Reduced Dataset Neural Network Model for Manuscript Character Recognition

Mohammad Anwarul Islam

REDUCED DATASET NEURAL NETWORK MODEL FOR MANUSCRIPT CHARACTER RECOGNITION

by

MOHAMMAD ANWARUL ISLAM

(Under the Direction of Ionut Emil Iacob)

ABSTRACT

The automatic character recognition task has been of practical interest for a long time. Nowadays, well-established technologies and softwares exist for accurately performing character recognition from scanned documents. Although, handwritten character recognition from the manuscript images is challenging, the advancement of modern machine learning techniques makes it astonishingly manageable. The problem of accurately recognizing handwritten characters remains of high practical interest since a large number of manuscripts are currently not digitized, and hence not open to the public. We built our repository of the datasets by cropping each letter image manually from the manuscript images. The scarcity of dataset was the major obstacle in our experiment. However, we overcame the problem by using resampling and convolutional techniques for performing character recognition in our neural network model with this reduced training dataset. The experimental result showed that our proposed model outperformed the previous work.

INDEX WORDS: Neural network, Convolution, Convolutional neural network, Resampling.

2010 Mathematics Subject Classification: 68U10, 68T10, 62H30

REDUCED DATASET NEURAL NETWORK MODEL FOR MANUSCRIPT CHARACTER RECOGNITION

by

MOHAMMAD ANWARUL ISLAM

B.S., University of Chittagong, Bangladesh, 2005

M.S., University of Chittagong, Bangladesh, 2006

A Thesis Submitted to the Graduate Faculty of Georgia Southern University in Partial

Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

REDUCED DATASET NEURAL NETWORK MODEL FOR MANUSCRIPT CHARACTER RECOGNITION

by

MOHAMMAD ANWARUL ISLAM

Major Professor:	Ionut Emil Iacob
Committee:	Goran Lesaja
	Felix Hamza-Lup

Electronic Version Approved:
July 2020

DEDICATION

I would like to dedicate this thesis to my late grandfather Eshak Ali and my parents.

ACKNOWLEDGMENTS

My heartfelt thanks and gratitude to such an amiable personality and mentor Dr. Ionut Emil Iacob and all the faculties of mathematical science department of Georgia Southern University.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

k	1000
$\mathcal{I}$	Set of images in our dataset
$\mathcal{A}$	Set of alphabet
$\overline{\mathcal{I}}$	Set of all Beowulf character images
$\mathcal{C}$	Character image recognition model
l	Total number of labels
$\forall$	For all
vec	Vector
$\mathbb{R}$	Set of real numbers
$\mathcal{I}_R$	Training images
$\mathcal{I}_T$	Test images
$\cup$	Union operation
$\mathbb{Z}$	Integers
$\mathbb{N}$	Set of natural numbers

CHAPTER 1
INTRODUCTION

Machine learning and artificial intelligence (AI) are now being used widely in science and technologies, such as image processing, pattern recognition, natural language processing, autonomous vehicles, etc. The neural network is one of the most powerful tools in solving problems in machine learning and AI. It is not a straight forward job to recognize letter images from a manuscript using machine learning algorithms. There are two types of manuscript letter image recognition, one is offline, and another is online. In an online manuscript letter image recognition, input information comes from a real-time writing sensor. In offline letter image recognition, input information comes from static images.

Handwritten letter image recognition is a classical vision problem. Though, it has been researched for a long time, it requires more improvement in its accuracy to recognize manuscript character images successfully. However, online handwritten character recognition has made significant improvement, but that of offline needs more research and attention. Historic handwritten documents, such as Issac Newton's papers at Cambridge University, George Washington's letters documents at the Library of Congress, Joseph Grinnell's collection in the Museum of Vertebrate Zoology at U.C Berkeley [5], and Beowulf's Old English epic poem, etc. require rigorous research in recognition work. In our thesis, we narrowed down this task only to the Beowulf Old English manuscript letter image recognition. Our work had some challenges, such as we did not have enough datasets for training and testing the model. Our thesis dealt with a small dataset of individual character images from the electronic version of Beowulf's Old English manuscript [10]. We tackled those challenges using the bootstrapping (resampling) and convolutional techniques. Our handwritten letter image recognition work outperformed the previous work [18]. We cropped the letter images from the electronic version of the Beowulf Old English manuscript images. We built a dataset of 440 cropped images from 22 categories of letter images, each category with 20 images. We trained our model with 80 percent of the data and tested with 20 percent of the data. In our model, we used convolution, resampling, pooling, etc. to maximize the result and obtained an accuracy rate of over 93 percent.

1.1 Artificial Intelligence

Artificial intelligence incorporates a set of intelligence in machines by replicating human-level intelligence. AI comprises four things [16]: think humanly, think rationally, behave humanly, and behave rationally. The first two attributes are about the thinking and reasoning processes and the third and fourth attributes are behavioral.

Definition 1. Artificial intelligence is an intelligence demonstrated by machines, where a well-defined set of algorithms is used by mimicking human intelligence.

Nowadays, the use of AI is ubiquitous. In mathematical calculation, solving a mathematical problem, searching content from a website, image recognition, robotics, natural language processing, autonomous car driving, medical image classification, and in many other fields AI has been used successfully. Speech recognition task has been successfully implemented by various application software such as Google assistant, Google home, Bixby, Cortana, etc. People can easily dictate these softwares to get jobs done. The venture of today's artificial intelligence started by proposing perceptron, which is the first mathematical model of a neural network [13]. Afterward, many research papers and scholarly articles were published on the hot cake topic of AI. As people become more and more dependent on machines, the research of machine intelligence and its development is growing faster. In the second decade of the twenty-first-century, machines have become more capable of performing jobs that require a considerable amount of human-level intelligence, such as jobs in medical science, education, research, transportation, business, entertainment, etc.

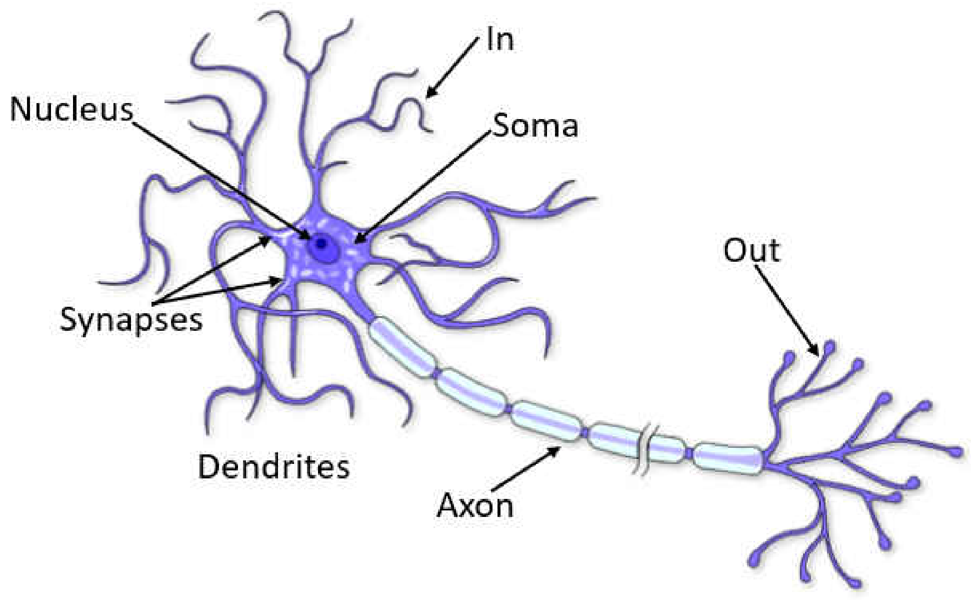


Figure 1.1: Biological neuron

1.1.1 BIOLOGICAL NEURONS

Natural intelligence is related to biological brains. The biological brain consists of billions of neuron cells. A neuron is the smallest unit of the brain. A neuron has different parts to make inter-communication among neurons. For example, dendrites are like tentacles that receive electrical and chemical signals from other neurons and are processed in the soma (nucleus of neuron). Then the axon of the neuron transmits the processed information to other neurons and the synapse establishes the connection with other neurons. Fig. 1.1 shows a biological neuron.

1.1.2 ARTIFICIAL NEURON

At first, Walter Pitts and Warren McCulloch came up with an artificial neuron's idea in 1943 based on the conception of the brain's neurons. This McCulloch-Pitts neuron was proposed as a logic gate with binary output.

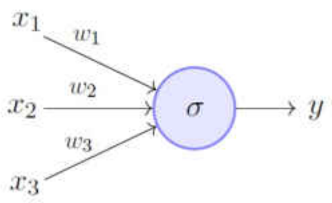


Figure 1.2: Artificial neuron

Definition 2. An artificial neuron is a mathematical function replicating the model of the biological neuron that receives one or multiple inputs, weighs each input separately, sums them up, and passes the final values through a smooth non-linear function to get the desired output.

In comparison to an artificial neuron with a biological neuron, we can consider that inputs act like dendrites, nodes are similar to neuron's nuclei, synapses are connections, and the axon is like the output.

1.1.3 Perceptron and Sigmoid Neuron

Perceptron is the first mathematical model and elementary unit of the neural network. A perceptron consists of inputs, weights, a bias, and an activation function. In 1957, the perceptron was introduced by Frank Rosenblatt. A perceptron is a computational algorithm, which usually performs binary classification. A perceptron is a uni-layer neural network, whereas a neural network is a multi-layer perceptron [20].

A perceptron takes multiple inputs, weighs each input, sums up the weighted inputs, and then adds up with the bias. If this value exceeds a certain threshold value, then it gives an output 1, if the value does not, then it gives an output 0. Perceptron's algorithm learns weights for each input and bias for each perceptron to produce a linear decision boundary curve. There are potential disadvantages of using perceptron no matter whether it is a uni-layer perceptron or a multi-layer perceptron. A perceptron gets fired if the output value is

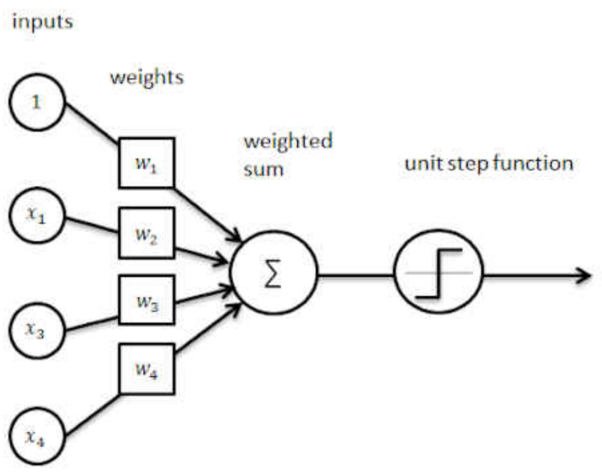


Figure 1.3: Perceptron model

greater than a certain threshold value, otherwise it does not. So, the output of the perceptron suffered from extreme properties at the boundary.

Suppose, we will decide whether a person will buy a car or not depending on his annual income. The threshold value for the perceptron is \$40k. If the person's income is \$40.01k, he or she will buy a car, but if the income is \$40k or \$39.99k the person will not buy a car. This means the output of a perceptron is either 0 or 1, there is nothing in between. To overcome this problem, sigmoid neurons come in the front, which gives a more logical and smooth output curve. Sigmoid neurons give output between 0 and 1. In other words, we can say sigmoid neurons can give more variation in output values than that of the perceptron.

Definition 3. The sigmoid neuron is a mathematical model, which mimics the biological neuron, and receives one or more inputs, weighs each input separately, and sums them up along with bias, and finally passes through the sigmoid function $y(x) = \dfrac{1}{1+e^{-x}}$ to get an output between 0 and 1.

The main advantage of the sigmoid neuron is that it produces infinitely many different values as outputs between 0 and 1. Thus, the sigmoid function gives a human level decision-making smooth decision curve.

1.2 Historical Background of Artificial Neural Network (ANN)

At first in 1943, Warren McCulloch and young mathematician Walter Pitts made an electrical model circuit of a simple neural network (NN). After that, the concept and mechanism of NN were described [7]. In 1950, the first NN simulation was led by Nathanial, a researcher of IBM. The research project on AI led by Dartmouth Summer in 1956, was a milestone of ANN. Afterward, a simple neuron function and a perceptron were introduced [15]. In 1959, Adaline and Madaline models were introduced by Bernard Widrow and Marcian Hoff. In 1973, Dreyfus introduced backpropagation to adjust parameters in NN. In 1975, the back propagation algorithm was developed by Werbos to train a multi-layer NN. Between the years 1985 and 1987, both the American Institute of Physics and International Electrical and Electronic Engineers adopted various steps for developing NN. Max-pooling was introduced in 1992. In 1997, long short-term memory [8] and recurrent neural networks were proposed. In 1998, Yann LeCun introduced gradient-based learning for document recognition. At the very beginning of the twenty-first century, Geoffre Hinton et al. (2006) proposed a high-level representation using successive layers of real-valued latent variables.

1.3 Artificial Neural Network

The artificial neural network (ANN) is one of the most dominating machine learning techniques, especially in image processing. ANN performs relatively complex computing tasks loosely inspired by the biological network of neurons. ANN is made up of an input layer, one or multiple hidden layers, and an output layer [2]. Depending on the nature and complexity of the problem, ANN might consist of thousands of hidden layers Figure 1.4.

Each layer has several inter-connected neurons, which are called artificial neurons. To get the desired output, activation functions are used in every neuron of the network. Activation functions are usually non-linear, which gives an output between 0 and 1, and in some cases either 0, or 1. In the input layer, ANN receives information to process in the first hidden layers. Outputs from the first hidden layer work as the inputs for the second hidden layer and outputs from the second hidden layer work as the inputs for the third hidden layer. This process continues until it considers all the hidden layers and the output layer.

Let us consider a neural network model with an input layer, one hidden layer and an output layer. Suppose, we have N inputs in the input layer i.e., the number of nodes or neurons in the input layer is N, number of neurons in the hidden layer is L, and a single output Y for the output layer. The ANN model with the above-mentioned neurons and layers can be described mathematically as follows-

$$Y : \mathbb{R}^n \to \{0, 1\} \tag{1.1}$$

$$Y(x_1, .., x_n; w_{11}, .., w_{NL}; z_1, .., z_n; b_{11}, .., b_{1L}, b_2) = \sigma(\sum_{j=1}^{L} z_j \sigma(\sum_{i=1}^{N} w_{ij} x_i + b_{1j}) + b_2) \tag{1.2}$$

where $1 \times N$ is the dimension of the input vector, $N \times L$ is the dimension of the of the weight matrix, and $1 \times L$ is the dimension of the bias vector in the hidden layer.
$x_1, x_2, .., x_n; w_{11}, w_{12}, .., w_{NL}; z_1, z_2, .., z_n; b_{11}, b_{12}, .., b_{1L}, b_2$ are the parameters of the above mentioned ANN model, $\sigma()$ is the activation function acting on each neuron of the model. For the above ANN model, a sigmoid activation function is defined by the equation (1.4):

$$\sigma : \mathbb{R}^n \to \mathbb{R} \tag{1.3}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1.4}$$

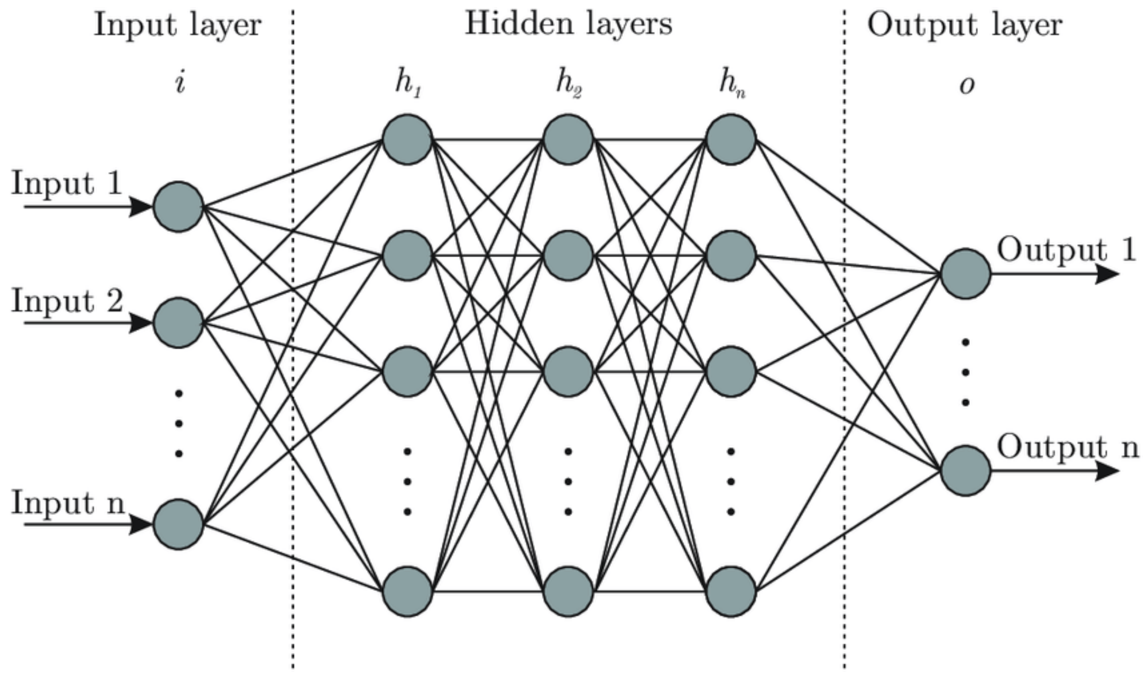


Figure 1.4: Artificial neural network

1.4 Activation Function

An activation function is one kind of mathematical function, which is used in the neural network model to define the output of a neuron for some given input values. The main role of an activation function is to transfer the weighted sum of input values from one layer of neurons to the next layer of neurons, and this continues in all the layers of neurons until the output layer. There are two kinds of activation functions, namely linear activation functions and non-linear activation functions. Non-linear activation functions are vastly used activation functions in deep neural networks.

1.4.1 Sigmoid Activation Function

The sigmoid activation function is known as a logistic function. The curve of this function is s-shaped. The mathematical formula of the sigmoid activation function is given

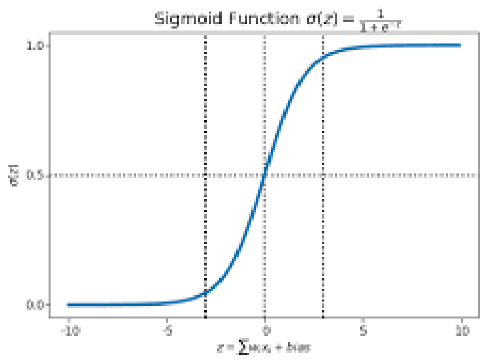


Figure 1.5: Curve of sigmoid function

below by the equation (1.5).

$$f(x) = \frac{1}{1+e^{-x}} \tag{1.5}$$

From the above function (1.5), we can see that for any input x, it gives values between 0 and 1. Therefore, the sigmoid activation function is useful for the models in which we need to predict probability as our output, because the value of probability is between 0 and 1 [21]. The sigmoid activation function changes the weights and biases slowly, so that the NN model gets more scope to adjust its weights before a drastic change in its output. The sigmoid activation function is computationally expensive.

1.4.2 ReLu Activation Function

The rectified linear unit (ReLu) activation function is a piecewise linear function, which takes both positive and negative values as inputs and produces outputs, either zero or positive numbers [14]. The mathematical formula for ReLu activation is given by the equation (1.6).

$$f(x) = max(0, x) \tag{1.6}$$

In the equation (1.6), x is input variable. If x is negative, it gives 0 as output; if x is

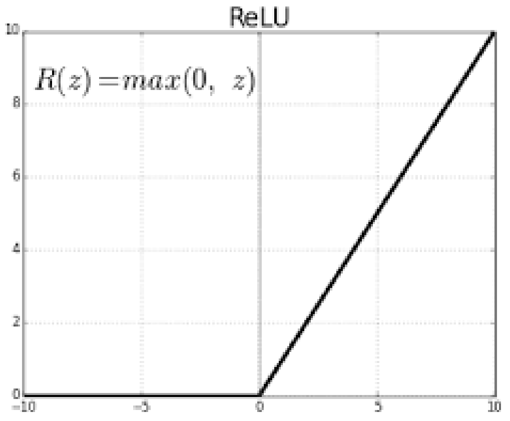


Figure 1.6: Graph of ReLu activation function

positive, then it gives x as output, which is positive again. ReLu is computationally cheaper than the sigmoid activation function. ReLu suffers from the dying ReLu problem.

1.4.3 SoftMax Activation Function

Softmax activation function is generally used in the final layer of a deep neural network model. A softmax activation function is known as a normalized exponential function. Softmax activation function takes a vector of n real numbers as input and normalizes the vector into n probabilities. The entries of the vector can be positive or negative, and their sum may not be equal to 1. However, when a softmax function is applied to all the entries of the vector, it gives outputs between 0 and 1, and their summation is up to 1; hence each entry is considered as the probabilities [4]. A softmax function can be defined by the equation (1.7).

$$f(x)_i = \frac{e_i^x}{\sum_{j=1}^{n} e_j^x} \tag{1.7}$$

where $i = 1, 2, ..., n$ and $x = (x_1, x_2, ..., x_n)$

A softmax activation function is usually used in a deep neural network for multi-class classification.

1.4.4 Binary Classification

Binary classification is one type of machine learning technique in which the data points are classified into two different classes. The decision boundary for binary classification can be both liner (hyperplane) or non-linear. In the binary classification an $N \times 1$ vector passes through the network and it gives output either "Yes" or "No," or numerically 1 or 0. A neural network model for binary classification can be defined as follows:

$f : \mathbb{R}^n \to \{0, 1\}$

$$f(x_1, x_2, ..., x_n) = \begin{cases} 1 & , if Y(x_1, x_2, ..., x_n) > b \\ 0 & , if Y(x_1, x_2, ..., x_n) <= b \end{cases}$$

1.4.5 Multi-class Classification

Multi-class classification is one kind of machine learning technique in which the data points are classified into more than two different classes. A nonlinear decision boundary can be drawn to classify a dataset into multiple classes [6]. In multi-class classification, each sample can only be labelled as one class. In our work, classification using features extracted from a set of Beowulf manuscript letter images, where each image may either be of an 'a,' or an 'ae,' or a 'b,' or a 'c,' and so on. Each image is one sample and is labelled as one of the 22 possible classes. Multi-class classification assumes that each sample is assigned to one and only one label, for example, one sample can not be both 'a' and 'b.'

CHAPTER 2
THE BEOWULF MANUSCRIPT

Beowulf is an Old English epic poem consisting of 3182 alliterative verses [19]. The author of this Beowulf epic poem is an anonymous poet of the Anglo-Saxon period. It is considered that this epic poem was composed between 975 CE and 1025 CE. This Old English epic poem has great importance among the scholars of both the humanities and sciences because of its ancient nature. Primarily, this Old English manuscript has a great importance in handwritten character recognition. The manuscript text extraction is manually performed by humanities scholars and takes a fairly large amount of time. However, this amount of time taken by manual recognition can be reduced considerably using automatic recognition approaches. We were inspired by the activity of performing digital image recognition to recognize handwritten letter images. The Beowulf Old English manuscript is a great source of historical electronic image documents. Character recognition from historic Old English electronic image documents is very difficult and time-consuming because there is no available dataset to apply necessary machine learning approaches to it. For our work we require a very large repository of handwritten single letter images. This is mostly because the current and most accurate machine learning techniques for handwritten character rely on large amounts of training data to create a viable model. To recognize handwrittern character from Old English historic image documents, we used the most updated approach, such as a deep neural network model for machine learning algorithms. Along with other machine learning techniques, we also used a convolutional neural network technique in our recognition work.

This Beowulf manuscript is a stunning source of historic image documents for machine learning enthusiasts to improve handwritten character recognition task. The electronic images of the Beowulf manuscript are available in the fourth edition of Electronic Beowulf, which is a free online version [10]. This free online version is an open source

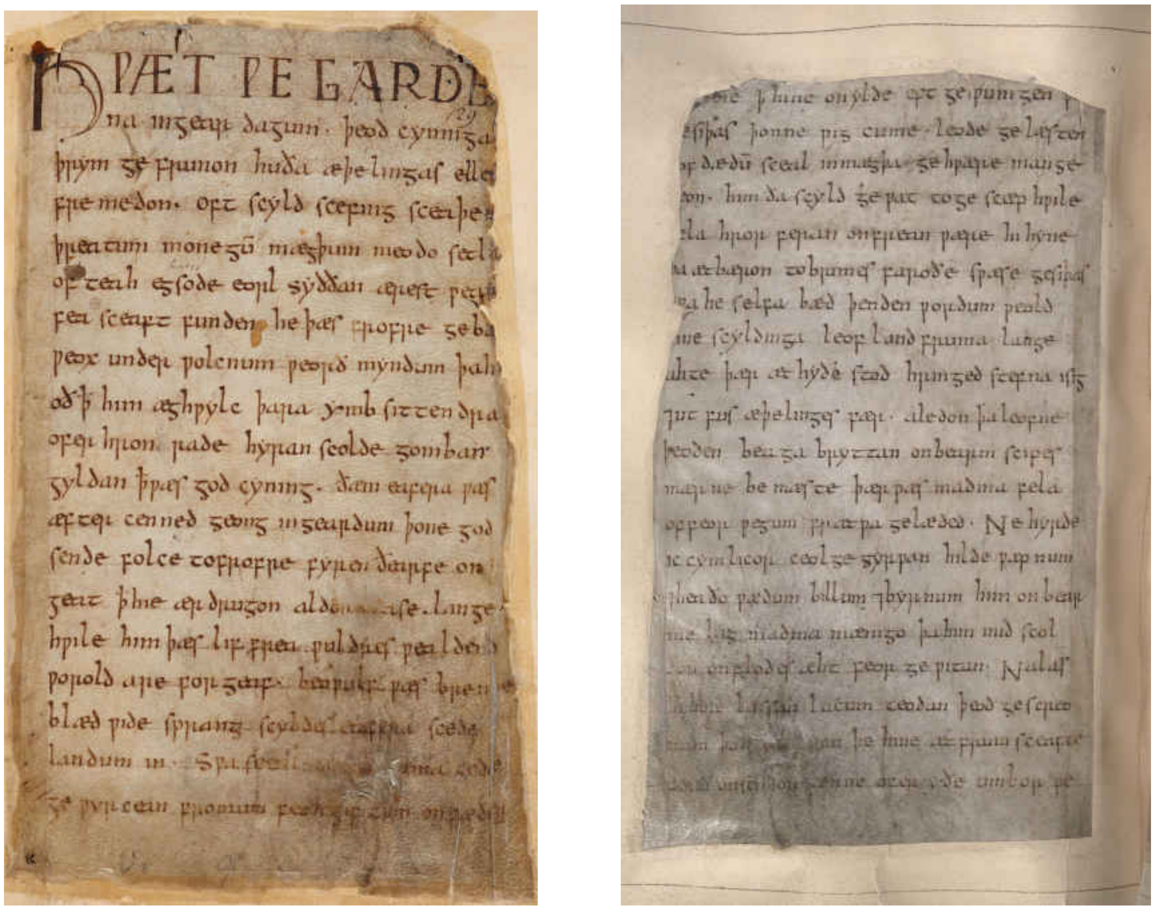


Figure 2.1: Beowulf electronic manuscript images

for all levels of researchers and students in the humanities, sciences, and engineering. This version has higher resolution images of 70 folios, over 130 ultraviolet images, and over 750 newly processed backlit images. Electronic Beowulf manuscript images are shown in Fig. 2.1.

2.1 Background of The Beowulf Manuscript

The Beowulf manuscript is a great source for machine learning and data science lovers, especially those working on handwritten character recognition and image detection. The Beowulf manuscript has 3182 alliterative lines, all of which are handwritten, and available in a free online electronic version. One of the main difficulties of image detection work is the scarcity of datasets. The Beowulf manuscript comes up with an enormous source of large electronic images. The Electronic Beowulf is a rich source for large electronic images of the Beowulf manuscript, but it does not have a huge repository of the single letter images of the manuscript. We need a dataset that consists of single letter images collected directly from the large electronic images of the Beowulf manuscript. We needed to crop those large electronic images manually to build a dataset of single-letter images of the Beowulf manuscript images. There were also difficulties and challenges in cropping the large images. For example, the size and texture of each handwritten letter of the same manuscript were not equal; there were many letters in which some parts of a letter overlapped with some parts of another letter, some letters had curly shapes that cover some parts of other letters and were difficult to crop, there were lots of noise (spots) on the original manuscript images, which changed the shape of the original letter images, etc. Some examples of such images are shown in Fig. 4.1. To build a dataset of single letter images, we needed to be very careful to crop letter images from large manuscript images. For examples, if we cropped a letter ‘d’ in the middle it looked like a ‘c’ and an ‘l,’ if we cropped ‘f,’ we may crop some part of another letter with ‘f,’ or we lost some part of ‘f’ for its curly shape in the upper part. There was also confusion between ‘c’ and ‘e’ because of noise (spots) was lain on the manuscript images. As the manuscript was written between the years 975 CE and 1025 CE, there were some difficulties understanding the letter itself too. Cropping the images was also very time-consuming. At first, we had to remove noise (spots) from large electronic images before cropping the single images. If we were not careful enough, then after cropping a letter image, we saw that this was not going to meet the purpose, because it lost the original shape of the letter that was found in the original version of the manuscript images. After cropping a letter image we also removed the noise from it, if there was still any noise on it. When we cropped the large electronic images, we tried to remove the maximum noise from the images. We cropped all the letter images from the large electronic images of the manuscript very carefully and tediously to build our dataset for training and to test our neural network model.

CHAPTER 3
CONVOLUTIONAL NEURAL NETWORK

Convolutional neural network (CNN) is referred to as a simple neural network in which a convolution operation is performed. The architecture of the CNN has been designed by replicating the organization of the human brain's visual cortex. CNN is also known as a deep neural network, which is used in deep learning for image recognition, image classification, video stream recognition, natural language processing, etc. A CNN architecture has input and out layers as well as several hidden layers, or in some cases, hundreds of hidden layers. The number of hidden layers of CNN may vary depending on the nature and complexity of the problem. A typical CNN is made up of a series of convolutional layers that convolve by componentwise matrix multiplication. A CNN architecture consists of convolutional layers, pooling layers, fully connected layers, and there are activation functions in each layer of the architecture. In the CNN architecture, backpropagation is necessary to adjust weights and biases accurately. CNN requires less pre-processing than a regular neural network. After performing convolution in a deep neural network architecture, CNN starts working as a regular neural network.

3.1 How does Convolutional Neural Network Work

CNN is a type of vanilla neural network. CNN transforms input images into a reduced form that is more convenient to process, and at the same time, it retains the most important features of the original images, which can accurately summarize the original input image data. The act of extracting the most important and relevant information, while reducing the redundant and irrelevant information from the input data is known as feature extraction. For this purpose, a filter is passed over the original images, which conducts componentwise matrix multiplication over a subsection of the pixel matrix of the original images; it repeats throughout subsets until it has considered all the subsets [17]. These matrix multiplications between the pixel matrix of the images and the filter matrix are performed according to the formula (3.1).

$$G[m,n] = (f * g)[m,n] = \sum_j \sum_k h[j,k] f[m-j, n-k] \tag{3.1}$$

After having done feature extraction by kernel convolution, we need to perform dimensionality reduction, which is done in the pooling layers. In the pooling layers, we need to reduce the spatial size of the convoluted features, which is known as *dimensionality reduction*. After performing the pooling layers, information about the image data is squeezed enough to pass through a regular neural network for further processing. This compressed information about image data is obtained by kernel convolution and pooling. After the pooling layers, we need to flatten the information by converting the matrix of pixels data into a vector of pixels data. Therefore, this information of the input images is reduced enough to pass through the vanilla (regular) neural network, in other words, after having done proper kernel convolution and pooling, CNN returns to a vanilla (regular) neural network to process the information, and produce desired output with optimal computational burden. To get the best result from the model, the vanilla neural network needs to adjust its weights and biases in each layer of the network, and selects the best possible set of weights and biases. In this case, cost function plays an important role in selecting the best possible set of weights and biases. Cost function calculates how far away a particular solution is from the best possible solution. Then the information received from the cost function is passed onto the optimization function, which calculates the values of a new set of weights and biases. Some terminology related to CNN is given in the subsequent subsections that helps to have more insight into the mechanism of CNN.

3.1.1 Kernel and Filter

In CNN, there is a subtle difference between kernel and filter. The term filter referred to a three-dimensional structure of multiple kernels piled together one on top of the other, while the two-dimensional filter is referred to as the kernel [1]. In other words, a filter is a collection of kernels.

3.1.2 Kernel Convolution

The kernel or filter passes over the original images aiming to capture the most relevant and crucial features of the images, and at the same time it allows the redundant features to be eradicated. These passing of filters over the original pixel matrix of the images are known as the kernel convolution. Componentwise matrix multiplication calculates the extracted feature by kernel convolution. In this layer, information is constricted enough and the analysis of the image data requires less computational cost by the neural network model.

3.1.3 Strides

The stride of the filters is a very important feature, which is applied to the input pixel matrix of the images to reduce the dimension (size) of the output pixel matrix. Stride is the number of pixels that shifts over (or jumps over) the pixel matrix of the input images. Stride takes control over how the filter convolves around input images. If the stride is 1, the filter moves 1 pixel around the pixel matrix of the input image by shifting (or jumping) one pixel at a time. If the stride is 2, the filter shifts (or jumps) 2 pixels at a time, and so on and so forth [3]. If we select a large stride, we are supposed to get a relatively squeezed feature matrix as the output of the input images. Usually, strides need to be selected to give an integer volume of the output pixel matrix rather than a fraction of the pixel matrix.

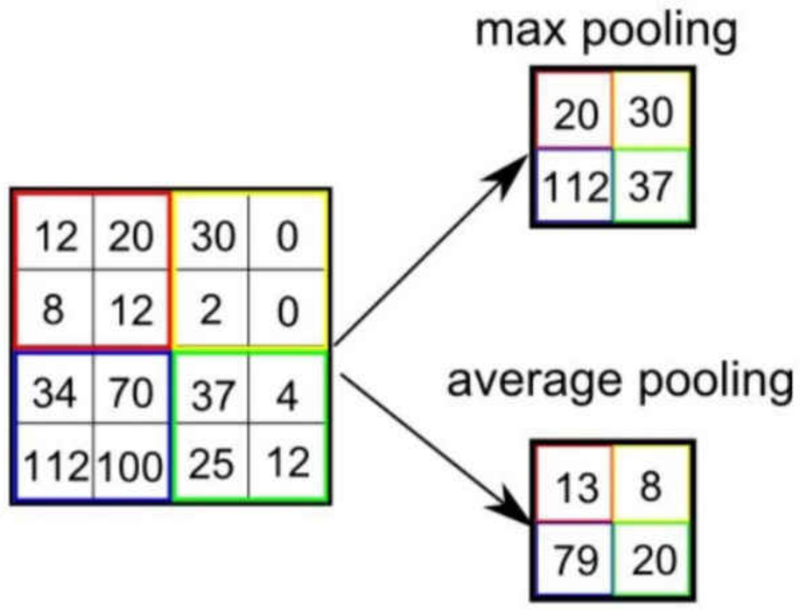


Figure 3.1: Max pooling and average pooling

3.2 Pooling

After the convolutional layer, a kernel or a filter of pixel matrix again passes over all the subsets of the images' convolved features to collect the more squeezed features. The pooling layer aims to reduce the spatial size of the convoluted features. This reduction is known as *dimensionality reduction*, which decreases the computational expenses of performing analysis on the dataset and gives more room to the method to be robust. Two types of pooling kernels are generally used for this purpose, namely max-pooling and average pooling. Max-pooling extracts the maximum pixel's values of the subsets, while average pooling extracts the subsets' average values. There is another type of pooling, which is called sum pooling, and it retains the total value of the subsets. The operation of max-pooling and average pooling is shown in Fig. 3.1 [17].

3.2.1 Flatten

Flatten is very important because after this layer, our CNN model returns to the form of a vanilla (regular) neural network model. Flatten is the last layer of CNN. In this layer, compressed information comes from the pooling layer as pixel matrices. Flatten is done by converting matrices of pixels into vectors of pixels. After having done flatten, information about the images passes through the vanilla (regular) neural network model to process and get the desired output.

3.3 The LeNet-CNN Architecture Implemented in Our Work

At first, we need to import necessary packages such as Conv2D, sequential, maxpooling2D, activation, flatten, dense, etc. After calling the LeNet architecture, we initialize the model and input shapes. To use channels, we need to update the input shapes. After updating the input shapes, we need to initiate the first set of the convolutional layers in which the network learns 32 filters with 5×5 kernel convolution. In the first set of convolutional layers, we need to incorporate ReLu activation functions. After kernel convolution, we perform maxpooling with a pool size of 2×2, and pass the window by a stride of 2×2, which reduces the height and width of the images by half, and retains the most important features of the images. Next, we initiate the second set of convolutional layers in which the network learns 64 filters with 5×5 kernel convolution. In the second set of convolutional layers, again, we need to incorporate ReLu activation functions. After kernel convolution, we perform maxpooling with the same pool size of 2×2, and pass the windows by a stride of 2×2, which reduces the height and width of the images by half, and retains the most important features of the images [9]. CNN is a variation of vanilla (regular) neural networks. CNN has some advantages over vanilla (regular) neural networks, especially in image processing. We all know that images are made up of pixels. If the images are big with many

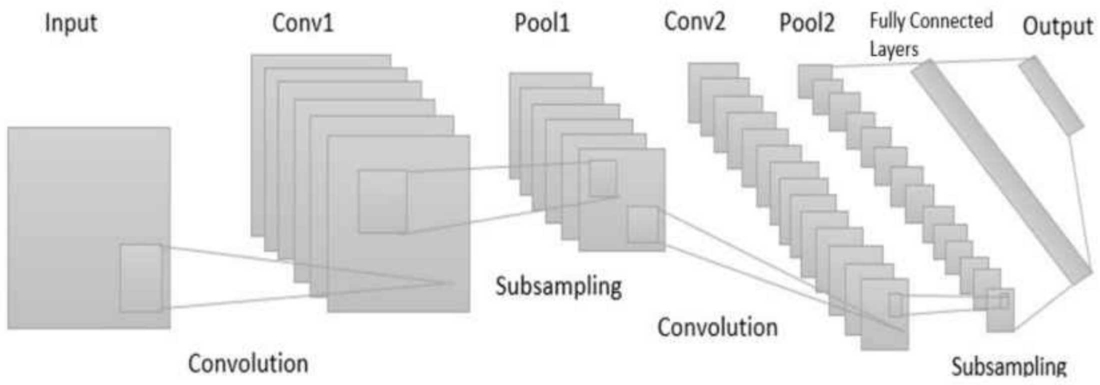


Figure 3.2: LeNet-CNN architecture

color channels, the classification task becomes computationally expensive, and infeasible to train the model. CNN transforms the images to a computationally more manageable form, and retains the most important features. We add a fully connected layer of 256 neurons by dense function, each neuron in this layer is connected with other neurons, and each neuron has a ReLu activation function in it. In the top layer of the network architecture, we use a softmax classifier, which calculates the probabilities for each category, and returns the label with the highest probability of a specific class. LeNet architecture used in our work is shown in Fig. 3.2

CHAPTER 4

MANUSCRIPT CHARACTER RECOGNITION USING CONVOLUTIONAL NEURAL NETWORKS

In this chapter we give an exact formulation of our problem, we describe the data we process, present the implementation details of the CNN model we use for character image recognition, and define how we measure our model's accuracy.

4.1 Character Images and the Problem Formulation

The problem we tackle in this work can be informally described as performing text retrieval from the Beowulf manuscript images [10] like the one presented in Fig. 2.1. For this purpose, we collected individual character images (rectangular areas from folio images like the one in Fig. 2.1) using a simple selection tool. Such individual character images had various dimensions, even for multiple images selected for the same character. This is the result of extracting character images from a manuscript, where each letter was produced manually, as opposed to printed letters, which would have equal dimensions for the same character.

We denote the set of all characters that appear in the manuscript we analyze by $\mathcal{A}$ (the alphabet):

$$\mathcal{A} = \{a, æ, b, c, d, e, ð, f, g, h, i, l, m, n, o, p, r, s, t, þ, u, w\}$$

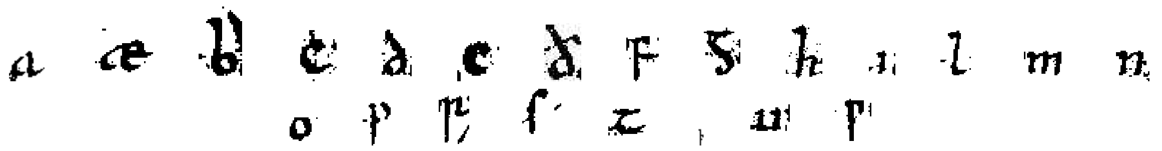


Figure 4.1: Sample images for each letter in the alphabet

One sample for each letter in $\mathcal{A}$ is shown in Figure 4.1. We collected 20 character images for each letter in $\mathcal{A}$, for a total of $20 \times |\mathcal{A}| = 20 \times 22 = 440$ images. We denote the set of all these images by $\mathcal{I}$. We denote the set of all such character images in the whole Beowulf manuscript [10] by $\overline{\mathcal{I}}$. Each character image $i \in \overline{\mathcal{I}}$ has an associated alphabet letter in $\mathcal{A}$, and we denote by $label(i)$ the corresponding letter of image i.

We define the character image recognition function as follows.

Definition 4. [Character image recognition function] A manuscript character image recognition function is a function $f : \overline{\mathcal{I}} \to \mathcal{A}$ such that:

$$f(i) = label(i)$$

That is, a function that correctly identifies the label of each character image.

Our goal is to create a practical model that approximates the character image recognition function. Note that, a function f as in definition 4 can be exactly found, if the whole set of character images $\overline{\mathcal{I}}$ are collected, and each image is correctly labeled. However, this would be impractical, as it would take a considerable amount of work. Instead, the practical approach for such a task relies on working with a much smaller subset of $\overline{\mathcal{I}}$, such as $\mathcal{I}$, for which all labels are known to create a model capable of recognizing most of the images in $\overline{\mathcal{I}}$. The problem of manuscript character image recognition can be formulated as follows:

> Given a set of images and their labels $\{(i, l) \mid i \in \mathcal{I},\ l = f(i)\}$, find a model $\mathcal{C} : \overline{\mathcal{I}} \to \mathcal{A}$ that approximate the character image recognition function, $\mathcal{C}(i) \approx f(i),\ \forall i \in \overline{\mathcal{I}}$.

There are two practical questions regarding the above formulation. The first is how to create such a model $\mathcal{C}$. The second is to determine how well $\mathcal{C}()$ approximates $f()$. We will address these questions in the subsequent sections.

4.2 The CNN Model Implementation

In early work [18] a model for solving the character image recognition problem was proposed based on a regular neural network model. The model can be best explained by the following commutative diagram:

$$\begin{tikzcd}
\mathcal{I} \arrow[r, "norm"] \arrow[rrd, "M"'] & \mathbb{R}^{p\times q} \arrow[r, "vec"] & R^{pq} \arrow[d, "NN"] \\
 & & \mathcal{A}
\end{tikzcd}$$

The model $\mathcal{I} \xrightarrow{M} \mathcal{A}$ is implemented by performing the following steps:

1. Normalization $\mathcal{I} \xrightarrow{norm} \mathbb{R}^{p\times q}$: each image is normalized to a matrix of fixed dimension $p \times q$.

2. Vectorization $\mathbb{R}^{p\times q} \xrightarrow{vec} \mathbb{R}^{pq}$: each matrix is linearized (column by column, row by row, or other techniques) as a vector.

3. Neural Network model (NN) $\mathbb{R}^{pq} \xrightarrow{NN} \mathcal{A}$: a neural network model is used to map each vector into an alphabet letter.

The model we propose in this work, $\mathcal{I} \xrightarrow{\mathcal{C}} \mathcal{A}$, can be explained by the following commutative diagram:

$$\begin{tikzcd}
\mathcal{I} \arrow[r, "norm"] \arrow[rrrd, "\mathcal{C}"'] & \mathbb{R}^{20\times 20\times 1} \arrow[r, "conv"] & R^{p\times q} \arrow[r, "vec"] & R^{pq} \arrow[d, "NN"] \\
 & & & \mathcal{A}
\end{tikzcd}$$

The model performs a normalization, as before, but then a convolution step is performed on the normalized image's matrix. The purpose of convolution is to *extract image's features* (like edges), then vectorize these features, and finally apply a neural network model. The key difference: the previous work's model, M classifies the image's pixels values, whereas the new model $\mathcal{C}$, we propose, classifies the image's features.

The implementation of the model is illustrated in Fig. 4.2. The model uses two convolution steps, of 32 and 64 filters, respectively. A traditional neural network model has been applied to get output from the convolutional layers by fully-connected layers with 256 neurons. Finally, the 22-dimensional output (indicating likelihoods for each of the 22 characters in the alphabet) is processed through the “softmax" function, which selects the highest probability, and returns the corresponding image character.

The parameters of the model with the architecture, are then determined through an optimization process for the loss function, as described in chapter 3. The optimization process of finding the best parameters (learning) of the model from the given set of images and labels $\{(i, l) \mid i \in \mathcal{I},\ l = f(i)\}$ is called *training*. As we will describe in the next section, the typical training process is performed on a subset of the images set $\mathcal{I}$.

Some terminologies are given below, which are useful for understanding the experimental results in chapter 5.

Definition 5. [Epoch] An epoch is one complete presentation of training data to the model.

Definition 6. [Batch size] The batch size is a subset of the training data on which the model's parameters adjustment is performed during the optimization process.

4.3 The Accuracy of the Model

The exact accuracy of the model is difficult to compute in practice. It would require collecting all character images in the manuscript, which is impractical. Instead, *we estimate* the model accuracy, as follows.

The following steps of a *trial* are performed to compute the accuracy of the trial, acc_t:

1. The set of images $\mathcal{I}$ is partitioned in two disjoint subsets, a training set and a testing set: $\mathcal{I} = \mathcal{I}_R \cup I_T$ (a typical partitioning of the dataset is 80% and 20% for the training, and testing sets, respectively).

2. The model $\mathcal{C}$ is trained (optimized) for a number of epochs on all images (and their labels) in $\mathcal{I}_R$.

3. The accuracy of the trial is estimated as:

$$acc_t(\mathcal{C}) = \sum_{i \in \mathcal{I}_T} (1 - eq(f(i), \mathcal{C}(i)))$$

   where the equality operator $eq()$ is defined as:

$$eq(x, y) = \begin{cases} 1, & \text{if } x = y \\ 0, & \text{if } x \neq y \end{cases}$$

The average accuracy of the model, acc_{model}, is defined as the average trials' accuracy over a number of trials N:

$$acc_{model}(\mathcal{C}) = \frac{1}{N} \sum_{t=1}^{N} acc_t(\mathcal{C})$$

```
model = Sequential()
    inputShape = (20, 20, 1)
    # first set of CONV => RELU => POOL layers
    model.add(Conv2D(32, (5, 5), padding="same,"
            input_shape=inputShape))
    model.add(Activation("relu"))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

    # second set of CONV => RELU => POOL layers
    model.add(Conv2D(64, (5, 5), padding="same"))
    model.add(Activation("relu"))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

    # first (and only) set of FC => RELU layers
    model.add(Flatten())
    model.add(Dense(256))
    model.add(Activation("relu"))

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

Figure 4.2: Implementation of the CNN model for character image recognition

CHAPTER 5

IMPLEMENTATION AND EXPERIMENTAL RESULTS

Convolutional neural networks models (CNNs models) are powerful models for image classification, and in general, for identifying patterns in images. These models are de facto models for machine learning applications involving images. A very well-known disadvantage of CNN models is that they require a very big dataset for training the models. This is a natural consequence of the complexity of these models, which typically involves thousands of parameters, even for a relatively small model. The MNIST [11] dataset, for instance, contains 60,000 hand-written images of numbers for training and 10,000 images for testing the models. This was a huge obstacle we needed to overcome in our research. On the one hand, we wanted to collect a reasonably small set of character images (which corresponds to a practical usage scenario), and on the other hand, we wanted a higher rate of accuracy in our recognition work. We used the Electronic Beowulf manuscript images [10, 19] for our experiments with an alphabet consisting of the following 22 characters:

a, æ, b, c, d, e, ð, f, g, h, i, l, m, n, o, p, r, s, t, þ, u, w

For each alphabet character, we collected 20 images (of various sizes). One sample for each character image is shown in Fig. 5.1. A quick experiment of these hand-written character images reveals the inherent difficulties of our character recognition task: not only similar characters (such as ‘u’ and ‘n,’ ‘p’ and ‘r,’ etc.), but also artifacts of the manuscript itself (ink spots, manuscript damages, etc.). Our experimental results show, an out-of-shelf CNN model completely failed, as the set of images for training the model was too small.

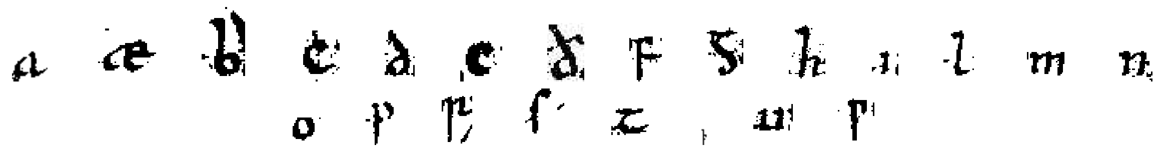


Figure 5.1: Sample images of characters used in experiments

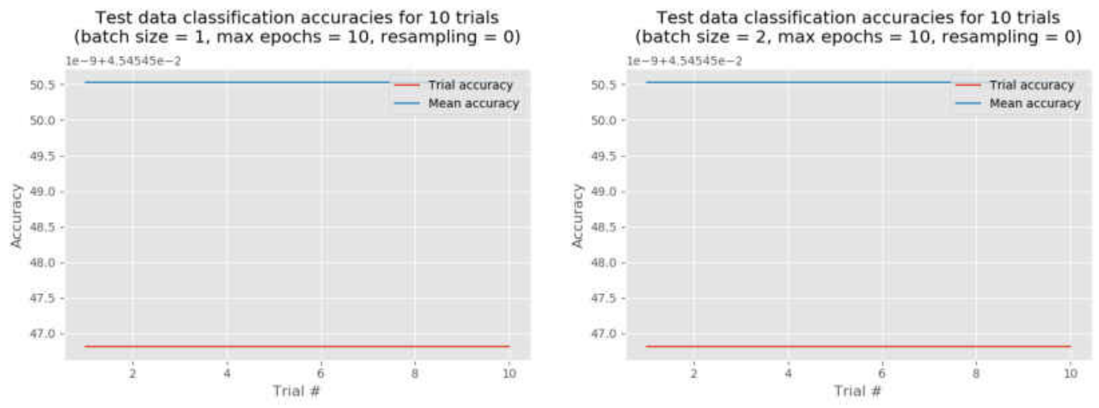


Figure 5.2: Experimental results for accuracies of the vanilla CNN model

We then used a simple technique, resampling, to accommodate the insufficient training data, and the results were overwhelming.

We ran extensive experiments which we organized into two categories, with many subcategories, as follows:

1. No resampling, with the training batch sizes: 1 and 2. We called these models the "vanilla CNN models."

2. Resampling: 1, 2, and 3 times. For the resampling experiments, we trained the models with various batch sizes: 1, 2, 4, 8, and 12. We called these models the "CNN models with resampling."

For all the experiments, we used a personal computer (PC) equipped with an Intel Core i7-4770 CPU @3.40GH. The experiments and their results were reported in the subsequent sections, with a summary of all results in table 5.1. The Python code for producing results was included in Appendix A.

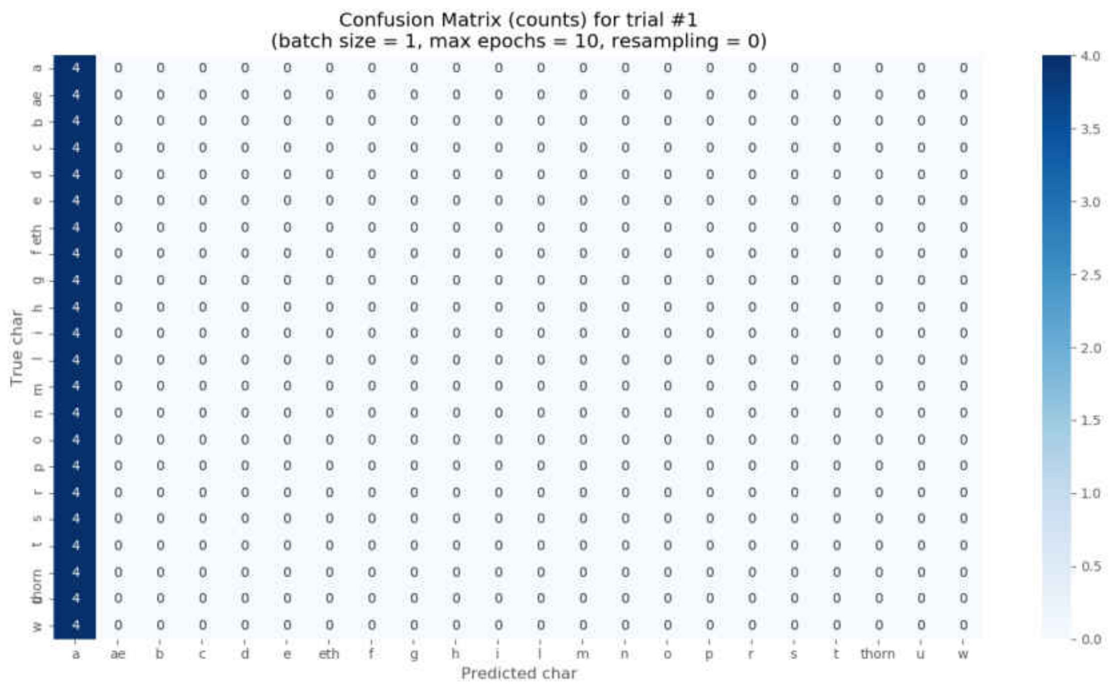


Figure 5.3: The confusion matrix for test character recognition (vanilla CNN model)

5.1 The Vanilla CNN Models

For this experiment, we created a CNN model with two convolutional layers, as described in chapter 4. We trained the model with randomly chosen 16 character images for each character in the alphabet (16 images x 22 letters = 352 images), and we trained for 10 epochs. We reserved 4 character images (for a total of 4 images x 22 letters = 88 images) for testing the model. We ran 10 trials (for each trial a distinct, randomly chosen training set was used) twice: for training batch sizes 1 and 2, respectively.

The results were consistently and completely in error: the model did not work with little training data. Fig. 5.2 shows that the model does not train at all for too little data and too many model parameters. For both the cases, (for batch sizes 1 and 2, respectively) the accuracy is unchanged, which is approximately 4.5%. The results for batch sizes 1 and 2 are summarized in table 5.1.

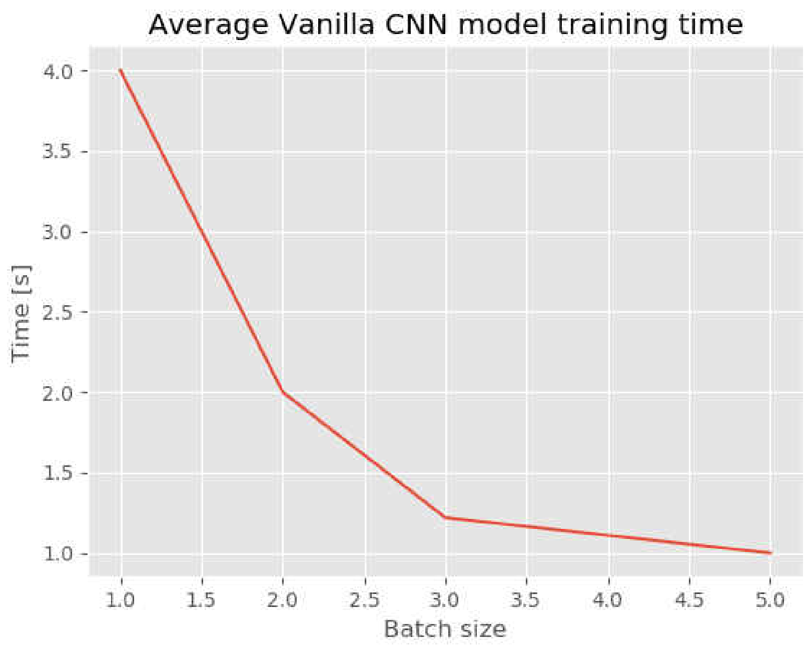


Figure 5.4: The average training time (over 10 epochs and 10 trials) for different batch sizes

The confusion matrix in Fig. 5.3 shows that the model completely fails character recognition task: the model responded to each input character with an 'a,' the first character in the alphabet (all predictions are in the first column of the confusion matrix). It is, therefore, no surprise that the "accuracy" of the vanilla CNN model is $(4/(4 \times 22)) \times 100 \approx 4.5\%$ (there are four 'a's in the whole test set of 22×4 character images).

Fig.5.4 shows that as batch size increases, training time taken for each epoch decreases, which is the only benefit of the vanilla CNN model. However, as expected and we will see later on, larger batch sizes require more training (epochs), or else accuracy will be affected.

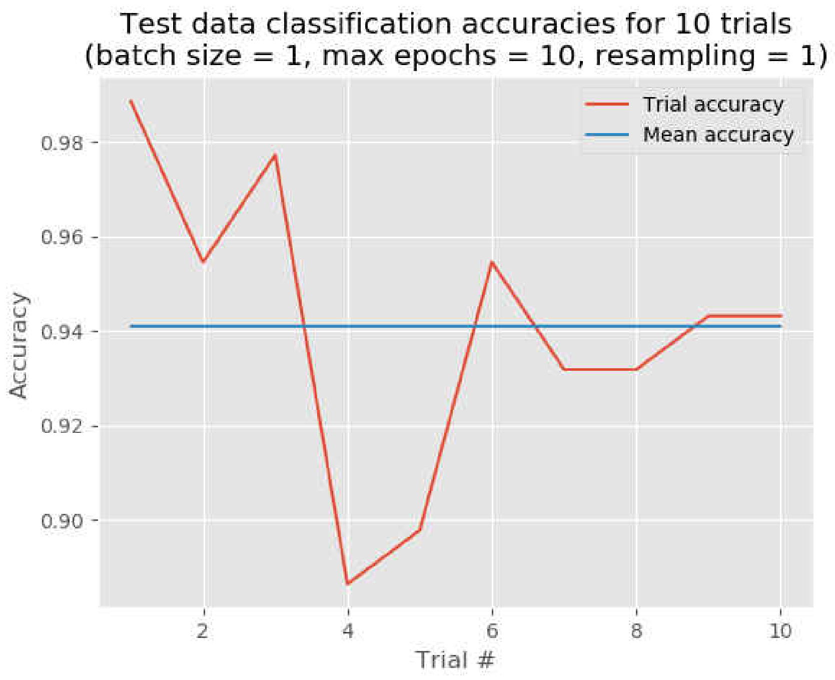


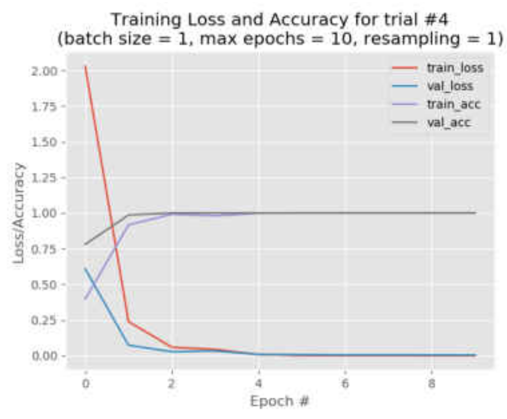
Figure 5.5: Test data classification accuracies of a CNN model with resampling

5.2 The CNN Model with Resampling

We organized these experiments similarly as before. This time we used resampling (by expanding the original image dataset one, two, and three times), and we trained our model for 10 epochs each time. We ran every experiment for 10 trials. For each trial, a distinct, randomly chosen training set was used for training the model with batch sizes 1, 2, 4, 8, and 12, respectively.

5.2.1 CNN Model with Resampling One Time

The first breakthrough in producing practical classification results using a reduced dataset for training happened at the very first time we used resampling. Fig. 5.5 shows the model's test accuracy results over the 10 trials with batch size one. Indeed, it is a big improvement from the vanilla CNN models that a CNN model using resampling produces



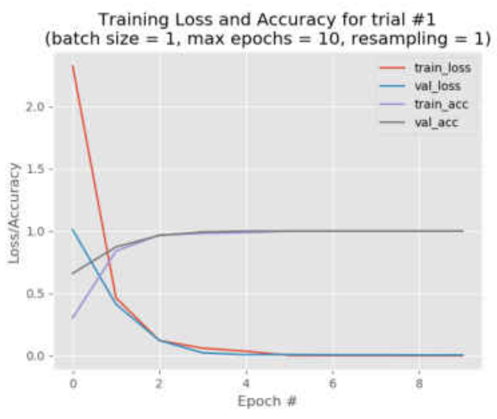


Figure 5.6: Training CNN models with resampling (the worst and best accuracies case scenarios)

a good result (the average accuracy for all trials hovers just above 94%) even on a small training dataset. Fig. 5.2 shows the typical evolution of the model's training accuracy, which means it develops asymptotically from low initial accuracy to the highest attainable accuracy of the model. This evolution is consistently similar over all trials, the trails with the best accuracies are in the left and the trials with the best accuracies are on the right of the Fig. 5.6. Another conclusion stemming from Fig. 5.6, is that the model (in each of the 10 trials) quickly evolves to attain maximum accuracy (in about 5 epochs). This is a direct consequence that, even by using resampling to increase the size of training data, the model is trained on the same small *distinct* set of images.

The confusion matrices in Fig. 5.7 and Fig. 5.8 for the worst and best accuracies (overall 10 trials), respectively, show the models' misclassification of the characters. Fig. 5.8 shows that, in the best-case-scenario, one 'c' is misclassified as 'e' (but not the other way around). In the worst-case-scenario in Fig. 5.7, one 'b' is misclassified as 'i,' two 'ð' characters (Old English characters, abbreviated as "eth" in the figure) are misclassified as 'd,'s two 'n's as 'u,'s etc.

We continued the experiments by running 10 trials, but slowly increasing the batch

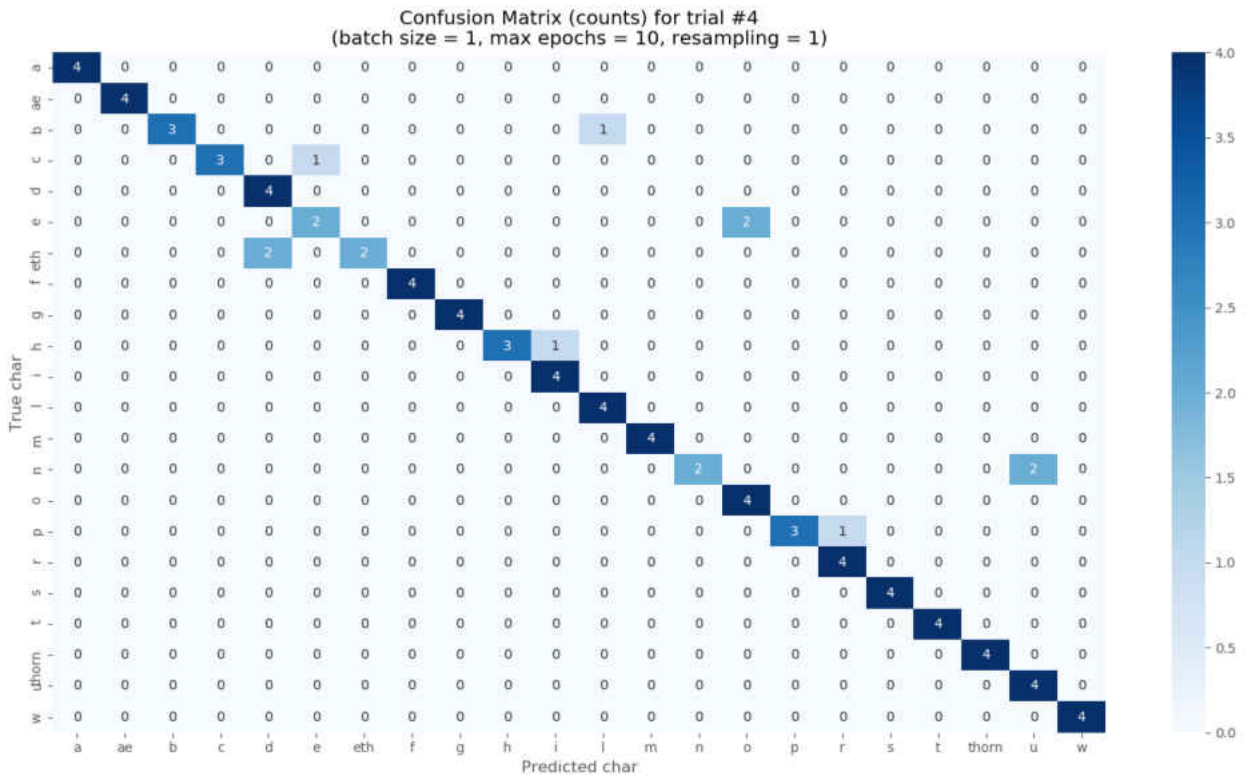


Figure 5.7: The confusion matrix for test character recognition (CNN model with resampling-1, batch size 1, the lowest model accuracy)

sizes to 2, 4, 8, and 12. The main purpose of these experiments was to observe the accuracy outcome for larger batch sizes. As expected, larger batch sizes make the training of the model faster, at the expense of slightly decreasing accuracies. We observed an overall decrease in accuracy to 92%, but the model training time was significantly reduced (Fig. 5.9), which (all measurements for different batch sizes) are summarized in table 5.1 (rows 3 to 7).

5.2.2 CNN Model with Resampling Two Times

For this set of experiments, we used the same training data set as before, but we resampled twice. We subsequently ran five experiments (for five batch size values: 1, 2, 4,

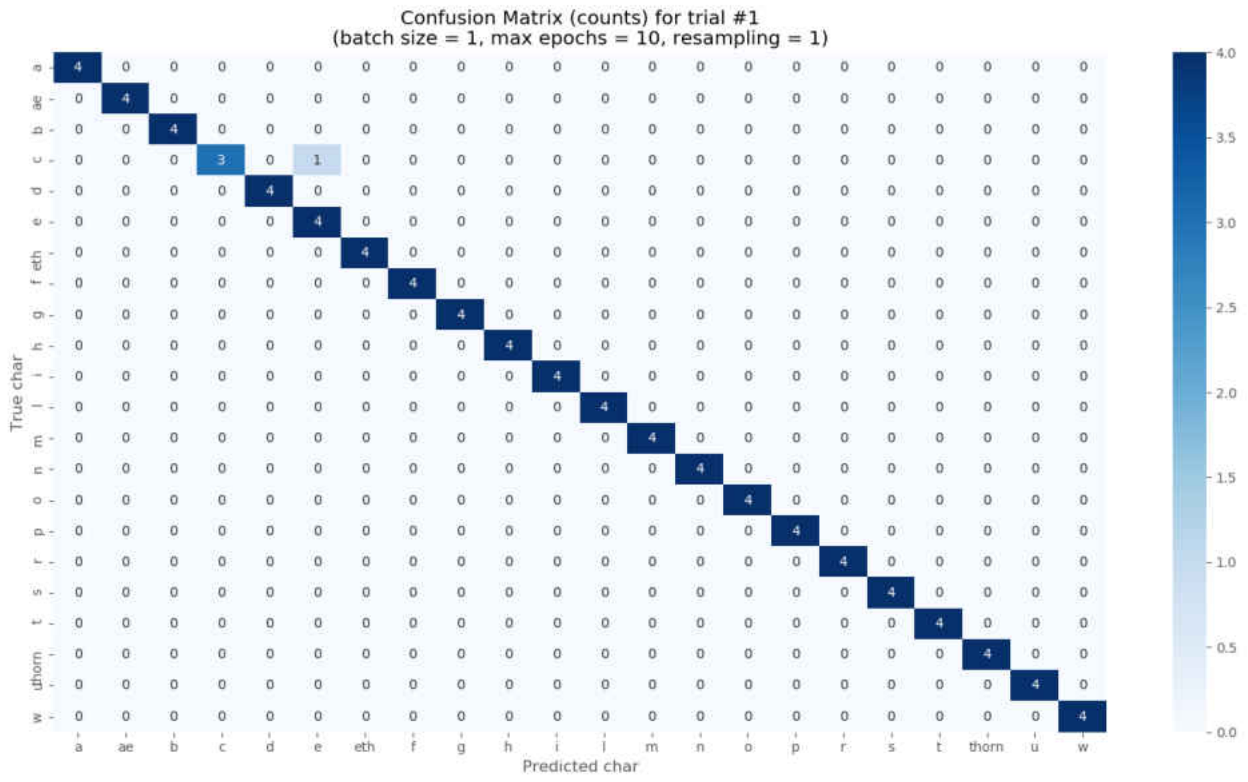


Figure 5.8: The confusion matrix for test character recognition (CNN model with resampling-1, batch size 1, the best model accuracy)

8, and 12), with 10 experimental trials for each batch size value.

As before, the best average accuracy (94%) was obtained for batch size 1 (over 10 trials). Fig. 5.10 shows the accuracies for each of the trials (batch size 1), and Fig. 5.11 shows the training processes for the worst and best-case scenarios of the 10 trials. One quick conclusion is that the best average accuracy does not improve with more resampling. The models train slightly faster in terms of the number of epochs (Fig. 5.11 vs Fig. 5.6). However, there is no gain in terms of the overall training time since resampling two times increases the size of the training data. In comparison to the case of one-time resampling, *the training time* for the models with resampling two-times decreases quickly (Fig. 5.12). Moreover, we produced the corresponding confusion matrices (for the worst and best-case

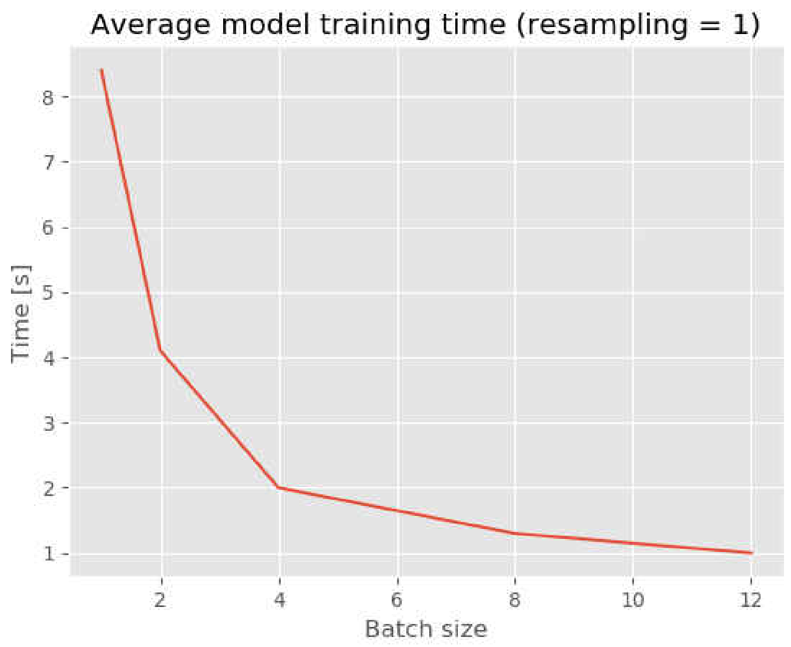


Figure 5.9: The average training time (over 10 epochs and 10 trials) for different batch sizes of models with resampling one time

scenarios with resampling two times) and did not exhibit differences from the model with resampling one-time. The average accuracies and training times for all batch sizes for this experiment are summarized in table 5.1 (rows 8 to 12).

5.2.3 CNN Model with Resampling Three Times

For these set of experiments, we resampled three times. As before, we ran five experiments (for five batch size values: 1, 2, 4, 8, and 12), with 10 experimental trials for each batch size value.

The best average accuracy over 10 trials was obtained for batch size 1. Accuracies for each trial for batch size 1 are plotted in Fig. 5.13. The model with resampling three-times displays similar average accuracies as the models with one or two-times resampling. Also, the standard deviations for the three models are similar (table 5.1 shows the summary of

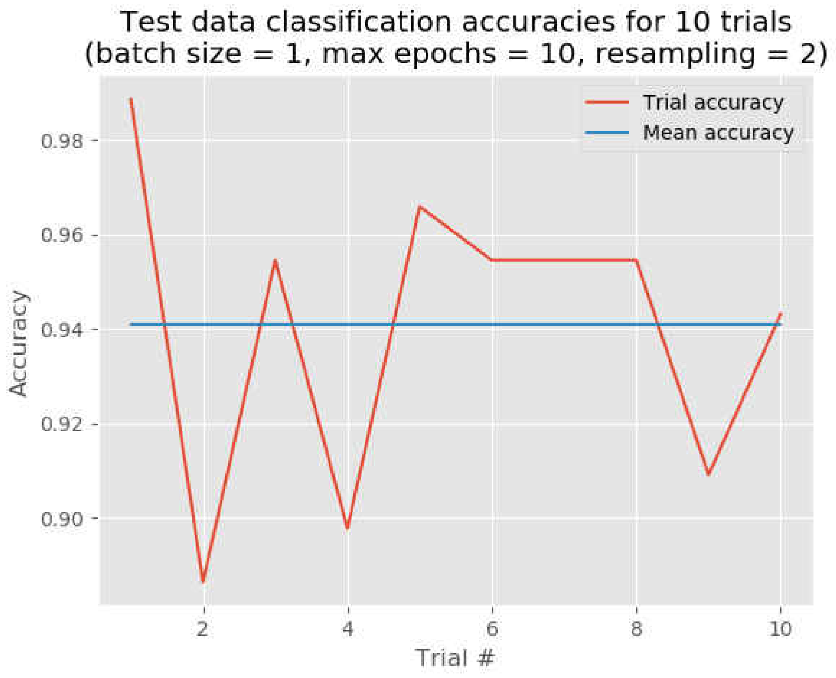


Figure 5.10: Experimental results for accuracies of a CNN model with resampling two times

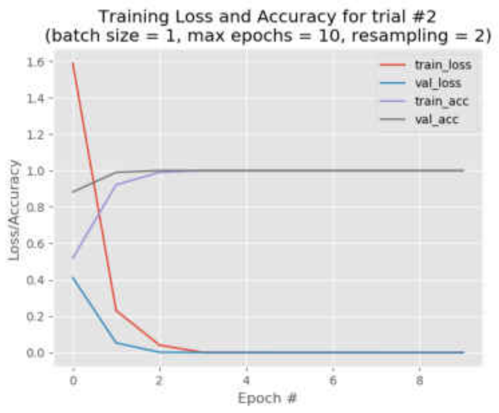
the results for models with resampling three-times in rows 13 to 17). The model's training time is larger, due to the larger training dataset, but the training time vs. the batch size curve (Fig. 5.15) has a similar evolution as before. The worst and best-case scenarios for accuracies (batch size 1) in Fig. 5.14 show the fast convergence of the model (just a few epochs) during training.

5.3 Summary of the Experimental Results

For each experiment, we ran 10 trials (as described above) and recorded the average accuracy, the standard deviation for accuracies, and the average time for training the models in each experiment. The summary of these results is presented in table 5.1. The table shows the number of resampling performed for each experiment (with zero indicating no resampling, that is, the vanilla CNN model), the batch size, average accuracy (over 10 trials

No	Resampling	Batch size	Mean accuracy	SD	Average time [sec]
1	0	1	0.47	0	4
2	0	2	0.47	0	2
3	1	1	0.9409	0.030	8.4
4	1	2	0.9318	0.028	4.1
5	1	4	0.9284	0.036	2
6	1	8	0.9216	0.033	1.3
7	1	12	0.9193	0.026	1
8	2	1	0.9409	0.031	14.6
9	2	2	0.9364	0.030	8.2
10	2	4	0.9318	0.037	4.6
11	2	8	0.9272	0.032	3
12	2	12	0.9307	0.035	2
13	3	1	0.9477	0.028	26.8
14	3	2	0.9409	0.020	14.9
15	3	4	0.9284	0.029	8
16	3	8	0.9250	0.026	5
17	3	12	0.9284	0.035	4

Table 5.1: Summary of the experimental results



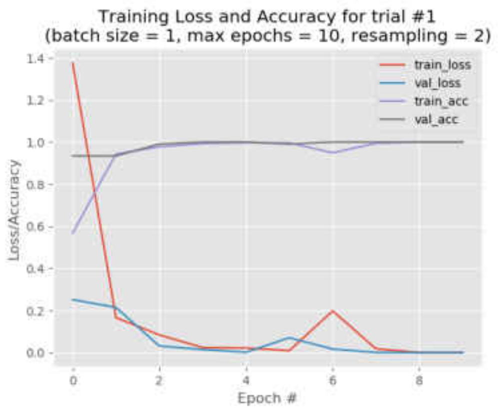


Figure 5.11: Training CNN models with resampling (the worst and best accuracies case scenarios for resampling two times)

per each batch size and per each resampling), the standard deviation over all 10 trials, and the average training time *per each training epoch*. We trained each model with 10 epochs.

These results clearly show that the CNN models trained with insufficient data (with zero resampling) produce no results (experiments 1 and 2). However, the outcomes dramatically change when resampling is used. Experiments 3 – 17 show that CNN models can be trained using little data combined with a resampling technique to produce excellent results.

However, the results show that over-resampling does not produce significant improvement in accuracy. If we compare the best average accuracies for one, two, and three times resampling (rows 3, 8, and 13, respectively), we see that more resampling does not produce an increase in accuracy already obtained with resampling one time. One can conclude that resampling makes a relatively complicated model (with many parameters) work with a small training dataset (which otherwise would not work, as one can conclude from rows 1 and 2), but it cannot help improve accuracy over a certain threshold. We also notice that increasing the batch size during training (which always yields better training time and often increases accuracy) produces a slight decrease in the model's accuracy. This is a side effect of resampling, with a larger batch size, the chance to have identical samples increases and

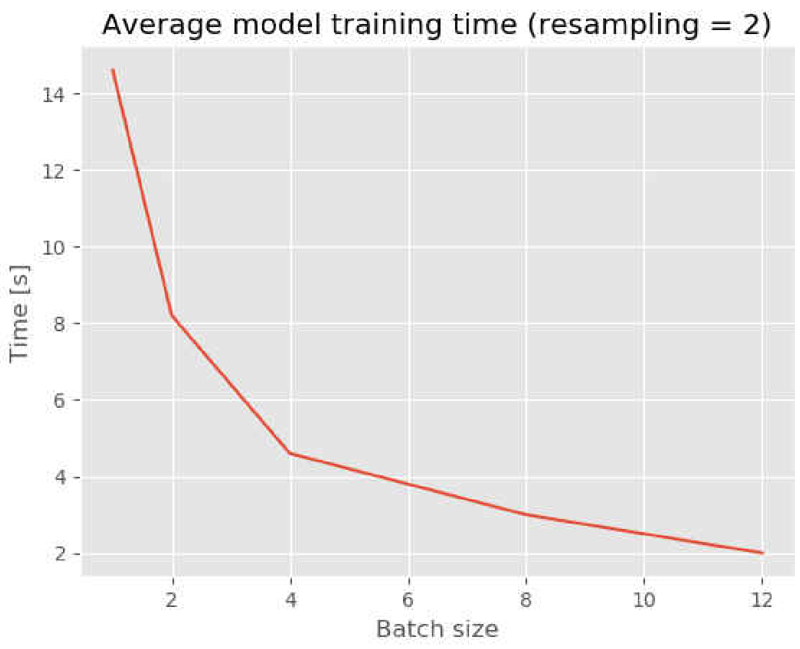


Figure 5.12: The average training time (over 10 epochs and 10 trials) for different batch sizes of models with resampling two times

identical data samples in the same batch will not help increase accuracy.

The ‘out of the box’ CNN model does not work on the dataset we have, because the CNNs rely on optimizing a high non-convex function, and a huge number of model parameters. As a result, the model quickly gets under-fitted because of too little data, which yields a complete failure in our recognition work. We overcome this problem, using a resampling technique, and it significantly improves our results with an accuracy of 93.19%.

In case of resampling one, if we compare mean accuracy and time taken for batch size one and batch size twelve (rows 3 and 7), we observe that the mean accuracy drops by 2.35%, but the training time for each epoch drops by 740%. There is also a drop in the mean accuracy from batch size 2 to batch size 8 (rows 4 and 6) by 1.11%, but the training time for each epoch drops by 215%. For resampling two, time taken for each epoch from batch size 1 to batch size 12 (rows 8 and 12) drops by 630%, and the mean accuracy drops

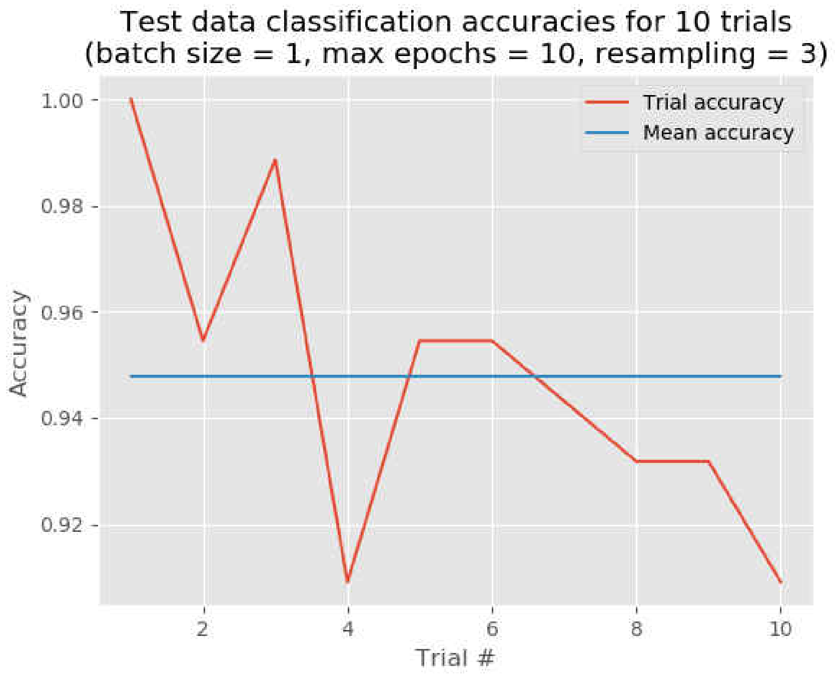


Figure 5.13: Experimental results for accuracies of a CNN model with resampling three times

by 1.1%. Similarly, for resampling 3, time taken for each epoch from batch size 1 to batch size 12 (rows 13 and 17) drops by 570%, and the mean accuracy drops by 2.08%. Hence, we can conclude that with the increase of batch size, the model's mean accuracy has a negligible decrease (in other words, it does not affect model's mean accuracy severely), but the time taken to complete each epoch improves geometrically (i.e., with the increase of batch size, our machine takes considerably very less time to complete each epoch).

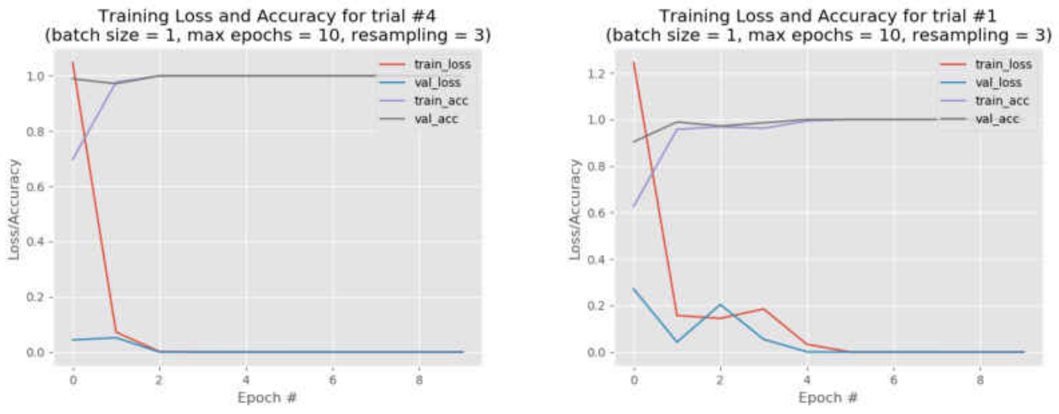


Figure 5.14: Training CNN models with resampling (the worst and best accuracies case scenarios for resampling three times)

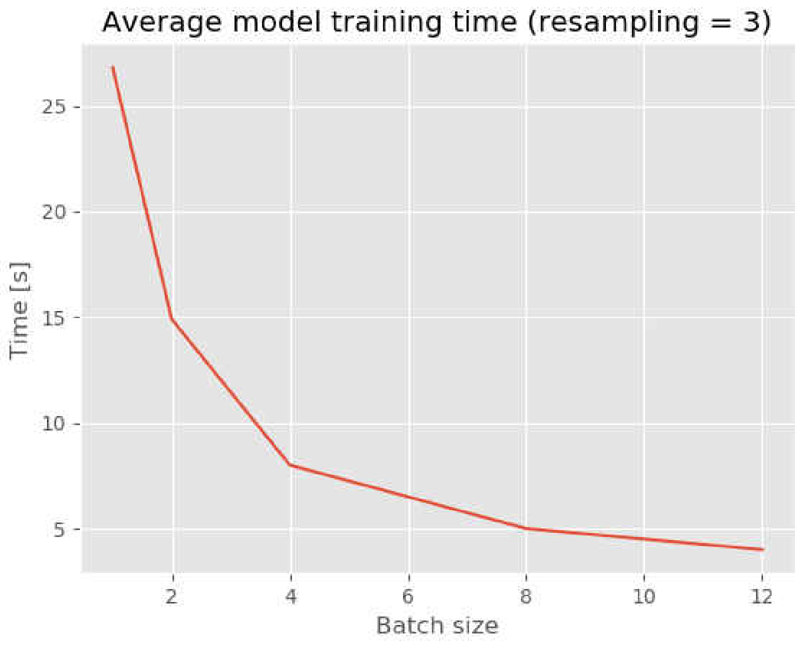


Figure 5.15: The average training time (over 10 epochs and 10 trials) for different batch sizes of models with resampling three times

CHAPTER 6
CONCLUSIONS

Convolutional neural network (CNN) models are de facto models for machine learning in image recognition and classification problems. While these models produce very good results, they also have a big practical disadvantage: they typically require huge amounts of data for optimizing the models' parameters. For instance, CIFAR-10 [12] has 60,000 color images, and the classic MNIST [11] contains 60,000 images of hand-written digit for training the model. This is an obstacle that can hardly be overcome in practice due to limited data availability. Previous work [18] used multiple smaller vanilla neural network models (which were relatively simpler), organized in a hierarchical way to perform hand-written character recognition using a much smaller data set for training the models. The result was very encouraging, with an overall accuracy of about 75%.

In this work, we tackled the challenge of using CNN models on a reduced dataset of manuscript letter images [10] intending to improve significantly the overall model's accuracy. Our dataset of manuscript letter images consists of 20 images for each character in a 22-letter alphabet, for a total of 440 images. As we tried to mimic a practical application when a user selects a character image and expects an automatic recognition, we did not impose any restriction on the size of each character image. Hand-written characters naturally imply various sizes for images of the very same character. We normalized each character image to a 20×20 matrix and created a CNN model taking such a 20×20 matrix as input and produced 22 numerical outputs, indicating the likelihoods of one of the characters in the alphabet. We created a relatively small-sized CNN model and combined it with a simple technique (resampling) to overcome the model's appetite for a large training dataset. Our model performs very well in practice, with an accuracy over 93% (on average), which is well above previous results and is more than satisfactory for any application seeking character recognition in manuscript images.

In the thesis, we provided a complete description of the model and data being used (including codes), and we performed extensive experiments and reported the results to support our claims. As our character recognition results are not complete, we showed the confusion matrix of each experiment we performed, indicating how certain characters were "confused" with others. This information can be used in practice, for instance, not only to cast doubt for some predictions but also to determine which actual character may create the confusion. Our model can produce not only the best prediction (indicated by the highest likelihood of the model's 22 outputs), but also the second or third predictions, which are of practical importance. We did not perform an analysis of these second or third predictions in this work, leaving this aspect for future work.

Our models and the experimental results suggest that resampling can be successfully used to overcome insufficient training data. However, as the results in chapter 5 show, resampling makes a complex model work on a small training dataset, but over-resampling does not increase accuracy. That is, one should limit resampling strictly to circumvent the model overfitting (due to a large number of model's parameters and small number of training data) but then combine this method with other techniques to increase the model's performance.

Our research focused on character image recognition for lower case hand-written characters. All character images we collected are lower case characters. There are very few upper case characters in the manuscript we studied. This would make the upper case character recognition from images a much more difficult problem. We did not address this problem in this study, and we leave it for future work. In our work, we addressed only single character image recognition; we did not address a group of two or more letter image recognition, and we leave it for future work too.

REFERENCES

[1] Kunlun Bai, *A comprehensive introduction to different types of convolutions in deep learning*, https://en.wikipedia.org/wiki/TensorFlow, April 30 2020.

[2] Hagan Demuth Beale, Howard B Demuth, and MT Hagan, *Neural network design*, Pws, Boston (1996).

[3] Adit Deshpande, *A beginner's guide to understanding convolutional neural networks part 2*, https://en.wikipedia.org/wiki/TensorFlow, April 29 2020.

[4] Rob A Dunne and Norm A Campbell, *On the pairing of the softmax activation and cross-entropy penalty functions and the derivation of the softmax activation function*, Proc. 8th Aust. Conf. on the Neural Networks, Melbourne, vol. 181, Citeseer, 1997, p. 185.

[5] Shaolei Feng, R Manmatha, and Andrew McCallum, *Exploring the use of conditional random field models and hmms for historical handwritten document recognition*, Second International Conference on Document Image Analysis for Libraries (DIAL'06), IEEE, 2006, pp. 8–pp.

[6] Peter Gehler and Sebastian Nowozin, *On feature combination for multiclass object classification*, 2009 IEEE 12th International Conference on Computer Vision, IEEE, 2009, pp. 221–228.

[7] D. O. Hebb, *The organization of behavior: A neuropsychological theory*, None, Psychology Press, 1947.

[8] Schmidhuber J. Hochreiter S, *Long short-term memory. neural computation.*, MIT Press, November 15 1997.

[9] Le Kang, Jayant Kumar, Peng Ye, Yi Li, and David Doermann, *Convolutional neural networks for document image classification*, 2014 22nd International Conference on Pattern Recognition, IEEE, 2014, pp. 3168–3172.

[10] Kevin Kiernan, *Electronic beowulf*, https://ebeowulf.uky.edu, February 2020.

[11] Yann LeCun, Corinna Cortes, and CJ Burges, *Mnist handwritten digit database*, ATT Labs [Online]. Available: http://yann. lecun. com/exdb/mnist **2** (2010).

[12] Shuying Liu and Weihong Deng, *Very deep convolutional neural network based image classification using small training sample size*, 2015 3rd IAPR Asian conference on pattern recognition (ACPR), IEEE, 2015, pp. 730–734.

[13] Warren S McCulloch and Walter Pitts, *A logical calculus of the ideas immanent in nervous activity*, The bulletin of mathematical biophysics **5** (1943), no. 4, 115–133.

[14] Prajit Ramachandran, Barret Zoph, and Quoc V Le, *Searching for activation functions*, arXiv preprint arXiv:1710.05941 (2017).

[15] F. Rosenblatt, *Principles of neurodynamics, perceptrons and the theory of brain mechanisms*, Cornell Aeronautical Lab Inc Buffalo NY, March 15 1961.

[16] Stuart Russell and Peter Norvig, *Artificial intelligence: A modern approach*, 3rd ed., Prentice Hall Press, USA, 2009.

[17] Oliver Knocklein Towards Data Science, *Classification using neural networks*, https://towardsdatascience.com/ classification-using-neural-networks-b8e98f3a904f, February 27, 2020.

[18] Sattajit Sutradhar, *Old English Characters Recognition Using Nerual Network*, As Thesis in department of Mathematical Science, Georgia Southern University, March 2018.

[19] Wikipedia, *Beowulf*, https://en.wikipedia.org/wiki/Beowulf, February 2020.

[20] Tony Yiu, *Understanding neural networks*, https://towardsdatascience.com/understanding-neural-networks-19020b758230, February 2020.

[21] Mehdi Rezaeian Zadeh, Seifollah Amin, Davar Khalili, and Vijay P Singh, *Daily outflow prediction by multi layer perceptron with logistic sigmoid and tangent sigmoid activation functions*, Water resources management **24** (2010), no. 11, 2673–2688.

APPENDIX

PYTHON CODE FOR CNN CHARACTER IMAGE CLASSIFICATION

```
#%%============== all libraries =======================
from __future__ import print_function
#set the matplotlib backend so figures can be saved
# in the background
import os
os.environ['PYTHONHASHSEED']=str(2020)
import random
random.seed(2020)
import numpy as np
np.random.seed(2020)
import tensorflow as tf
tf.set_random_seed(2020)
from keras import backend as K
session_conf = tf.ConfigProto(intra_op_parallelism_threads=1,
                                inter_op_parallelism_threads=1)
sess=tf.Session(graph=tf.get_default_graph(),config=session_conf)
K.set_session(sess)

# import the necessary packages
from keras import initializers
from keras.preprocessing.image import ImageDataGenerator
from keras.optimizers import Adam
from keras.preprocessing.image import img_to_array
from keras.utils import to_categorical
from imutils import paths

import matplotlib.pyplot as plt

import cv2

import pandas as pd

import seaborn as sns

from keras.models import Sequential

from keras.layers.convolutional import Conv2D

from keras.layers.convolutional import MaxPooling2D

from keras.layers.core import Activation

from keras.layers.core import Flatten

from keras.layers.core import Dense

from keras.callbacks import EarlyStopping


#%%===================== the NN class ====================

class LeNet:

@staticmethod

def build(width, height, depth, classes, seed):

# initialize the model

model = Sequential()

inputShape = (height, width, depth)


my_init = initializers.glorot_uniform(seed=seed)


# first set of CONV => RELU => POOL layers

model.add(Conv2D(32, (5, 5), padding="same",

input_shape=inputShape, kernel_initializer=my_init))
```

```
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

# second set of CONV => RELU => POOL layers
model.add(Conv2D(64, (5, 5), padding="same",
                                    kernel_initializer=my_init))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

# first (and only) set of FC => RELU layers
model.add(Flatten())
model.add(Dense(256, kernel_initializer=my_init))
model.add(Activation("relu"))

# softmax classifier
model.add(Dense(classes, kernel_initializer=my_init))
model.add(Activation("softmax"))

# return the constructed network architecture
return model


#%%==================== parameters =====================
#datasource directory
DATASRCDIR = ’../R/OE/’
```

```
#all letters
LETTERS =["a", "ae", "b", "c", "d", "e", "eth", "f", "g", "h",
 "i", "l", "m", "n", "o", "p", "r", "s", "t", "thorn", "u", "w"]

def reset_random_seeds(seed):
    os.environ['PYTHONHASHSEED']=str(seed)
    np.random.seed(seed)
    random.seed(seed)
    tf.compat.v1.random.set_random_seed(seed)

#%%============= set up and train =====================
# initialize the number of epochs to train for, initial
# learning rate, and batch size
INIT_LR = 1e-3
BS = 2
# this is the maximum number of epochs! it may stop before
# that, when two consecutive 100% accuracies happen
EPOCHS = 10

#resampling
RESAMPLE = 0

#we set asside NTEST images for testing
NTEST = 4

#store the results
```

```
accuracies = []
models = []
Hs = []
classifications = []
testY0s = []

seeds = [2020, 752, 90177, 54411, 321, 6543, 1103, 820,
                                              400, 12345]

for sidx in range(len(seeds)):

    # grab the image paths and randomly shuffle them
    imagePaths = []
    testImagePaths = []

    seed = seeds[sidx]
    #ensure some reproducibility
    reset_random_seeds(seed)

    for let in LETTERS:
        letdir = DATASRCDIR + let + "/"
        letimgs = list(paths.list_images(letdir))
        random.seed(seed)#ensure some reproducibility
        timgs = np.array(random.choices(range(0,len(letimgs)),
                                                   k = NTEST))
        testImagePaths.extend(np.array(letimgs)[timgs])
        imagePaths.extend(sorted(letimgs[i] for i in range(0,
                            len(letimgs)) if i not in timgs))

    # initialize the data and labels
    print("[INFO] loading images...")
    trainX = []
    trainY0 = []

    CLASSES = len(LETTERS)
    # loop over the input images for imagePath in imagePaths:
    # load the image, pre-process, and store it in the data list
     image = cv2.imread(imagePath)
     image = cv2.resize(image, (20, 20))
     image = img_to_array(image)[:,:,0].reshape((20,20,1))
     trainX.append(image)
     for i in range(RESAMPLE):
            trainX.append(image)

     # extract the class label from the image path and
     # update the labels list
     label = LETTERS.index(imagePath.split("/")[-2])
     trainY0.append(label)
     for i in range(RESAMPLE):
            trainY0.append(label)
    # print(labels)
    if (RESAMPLE > 0):
        idx = np.array(range(len(trainX)))
        random.shuffle(idx)
        trainX = np.array(trainX, dtype="float")[idx]
        trainY0 = np.array(trainY0)[idx]

    # scale the raw pixel intensities to the range [0, 1]
    trainX = np.array(trainX, dtype="float") / 255.0
    trainY0 = np.array(trainY0)

    testX = []
    testY0 = []

    CLASSES = len(LETTERS)
    # loop over the input images
    for imagePath in testImagePaths:
    # load the image, pre-process, and store it in the data list
     image = cv2.imread(imagePath)
     image = cv2.resize(image, (20, 20))
     image = img_to_array(image)[:,:,0].reshape((20,20,1))
     testX.append(image)

     # extract the class label from the image path and
     # update the labels list
     label = LETTERS.index(imagePath.split("/")[-2])
     testY0.append(label)
    # print(labels)
    # scale the raw pixel intensities to the range [0, 1]
    testX = np.array(testX, dtype="float") / 255.0
    testY0 = np.array(testY0)
    testY0s.append(testY0)

    # partition the data into training and testing using 80%
    # of the data for training and the 20% for testing

    # convert the labels from integers to vectors
    trainY = to_categorical(trainY0, num_classes=CLASSES)
    testYcat = to_categorical(testY0, num_classes=CLASSES)



    # initialize the model
    print("[INFO] compiling model...")
    #ensure some reproducibility
    reset_random_seeds(seed)
    model = LeNet.build(width=20, height=20, depth=1,
                            classes = CLASSES, seed=seed)
    #opt = Adam(lr=INIT_LR, decay=INIT_LR / EPOCHS)
    model.compile(loss="categorical_crossentropy",
        # optimizer=opt,
        optimizer="adam", #"nadam", "sgd"
     metrics=["accuracy"])
    models.append(model)
```

```
# train the network
print("[INFO] training network...")
#ensure some reproducibility
reset_random_seeds(seed)

earlyStop1 = EarlyStopping(monitor='val_acc', min_delta=0,
                patience=1, verbose=0, mode='auto',
                baseline=None, restore_best_weights=False)
earlyStop2 = EarlyStopping(monitor='acc', min_delta=0,
                    patience=3, verbose=0, mode='auto',
            baseline=None, restore_best_weights=True)
earlyStop3 = EarlyStopping(monitor='loss', min_delta=0,
                     patience=3, verbose=0, mode='auto',
            baseline=None, restore_best_weights=False)
earlyStop = []
H = model.fit(trainX, trainY, batch_size=BS,
    validation_split=0.2,
    callbacks = earlyStop,
 epochs=EPOCHS, verbose=2, shuffle=RESAMPLE > 0)
Hs.append(H)

classes = LETTERS

classification = model.predict(testX)
classifications.append(classification)
```

```
    res = classification.argmax(axis=1)
    con_mat = tf.math.confusion_matrix(testY0, res)
                    .eval(session=tf.compat.v1.Session())

    correct_prediction = tf.equal(testY0, res)
    accuracy = tf.reduce_mean(tf.cast(correct_prediction,
     "float"))
    a = accuracy.eval(session=tf.compat.v1.Session())
    print("[Trial #" + str(len(accuracies)+1) + "] Accuracy:
                                                 " + str(a))
    accuracies.append(a)



#%%================ plot accuracies from above ============
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(1, len(accuracies)+1), accuracies,
                                    label="Trial accuracy")
plt.plot(np.arange(1, len(accuracies)+1),
         np.repeat(np.mean(accuracies),len(accuracies)),
                                     label="Mean accuracy")
plt.title("Test data classification accuracies for "
            + str(len(seeds)) + " trials" +"\n(batch size = "
            + str(BS) + ", max epochs = " + str(EPOCHS)
          + ", resampling = " + str(RESAMPLE) + ")\n")
plt.xlabel("Trial #")
plt.ylabel("Accuracy")
plt.legend(loc="upper right")

print("Mean: " + str(np.mean(accuracies)))
print("SD: " + str(np.std(accuracies)))

#%%================= plot last model =================
TESTNO = 9
H = Hs[TESTNO]

# plot the training loss and accuracy
plt.style.use("ggplot")
plt.figure()
N = len(H.epoch)
plt.plot(np.arange(0, N), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, N), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, N), H.history["acc"], label="train_acc")
plt.plot(np.arange(0, N), H.history["val_acc"], label="val_acc")
plt.title("Training Loss and Accuracy for trial #"
            + str(TESTNO+1)+"\n(batch size = " + str(BS)
                        + ", max epochs = " + str(EPOCHS)
                + ", resampling = " + str(RESAMPLE) + ")")

plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend(loc="upper right")
```

```
#plt.savefig(args["plot"])


#%%============= confusion matrix (any test) ============
classes = LETTERS


TESTNO = 0
classification = classifications[TESTNO] #model.predict(testX)
res = classification.argmax(axis=1)
testY0 = testY0s[TESTNO]
con_mat = tf.math.confusion_matrix(testY0,
                     res).eval(session=tf.compat.v1.Session())


correct_prediction = tf.equal(testY0, res)
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
print("Accurracy: " + str(accuracy.eval(session
                                    =tf.compat.v1.Session())))


#%%================== show results ====================
#To normalize the result as from 0 to 1.
# Replace 'con_mat_df = pd.DataFrame(con_mat_norm,...)'
con_mat_norm = np.around(con_mat.astype('float') / con_mat.
                   sum(axis=1)[:, np.newaxis], decimals=2)


con_mat_df = pd.DataFrame(con_mat_norm,
                     index = classes,
                     columns = classes)
```

```
figure = plt.figure(figsize=(len(classes), len(classes)))
sns.heatmap(con_mat_df, annot=True,cmap=plt.cm.Blues )#,
                                                        fmt='d')
plt.tight_layout()
plt.ylabel('True char')
plt.xlabel('Predicted char')
plt.title("Confusion Matrix (percentages) for trial #"
 + str(TESTNO+1)+"\n(batch size = "+ str(BS)+ ", max epochs = "
     + str(EPOCHS) + ", resampling = " + str(RESAMPLE) + ")")

plt.show()

con_mat_df = pd.DataFrame(con_mat,
                     index = classes,
                     columns = classes)

figure = plt.figure(figsize=(len(classes)+1, len(classes)+1))
sns.heatmap(con_mat_df, annot=True,cmap=plt.cm.Blues )#, fmt='d')
plt.tight_layout()
plt.ylabel('True char')
plt.xlabel('Predicted char')
plt.title("Confusion Matrix (counts) for trial #" + str(TESTNO+1)
          +"\n(batch size = " + str(BS) + ", max epochs = "
         + str(EPOCHS) + ", resampling = " + str(RESAMPLE) + ")")
```

```
plt.show()


#%%================ TSNE visualization ===================


import time

import pandas as pd

from sklearn.datasets import fetch_mldata

from sklearn.decomposition import PCA

from sklearn.manifold import TSNE

from mpl_toolkits.mplot3d import Axes3D

import seaborn as sns

#%%=====================================================


X = trainX[:,:,:,0].reshape(trainX.shape[0],-1)

y = trainY0


print(X.shape, y.shape)


feat_cols = [ 'pixel'+str(i) for i in range(X.shape[1]) ]

df = pd.DataFrame(X,columns=feat_cols)

df['y'] = y

df['Char'] = df['y'].apply(lambda i:LETTERS[i])

X, y = None, None

print('Size of the dataframe: {}'.format(df.shape))


#%%plot some images
```

```
np.random.seed(42)
rndperm = np.random.permutation(df.shape[0])


plt.gray()
fig = plt.figure( figsize=(16,7) )
for i in range(0,15):
    ax = fig.add_subplot(3,5,i+1, title="Char: {}"
                        .format(df.loc[rndperm[i],'Char']) )
    ax.matshow(df.loc[rndperm[i],feat_cols].values
                            .reshape((20,20)).astype(float))
plt.show()


#%%===================PCA==========================
pca = PCA(n_components=3)
pca_result = pca.fit_transform(df[feat_cols].values)
df['PC1'] = pca_result[:,0]
df['PC2'] = pca_result[:,1]
df['PC3'] = pca_result[:,2]
print('Explained variation per principal component: {}'
                    .format(pca.explained_variance_ratio_))


#%%========plot some images========================
plt.figure(figsize=(8,6))
sns.scatterplot(
    x="PC1", y="PC2",
    hue="Char",
    palette=sns.color_palette("hls", len(LETTERS)),
    data=df.loc[rndperm,:],
    legend="full",
    alpha=0.3)

#%%=============Plot some 3D images==================
ax = plt.figure(figsize=(16,10)).gca(projection='3d')
ax.scatter(
    xs=df.loc[rndperm,:]["PC1"],
    ys=df.loc[rndperm,:]["PC2"],
    zs=df.loc[rndperm,:]["PC3"],
    c=df.loc[rndperm,:]["y"],
    cmap='tab10')

ax.set_xlabel('PC1')
ax.set_ylabel('PC2')
ax.set_zlabel('PC3')
plt.show()

#%% t-sne=========================================
df_subset = df.loc[:,:].copy()
data_subset = df_subset[feat_cols].values
pca = PCA(n_components=3)
pca_result = pca.fit_transform(data_subset)
df_subset['pca-one'] = pca_result[:,0]
df_subset['pca-two'] = pca_result[:,1]
```

```
df_subset['pca-three'] = pca_result[:,2]
print('Explained variation per principal
component: {}'.format(pca.explained_variance_ratio_))

#%%==================================================
time_start = time.time()
tsne = TSNE(n_components=2, verbose=1, perplexity=40, n_iter=300)
tsne_results = tsne.fit_transform(data_subset)
print('t-SNE done! Time elapsed: {} seconds'.format(time
                                        .time()-time_start))

#%%==================================================
df_subset['tsne-2d-one'] = tsne_results[:,0]
df_subset['tsne-2d-two'] = tsne_results[:,1]
plt.figure(figsize=(12,10))
sns.scatterplot(
    x="tsne-2d-one", y="tsne-2d-two",
    hue="y",
    palette=sns.color_palette("hls", len(LETTERS)),
    data=df_subset,
    legend="full",
    alpha=0.3)

#%%compare PCA and tSNE
plt.figure(figsize=(16,7))
ax1 = plt.subplot(1, 2, 1)
```

```
sns.scatterplot(
    x="PC1", y="PC2",
    hue="y",
    palette=sns.color_palette("hls", len(LETTERS)),
    data=df_subset,
    legend="full",
    alpha=0.3,
    ax=ax1)

ax2 = plt.subplot(1, 2, 2)
sns.scatterplot(
    x="tsne-2d-one", y="tsne-2d-two",
    hue="y",
    palette=sns.color_palette("hls", len(LETTERS)),
    data=df_subset,
    legend="full",
    alpha=0.3,
    ax=ax2)

#%% 50 components=====================================
pca_50 = PCA(n_components=50)
pca_result_50 = pca_50.fit_transform(data_subset)
print('Cumulative explained variation for 50 principal components
            :{}'.format(np.sum(pca_50.explained_variance_ratio_)))

#%% perform tsne=======================================
time_start = time.time()
tsne = TSNE(n_components=2, verbose=0, perplexity=40, n_iter=300)
tsne_pca_results = tsne.fit_transform(pca_result_50)
print('t-SNE done! Time elapsed: {} seconds'.format(time
                                              .time()-time_start))

df_subset['tsne-pca50-one'] = tsne_pca_results[:,0]
df_subset['tsne-pca50-two'] = tsne_pca_results[:,1]
plt.figure(figsize=(16,4))
ax1 = plt.subplot(1, 3, 1)
sns.scatterplot(
    x="pca-one", y="pca-two",
    hue="y",
    palette=sns.color_palette("hls", len(LETTERS)),
    data=df_subset,
    legend="full",
    alpha=0.3,
    ax=ax1)

ax2 = plt.subplot(1, 3, 2)
sns.scatterplot(
    x="tsne-2d-one", y="tsne-2d-two",
    hue="y",
    palette=sns.color_palette("hls", len(LETTERS)),
    data=df_subset,
    legend="full",
    alpha=0.3,
    ax=ax2)

ax3 = plt.subplot(1, 3, 3)
sns.scatterplot(
    x="tsne-pca50-one", y="tsne-pca50-two",
    hue="y",
    palette=sns.color_palette("hls", len(LETTERS)),
    data=df_subset,
    legend="full",
    alpha=0.3,
    ax=ax3)
```