THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

Arabic Handwritten Characters Recognition using Support Vector Machines and Neural Networks

by
Haythm Alshehab
School of Information Technology and Electrical Engineering,
University of Queensland.

Submitted for the degree of
Bachelor of Engineering
in the division of Software Engineering,
November 2017.

Haythm Alshehab
[redacted]

June 28, 2018

Prof Michael Brünig
Head of School
School of Information Technology and Electrical Engineering
University of Queensland
St Lucia, Q 4072

Dear Prof Michael Brünig,

In accordance with the requirements of the degree of Bachelor of Engineering in the division of Software Engineering, I present the following thesis entitled "Arabic Handwritten Characters Recognition using Support Vector Machines and Neural Networks". This work was performed under the supervision of Prof. Neil Bergmann and Assoc. Prof. Marcus Gallagher.

I declare that the work submitted in this thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at the University of Queensland or any other institution.

Yours sincerely,

[redacted]

Haythm Alshehab

To ...

Acknowledgments

Foremost, I would like to express my deepest appreciation and gratitude to my supervisor Professor Neil Bergman who throughout the year guided me through the difficulties of my research/thesis. His immense knowledge was always an inspiration to me to work harder at my research. I also sincerely thank Mr Jordan Bishop and Assoc. Prof. Marcus Gallagher for their continuous support. I thank as well my family and friends for their ongoing support, trust and encouragement.

Abstract

Handwritten data sometimes get very difficult to recognize by humans let alone computers. Handwriting recognition and optical character recognition are among the oldest problems in computer vision. Arabic script is even harder to recognize than English one. The reason being that depending on the position of the letter in a word, its shape changes. Also there are many Arabic letters which appear quite similar in handwriting. This project aims at performing Character Recognition on a dataset of Arabic Handwritten characters. We use Support Vector Machines and Neural Networks to achieve our goal. Broadly speaking, we first extract the features from the raw images, transform them to achieve non-linearity and then classify them using either an SVM or a Softmax classifier. We used both classical and modern methods to perform these tasks. We used histogram of oriented gradients and convolutional neural networks to extract features, Kernels followed by SVM and fully-connected network to perform non-linear classification. In total we apply eight different methods (depending on the feature extraction and transformation and classifier being used) and report a 93.27% test accuracy.

Keywords: Support Vector Machines, Histogram of Oriented Gradients, Handwriting Recognition, Radial Basis Function, Fully-connected Neural Network, Convolutional Neural Network.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Writing has been credited with one of the first few inventions that pushed the mankind towards civilization. The invention of paper and printing press further made their contributions in making humans more and more advanced. Paper became a medium of writing and was used in all the fields of life including education, politics etc. In the 19^{th} and 20^{th} century, inventions like the radio, the television, the computer and the internet brought the era of electronic media. Because of its space saving and ease to access, it was feared that electronic communication will soon replace the paper but the paper never lost its value. Therefore the popularity of the paper because of its convenience in use, demands the development of methods to make the computers understand the text.

Optical Character Recognition is the conversion of image of handwritten or printed text into the machine code [1]. The concept of OCR has been around since before the computers have even been invented. Handwriting Recognition (HWR) is the conversion of handwritten text coming from different sources into a form which computer can easily understand [1]. So in a sense, HWR is a sub-category of OCR. Note that, OCR only translates the text into the machine code. It doesn't tell what the meaning of data is. For that purpose we have to integrate it with the Natural Language Processing methods. In fact, the OCR's accuracy can be increased by further applying spelling correction etc.

Handwritten data can be really messy and equally important. This can be realized with an historical example. The famous French Mathematical Prodigy Évariste Galois who is considered as one of the founders of Abstract Algebra, died in a dual at age 20. Before the day of the dual, he compiled all of his findings and sent them to famous mathematicians of the time. His work remained untouched for a decade and even after that when mathematicians tried reading his work they found it extraordinary but very hard to read because his handwriting was very bad [43]. He was also denied of entering many prestigious universities because of his handwriting. This explains, how hard the problem of HWR can get.

Name	Isolated	Initial	Medial	Final
alif	ا	-		ـا
baa	ب	بـ	ـبـ	ـب
taa	ت	تـ	ـتـ	ـت
thaa	ث	ثـ	ـثـ	ـث
jiim	ج	جـ	ـجـ	ـج
Haa	ح	حـ	ـحـ	ـح
khaa	خ	خـ	ـخـ	ـخ
daal	د	-		ـد
dhaal	ذ	-		ـذ
raa	ر	-		ـر
zaay	ز	-		ـز
siin	س	سـ	ـسـ	ـس
shiin	ش	شـ	ـشـ	ـش
Saad	ص	صـ	ـصـ	ـص
Daad	ض	ضـ	ـضـ	ـض
Taa	ط	طـ	ـطـ	ـط
Dhaa	ظ	ظـ	ـظـ	ـظ
ayn	ع	عـ	ـعـ	ـع
ghayn	غ	غـ	ـغـ	ـغ
faa	ف	فـ	ـفـ	ـف
qaaf	ق	قـ	ـقـ	ـق
kaaf	ك	كـ	ـكـ	ـك
laam	ل	لـ	ـلـ	ـل
miim	م	مـ	ـمـ	ـم
nuun	ن	نـ	ـنـ	ـن
haa	ه	هـ	ـهـ	ـه
waaw	و	-		ـو
yaa	ي	يـ	ـيـ	ـي

Figure 1.1: Shape of Letters depending on their Position in the word

HWR can be performed offline or online, the difference being that in online recognition, data is provided as a stream or sequence in time whereas in an offline recognition system, the data is previously saved and is processed later. Our concern in this project will be with Offline Arabic Handwriting Recognition (AHWR). We introduce the Arabic Language and why we need to study methods of applying HWR to it in the following sections.

1.1 Motivation

Arabic is the fourth most spoken language in the world. It is used by 295 million people from fifty seven different countries [1]. The vocabulary and grammar may differ from country to country or even city to city but the written Arabic which is referred to as Modern Standard Arabic (MSA) is used officially and is same throughout the Arabian Peninsula (as the name suggests, it is a standard). The Arabic script has also inspired many other languages' scripts as well [5]. The most popular of them are Persian and Urdu which use Nastalik Script, a variation of the Arabic script. Arabic alphabets differ from English and other languages which use Latin or Roman script alphabets in a sense that Arabic alphabets change depending on the position in the word (See Next section for more detail). Therefore the methods that are studied on Arabic Documents can be easily extended to Persian and Urdu documents (after some variations) making the task of Arabic Handwritten Recognition (AHWR) considerably important.

1.2 Arabic Writing

Arabic Language has twenty eight letters and as we indicated in the previous section, most of them possess different shapes (mainly two to four) depending on their position in the word and is written from right to left unlike the languages based on Latin script. [5, Fig. 1.1] lists the Arabic alphabets, their pronunciation and their different appearances. The appearances correspond to the positions namely; the beginning, the middle, the end and the isolation. Letters like daal cannot be connected to any letter following it in a word, therefore the beginning and the middle shapes make no sense for them and thus have been shown as blank. The letters alif, waaw and yaa are used to produce long vowel sounds, when they are in the middle or end of the word. Only alif act as a vowel, similar to the letter a, when it appears in the beginning of the word, the other two act as consonants. The letter ayn and haa sometimes act as vowels in Persian and Urdu but they always have consonant sounds in Arabic (Here we are talking about haa and not Haa as they are both different in pronunciation).

In English if we compare the letters a and o handwritten in small, the letter I in capital and the letter L in small (as l) both printed, we notice that they are not very easily distinguishable. Arabic Language has several such cases. Consider the consecutive letters jiim, Haa and khaa in [5, Fig. 1.1]. They have different pronunciation and but in written form, they differ from each other only by the placement of the dot. Similarly the letters baa, taa and thaa differ from each other by the placement and number of dots. Other examples can be seen in the Fig. 1.1. Once again we should point out that, Arabic has the least number of such cases. Other languages have more of these for example Sindhi Language, a South Asian Language comprising of fifty six alphabets and based on Arabic script, has nine such cases where a letter looks like baa but has different number of dots on or under it [5, Fig. 1.2].

Arabic Language also has something called the diacritical marks or simply diacritics which are shown in the [5, Fig. 1.3]. They have functions like producing short-vowel and long-vowel sounds. Consider, in [5, Fig. 1.3], the first three letters from the right in the first line. These are the three kinds of short vowel sounds and notice that they are produced just by the diacritics. While the fourth, sixth and seventh are the long vowel sounds. These are produced by the combination of the diacritics with the vowel letters. Sometimes diacritics are also combined with the vowel letters to produce new vowel sounds similar to the diphthongs in English Language. There is only one such case in Arabic Language, as depicted by the sixth example from the right in [5, Fig. 1.3], but Persian and Urdu have some other cases as well which use the same principle. There is also a diacritic which is used at the end of the syllables. It is linked with one of the previous short-vowel diacritics to produce the complete syllable (See the first example from the right in the second line of [5, Fig. 1.3]). There is another diacritic which when put on top of a consonant along with one of the short-vowel diacritic produces doubled or emphasized sound. See, for example, the second, third, fourth and fifth letter from the right in second line of Fig. 1.3. Lastly, [5, Fig. 1.4] depicts two other kinds of diacritics namely the Nunation, which is the doubling of the short-vowel sound, and Hamza, which is the glottal stop.

Diacritics are normally omitted from the text [5, Fig. 1.5] and only a few of them which are really important are placed. Some texts also use them as they are. There are many other properties of Arabic words and letters for example the notion of sub-word and ligature and the writing of the letters relative to the baseline that we will not discuss in detail as they are out of the scope of our project.

The above discussion clearly shows that the real challenge is not only in detecting the actual letters in an Arabic text but also in recognizing the words, their pronunciation and meaning.

Figure 1.2: Some similar looking Alphabets in Sindhi Language

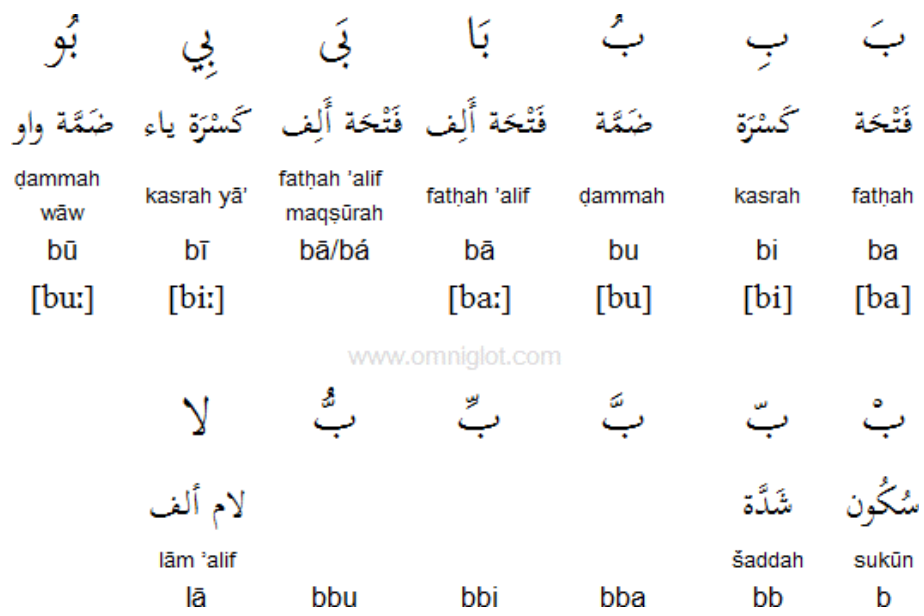

Figure 1.3: Various Diacritics and their pronunciation

As the task of Recognition is really hard, we will restrict ourselves to Arabic letter classification problem which in itself is a very challenging task. The structure of the rest of the report is as follows; Chapter 2 provides a literature review discussing various methods that have been studied for HWR and AHWR. Chapter 3 and Chapter 4 give an introduction to the theory and practical aspects of the method that we used. The results of our method are given in the Chapter 5. Chapter 6 concludes our report and the various resources used in the study and implementation of this project are mentioned in the last chapter.

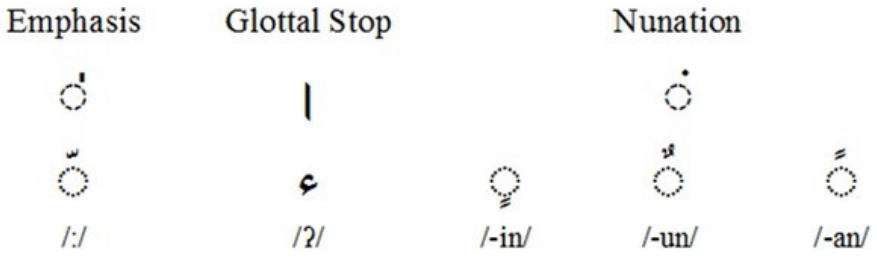

Figure 1.4: Nunation and other Diacritics and their pronunciation

Figure 1.5: Arabic text with and without diacritics

Chapter 2

Literature Review

As we discussed in the last section, because of its sheer importance, Handwriting Recognition has been one of the oldest and most studied tasks in Computer Vision. Since the problem of handwriting recognition is quite hard, we will restrict ourselves to handwritten character recognition in this project. This reduces our project to a supervised learning problem, that is, a machine learning problem where we are given a dataset of input features and the corresponding outputs (or classes or labels, if the outputs are finite, as in our case) and we are to find a relation between the two so that if we are provided features of an unseen input, we can predict its output accurately. A more general problem than this is object recognition. In this section, we will briefly discuss the evolution of computer vision techniques on the related problems.

The problem of Handwriting recognition has already been studied a lot as of today. Plamondon, and Sargur have given a comprehensive overview on research on handwriting recognition in [4]. Purohit and Chauhan provide a detailed analysis of handwritten character recognition in their paper [9]. Like English, the Arabic handwriting recognition has been worked on by many people as well. Al-Badr published a survey on Arabic optical text recognition in 1995 [6], Lorigo and Govindaraju did an extensive study on the offline Arabic handwriting recognition in 2006 [5]. A recent survey on Arabic handwriting recognition was published by Tagougui et al. [10] though it discusses only those techniques which are faster and work well online.

Hubel and Wiesel in 1959 carried out their famous experiments on a cat [25] to study the visual processing mechanism in mammals through Electrophysiology. They stuck electrodes in the back of the cat's brain, the region which contains primary visual cortex (see [25, Fig. 2.1]). They wanted to monitor the response of the visual cortex with various stimuli. They found out that the visual cortex showed higher response when the stimuli is moved in certain directions as compared to static ones. They deduced that this behavior of visual cortex is because of the oriented edges.

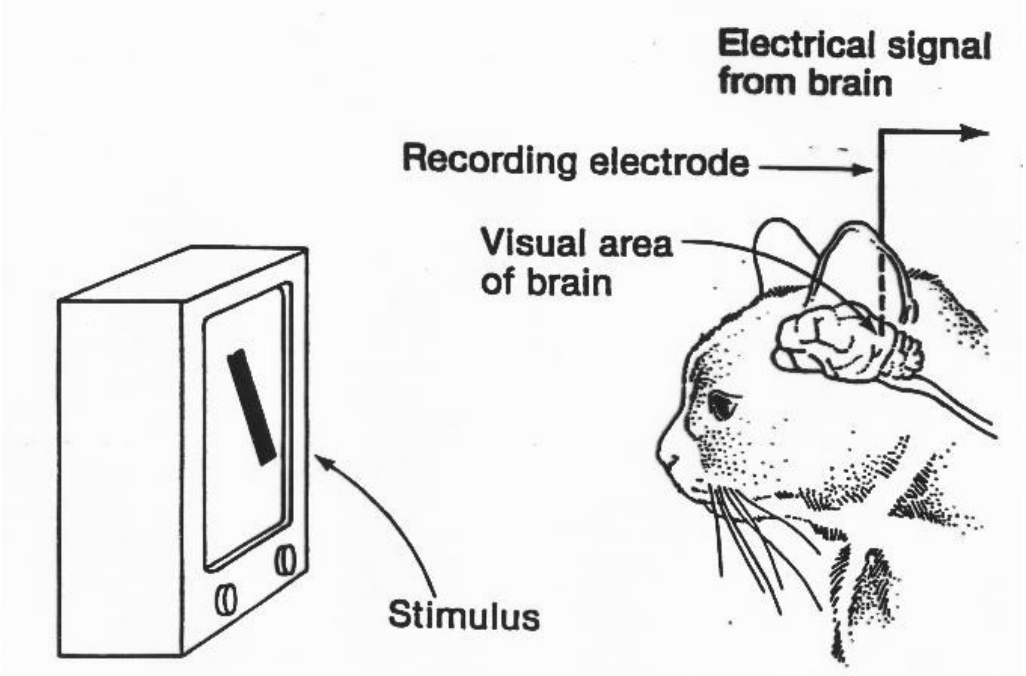

Figure 2.1: Hubel & Wiesel Cat Experiment

Before deep learning (learning from data through neural networks, we will talk about it in detail in section 3), the computer vision community focused more on detecting and extracting better and better features from the images which were then fed into some classifier. These features needed to be invariant or covariant to some transformations like scaling, rotation, translation, warping etc. because in that way they would produce similar features for similar objects and different ones for those which are different, or in other words decrease and increase the intra-class and inter-class covariance respectively. Simply speaking, features should be extracted in such a way that the change in the perspective, location, and orientation etc. of the object of one class in an image should not affect significantly, the ability of the classifier being used to predict its label correctly. One of the earliest known feature of an image is edge. Since the edges are robust to additive illumination changes they can be used to perform matching between images. Marr and Hildreth published a paper in 1980 on the theory of edge detection which is the title of the paper as well [13]. In their paper they discussed the first and second derivative operators as the edge detectors. The first derivative detects the intensity changes and thus the maximum of the first derivative which is above some threshold is an edge. The same thing can be detected using the second derivatives which would be zero at the maximum of the first derivatives. Since an image is a function of two variables we compute the gradient vectors and the Laplacians respectively. They also discussed some practical issues in these approaches like how these operators would behave to noise. One of the revolutionary papers in edge detection was [14] which made it

usable for realistic data. In this paper, John Canny, after proving that the best edge detector is the derivative of Gaussian, developed an algorithm which aimed at making the edges thin to one pixel length. The derivative of Gaussian is more powerful than simple derivative in a sense that it removes some noise while finding the edges. Canny edge detector is still used today for edge detection.

Harris and Stephens presented a feature detector called the Harris corner detector in 1988 [30, Fig. 2.2]. It was used in object tracking and gave very good results at the time. The corner points, which are important points of interest are detected using the second order moment matrix, the matrix which is formed by computing the outer product of the gradient of the image with itself. The matrix is smoothened with a Gaussian filter so that we can accumulate the neighboring information as well. It is also sometimes called the structure tensor as it gives some information about the structure of the image, that gradient alone can't. In order to distinguish whether we are standing on a corner point or an edge or a flat region, we compute the Eigen values of the matrix. The magnitude of an Eigen value is directly related to the amount of the intensity change or edge in the direction along the corresponding Eigen vector. So if both Eigen values are large, we have strong edges in the two orthogonal directions which means that we have a corner. These corner points serve as the interest points which can further be used for some other task like matching for object tracking in an image sequence.

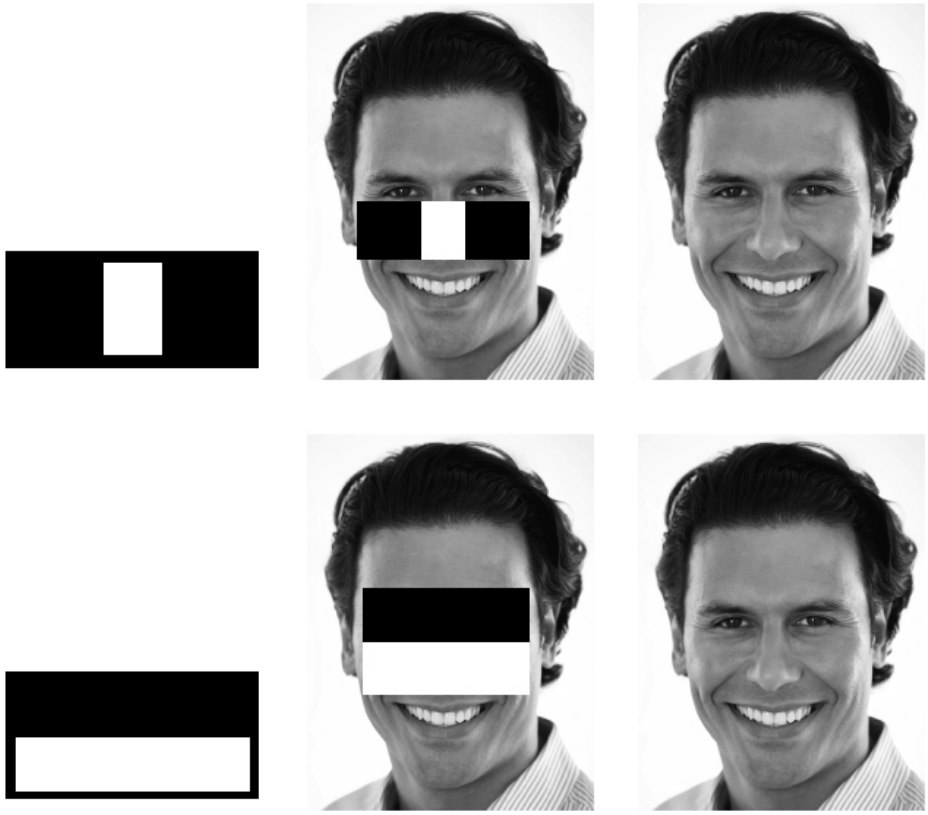

Figure 2.2: Haar-Like Features on Face Detection Application

Viola and Jones introduced a new method of object detection [16] mainly face detection using something which are today known as Haar-like features because of their similarity with the Haar wavelets. The approach used in their paper is similar to that of machine learning as it is trained on a dataset of positive and negative images. For example for faces, we have images of faces which are of positive class and those without faces belonging to negative class. In order to train a classifier some features are extracted from them. The way they did it was that they considered the convolution kernels similar to Haar basis functions, and compute the features for all possible sizes and locations of such kernels. The computations required were mere summations. The authors used this fact in their advantage and introduced the integral images in order to reduce kernel. So if we have the integral of a function, we can easily compute the area under its curve between any two points in the input domain. In a similar way, the integral images help to compute the summation of pixel values over a window in an efficient way. Lienhart and Maydt extended the Haar-like features to a general application of object detection in their paper [17].

A new type of feature extractor or descriptor, called the Histogram of Oriented Gradients, were first introduced by McConnell in 1986 in their patent [18]. They

weren't used in a significant way in computer vision until 2005 when Dalal and Triggs published their work on human (pedestrian, to be more specific) detection [19]. The idea was extended to further perform detection on the animals and traffic as well. We discuss the HOG's in detail in Chapter 5 where we combine this feature descriptor with the Linear and Gaussian SVM to perform the character recognition. HOG's show quite robustness to different transformations in the images. They are invariant to small rotations. As the name suggests, they are the histograms computed over a window size, of the gradients magnitude for different orientations. The main thing in HOGs is that unlike simple histograms, we, for a particular bin which is one of the many orientations of the gradient (the angle) possible, don't just increment by one, rather by the magnitude of the gradient.

Lowe in 1999, presented a new technique [20] of feature extraction called the Scale Invariant Feature Transform. The technique as can be observed from the name, is invariant of the scaling transformations on objects. The way Lowe achieved the scale invariance was that he generated the Gaussian Pyramid of an image (A quite redundant but very helpful representation of an image on multiple resolutions, obtained by repeatedly downsampling it) and he computed each resolution or octave at a scale (by smoothing it with a Gaussian filter of that scale). In order to see the motivation behind that, consider two arcs, one of radius ten times that of the other. The arc of smaller radius can easily by detected if we go for Harris corner detector but the larger one can't be because in this case the second Eigen value won't be large enough. The reason behind this is obviously that the bend in the arc is very large. In order to detect the bend we need to find a way to decrease the radius of the arc and fortunately we can do it by decreasing the resolution of the image. This phenomenon is depicted in [31, Fig. 2.3] in detail. Lowe approximated the Laplacian of Gaussian with Difference of Gaussian to speed up the process. The extrema of different feature detectors like Hessian and Second Order Moment Matrix were computed at all the scales and resolutions. These were then used to detect the key-points which were scale invariant. They were also made invariant to rotation by assigning some orientation to each key-point.

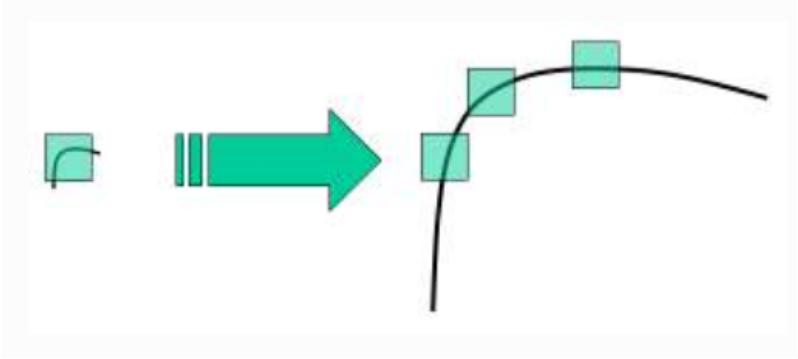

Figure 2.3: Change in the bend as the Object size increases

The problem with the SIFT was that they were quite robust to many transformations but very slow. Bay et al. in 2006 came up with the Speeded up Robust Features [21]. It is a speeded up version of the SIFT key-point descriptor, as apparent from its name. In order to compute difference of Gaussian, the Gaussian filtering was replaced by box filtering which can be computed efficiently using the integral images. The orientation assignment is carried out by computing the wavelet responses along the two axes. The rotation invariance is not desired for many applications, therefore the orientation assignment can be ignored which makes the process even faster.

In 1962, Hough introduced an innovative way of detecting lines in his patent [22]. The idea of Hough was very simple. $y = mx + c$ is the equation that represents a straight line. The equation can represent all the straight lines in a plane except for the vertical one. So that the set of all the points (x, y) that are on a straight line can be represented by a point or vector (a, b). Similarly a point (x, y) can lie on an infinite number of lines each represented by a vector (a, b). So in other words a line in XY domain is a point in AB domain whereas a point in XY domain is a point in AB domain. So if a set of N points (x_n, y_n) for $n = 1$ to N lie on a straight line (a_0, b_0), the corresponding lines of the points in AB domain will intersect each other at the point (a_0, b_0) giving us the analytic form of straight line. Note that the points are not even required to touch each other. They only needed be on the straight line for detection. Duda and Hart in 1972 extended improved the idea of Hough for straight lines (i.e. made it useful for vertical lines as well as efficient to compute) and applied it to circles and ellipses in images [23]. But it was Ballard in 1981 who not only generalized the Hough transform to arbitrary shapes but also made it popular in computer vision [24].

The first groundbreaking paper in the field of document recognition was published by Yann LeCun et al. [11]. The famous convolutional neural networks architecture introduced in the paper was called LeNet. Originally, LeNet was used for applications like OCR and character recognition and it proved very useful. It has a

very simple architecture comprising of six layers out of which two are convolution layers, each of them following a pooling layer and two are fully connected layers at the end of the network. The network worked really well on the problems like handwritten digit recognition as it aimed at learning the features which were to be extracted from the images instead of letting users decide. That is, it made the feature extraction process automatic.

In 2009, researchers from Princeton University presented a new database called the ImageNet [26] in CVPR. The dataset contains around 1.3 million images with one thousand different classes. The images were taken directly from Google and were distributed to thousands of people from around the world for annotation task. For employing that number of people, Amazon's services were acquired. In 2010, the first ImageNet Large Scale Visual Recognition Challenge (ILSVRC) was held. In this competition, research teams were invited to perform different visual recognition tasks and achieve as high test accuracy as possible.

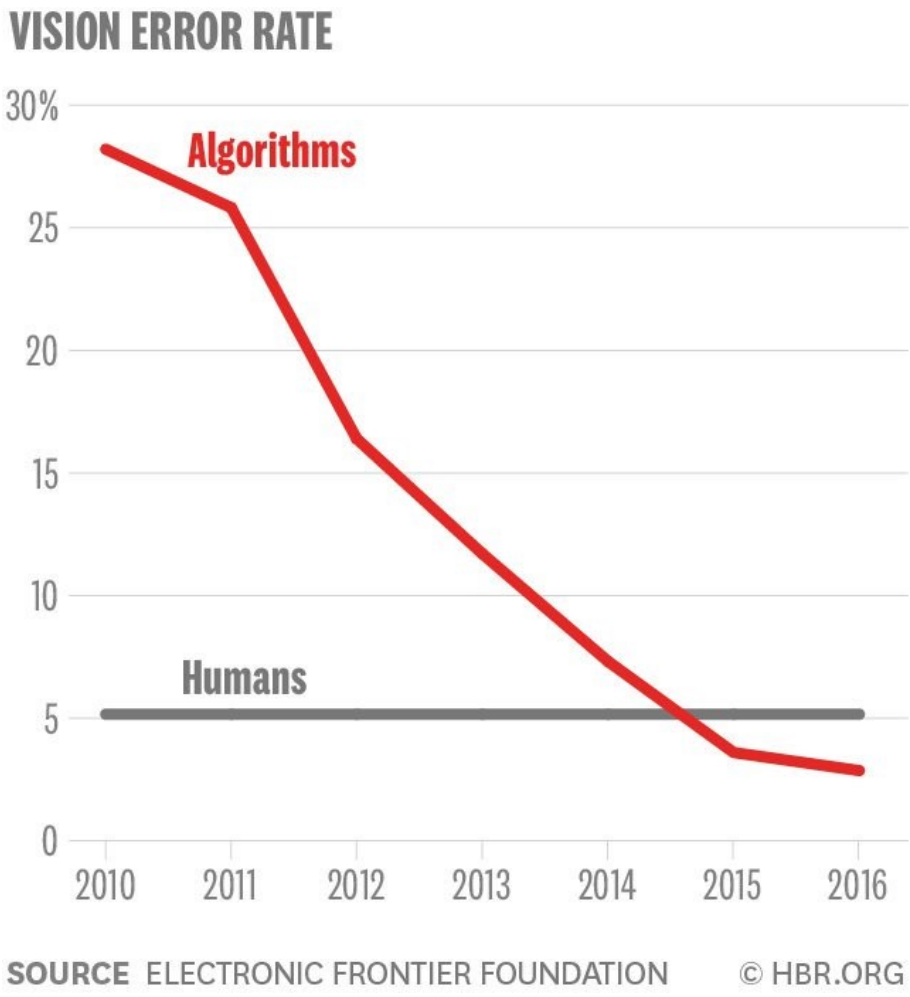

Figure 2.4: Image-net Results over time

From Late 90's to 2012, most of the object recognition was feature based. That

is, first the features of images were extracted depending on the application and those features were then fed into some classifiers for example Linear-SVM. We have already discussed some of these features. The feature extraction process was designed to cater for the fact that an object can undergo transformations like translation, rotation and scaling etc. when it is converted to digital form. The more robust the features are to these transformations the better the performance of the classifier would be. The 2010 and 2011 winners of ILSVRC competition used these approaches. In 2012, Krizhevsky et al. presented their revolutionary paper in NIPS [8] and also submitted for ILSVRC 2012 competition. They won the competition and significantly improved the results. Notice the sharp fall in the graph in [32, Fig. 2.4]. This result was so shocking that it changed the way researchers thought of computer vision. Krizhevsky et al. used a convolution neural network, now famously called AlexNet (a nine layer network). Although CNN's were around for more than two decades but after combining with GPU's they gave results other methods could not. The competitions held in the following years further improved the results, all of them used CNN's but with more layers and more features. Szegedy et al. won the ImageNet competition in 2014 [28]. Their network, called the GoogLeNet used filter kernels of different sizes in the same layer, thus increasing its width as compared to other networks. Simonyan and Zisserman's VGGNet [27] was the runner-up in ILSVRC 2014. They used smaller kernels throughout their network i.e. all the filters were of size 3 by 3. The depth of their network was 16 layers. The comparison of the AlexNet, GoogLeNet and VGGNet are shown in [33, Fig. 2.5].

In 2015, Kaiming et al. noted an interesting thing in CNN's (or in general, any neural network). They found out that as the layers of the CNN are increased, the training error starts to increase instead of decreasing. It was shocking given the fact that the new function space contained the old one. Therefore the training error should have either remained the same or have decreased. They showed that by setting the common layers of the new and old networks with same parameters and extra layers with identity parameters. The training error then became the almost the same for the two networks. The training error further increased when even more layers were added. They argued that this was because of the fact that the optimization problem at hand is not convex. The loss function has many local minima with respect to the parameters. The network with more number of layers is stuck at a bad local optima after training. In order to deal with that, they introduced residual learning i.e. they connected the first layer with not only second one but also with the third layer as well. Intuitively this would either give an advantage of having the second layer. If the presence of the second layer doesn't help that much then the information will be bypassed from first layer to third layer. They tried with 150 layers and reported a decrease in the training and test error. They were

the winners of ILSVRC 2015.

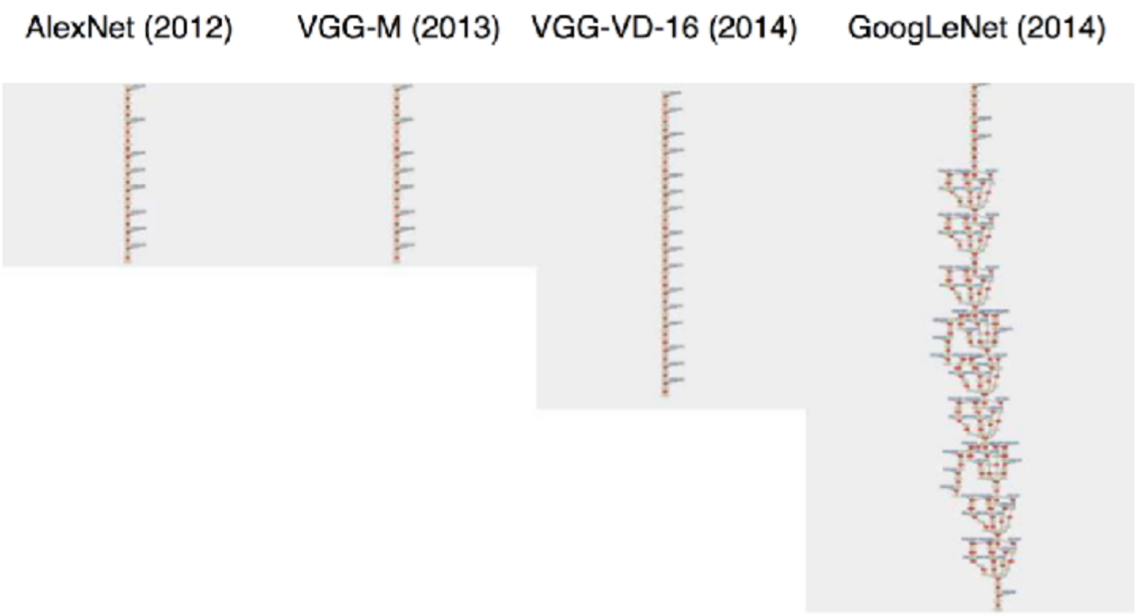

Figure 2.5: Comparison of the depth of Various Networks presented in ILSVRC Competition

Chapter 3

Theory

3.1 Classification Problem – an Overview

In this chapter we provide theoretical understanding of the concepts that we will use in our implementation, which are discussed in Chapter 4. The reader is advised to skip this chapter in case he already knows these things. We will refrain ourselves from going into intricate mathematical details for example how the loss function of support vector machines is derived and optimized. We will instead give the basic notion of how and why the Support Vector Machines (SVM) and the Neural Networks (NN) work amazingly well in classification.

3.2 Linear Support Vector Machines

We consider a binary classification problem on a data which is linearly separable that is, the two classes can be separated by a straight line called the decision boundary in classification to avoid under-fitting. To further simplify our problem, we don't consider the problem of over-fitting either, that is, we assume that our data is perfect and doesn't contain any outliers. An example of such data is shown in the [34, Fig. 3.1]. Our feature space can be of dimension higher than two. In that case, the decision boundary changes from a straight line to a hyperplane. But to keep things simple, we restrict ourselves to a two dimensional feature space. The data can be separated into two classes easily by an infinite number of decision boundaries or the straight lines in this case. Three such boundaries are drawn in Fig. 3.1. A question may arise, what is the optimal decision boundary? Different classification techniques have been proposed and all of the try to find the optimal boundary by minimizing some loss function. If we choose a different loss function, we will get a different boundary. This difference is due to the difference in dependence on the distribution of the data points from the two classes. Note that the labels assigned

to the classes in binary classification can be 0 and 1 or -1 and $+1$. The loss function and the function evaluated (we call it evaluation function in this report) then changes accordingly. In case of the labels $\{0, 1\}$, the evaluation function can act as probability because it belongs to the set $[0, 1]$. For labels belonging to the set $\{-1, 1\}$, the evaluation function lies between negative and positive infinity (the magnitude depend on the distance from the boundary and sign on the predicted class). We can also call them positive (which have label 1 or $+1$) and negative classes (which have label 0 or -1).

The Fisher's Linear Discriminant Analysis (LDA) is one of the earliest classification methods. It implicitly uses a squared loss function to find the boundary. The problem with the squared loss is that, the points which are very far away from the boundary have higher errors irrespective of the class they are put into. For example, in [34, Fig. 3.1], consider the data point closest to the origin, shown in blue. If we think for a moment, we can come easily to this conclusion that this point should have no effect on how the boundary should look like because it very far away from the boundary or the other class to be accurate. But a classifier like LDA tries to output a boundary which is both closer to this point and separates the two classes. The boundary given by the LDA for a dataset shown in the [34, Fig. 3.1] would therefore be horizontal to satisfy these two conditions, which will clearly not work well on the test set. The point mentioned above in some cases may not be an outlier of the data generating distribution and thus should not affect the classifier like that. Logistic Regression tries to solve this problem by using the logistic loss which is, simply put, the logarithm of the exponential function. This solves the problem because the log function decreases as the point moves away from the boundary and dies to zero as the point goes to infinity which was not the case in LDA. Thus the logistic loss defines a boundary which depends more on the points closer to the other class, thus making it robust against the farther points.

SVM's take one step further from the Logistic Regression. It only considers the points which are closer to the other class and tries to find a boundary which has maximum distance from these points. These points are called the support vectors for reasons that they support the decision boundary and thus give the method, its name. All the farther points have no impact whatsoever on the shape of the boundary. The loss function used by SVM method is called the hinge loss which is sort of an approximation of the logistic loss. Notice that the distance between the decision boundary and the support vectors is very important and is called margin. Since no training example exists on the other side of the margin, the one which includes the decision boundary, such a margin is called hard-margin.

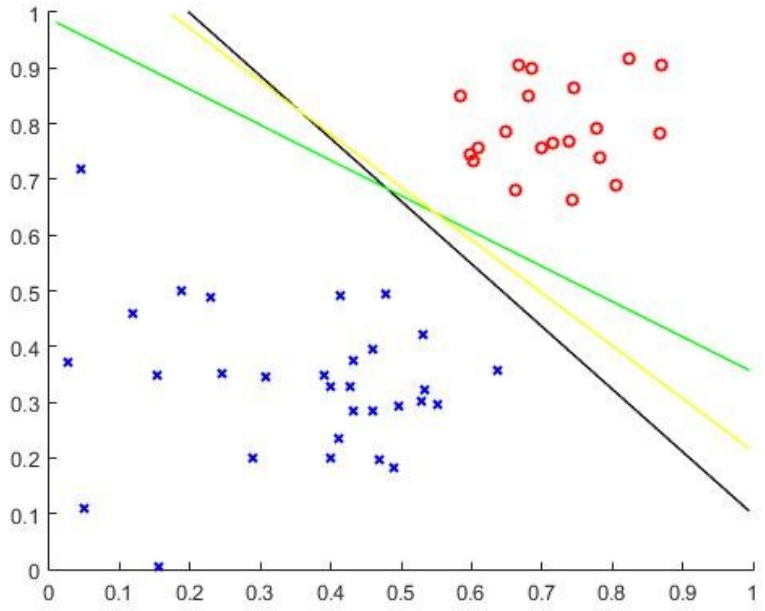

Figure 3.1: Example of Data which can be separated into two classes by a line

Before starting our discussion, we made several assumptions about the data. Now we will try to address all the problems that arise once those assumptions are violated.

3.3 Multi-class Classification

Let's first consider the multi-class classification problem. That is, the number of classes is, say, K which is greater than 2. This problem can be tackled in two ways.

The one-vs-all classification: In this method, we first take the data points of the first class and assign them a positive label and assign a negative label to all the data points of remaining classes. We train a Binary Classifier and find its necessary parameters which are used to find the evaluation function which is further used to predict the positive or negative label of the given points. We save these parameters. After this we take the 2^{nd} class and assign a positive label to its data points and a negative label to the rest of the points. We again solve the Binary classification problem and save the resulting parameters. We move to K through all the classes and find the parameters, in a similar fashion. Once we have all the parameters. We find the evaluation functions for all the classes. The class corresponding to the highest evaluation function is the predicted class of the one-vs-all multi-class classifier.

The one-vs-one classification: In this method, we take the data points of any two out of all K classes and fit a Binary Classifier to get the corresponding parameters.

We perform this binary classification on all the $K(K-1)/2$ possible combinations and store their parameters. In order to decide the class for a data point. We find the label corresponding to all $K(K-1)/2$ parameters. Since every class gets to "participate" in $(K-1)$ classifications, the label can be assigned to this class, at most, $(K-1)$ times. All we do is we count all such assignments for each class called the votes. The class which gets the highest number of votes is the predicted class of this method. The complexity of one-vs-all method is K times the complexity of the Binary classifier used whereas that of one-vs-one method is $K(K-1)/2$ times the complexity of the binary classifier. Evidently one-vs-one method is more expensive than its counterpart, but it also gives a more accurate label because of increased confidence. It also has an advantage that it doesn't explicitly require the knowledge of the evaluation functions, just the predicted binary labels.

3.4 Non-Linear Decision Boundaries

The real world data is not linearly separable, for example, see [34, Fig. 3.2], and is thus under-fitted by a linear classifier and since we made an assumption while discussing the SVM's that the decision boundary has to be linear, we can't use it on such data. In order to make it work on a linearly non-separable data or in other words introduce non-linearity in our classification, we transform the original feature space into a new space in which the data is linearly separable and a linear classifier can do its job. We will discuss two ways of feature transformation in this report as they both are used in the project. Another thing that we have to do to achieve non-linearity is to relax the hard-margin case to something called soft-margin. In this case there are training examples which lie on the other side of the margin though not so many if the classifier is working well.

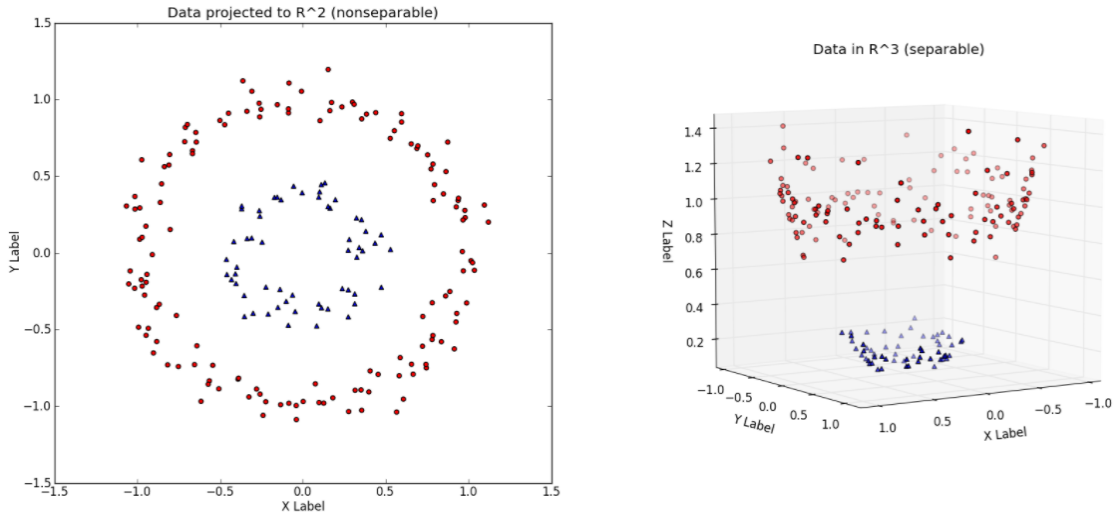

Figure 3.2: Feature Transformation Example Left Data in original space Right Data in transformed space

3.5 The Kernel Trick

Like we discussed, the most promising way of making a linear classifier work well on linearly non-separable would be by moving to a different space. One way of doing that is by introducing new features. We can apply non-linear functions on already existing features to get new features. For example consider the dataset on the left side in [34, Fig. 3.2]. It is clear that no straight line can perfectly separate the R^2-plane into two regions each containing data points from only class. And if we tried doing that, we would be underfitting our dataset. The valid decision boundary here would be a circle. We can use this knowledge to our advantage and add another feature $x_1^2 + x_2^2$, which will be higher for the class outside the circle and lower for the one inside. The resulting three features are plotted in R^3-space on the right in [34, Fig. 3.2]. Notice that now the two classes can be easily separated by a plane and thus the data is now linearly-separable. In the case where the decision boundary had been an ellipse, we would have added x_1^2 and x_2^2 as separate features. An in order to also cater for a tilted ellipse, we would need $x_1 x_2$ as a feature as well.

The example of [34, Fig. 3.2] was simple, that's we easily figured out what functions we should choose. Now consider the dataset show in [35, Fig. 3.3]. In this case we can't just separate the data by adding polynomial features of degree 2 or even 3. This requires us to go quite higher than that. Also an obvious question would be what functions we should use because the data is almost always never two or three dimensional and thus can't be plotted to get a better understanding for example in case of images, the dimension of original feature space can vary from thousands to a million. Adding more features would also be impractical for such a case because

if we only consider of adding polynomial features of degree d with n initially given features, the number of added features m would be $\binom{n+d}{d}$ and not all of them are valuable for example in [34, Fig. 3.2] adding x_1^2 and x_2^2 as separate features is not going to help since the decision boundary is a circle and the parameters of these two features would be almost of same size after training. A clue like this is not apparent in case of datasets with large number of features.

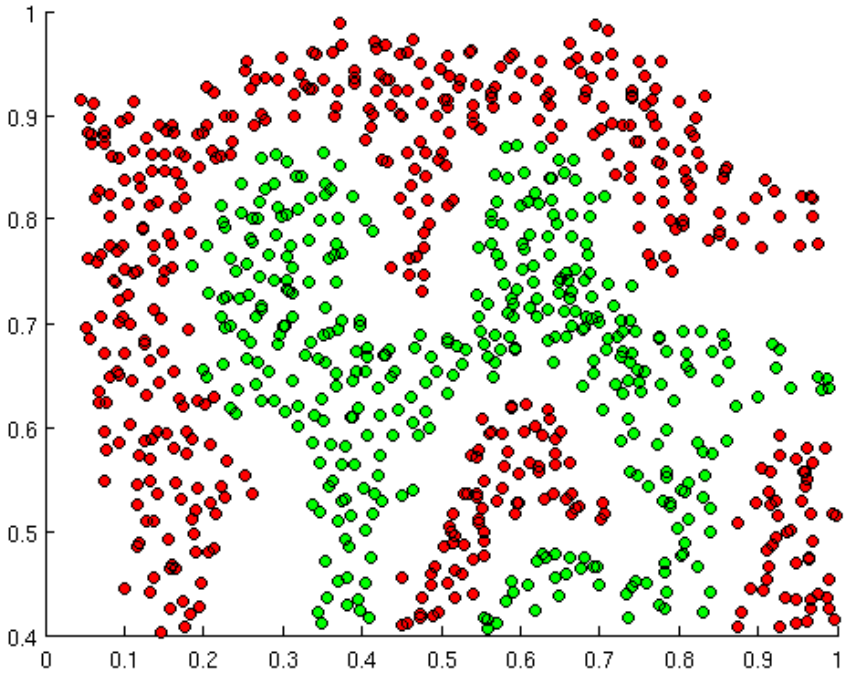

Figure 3.3: Linearly Non-separable data

This is where the Kernel Method – or more formally, the Kernel trick, comes to our rescue. Kernels are the most popular feature transformation functions used when working with the SVM's. We will not go into detail how kernels and SVM's, when combined, can provide the same functionality of extra features that we discussed above, without any overhead of memory and a small effect on execution time as it requires the knowledge of functional analysis and we already promised of refraining from deeper-digging of any topic. But it would also be wrong to not say anything about it. So we would consider the example of [34, Fig. 3.2] once again. This time we would deal with problem using the kernel trick. But firstly we will see what a kernel actually is. A kernel is a function which takes in two inputs of same space (in our case the inputs would be feature vectors) and spits out a scalar value. That scalar value can tell us about the similarity or dissimilarity between the two inputs. An example of kernel would be inner product normalized by the L2-norms of the

vectors. Such a kernel is called cosine similarity kernel. Other examples of kernels can be polynomial kernel, radial basis function (RBF) kernel also referred to as Gaussian kernel.

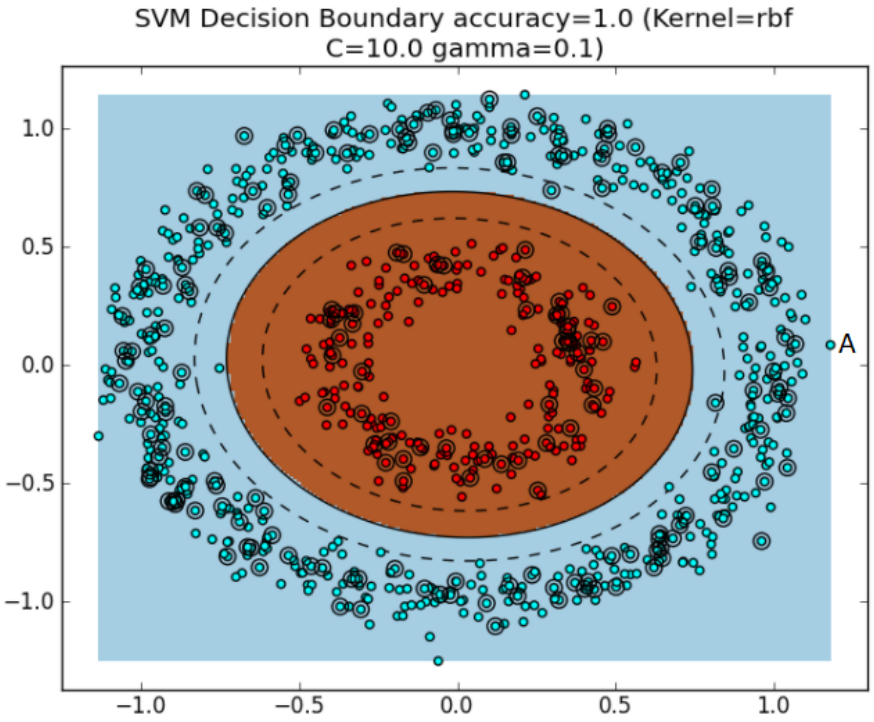

Figure 3.4: Classification using Gaussian Kernel and Linear SVM

Now coming back to our example, we will consider the solution of our problem using the RBF kernel. Consider [34, Fig. 3.4] where the data is classified using Linear SVM after applying RBF kernel. In order to generate new features using the kernel, some (fraction of total training examples) arbitrary points, let's call them landmarks, are selected from the training examples (they are shown as bigger circles in the [34, Fig. 3.4]). After that, for each training example, we compute the kernel of the example and a landmark. For each landmark we get a new feature and since the kernel is Gaussian the features produced are very different from the original ones. After that we apply linear SVM to the dataset with new features. And as it can be seen the classification has been performed with a very high accuracy. In order to understand why this scheme is working so well, we consider a data point, let's say the one on the far right and labelled as A. Now since the RBF kernel is sort of like a similarity measure. The closer the two points are the higher the value of their kernel is. So the feature values corresponding to the landmarks closer to this data point would be fairly high and those which are far away from it would be low. And since a Gaussian function is sort of a relaxation of the box function the landmarks which are far away from the point A would have very kernel value close to zero

whereas those which are closer to the data point would have kernel value close to one. And for each landmark, only those data points will have higher features which will be close to it so will sort of get an island of these points with high feature value, hence the name landmark. So if we have sufficiently large number of landmarks and small standard deviation of the Gaussian, we will be able to make our data linearly separable in the transformed domain.

3.6 Histogram of Oriented Gradients

Since we are dealing with the images, we can use the statistic of images to our advantage. Instead of generating new features, we can extract the features of the images, which are invariant or at least covariant to image transformations like scaling and rotations. In computer vision terminology, these are called the feature descriptors of the image because they are "describing" its features. Formally, the feature descriptor is a new representation of an image, which is formed by keeping only the useful information of the image and discarding the redundant one. Such a representation is similar for similarly looking images regardless of the difference in rotation, scale and translation or even the noise content in them. One such popular feature descriptor that we will use in our project is Histogram of Oriented Gradients (HOG).

In order to find the HOG features of an image, we first compute its gradient at each location. The way this is done is fairly simple. We first compute the x and y component of the gradient by convolving the image with the corresponding Sobel or Prewitt operators. As a result we get two images just like the original image. Now these are the Cartesian form of the gradient vector. We convert it into polar form and get magnitude and angle images. We now divide the image domain into 8 by 8 cells (this parameter can be changed) as shown in the left image of [36, Fig. 3.5]. If we had computed the histogram for the entire image, the localization would have been lost but the computation of histogram on these cells prevents that from happening thus giving us stronger features. Let's examine one of the cells as have been done in [36, Fig. 3.5]. Notice the gradient vectors in the zoomed cell at the center. The gradient vector is shown in polar form on the right. The angles are from 0 to 180 degrees (we can use the angles from 0 to 360 degrees as well but in this case the placement of the arrow on the vector is unimportant) and that is what the x axis of the histogram would be. We will discretize this axis to 9 bins (another parameter of HOG) so that the histogram is just a vector of length 9. We initialize all the entries of this vector to zero and for each pixel of the cell, we take the gradient magnitude and accumulate the previously stored value at the bin corresponding to the gradient angle. But notice that we have discretized our angle axis. So if our two consecutive bins are 40 and 60 and gradient magnitude is 80 at some location

with angle 45 degrees, we split the magnitude into two according to the distance of the angle from the two bins. In this case 60 will be added to the bin 40 and 20 will be added to the bin 60. In order to eliminate the effect of intensity changes we normalize the histograms by total gradient magnitude of the cell. After computing the histograms for each cell, we concatenate them to make a huge vector. This is the feature vector which can be used as an input to Linear or Gaussian SVM.

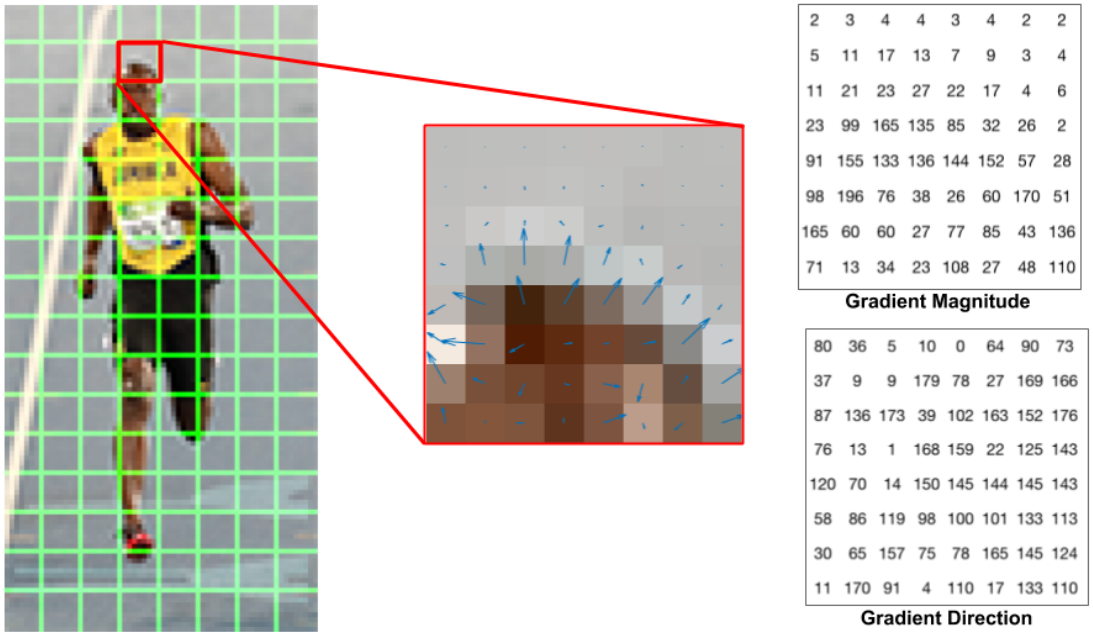

Figure 3.5: Demonstration of HOG computation procedure **Left** Original Image divided into cells **Center** an 8 by 8 cell, zoomed in gradients are shown as vectors with their tails at the center of the pixels **Right Up** gradient magnitude of the zoomed cell **Right Down** gradient angle of the zoomed cell

3.7 Cross Validation

The problem that we haven't looked into so far is overfitting. We have been constantly making the assumption that the data is perfectly separable, linearly or not. In reality this almost never happens. There is always some noise in the acquisition, transmission methods etc. that somehow corrupt the data. So whenever we are trying to train our model, we need to watch out for this noise. The reason why we should be cautious about it is demonstrated in the [37, Fig. 3.6]. The only way of separating the two classes perfectly in the figure would be by the green decision boundary but evidently that's not a good generalization. The red points present in the blue class and vice versa are the outliers. They are there perhaps because of the noise and a good classifier should ignore these points and look at the bigger

picture. Therefore the correct decision boundary would be the black one. This problem is called overfitting in the literature. The reason why this problem arises is not that hard to figure out. We used kernels and polynomial features to deal with the non-linearity of the decision boundary. For example, the degree of the polynomial features required for the example in [37, Fig. 3.6] would be at max 3 or 4. But if we increase the degree to let's say 20 then the classifier tries to separate the two classes perfectly because the curve of the decision boundary generated would have the degree of 20. Every kernel and new feature has a parameter like the degree which increases this non-linearity. The other thing that we not restricting the size of all the parameters called regularization in the literature. What we do is we try to minimize a weighted combination of the original loss function and the regularization term which is, in abstract manner, the measure of the complexity or non-linearity of our model. The weight used is called the regularization parameter, normally denoted by C in SVM and by λ in other techniques like logistic regression. The simplest way of measuring the complexity of the function is by finding the L2-norm of the parameter vector.

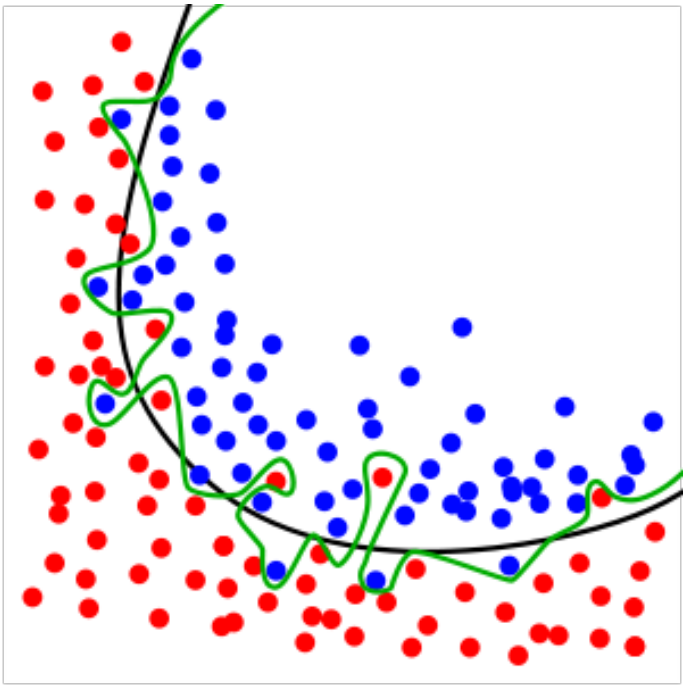

Figure 3.6: Example of Overfitting

But how do we find the optimal values of the degree and regularization parameter (called the hyper-parameters in literature for obvious reasons). We use a something called the cross-validation technique. In this technique, the original dataset is split into two (or three) datasets called the training and validation sets. The ratio of the split can be different but a good choice is 70:30. We try the different lambdas and

degrees in increasing order, train our model on the training set and evaluate it on both sets and find the training and validation errors. The hyper-parameters' values which minimize the validation error are the optimal ones.

We can have other problems in classification as well for example the class imbalance problem, which requires us to consider factors other than accuracy as well for example precision, recall, the ROC curve etc. but fortunately we don't encounter such a problem here so we will skip it.

3.8 Neural Networks

After the revolutionary paper by Alex et al. in 2012 [8], which introduces a new architecture of convolutional neural networks called AlexNet to perform object recognition task on ImageNet dataset, Deep Learning has taken over the computer vision. Since then, numerous other (and better) architectures have been proposed which we have already covered in the Chapter 2. We therefore will also try to perform the Arabic handwritten character recognition using the deep learning. But before we do so, we need to understand what it actually is.

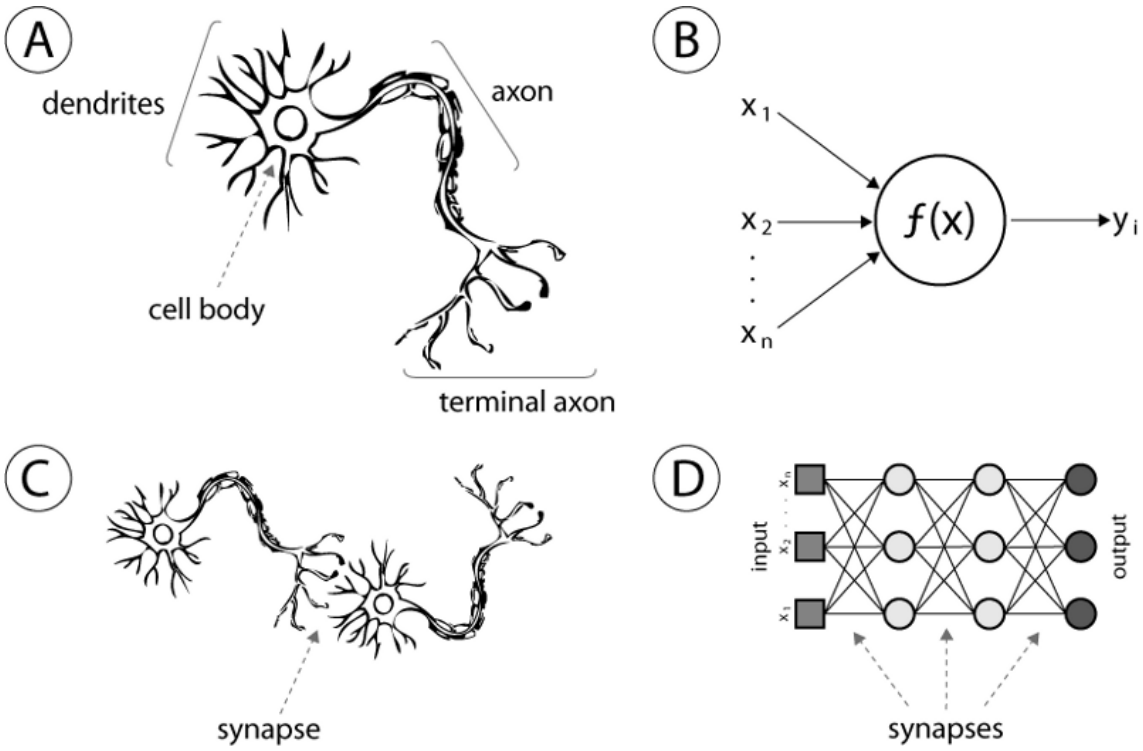

Figure 3.7: Intuition of Neural Networks

The motivation of neural networks come from the nervous system (Processing unit of living beings). The nervous system is a network of nerves which are made of what is called the neurons. A neuron is composed mainly of three components which

are; dendrites, cell body and axon, see [38, Fig. 3.7] (A & C). Dendrites take the information from the axons of the previous neurons and pass it to the cell body. The cell body takes the information and performs some processing on it and produces the output which is passed on to the next neuron through the axon.

Consider the image B in [38, Fig. 3.7]. In a simple system where we are given some features of an example (or point) and we want to predict its label, what we do is we apply some function on it to map it to the label space. This thing is analogous to a neuron. The input features of this system are like dendrites while the output is like the axon. The function (mostly a linear operation on input features followed by some activation function like ReLU or softmax to introduce non-linearity) is acting as the processing unit just like the cell body. Now imagine if we have a network of these neurons, for example the one in image D of [38, Fig. 3.7]. We connect the output of a neuron to one of the inputs of the next neuron. Note that we don't connect these neurons arbitrarily with any other neuron but instead of we have something called layers of neurons. All of the neurons in a layer have their outputs connected to those in the next layer. We have an input layer and an output layer. All the addition layers that are added in between these two layers are called hidden layers. A network without any hidden layer is called a shallow network while that with one or more is called a deep network. [39, Fig. 3.8] shows a deep neural network with five hidden layers shown in blue. The input layer has four neurons (four input features) and the output layer has three neurons (three classes).

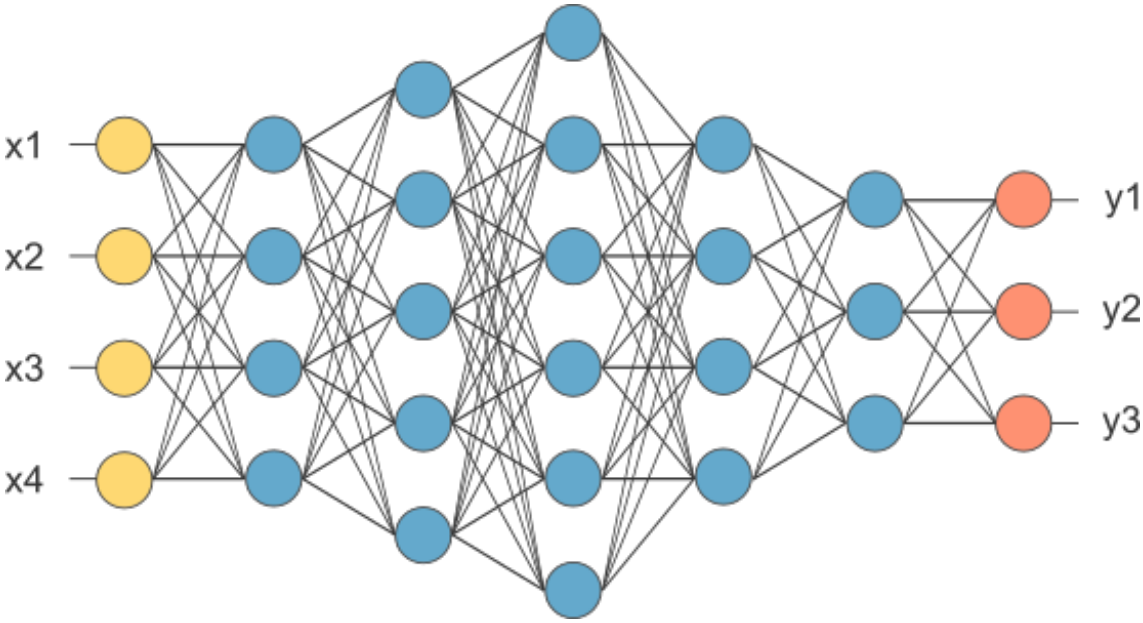

Figure 3.8: A neural network with five hidden layers applied on a three-class classification problem with four input features

Consider again the neuron in image B of [38, Fig. 3.7]. Now the output of the

neuron is the function of its inputs. The function as we already said is the linear combination with a bias of the inputs followed by some activation some functions. The weights of the linear combination and the bias are the parameters of the neural network. Every connection (shown by lines, see [39, Fig. 3.8]) between two neurons corresponds to a weight. Let's say that we know weights of all the connections. Therefore, given the input, we can pass the information through the neurons and compute the output of the network (the last layer). This procedure is called the forward propagation. But the problem is that we don't know the weights and therefore we have to learn them. For that we define a cost function that we want to minimize with respect to these weights. The most common cost function used in neural networks is cross-entropy loss. In order to find optimal weights, we need to compute the gradients of the loss with respect to the weights. The gradients weights of the connections between final two layers are easiest to compute, the rest of the gradients are computed using chain rule. Once we have the gradients of the weights between say layers l and $l+1$, we compute the gradients of those between layers $l-1$ and l using the chain rule. Notice that this time we are going backward and thus this process is called backward propagation (algorithm, sometimes shortened to BackProp).

Since there are many variants of the neural networks like CNN and RNN and we will examine the former here therefore we will call the original one, FCNN (fully-connected neural network) which is also sometimes referred to as Vanilla Neural Network.

3.8.1 Convolutional Neural Networks

Now that we have a little bit of understanding of the neural networks we can easily grasp how the convolutional neural networks (CNN) work. In images we talk about features whose number vary from a few thousand to millions. Therefore FCNN's cannot work well on these simply because the training process will get computationally exhaustive as we add more layers. If we consider the statistics of natural images we notice that every image pixel is sort of dependent on its neighboring pixel and this dependence decreases as we move farther away from that pixel. Therefore, instead of computing the values of neurons in the next layer from all of the neurons in the previous layer, we can use only some of them, defined by window or mask or kernel of some size (A hyper-parameter in CNN's). For example in [40, Fig. 3.9], the kernel is of size 3x3 and is shown as a gray colored 3x3 square on the blue square.

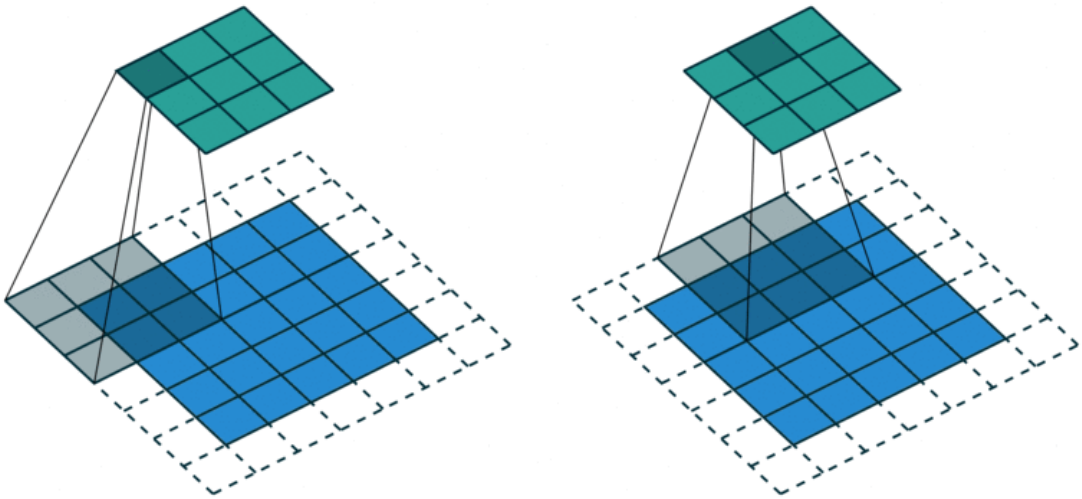

Figure 3.9: Convolution Operation with a kernel of size 3x3, stride 2 and padding 1

Also, like convolution, the kernel doesn't change weights as we apply it on the entire layer, unlike FCNN's. So, instead of changing the weights we keep the kernel or filter fixed for an entire layer and define more than one filters which thus generates more than one images or feature maps in the next layer, as shown in the Fig. 3.11. Therefore we don't just have a length and width for a layer (which is defined by the kernel size and some other things that we will see shortly) but also a depth or number of channels (which is defined by the number of filters). Therefore every filter automatically has a three dimensional size, the third corresponding to the depth of the previous layer. Also if the kernel doesn't fit properly as we move it across the image (which is not always necessary), we can pad the original image by zeros. For example in [40, Fig. 3.9], the padding is 1 for both axes.

Let's consider an example in order to appreciate the fact that we have indeed make our job easier by shifting from CNN's to FCNN's. Let's say that we have a network with two consecutive layers of size 1000 (a very small number when we talk about images). The number of full connections and thus the parameters to learn would be $1000 \times 1000 =$ one million. Now if we have convolution layers with 10×10 filter size (in practice the size can be even smaller like 3×3) and 100 filters (a considerable number) we will still have $100 \times 10 \times 10 =$ ten thousand parameters.

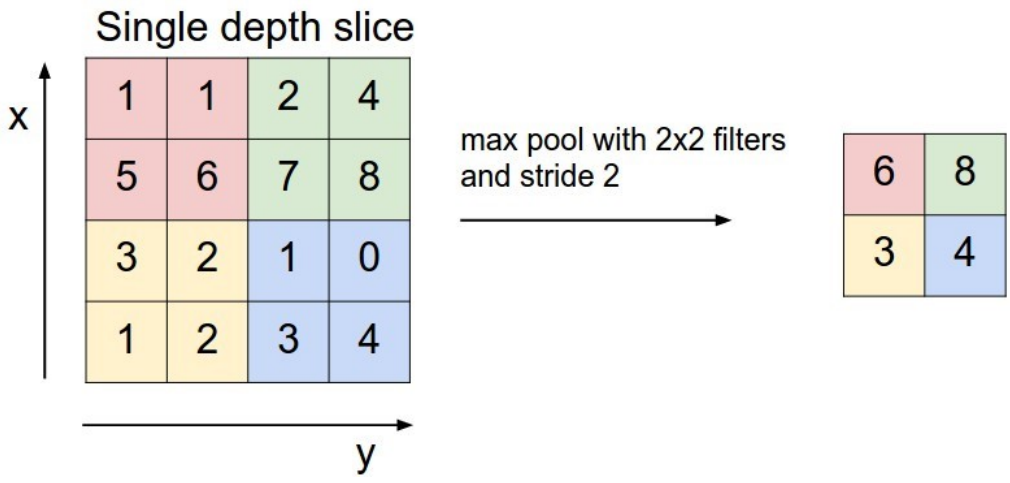

Figure 3.10: Max Pooling operation with kernel of size 2x2 and stride 2

Although we have reduced the number of parameters by introducing convolutions, the computations are still around the same. This can be solved by considering (again) the fact that neighboring pixels of images are correlated and therefore even if we down-sample the image we will still have sufficient information to perform classification. This can be achieved by two ways. One way is to use a step size greater than one during the convolution operation. That is, we move the convolution kernel by more than one pixels across the input image. This step is defined by a parameter named stride. For example in [40, Fig. 3.9], the kernel is being moved by 2 pixels two compute the next output, therefore the stride here is 2.

Another thing that can be done to reduce the size of image is the pooling operation. We consider the pixels in a window (normally a 2×2 window is considered big enough as it reduces the size of images by 75%) and take the maximum or average (max-pooling is more common than average-pooling nowadays) of all the pixel values in it. The pooling is performed after a convolution layer (so we consider it a separate layer, i.e. the pooling layer, even though there are no parameters to learn here). To decrease the size of the images, max-pooling is done with stride greater than two (see [41, Fig. 3.10]). A Pooling layer with stride 2 follows a convolution layer until the size of the image has been dropped considerably. Unlike convolution, pooling is done only across the spatial domain for each channel. Therefore the depth of the image remains the same after pooling.

Convolution layers are normally used to learn important features in the images. After sufficient number of convolution layers, the image, which is now very small in size, is stretched and passed through an FCNN to perform classification. Therefore we can say that a CNN acts as an automatic feature extractor (like HOG's), see [41, Fig. 3.11].

The more layers we add to our network (both FCNN and CNN) the better

performance we get on the training set but the training becomes more difficult because the loss function which is non-convex, become more complicated in the parameters.

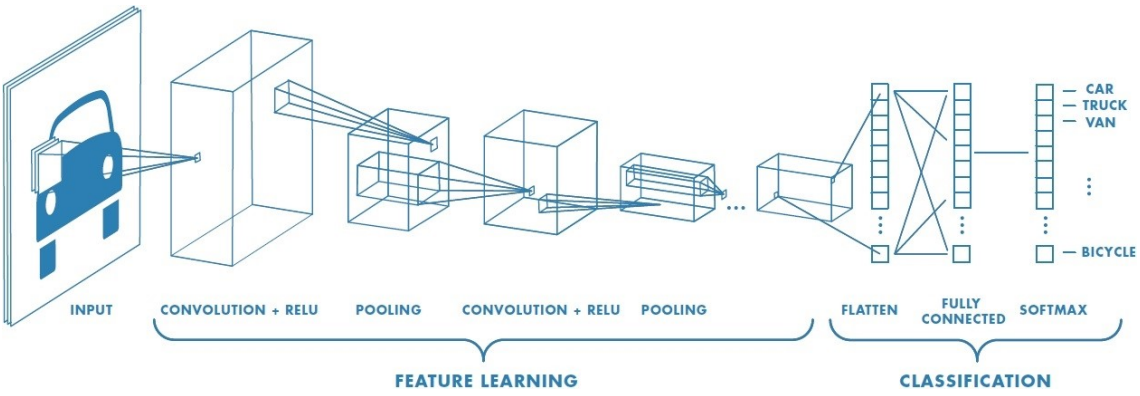

Figure 3.11: A Convolutional Neural Network

Apart from the L2 regularization, there is another way of tackling the problem of over-fitting in neural networks, called the dropout method. After a particular layer, a dropout layer is inserted which at every iteration of training drops (or turns them into zero) randomly, a fraction p (called the dropout rate) of the outputs from the previous layer. As the dropout increases, it forces the neural network to memorize the training data and therefore pushes it towards generalization.

Chapter 4

Methodology

From the knowledge that we have now, we try to build a multi-class classification model to predict the Arabic Letters from the given dataset, which is the goal of this project (Fig. 4.1). The dataset that we used was collected by El-Sawy et al. [7] for their Handwriting character recognition task. It comprises of 13,440 training examples and 3,360 test examples. How this data was collected and other details are provided in their paper.

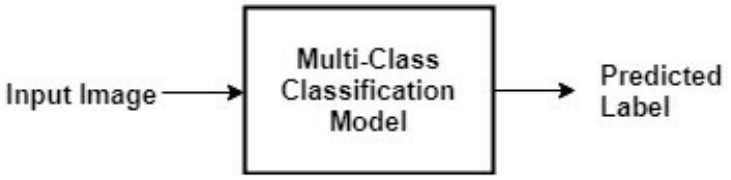

Figure 4.1: Our desired task, or in other words, the subject matter of this project

Instead of making our model complicated, we start off with a simple model. We try to perform the Arabic character recognition using a Linear SVM. We denote this algorithm with A1. The block diagram of this method is shown in Fig. 4.2.

Figure 4.2: Linear SVM (A1)

We then try to train the linear SVM after passing the image through RBF Kernel. In Fig. 4.3, we have placed a block of feature transformation just before the classifier of Fig. 4.2. We will refer to this method as A2 in the future.

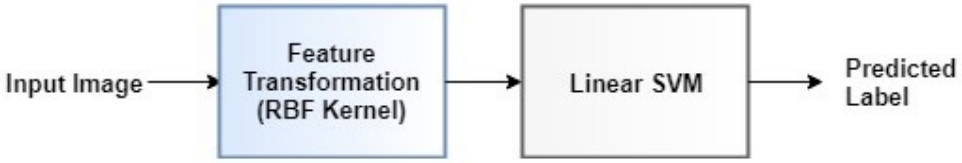

Figure 4.3: Linear SVM with Gaussian Kernel (A2)

Now, instead of using a Kernel, we extract the features from the input image using HOG feature descriptor. See Fig. 4.4 where we insert a block of Feature Extractor before the classifier of Fig. 4.2 in order to achieve our task. We call this method A3.

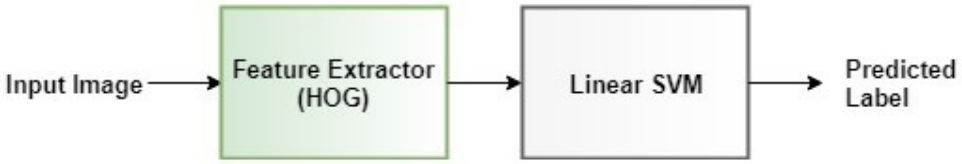

Figure 4.4: HOG feature extractor followed by Linear SVM (A3)

Finally we combine the two methods A2 and A3 to get the new method A4 i.e. we first pass the input image through the block of Feature Extractor, whose output is passed through the Feature transformation block which is fed into the Linear SVM classifier, see Fig. 4.5.

Figure 4.5: HOG feature extractor followed by Linear SVM with Gaussian Kernel (A4)

Now we try Deep Learning to predict the characters from handwritten images. First of all we use a fully-connected neural network (FCNN-1) as a classifier to achieve it. The architecture of FCNN1 is depicted in detail in Fig. 4.10. The image is given as it is to the classifier. This method is called A5 and is demonstrated in Fig. 4.5.

Figure 4.6: Fully-Connected Neural Network Classifier FCNN1 (A5)

Now, to make the model of A5 slightly more complicated, we insert the HOG feature extractor before FCNN1 (see Fig. 4.7). We call this method, A6.

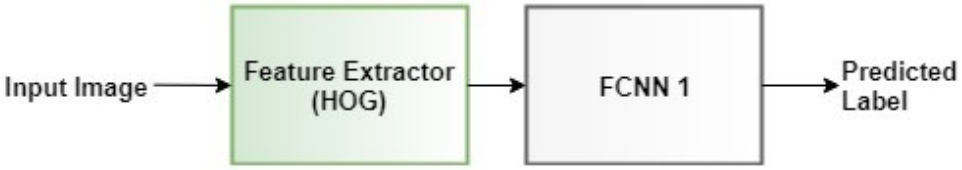

Figure 4.7: HOG feature extractor followed by Fully-Connected Network FCNN1 (A6)

The method A7 and A8 uses a CNN's as feature extractor and a fully-connected neural network FCNN2 as a classifier, see Fig. 4.8 and Fig. 4.9. The architectures of CNN1, CNN2 and FCNN2 are shown in Fig. 4.11, Fig. 4.12 and Fig. 4.13.

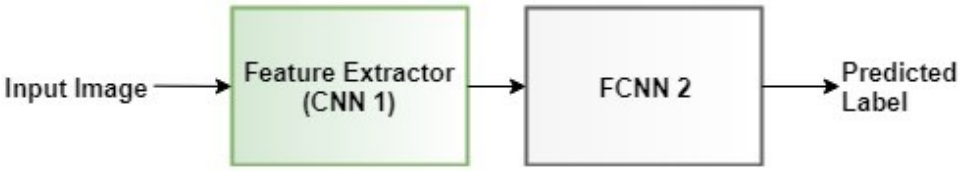

Figure 4.8: CNN 1 Feature Extractor followed by Fully-Connected Network FCNN2 (A7)

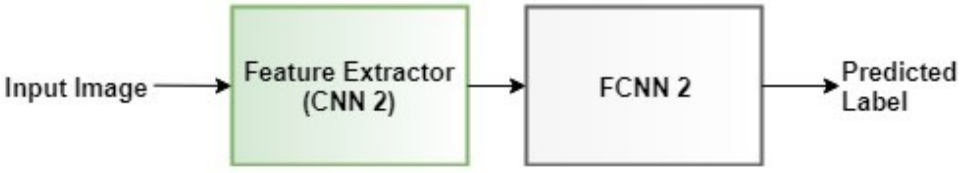

Figure 4.9: CNN 2 Feature Extractor followed by Fully-Connected Network FCNN2 (A8)

Once we have decided which model to use, we try to find the best possible values of the hyper-parameter which performs well on both training and test set.

The hyper-parameters in our control are C (the regularization parameter of Linear SVM) and γ (the standard deviation of the RBF kernel). The parameters of HOG which are cell size and number of bins are fixed to 8 by 8 and 8 respectively because we found out from the experiments that 8 by 8 is the best possible cell size which captures the edges information with less number of features thus not affecting the computation cost as opposed to 6 by 6 and 4 by 4 cell sizes. The bins also have a similar effect on the total number of features given by the HOG. A total of 8 bins capture all the angles of multiple, $\pi/8$, in the interval $[0, \pi)$. In order to make our model perform well on test set as well, or simply put, to generalize our model, we use cross-validation.

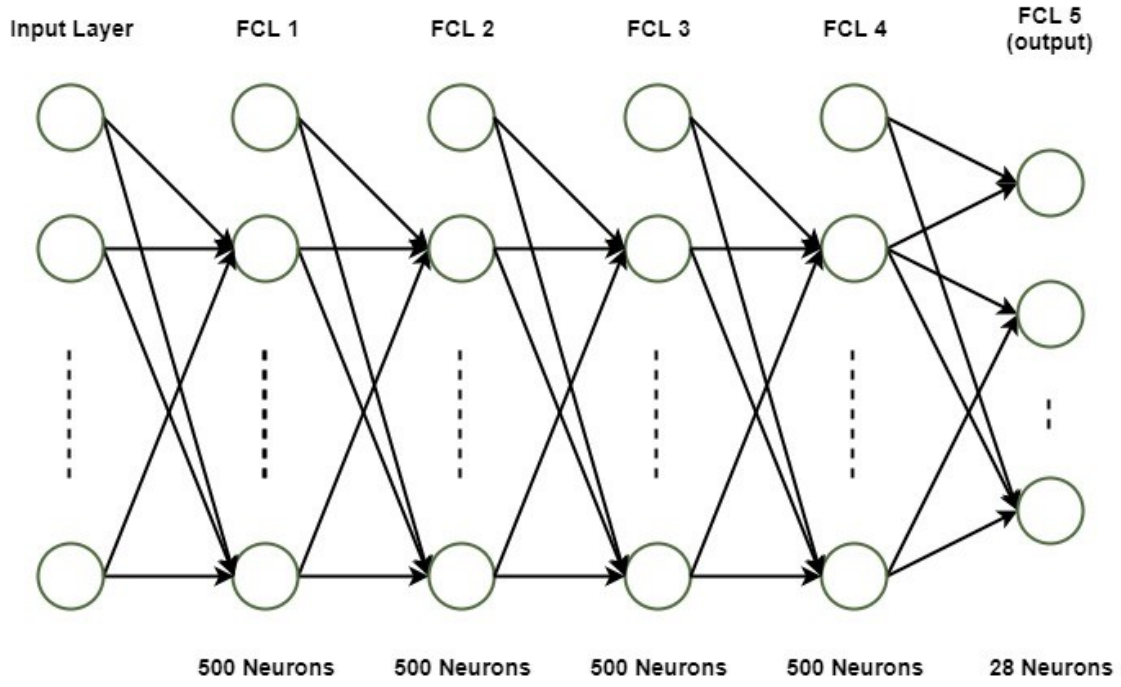

Figure 4.10: Architecture of FCNN1

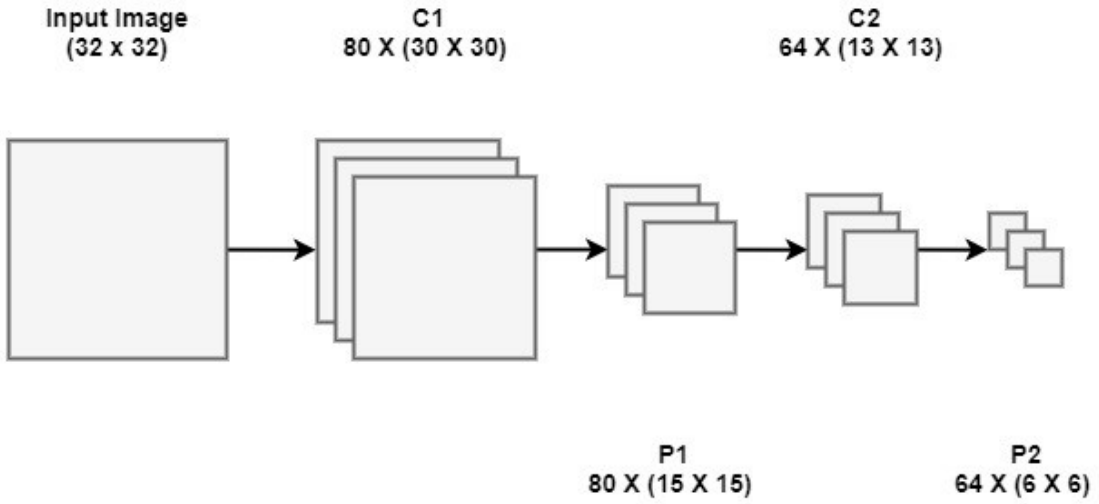

Figure 4.11: Architecture of CNN1

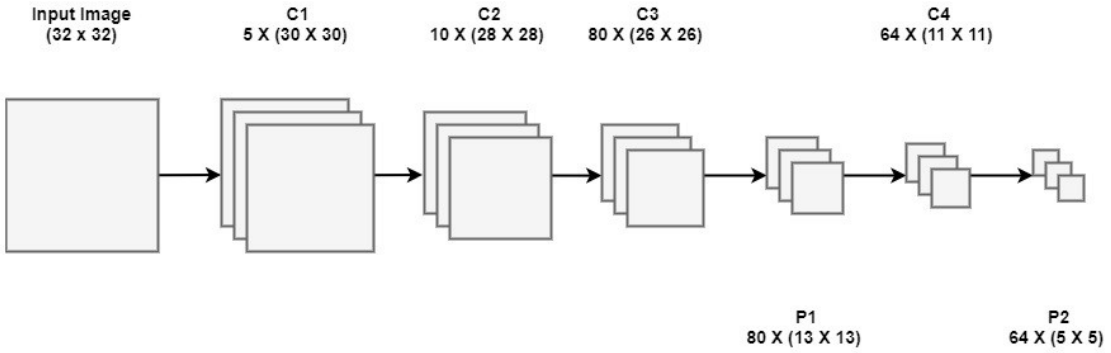

Figure 4.12: Architecture of CNN2

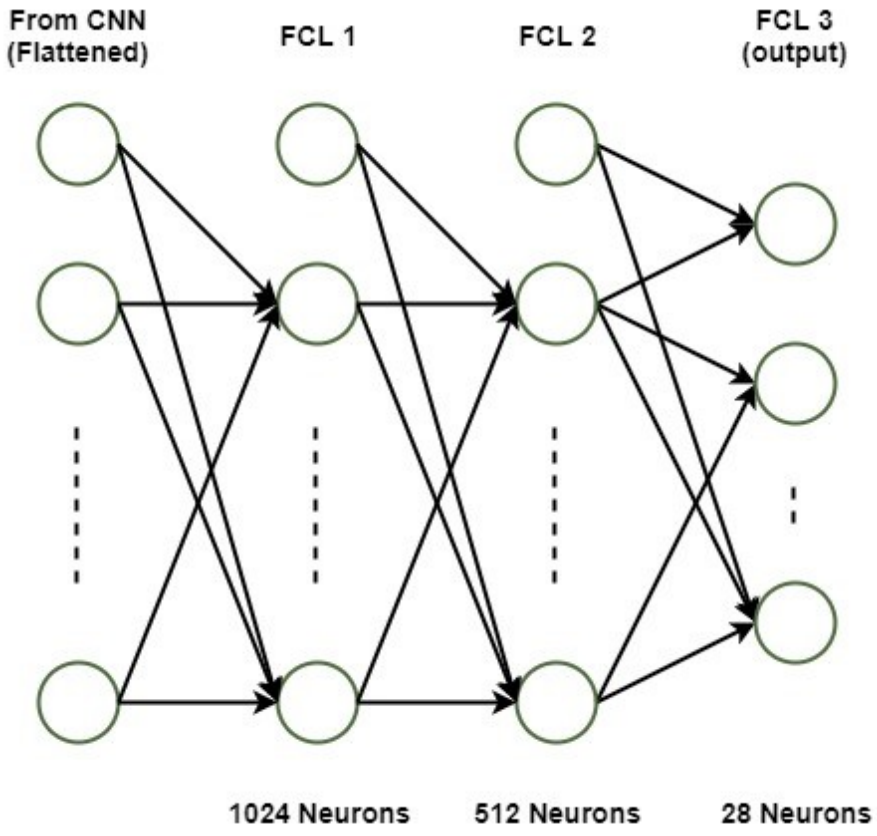

Figure 4.13: Architecture of FCNN2

In methods A5 and A6 we used four layered FCNN with each having 500 neurons. We experimented around with these parameters and came up with the conclusion that making the classifier any more complicated than this didn't have much effect on it. In some cases it even made it worse.

The method A7 which uses CNN's is as it is as the one proposed in [7]. We did some tweeking in it to make a more complicated feature learner (A8) and managed to increase the test accuracy.

Method	**Feature Extractor**	**Kernel**	**Classifier**
A1	None	None	Linear-SVM
A2	None	Gaussian	Linear-SVM
A3	HOG	None	Linear-SVM
A4	HOG	Gaussian	Linear-SVM
A5	None	None	FCNN
A6	HOG	None	FCNN
A7	CNN	None	FCNN
A8	CNN	None	FCNN

Table 4.1: Different methods used for classification

We split the original training set further into two parts by a ratio of 70 to 30, former being the training set while the latter being the validation set in cross-validation. We don't shuffle the original training set because it improves the validation accuracy for all the hyper-parameters and doesn't change the optimal hyper-parameters values. We used the coarse and fine tuning approach here. What we did was for all hyper-parameters (C and γ) we start with 10^{-10} and increase their values by multiplying with a factor of 100 and stop at 1. We the find the best values of hyper-parameters by selecting those which minimize the validation error. This is called the coarse tuning. In order to further improve these values, we try to minimize the validation error around it. Let's say that the optimal value that we got from coarse tuning is 10^{-X}, what we do now is we start with the value of $10^{-X-0.5}$ and move to $10^{-X+0.5}$ by multiplying with the factor of $10^{-0.25}$. These values can be changed but we got satisfying results with these as would be evident from the training and validation accuracies plots in the next chapter.

We insert a dropout layer after every fully-connected layer in methods A5, A6, A7 and A8. We keep the dropout rates fixed for each layer in a model. To perform cross-validation we don't use the coarse and fine tuning here. We simply vary the dropout from 0.1 to 0.65 with a step-size of 0.05 and select the one with highest validation accuracy.

Once we have found the optimal values of C and/or γ or p, we train the model using these values on the original training set of 13,440 examples and then evaluate the model on the test examples. This may seem unnecessary but the test accuracy of the methods was improved by as much as 5% because of this.

The programming language used was MATLAB for. Classification through SVM was achieved through the famous library LIBSVM which is written in C to perform at higher speed. Apart from LIBSVM, MATLAB's image processing toolbox was mainly used to compute HOG features and generate results in the next chapter. All of these computations were carried out on a CPU. The time taken by cross-validation of the methods A1 and A3 was 25 to 30 minutes. The methods A2 and A4 on the other hand took over 3 and 4 hours respectively to run. This huge difference is because of the fact that in A2 and A4 we have to perform validation across two parameters instead of one. Also, the RBF kernel makes an otherwise efficient SVM, computationally expensive.

For Deep Learning, we used Python and TensorFlow and since all the training operations were carried out on GPU's the maximum training time was less than 10 minutes.

Chapter 5

Results

We apply the methods that we discussed in Chapter 4 and present the results in this chapter. Table 5.1 shows the optimal values of all the hyper-parameters learned through cross-validation. We have already discussed that we didn't change the parameter values of HOG descriptor as they were performing pretty well with reasonable computational cost.

Method	Cell Size	Hisogram bins	C	Gamma	P	Training Accuracy	Test Accuracy
A1	-	-	10^{-6}	-	-	65.03%	50.24%
A2	-	-	3.16	5.62×10^{-7}	-	99.13%	72.38%
A3	8×8	8	1	-	-	92.30%	86.07%
A4	8×8	8	3.16	1	-	99.99%	89.02%
A5	-	-	-	-	0.45	99.99%	73.54%
A6	8×8	8	-	-	0.65	99.47%	90.65%
A7	-	-	-	-	0.3	99.56%	91.40%
A8	-	-	-	-	0.2	99.46%	93.27%

Table 5.1: Optimal Values of Hyper-parameters of all the methods shown in Table 4.1 found through Cross-Validation and Training and Test accuracies corresponding to those values

Fig. 5.2 through Fig. 5.11 shows the training and validation accuracies against different parameter values. Notice that Fig. 5.2, Fig. 5.5, Fig. 5.8, Fig. 5.9, Fig. 5.10 and Fig. 5.11 are two dimensional plots of training and validation accuracies. This is due to the fact that in the methods A1, A3, A5, A6, A7 and A8 we only have one parameter, that is, C or p. In the methods A2 and A4 we have 2 parameters, namely, C and γ, therefore either we need to generate a three dimensional plot or we can use heat maps in order to demonstrate the change in training and validation accuracies with the given parameter values. In our case, we preferred the latter option because it gives a better idea of the parameter values as is evident from the figures Fig. 5.3, Fig. 5.4, Fig. 5.6 and Fig. 5.7.

The color-maps used to depict different values in the heat maps is shown in

Fig. 5.1. The blue color represents lowest possible value. On the other hand, red is for highest achievable value in the heat map. For example in heat maps of accuracies, the highest and the lowest possible values are 100% and 0% and thus these would be represented by red and blue respectively, in case they appear in the heat map. As the value increases from lowest to highest, the color of corresponding location changes from red to blue through the spectrum shown in the color-map.

Figure 5.1: Color-map used for all the heat maps

Table 5.1 also shows the comparison of the training and test accuracy between all the methods after cross-validation, i.e. with optimal values of C and/or γ or p. Both the training and test accuracies increase as we move from a simpler to more complex model.

As discussed in Chapter 4, linear SVM under-fits the complex data and thus is not very good in generalization. Using an RFB kernel makes the performance of the classification model better than before but it is still not good enough. The model works very well on the training set though. This may have been because of the fact that the RBF kernel fails to capture the important differences between the classes. This is where we bring in the HOG feature descriptor. HOG is not scale or rotation invariant but it still is very good in catching the differences in the shapes. That explains the high test accuracy values for methods A3 and A4.

The results of method A5 show that even though the neural network classifier is very strong but it still fails to learn the features of the images in a better way. The accuracies of methods; A6, which uses HOG feature extractor and A7 and A8, which use CNN's are certainly high. The method A8 gives the highest test accuracy. We notice that all the deep learning methods (apart from A5), beat all the SVM methods in the performance on test data.

Fig. 5.12 through Fig. 5.19 show the confusion matrices in the form of heat-maps, evaluated on test set, of the methods A1 through A8 respectively. A particular entry with row r and column c of such a matrix represent that this many number of examples actually belonged to the class r but were predicted as c. So ideally, A confusion matrix should be a diagonal matrix with each diagonal element corresponding to the number of examples for the corresponding class (i.e. in this case, the image should be all dark-red along the diagonal and all dark-blue elsewhere). As a confusion matrix deviates from being diagonal, the classifier gets worse and worse. Notice that the confusion matrices for the methods A3, A4, A6, A7 and A8 are closest to being diagonal so in a way they are doing a good job at predicting the right character. The classes which are far away from dark-red are mainly, baa,

taa, thaa and noon and jeem, Haa, khaa, ayen and ghayen etc. This is because of the fact that they are very similar and thus have very low inter-class variance. One thing that should be noticed that the method A6 performs very poorly on taa and qaaf and therefore should not be preferred over A3 and A4 even though it has higher accuracy than the two.

Fig. 5.20 through Fig. 5.28 shows some examples of the working of all the methods on several cases of nine different letters. Top row of images in each figure represents the input image and the remaining rows (2 to 9) of images show the label predicted by the classifiers A1 to A8 respectively. Notice when one letter is written wrongly in such a way that it gets close to another one, some classifiers still can tell the difference. For example, consider the third example from the right in Fig. 5.23. The original letter was ayen, but almost all the classifiers have it confused with Taa or Dhaa because of the shape of the letter. The method A3 and A4 however got it correct. This shows how hard the task of Arabic character recognition is.

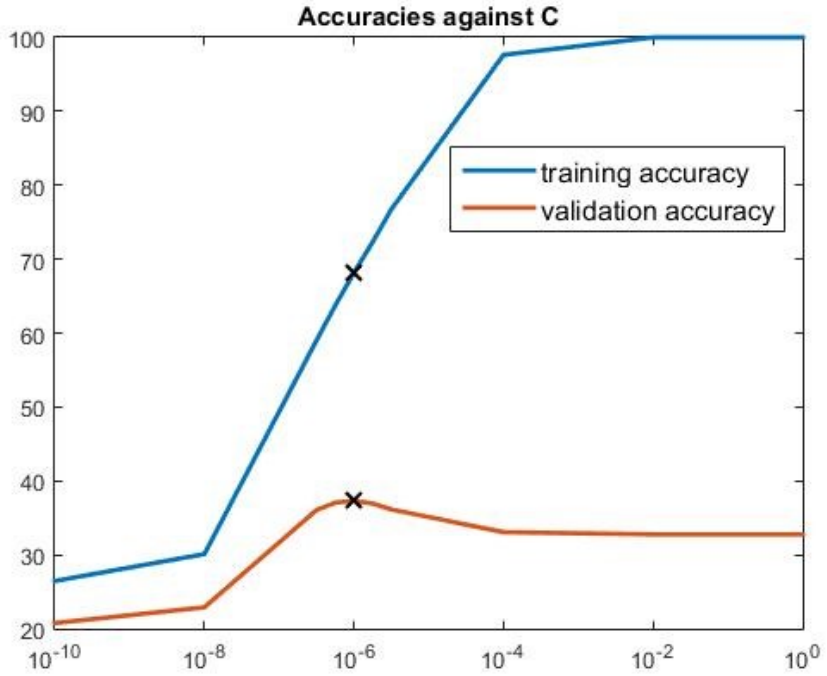

Figure 5.2: Plot of **training and validation** accuracies against the hyper-parameter C. By A1 Method.

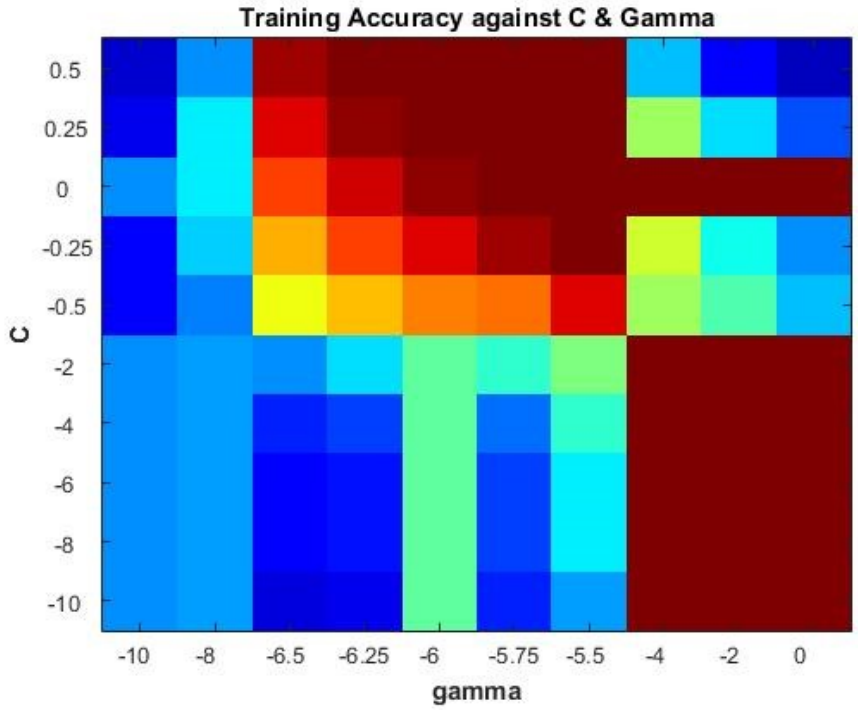

Figure 5.3: Heat map of **training** accuracies against the hyper-parameter C & gamma. By A2 Method.

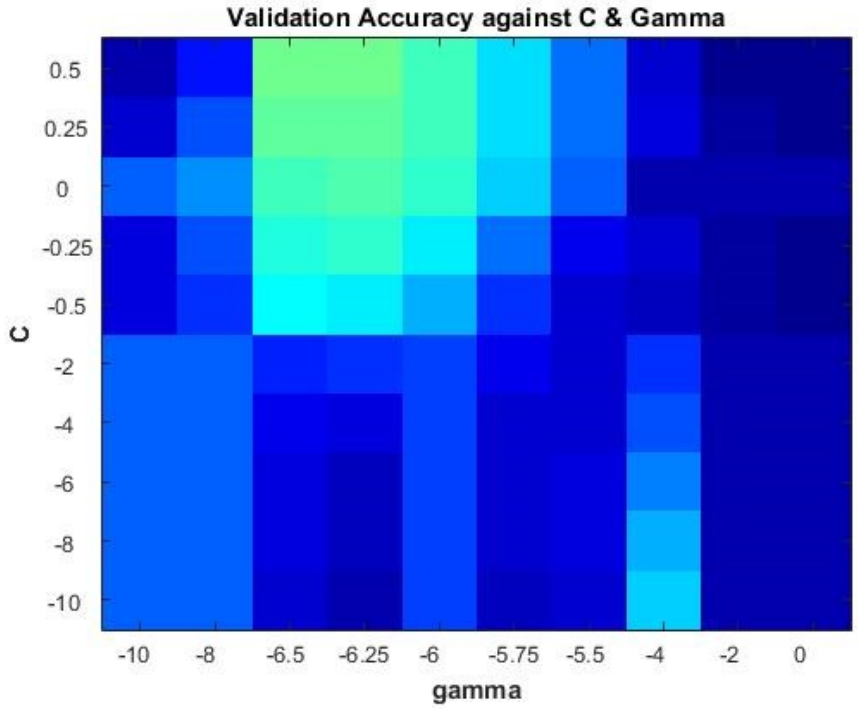

Figure 5.4: Heat map of **validation** accuracies against the hyper-parameters C & gamma. By A2 Method.

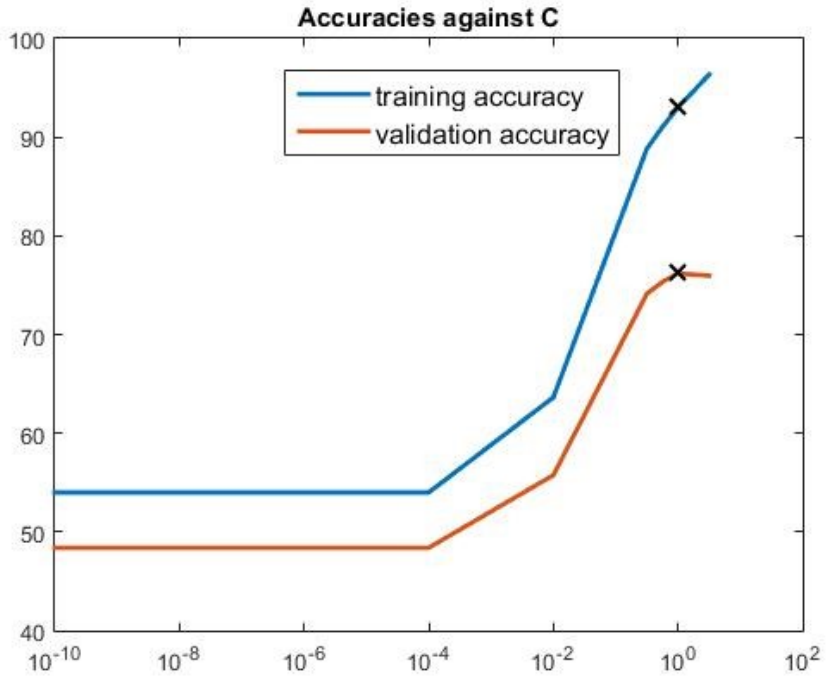

Figure 5.5: Plot of **training and validation** accuracies against the hyper-parameter C. By A3 Method

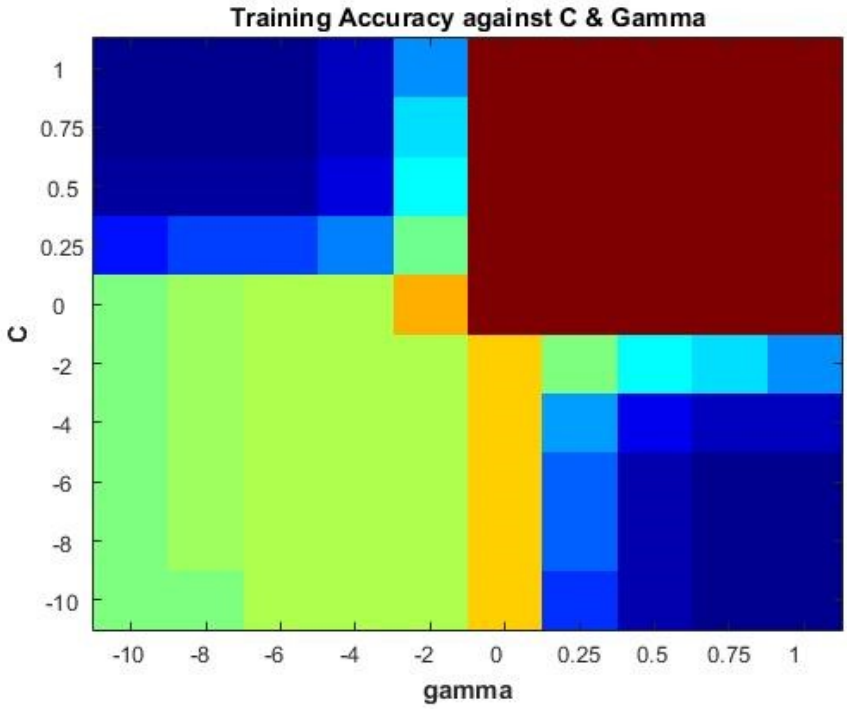

Figure 5.6: Heat map of **training** accuracies against the hyper-parameters C & gamma. By A4 Method.

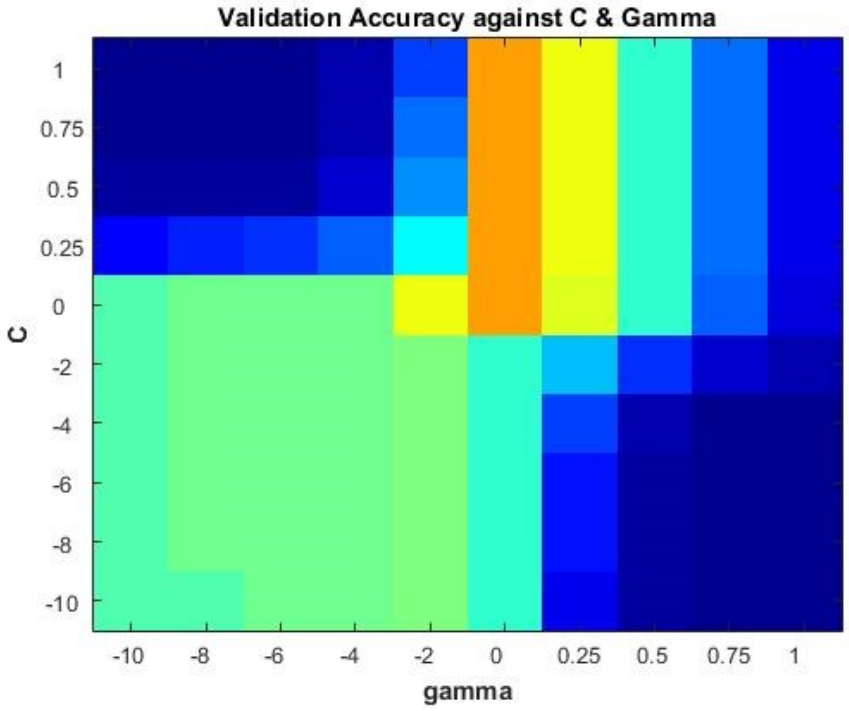

Figure 5.7: Heat map of **validation** accuracies against the hyper-parameters C & gamma. By A4 Method.

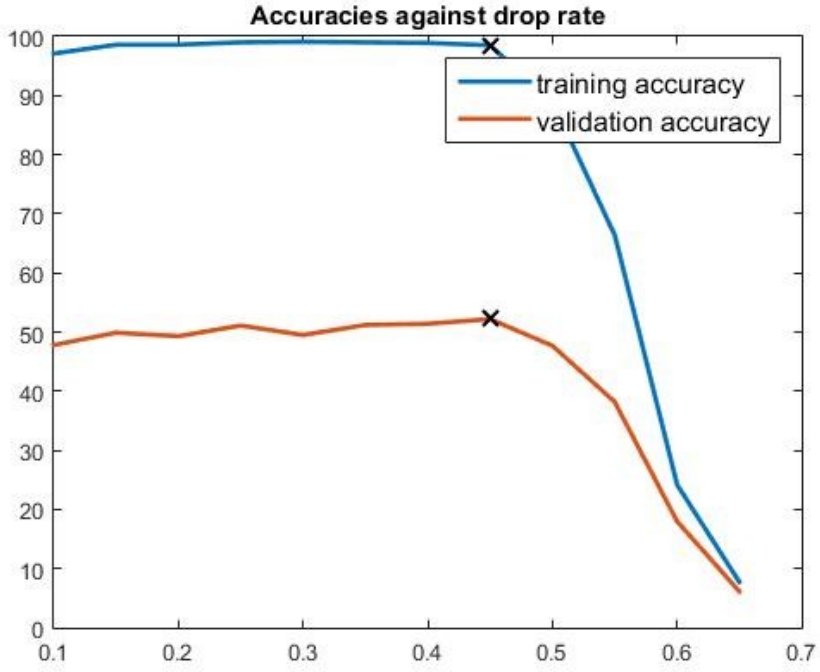

Figure 5.8: Plot of training and validation accuracies against the hyper-parameter C. By A5 Method

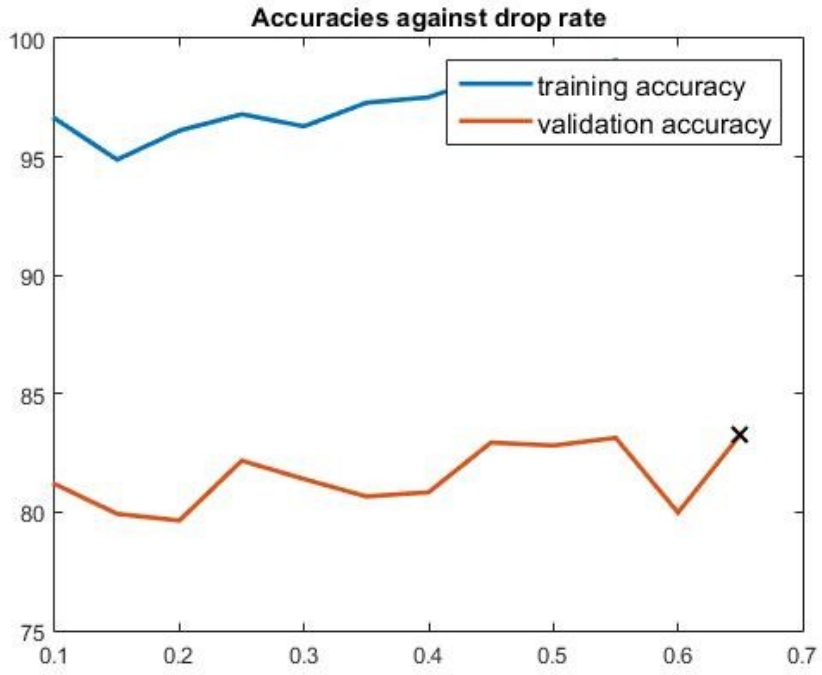

Figure 5.9: Plot of training and validation accuracies against the hyper-parameter C. By A6 Method

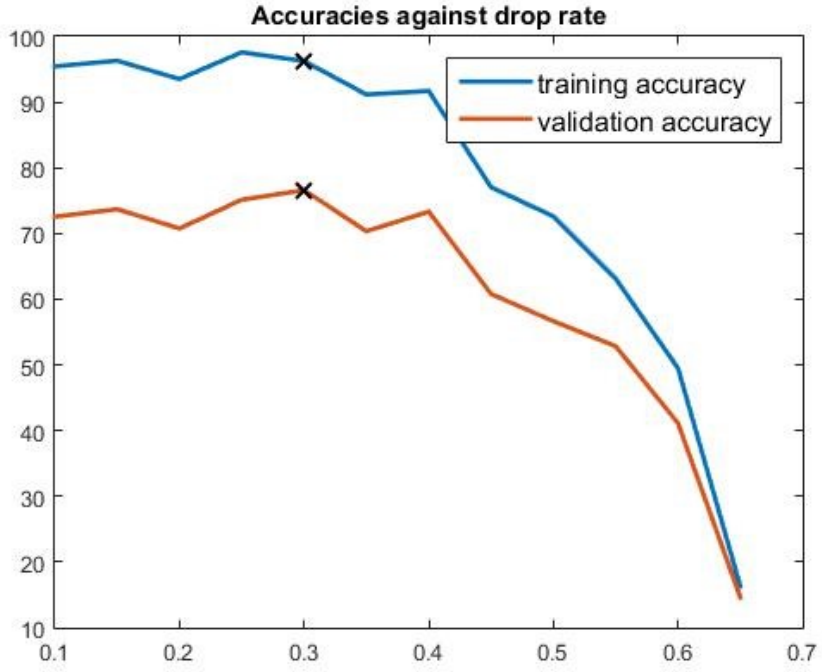

Figure 5.10: Plot of training and validation accuracies against the hyper-parameter C. By A7 Method

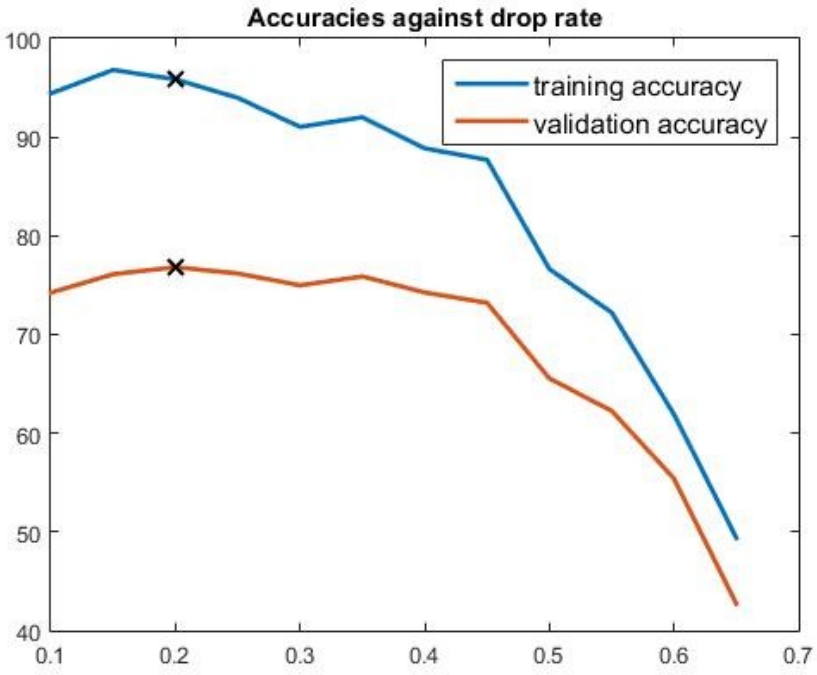

Figure 5.11: Plot of training and validation accuracies against the hyper-parameter C. By A8 Method

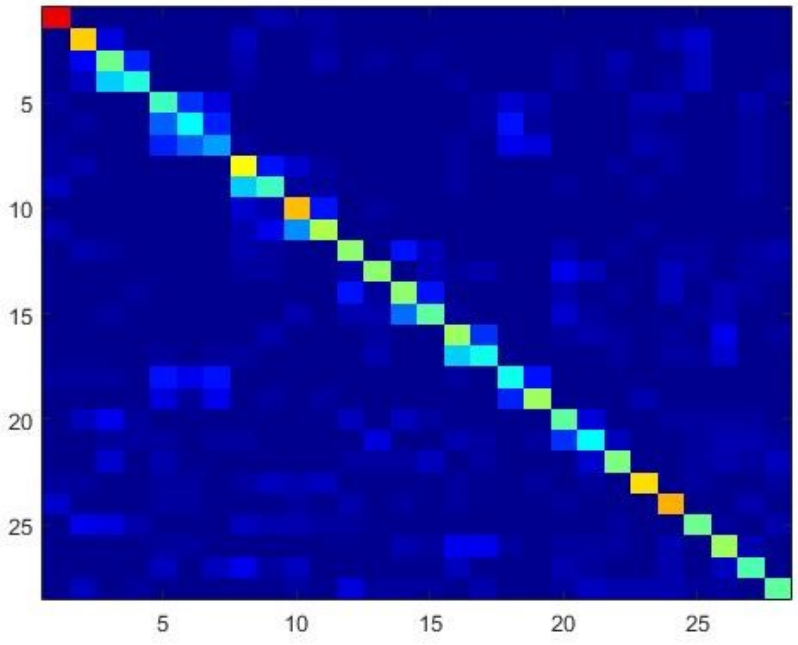

Figure 5.12: Confusion Matrix of Method A1 evaluated on Test set.

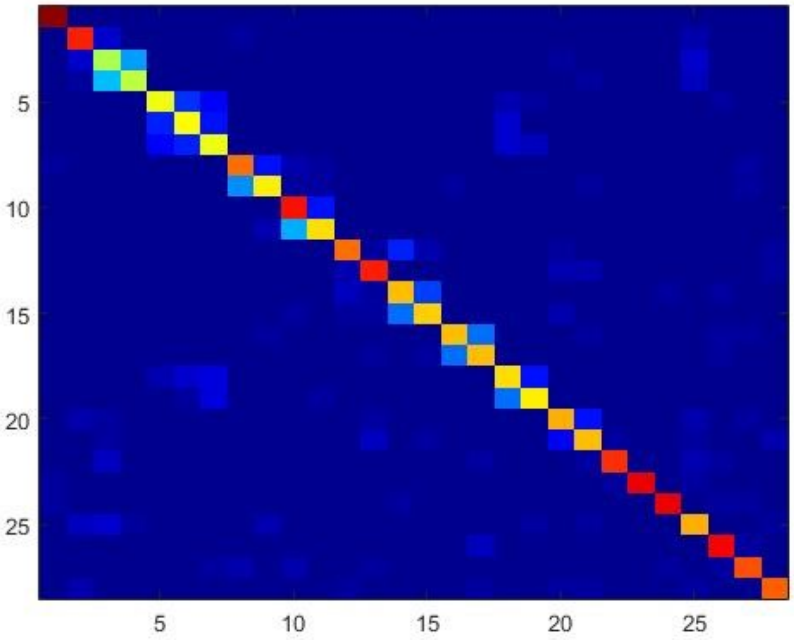

Figure 5.13: Confusion Matrix of Method A2 evaluated on Test set.

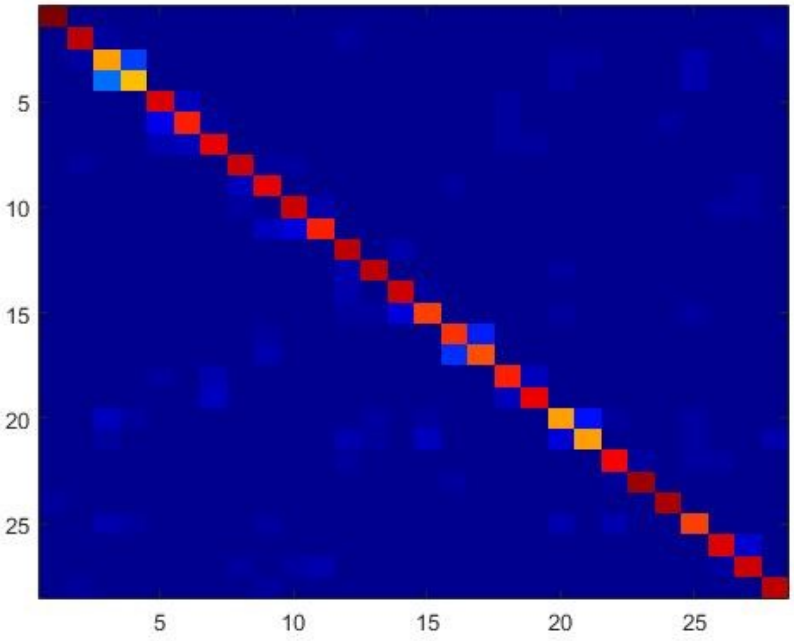

Figure 5.14: Confusion Matrix of Method A3 evaluated on Test set.

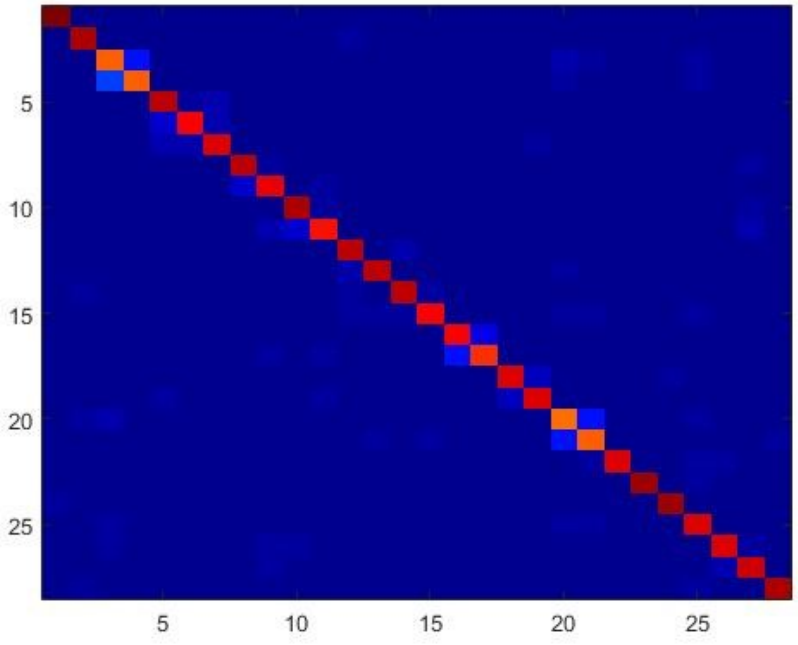

Figure 5.15: Confusion Matrix of Method A4 evaluated on Test set.

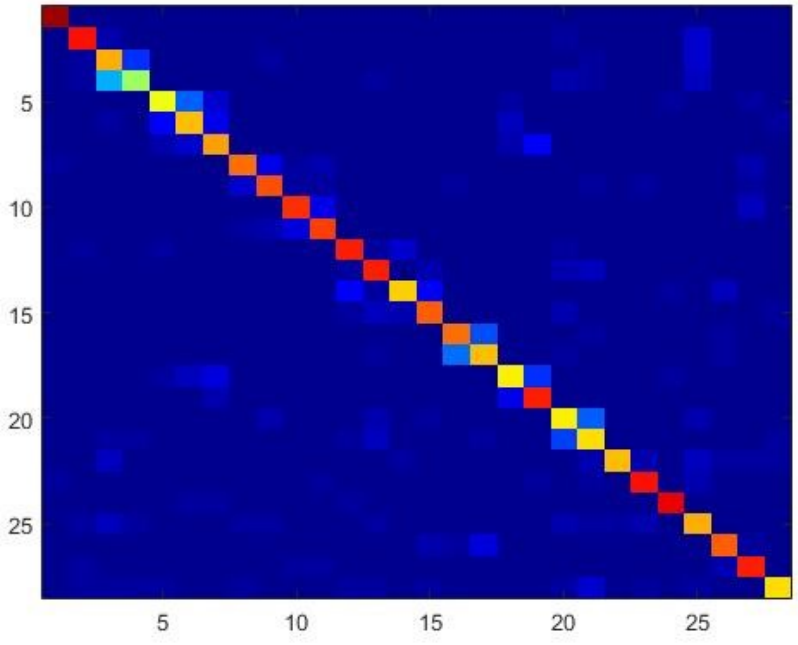

Figure 5.16: Confusion Matrix of Method A5 evaluated on Test set.

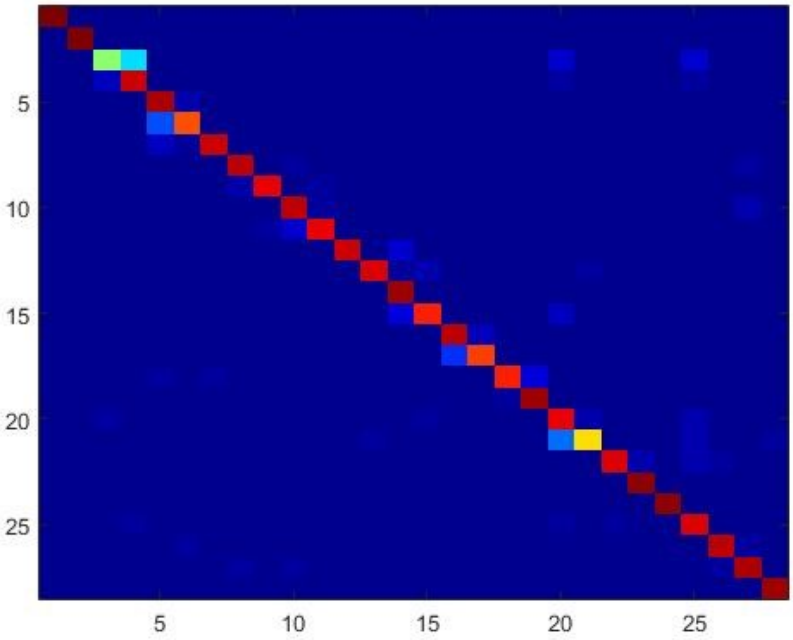

Figure 5.17: Confusion Matrix of Method A6 evaluated on Test set.

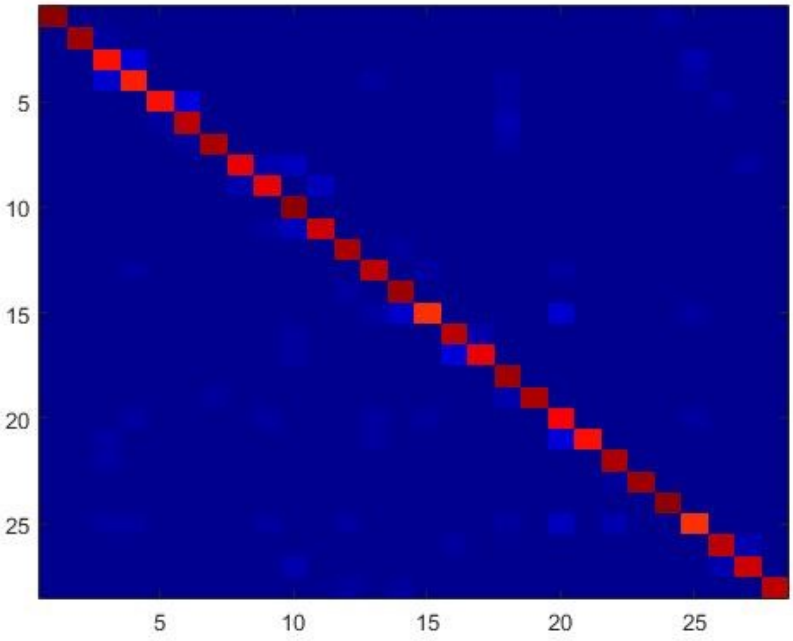

Figure 5.18: Confusion Matrix of Method A7 evaluated on Test set.

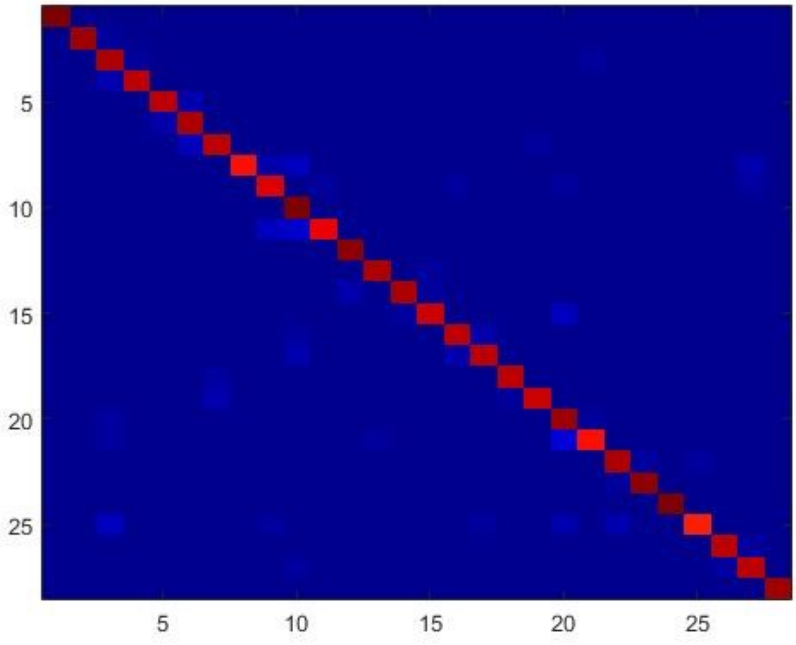

Figure 5.19: Confusion Matrix of Method A8 evaluated on Test set.

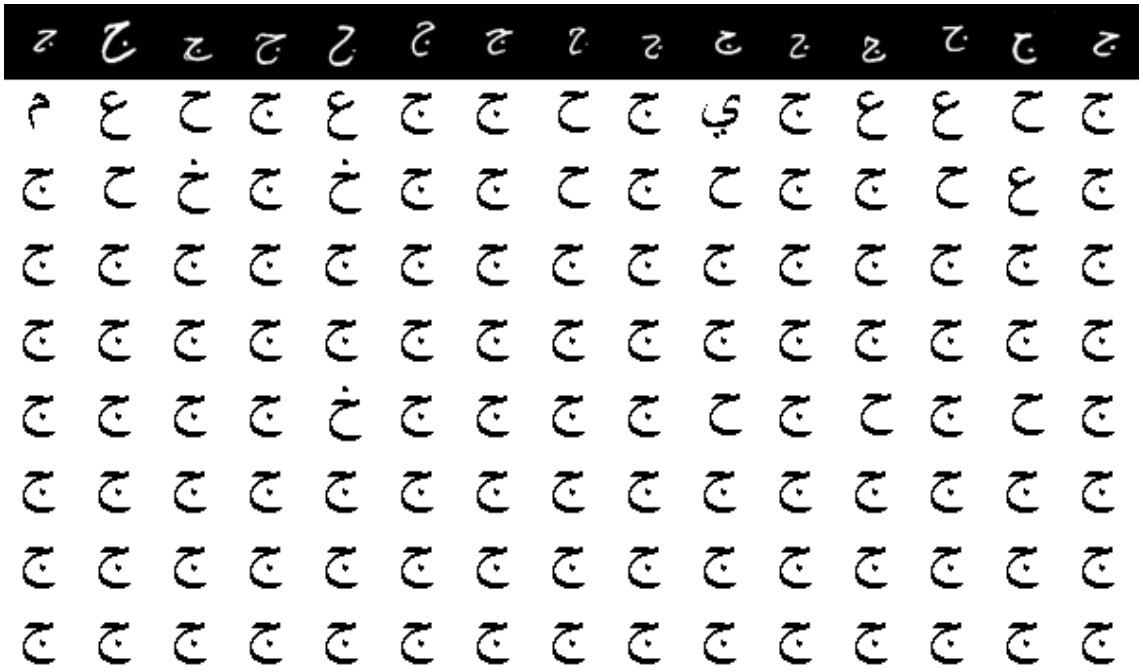

Figure 5.20: Working of all the methods discussed above on letter Jiim

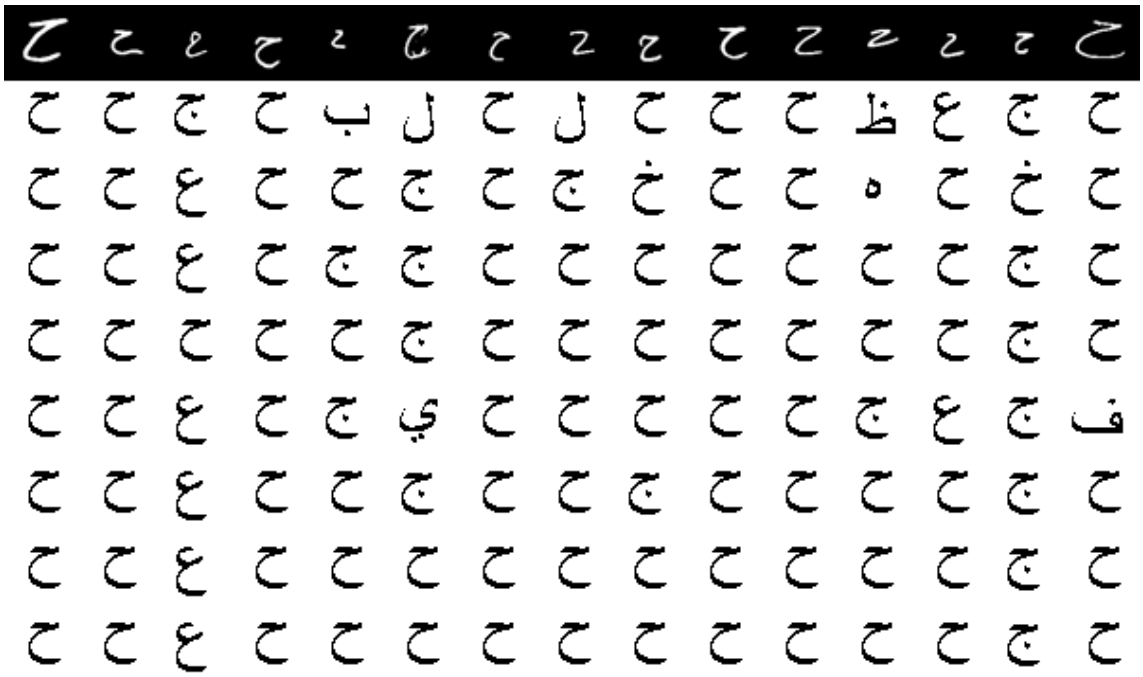

Figure 5.21: Working of all the methods discussed above on letter Haa

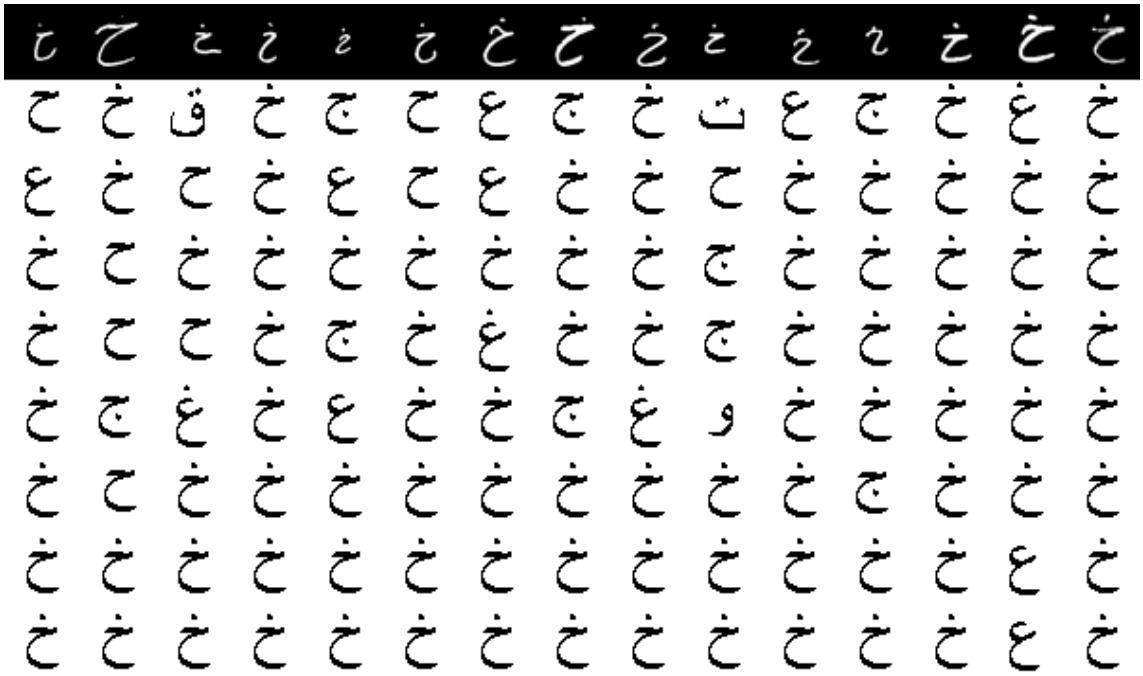

Figure 5.22: Working of all the methods discussed above on letter Khaa

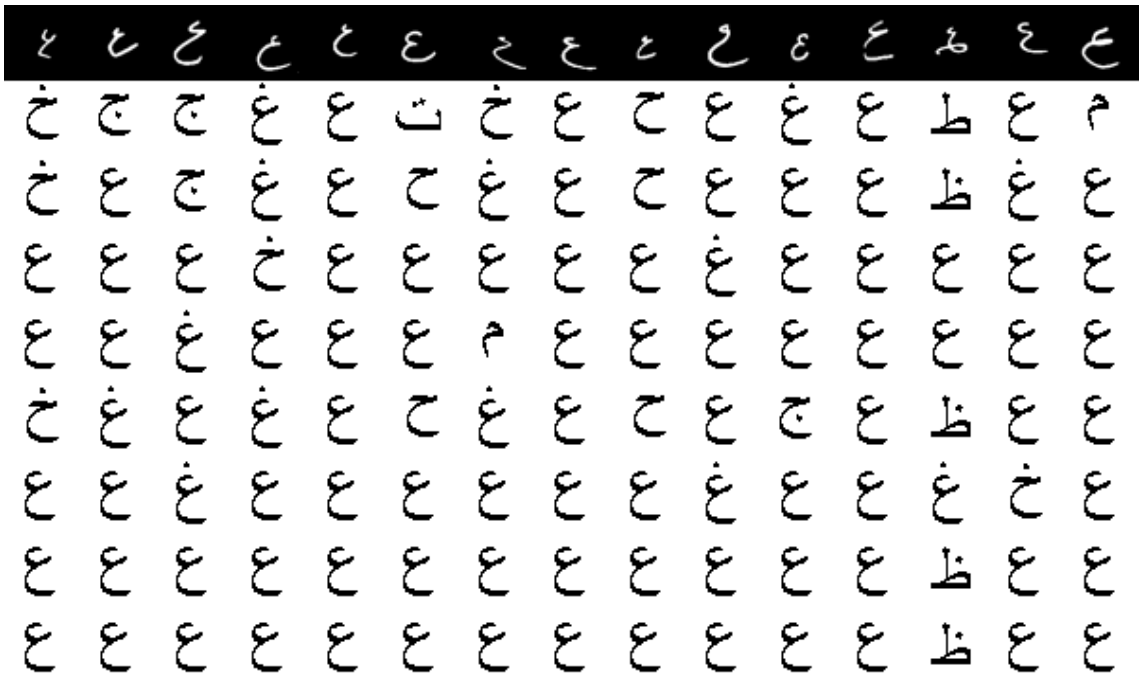

Figure 5.23: Working of all the methods discussed above on letter Ayn

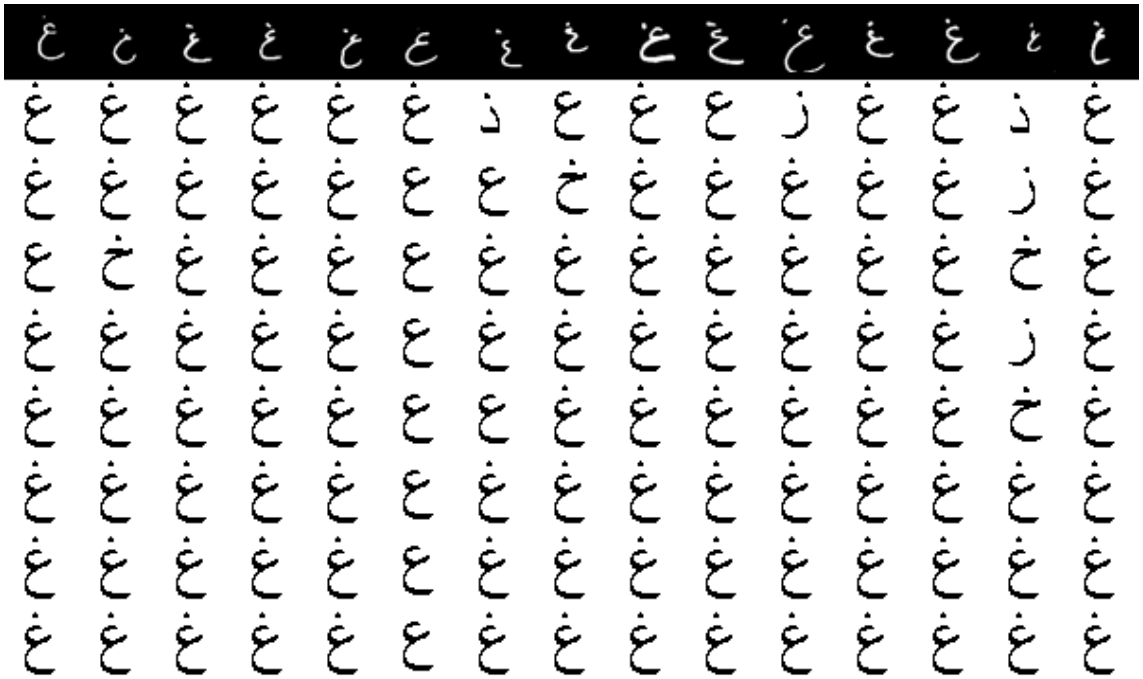

Figure 5.24: Working of all the methods discussed above on letter Ghayn

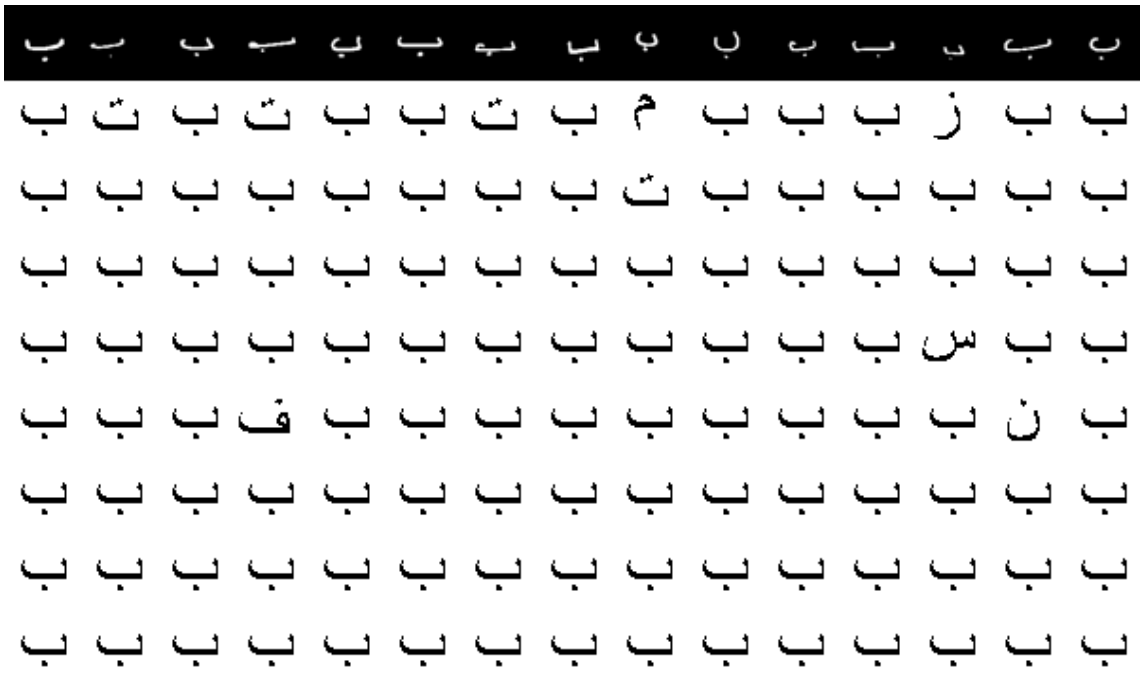

Figure 5.25: Working of all the methods discussed above on letter Baa

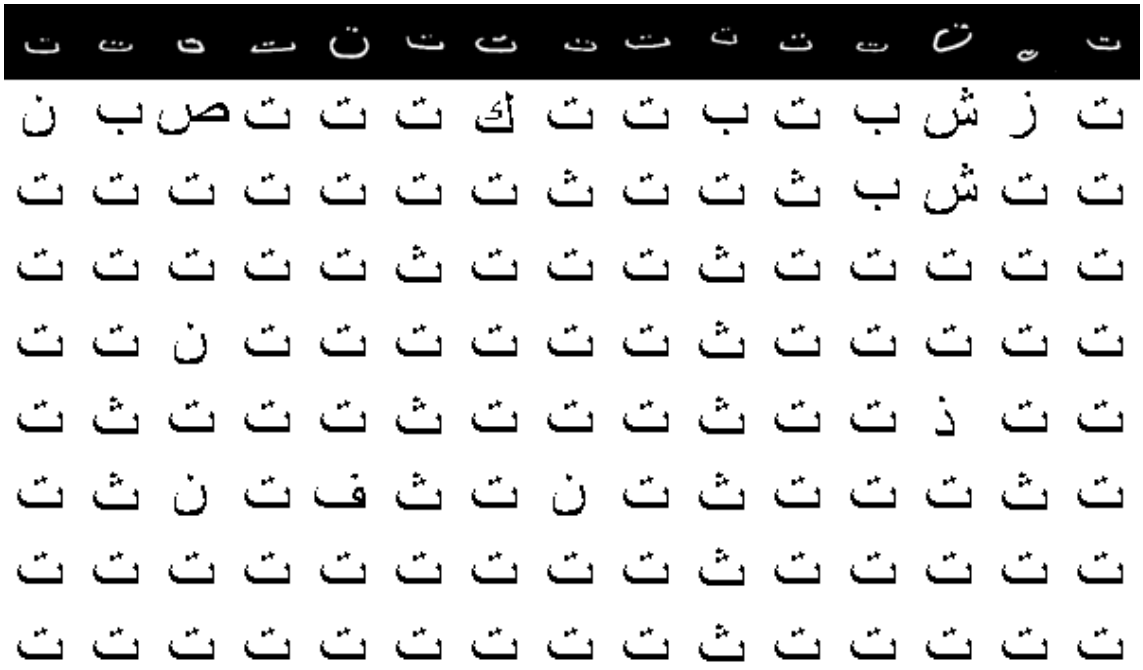

Figure 5.26: Working of all the methods discussed above on letter Taa

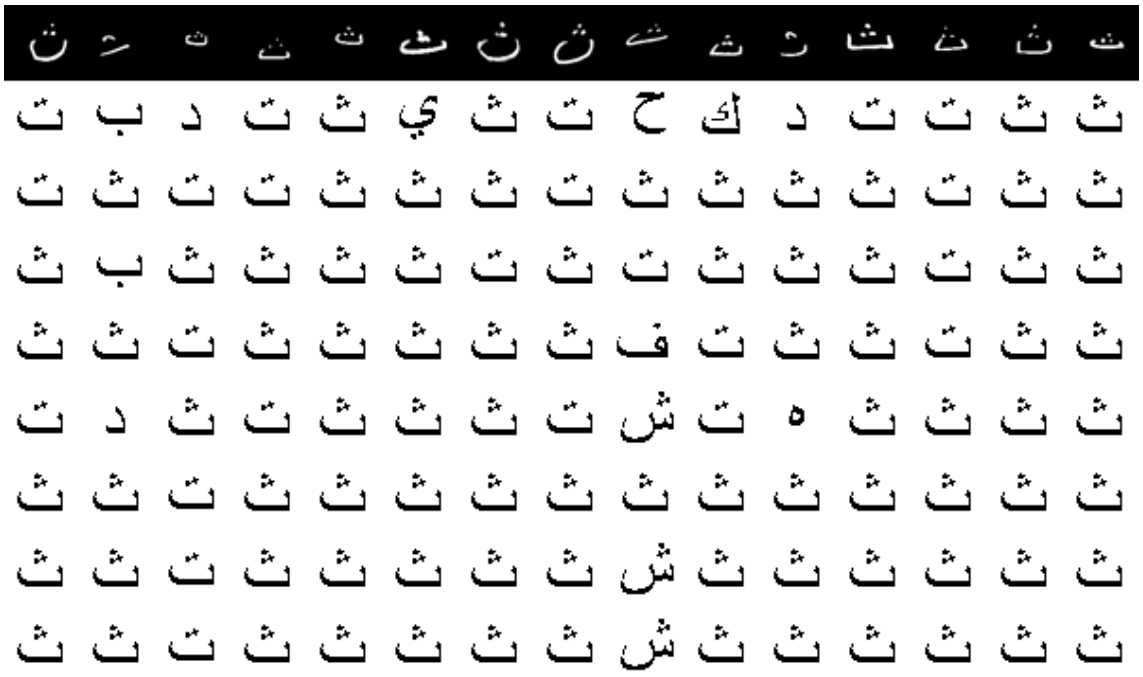

Figure 5.27: Working of all the methods discussed above on letter Thaa

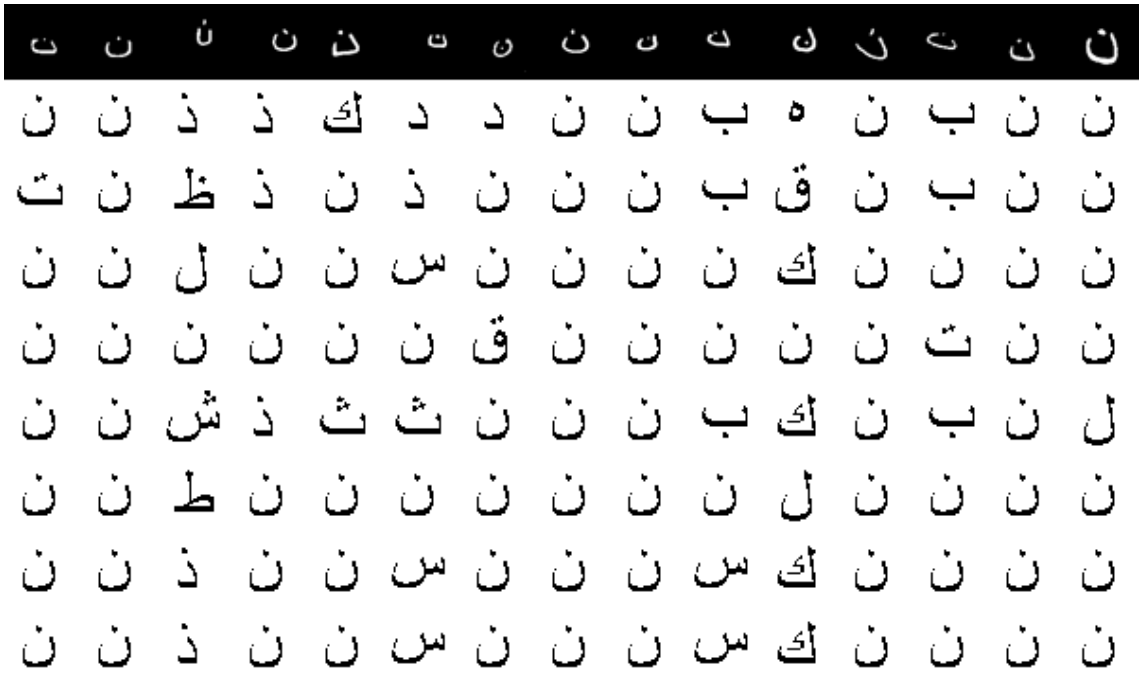

Figure 5.28: Working of all the methods discussed above on letter Nuun

Chapter 6

Conclusions

Arabic Handwritten character recognition is a difficult supervised learning problem. This is clear from the examples shown in figure 8 in the last section. It is clear from the figure that many Arabic characters when written with hand look strikingly similar. Sometimes a missing dot can change the character entirely as is clear from the figures 1a and 8. We tried deep learning, Convolutional Neural Networks to be specific, to perform the recognition task. We achieved more than 93% test accuracy. One other thing that can be done to improve the performance of the classifier is data augmentation i.e. by generating more training examples. If we just rotate or shift an image, the class will remain the same but the new image will be different from the original image. We can also add Gaussian Noise or blur the original image. In this way, we can easily increase our training data from 13,000 to more than 100,000 examples, with every example independent of the other. For example if we have 3 rotations and 3 translations for each image (9 transformations in total), size of the training data after data augmentation would be 13,000 X 9 = 117,000. This is one way of improving the character recognition task. In Handwriting recognition, we can combine the character recognition with Natural language processing to increase the test accuracy. That is we can improve the performance by not relying on just one method.

Appendix A

Program listings

A.1 Neural Networks Soruce Code

```
import numpy as np
import pandas as pd
import tensorflow as tf

# Load data from hard-disk (provide path in path variable).
# TrainImages.csv and TestImages.csv contain the input images.
# The input features are stored in matrix (DataFrame)
# of order number of examples by number of features.
# TrainLabels.csv and TestLabels.csv containt the corresponding labels.
# The output labels are stored in a vector (DataFrame)
# of order number of examples by 1.
path = ""
XTr = pd.read_csv(path + "TrainImages.csv", header=None)
yTr = pd.read_csv(path + "TrainLabels.csv", header=None)
XTe = pd.read_csv(path + "TestImages.csv", header=None)
yTe = pd.read_csv(path + "TestLabels.csv", header=None)

# Change the label vector to one-hot format.
# The labels are now stored as a matrix (DataFrame) of
# order number of features by number of classes.
# Change the data types of the training and test data
# from DataFrame to Numpy Array.
YTr = pd.get_dummies(yTr[0])
YTe = pd.get_dummies(yTe[0])
XTr = XTr.values.astype('float32')
XTe = XTe.values.astype('float32')
YTr = YTr.values.astype('float32')
YTe = YTe.values.astype('float32')

# Initialize some variables for example number of training
```

```
# and test examples pixels, classes.
# Initiallize parameters for the Convolution Neural Network.
# Number of layers and neurons in each layer and batch size.
NTr = XTr.shape[0]
NTe = XTe.shape[0]
D = XTr.shape[1]
K = len(np.unique(yTr))
nodes = [D, 500, 500, 500, 500, K]
L = len(nodes) - 1
batchSize = 120

# Define necessary placeholders for running tensorflow
# sessions. X and Y are for training on training set and
# evaluation on training and test data. train and rate are
# used for dropout which is done only during training.
X = tf.placeholder('float', [None, D])
Y = tf.placeholder('float', [None, K])
train = tf.placeholder('bool')
rate = tf.placeholder('float')


def neuralNet(Xdata, train, dropRate):
    """
     Define neural network model.
     This method peforms the forward propagation of the neural network
     with parameters defined above.

    :param Xdata: Input Features.
    :param train: the boolean indication of training or evaluation.
    :param dropRate: dropout rate.
    :return: the predicted output of the network.
    """

    network = Xdata
    for i in range(L):
        network = tf.contrib.layers.fully_connected(network, nodes[i + 1],
                                                    activation_fn=None)
        if (i < L - 1):
            network = tf.nn.relu(network)
        if (i == 2 or i == 3):
            network = tf.layers.dropout(network, rate=dropRate, training=train)
    return network


def trainNeuralNet(XTr, YTr, XTe, YTe, dropRate, showMess=True, epochs=10):
    """
    This method trains the neural network defined above.
```

```
78      It uses the neuralNet method to predict the output of
79      given input image for training and evaluation.
80
81      :param XTr: Images in the training set.
82      :param YTr: Lables in training set (one-hot).
83      :param XTe: Images in the test set.
84      :param YTe: Lables in test set (one-hot).
85      :param dropRate: dropout rate.
86      :param showMess: show the accuracies and loss for every epoch during training.
87      :param epochs: total number of epochs.
88      :return: The method returns the training and test accuracy and evaluation of the
89  trained network on test data.
90      """
91      H = neuralNet(X, train, rate)
92      cost = tf.reduce_mean(
93          tf.nn.softmax_cross_entropy_with_logits(logits=H, labels=Y))
94      optimizer = tf.train.AdamOptimizer().minimize(cost)
95      init = tf.global_variables_initializer()
96
97      h = tf.argmax(H, 1)
98      correctPredictions = tf.equal(h, tf.argmax(Y, 1))
99      accuracy = tf.reduce_mean(tf.cast(correctPredictions, 'float'))
100     with tf.Session() as sess:
101         sess.run(init)
102         for epoch in range(epochs):
103             epochLoss = 0
104             NTr = XTr.shape[0]
105             M = int(NTr / batchSize)
106             for i in range(M):
107                 ind = np.arange(0, batchSize) + i * batchSize
108                 XBatch = XTr[ind, :]
109                 YBatch = YTr[ind, :]
110                 _, loss = sess.run([optimizer, cost], feed_dict={
111                     X: XBatch, Y: YBatch, train: True, rate: dropRate
112                 })
113                 epochLoss += loss / float(M)
114             accTr = accuracy.eval(
115                 {X: XTr, Y: YTr, train: False, rate: dropRate})
116             accTe = accuracy.eval(
117                 {X: XTe, Y: YTe, train: False, rate: dropRate})
118             hTe = h.eval({X: XTe, Y: YTe, train: False, rate: dropRate})
119             if (showMess):
120                 print(
121                     "Epoch ", epoch + 1, " completed out of ", epochs,
122                     ", Loss: ",
123                     loss)
124                 print("Training Accuracy: ", accTr)
```

```
125                 print("Test Accuracy: ", accTe)
126                 print()
127
128     return accTr, accTe, hTe
129
130
131 def validateNeuralNet(Xdata, Ydata, split):
132     """
133     Find optimal dropout rate through cross-validation.
134     This method calls The trainNeuralNet method.
135     :param Xdata: Input Features of given set.
136     :param Ydata: Labels of given set (one-hot).
137     :param split: Fraction of the given set to be used for validation.
138     :return: optimal value of dropout rate.
139     """
140
141     numRates = 12
142     start = 0.1
143     step = 0.05
144     dropRates = step * np.arange(numRates) + start
145     accTr = np.zeros(numRates)
146     accCv = np.zeros(numRates)
147     N = Xdata.shape[0]
148     ind = np.random.rand(N) <= split
149     XTr = Xdata[ind, :]
150     XCv = Xdata[~ind, :]
151     YTr = Ydata[ind, :]
152     YCv = Ydata[~ind, :]
153     for i in range(0, numRates):
154         accTr[i], accCv[i], _ = trainNeuralNet(XTr, YTr, XCv, YCv, dropRates[i],
155                                                False, 50)
156         print("Drop Rate: ", dropRates[i])
157         print("Training Accuracy: ", accTr[i])
158         print("Validation Accuracy: ", accCv[i])
159         print()
160     i = np.argmax(accCv)
161
162     dropRates = np.reshape(dropRates, [numRates, 1])
163     accTr = np.reshape(accTr, [numRates, 1]) * 100
164     accCv = np.reshape(accCv, [numRates, 1]) * 100
165     arr = np.concatenate((dropRates, accTr, accCv), axis=1)
166     np.savetxt(path + "HogNN.csv", arr, delimiter=",")
167     return dropRates[i, 0]
168
169
170 # Call the method validateNeuralNet to perform
171 # dropRate = validateNeuralNet(XTr, YTr, 0.3)
```

```
# print(dropRate)
_, _, hTe = trainNeuralNet(XTr, YTr, XTe, YTe, 0.45, epochs=1)
# np.savetxt(path + "PredictionsHogNN.csv", hTe.astype(int), delimiter=",")
```

A.2 Convolutional Neural Network Source Code

```
import numpy as np
import pandas as pd
import tensorflow as tf

# Load data from hard-disk (provide path in path variable).
# TrainImages.csv and TestImages.csv contain the input images.
# The input features are stored in matrix (DataFrame)
# of order number of examples by number of features.
# TrainLabels.csv and TestLabels.csv containt the corresponding labels.
# The output labels are stored in a vector (DataFrame)
# of order number of examples by 1.
path = ""
XTr = pd.read_csv(path + "TrainImages.csv", header=None)
yTr = pd.read_csv(path + "TrainLabels.csv", header=None)
XTe = pd.read_csv(path + "TestImages.csv", header=None)
yTe = pd.read_csv(path + "TestLabels.csv", header=None)

# Initialize some variables for example number of training
# and test examples pixels, classes.
NTr = XTr.shape[0]
NTe = XTe.shape[0]
D = XTr.shape[1]
d = int(np.sqrt(D))
K = len(np.unique(yTr))

# Change the label vector to one-hot format.
# The labels are now stored as a matrix (DataFrame) of
# order number of features by number of classes.
# Change the formate of input feature matrix to a 3D tensor
# of order number of examples by number of rows by number
# of columns.
# Change the data types of the training and test data
# from DataFrame to Numpy Array.
XTr = XTr.values.astype('float32')
XTe = XTe.values.astype('float32')
XTr = XTr.reshape(NTr, d, d, 1)
XTe = XTe.reshape(NTe, d, d, 1)
YTr = pd.get_dummies(yTr[0])
YTe = pd.get_dummies(yTe[0])
```

```
40 YTr = YTr.values.astype('float32')
41 YTe = YTe.values.astype('float32')
42
43 # Initiallize parameters for the Convolution Neural Network.
44 # Number of Convolution layers, and properties like number
45 # of filters, kernel size, convolution strides and pooling
46 # for each layer.
47 # Number of Fully-Connected layers and neurons in each of
48 # them and batch size.
49 featureMaps = [80, 64, 40]
50 kernelSizes = [3, 3, 3]
51 strides = [1, 1, 1]
52 pooling = [True, False, False]
53 neurons = [1024, 512, K]
54 convLayers = len(featureMaps)
55 fullLayers = len(neurons)
56 batchSize = 120
57
58 # Define necessary placeholders for running tensorflow
59 # sessions. X and Y are for training on training set and
60 # evaluation on training and test data. train and rate are
61 # used for dropout which is done only during training.
62 X = tf.placeholder('float', [None, d, d, 1])
63 Y = tf.placeholder('float', [None, K])
64 train = tf.placeholder('bool')
65 rate = tf.placeholder('float')
66
67
68 def convNet(Xdata, train, dropRate):
69     """
70     Define convolutional neural network model.
71     This method peforms the forward propagation of the conv network
72     with parameters defined above.
73     :param Xdata: Input Features
74     :param train: the boolean indication of training or evaluation
75     :param dropRate: dropout rate
76     :return: the predicted output of the network.
77     """
78
79     network = Xdata
80     for layer in range(convLayers):
81         network = tf.layers.conv2d(network, featureMaps[layer],
82                                    kernelSizes[layer],
83                                    strides=strides[layer],
84                                    data_format="channels_last")
85         print("C", layer + 1, ": ", network)
86         network = tf.nn.relu(network)
```

```
87          if (pooling[layer]):
88              network = tf.layers.max_pooling2d(network, 2, 2)
89              print("P", layer + 1, ": ", network)
90
91      network = tf.contrib.layers.flatten(network)
92
93      for layer in range(fullLayers):
94          network = tf.contrib.layers.fully_connected(network, neurons[layer],
95                                                      activation_fn=None)
96          if (layer < fullLayers - 1):
97              network = tf.nn.relu(network)
98          network = tf.layers.dropout(network, rate=dropRate, training=train)
99
100     return network
101
102
103 def trainConvNet(XTr, YTr, XTe, YTe, dropRate, showMess=True, epochs=10):
104     """
105     This method trains the conv network defined above.
106     It uses the convNet method to predict the output of
107     given input image for training and evaluation.
108
109     :param XTr: Images in the training set.
110     :param YTr: Lables in training set (one-hot).
111     :param XTe: Images in the test set.
112     :param YTe: Lables in test set (one-hot).
113     :param dropRate: dropout rate.
114     :param showMess: show the accuracies and loss for every epoch during training.
115     :param epochs: total number of epochs.
116     :return: the training and test accuracy and evaluation of the trained
117     network on test data.
118     """
119
120     H = convNet(X, train, rate)
121     cost = tf.reduce_mean(
122         tf.nn.softmax_cross_entropy_with_logits(logits=H, labels=Y))
123     optimizer = tf.train.AdamOptimizer().minimize(cost)
124     init = tf.global_variables_initializer()
125
126     h = tf.argmax(H, 1)
127     correctPredictions = tf.equal(h, tf.argmax(Y, 1))
128     accuracy = tf.reduce_mean(tf.cast(correctPredictions, 'float'))
129     with tf.Session() as sess:
130         sess.run(init)
131         NTr = XTr.shape[0]
132         NTe = XTe.shape[0]
133         M = int(NTr / batchSize)
```

```
134         for epoch in range(epochs):
135             epochLoss = 0
136             for i in range(M):
137                 ind = np.arange(0, batchSize) + i * batchSize
138                 XBatch = XTr[ind, :, :, :]
139                 YBatch = YTr[ind, :]
140                 _, loss = sess.run([optimizer, cost],
141                                    feed_dict={X: XBatch, Y: YBatch, train: True,
142                                               rate: dropRate})
143                 epochLoss += loss / float(M)
144             accTr = 0.
145             accTe = 0.
146             hTe = np.zeros([0, 1])
147             B = 840
148             MTr = int(NTr / B)
149             MTe = int(NTe / B)
150             for i in range(MTr):
151                 ind = np.arange(0, B) + i * B
152                 XBatch = XTr[ind, :, :, :]
153                 YBatch = YTr[ind, :]
154                 accTr += accuracy.eval(
155                     {X: XBatch, Y: YBatch, train: False, rate: dropRate}) / MTr
156             for i in range(MTe):
157                 ind = np.arange(0, B) + i * B
158                 XBatch = XTe[ind, :, :, :]
159                 YBatch = YTe[ind, :]
160                 accTe += accuracy.eval(
161                     {X: XBatch, Y: YBatch, train: False, rate: dropRate}) / MTe
162                 hBatch = h.eval(
163                     {X: XBatch, Y: YBatch, train: False, rate: dropRate})
164                 hBatch = np.reshape(hBatch, [B, 1])
165                 hTe = np.concatenate((hTe, hBatch))
166 
167             if (showMess):
168                 print(
169                     "Epoch ", epoch + 1, " completed out of ", epochs,
170                     ", Loss: ",
171                     loss)
172                 print("Training Accuracy: ", accTr)
173                 print("Test Accuracy: ", accTe)
174                 print()
175 
176     return accTr, accTe, hTe
177 
178 
179 def validateConvNet(Xdata, Ydata, split):
180     """
```

```
    Find optimal dropout rate through cross-validation.
    This method calls The trainConvNet method.
    :param Xdata: Input Features of given set
    :param Ydata: Labels of given set (one-hot)
    :param split: Fraction of the given set to be used for validation
    :return: optimal value of dropout rate
    """

    numRates = 12
    start = 0.1
    step = 0.05
    dropRates = step * np.arange(numRates) + start
    accTr = np.zeros(numRates)
    accCv = np.zeros(numRates)
    N = Xdata.shape[0]
    ind = np.random.rand(N) <= split
    XTr = Xdata[ind, :]
    XCv = Xdata[~ind, :]
    YTr = Ydata[ind, :]
    YCv = Ydata[~ind, :]
    for i in range(0, numRates):
        accTr[i], accCv[i], _ = trainConvNet(XTr, YTr, XCv, YCv, dropRates[i],
                                             False, 10)
        print("Drop Rate: ", dropRates[i])
        print("Training Accuracy: ", accTr[i])
        print("Validation Accuracy: ", accCv[i])
        print()
    i = np.argmax(accCv)

    dropRates = np.reshape(dropRates, [numRates, 1])
    accTr = np.reshape(accTr, [numRates, 1]) * 100
    accCv = np.reshape(accCv, [numRates, 1]) * 100
    arr = np.concatenate((dropRates, accTr, accCv), axis=1)
    np.savetxt(path + "CNN2.csv", arr, delimiter=",")

    return dropRates[i, 0]


# dropRate = validateConvNet(XTr, YTr, 0.3)
# print(dropRate)
_, _, hTe = trainConvNet(XTr, YTr, XTe, YTe, 0.2, epochs=30)
# np.savetxt(path + "PredictionsCNN2.csv", hTe.astype(int), delimiter=",")
```

A.3 Histogram of Oriented Gradients Source Code

```
function Hogs = hogFeatures(X, cellSize, bins, clip)

% Inputs;
% X - Matrix of Images of order N by D where N is number of images and D is
% number of pixels
% cellSize = a 2-D vector which contains the cellSize. a parameter needed
% for extractHOGFeatures method
% bins - number of histogram bins. a parameter needed for
% extractHOGFeatures method
% clip - optional parameter, the method clips the image from all four
% boundaries by number of pixels given be clip
%
% Output;
% Hogs - Matrix of HOG Features of Order N by Dh where N is number of
% images and Dh is number of Hog features for each image. Dh
% depends on D, cellSize and bins.

if (nargin<4)
    clip = 0;
end

% Reshape the original Matrix to a 3D tensor of order N by d by d. where d
% is the number of rows and number of columns of each image.
[N, D] = size(X);
d = sqrt(D);
imgs = reshape(X', d, d, N);
imgs = imgs(1+clip:end-clip, 1+clip:end-clip, :);

% Extract HOG features
hogs = extractHOGFeatures(imgs(:, :, 1), ...
        'CellSize', cellSize, 'NumBins', bins);
Hogs = zeros(N, length(hogs));
Hogs(1, :) = hogs;
for k = 2:N;
    Hogs(k, :) = extractHOGFeatures(imgs(:, :, k), ...
        'CellSize', cellSize, 'NumBins', bins);
end


end
```

A.4 Gaussian Support Vector Machines Sources Code

```
close all; clear; clc;

% load data (training and test set) from a mat file
load('data.mat');

% Split the original training set into training and validation sets
NTr = length(XTr); p = 0.3;
XCv = XTr(1:p*NTr, :);
YCv = YTr(1:p*NTr);
XTr = XTr(p*NTr+1:NTr, :);
YTr = YTr(p*NTr+1:NTr, :);

% Find HOG features of all the images. Comment this part if feature
% extraction is not required
cellSize = [8 8]; bins = 8; clip = 0;
XTr = hogFeatures(XTr, cellSize, bins, clip);
XCv = hogFeatures(XCv, cellSize, bins, clip);
XTe = hogFeatures(XTe, cellSize, bins, clip);

% Convert the feature matrices to sparse format which is the
% requirement of libsvmtrain
XTr = sparse(XTr);
XCv = sparse(XCv);
XTe = sparse(XTe);

% initiallize some variables for examples C and Gamma as vectors on which
% we will perform cross-validation and select the optimal parameters.
% Train the Gaussian SVM model for all possible values of C and Gamma
% and store the parameters in a cell-matrix
Lc = 6; C = logspace(-10, 0, Lc)';
Lg = 6; Gamma = logspace(-10, 0, Lg);
model = cell(Lc, Lg);
start = tic;
disp(['started at ', num2str(toc(start)/60), ' minutes']);
for lc = 1:Lc;
    disp(lc);
    for lg = 1:Lg;
        options = ['-q -c ', num2str(C(lc)), ' -g ', num2str(Gamma(lg))];
        model{lc, lg} = libsvmtrain(YTr, XTr, options);
        disp(['   ', num2str(lg), ' finished at ', ...
            num2str(toc(start)/60), ' minutes.']);
    end
end
```

```
44
45  % Find and Store the training and validation errors in the corresponding
46  % matrices
47  JTr = zeros(Lc, Lg, 3);
48  JCv = zeros(Lc, Lg, 3);
49  for lc = 1:Lc;
50      disp(lc);
51      for lg = 1:Lg;
52          [~, JTr(lc, lg, :), ~] = libsvmpredict(YTr, XTr, ...
53              model{lc, lg}, '-q');
54          [~, JCv(lc, lg, :), ~] = libsvmpredict(YCv, XCv, ...
55              model{lc, lg}, '-q');
56          disp(['   ', num2str(lg), ' finished at ', ...
57              num2str(toc(start)/60), ' minutes.']);
58      end
59  end
60
61  % Choose the parameters C and Gamma with smallest validation error
62  % Save the results for the report
63  trainAccuracy = JTr(:, :, 1);
64  validationAccuracy = JCv(:, :, 1);
65  [i, j] = find(validationAccuracy == max(validationAccuracy(:)));
66  modelGaussianSVM1 = model{i, j};
67  [~, JTe, ~] = libsvmpredict(YTe, XTe, modelGaussianSVM1, '-q');
68  testAccuracy = JTe(1);
69  save('GaussianSVM1.mat', 'modelGaussianSVM1', 'trainAccuracy', ...
70      'validationAccuracy', 'testAccuracy', 'C', 'Gamma');
```

A.5 Linear Support Vector Machines Sources Code

```
1   close all; clear; clc;
2
3   % load data (training and test set) from a mat file
4   load('data.mat');
5
6   % Split the original training set into training and validation sets
7   NTr = length(XTr); p = 0.3;
8   XCv = XTr(1:p*NTr, :);
9   YCv = YTr(1:p*NTr);
10  XTr = XTr(p*NTr+1:NTr, :);
11  YTr = YTr(p*NTr+1:NTr, :);
12
13  % Find HOG features of all the images. Comment this part if feature
14  % extraction is not required
15  cellSize = [8 8]; bins = 8; clip = 0;
```

```
16 XTr = hogFeatures(XTr, cellSize, bins, clip);
17 XCv = hogFeatures(XCv, cellSize, bins, clip);
18 XTe = hogFeatures(XTe, cellSize, bins, clip);
19
20 % Convert the feature matrices to sparse format which is the
21 % requirement of libsvmtrain
22 XTr = sparse(XTr);
23 XCv = sparse(XCv);
24 XTe = sparse(XTe);
25
26 % initiallize some variables for examples C as a vector on which
27 % we will perform cross-validation and select the optimal parameterss.
28 % Train the Linear SVM model for all possible values of C and store the
29 % parameters in a cell-vector.
30 Lc = 6; C = logspace(-10, 0, Lc)';
31 model = cell(Lc);
32 start = tic;
33 disp(['started at ', num2str(toc(start)/60), ' minutes']);
34 for lc = 1:Lc;
35     options = ['-q -c ', num2str(C(lc)), ' -t 0'];
36     model{lc} = libsvmtrain(YTr, XTr, options);
37     disp([num2str(lc), ' finished at ', ...
38         num2str(toc(start)/60), ' minutes.']);
39 end
40
41 % Find and Store the training and validation errors in the corresponding
42 % vectors
43 JTr = zeros(Lc, 3);
44 JCv = zeros(Lc, 3);
45 for lc = 1:Lc;
46     [~, JTr(lc, :), ~] = libsvmpredict(YTr, XTr, ...
47         model{lc}, '-q');
48     [~, JCv(lc, :), ~] = libsvmpredict(YCv, XCv, ...
49         model{lc}, '-q');
50     disp([num2str(lc), ' finished at ', ...
51         num2str(toc(start)/60), ' minutes.']);
52 end
53
54 % Choose the parameter C with smallest validation error
55 % Save the results for the report
56 trainAccuracy = JTr(:, 1);
57 validationAccuracy = JCv(:, 1);
58 i = find(validationAccuracy == max(validationAccuracy));
59 modelHogLinearSVM2 = model{i};
60 [~, JTe, ~] = libsvmpredict(YTe, XTe, modelHogLinearSVM2, '-q');
61 testAccuracy = JTe(1);
62 save('HogLinearSVM2.mat', 'modelHogLinearSVM2', 'trainAccuracy', ...
```

```
63      'validationAccuracy', 'testAccuracy', 'C');
```

Bibliography

[1] Simons, Gary F. and Charles D. Fennig (eds.). 2017. Ethnologue: Languages of the World, Twentieth edition. Dallas, Texas: SIL International

[2] Tappert, C.C., Suen, C.Y. and Wakahara, T., 1990. The state of the art in online handwriting recognition. *IEEE Transactions on pattern analysis and machine intelligence*, 12(8), pp.787-808.

[3] Xu, L., Krzyzak, A. and Suen, C.Y., 1992. Methods of combining multiple classifiers and their applications to handwriting recognition. IEEE transactions on systems, man, and cybernetics, 22(3), pp.418-435.

[4] Plamondon, R. and Srihari, S.N., 2000. Online and off-line handwriting recognition: a comprehensive survey. IEEE Transactions on pattern analysis and machine intelligence, 22(1), pp.63-84.

[5] Lorigo, L.M. and Govindaraju, V., 2006. Offline Arabic handwriting recognition: a survey. IEEE transactions on pattern analysis and machine intelligence, 28(5), pp.712-724.

[6] Al-Badr, B. and Mahmoud, S.A., 1995. Survey and bibliography of Arabic optical text recognition. Signal processing, 41(1), pp.49-77.

[7] El-Sawy, A., Loey, M. and Hazem, E.B., Arabic Handwritten Characters Recognition using Convolutional Neural Network.

[8] Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

[9] Purohit, A. and Chauhan, S.S., 2016. A Literature Survey on Handwritten Character Recognition. International Journal of Computer applications & Information Technology, 7(1).

[10] Tagougui, N., Kherallah, M. and Alimi, A.M., 2013. Online Arabic handwriting recognition: a survey. International Journal on Document Analysis and Recognition (IJDAR), 16(3), pp.209-226.

[11] LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P., 1998. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), pp.2278-2324.

[12] McConnell, R.K., Wayland Research Inc., 1986. Method of and apparatus for pattern recognition. U.S. Patent 4,567,610.

[13] Marr, D. and Hildreth, E., 1980. Theory of edge detection. Proceedings of the Royal Society of London B: Biological Sciences, 207(1167), pp.187-217.

[14] Canny, J., 1986. A computational approach to edge detection. IEEE Transactions on pattern analysis and machine intelligence, (6), pp.679-698.

[15] Harris, C. and Stephens, M., 1988, August. A combined corner and edge detector. In Alvey vision conference (Vol. 15, No. 50, pp. 10-5244).

[16] Viola, P. and Jones, M., 2001. Rapid object detection using a boosted cascade of simple features. In Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on (Vol. 1, pp. I-I). IEEE.

[17] Lienhart, R. and Maydt, J., 2002. An extended set of haar-like features for rapid object detection. In Image Processing. 2002. Proceedings. 2002 International Conference on (Vol. 1, pp. I-I). IEEE.

[18] McConnell, R.K., Wayland Research Inc., 1986. Method of and apparatus for pattern recognition. U.S. Patent 4,567,610.

[19] Dalal, N. and Triggs, B., 2005, June. Histograms of oriented gradients for human detection. In Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on (Vol. 1, pp. 886-893). IEEE.

[20] Lowe, D.G., 1999. Object recognition from local scale-invariant features. In Computer vision, 1999. The proceedings of the seventh IEEE international conference on (Vol. 2, pp. 1150-1157). Ieee.

[21] Bay, H., Tuytelaars, T. and Van Gool, L., 2006. Surf: Speeded up robust features. Computer vision–ECCV 2006, pp.404-417.

[22] VC, H.P., 1962. Method and means for recognizing complex patterns. U.S. Patent 3,069,654.

[23] Duda, R.O. and Hart, P.E., 1972. Use of the Hough transformation to detect lines and curves in pictures. Communications of the ACM, 15(1), pp.11-15.

[24] Ballard, D.H., 1981. Generalizing the Hough transform to detect arbitrary shapes. Pattern recognition, 13(2), pp.111-122.

[25] Hubel, D.H. and Wiesel, T.N., 1962. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. The Journal of physiology, 160(1), pp.106-154.

[26] Deng, J., Dong, W., Socher, R., Li, L.J., Li, K. and Fei-Fei, L., 2009, June. Imagenet: A large-scale hierarchical image database. In Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on (pp. 248-255). IEEE.

[27] Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

[28] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A., 2015. Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1-9).

[29] He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

[30] D. Tran, "Face Detection on Cloud Foundry – Dat Tran – Medium", Medium, 2017. [Online]. Available: https://medium.com/@datitran/face-detection-on-cloud-foundry-37887631096a. [Accessed: 01- Sep- 2017].

[31] "Introduction to SIFT (Scale-Invariant Feature Transform) — OpenCV 3.0.0-dev documentation", Docs.opencv.org, 2017. [Online]. Available: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_sift_intro/py_sift_intro.html. [Accessed: 07- Aug- 2017].

[32] "The Business of Artificial Intelligence", Harvard Business Review, 2017. [Online]. Available: https://hbr.org/cover-story/2017/07/the-business-of-artificial-intelligence. [Accessed: 10- Sep- 2017].

[33] High Level Computer Vision - Intro to Deep Learning for Computer Vision. Saarbrücken: Bernt Schiele, 2017, p. 7.

[34] E. Kim, "The Kernel Trick", Eric-kim.net, 2017. [Online]. Available: http://www.eric-kim.net/eric-kim-net/posts/1/kernel_trick.html. [Accessed: 09- Oct- 2017].

[35] A. Ng, "Machine Learning", Open Classroom, 2017. [Online]. Available: http://openclassroom.stanford.edu [Accessed: 05- Sep- 2017].

[36] S. Mallick, "Histogram of Oriented Gradients — Learn OpenCV", Learnopencv.com, 2017. [Online]. Available: https://www.learnopencv.com/histogram-of-oriented-gradients/. [Accessed: 28- Aug- 2017].

[37] "Overfitting", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Overfitting. [Accessed: 02- Aug- 2017].

[38] P. Stenius and S. Yoo, "Artificial intelligence - tailored smart solutions today, true intelligence tomorrow", Reddal, 2017. [Online]. Available: http://www.reddal.com/insights/artificial-intelligence-tailored-smart-solutions-today-true-intelligence-tomorrow/. [Accessed: 08- Oct- 2017].

[39] A. Narwekar and A. Pampari, Recurrent Neural Network Architectures. Illinois: University of Illinois at Urbana-Champaign, 2017, p. 10.

[40] "Convolution arithmetic tutorial — Theano 0.9.0 documentation", Deeplearning.net, 2017. [Online]. Available: http://deeplearning.net/software/theano/tutorial/conv_arithmetic.html. [Accessed: 21- Oct- 2017].

[41] F. Li, "CS231n Convolutional Neural Networks for Visual Recognition", Cs231n.github.io, 2017. [Online]. Available: https://cs231n.github.io/convolutional-networks/. [Accessed: 04- Oct- 2017].

[42] "Convolutional Neural Network", MathWorks, 2017. [Online]. Available: https://de.mathworks.com/discovery/convolutional-neural-network.html. [Accessed: 03- Sep- 2017].

[43] L. Toti Rigatelli and J. Denton, Evariste Galois. Basel [u.a.]: Birkhauser, 1996.