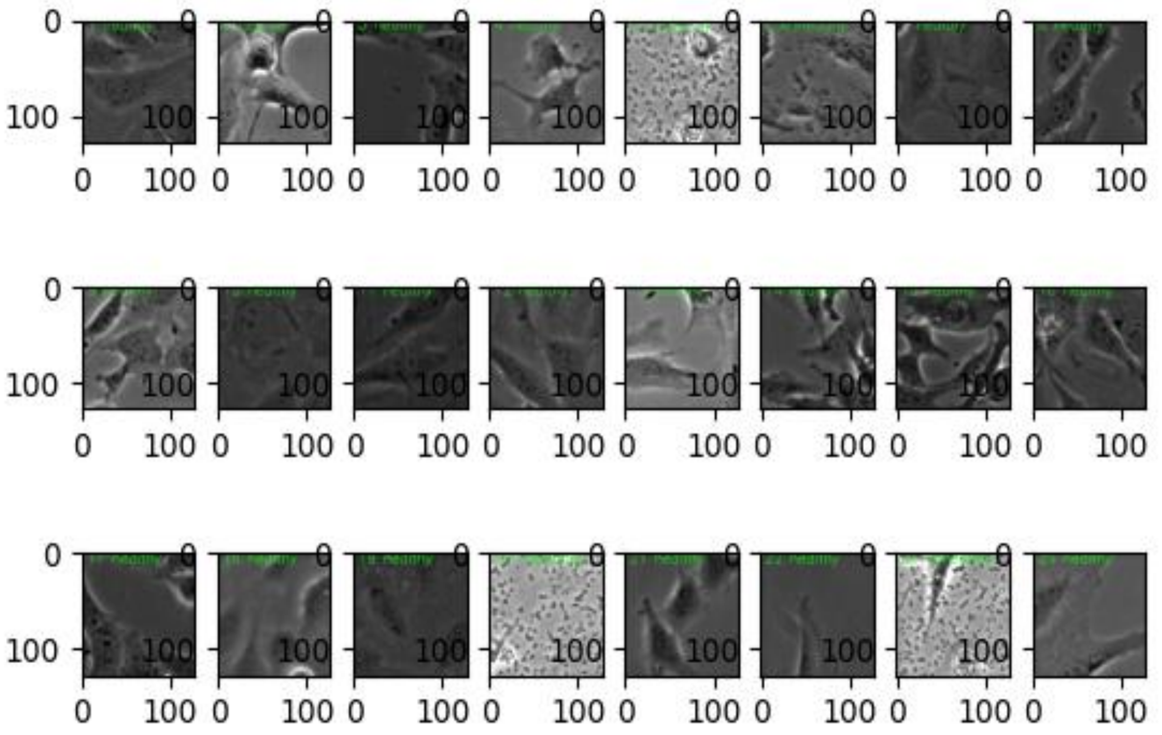

Figure 21: Cell images of testSet1 and their predicted labels

Let us test our network on the second testing set, let's call it testSet2, consisting of 11,360 images. We test with this new dataset the neural network we trained earlier with unpreprocessed images, and we achieve a classification accuracy of approximately 79%. Once again, we give a sample of the predicted classes in figure 24.

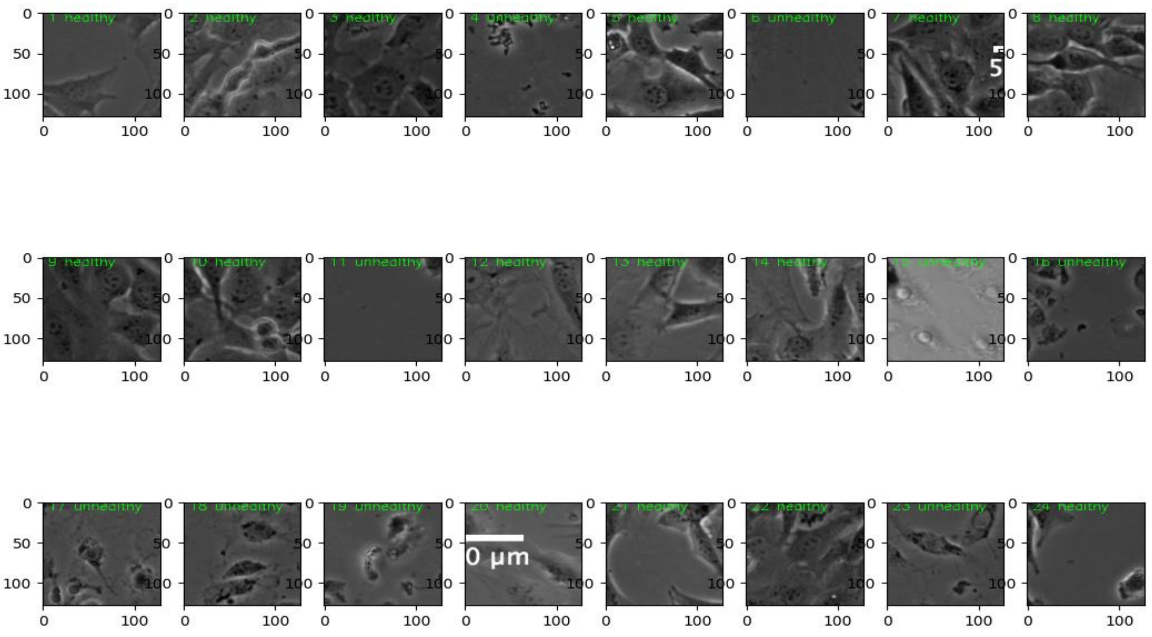

Figure 22: Cell images of testSet2 and their predicted labels

We are interested in comparing the results that preprocessing will have on the classification accuracy, thus let us apply preprocessing to our two testing sets.

4.2. Experiment 2 – Unsharp Masking with Radius = 5, Amount = 2

We apply unsharp masking with radius 5 and amount=2 to testSet1, test our trained model on this preprocessed set and achieve an accuracy of 73%, a staggering decrease of almost 25% from the accuracy of testing with the unpreprocessed images. We apply the same preprocessing to testSet2 and achieve an accuracy of approximately 69%, a decrease of almost 10% from the accuracy of testing testSet2 with no preprocessing applied. In the dataset, the unhealthy cell images have more enhanced edges, and the healthy cell images have lower contrast. Recall that the preprocessing technique we are applying is unsharp masking, which sharpens the edges of the images, thus healthy images probably have more pronounced edges confusing the model into predicting them as unhealthy.

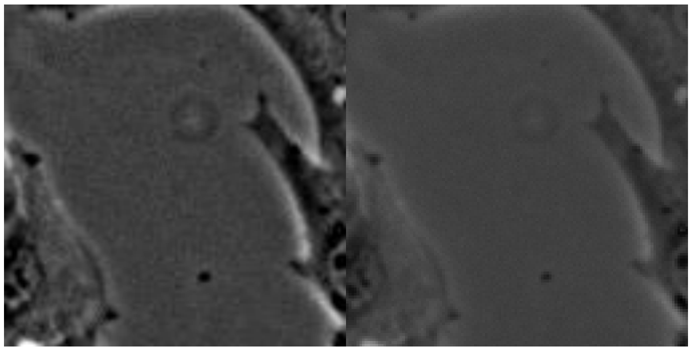

Figure 23: Sharpened image (left) and original image (right). Amount = 5, Radius = 2.

4.3. Experiment 3 – Unsharp Masking with Radius = 1, Amount = 1

Setting radius to 1 and the amount to 1, we perform the same experiment as we did previously. For testSet1 we achieved an accuracy of 98%, which is approximately equal to the accuracy in the first experiment, and significantly better than the accuracy in the second experiment. We noticed an increase in accuracy when testing on testSet2 with an accuracy of approximately 81%, a 2% increase in accuracy from experiment 1 and a 12% increase in accuracy from experiment 2. The value of the amount is the factor by which the details of the images will be amplified, decreasing this value to 1 may have providing just enough detail amplification so as to not confuse the model into falsely classifying the images, but also provided the necessary detail to help the model distinguish between images in testSet2.

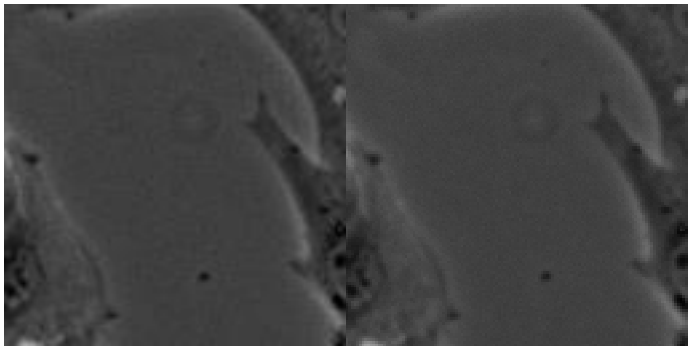

Figure 24: Sharpened image (left) and Unsharpened image (right). Amount = 1, Radius = 1.

4.4. Experiment 4 – Unsharp Masking with Radius = 10, Amount = 1

Let us now explore the effect that an increase in blurring has in classification accuracy. We are increasing the value of the radius to 10, while keeping the value of the amount at 1. Testing the model with the preprocessed images of testSet1 yields a classification accuracy of about 96%, an almost 2% decrease from the classification accuracy experiment 3 and 1. We noticed a similar decrease in classification accuracy for the preprocessed images of testSet2, which was 78% for this specific experiment – a 1% decrease from experiment 1 and a 3% decrease from experiment 2. A smaller radius enhances smaller-scale detail, while a larger radius enhances larger-scale details. In experiment 3, where we used a radius = 1, enhancing the smaller scale details may have allowed the model to pick up on the necessary features to properly conduct classification. With radius=10, we are enhancing larger-scale details

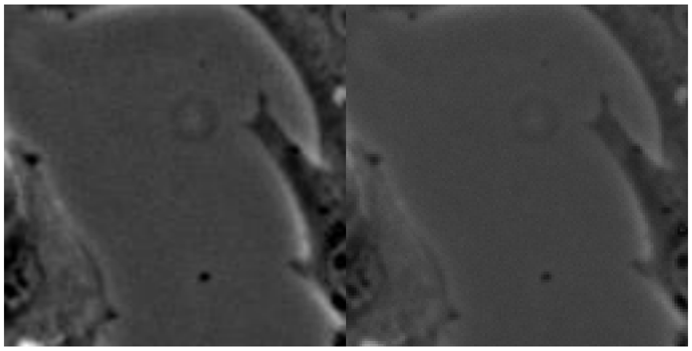

Figure 25: Sharpened image (left) and Unsharpened image (right). Amount = 1, Radius = 10.

4.5. Experiment 5 – Unsharp Masking with Radius = 1, Amount = 0.5

Setting radius to 1 and amount to 0.5 yields a classification accuracy of approximately 99% on TestSet1, the best result we have seen for the first testing set. On the other hand, for testSet2, we achieved a classification accuracy of slightly over 79%, a bit better than in the first experiment, 10% better than the second experiment, 2% worse than the third experiment, and 1% better than the previous experiment.

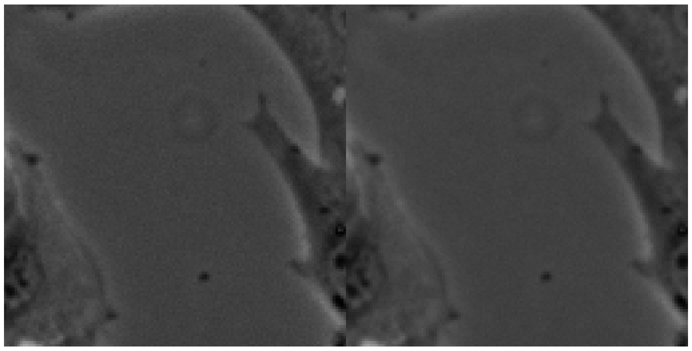

Figure 26: Sharpened image (left) and Unsharpened image (right). Amount = 0.5, Radius = 1.

CHAPTER 5

Conclusion

Medical imaging allows healthcare workers to provide accurate clinical analysis and diagnosis, as well as prescribe proper treatment. The combination of this field of study with Deep Learning expedites the accuracy of diagnosis while simultaneously producing accurate results. Cell image classification is a particular branch of study in this area that has the potential to advance our knowledge and understanding of molecular biology, molecular mechanism for drug resistance, effective drug discovery, and accurate classification of fatal diseases, such as cancerous cells. Convolutional Neural Networks dominate the field of computer vision and have been deployed with remarkable results for classification problems. In this thesis, a Convolutional Neural Network was deployed in order to group cell images into two distinct class, healthy and unhealthy.

As a basis for our model, a LeNet architecture was used, which was modified and customized in order to suit the images of the dataset. This was done in order to retain the original image size of 128x128, as LeNet is designed for a 28x28 pixel input image. We trained our model on a dataset of 20,102 cell imagese with 20% of the dataset used for validation purposes, for which we achieved an accuracy of 94%. Running the trained model produced satisfactory classification results, especially for testSet1 with which we were able to achieve an accuracy of approximately 97%, while running the trained model for testSet2 produced an accuracy of 79%.

Lastly, we explored the effect preprocessing has on classification, where we noticed a substantial decreases in accuracy in experiment 2 and experiment 4 relative to experiment 1, but an increase in accuracy in experiment 3. Experiment 5 provided the best results on one of our testing sets, and the second best in the other. As we stated earlier, since the preprocessing technique we applied to the testing sets is unsharp masking, which enhances the edges of the images, and in our dataset unhealthy cell images typically have more pronounced edges then healthy images, the network may mistakenly classify healthy images as unhealthy. Thus, appropriate values for the amount and radius ought to be selected, with respect to the dataset.

References

Altman, N. S. (1990). An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression. *The American Statistician, 46*, 175-185.

Chapelle, O., Schölkopf, B., & Zien, A. (2006). 1.1.2 Semi-Supervised Learning. In *Semi-Supervised Learning* (pp. 2-3). Cambridge, Massachusetts: The MIT Press.

Friedman, J. H., Tibshirani, R., & Hastie, T. (2008). 4.5.2 Optimal Separating Hyperplanes. In *The Elements of Statistical Learning (Second Editon)* (p. 134). New York: Springer.

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research - Proceedings Track, 9*, 249-256.

Glorot, X., Bordes, A., & Bengio, Y. (n.d.). Deep Sparse Rectifier Neural Networks. *Journal of Machine Learning Research, 15*.

Heung-Il, S., Lee, S.-W., & Shen, D. (2013). Latent feature representation with stacked auto-encoder for AD/MCI diagnosis. *Brain Structure and Function*, 841–859.

Hu, L.-Y., Huang, M.-W., Ke, S.-W., & Tsai, C.-F. (2016). The distance function effect on k-nearest neighbor classification for medical datasets. *SpringerPlus* .

Ioffe, S., & Szegedy, C. (2015). *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.*

Kim, J., Kim, B.-S., & Savarese, S. (2012). Comparing Image Classification Methods: K-Nearest-Neighbor and Support-Vector-Machines . *AMERICAN-MATH'12/CEA'12: Proceedings of the 6th WSEAS* *international conference on Computer Engineering and Applications, and Proceedings of the 2012 American conference on Applied Mathematics*, 133–138.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep Learning. *Nature*, 436-444.

Lu, L., Shin, Y., Su, Y., & Em Karniadakis, G. (2020). Dying ReLU and Initialization: Theory and Numerical Examples. *Communications in Computational Physics, 28*, 1671-1706.

Nair, V., & Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines Vinod Nair. *In Proceedings of the 27th International Conference on International Conference on Machine Learning*, 807–814.

O'Shea, K., & Nash, R. (2015). An Introduction to Convolutional Neural Networks. *ArXiv e-prints*.

Ramachandran, P., Zoph, B., & V. Le, Q. (2017). Searching for Activation Functions.

Ronneberger, O., Fischer, P., & Brox, T. (2015). U-Net: Convolutional networks for biomedical image segmentation. Proceedings of Medical Image Computing and Computer-Assisted Intervention. *MICCAI*, 234–241.

Schölkopf, B. (2000). The kernel trick for distances. *NIPS'00: Proceedings of the 13th International Conference on Neural Information Processing Systems*, 283–289.

Song, Y., Liang, J., Lu, J., & Zhao, X. (2017). An efficient instance selection algorithm for k nearest neighbor regression. *Neurocomputing, 251*, 26-34.

Suk, H.-I., Wee, C.-Y., Lee, S.-W., & Shen, D. (2016). State-space model with deep learning for functional dynamics estimation in resting-state fMRI. *NeuroImage*, 292–307.

Uka, A., Polisi, X., Barthes, J., Halili, A. N., Skuka, F., & Vrana, N. E. (2020). Effect of Preprocessing on Performance of Neural. *2020 International Conference on Computing, Electronics & Communications Engineering.*

Uka, A., Polisi, X., Halili, A., Dollinger, C., & Vrana, N. E. (2017). Analysis of cell behavior on micropatterned surfaces by image processing algorithms. *IEEE EUROCON 2017 -17th International Conference on Smart Technologies.* Ohrid.

Uka, A., Polisi, X., Halili, A., Dollinger, C., & Vrana, N. E. (2017). Analysis of cell behavior on micropatterned surfaces by image processing algorithms. *IEEE EUROCON 2017 -17th International Conference on Smart Technologies.* Ohrid.

Uka, A., Tare, A., Polisi, X., & Panci, I. (2020). FASTER R-CNN for cell counting in low contrast microscopic images. *2020 International Conference on Computing, Networking, Telecommunications & Engineering Sciences Applications (CoNTESA).*

Wu, G., Minjeong, K., Munsell, B., & Wang, Q. (2015). Scalable High Performance Image Registration Framework by Unsupervised Deep Feature Representations Learning. *IEEE Transactions on Biomedical Engineering* , 1505-1516.

Zhang, S., Li, X., Zong, M., Zhu, X., & Wang, R. (2018). Efficient kNN Classification With Different Numbers of Nearest Neighbors. *IEEE Transactions on Neural Networks and Learning Systems, 29*, 1774-1785.

Appendix

train_model.py

```
# import the necessary packages
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from keras.preprocessing.image import img_to_array
from keras.utils import np_utils
from pyimagesearch.nn.conv.lenet import LeNet
from pyimagesearch.nn.conv.customLenet import LeNetCustom
from imutils import paths
import matplotlib.pyplot as plt
import numpy as np
import argparse
import imutils
import cv2 as cv
import os
import PIL
import xlsxwriter


# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-d", "--dataset", required=True,
                help="path to input dataset of faces")
ap.add_argument("-m", "--model", required=True,
                help="path to output model")
```

```
ap.add_argument("-mj", "--model_json", required=True,
                help="path to output model to json")
args = vars(ap.parse_args())


data = []
labels = []

# loop over the input images
for imagePath in sorted(list(paths.list_images(args["dataset"]))):
    pil_image = PIL.Image.open(imagePath).convert('RGB')
    open_cv_image = np.array(pil_image)
    open_cv_image = open_cv_image[:, :, ::-1].copy()  # Convert RGB to BGR
    image = cv.cvtColor(open_cv_image, cv.COLOR_BGR2GRAY)
    image = imutils.resize(image, width=128)
    image = img_to_array(image)
    data.append(image)
    label = imagePath.split(os.path.sep)[-2]
    label = "healthy" if label == "healthy" else "unhealthy"
    labels.append(label)

data = np.array(data, dtype="float") / 255.0
labels = np.array(labels)
le = LabelEncoder().fit(labels)
labels = np_utils.to_categorical(le.transform(labels), 2)

classTotals = labels.sum(axis=0)
classWeight = classTotals.max() / classTotals

#partition into training and testing
(trainX, testX, trainY, testY) = train_test_split(data,
                                                  labels, test_size=0.20,
stratify=labels, random_state=42)

# compile the network
model = LeNetCustom.build(width=128, height=128, depth=1, classes=2)
model.compile(loss="binary_crossentropy", optimizer="adam",
              metrics=["accuracy"])
# train the network
H = model.fit(trainX, trainY, validation_data=(testX, testY),
              class_weight=classWeight, batch_size=64, epochs=10, verbose=1)

# evaluate the network
predictions = model.predict(testX, batch_size=64)
print(classification_report(testY.argmax(axis=1),
                            predictions.argmax(axis=1), target_names=le.classes_))

# save the model to disk
model.save(args["model"])
```

```
model_json = model.to_json()
with open(args["model_json"], 'w') as json_file:
    json_file.write(model_json)

#####

workbook = xlsxwriter.Workbook('Q6_v1.xlsx')
worksheet = workbook.add_worksheet()
worksheet.write(0, 0, "Accuracy")
worksheet.write(0, 1, "Val_accuracy")
worksheet.write(0, 2, "Loss")
worksheet.write(0, 3, "Val_loss")
row = 1
col = 0

for item in H.history['accuracy']:
    worksheet.write(row, col, item)
    row += 1
row = 1
col = 1
for item in H.history['val_accuracy']:
    worksheet.write(row, col, item)
    row += 1
row = 1
col = 2
for item in H.history['loss']:
    worksheet.write(row, col, item)
    row += 1
row = 1
col = 3
for item in H.history['val_loss']:
    worksheet.write(row, col, item)
    row += 1

workbook.close()

#######

# plot the training + testing loss and accuracy
# plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, 10), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, 10), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, 10), H.history["accuracy"], label="accuracy")
plt.plot(np.arange(0, 10), H.history["val_accuracy"], label="val_accuracy")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.show()
```

costumLenet.py

```
# import the necessary packages
from keras.models import Sequential
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Flatten
from keras.layers.core import Dense
from keras.layers.core import Dropout
from keras import backend as K


class LeNetCustom:
    @staticmethod
    def build(width, height, depth, classes):
        # initialize the model
        model = Sequential()
        inputShape = (height, width, depth)

        # if we are using "channels first", update the input shape
        if K.image_data_format() == "channels_first":
            inputShape = (depth, height, width)

        # first set of CONV => RELU => POOL layers
        model.add(Conv2D(20, (5, 5), padding="same",
                         input_shape=inputShape))
        model.add(Activation("relu"))
        model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
        model.add(Dropout(0.1))  # adding new keras.layer

        # second set of CONV => RELU => POOL layers
        model.add(Conv2D(50, (5, 5), padding="same"))
        model.add(Activation("relu"))
        model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
        model.add(Dropout(0.2))  # adding new keras.layer

        # third set of CONV => RELU => POOL layers  for 64x64
        model.add(Conv2D(50, (5, 5), padding="same"))
        model.add(Activation("relu"))
        model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
        model.add(Dropout(0.3))  # adding new keras.layer
        # fourth set of CONV => RELU => POOL layers for 128 x 128
        model.add(Conv2D(50, (5, 5), padding="same"))
        model.add(Activation("relu"))
        model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
        model.add(Dropout(0.4))  # adding new keras.layer

        # first (and only) set of FC => RELU layers
        model.add(Flatten())
        model.add(Dense(500))
        model.add(Activation("relu"))

        # softmax classifier
        model.add(Dense(classes))
        model.add(Activation("softmax"))


        return model
```

unsharpmask.py

```
from skimage.filters import unsharp_mask
from PIL import Image
from imutils import paths
import numpy as np
import argparse
import cv2 as cv
import PIL
import matplotlib.cm as mplcm
from PIL import Image as im

#create an empty file in datasets/cells/ and name it whatever you like (smthng like
preprocessedQ suggested for cohesion)
#Create two subfiles in this dataset, one to hold healthy preprocessed images, the
other to hold unhealthy preprocessed images.

copyPath="E:\\K Cuedari\\K Cuedari\\datasets\\Cells\\preprocessedQ6\\healthy"
ap = argparse.ArgumentParser()
ap.add_argument("-d", "--dataset", required=True, help="path to input dataset of
faces")
args = vars(ap.parse_args())

for imagePath in sorted(list(paths.list_images(args["dataset"]))):
    copyString = imagePath[25:len(imagePath)] #healthy path
#   copyString = imagePath[27:len(imagePath)] #unhealthy path, comment line 23 and
uncomment this line if you want to preprocess, then save, unhealthy images
    pil_image = PIL.Image.open(imagePath).convert('RGB')
    open_cv_image = np.array(pil_image)
    open_cv_image = open_cv_image[:, :, ::-1].copy()  # Convert RGB to BGR
```

```
ppimg = unsharp_mask(open_cv_image, radius=5, amount=2)
imageToBeSaved = Image.fromarray((ppimg * 255).astype(np.uint8))
imageToBeSaved.save(copyPath+copyString)
```