Imperial College London

# A Hypothesis-Based Approach To Offline Handwriting Recognition

*Written by*
Thomas G. Grigg

*Supervised by*
Dr. Mark J. Wheelhouse

## Abstract

The ultimate goal of the field of Document Analysis/Document Recognition is that any handwritten piece of text, perhaps even including mathematics and diagrams, could quickly be turned into an accurate digital representation of itself at the press of a button. A subproblem in this domain is the recognition of written bodies of text in English - whilst this subproblem can certainly be considered solved in the case of online recognition - the offline problem is still considered unsolved - although there are various proposed semi-successful approaches. The best methods we have at the moment use the same model that is used for speech recognition or online handwriting recognition, but do not seem to offer the same level of accuracy. Herein, we discuss an attempt at attacking the (disconnected character) offline handwriting recognition problem. The attempt is based on using a hypothesis-based method combined with a dictionary, exploiting the similarity of the output of a convolutional neural network to a probability distribution over the characters it is taught to recognise. We build a recognition system with which we can apply this approach, and demonstrate that it is powerful in that it can significantly boost the accuracy of an underlying classifier (achieving improvements of up to 35% in word accuracy). Whilst doing so, we document and analyse the methods for building such a system for the purposes of comparison, such that we are able to optimise our implementation, and potentially to act as reference material for any future attempts.

## Acknowledgements

I would of course like to thank Dr. Wheelhouse for lending me a large amount of his time over the course of the project. Our discussions inspired many of the approaches I took towards progressing through the recognition pipeline, and they were always engaging and often fun. I would also like to thank my Mum and Dad, Becca, Ella and Steven for allowing me to pester them into writing test sentences, and for supporting me in my endeavours.

# Contents

# 1 Introduction

## 1.1 Terminology

Throughout this report it will be necessary to refer to various terms and abbreviations which we make clear here:

**Document Recognition** - The field dealing with translating human written documents (whether font-printed or handwritten) into digital representations of themselves.

**OCR** - Optical Character Recognition: The subfield of Document Recognition that deals with recognising individual, isolated characters.

**HWR** - Handwriting Recognition: The subfield of Document Recognition that aims to recognise handwritten sentences. For the purposes of this report, when we refer to handwriting recognition we are invariably talking about recognition of handwritten English, unless explicitly stated otherwise.

**Online recognition** is a method of handwriting recognition where the system does its recognition in real-time, as the user is writing, tracking the motion of the input device to use for classification.

**Offline recognition** is a method of handwriting recognition where the system takes only a static image of the written text to be recognised.

## 1.2 Motivation

This project began by setting out to investigate the possibility of creating a full document recognition system, capable of taking an image of handwritten writing/mathematics and translating it into LaTeX code, somewhat inspired by the Live Whiteboard Encoding project set forth by Dr. Wheelhouse. The applications of such a system would be widespread. One application that particularly motivates the need for such technology is the preservation of historical documents - currently, handwritten documents of immense value are required to be transcribed into digital form by humans - this task is tedious and time-consuming, also, due to the sheer volume of material, it is possible that many documents will never undergo such a transition. A fully realised document recognising system would reduce the problem to simply presenting the handwritten material clearly, thus allowing humanity to quickly proceed in this fairly important task (from an educational point of view). Such an application would also be an extremely useful tool for those taking notes in an educational environment.

The problem of total document recognition could be considered solved with the current state of the field in an online sense, since there already exist multiple pieces of online handwriting recognition and online mathematics recognition software, so realistically it would just be a matter of combining the two - i.e. not very interesting. Thus, we have decided to focus on the offline problem. In the early stages of the project, it became clear that a complete offline version of this system was in fact not realistically achievable, as even the problem of recognising handwriting offline is unsolved. This lead to the current form of the project, in which we are attempting to exploita classifier's distribution-like output to increase the overall accuracy of a recognition system's output.

## 1.3 Aims

The primary aim of this project is to investigate and develop the methods behind building a recognition-system with some capability of reading (disconnected) handwritten pieces of writing: special focus is given to constructing a method which increases the accuracy of an underlying and otherwise naive character classifier, by exploiting the probabilistic nature of its output to postulate hypotheses for what the output *could* be, and then deciding on the best output based on some measure. We go about this by actively attempting to build a recognition system that implements such a scheme. A secondary aim that evolved due to the experimental nature of such a system was one of scientific analysis: there are numerous methods for the various subproblems associated with the larger problem of offline handwriting recognition, and we aimed to analyse some subset of these and argue for their (in)effectiveness here in this report, presenting empirical data where appropriate. In section 5, the reader will find a summary and commentary on our achievements.

## 1.4 Contributions of this Project

During the course of this project,

- We designed an appropriately structured recognition pipeline. (3.1)
- We implemented the pipeline in an object-oriented manner in Java.
- Due to the experimental nature of our implementation, we documented our approaches and analyses for each step of the pipeline, comparing standard algorithms with our own. (4)
- We proposed a dictionary-focused approach to increase the recognition power of this pipeline. (1.7)
- We used deep learning libraries to create and train neural networks to act as likeness-functions for classifying handwritten characters. (4.5)
- We implemented an algorithm that allows us to get the most likely hypotheses over a joint distribution. (4.5.6)
- We implemented and evaluated three different scoring schemes for selecting the correct hypothesised words. (4.6)

All of this amounts to assessing an alternative approach to the most common method implemented by the vast majority of modern recognition systems, this standard approach is discussed in section (2.1).

## 1.5 Brief History and Related Work

Early motivations for OCR systems came from the application of recognising postal codes so that mail could be automatically sorted, with the goal of increasing the efficiency of postal services in several countries. OCR is widely applied to this problem today, and interestingly, the limitations of the accuracy of OCR actually have an effect on the characters that are allowed to appear in a code. (As a particular example of this, Canadian postal codes omit the letters D, F, I, O, Q, or U as they can be easily confused for other characters/digits by the OCR systems in place [2]). Many of the tools used to tackle this and similar problems have their origins in the 1980s, when the funding and interest in the field of Artificial Intelligence began to grow, and tools that were in some sense capable of learning began to arise, including the now well-known neural network model. With the fairly recent successes in applying Hidden Markov Models to the problem of speech recognition and online handwriting recognition, many pieces of software have been created that can interpret human speech and live writing to near-human levels of accuracy, and the problem is considered solved today. The problem of offline handwriting recognition is often related to these problems in the literature, the similarities are clear: a system is given an encoded form of the English language (i.e. as a sequence of audio signals, or a writing time-sequence, or images of handwritten text), and expected to decipher the intended message. Thus, these recent successes in these problems have certainly contributed to the renewed interest in offline handwriting recognition - bringing with it a variety of powerful new methods, two of the most prominent of which are presented succinctly in Section 2.

The ICDAR (International Conference on Document Analysis and Recognition) has been held every two years since 1991 to act as a driving force for the most outstanding advances in the field. They typically host various competitions, with varying focus each year. Often the majority of competitions are focused on printed-font OCR - but there are normally a number of competitions involving recognising handwriting presented too. Notably, ICDAR 2013 included a competition focused on recognising handwritten chinese, in which offline text recognition results of 86% were achieved [4]. It should be noted however that Chinese characters are logograms - each character is a symbol representing an entire word, and this actually reduces the problem of word recognition to character recognition, and in fact makes handwritten Chinese arguably easier to recognise than a language written in Latin script such as English. Modern approaches to the full unconstrained offline problem are frequently based upon Hidden Markov Models, and more recently, hybrid approaches involving neural network classifiers assisting in the classification steps [5][6]. A fairly simple but highly effective convolutional neural network architecture used for digit classification is discussed in section 2.2.10 - in this project we modify this architecture and use it to classify lower-case English character. While attempts involving the use of Hidden Markov Models often tackle the wider problem of unconstrained connected handwriting, we propose a possible critique of basing a recognition system on this approach in 2.1.8. The use of what we are attempting to achieve in this project, and its application to the wider field of Document Recognition essentially follows on from this analysis.

## 1.6 Core Problems

A character-classifying system that endeavours to recognise handwriting given only an image would be required to effectively solve the following three problems, none of which are considered solved in the state of the art.

### 1.6.1 Character Segmentation

The problem of character segmentation refers to the bounding of characters within the image. Typically, an image is binarized to have its pixels be totally black or totally white, and the black-regions are considered active, i.e. the black-regions are treated as a piece of written script, whilst the white regions are considered inactive, or background regions. (Of course, the paradigm can be reverse to the same effect, but this is the convention we will adopt throughout.) The problem of segmentation is different depending on whether we are handling the disconnected or connected case.
In the disconnected character case, we are assured that all valid whitespace-separated active regions are pieces of a single character, which possibly needs to be conjoined to the rest of the character, i.e. in the case of the dot of an ‘i’ character, we need to connect it to the body of the ‘i’ so that it can be passed to a classifier. This problem is difficult essentially because at the segmentation stage, a system has no knowledge of what each individual character is supposed to look like at all - so we are effectively given $n$ active regions, and in general the number of possible groupings of these regions is large, although can be constrained if we know, for example, that a character

can consist of at most two parts (in the case of the English alphabet) - but even with such constraints, trying every possibility is simply infeasible. For this reason, segmentation is often performed in a local and heuristic manner [3], or the problem is solved implicitly, such as is the case when using Hidden Markov Models [5].

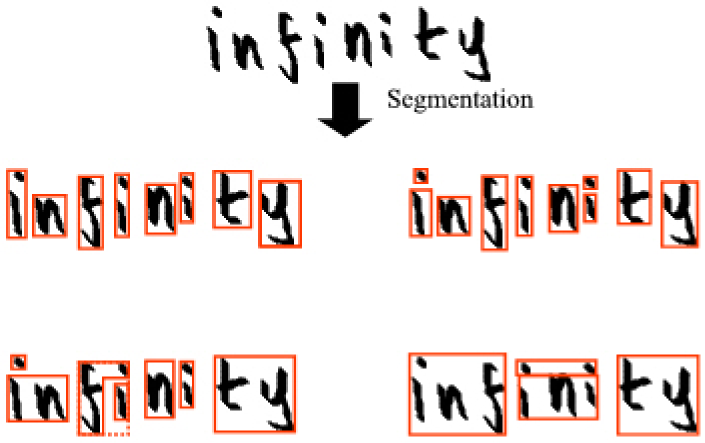


Figure 1: Top left: Correct segmentation of the presented active pixel regions. Top right: Segmentation by white-space. Bottom left: A possible segmentation given the constraints that a character has at most two disconnected pieces which must be adjacent (in some sense). Bottom right: A possible segmentation given by allowing arbitrarily many adjacent pieces in a character.

The problem would become even more difficult if we were to consider the unconstrained, connected character case. Since now, not only do we have to connect pieces of characters that are semantically connected but syntactically disconnected, we also need to potentially internally segment characters from connected active regions.

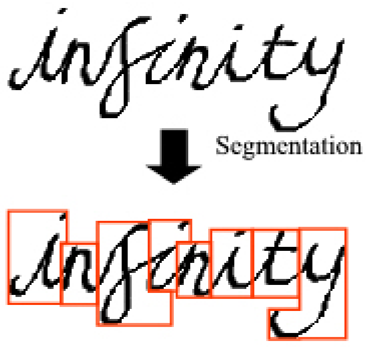


Figure 2: Correct segmentation for a connected version of the previous example. Explicitly computing this segmentation is extremely difficult.

The scope of this project will be dealing with the disconnected character case only, so that we can use simpler heuristic approaches focused on local positioning and whitespace, as segmentation in itself is a largely open problem in the field.

### 1.6.2 Layout Analysis

Given a document image, layout analysis is performed in order to extract the ordering and relative positioning of elements on the page. For an image of handwritten text, the first step for layout analysis is almost always to find the lines of text (often referred to as line segmentation). These lines of text are then typically analysed internally to group characters into words (often referred to as word segmentation). These steps are used when the document is transcribed essentially to define when a newline or space character is required. More advanced layout analysis would calculate relative offsets of different paragraphs of text in an image, and handle them accordingly, possibly even automatically avoiding any drawn diagrams (although, one of the final goals of Document Recognition would be to be able to transcribe arbitrary diagrams as well).

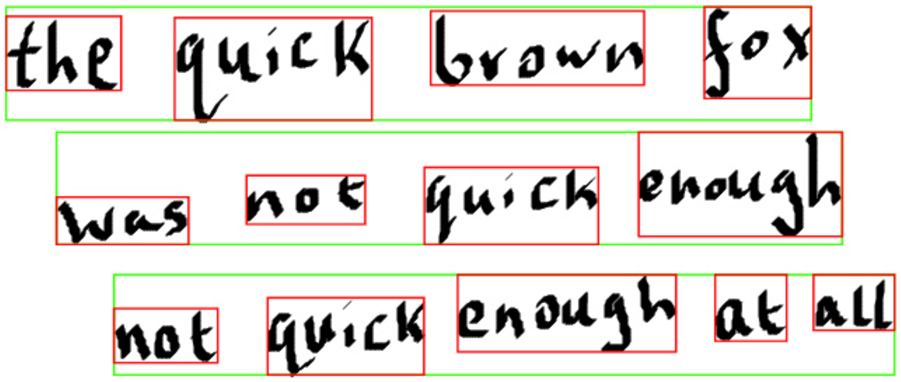


Figure 3: A basic example of line (green) and word (red) segmentation.

### 1.6.3 Character Classification

The problem of character classification is self-explanatory: given a picture of an (isolated) character, we attempt to classify it. There are two approaches to this problem: we can train a classifier with the intention that it should operate entirely as a black-box, i.e. we do not understand nor do we need to understand exactly which features the classifier is taking into consideration/what the classifier has learned about the character dataset, we only care that the classifier is capable of accurately classifying characters. In this first approach, usually only the isolated and preprocessed image of the character is fed to the classifier. The second approach is more involved, in that we explicitly extract features from a given character image and place them in a vector of values, before feeding this feature vector to the classifier. The first approach is more brain-like, in that we do not know exactly how our human brains are able to classify characters - typically a neural network or in some cases a genetic algorithm would be used to learn the features. The second approach is more algorithmic - perhaps one feature would be the length of a letters tail, and another could be the number and position of any loops. In practice, it is very hard to describe the (handwritten) alphabet like this, and the features used in models adopting the second approach are more abstract (we will see an example of such features later, in Section 2.1).

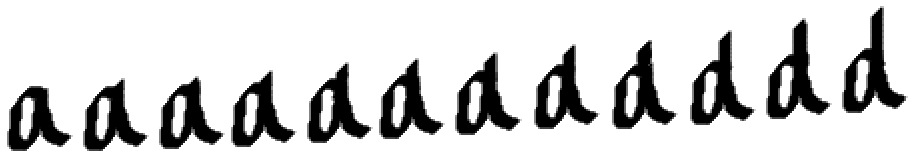


Figure 4: At what point along this line does ‘a’ end and ‘d’ begin?

Visuals like the one presented in Figure 4 could be considered arguments against the second approach. Whilst there do exist clear distinctions between letters when we learn them as children, and it would therefore likely be possible to exactly classify letters based on their features if we were looking at an educational diagram like the one presented in Figure 5, an explicitly defined feature based approach seems like it should not really work very well in the case of handwritten characters.

a b c d  
e f g h  
i j k l  
m n o p  
q r s t  
u v w x  
y z

Figure 5: An example of a simple educational diagram for learning the lower-case English alphabet.

## 1.7 A Hypothesis-Based Approach

Regardless of the method used to recognise handwriting, there is always a degree of uncertainty in the result. If we are given a classifier that explicitly classifies individual characters, is it possible that this uncertainty could be exploited to increase the accuracy of such a classifier?

If we take inspiration from the human thought process, written words could be treated as puzzles to be decrypted. Take for example Figure 6

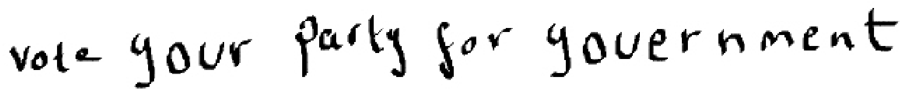


Figure 6: It is hopefully clear that this example is intended to read "vote your party for government".

The sentence above is clear and easy to read/decrypt, but if given to an isolated character classifier...

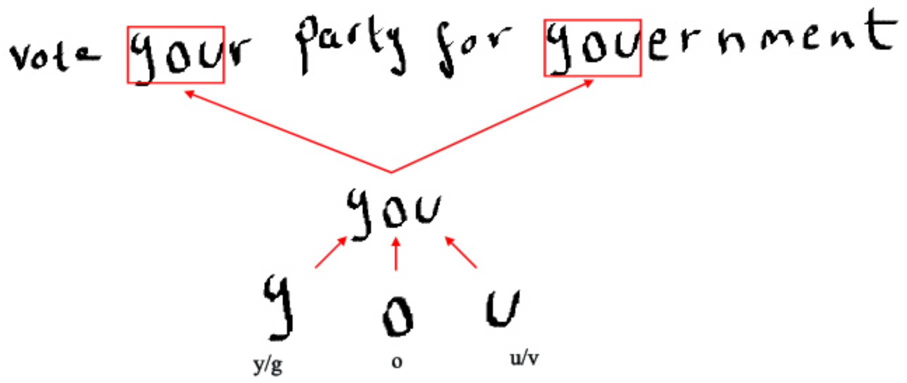


Figure 7: On a character by character basis, we see that the word "your" and "government" actually have identically written characters for the substrings "you" and "gov".

Thus, an isolated character classifier whose output is a single character corresponding to the chosen classification is never going to be good enough to solve the problem in the real world. A better approach would be to construct a fuzzy classifier, able to compute a likeness function over a given character image, and return an array of likeness values or a probability distribution. This fuzzy classifier could then be used to propose hypotheses for the transcription of a piece of text, and we could then reject or accept these hypotheses based on the evidence provided by the distributions over the characters of each word. This method is inspired by a very natural (and human) method for reading difficult to read words:

1. Assign the next most probable classification to each character to generate a string.

2. If this string makes sense, then we're done. Otherwise, is the string very close to anything that would make sense?

3. If it is, then return that string. Otherwise, go back to step 1.

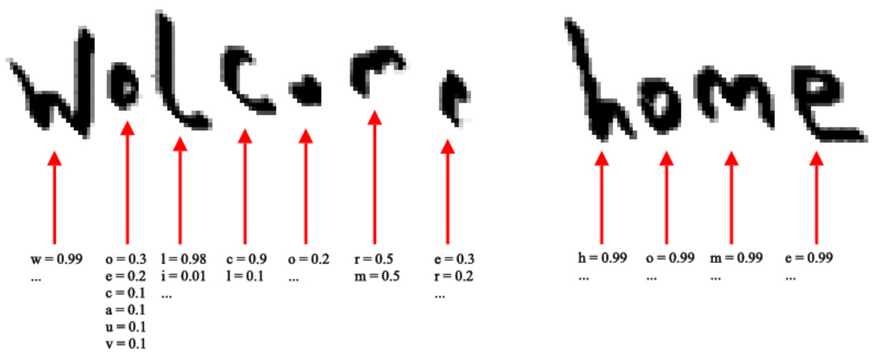


Figure 8: Visualisation of one possible assignment that such a function could give to some isolated characters in the piece of text.

Notice that the first hypothesis we would put forward for recognising Figure 8 under this method would be "wolcore home". We might not be able to get the word "welcome" from "wolcore", so we would then propose "wolcome home" as the next most likely, which is very close to "welcome home", which would be our result. In fact, this is the next most likely realisation of the string anyway[1].

We set out to investigate this notion. To do so, we must find a character classifier capable of approximating a function which assigns a likeness to members of the alphabet to an image.

Essentially, we want a classifier that can approximate a function

$$\mu : \texttt{Image} \to [0,1]^{26}$$

where $\mu$ can be decomposed as $\mu = (\mu_a, \mu_b, \mu_c, ..., \mu_z)$, where $\mu_\alpha : \texttt{Image} \to [0,1]$ is an $\alpha$-likeness function, i.e. if $\mu_z = 0$ then the image is definitely not depicting the handwritten character $z$, and if $\mu_z = 1$ then the image is certainly depicting the character $z$. We may also want the property that $\mu$ computes a probability distribution, i.e.

$$\sum_{\alpha \in a...z} \mu_\alpha = 1$$

but this property is not absolutely necessary, and can be obtained in either case simply by normalizing the likeness values. The type of classifier presented in Section 2.2.6 turns out to satisfy our requirements for being able to approximate an alphabetical likeness-function $\mu$.

[1]Of course, the human reader would likely recognise that "welcome" is very likely to precede "home" in a piece of writing, which could speed up this process such that it could be done without engaging conscious thought. The problem domain of constructing and analysing such lines of reasoning (called language models, see 2.1.5), is called natural language processing, and often supplements language recognition problems.

# 2 State of the Art

## 2.1 Hidden Markov Models

One such state-of-the-art approach to handwriting recognition is through the use of Hidden Markov Models. Generally, this technique is used to ascertain a hidden state sequence given some sequence of outputs probabilistically generated by these states. A common and particularly useful analogy is this: suppose we know that there is a man in a room which we cannot enter. Suppose further that we know he has with him a set of urns, each containing a distribution of balls of certain colours. If the man repeatedly selects an urn and takes out a ball (in some algorithmic way), and then shows them to us by sticking his hand out of the door, can we deduce the sequence in which he selected the urns from the sequence of coloured balls that we observe? The state space that we are interested in, the urns, is hidden from us - we must deduce the sequence by observing the stochastic outputs of the sequence alone.

Mathematically, a HMM consists of the following:

1. A (finite) set of states $\sigma = \{s_i | i = 1..n\}$
2. A set of outputs $\Omega$.
3. A transition probability function of two arguments $\tau : \sigma \times \sigma \to [0, 1]$ defined by $\tau(s_i, s_j) = P(s_i \text{ at } t | s_j \text{ at } t - 1)$
4. An emission probability function of two arguments $\epsilon : \sigma \times \Omega \to [0, 1]$ defined by $\epsilon(s_i, o) = P(o \text{ emitted } | s_i \text{ occurred})$.

The set $\sigma$ contains all possible states of the hidden sequence; the function $\tau$ describes the probability that one state will transition into another. The set $\Omega$ consists of possible observations caused by the underlying hidden sequence, and the function $\epsilon$ describes the probability that a given state will produce a given observation. This simple mathematical model is presented in the diagram below.

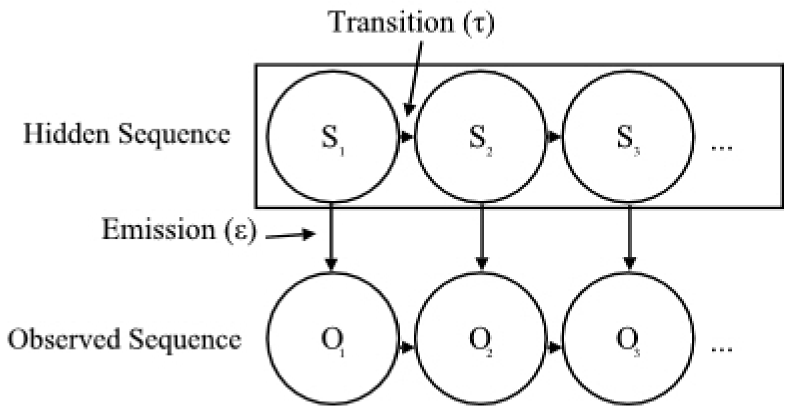


Figure 9: The Hidden Markov Model, as described. Here, the $s_i$ are members of $\sigma$ and the $o_i$ are members of $\Omega$

### 2.1.1 The Markov Property

It is important to note that the Hidden Markov Model is so named because the underlying state sequence is assumed to satisfy the Markov property (which is why we are able to completely describe its transitions with a 2-argument function $\tau$). The Markov Property states that

$$P(\text{state } u \text{ at } t |\text{ state } w \text{ at } t{-}1) = P(\text{state } u \text{ at } t |\text{ state } w \text{ at } t{-}1,\ \text{state } x \text{ at } t{-}2, ...)$$

i.e. essentially the transition process is "memoryless" - the next state depends only on the current state.

### 2.1.2 Application

Now that we have defined the Hidden Markov Model, we will see how to use it. There exist key algorithms that have been developed to allow us to infer underlying state sequences from sequences of observations given that we know the functions $\tau$ and $\epsilon$; most notably, these techniques have been very successfully applied to the problem of speech-recognition, where the hidden state sequence is treated as a string of text, and the output sequence it stochastically generates is the audio signal spoken by the user. This problem is entirely suited to the use of Hidden Markov Models as there is a natural ordering to the observed sequence (i.e. chronological ordering). The same is true of online handwriting recognition - the pen's position at time $t$ is used to define a 1-dimensional sequence of observations from which we can attempt to deduce the underlying text string. With the problem of offline handwriting recognition however, we are presented with an image only - and little knowledge of how exactly the pen moved in order to obtain this image. It is therefore necessary to somehow sequentialize this image so that it can be treated as an ordered sequence of observations generated by the hidden state.

### 2.1.3 Sequentialization: Sliding Window Approach

The standard solution to this problem is to first segment the image into individual lines of text and then to "slide" a small analysis window over the line, capturing images every few pixels as seen in Figure 10.

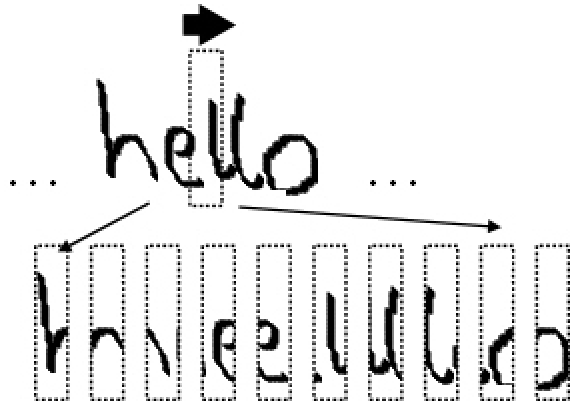


Figure 10: A visualisation of the sliding window technique.

These captured slices can then be used as a pseudo-chronological sequence of observations upon which we can perform feature extraction in order to generate a useful sequence of observed data.

#### Feature Extraction

Once we have a sequence of windows of this form, it becomes necessary to define procedures which can extract features from these small images in order to form a feature vector. Common features of interest include: the upper, lower and average contours, angles of orientation from baseline, normalized pixel-density, number of black to white transitions, and other similar geometric values, (features should ideally be independent of the appearance of the handwriting, and more focused on its form).

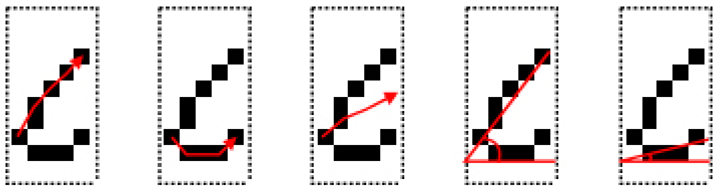


Figure 11: Visualisations of relevant features, from left to right: upper contour, lower contour, average contour, high angle from baseline, low angle from baseline.

### 2.1.4 Architecture

The application of Hidden Markov Models to this problem domain varies somewhat, but an attempt to present a more-or-less general architecture used to identify handwritten text is seen below.

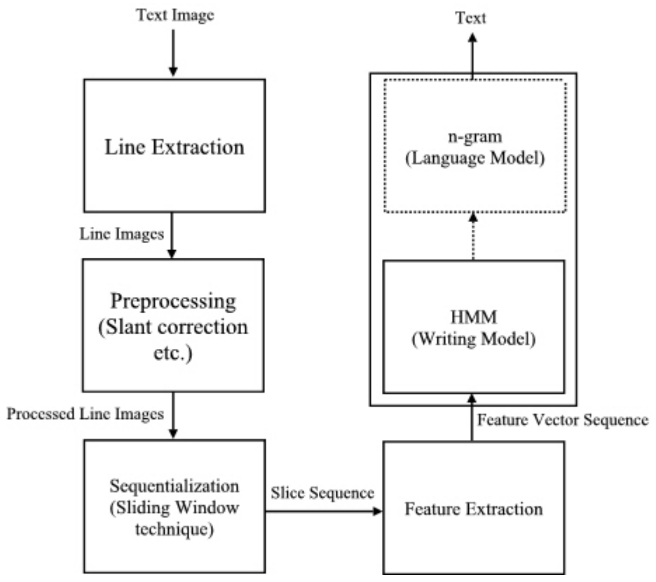


Figure 12: This diagram is essentially identical to the one given in [5].

**Decoding/Recognition**

The approach to recognition taken by an HMM classifier is to simply compute the most likely sequence of characters that could have resulted in the observed sequence of feature vectors. That is, we compute:

$$q^* = \text{argmax}_q \, P(q|\omega)$$

where $q$ ranges over all possible sequences with elements in $\sigma$ and $\omega$ is the observed sequence of feature vectors.

Notice that using Baye's rule we have:

$$q^* = \text{argmax}_q \, P(q|\omega) = \text{argmax}_q \, \frac{P(q)P(\omega|q)}{P(\omega)} = \text{argmax}_q \, P(q)P(\omega|q)$$

### 2.1.5 Writing and Language Models

Hence, to compute $q^*$ we need our model to describe the probabilities $p(q)$ and $p(\omega|q)$. The Hidden Markov Model describes the latter, $p(\omega|q)$ is the probability that we will obtain an observed $\omega$ feature vector sequence derived from handwritten text given that we have a given hidden character/word sequence $q$. If we have computed the parameters for a Hidden Markov Model describing this situation, then this exact notion is described by our emission function $\epsilon$ given as part of the mathematical definition of HMMs earlier. For this reason, the HMM is typically referred to as the writing model. The other probability of interest, i.e. the marginal probability of the sequence ($P(q)$) is simply the probability that the sequence of characters/words occurs at all, and the need to compute this probability is why we require a second model called the language model. Such a model is normally taken as a Markov chain of some order.

### 2.1.6 The n-gram Model

The most typical language model used to accompany HMMs in their task of handwriting recognition are called n-gram models. They are a very simple but powerful tool for calculating the probability that a given sequence of characters/words occurs. Essentially, the concept is that the model records a history of $n-1$ previously seen elements and assigns a conditional probability to the realisation of the next element. Mathematically, we are interested in tabulating the probability $P(x_i|x_{i-(n-1)}, x_{i-(n-2)}, ..., x_{(i-1)})$, where the $x_j$s belong to some set of elements, (n-grams used as language models for handwriting recognition will have the elements being either characters or words).

We can then easily calculate the probability we are interested in, i.e. the marginal probability $q$ of a sequence occurring, $P(q)$:

$$P(q) = P(q_1)P(q_2|q_1)P(q_3|q_1, q_2)...P(q_n|q_1, q_2, ..., q_{n-1}) = \prod_{t=1}^{n} P(q_t|q_1, ..., q_{t-1})$$

where $q = (q_1, q_2, ..., q_n)$.

Training an n-gram model is simple in practise. Sample probabilities can be calculated simply by counting occurrences given enough training data and time. However, in practice, alternative methods are used in order to speed up the training process.

### 2.1.7 The Viterbi Algorithm

Assuming we know the parameters of both models, the actual computation of the optimal sequence $q^* = \text{argmax}_q P(q|\omega) = \text{argmax}_q P(q)P(\omega|q)$, is done via the Viterbi algorithm.

Suppose we are given a Hidden Markov Model with state space $\sigma = \{s_1, ..., s_k\}$ and transition probability function $\tau$, as well as emission probability function $\epsilon$ and a probability distribution for the initial probabilities for the state space $(\pi_1, \pi_2, ..., \pi_k)$. Now, suppose we have a sequence of observations $o = (o_1, o_2, o_3, ..., o_T)$ - we wish to calculate the most likely sequence of hidden states $(x_1, ..., x_T) \in \sigma$ that emitted this sequence of observations. To do so, we introduce a new quantity: let $\mathcal{R}(i, s_j)$ be the probability of the most probable state sequence responsible for the first $i$ observations that ends in the state $s_j$. We then have that:

$$\max_{s_r} \mathcal{R}(T, s_r)$$

corresponds to the probability of the most probable state sequence responsible for the sequence of observations. Knowing this, we can calculate the

Viterbi path (the sequence of states that give rise to this probability), by calculating the probability iteratively and holding onto the values that we use along the way:

$$\mathcal{R}(1, s_r) = P(o_1|s_r)P(s_r) = \epsilon(s_r, o_1)\pi_r$$

$$\begin{aligned}\mathcal{R}(t, s_r) &= \max_{s_k \in \sigma} P(o_t|s_r)P(s_r|s_k)\mathcal{R}(t-1, s_k) \\ &= \max_{s_k \in \sigma} \epsilon(s_r, o_t)\tau(s_r, s_k)\mathcal{R}(t-1, s_k)\end{aligned}$$

If we let $x_{t,s_r}$ be the argument of the maximum in the calculation of $\mathcal{R}(t, s_r)$ (or $s_r$ if $t = 1$), we now have:

$$\begin{aligned}x_T &= \operatorname{argmax}_{s_r} \mathcal{R}(T, s_r) \\ x_{t-1} &= x_{t,x_t}\end{aligned}$$

which gives us the ability to calculate the Viterbi path.

### 2.1.8 Training via Baum-Welch

Of course, all of this relies on the fact that we have already obtained all of the necessary parameters for a HMM - namely, the probability functions $\epsilon$ and $\tau$ need to be calculated. Given a set of observed feature vectors and the corresponding hidden state sequence, the standard approach to training a HMM classifier (which involves iteratively improving an estimate of the HMMs parameters) is to use the Baum-Welch algorithm, which finds maximum-likelihood estimates for each of the probability functions. We refer the reader once more to Fink and Plötz [5] for more information on Baum-Welz.

#### Comments on HMM

The majority of today's approaches towards offline handwriting recognition are based in some way on the Hidden Markov Model. This is because the model has a key advantage in that it does not require an explicit segmentation step before classification. The recognizer is therefore segmentation-free (beyond line segmentation), and thus avoids all of the fairly nasty problems that come with attempting to pre-identify and order characters for classification by a more traditionally character-based classifier (problems with this are seen in later sections such as 4.4.1).

One of the primary disadvantages of using the Hidden Markov Model for offline handwriting recognition is its reliance on our ability to carry out an adequate and sensible feature extraction and use this to build an observed sequence. This is not a problem when HMMs are used for speech recognition, or online handwriting recognition - the observed sequence makes sense as a continuous signal generated by the underlying (unknown/hidden) text, without any extra effort on our part. However, when using the same techniques applied to offline handwriting recognition, we must construct the observed sequence ourselves, and this results in it being potentially hard to justify as a signal reliant solely on the text.

Further, the Markov property upon which the HMM depends for both the Baum-Welch and Viterbi algorithms is not necessarily representative of the underlying problem domain. For example, it is not realistically true that the next letter in a sequence of characters belonging to a valid sentence in the English language is entirely dependent on the previous letter. i.e. the true probability distribution for the next character given the string “exampl” is very different to the true probability distribution for the next character given only the string “l”.

The difficulty in this is that the reason HMM is such an attractive prospect is that layout analysis can be very very hard, and this method avoids it. However, avoiding the problem might not be the best idea in the long term: If we look ahead at the more general aims of the field of document recognition, this avoidance of segmentation is not necessarily a good thing. When it comes to recognising a document that contains mathematics, say, segmentation plays a key role, as key information is presented in the number of segments belonging to a symbol, and we also do not have the assumption that we can simply read from left to right (so we definitely could not create pseudo-time-sequence to be used in an HMM to recognise mathematics!) In fact, even when only considering the recognition of handwriting, there is information presented in the whitespace between words: the syntax works with the semantics of the words to offer a clearer meaning (if this was not the case, then we humans would likely not bother writing spaces at all.)

Part of the purpose of this project then, is to investigate an alternative approach to offline handwriting recognition. Whilst this does mean that we are only dealing with disconnected characters, it may one day be possible to convert a disconnected recogniser into a connected recogniser, just like it is possible to assign a pseudo-time-sequence to a handwritten image upon which HMM can do its work. The importance of what we are investigating is not necessarily in its ability to beat HMM recognisers in accuracy, (indeed, this is unlikely) - but to explore a more 'natural' way of recognising a library of symbols. One that, if developed far enough, could eventually lead to a more general method for document recognition. Replacing the alphabet with mathematical symbols, and the dictionary used to evaluate hypotheses for the realisation of a word with a database that contains knowledge of the structure of mathematical expressions - in theory a similar hypothesis-based approach could be applied.

## 2.2 Artificial Neural Networks

An Artificial Neural Network is a mathematical structure loosely modelling the interactions between neurons via axons to perform learning tasks within the biological structure of a brain. This mathematical model for learning is exceptionally effective for classifying pieces of data which are otherwise difficult to algorithmically classify. Essentially, the network acts as a black-box - by training a network to reduce its error rate over a large dataset of examples, ideally, the network will be able to effectively classify unseen examples, i.e., the aim of the training process is to force the network to generalise about the data it is presented, rather than memorize every single example it comes across. However, once this training process is complete, a human user cannot say exactly what the network has learned - just that it appears to be capable of solving the given classification problem to a certain degree of accuracy.

Certain types of neural network could be considered as state of the art for the problem of image classification, and for this reason they are often used to attack the problem of OCR and are used in several commercial applications that are capable of reading printed-font documents in the real-world; more recently, they are being used to attack the problem of (disconnected character) offline handwriting recognition.

### 2.2.1 Architecture

Typically, neural networks consist of a single input layer, a number of hidden layers, and a single output layer. The input layer is where we enter the values of a normalized feature vector corresponding to a piece of data (i.e. in the case of character recognition, corresponding to an image of a character) which the network can attempt to classify. Traditionally, each layer of neurons is connected to its immediate neighbours only (as seen in Figure 14) - however, within the last decade a large amount of research has gone into neural networks that can have more complex connections, (some even with loops), resulting in large advances in the power and accuracy of neural networks as classifiers.

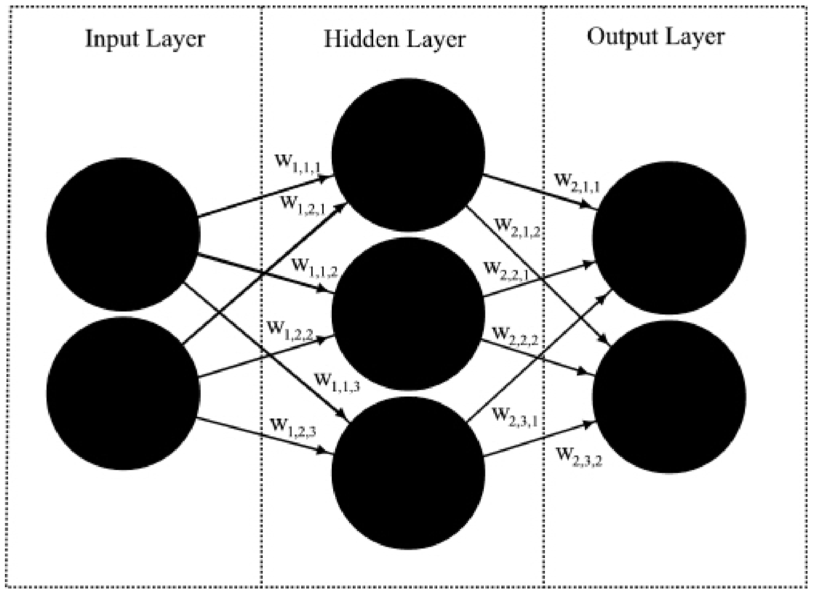


Figure 13: A simple feed-forward neural network architecture with a single hidden layer. The network has two input neurons, three hidden neurons and two output neurons.

### 2.2.2 Computation

Given a normalized feature vector $x = (x_1, x_2, ..., x_n)$ and a neural network with $n$ input neurons, computation is done by assigning the values of the feature vector, in order, to the $n$ input neurons, and then sending each value along its connections to be delivered to the next set of neurons. This value is multiplied by the weight of each connection before being used in the argument for the activation function of the connected neuron in the first layer. The activation function of a neuron is so named because when neural networks first arose, it would be a form of step-function, which only fired (activated) given that its received weighted input was large enough.

The general formula for the output of any given neuron is:

$$\sigma\left(\beta + \sum_{i=1}^{k} w_i o_i\right)$$

where $o_i$ refers to the output of the $i^{th}$ neuron feeding into the current one, and $w_i$ refers to the weight assigned to the connection between these two neurons, and $\sigma$ is the neuron's activation function. The $\beta$ term is referred to as a bias term, each neuron has its own (possibly zero) bias term. The purpose of a bias term is to give the network a greater degree of control over any single neuron's output, i.e. they make a neuron more or less likely to activate, or make a neuron have stronger or weaker output. We iterate this calculation through the network's layers until we arrive at the output layer, which can be interpreted by some (simple) procedure as our answer.

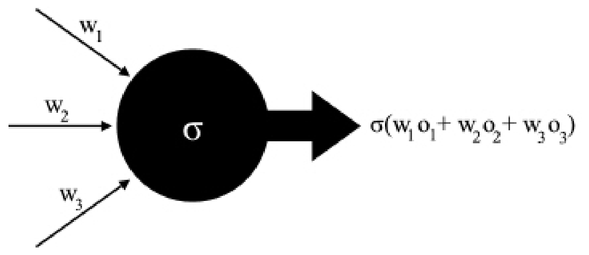


Figure 14: Depicted is a neuron which has three input connections each weighted with weights $w_1, w_2$ and $w_3$ respectively. The output of a neuron is calculated by the activation function $\sigma$, with the weighted sum of the outputs of the neurons connected to this one ($o_1, o_2$ and $o_3$, respectively), as an argument. The bias term in this case is zero and so is omitted.

### 2.2.3 Activation Functions

The choice of activation function is typically the same for any two neurons in the same layer of a network architecture, but each layer does not necessarily have the same activation function. This is because different activation functions promote different behaviours, and they can be used in combination to produce different networks with vastly different properties. Examples of very common activation functions are presented in the diagram below.

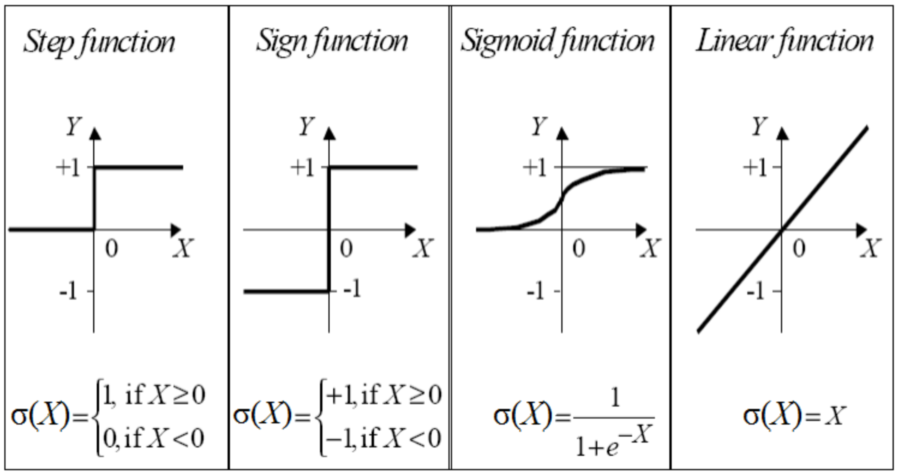

Step function
Sign function
Sigmoid function
Linear function
Y
X
+1
0
-1
$\sigma(X)=\begin{cases}1, \text{ if } X\geq 0\\0, \text{ if } X<0\end{cases}$
$\sigma(X)=\begin{cases}+1, \text{ if } X\geq 0\\-1, \text{ if } X<0\end{cases}$
$\sigma(X)=\frac{1}{1+e^{-X}}$
$\sigma(X)=X$

Figure 15: This image was taken from [14].

**Elementary Example**

To illustrate how neural networks are used in classification problems, we shall include a simple example. Given the following classification problem:

Suppose we have trained a network such that it will tell us whether or not the poles of two magnets will attract, or repel. Our set of training examples is thus enumerable and very small:

$$\begin{aligned}&(South, South, Repel)\\&(North, South, Attract)\\&(North, North, Repel)\\&(South, North, Attract)\end{aligned}$$

If, when considering the features $North$ and $South$, we elect to set $North = 1, South = 0$ such that the feature vector $(1, 0)$ corresponds to $(North, South)$ and we also elect to assign $Repel = 0, Attract = 1$, then these training examples can be written as

$$(0,0,0)$$
$$(1,0,1)$$
$$(1,1,0)$$
$$(0,1,1)$$

and with this scheme we can use these examples as a training set for a neural network. (It should be noted that this example is being presented only as a case for how a neural network can be used to compute something useful, namely, the XOR function - in reality, the use of neural nets is in their ability to be trained to generalise and classify unseen examples, training this particular network is a somewhat pointless exercise since we are easily able to enumerate all possibilities.) Nevertheless, suppose we fix a network architecture to have a single hidden layer with two neurons, and train it such that we end up with the network given in Figure 16.

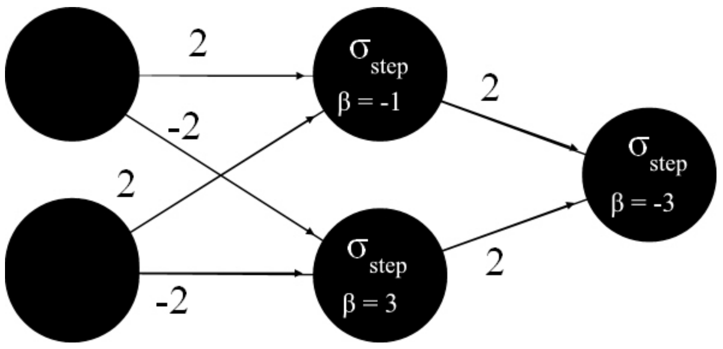


Figure 16

We can check that the network does indeed compute the function we trained it to compute.

$$\begin{aligned}
(0,0) &\to (\sigma_{step}(-1+2\times 0+2\times 0), \sigma_{step}(3-2\times 0-2\times 0)) = (0,1)\\
&\to (\sigma_{step}(-3+2\times 0+2\times 1)) = (0)\\
(0,1) &\to (\sigma_{step}(-1+2\times 0+2\times 1), \sigma_{step}(3-2\times 0-2\times 1)) = (1,1)\\
&\to (\sigma_{step}(-3+2\times 1+2\times 1)) = (1)\\
(1,0) &\to (\sigma_{step}(-1+2\times 1+2\times 0), \sigma_{step}(3-2\times 1-2\times 0)) = (1,1)\\
&\to (\sigma_{step}(-3+2\times 1+2\times 1)) = (1)\\
(1,1) &\to (\sigma_{step}(-1+2\times 1+2\times 1), \sigma_{step}(3-2\times 1-2\times 1)) = (1,0)\\
&\to (\sigma_{step}(-3+2\times 1+2\times 0)) = (0)
\end{aligned}$$

(where the tuples correspond to the output of each neuron in each layer).

So this very simple neural network does indeed compute the XOR function, and thus is a 100% accurate classifier for our original magnet situation. So, as we have now seen - classification using a neural network really is as simple as extracting features from our piece of data and plugging in the feature vector into the neural network. Thus, once we have a trained neural network, classification can be done very quickly. On the other hand, actually training an accurate neural network itself can take days or even weeks for especially complex recognition problems.

### 2.2.4 Universality

The most important property of the neural network model is stated succinctly as follows:

**Any mapping can be approximated to *arbitrary* accuracy by a network with at least two hidden layers**. (Cybenko 1988) [21]

This means that, so long as we use a neural network with at least two hidden layers: Given any mapping, we are guaranteed the existence of an architecture and weight-configuration on that network that computes the mapping with a classification rate of $(100 - \epsilon)\%$, for $\epsilon$ an arbitrarily small positive number. The issue, of course, is that the proof of this statement is non-constructive, essentially meaning we have little formal idea of how to find that architecture or weight-matrix for a given mapping. For this reason, obtaining a classifier with high accuracy is generally a highly empirical process - often beginning with a simple grid search over some range of parameters, and repeatedly narrowing down the search space by a mix of experimental intuition, and often pure luck. For the problem of English character classification, what this means is that even if there were to exist a perfect classification function for all handwritten English symbols, the odds of finding an approximation to it with negligible error are essentially zero. However, we can attempt to train a neural network to have a high degree of accuracy for this problem.

**Training an Artificial Neural Network**

The art of training an artificial neural network for a recognition task is usually done by adjusting training parameters in a process involving a lot of trial and error and heuristic-based approaches. The training process can be roughly summarised as the following:

1. Collect a large amount of training data. For a classification problem training data takes the form of a feature vector $x = (x_1, x_2, ..., x_n)$ with an attached class label corresponding to the output vector we ideally expect to see from the output neurons, $o = (o_1, o_2, ..., o_m)$. The pair $(x, o)$ form a training example for the classification problem at hand.

2. Fix a network architecture: choose the number of hidden layers and the number of neurons, along with their activation functions, in each layer, and specify how each layer is connected to the rest of the network.

3. Initialise the weights of the connections. Initialisation is usually done randomly, by picking weights from a Gaussian distribution, but there are several methods heuristically suited to different tasks.

4. Define a sensible error/loss function $e(t, o)$ that gives an accurate quantitative measure of the error of a computed output vector $t = (t_1, t_2, ..., t_m)$ from the desired correct label output vector $o = (o_1, o_2, ..., o_m)$. The best choice of error function will usually depend on the activation functions used within the network, as the derivative of the error function with respect to the networks parameters, which we will need to compute in the next training step, will be dependent on the selected activation

functions. A typical choice of error function when using step or sign activation functions is the squared Euclidean difference $\frac{1}{2}\sum_{k=1}^{m}(t_k - o_k)^2$.

5. Apply the gradient descent algorithm to the weights of the network, iteratively reducing the error of the function that our network computes over our set of training examples.

### 2.2.5 Back Propagation and Gradient Descent

Gradient descent is aptly named, as it is based upon the observation that if a function is differentiable, the quickest way to find a local minimum is to travel in the direction of greatest decrease (which is the opposite direction to the direction of greatest increase, i.e. the gradient). In the case of artificial neural networks, supposing we are given a network architecture with variable weights, and an error function that is differentiable with respect to each of these weights, we can use this idea to traverse the hyper-dimensional error surface and arrive at a (local) minimum.

Back propagation refers to the part of the algorithm that actually calculates the gradients: the partial derivatives of the error function with respect to each weight are pushed backwards through the network and used to calculate successive partial derivatives. Essentially, the partial derivative of the error function with respect to a given weight depends on the partial derivatives of the error function with respect to the weights attached to the connections of all the successively connected neurons between the given weight's starting neuron and the output.

For a given error function $e$, and training example $(x, o)$ we can express the error of this training example as a function of the network's weights. We do this by replacing the weights of the network with (a vector of) variable quantities $\overline{w}$ and plugging the feature vector $x$ into the network to get a computed output $t(\overline{w})$ in terms of the weights of the network. Computing the error $e(t(\overline{w}), o)$ thus gives us an expression for the error $e(\overline{w})$ in terms of the weights of the network. Thus, for any individual weight $w$ we can perform gradient descent by doing the following:

1. Calculate the partial derivative $\frac{\delta e(\overline{w})}{\delta w}$
2. Calculate the weight update $\Delta w = -\eta \frac{\delta e(\overline{w})}{\delta w}$
3. Perform the weight update: $w \leftarrow w + \Delta w$

Here, $\eta$ is a training variable called the learning rate, which dictates how large of a step across the error surface the algorithm takes at each iteration. Larger learning rates typically result in faster convergence to weaker local minima, however, if a learning rate is too small then convergence, and therefore training, will take a very long time - so fine-tuning and finding a balance for this parameter plays an important role when training a neural network.

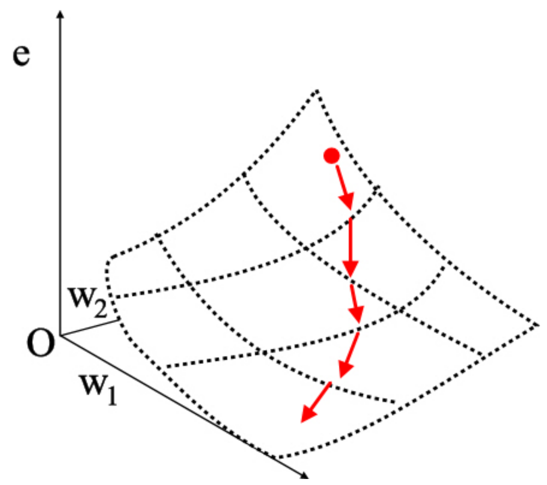


Figure 17: A simple illustration demonstrating the primary concept behind gradient descent: traversing the error surface to find a strong local minimum.

In practice, the learning rate is far from the only variable that requires tweaking by a human trainer. For example, while we do know that we can theoretically train a network to approximate any function of interest to an arbitrary degree of accuracy, we do not know how to construct such a network - thus, the architecture of the network itself is a parameter that requires experimentation and adjustment, even for the most simple classification problems.

The list of parameters grows further when we use more advanced versions of the gradient descent algorithm, incorporating momentum and learning rate decay, which are both necessary if a network is required to achieve competitive classification rates. Essentially, the inclusion of momentum ($\mu \in (0, 1)$) in the algorithm modifies the update rule to remember its previous updates and incorporate them into the calculation (thus giving our traversal over the error surface some momentum). Mathematically, the update rule with momentum can be expressed as $w \leftarrow (1-\mu)\Delta w_t + \mu \Delta w_{t-1}$. Learning rate decay simply allows the learning rate to drop off so that once we fall into a local minimum it becomes harder to come back out again.

It is also almost always necessary to choose a method of regularization (typically L1, L2 or maxnorm regularization) to be applied to the network's weights during training, in order to force the network to generalise, rather than memorize its training data.

### 2.2.6 Convolutional Neural Networks

Convolutional neural networks are a specialised type of network that are currently the front runners in image classification problems. This type of network was first inspired by the structures in place within the portions of the brain responsible for vision.

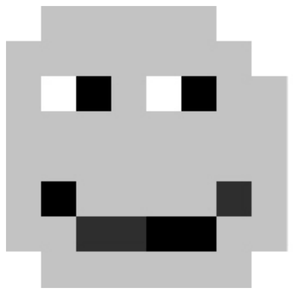
They are built using the following three types of neuron layer:

1. Convolutional Layers.
2. Sub-sampling/Pooling Layers.
3. Fully-connected/classification Layers.

The third of these, the fully-connected layer, is the same as the hidden layers discussed in the previous section. These layers act as a trainable black-box where a form of reasoning as to the correct classification of any provided data is done. The first two types of layers are essentially combined in order to learn to detect relevant visual features such that we can deliver these features to the fully-connected reasoning layer. If we are training a convolutional neural network to recognise handwritten characters, the idea is that the convolution and pooling layers will effectively learn what to look for to distinguish an ‘a’ from a ‘d’, for example. In this case, a longer and usually straighter tail tells us we have a ‘d’, but as we saw before, it can be problematic to try to differentiate characters like this, and so we don’t - we let the network work out how to do it.

### 2.2.7 Convolutional Layers

The input to a convolutional neural network is almost always an image represented as a matrix of pixel values. If the image was in colour, we would require three matrices, one for each colour component in the RGB colour model. For simplicity then, we will assume that such an image is grayscale. Now, to explain what a convolutional layer does, it is necessary to first briefly explain the notion of an image convolution. If the reader is familiar with the concept of a convolution in mathematics, then an image convolution is just a (mathematical) convolution of two functions, one of which is an image and the other a filter. We will use the following image with accompanying pixel matrix to make this clear:

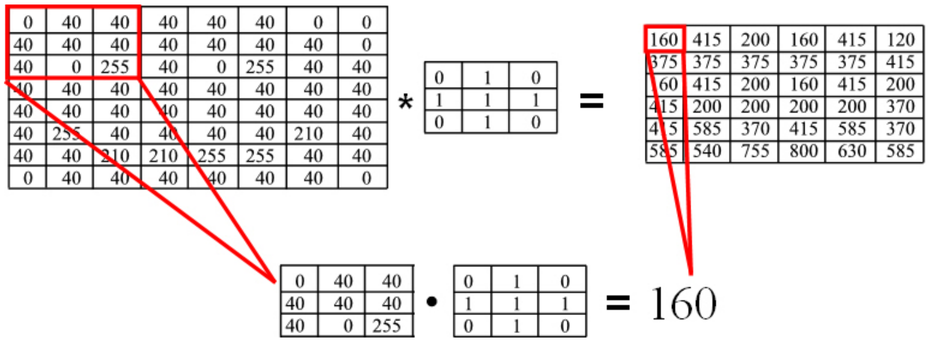


| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 40 | 40 | 40 | 40 | 40 | 0 | 0 |
| 40 | 40 | 40 | 40 | 40 | 40 | 40 | 0 |
| 40 | 0 | 255 | 40 | 0 | 255 | 40 | 40 |
| 40 | 40 | 40 | 40 | 40 | 40 | 40 | 40 |
| 40 | 40 | 40 | 40 | 40 | 40 | 40 | 40 |
| 40 | 255 | 40 | 40 | 40 | 40 | 210 | 40 |
| 40 | 40 | 210 | 210 | 255 | 255 | 40 | 40 |
| 0 | 40 | 40 | 40 | 40 | 40 | 40 | 0 |

Suppose we have a matrix with smaller dimension than the image matrix, take for example:

| | | |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 1 | 1 |
| 0 | 1 | 0 |

We can use these two matrices to generate an image convolution. To do so, we apply the filter (the smaller matrix) to the image matrix by taking the dot product of the filter with each sub-matrix of the same dimension as the filter within the image matrix.

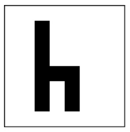


$$\begin{bmatrix} 0 & 40 & 40 & 40 & 40 & 40 & 0 & 0 \\ 40 & 40 & 40 & 40 & 40 & 40 & 40 & 0 \\ 40 & 0 & 255 & 40 & 0 & 255 & 40 & 40 \\ 40 & 40 & 40 & 40 & 40 & 40 & 40 & 40 \\ 40 & 40 & 40 & 40 & 40 & 40 & 40 & 40 \\ 40 & 255 & 40 & 40 & 40 & 40 & 210 & 40 \\ 40 & 40 & 210 & 210 & 255 & 255 & 40 & 40 \\ 0 & 40 & 40 & 40 & 40 & 40 & 40 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 160 & 415 & 200 & 160 & 415 & 120 \\ 375 & 375 & 375 & 375 & 375 & 415 \\ 160 & 415 & 200 & 160 & 415 & 200 \\ 415 & 200 & 200 & 200 & 200 & 370 \\ 415 & 585 & 370 & 415 & 585 & 370 \\ 585 & 540 & 755 & 800 & 630 & 585 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 40 & 40 \\ 40 & 40 & 40 \\ 40 & 0 & 255 \end{bmatrix} \bullet \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} = 160$$

The resulting matrix of values is referred to as the image convolution. Notice that the dimension of the image convolution depends on the dimension of both the filter and the original image matrix - sometimes we wish to control the size of the image convolution matrix, and this is easily done by zero-padding the input image matrix's borders (to a suitable size). Also, if we wish to display the image convolution as an image, in this case we may wish to normalize its values to lie between 0 and 255 to correspond to valid grayscale intensities.

Different image filters correspond to doing different things to the image, the image filter we just used as an example would blur the image, but other image filters could achieve things such as sharpening the image or even highlighting perceived edges within the image. Most importantly, we can use filters to detect evidence of features in an image - for an example of this, take a binarized image matrix of the character presented below:



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

and produce an image convolution using the filter:

| | | |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 1 | 0 |
| 0 | 1 | 0 |

obtaining:

$$\begin{array}{|c|c|c|c|c|c|c|c|}\hline 0&0&0&0&0&0&0&0\\\hline 0&0&1&0&0&0&0&0\\\hline 0&0&1&0&0&0&0&0\\\hline 0&0&1&0&0&0&0&0\\\hline 0&0&1&1&1&0&0&0\\\hline 0&0&1&0&1&0&0&0\\\hline 0&0&1&0&1&0&0&0\\\hline 0&0&0&0&0&0&0&0\\\hline \end{array} * \begin{array}{|c|c|c|}\hline 0&1&0\\\hline 0&1&0\\\hline 0&1&0\\\hline \end{array} = \begin{array}{|c|c|c|c|c|c|}\hline 0&2&0&0&0&0\\\hline 0&3&0&0&0&0\\\hline 0&3&1&1&0&0\\\hline 0&3&1&2&0&0\\\hline 0&3&1&3&0&0\\\hline 0&2&0&2&0&0\\\hline \end{array}$$

Now, the larger the value present at a location in the image convolution, the higher the similarity between the feature presented in the filter and the corresponding subsection of the original image. This particular filter is detecting vertical line segments. When a filter is used like this, it is referred to as a feature detector (or feature detecting filter), and a convolutional layer in a neural network is precisely designed to apply feature detectors to an image. Of course, the power of this lies in the fact that the convolutional layer is able to learn what features to look for without being explicitly told.
When building a convolutional layer, we effectively have two parameters: we must choose the size of the filter, and the number of filters we wish to apply. Thus, choosing an architecture for the network becomes more complex and requires even more experimentation.

The convolutional layer requires a special structure in order to actually implement the application of a feature detector. The input to a convolutional layer will be an image matrix, and the layer it is connected to will correspond to the image convolution matrix. The feature detecting filter is constructed by connecting each submatrix with the same dimension as the filter matrix to its output location in the image convolution matrix. The weights assigned to these connections then correspond to their respective value of the filter matrix. It should be noted that we are required to modify the gradient descent algorithm slightly so that when manipulating these weights they are considered as the same variable, such that the filter stays constant as it moves over the entire image matrix.

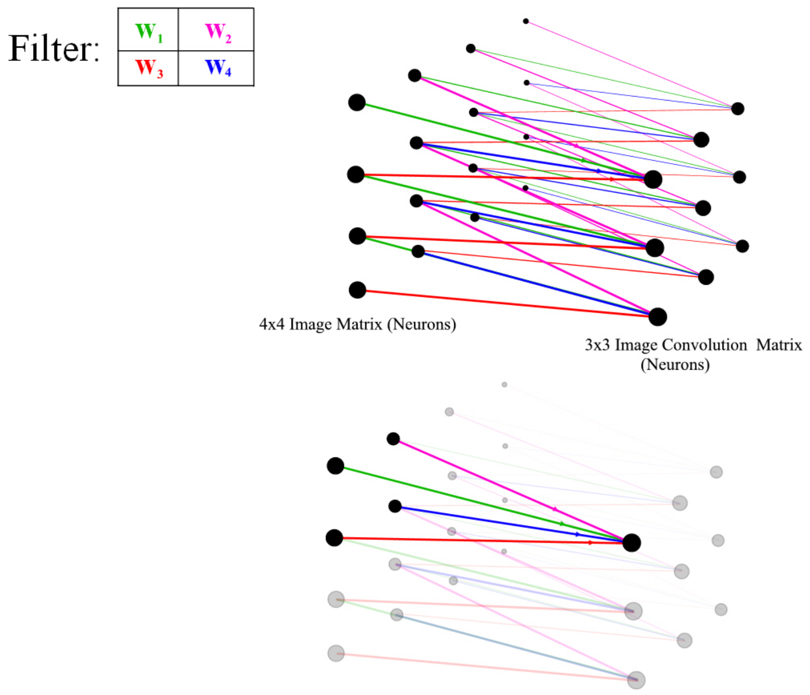

Filter:
W1
W2
W3
W4
4x4 Image Matrix (Neurons)
3x3 Image Convolution Matrix (Neurons)

Figure 18: Top: An illustration of the connections of a $4 \times 4$ image matrix when considering a $2 \times 2$ filter. Bottom: provided to clarify the exact structure of the connections: every $2 \times 2$ submatrix of neurons of the image matrix representation is connected to a single neuron in the image convolution representation.

Since a given neuron's argument for its activation function is simply $\sum_{i=1}^{n} w_i o_i$ where $o_i$ is the output of the $i^{th}$ connected neuron, and $w_i$ is the weight of the connection, it is clear to see why this network structure corresponds to computing the dot product of the submatrix and filter, and thus applying the filter.

The key here is that the filter has been expressed as a configuration of weights in the neural network - this means that the gradient descent training procedure will be able to modify them to minimise the network's error function - i.e. in the case of recognising handwriting, the network will be able to learn which filters are useful to consider when classifying characters!

The activation function used by the neurons corresponding to the image convolution matrix is typically the rectifier function $\text{rectifier}(x) = \max(0, x)$ - the motivation for this is that it introduces non-linearity into the image convolution process, and as a by-product, enforces that any detected features are more clearly defined in the convolved image matrix.

### 2.2.8 Pooling Layers

Pooling layers reduce the dimensionality of an image, typically after a convolution has taken place. They act to help focus the decision making process on the important features detected by a convolution operation. Returning to our visualisation of operating on matrices: a pooling layer simply reduces the disjoint submatrices of the input matrix to a given size, retaining information about the features present in each submatrix.

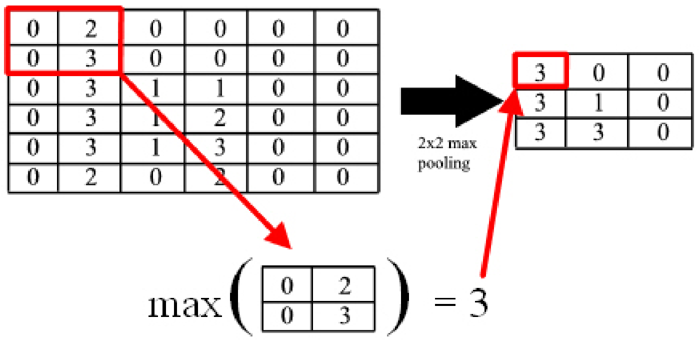


Figure 19: The most commonly used type of pooling layer is a max pooling layer, shown as a matrix operation here, which just takes the maximum element in each disjoint submatrix. This can be thought of as the pooling layer retaining the 'most important' portion of information from the input image.

Most pooling layers (including max pooling) are not trainable, instead they are a fixed function chosen by the trainer. As such they are yet another parameter to consider when designing a convolutional neural network.

### 2.2.9 Convolutional Architecture

A convolutional neural network normally begins with a convolutional layer with the capacity to learn several feature detecting filters, and convolutional layers are usually followed by a pooling layer. This pattern is usually repeated a few times until the network has systematically identified and reduced the dimensionality of multiple features in the input image such that the classification layer is in a better position to be able to classify the given information. Figure 20 showcases a very simple architecture for a possible convolutional network.

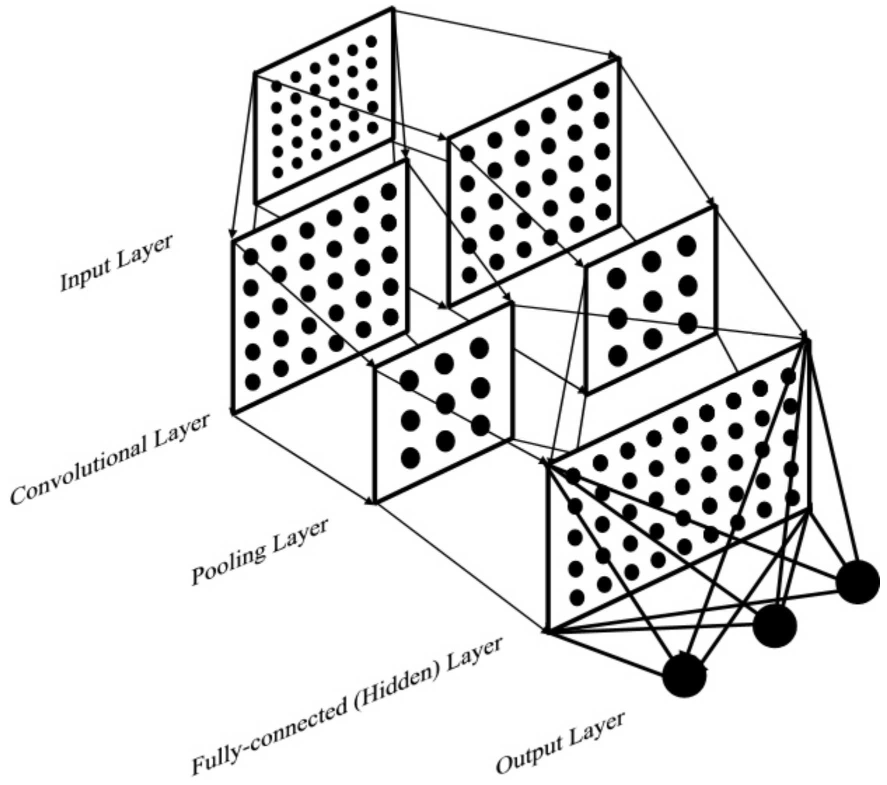


Figure 20: An example architecture for a convolutional network that classifies an input image into three possible classes. The convolutional layer has two feature detecting filters, creating two image convolutions which are then both subsampled separately, the features from both subsamples are then considered together in the fully-connected classification layer.

### 2.2.10 LeNet-5

The name LeNet refers to a family of network architectures developed by LeCun et al between the 1980s and late 1990s. LeNet-5 was the first neural network to be referred to as a convolutional neural network [11]. Variations of this network have since been applied to the MNIST dataset of handwritten digits to achieve near-human-level performance. The original network was capable of just over 99%[10], but networks based on this architecture achieved classification rates of 99.77% in 2012, which is considered human-level performance. We used an architecture identical to the original LeNet-5 as a base for building and training singular CNN classifiers on which to test any improvements in recognition capability (of words) eventually provided by our hypothesis-based method. The benefit of this, as discussed above in our comments, is that a CNN can learn to approximate a likeness function, which is a requirement of our method. We elected to base our classifiers on this architecture, rather than a more complex (but possibly more powerful) architecture, for two reasons: the first being that it is a relatively fast network to train due to its small size and simple structure, and the second reason being that it is a relatively simple network to understand, and it is well-documented, thus easier to experiment with. The architecture of the network is presented below in Figure 21.

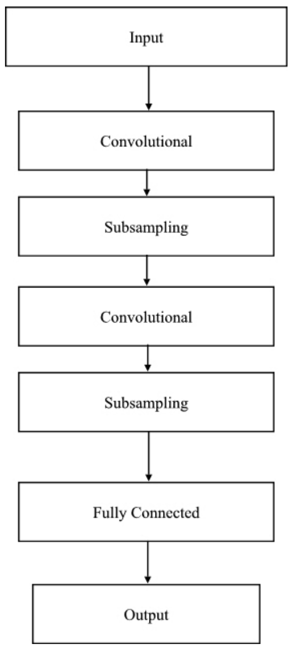


Figure 21: The basic structure of the LeNet-5 architecture. The 5 in its name comes from the fact that it has a structure with 5 layers (not including its input/output)

For the purposes of this project, we are more interested in the capabilities of our approach to improve the accuracy of an underlying classifier, rather than the final accuracy of the overall system. Thus, this simple but effective architecture will do for our evaluative needs.

# 3 System Design

The system is designed to be capable of attempting to recognise text written in lower-case letters only. This suffices to showcase any perceived benefits of our hypothesis-based approach, and allows the training times for our CNNs to be minimised. During our meetings, Dr. Wheelhouse and I talked about the ramifications and necessary modifications for introducing punctuation and capitalisation into the system, some of which are discussed in 6.2.

## 3.1 Recognition Pipeline

For a disconnected character HWR system, the necessary tasks to convert an image depicting handwritten words into digital text have already been outlined in Section 1.6, and can be visualised as the pipeline displayed on the left in Figure 22.

.

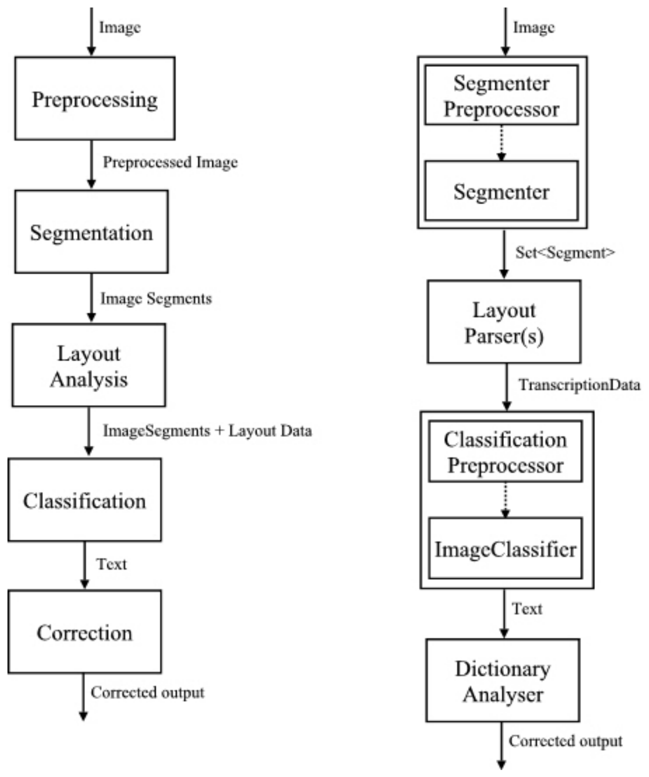


Figure 22: Left: A model isolated character HWR pipeline. Right: Object-oriented design for the pipeline used in the system.

The diagram on the right displays the design of our system's pipeline, based upon the pipeline on the left.

### Encapsulation of Components

Due to the experimental nature of building a document recognition system, it was vital to design the system with interchangeability in mind. This allowed us to switch between components easily throughout the pipeline. We experimented with using multiple different techniques and algorithms, comparing and evaluating these was facilitated by sticking closely to this fundamental design principle. As such, our system was designed for and implemented in Java, since Java is entirely object-oriented and has a few powerful libraries that can help us with, in particular, building and training Convolutional Neural Networks.

## 3.2 System Overview

### Preprocessing

Within the system, both the segmentation step and the classification step require the input image to be preprocessed. As such, we have a `Preprocessor` class which has has a field `preprocesses` which is a list of `ImageProcess` objects to be applied to an image. An `ImageProcess` is an interface with a single public method, `process`, which modifies the input image to generate the processed image. The package `Preprocessing` is where we store all of our implemented preprocesses and the class `ImageProcessLibrary` where we define several static methods which are commonly used by instances of `ImageProcess`.

### Segmentation

An implementation of the `Segmenter` interface's purpose is to perform the segmentation step, returning a set of `Segments`. A `Segmenter` has its own `Preprocessor`, used to make the image suitable for a simple whitespace separated segmentation operation. Contrary to the definition of character segmentation given in 1.6, a `Segmenter` makes no attempt to join disconnected components of the characters 'i' and 'j', instead, this operation is left until after we perform line segmentation, since in practice it is far easier to do there. The `Segment` class holds a reference to the preprocessed image and four integer values `top, bottom, left, right` corresponding to the tightest possible bounding box around the segment in the co-ordinate system of the image. The `Segment` class also has a method `conjoin(Segment)` which, when called by segment 1 with segment 2 as an argument, returns a segment with `top, bottom, left, right` corresponding to the tightest possible bounding box around both segments. This method is used to join disconnected components of the same character.

### Layout Analysis (Line and Word Segmentation)

Once a whitespace separated segmentation has been performed, the set of `Segment`s is used to construct an instance of the class `TranscriptionData`. `TranscriptionData` can essentially be considered as a tuple `(segments, lineDividers, wordDividers)` where `segments` is a list of `Segment`s and `lineDividers` and `wordDividers` are lists of integers.. `TranscriptionData` is initialised with `segments` having no guaranteed ordering, and `wordDividers` and `lineDividers` being empty. The idea is that `TranscriptionData` will then be fed into and permuted by realisations of the `Parser` interface, which are designed to place a read-ordering on the segments and insert integers into `wordDividers` and `lineDividers`. These integers correspond to the indices of the final segment of all words and lines in the now ordered list of `segments`, respectively. After this has been completed, we are ready to begin the recognition process.

### Classification

`ImageClassifier` is the interface designed to handle the training, testing and utilisation of a classifier. Realisations of `ImageClassifier` are somewhat dependent on the underlying classifier to be implemented, but every classifier must have a `train` method which must be given a path to a file system of training examples. Similarly, each classifier must be given access to a set of testing examples so that its post-training accuracy statistics can be obtained. An `ImageClassifier` must of course implement a `classify(Image)` method which, given an image, returns the image's `ImageClass` according to the approximate function that the classifier has learned to compute. An `ImageClass` simply has a string `label` field, referring to the classification of the image. An `ImageClassifier` also has a `Preprocessor` object used by the `classify` function. Detailed logs of the training and testing process are stored in instances of `TrainLog` and `TestLog` respectively - these classes are updated as training/testing progresses and perform their calculations once complete, outputting the results and metrics of training and testing.

### Fuzzy Classification

A `FuzzyClassifier` is designed to learn to compute a likeness function like the one discussed in 1.7. As such, a `FuzzyClassifier` is an extension of an `ImageClassifier` with the added method of `fuzzyClassify(Image)`, which returns a `FuzzyChar`. A `FuzzyChar` holds a collection of `ImageClass`es corresponding to different character classes, along with a probability distribution over them. We can use a `FuzzyClassifier` to begin our hypothesis-based method of recognising handwriting by calling `getFuzzyStrings(FuzzyClassifier)` on our `TranscriptionData`. This will return a list of lists of `FuzzyString`s, where each sublist corresponds to a line. A `FuzzyString` is just a list of `FuzzyChar`s which have been delimited into words by the `wordDividers` we computed earlier.

## Dictionary Analyser

Given such a list of lists of `FuzzyString`s, we can use a dictionary to find the best hypothesis for a given `FuzzyString`. We have not built an interface to correspond to a dictionary analyser, as in principle they should not be constrained to any one particular structure, but in general they have a `parse(List<List<FuzzyString>>)` function which takes a list of lists of `FuzzyString`s and returns a single string - i.e. the analyser chooses the best realisation of the `FuzzyString`s according to its own idea of 'best', and then concatenates them. `FuzzyString` has a method to aid with this operation, `getNMostLikely(Integer n)`, which returns the $n$ most likely realisations of the `FuzzyString` based on the distributions of the sequence of `FuzzyChar`s.

### Assembling the Pipeline

All of this is tied together in the `HandwritingRecogniser` class, which builds and co-ordinates the flow of the recognition pipeline. It is built via a `Segmenter` with a `Preprocessor`, and two `Parser`s, `LineParser` and `WordParser` which handle the layout analysis step. The recogniser has two classifiers, one fuzzy and one non-fuzzy, for testing purposes (i.e. we can measure any improvements using the hypothesis-based method), and it has a single dictionary analyser to implement the hypothesis-based method on `FuzzyString`s.

## 3.3 Test Environment

In order to aid in the testing of a `HandwritingRecogniser`, we built a simple test environment GUI that displays key information from the recognition process.

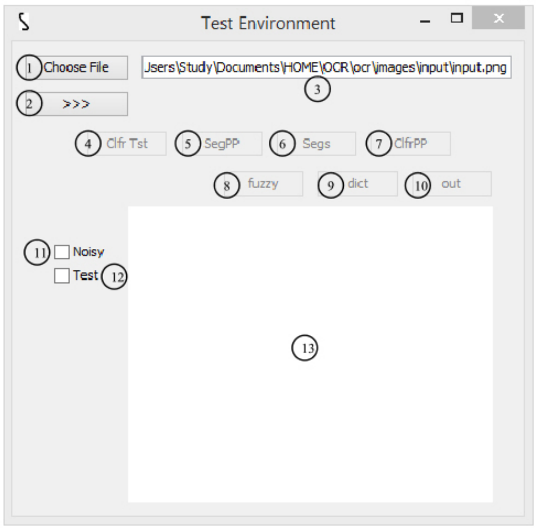


Figure 23: Graphical User Interface of the Test Environment. 1 - Opens a file dialogue so that an input image can be chosen. 2 - Transcribes the image. 3 - Displays path for selected image. 4 - Displays the most recently tested classifier's test statistics (including confusion matrix). 5 - Displays the input image after it has been preprocessed ready for segmentation. 6 - Displays the segments of `TranscriptionData` after word and line segmentation, i.e. in read-order. 7 - Displays the segments after they have been preprocessed by the classifier preprocessor. 8 - Allows us to see a given segment's `FuzzyChar` as dictated by the `FuzzyClassifier`. 9 - Shows the hypothesis-based method's output. 10 - Shows the naive output. 11 - Notifies the recogniser that the given image will need to be denoised before recognition. 12 - Notifies the system that, during the next transcription, we wish to test the statistics of the classifier. 13 - Display area for transcription output.

# 4 Implementation and Experimentation

In the previous section we showcased an overview of how the components of our system were designed to operate together. In this section we will discuss and compare the different algorithms and their implementations used within each stage of the pipeline.

## 4.1 Evaluation of Methods

While there do exist standard databases of connected handwritten sentences for testing HMM-based models, such as the IAM database [20], we were unable to find one for explicitly disconnected handwritten sentences. As such, we collected a sample of 40 pieces of writing, with sizes ranging from a single sentence to a short paragraph. These sentences were collected from 6 writers. We stored the sentences in raw form, in a denoised form and in segmentation-correct form. The first of these is simply the untouched form of the image, i.e. either a picture of handwritten text taken via camera-phone, or an image created via digital tablet. The second is the image after it has been automatically denoised by our best method, and then corrected by hand (if necessary). The third is a more invasive modification of the image, where we have moved the relative positioning of characters where necessary, so that our segmentation step is correctly performed - i.e. so that we get the correct line and word segmentation. The purpose of having these three versions of the test set is so that we can test the accuracy of our methods independently of any failures earlier in the pipeline. Essentially, to test a layout analysis algorithm fairly, we should assume that the denoising step (if any) has succeeded, and to test for an improvement in accuracy using a hypothesis-based approach via a `DictionaryAnalyser` with a scoring method, it is good practice to assume that the layout analysis step was completed successfully. We were also of course able to test the entire pipeline on each raw image, just for good measure, see 8.1. 20 of the images were noisy, i.e. taken on a camera-phone, whilst 20 were written on a digital tablet. Since 40 is a relatively small number of examples, the results of these experiments should not be interpreted as a presentation of how well the tested methods would fare in a real world scenario. Instead, the results are provided only as a comparative measure between the various methods we have implemented.

## 4.2 Preprocesseses

A call to a `Preprocess`'s `process` method works directly on a pixel array corresponding to the input image. As such, a `Preprocessor` must convert the image to a 2D integer array, and convert back once all of its `preprocesses` have been applied to the array. We implemented various image preprocesses to aid in the segmentation and classification steps of our recogniser.

### 4.2.1 Preprocessing Noisy Images

Depending on the source of the image, different preprocesses may need to be applied. For example, if the image is of a piece of paper taken on a camera in the real world, then the image is likely to be subject to unfavourable lighting conditions and non-constant background values. This would be extremely problematic for the segmentation step, as this typically relies on being able to distinguish the written text to be classified from the background by pixel value alone. Various methods (borrowed from computer vision) exist to attempt to normalize the image so that it has a standard and fixed background colour (that a segmentation algorithm can use to distinguish what it needs to segment, and what it can ignore). It is also usually makes things easier to binarize the image so that a segmentation algorithm is essentially presented with a 2-dimensional array of active (text) and inactive (background) pixels. When denoising an image, it is unreasonable to expect that there will be absolutely no leftover artefacts, so we implemented a `Segmenter`, `NoiseResistantWhitespaceSegmenter` which removes any segments which are sufficiently tiny when compared with the average segment. As such, when discussing the results in 8.1, a successful denoising is considered to have occurred when the denoising process does not destroy anything that we want to keep in the image, and any artefacts are sufficiently small such that they can be removed by `NoiseResistantWhitespaceSegmenter`.

#### Using the Average Pixel Value

The first algorithm we implemented to denoise an image was naive but surprisingly effective for its simplicity. `ContrastifyByAverage` calculates the average red, green and blue intensity of a pixel in an image, and defines an experimentally tuned `threshold` such that a pixel is set to black if its intensity[2] lies below the sum of each of the averages and the threshold, and white otherwise. We found the optimal value of the threshold over our test set to be 34 or 35. It should be noted that threshold value is relative to the same 0-255 standard scale for RGB colour intensities. For `threshold = 35`, we see in 8.1 that this preprocess adequately denoised 8 of the 20 noisy images in our test set. The pseudo-code of the algorithm is presented below in Figure 24.

```
redAvg ← getAverageRed(image)
greenAvg ← getAverageGreem(image)
blueAvg ← getAverageBlue(image)
for y = 0..height do
   for x = 0..width do
      if getRed(image[x][y]) > redAvg + threshold
&& getGreen(image[x][y]) > greenAvg + threshold
&& getBlue(image[x][y]) > blueAvg + threshold
then
         image[x][y] ← WHITE
      else
         image[x][y] ← BLACK
      end if
   end for
end for
```

Figure 24

We also developed a grayscale version of the algorithm, but this was not nearly as effective due to some of the noisy images being written on a white-board with a lighter coloured marker. Slight shadows were a problem in the non-grayscale version of the algorithm, and were even more so in this version.

[2]It should be noted that in the class `ContrastifyByAverage`, some of the inequalities are reversed with respect to the pseudo-code of the algorithm, this is because we essentially used the 'inverse' intensity scale, so that darker pixels (more active) would correspond to the larger numbers.

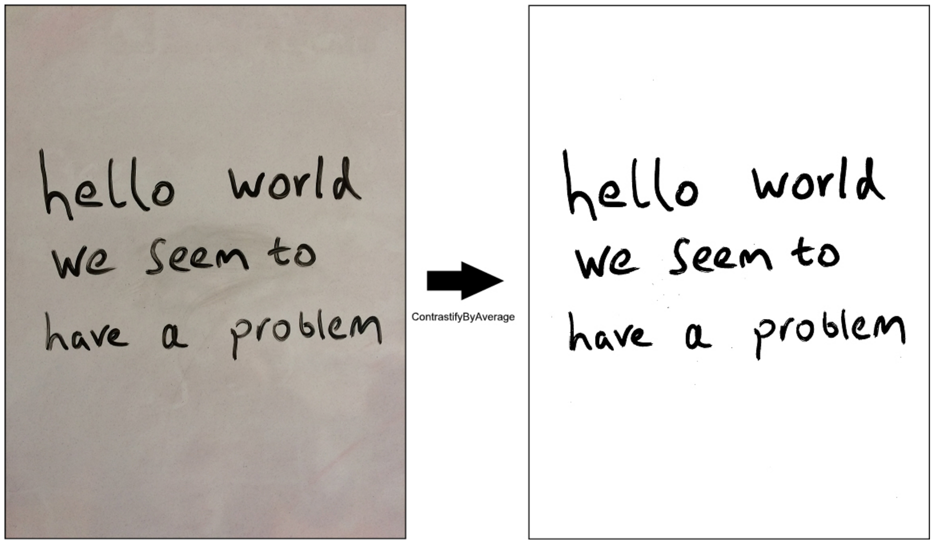


Figure 25: A successful application of `ContrastifyByAverage` on a noisy image.

The image in Figure 25 is fairly high contrast and clean, hence the algorithm succeeds. However, a key technical flaw of this naive algorithm is that it is not invariant to the size of the image's background relative to the size of the text in the image, nor does the algorithm deal well with varying background elements such as shadows, or low contrast due to poor lighting. A good example of a worst case scenario is presented in Figure 26.

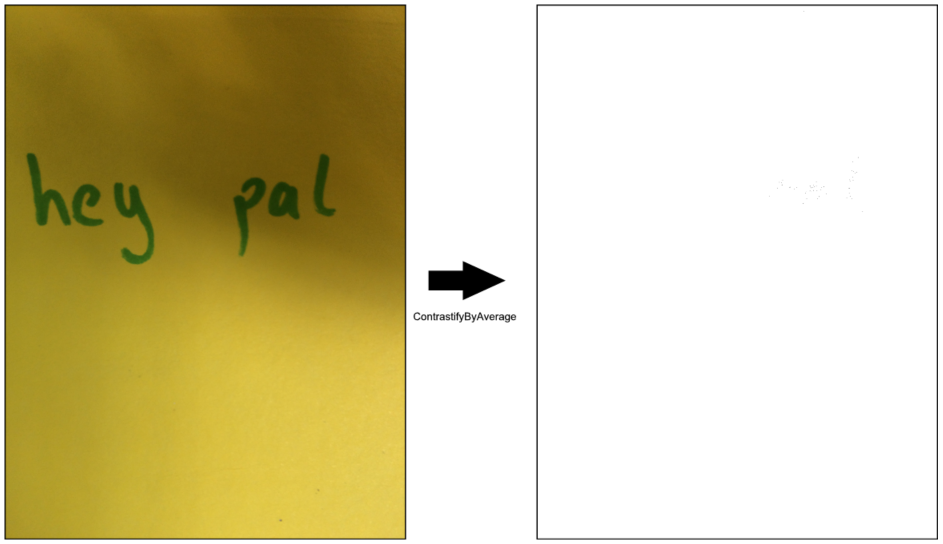


Figure 26: An unsuccessful application of `ContrastifyByAverage`.

The input image in 26 is non-standard: It is written in green pen on a yellow background; this reduces the contrast between the desired active and inactive regions. There is also the presence of a prominent shadow in the top-right of the image. We can see that `ContrastifyByAverage` fails drastically to denoise the image, almost entirely destroying the desired active regions in the process. In the top right, it is possible to see small artefacts of pixel regions that likely correspond to pieces of the text in the most intensely shadowed portion of the image.

The algorithm implemented by `ContrastifyByAverage` is therefore inappropriate for use in non-standard cases. One of its primary problems lies in that it examines the image globally, and applies a global property across the image. We require a smarter and localized method.

### Adaptive Thresholding

Adaptive thresholding is an umbrella term that refers to a number of similar algorithms that attempt to perform a thresholding operation similar to the one implemented globally in `ContrastifyByAverage` - the key difference is that this operation is applied locally, generally making such an algorithm far more robust. We implemented the proposed variant of the algorithm presented in [9] as an `ImageProcess` called `ContrastifyWithAdaptiveThreshold`. This version of the algorithm calculates the average intensity of pixels in a box of size $s$ around the currently examined pixel. We are required to provide another thresholding parameter, $t$ - the value of $t$ tells us what percentage below the average intensity we will set to black. The calculation of an average over every $s \times s$ rectangle in the image is an expensive operation, so instead the algorithm proposed in [9] calculates the integral of the pixel intensities from the image co-ordinate $(0, 0)$ to each pair $(x, y)$ and stores the result in a so-called integal image. This calculation can be done incrementally in a single pass over the input image. In the second pass, we can then calculate the average intensity in each $s \times s$ square around every pixel in the image quickly, since for any given rectangle $(x_1, x_2, y_1, y_2)$ we can obtain the sum of the pixel intensities as seen in 28. The pseudo-code for the algorithm is as follows (presented exactly as found in [9]):

```
for i = 0..width do
   sum ← 0
   for j = 0..height do
      sum ← sum+inputImage[i][j]
      if i = 0 then
         integralImage[i][j] ← sum
      else
         integralImage[i][j] ← integralImage[i-1][j]+sum
      end if
   end for
end for
for i = 0..width do
   for j = 0..height do
      x1 ← i - s/2
      x2 ← i + s/2
      y1 ← j - s/2
      y2 ← j + s/2
      count ← (x2 - x1)*(y2 - y1)
      sum ← integralImage[x2][y2] - integralImage[x2][y1]
      - integralImage[x1][y2] + integralImage[x1][y1]
      if (inputImage[i][j]*count) ≤ (sum*(100 - t)/100) then
         outputImage[i][j] ← BLACK
      else
         outputImage[i][j] ← WHITE
      end if
   end for
end for
```

Figure 27: The algorithm proposed in [9]. Border-checking for assignments to $x_1, x_2, y_1, y_2$ is omitted.

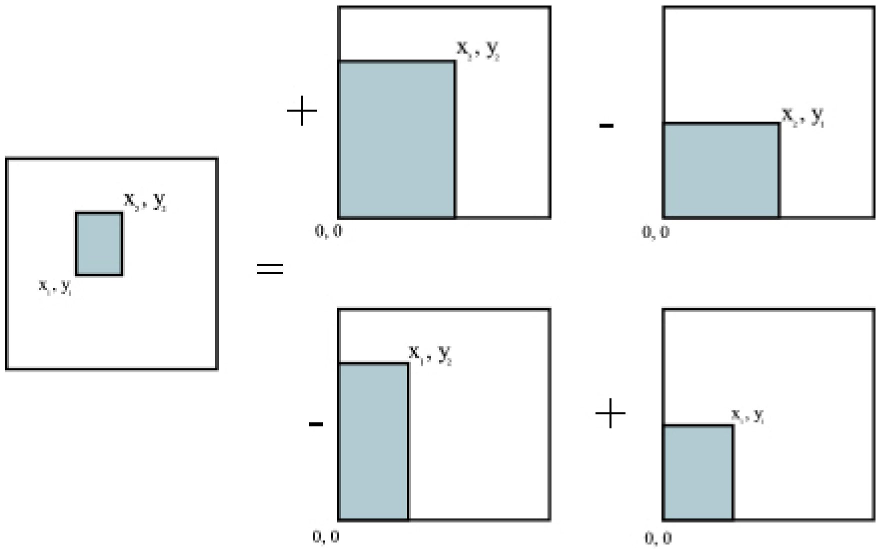


Figure 28: A diagram outlining how to calculate the pixel intensity sum of a rectangle $(x_1, x_2, y_1, y_2)$ given the integral image.

This variant of adaptive thresholding turns out to be much more powerful than our previous method. It adequately denoised all 20 out of 20 of our noisy examples. We found that on our testing set, the best values for $s$ and $t$ were given by $s$ set to an eighth of the image's smallest dimension, and $t = 14\%$. These are nearly identical to the values suggested in [9]. Figure 29 showcases the power of the method on the example that defeated our previous method.

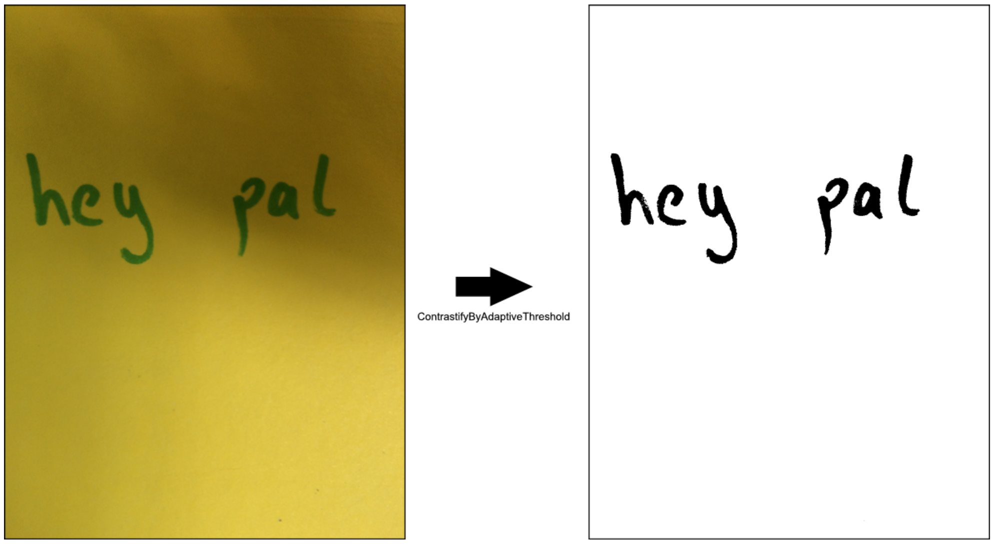


Figure 29: `ContrastifyByAdaptiveThreshold` succeeds without issue on the given image, despite the prominent shadow.

### 4.2.2 Preprocesses for Classification

#### A Colouring Procedure

In the very early phases of the project, we experimented with attempting to add information to a training set in order to boost the learning capacity of a classifier. Our primary idea was to colour the image of a character in some systematic way, such that we would effectively be adding a new dimension of information that the classifier could exploit in training. The `ImageProcess` which we implemented this idea in is called `ColourPlanes`, and it operates on an image by, beginning in the top left corner, spiralling in a clockwise direction and filling closed regions in a systematic way as it reaches them. However, this method actually negatively affected the training process - it seemed only to add noise to the images. This is likely because, despite our best efforts, the spiralling process was subject to variance depending on the position of the letter in the image.

## Framing the Character

When using a CNN to classify an image, it is necessary to define a standard size of input (since a given network has fixed architecture, and we have one pixel corresponding to one input neuron). We thus elected to initially use a size of $32 \times 32$, and so when we were training our first CNNs, we were preprocessing the training data into a $32 \times 32$ image. We did this naively: First, we fit an input image to its boundaries by removing all of the surrounding white space, this is implemented as the `FitToBoundaries` preprocess. Then, we resize the image to be input into the neural network by the preprocess `Resize(int width, int height)`. The results of the composition of these preprocesses are shown in Figure 30.

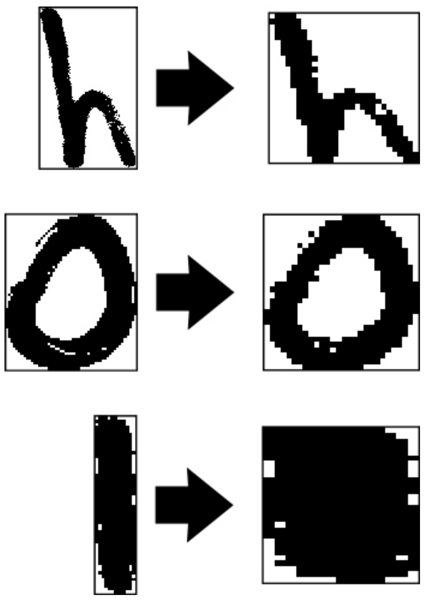


Figure 30: Applying `FitToBoundaries` and then `Resize(32, 32)` to two example character segments. The images on the right have, of course, been blown up from their $32 \times 32$ actual size.

Notice that the final example is degenerate in that its preprocessed form is almost a completely black square. When fed to a CNN, this black region is likely to have a high match rate with a large number of filters of different shapes and sizes (as discussed in 2.2.7). This will not only be problematic in that it will be difficult to train and find a filter that characterises this image, but also this image will activate filters that might otherwise be good characterisations for characters with different, more subtle features. Another issue with this method is that the feature detecting filters that the network learns are not necessarily computed near to the edge of an image, (recall from 2.2.7 where we talked about potentially needing to zero-pad). This means that a character such as the ‘h’ presented in Figure 30 could be confused with a ‘b’, since the CNN will have a harder time seeing whether the bottom loop is connected or not (in fact ‘h’ and ‘b’ were frequently confused when using this method of preprocessing). To attempt to solve both of these problems at once, we defined a preprocess `DefineSquareBoundaries` to be applied before we resize the image. The preprocess operates by framing the image on a square with a white boundary with a thickness equal to 30% of the smallest input image dimension. The results of applying this preprocess and then resizing are displayed in Figure 31.

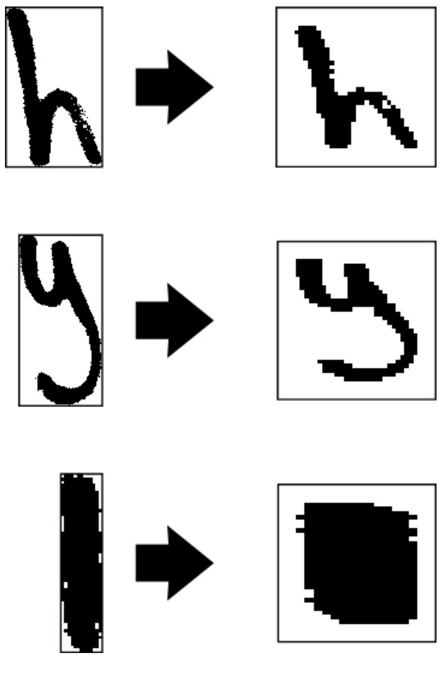


Figure 31: Here we see the effects of applying `DefineSquareBoundaries` before resizing.

For a given training time, we saw better performance in particular in the classifications of $y$, $g$, $i$, $j$ and $l$ classes. However, it would not be practical or sensible to give a numerical figure for an increase in accuracy when using `DefineSquareBoundaries`, since using it effectively changes the error-surface and the function that we are trying to approximate, meaning we cannot apply an independent test, since parameters that work for a network without this preprocess will differ from those that work for a network with it. We can however say that generally it *appears* to result in a smoother training process, and a classifier with a more desirably distributed confusion matrix.

## 4.3 Segmentation

### Fill-based Segmentation

At this stage of the pipeline, the system should be looking at a denoised and binarized image of the entire image of text to be transcribed. This image is passed to a `Segmenter` to be segmented. Finding whitespace separated segments is as simple as repeatedly selecting any black pixels in the binarized image and flood-filling that pixel with white, removing the black region and simultaneously recording the $x$ and $y$ values corresponding to the `left` (minimum $x$), `right` (maximum $x$), `bottom` (minimum $y$) and `top` (maximum $y$) of the newly filled region. The algorithm for whitespace separated segmentation is presented below:

```
workingImage ← copyOf(inputImage)
for x = 0..width do
    for y = 0..height do
        if workingImage[x][y] = BLACK then
            (left, right, bottom, top) ← getSegParams(workingImage, x, y)
            newSegment ← new Segment(left, right, bottom, top, inputImage)
            add newSegment to set of segments segmentSet
        end if
    end for
end for
return segmentSet
```

Where `getSegmentParameters(workingImage, x, y)` is the flood-fill operation. We operate on a copy of the image because the `getSegmentParameters` algorithm removes the active region of the segment from the image by filling it with white (to avoid finding the segment more than once). However, we need to keep the input image intact since it is used in the constructor of a `Segment`. Both the recursive and iterative form of the `getSegmentParameters` function are relatively simple to implement, so we implemented both in order to see which was better. The iterative method was both faster and less error-prone, since the recursive version can potentially run out of stack space if the input image was large enough. The iterative version of `getSegmentParameters` can be written as follows:

```
bottom ← y
top ← y
left ← x
right ← x
add (x, y) to set of points fillSet
while fillSet is non-empty do
    (i, j) ← an element of fillSet
    bottom ← min(y, bottom)
    top ← max(y, top)
    left ← min(x, left)
    right ← max(x, right)
    image[i][j] ← WHITE
    add any new black pixels adjacent to (i, j) to
fillSet.
end while
return (left, right, bottom, top)
```

The algorithm removes the connected black region from the image, ensuring termination of the segmentation algorithm, whilst also recording the bounding box of each segment. Thus, we can create our set of `segments` as discussed in 3.2.

## 4.4 Layout Analysis

We performed a whitespace segmentation in the previous step, and now the system has a set of segments, some of which may need to be conjoined in order to form a full character image. In the initial discussion of these problems (1.6), this joining operation is part of the Character Segmentation step. However, in practice, it turns out to be a better idea to do line segmentation first.

### 4.4.1 A Fundamental Problem for Layout Analysis

During the development of the system, it quickly became clear that layout analysis, as a problem on its own, is difficult as it is generally hard to be robust to any possible case. The primary source of the problem, as briefly mentioned in 1.6, is essentially that the system is looking at a bunch of bounding boxes (or contours) on the page, with no knowledge about what these elements *are*, just that they need to be grouped according to some scheme. For line segmentation, the system is looking to establish an ordering on the segments, and then to distinguish which segments lie in which line - the definition of a line needs to come from the positioning of the segments alone. We will see in a moment that this can be problematic if lines are malformed in the sense that they come close to overlapping. Assuming that line segmentation is successful, we come to an arguably even more difficult problem in that it can be somewhat unclear, based solely on the positions of the character segments, where word spaces are supposed to occur. This is particularly problematic in our system, as the hypothesis-based method requires us to be able to hypothesise a realisation of a given word, and this is not possible if we are unsure of which characters belong to the word. These problems, and our approaches to tackling them, are discussed in the next two subsections.

### 4.4.2 Line Segmentation

Recall from 3.2 that at this point in the system, we construct an instance of `TranscriptionData`, which can be thought of as a tuple `(segments, lineDividers, wordDividers)`. Line segmentation is done by a `LineParser` whose `parse(TranscriptionData)` method, if successful, returns the modified `TranscriptionData`, where `segments` will now be ordered as they are naturally read on the page, and `lineDividers` contains all of the integers corresponding to the indices of `segments` representing the final segments in each line.

The standard algorithm to use for line segmentation comes from printed-font OCR. It relies on the existence of obvious minima in the horizontal pixel histogram of the input image.

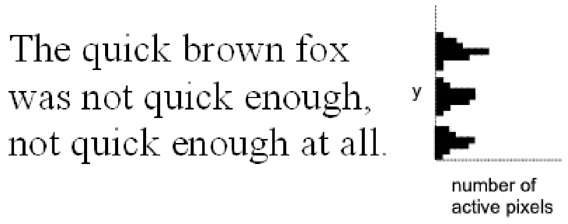


Figure 32: A simplified horizontal histogram for a small scanned and noise-reduced typeface document.

In the above diagrams we can see that line segmentation is as trivial as looking for whitespace in the horizontal pixel-sum histogram of the document. This method can work with handwritten sentences, so long as the writer abides to the constraints of the algorithm (ensuring that there is a detectable minimum in the horizontal histogram, or even distinct vertical whitespace between each line). Figure 33 showcases a simple case where these constraints can be easily violated, such that the required minima disappear.

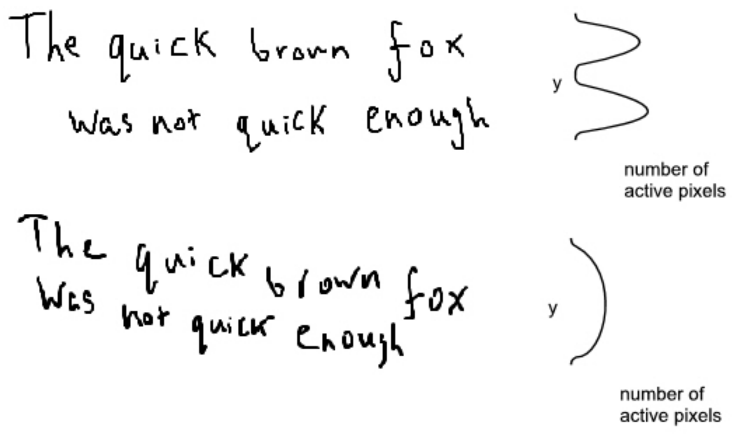


Figure 33: Top - a possible horizontal histogram shape for a small handwritten document. Bottom - a possible histogram for a handwritten document where strict adherence to writing in straight lines has not been followed.

As we can see, the target minima disappear if the lines are slanted (even if there is still a distinct non-vertical whitespace difference between the lines). Another problem is the overlapping of long-tailed letters into lines that they do not belong to. It can be difficult to tell where a letter belongs based solely on its bounding box.

Our proposed solutions to these problems involved algorithmically defining the lines segment by segment, i.e. in a more local manner. We came up with two solutions, but only implemented one since preliminary testing by hand showed one of the solutions would not suit our needs in practice.
The first proposed solution required us to define a 'centre of mass' for a given binarized character image. The 'centre of mass' or 'mass point' is defined as the average active coordinate in the pixel image. i.e.

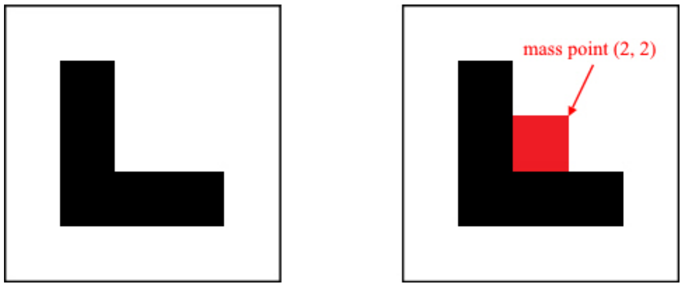


Figure 34: Left - an upscaled $5 \times 5$ character image. Right - the red pixel corresponds to the calculated mass point.

The formula to calculate the mass point is easily expressed:

$$\text{round}\left(\frac{\sum\limits_{(x,y)\,active}(x,y)}{\#active\,pixels}\right)$$

The calculation in Figure 34 is thus given by

$$\text{round}\left(\frac{(1,1)+(1,2)+(1,3)+(2,1)+(3,1)}{5}\right)$$
$$= \text{round}\left(\frac{(8,8)}{5}\right) = \text{round}\left((1.6,1.6)\right) = (2,2)$$

Finding lines of text is then as simple as the following algorithm which operates on `TranscriptionData = (segments, lineDividers, wordDividers)`:

```
seg ← getTopLeftSeg(segments)
segments.remove(seg)
add seg to newSegments
minDistance ← 0
while segments is non-empty do
    for candidateSeg in segments do
        segDistance ← distance(getMassPoint(seg), getMassPoint(candidateSeg))
        if segDistance < minDistance then
            minDistance ← segDistance
            nextSeg ← candidateSeg
        end if
    end for
    if minDistance < THRESHOLD then
        segments.remove(nextSeg)
        add nextSeg to newSegments
        seg ← nextSeg
    else
        add newSegments.length - 1 to lineDividers
        seg ← getTopLeftSeg(segments)
        segments.remove(seg)
    end if
end while
segments ← newSegments
```

Figure 35: The mass point algorithm for defining lines. Edge-case handling for when we only have a single segment has been omitted for clarity. `THRESHOLD` is a predefined parameter that ensures that two segments in the same line must be within a minimum distance.

This version of the algorithm is unsuccessful for two reasons. Firstly, it makes the assumption that the mass points of the next character in the line will be closer to the current character's mass point than the mass point of a character in the line below. This is simply not always the case. This can be solved by changing the `getDistance` function so that it no longer computes a Euclidean distance, instead giving preference to a difference with a smaller horizontal component, i.e. we could have for example `getDistance((x, y), (i, j))` $= \sqrt{((x-i)^2 + 2 * (y-j)^2}$. The second problem is not so easily surmountable: intuitively it seems that the mass point is possibly a good measure of the location of a character in a document, however, in practice - it is not. Take for instance a letter with an exceptionally long tail, like the one seen in Figure 36.

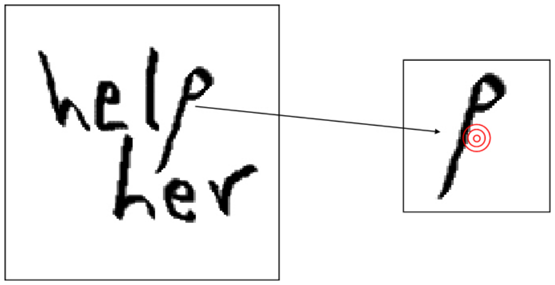


Figure 36: Left: An isolated example with which the mass point approach is likely to have an issue. Right: Approximate location of the mass point of this character image.

This method can be successful assuming a general degree of well-formedness, i.e. that lines are separated enough, and characters on the same line close enough - however, when it does go wrong it can be difficult to pinpoint where the input image is malformed, since it is hard to ascertain by eye where the mass point of a character image should be, and it is therefore even more difficult to work out each of the relevant distances. So far then, we have the horizontal histogram approach which has very strict constraints on the form of lines in the input image, and we have discussed an approach that reduces these constraints, but is still prone to failure - and, importantly, when it does fail, it is difficult to correct so that the next stage of the pipeline can be allowed to continue unhindered for testing purposes.

Our second proposed algorithm does not suffer this setback. We implemented this second algorithm in a `Parser` named `FrontrunnerLineParser`. While the algorithm is less constrained than the horizontal histogram approach, it still expects a degree of well-formedness. However, its constraints are much more clearly defined than the mass point method, and this makes it more suited to our needs. The method is very simple and yet highly effective, successfully performing line segmentation on 36 of our 40 test examples. We present the algorithm here:

```
averageSegmentHeight ← getAverageSegmentHeight(segments)
sort segments by their horizontal position, from left to right.
seg ← getTopLeftSeg(segments)
add seg to newSegments
remove seg from segments
topBound ← (getAverageSegmentHeight(segments)/2) - seg.y
bottomBound ← (getAverageSegmentHeight(segments)/2) - seg.y
while segments is non-empty do
    if there exists a segment cSeg s.t. seg.y - bottomBound ≤ cSeg.y ≤ seg.y + topBound, cSeg to
the right of seg then
        nextSeg ← leftmost such cSeg
        add nextSeg to newSegments
        remove nextSeg from segments
        if HEIGHT_FAC_HI*(topBound + bottomBound) ≥ nextSeg.height ≥ HEIGHT_FAC_LO*(topBound
+ bottomBound) then
            topBound ← nextSeg.top - nextSeg.y
            bottomBound ← nextSeg.y - nextSeg.bottom
        end if
        seg ← nextSeg
    else no such segment cSeg exists
        add newSegments.length - 1 to lineDividers
        seg ← getTopLeftSeg(segments)
        segments.remove(seg)
    end if
end while
segments ← newSegments
```

Figure 37: Our best line segmentation algorithm. Once again, handling the case where we have a single segment leftover in one line is omitted for clarity.

For the purposes of this algorithm, the co-ordinates of a segment are considered to be the co-ordinates of the centre of its bounding box. The algorithm uses a pair of bounds, `bottomBound` and `topBound` around the current letter on the line to decide the closest letter that could be considered to naturally follow on from the current one (see Figure 38). The algorithm also asserts that these bounds need to be of a reasonable size, so that the size of the underlying segment does not cause the bounding region to become excessively small or large (recall that, at this stage, we have not connected disjoint components of characters yet, so if we come across the dot of an ‘i’ or ‘j’, we do not want the acceptable region to be defined by the height of this character, as it will almost certainly miss the next character). This assertion is made through the use of the variables `HEIGHT_FAC_HI` and `HEIGHT_FAC_LO` which ensure that the bounding box cannot grow by more than a factor of

HEIGHT_FAC_HI and cannot shrink by more than a factor of HEIGHT_FAC_LO at each step. We set both factors to 50% for the purposes of our final implementation and testing. This algorithm essentially walks down a line of characters and selects the next character in the line by checking if its midpoint is in the given bounds.

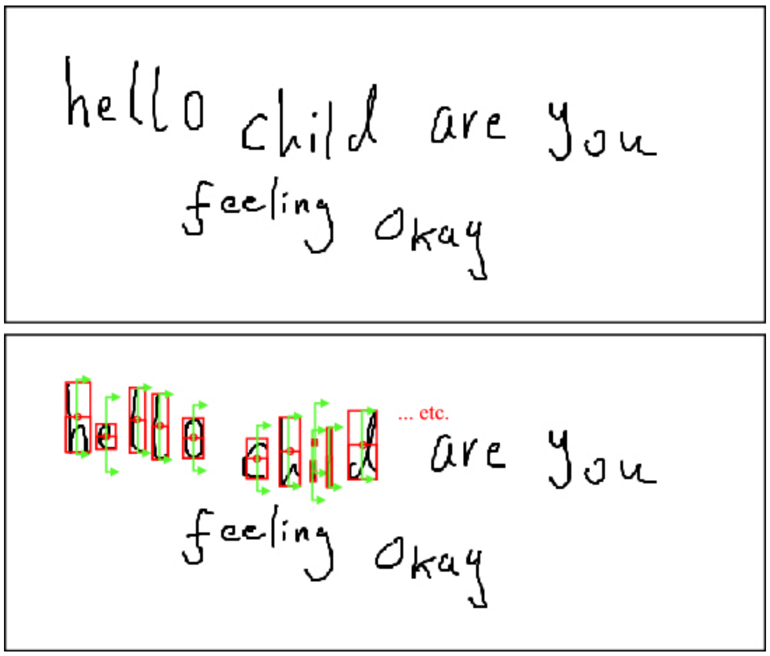


Figure 38: The operation of the FrontrunnerLineParser on the first few characters of an example.

The red boxes in Figure 38 correspond to the bounding boxes of the segments, and the green height bounds correspond to the values topBound and bottomBound used in the algorithm. Notice that the midpoints of each bounding box lie within the green region, and so each successive character will be added to the line. Notice also that the size of the green height bound only changes when the next letter is of similar size to the previous one, allowing the algorithm to correctly include smaller characters, and even tiny segments such as 'i' dots into the line.

We can then succinctly state the constraints of this algorithm as the following: to ensure the algorithm performs its operation successfully, the midpoint of each successive character segment must lie within the given bounding region around the midpoint of the previous character segment. It is much easier to see by eye roughly what this bounding region should be at any point on the line, thus allowing a better grasp of the algorithms reasons for failure, in the cases where it does fail. This algorithm can handle slanted lines much better than the horizontal histogram approach, but it is still somewhat susceptible to failure if a given input image has particularly pronounced and invasive tailed letters spanning multiple lines.

### Conjoining Segments

We are then required to connect any disconnected character segments. Since we are only considering the lower-case alphabet, we know that the only valid characters with whitespace separated elements would be 'i' and 'j'. Since the two pieces of these characters are predominantly vertically displaced, and at this stage we can assume that we have successfully performed line segmentation, we can simply connect any segments that lie vertically displaced from one another by following the simple heuristic presented in Figure 39. Since we have done line segmentation, we can consider the joining problem line by line, and thus avoid the risk of accidentally conjoining two segments that lie vertically displaced and close to each other but semantically are in different lines of the text.

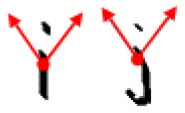


Figure 39: For each segment in a given line, we search for isolated segments that lie completely within a region such as the one presented in the diagram.

Once it has been decided by this heuristic that two segments need to be conjoined, a call to the `Segment` method `conjoin(Segment)` is made to turn the two segments into a single segment whose bounding box is the union of both segments' bounding boxes. The `segments` field in `TranscriptionData` is then updated to contain this new segment in place of the others, maintaining the read-order and decrementing `lineDividers` where necessary (as when joining two segments the size of `segments` decreases by one, and `lineDividers` correspond to indices of `segments`).

### 4.4.3 Word Segmentation

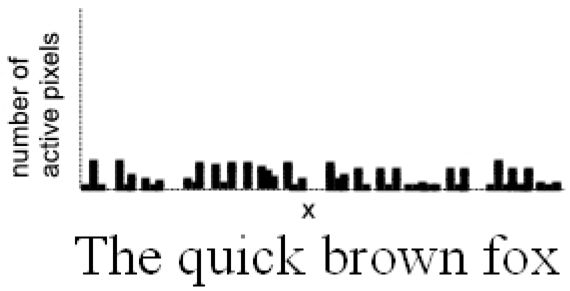


Figure 40: The vertical pixel histogram for a line of printed-font text.

As we can see in Figure 40, even in the case of printed-font OCR, word segmentation is not as simple as looking for zeroes or minima in the vertical histogram of the line, since these occur internally for the majority of words. However, the size of a character space are known for a given font, so in some cases it is sufficient to simply look for zero-regions above a given width. This method would not be effective at all in the case of grouping characters of handwritten words as the width of a word-space and character-space can vary greatly. In fact, as we began to discuss in 4.4.1, segmentation of words is quite a difficult problem. The problem essentially comes down to the fact that the overarching system is trying to solve the problem of recognising words and segmenting words in a serial manner, when the two problems in reality seem to require an inherently concurrent approach. Figure 41 demonstrates this problem.

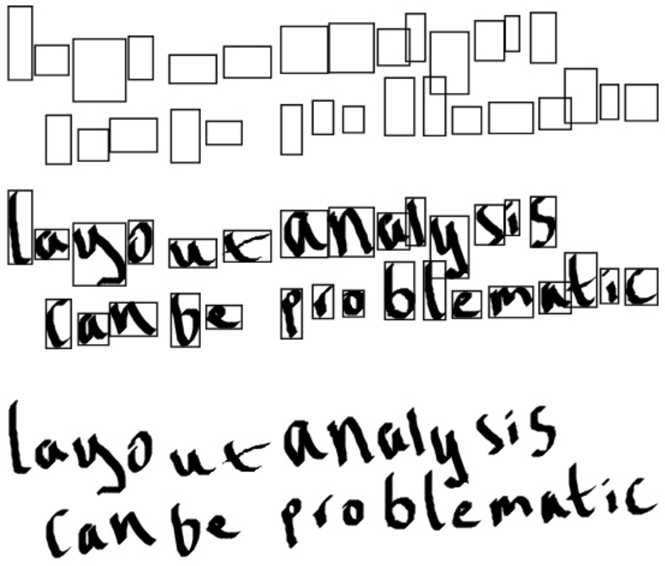


Figure 41: Top - Segments corresponding to a group of written words. Bottom - The sentence clearly reads “layout analysis can be problematic”. Deducing the word spacing from the segments alone is difficult.

It seems that in order to perfectly segment the written words, we ought to already be able to read them, but the system requires segmentation of the words in order to be able to attempt to read them in the first place! In theory, word spaces exist so that we (humans) may syntactically distinguish which characters belong to which word: spaces delimit text, making it easier to read one word at a time. In practice however, it is common to relax these constraints somewhat, because the semantics of what is written make clear which character belongs to which word (as seen in Figure 41). There is also the added complication that humans inscribe some natural ‘position’ to each written letter that is not possible to algorithmically deduce (we attempted and failed to do this in our mass point approach). For example, the location of the letter ‘p’ on a page should probably be considered as the centre of the loop - but this is highly dependent on the context and form of the written letters. This positioning scheme makes it easier for humans to syntactically distinguish words, but a system such as ours only has the bounds of the character segment to work with. We implemented two methods that ignored the problem and assumed the input image to be syntactically well formed, and designed two methods (one implemented, one proposed) to be slightly more resilient to this problem.

Assigning a word segmentation once it has been decided is simple within our system. Given some `TranscriptionData` with `segments` ordered in read-order and the correct end of line positions stored in `lineDividers`, to perform word segmentation we need only to insert the correct end of word positions into `wordDividers`.

### Word Segmentation By Line Average

The `Parser WordParserByLineAverage` implements a naive technique for selecting word spaces by simply calculating the average horizontal distance between the segments in each line, and then assigning a word space wherever the horizontal distance is larger than this average by some factor `threshold`. The algorithm looks like this:

```
lineStart ← 0
lineEnd ← lineDividers[0]
while lineStart is less than the last element of lineDividers do
   averageHozDistance ← getAverageHorizontalDistanceForLine(segments, lineStart, lineEnd)
   for i = lineStart..lineEnd - 1 do
      if    getHozDistance(segments[i], segments[i + 1]) > threshold*averageHozDistance
then
         add i to wordDividers
      end if
   end for
   lineStart ← lineEnd
   lineEnd ← next element of lineDividers (if any).
end while
```

The algorithm is not very effective - even with our best experimental value for `threshold`, it only successfully segmented 11 of the 40 test examples. This is not surprising, as word spacing is highly variable and very much dependent on the writer/what is being written.

### Smudging/Blurring

A better approach, which is arguably the simplest standard approach given in the literature, is to apply an image transformation to the binarized original input image such that characters that are closer together become connected, and characters that lie at a greater distance (corresponding to a word space) remain disconnected. The segments are then easily grouped into words by placing all characters that become connected after smudging into the same word. The image transformation is most often a Gaussian blur. For the purposes of this document, when we refer to a smudge we are referring to a blur-like operation that does not reduce the transparency of the original elements present in the image.

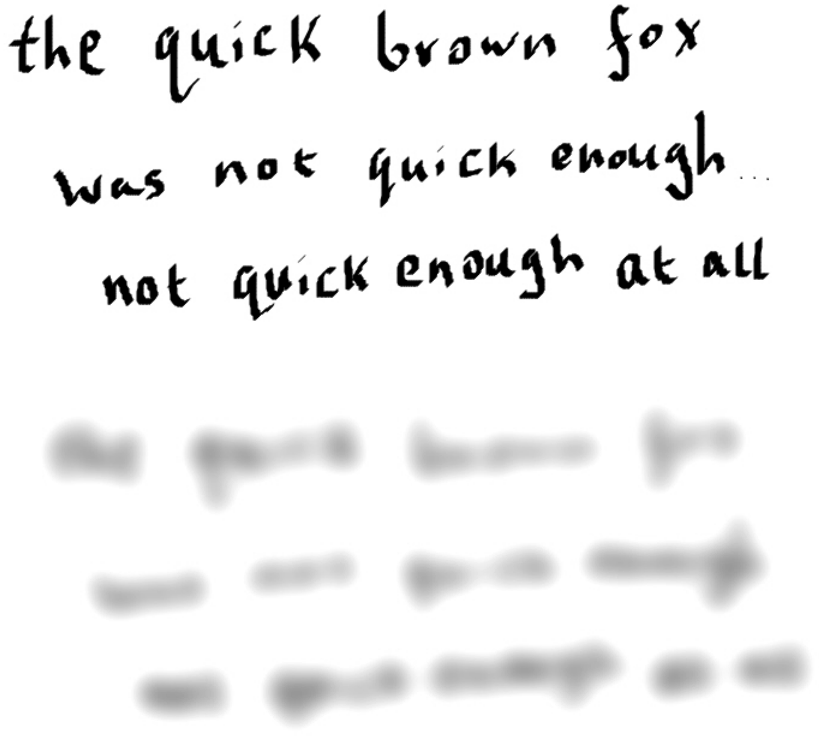


Figure 42: Top - The input image pre-smudge. Bottom - The result of smudging.

When performing a word segmentation by smudging it is necessary to define the strength of the operation. This is done via a field `smudge_fac` in

WordParserViaSmudge. The value of smudge_fac (between 0 and 1) can be thought of as the percentage of blackness transferred to a neighbouring pixel.

This approach fared better than the previous one, correctly segmenting 21 of the test examples with our best smudge_fac of 0.86. The method still suffers from the issue with word segmentation that we discussed at the beginning of this section. Note that in practise it is the horizontal distance between characters that defines word spacing, and thus we made the Gaussian blurring computation more efficient by simply computing and smudging a vertical histogram for the input image. This idea is implemented in our Parser named WordParserViaSmudge.

### Smudging Over An Interval

This method is essentially a brute force approach to performing word segmentation, and it attempts to attack the problem with the concurrency of recognition and word segmentation. It is implemented by co-ordinating a DictionaryAnalyser to work with the WordParserViaSmudge class. We define a special method for WordParserViaSmudge called specialParse(TranscriptionData, double) which allows us to update the smudge_fac and then smudge the vertical histogram incrementally. This is all tied together in the HandWritingRecogniser class. We choose an interval for the value of smudge_fac to range over. We then incrementally perform the smudging, and proceed to the classification step. We assign a score based on the resulting output of the recogniser (taking into account how much of the output makes sense relative to a dictionary, as well as the probability distributions present in each FuzzyChar). We then take the output with the maximum score to be the correct word segmentation, and at this point the system has finished its computations since the classification and recognition steps have also been completed. In effect, what we are doing here is hypothesising various possibilities for the structure of the words in the image, and taking the best one we can find. Of course, this method of word segmentation is expensive and can take a significant amount of time to compute, because really we are attempting to recognise the same image numerous times. The fact that we implement the smudge in 1-dimension over the vertical histogram helps to speed things up, but the method is still relatively slow. In section 4.6 we will go into more detail about how we score such hypotheses within the system.

With regard to evaluating this method, in theory it should be capable of achieving a perfect score on our test set - assuming we could make the interval arbitrarily fine-grained. However, we chose to test using an interval with 8 values ranging from 0.86 to 0.93, which resulted in 35 of the 40 test examples being segmented correctly within a reasonable time frame.

### Separating Distributions

The key flaw in the previous method is that it is not intelligent in its traversal of the search space. Ideally we would like to be able to find a method that in some sense converges to an integer such that if two segments are horizontally displaced by a width greater than this integer, then that space is a word space, and otherwise it is a character space. The following proposed procedure assumes that character spaces and word spaces belong to separate normal distributions of horizontal width in pixels, and attempts to converge to a value that best separates these distributions. The following method operates on `TranscriptionData = (segments, lineDividers, wordDividers)`. Again, the method presented here is supposed to tackle the issue with the problems of segmentation and recognition being connected, and as such requires that the rest of the recognition system has been constructed. See section 4.5 for the details of the classification and scoring process.

```
sep ← getAverageSegmentWidth(segments)
maximal ← false
i ← 0
while !maximal && i < limit do
   wordDividers ← getWordDividers(segments, sep)
   score ← getScore(output)
   leftSep ← sep - increment
   rightSep ← sep + increment
   wordDividers ← getWordDividers(segments, leftSep)
   leftOutput ← recognise(segments, lineDividers, wordDividers)
   wordDividers ← getWordDividers(segments, rightSep)
   rightOutput ← recognise(segments, lineDividers, wordDividers)
   leftScore ← getScore(leftOutput)
   rightScore ← getScore(rightOutput)
   if leftScore < score && rightScore < score then
      maximal = true
   else if leftScore >= score && rightScore < score then
      sep ← sep - (leftScore/score)*(increment)
      output ← leftOutput
   else if leftScore < score && rightScore >= score then
      sep ← sep + (rightScore/score)*(increment)
      output ← rightOutput
   else if leftScore >= score && rightScore >= score then
      if leftScore >= rightScore then
         sep ← sep - (leftScore/score)*(increment)
         output ← leftOutput
      else if leftScore < rightScore then
         sep ← sep + (rightScore/score)*(increment)
         output ← rightOutput
      end if
   end if
   i ← i + 1
end while
return output
```

Figure 43: Our proposed algorithm for the desired convergent behaviour. Note that the case where there is no change in the score on either side is omitted for clarity.

This algorithm is inspired by the concept of gradient descent. We adjust the `sep` parameter (corresponding to the boundary between what constitutes a character space, and what constitutes a word space). We adjust the parameter `sep` by iteratively checking whether it is likely to be more beneficial to travel in the negative or positive direction. We cannot, however, explicitly compute a derivative in this case, since, for one, we do not know the function, and in fact we do not even know if it is differentiable. The algorithm is hence not guaranteed to find the best possible separator, but it is arguably a much better heuristic approach than the trial and error in the previous method.

We did not implement this algorithm as it requires at least 3 full recognitions to occur per loop and so would not be much more efficient on our test set than the previous brute force approach which worked well with only 8 recognitions. We present this algorithm solely to highlight the notion of separating the distributions of character and word spaces, and as another example of a concurrent solution to the recognition/segmentation problem. This algorithm would operate most effectively if the distribution of horizontal spaces looks like the one in Figure 44.

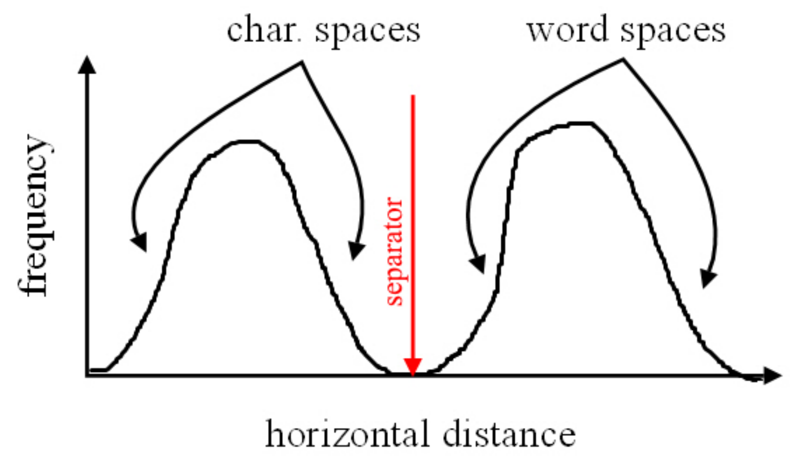


Figure 44: The distributions we see here have a well-defined separation between character and word space.

In Figure 45, we see a distribution of horizontal spaces that would make it impossible for any algorithm to make an intelligent decision as to where the separator might lie (because it does not exist). An example of text with a distribution that looks like this would have character spaces that were in some cases larger than some of its word spaces. In this case, segmentation and recognition are completely interwoven, and methods based on explicitly trying to find a separator will not work. Regardless of our technique here in the word segmentation step, we require that word spaces are separable from character spaces in order for our system to have a chance at successful recognition.

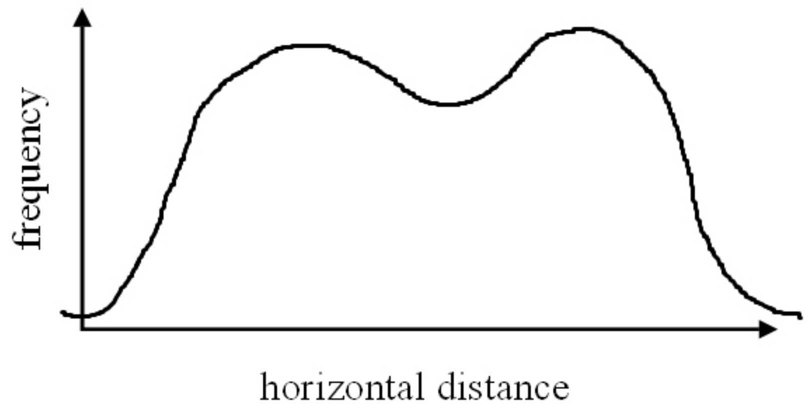


Figure 45: In this case, we can see that it is impossible to separate the distributions with a well-defined separator.

## 4.5 Classification

Assuming that the previous components in the pipeline have completed their jobs successfully, the system now has a piece of `TranscriptionData` whose `segments` are in read-order, and whose `lineDividers` and `wordDividers` contains each integer index corresponding to the end of each line and each integer index corresponding to the end of each word, respectively.

### 4.5.1 Building and Training Classifiers

We used the library `Encog` [15] to build our first classifiers. These were feed-forward networks, the most basic form of neural network architecture - each layer of neurons in a feed-forward network is fully connected. These networks are not specialised for any learning task, and thus they struggled to learn the full lower-case character set. This is what sparked our research into convolutional neural networks as it would be almost impossible to obtain any meaningful data from a classifier that did not have at least some workable baseline accuracy (otherwise the output with and without the hypothesis-based method would effectively be random noise).

We used the library `DeepLearning4Java` [16] to construct CNNs. We outline our training procedure for a neural network below:

1. Obtain a large dataset of examples.
2. Split the dataset into disjoint training, validation and test sets (a $60-20-20$ train-validation-test split is appropriate).
3. Fix a network architecture and fix some training parameters.
4. Train the network using the back propagation and gradient descent algorithm (2.2.5) with the training set.
5. Test the network on the unseen validation set, obtaining the confusion matrix, and compute useful values such as overall classification rate and per-class $F_1$ scores.
6. If the results are unsatisfactory, use these statistics (and the training progress graphs provided by `DeepLearning4Java`'s visualisation classes) to make adjustments to the network architecture (if necessary) and training parameters, then go to 4. However, if the results are satisfactory then go to 7.
7. Test the trained network on the unseen test set. The classification rate obtained from this test should be taken as the classification rate of the trained network. If the statistics at this stage are unsatisfactory, repeat the entire training process.

### 4.5.2 Utilising the LeNet-5 Architecture

Recall the LeNet-5 architecture discussed in 2.2.10. This architecture only hs 7-layers, but is effective at recognising the `MNIST` dataset of handwritten digits. This hints that the architecture is likely capable of recognising handwritten characters (in fact the classifier was originally intended to be used as part of a document recognition system [11]). When fixing an architecture in our training procedure, the starting architecture was always that of LeNet-5 with 26 output neurons instead of 10 (one for each letter in the alphabet, as opposed to one for each digit). We also elected to use the softmax activation function over our output layer.

The softmax activation function is given by $softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}$ where $\overline{x} = (x_1, x_2, ..., x_k)$ is the vector corresponding to each of the inputs to the $k$ output neurons.

A key property of this activation function is that it ensures the network outputs a distribution-like output over the set of characters to be recognised. Thus, this network, if trained well, could learn to approximate a likeness-function like the one discussed in 1.7.

### 4.5.3 Obtaining Data

Initially we constructed our own relatively small data set consisting of 100 examples for each letter, for a total of 2600 total examples. These examples were written by a group of 4 writers, and did not include the letters ‘i’ or ‘j’. Despite this, networks which were not trained to recognise ‘i’ and ‘j’ still did surprisingly well when we applied our hypothesis-based approach. Networks were quicker to train on this smaller dataset, but were typically less accurate than those we managed to train later. In the later stages of the project, we obtained our training, validation and test examples from the NIST data set of handwritten characters [17]. This dataset was written by over 3600 writers, (although we did not use the full dataset).
Initially we used 52,000 images (2000 per character) in our example set. However, the first few networks we trained with this number of examples were struggling immensely to classify, in particular, the characters ‘i’ and ‘j’. After taking a look over the training set, we found that the dataset was

substantially noisy - there were what appeared to be ‘j’ characters in the ‘i’ folder and vice versa. This meant we had to manually clean the dataset, until there were about 800 ‘i’ characters left over. We were thus forced to reduce all of the classes to a size of 800 to avoid having a minority class [3] This resulted in our final dataset consisting of around 21,000 examples.

### 4.5.4 Discrete Classification

Over the course of the project, we trained upwards of 500 convolutional neural networks using `DeepLearning4Java`’s visualisations. The training processes for a standard `ImageClassifier` and a `FuzzyClassifier` are identical. In fact, if we have previously trained and saved an `ImageClassifier` to a file, then it is possible to load that network into the system as a `FuzzyClassifier` to be used for the hypothesis-based method presented in the next section.
Discrete classifiers were trained by creating a file structure such that each training example lies in a folder whose name corresponds to the `ImageClass` that the example belongs to. We then call pre-existing methods within `DeepLearning4Java` to initite the training process over this file structure. To discretely classify our `TranscriptionData`, all we need to do is to call `classify(Image)` on the image of each character segment. The algorithm used to build the discrete output string from `TranscriptionData` is presented here:

```
output ← ""
for i = 0...segments.length - 1 do
    output ← output + classify(segments[i].image)
    if wordDividers contains i then
        output ← output + " "
    end if
    if wordDividers contains i then
        output ← output + "\n"
    end if
end for
return output
```

[3]If a network is trained on a set of examples with one class being in a significant minority, the training process will typically result in this class being ignored completely, since it has a lower weighting on the overall error-function.

### 4.5.5 Fuzzy Classification

For a given character segment's image, a call to `fuzzyClassify` will return a `FuzzyChar` that can be interpreted as what the network believes to be a suitable probability distribution over the set of `ImageClass`es that the character image could belong to. It does this by taking the output value of the $n^{th}$ output neuron as the probability that the input image corresponded to the $n^{th}$ character `ImageClass` (in alphabetical order). This allows `fuzzyClassify` to behave as a likeness function, upon which we can base our hypothesis-based approach.

It should be noted that, whilst the training process for a `FuzzyClassifier` is the same as for a standard `ImageClassifier`, a good `FuzzyClassifier` is not necessarily a good discrete classifier, and vice versa. Generally, a discrete classifier's only aim is to perfectly classify every single training example with 100% accuracy. This aim is unachievable since there will always be some element of unexpected noise in the training set or in real world examples fed to the classifier. Regardless, the focus when training a discrete classifier is typically on obtaining the highest possible classification rate. However, a high classification rate does not necessarily guarantee an objectively good `FuzzyClassifier`, because the aim of a `FuzzyClassifier` is just to output a sensible probability distribution. If given an image of a slightly strange looking 'e', it is not necessarily a bad thing if a `FuzzyClassifier` outputs $(c = 0.5, e = 0.49, ...)$ because the idea is that the classifier has recognised that the image looks like an 'e', it simply also has properties that align it with the character 'c', according to the classifier. A bad `FuzzyClassifier` would be one that frequently confuses classes that are not alike in any way - for example, if it confuses 'g' with 'u', then this would likely be a problem. Notably, one of the best `FuzzyClassifier`s we trained only has a class accuracy of 65% on the class 'i', but almost all of the other 'i' elements in the test set are classified as 'l', which is understandable if we consider the character's visual features. With this particular classifier, whenever an image is classified as an either an 'i' or an 'l', the alternative class will often appear with a non-negligible probability in the probability distribution - this is a good thing, even if the classifier gives the wrong `ImageClass` a higher probability. This is exactly the type of behaviour that our hypothesis-based approach attempts to exploit.

### 4.5.6 Fuzzy Strings

Given that in this point in the system, we have a piece of `TranscriptionData` that completely describes the text located in the input image in terms of ordered `segments` and the positions of newline characters and spaces (via `lineDividers` and `wordDividers`, we can concatenate `FuzzyChar`s which belong to the same word to obtain a `List<FuzzyChar>` which can then be passed to the constructor of a `FuzzyString`. The system can then use these `FuzzyString`s as follows:

1. Get the $n$ most likely strings corresponding to a given `FuzzyString`.
2. If any of these strings are words according to our stored dictionary then realise the `FuzzyString` as that string and move on to the next `FuzzyString`, otherwise go to 3.
3. For each string in the list of the $n$ most likely strings, get a list of words that are within a suitable Levenshtein[4] distance from each string. Concatenate these $n$ lists so that we have a single list of possibilities for the given `FuzzyString`.
4. Assign a score to each of the words in this list of possibilities, and pick the highest score string to be the realisation of the `FuzzyString`.
5. Repeat for the next `FuzzyString`.

This algorithm effectively amounts to being an efficient traversal of the given dictionary as a search space. Rather than run through every single element of the dictionary and calculate a score for each, we hypothesize realisations of the `FuzzyString` as starting points, and then make our guesses over a much smaller targeted subset of the words in the dictionary. Notice that we still need to compute Levenshtein distances, which (naively) requires a computation over the whole dictionary, but this computation is well-studied and can be optimised [13], as well as being less-expensive than explicitly calculating a probability distribution for large dictionaries.

[4]Levenshtein distance of two strings is defined to be the number of characters in which they do not match, i.e. levDist("hi", "ho") = 1, levDist("umbrella", "um") = 6, levDist("water", "water") = 0.

**Obtaining the $n$ Most Likely Strings**

Given a `FuzzyString` we required an efficient algorithm for acquiring the $n$ most likely realisations of this string. Since a `FuzzyString` of length $k$ is effectively a list of $k$ distributions over the 26 possible character classes, this problem is equivalent to finding the $n$ realisations with highest probability (with respect to the joint distribution) of a vector $(char_1, char_2, ..., char_k)$ of random variables, given the marginal distributions of $P(char_1), P(char_2), ..., P(char_k)$. Note that to do this we assume that the random variables are linearly independent.

The naive way to compute this would be to simply calculate and sort the joint distribution, taking the top $n$ elements after we have finished. However, there are $26^k$ elements to compute and then sort - which makes this a terrible idea, since, for a word of length 5, this is almost 12 million elements. Instead, to solve this problem, we found it easiest to treat as a game in which we must move through the space of random variables whilst maintaining the highest score (probability) possible at any given time. The method we have implemented to do this uses a priority queue to maintain a list of 'best moves'. We will first give the general form of our method, followed by an example to hopefully make things clear. Suppose that we order the marginal distributions in descending order, so that we obtain the matrix $P$, where the element $p_{ij}$ corresponds to the $i^{th}$ highest probability for the random variable corresponding to the $j^{th}$ character position in the string.

$$P = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1k} \\ p_{21} & p_{22} & \cdots & p_{2k} \\ \vdots & \vdots & \vdots & \vdots \\ p_{r1} & p_{r2} & \cdots & p_{rk} \end{bmatrix}$$

$$\text{where } p_{ij} \geq p_{xj} \text{ for } i \geq x$$

For a given realisation of the string, we can express it as a vector of positions $(a_1, a_2, ..., a_r)$ in the columns of this matrix. The value of this string in the joint probability distribution is then given as the product $p_{a_1 1} \times p_{a_2 2} \times ... \times p_{a_k k}$. With this scheme, the most probable string would correspond to

$(1, 1, ..., 1)$ (indexing from 1, in order to agree with the matrix notation).

Now, we can also define the matrix $M$, where for any choice of $a_1, ..., a_r$ with $a_j = i$, $m_{ij}$ is such that

$$P((a_1, a_2, ..., a_{(j-1)}, i{+}1, a_{(j+1)}, ..., a_r)) = m_{ij} \times P((a_1, a_2, ..., a_{(j-1)}, i, a_{(j+1)}, ..., a_r))$$

i.e. $m_{ij}$ $1 \leq i \leq (r-1), 1 \leq j \leq k$, tells us what happens if we modify the string so that the $j^{th}$ character moves from position $i$ to $i+1$, i.e. if we move to the next most likely realisation of the $j^{th}$ character's random variable. The matrix $M$ can be expressed as...

$$M = \begin{bmatrix} \frac{p_{21}}{p_{11}} & \frac{p_{22}}{p_{12}} & \cdots & \frac{p_{2k}}{p_{1k}} \\ \frac{p_{31}}{p_{21}} & \frac{p_{32}}{p_{22}} & \cdots & \frac{p_{3k}}{p_{2k}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{p_{r1}}{p_{(r-1)1}} & \frac{p_{r2}}{p_{(r-1)2}} & \cdots & \frac{p_{rk}}{p_{(r-1)k}} \end{bmatrix}$$

Now, we can express an algorithm for iteratively computing the $n$ most likely realisations of a `FuzzyString`:

1. Define a list `nMostLikely` where we will store (the column positions corresponding to) our strings, and a list `queue` where we will build a queue of best possible string modifications, ordered by the probabilities of the resulting string.

2. add the column positions corresponding to the most likely string to the list of `nMostLikely`. i.e. add $(1, 1, ..., 1)$ to `nMostLikely`. Calculate $p_0 = p_{11} \times p_{12} ... \times p_{1k}$ and add each of the tuples $(0, 1, m_{11} * p_0), (0, 2, m_{12} * p_0), ..., (0, k, m_{1k} * p_0)$ to `queue`.

3. Obtain and remove the first tuple in `queue`, `(stringPos, charPos, p)`.

4. Get the string in position `stringPos` of the list `nMostLikely`, i.e. let $(a_1, a_2, ..., a_k)$ `= nMostLikely[stringPos]`.

5. Increment the `charPos` character (indexed from 1) of this string, so that we have the string `new` $= (a_1, a_2, ..., a_{\texttt{charPos - 1}}, a_{\texttt{charPos}}{+}1, a_{\texttt{charPos + 1}}, ..., a_k)$.

This is the vector of column positions corresponding to the next most likely realisation of the string. (which has probability p).

6. Add `new` to `nMostLikely`.

7. (For clarity) rename the column positions of `new` as $(u_1, u_2, ...u_k)$.

8. Add any of the following tuples that are not already present in `queue` to `queue`: $($`nMostLikely.size`$-1, 1, m_{u_1\,1}) \times p)$, $($`nMostLikely.size`$-1, 2, m_{u_2\,2}) \times p)$, ..., $($`nMostLikely.size`$-1, k, m_{u_k\,k}) \times p)$.

9. If `nMostLikely.size` is $n$ then terminate, otherwise go to 3.

We implemented this algorithm in `FuzzyString`, and called it `morphMatrixGetNMostLikely`, because when deriving the algorithm we referred to the matrix $M$ as the "morph matrix". Although the algorithm probably looks quite complicated, all it is doing is keeping track of the next best possible moves and making the one that results in the word with the next highest probability each time before adding the moves that could follow on from this move to the queue of moves, and repeating. We will compute a simple example below:

Suppose we have the `FuzzyString` presented in Figure 46.

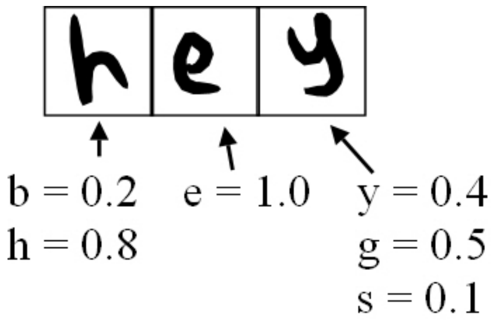


Figure 46: An example of a `FuzzyString`'s values, (0 values have been omitted).

Suppose we wish to find the first 3 most likely realisations of the `FuzzyString` given in Figure 46. The computations are as follows:

First, we compute $P$ and $M$.

$$P = \begin{bmatrix} 0.8 & 1.0 & 0.5 \\ 0.2 & * & 0.4 \\ * & * & 0.1 \end{bmatrix}$$

$$M = \begin{bmatrix} 0.25 & * & 0.8 \\ * & * & 0.25 \\ * & * & * \end{bmatrix}$$

Now, applying the algorithm:

```
             nMostLikely = [(1, 1, 1)]
                      p0 = 0.4
                   queue = [(0, 3, 0.32), (0, 1, 0.1)]

(stringPos, charPos, p) = (0, 3, 0.32)
            (a1, a2, a3) = (1, 1, 1)
                     new = (1, 1, 2)
             nMostLikely = [(1, 1, 1), (1, 1, 2)]
                   queue = [(0, 1, 0.1), (1, 1, 0.08), (1, 3, 0.08)]

(stringPos, charPos, p) = (0, 1, 0.1)
            (a1, a2, a3) = (1, 1, 1)
                     new = (2, 1, 1)
             nMostLikely = [(1, 1, 1), (1, 1, 2), (2, 1 , 1)]
```

And we do not actually need to calculate any further, as `nMostLikely` now has 3 elements. Remembering that the column positions correspond to the probabilities in the matrix $P$, it can be useful to construct a matrix of characters $S$ that matches the positions of the probabilities in $P$ with the correct character, just so that we can read off what these strings are.

$$P = \begin{bmatrix} 0.8 & 1.0 & 0.5 \\ 0.2 & * & 0.4 \\ * & * & 0.1 \end{bmatrix}$$

$$S = \begin{bmatrix} \mathtt{h} & \mathtt{e} & \mathtt{g} \\ \mathtt{b} & * & \mathtt{y} \\ * & * & \mathtt{s} \end{bmatrix}$$

We can read off from $S$ that `(1, 1, 1) = heg`, `(1, 1, 2) = hey`, `(2, 1, 1) = beg`. We can also verify that this order is correct, since we have that:

$$\begin{aligned} P(\mathtt{heg}) &= 0.4 \\ P(\mathtt{hey}) &= 0.32 \\ P(\mathtt{beg}) &= 0.1 \\ P(\mathtt{bey}) &= 0.08 \\ P(\mathtt{hes}) &= 0.08 \\ P(\mathtt{bes}) &= 0.02 \end{aligned}$$

Notice that the value of `p` in the algorithm is actually the joint probability that the `FuzzyString` will be realised as the word corresponding to `new` when it is assigned to in step 5 of the algorithm.

This method of getting the $n$ most likely strings is the fundamental algorithm in our system. Without it, a hypothesis-based approach would not be possible. We found that suitable values for $n$ lay in the range of 14 to 40; any more than this seemed to have negligible benefit. We also found that a suitable distance for the Levenshtein metric, applied in the next step was 30% of the word size.

## 4.6 Scoring

Once we have obtained our $n$ hypothesis strings, we use a `DictionaryAnalyser` to either find the first real world (according to our dictionary) in this list, or, if no real words are present - we find all words within suitable Levenshtein distances from each of these strings, and add them all to a list of possibilities. A scoring system then needs to be defined over the list of possible strings. We implemented three different methods and observed their performance. Their performance was tested on three networks, one of which was trained on the older dataset that did not contain ‘i’ and ‘j’, whilst the other two were trained on the newer dataset. More information can be found at 8.1. The first two scoring schemes were implemented as methods in `AdvancedFuzzyDictionary`, the third is a method in `FuzzyString` (since the third requires explicit use of the probability distribution within `FuzzyString`).

### By Source Position

If we assign a word a score based on the first hypothesis string that gave rise to it, i.e. the first hypothesis string that was within a suitable Levenshtein distance we gain an average of 12% in word accuracy over the discrete method - in some cases, however, we actually lose character accuracy as this method is essentially exactly the same as using a pure Levenshtein distance approach, its only benefit over this being that, if there are no words within a suitable distance around the first hypothesis string, it can consider the second and third and so on.

### By Counting Occurrences

In this method of scoring, we record the number of times a given word is added to the list of possibilities. This corresponds to the number of hypothesis words that lie in a suitable Levenshtein distance from the word. This is a better method of scoring than the previous one, obtaining an average increase of 25% in word accuracy.

### By Additive Probability Score

This method was our best approach, and as such is the scoring system that we use in our resulting system. Essentially, given a possible word `s` and the `FuzzyString` `fs` for which `s` is a proposed realisation. We use the following algorithm to calculate a score:

```
score ← fuzzyString.getPScore(possibilities[0]
string ← possibilities[0]
for possString in possibilities do
   if   fuzzyString.getPScore(possString) >
score then
      score ← fuzzyString.getPScore(possString)
      string ← possString
   end if
end for
return string
```

Here, `getPScore` is implemented in `FuzzyString` as the sum of the probabilities that each letter is realised to whatever the hypothesized string suggests - of course, we need to take into account the cases where the hypothesis word is of different length to the `FuzzyString`, in which case we slide the two words over each other until we find the best matching (maximal score), before penalising the difference in length by multiplying the score by $\frac{shortWord}{longWord}$. The reason why we implemented this score as a sum, rather than the perhaps more natural approach of polling the `FuzzyString` for the hypothesised string's actual 'probability', is that the underlying classifiers are not perfect likeness functions, and will sometimes assign a probability of 0 to the desired `ImageClass` of a character image. If we were to use the ordinary multiplicative probability measure, then we would find the correct word would then have a score of 0 in this situation. The additive approach simply makes more sense from an information theoretic point of view, i.e. as a measure of the hypothesis string agreeing with the information presented in the distribution of the `FuzzyString`. This method obtains an average increase in word accuracy of 35% over discrete classifier versions of the same CNNs, which is substantial.

Recall that in section 4.4.3, we discussed an algorithm that hypothesised multiple word segmentations for a given input image, and assigned a score to them, taking the best scoring output to be the output with the correct word segmentation (solving both problems together). This requires us to be able to score the entire output of a classifier for a given hypothesis of the word segmentation. This additive probability score method lends itself to this problem nicely, as it is simple to extend to the score of an output. The score of an output can simply be taken to be the score of each of the words the system has chosen to realise for that output, i.e. to get the score of an output we simply take the sum of `getPScore` on each of the output words.

# 5 Evaluation

## 5.1 Numerical Evaluation

As we discussed previously (4.1) as a prelude to talking about our experiments, we created our own small dataset of testing examples intended to give us grounds for comparison of our implemented methods. We had 40 examples, 20 of which were noisy (taken on a camera phone) and 20 of which were non-noisy (written digitally). The significant numerical results we have obtained during the course of this project can be found in 8. Numerically, we found that the average improvement in word accuracy given by the hypothesis-based method with scoring by source position was 12%. With the hypotheses being scored by occurrence count, this moved up to a 25% average improvement. For our best method, scoring by additive probability score, we obtained an average improvement in word accuracy over an otherwise naive classifier of 35%!

## 5.2 Achievements and Failures

Recall that in section 1.3 we stated two primary aims: we would attempt to build a system capable of reading disconnected hand-written characters, by focusing on increasing the accuracy of a classifier, and we would, in constructing such a system, implement and analyse various methods for building such a recogniser along the way.

We were indeed, for the most part, successful in overcoming and adapting to the numerous problems on the road to building a recognition system, the best accuracy we achieved where we were guaranteed to have solved the segmentation problem was 71% - this is competitive with most modern day HMM approaches, which score between 50% and 74% according to [5]. It is important to keep in mind however that HMM approaches work on connected writing, but their accuracy does not increase on disconnected writing - so the result is still competitive for the given problem domain.

Our true accuracy, when taking the entire pipeline into account, dropped to 68%, which is still respectable within the field - but certainly not practical. A user would spend more time correcting the system's output than they likely would if they were to just type up the written words themselves. 68%

of words correct on raw images is a good result, but on the flip-side, that means that 32% of the words that the system encounters, it in fact fails to recognise, possibly actually reducing the character accuracy in the process as it hypothesises and selects incorrect words. Regardless, the focus should not be on the 68% average word accuracy - our key achievement is demonstrating that the hypothesis-based approach is powerful enough to obtain a 35% improvement on its underlying classifier. This suggests that there is reason to study this approach further, possibly with regard to alternative applications such as recognising mathematics, since mathematics is typically explicitly whitespace separated (see 6.2).

We have found, in particular, that the hypothesis-based approach is far superior to the naive implementation of Levenshtein distance that many recognition systems attempt to use, for an example of this - take the following image.

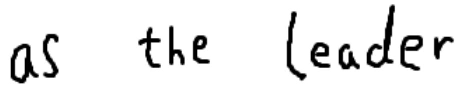


Figure 47

Our recognition system initially assigns the characters 'ceader' to the word on the far right. Of course, 'ceader' is not a word, and there are several words within Levenshtein distance of 1 to 'ceader', namely 'header', 'leader', 'reader'. A system that simply computes the Levenshtein distance has no way of knowing which of these is the best choice, and may select 'header', simply because it comes first in alphabetical order. Our system knows that what it initially thought to be a 'c', also looks like an 'l', and thus the word 'leader' has a relatively high score, and will promptly be selected.

Another interesting behaviour of the system appears to be that it tends to either be very accurate or make a significant mistake in classification or segmentation and suffer heavily for it. There are numerous examples where the system achieves 100% word accuracy, but also examples where it scores less than 30%. This behaviour is explained by the non-linear relationship that the system has with the information it is handling. Essentially, certainty in any given hypothesised words helps the system to be more certain of its selected segmentation, and thus may form stronger hypotheses for the words around it.

We can see from the results of the classification tables in 8 that the better the classifier is initially, the more capable our approach is of correcting its output. This implies that if we were to train a classifier that had a true real-world accuracy of, say, 90%+, the hypothesis-based approach would likely be able to almost totally correct the classifier's output. Unfortunately, training such a network is more difficult than we had initially thought, and it would require a significant amount of time, and possibly a more complex architecture than the one we are working with. On the other hand, we might actually expect to see some drop-off of the improvement results beyond a certain point - if one day we are able to train classifiers that are incorrect one in a million times, this approach would be rendered void.

Certainly one of our biggest failures during the course of the project was not taking more time to learn the nuances of training CNNs - we ended up training a lot of networks that ended up being discarded completely simply because we were operating via trial and error rather than initially trying to understand their intricacies. Now that we have become familiar with CNNs, in continuing this project, our first aim would be to focus and spend a significant period of time on training a good classifier so that we can truly demonstrate the power of this method with regard to a more usable application.

In undertaking the task of building this system, we certainly underestimated the amount of overhead that would be required to get the system to a point where we would be able to begin investigating our proposed approach. Whilst this slowed down our progress to our ultimate goal, it did make for a good analysis of sensible methods, measured against typical industry-standards, for each stage of the system. As such, our aim to document and analyse the methods we used in the process of constructing this system was certainly achieved, as can be seen throughout 4.

In terms of the algorithms we have derived: we have developed a simple method that parses lines by traversing them in a fairly natural manner, and this method is less constrained than the standard horizontal histogram analysis algorithm. Undoubtedly the most important algorithm we developed in order to implement this approach was the `morphMatrix` algorithm used to be able to realise the $n$ most likely strings of a `FuzzyString`. It was necessary to derive this algorithm ourselves, since it appeared there was no well-known solution for efficiently calculating the highest probability elements in a distribution (or at least, we were unable to find one). We have also recognised an issue with segmentation and recognition that few publications seem to discuss - namely that they are inherently connected and seem to be concurrent processes. Ideally, we would have liked to spend more time thinking about this problem and coming with a more satisfactory solution than the ones we have proposed, but this is a fundamental problem [19] that is perhaps beyond the scope of what we had set out to achieve.

# 6 Conclusive Remarks

Our primary conclusion is that the hypothesis-based approach was indeed an effective method for improving recognition capabilities: The approach was able to improve the word classification accuracy of an otherwise naive classifier by up to 35%. This suggests that the approach should be further investigated, and possibly combined with existing techniques, where appropriate. A secondary conclusion that has become clear after undertaking this project is that the state of the art in offline handwriting recognition has a long way to go before it will truly be capable of reading arbitrary writing from an image - the fundamental problem of the connection between recognition and segmentation is one that needs to be addressed if the field of document recognition as a whole is to move forward. HMMs are a good approach to *avoiding* the segmentation problem for the subdomain of offline handwriting recognition, but even then, they are not able to achieve useful levels of accuracy. (Hybrid methods [6] try to use the best of both approaches, but in some sense the two methods of classification are difficult to correlate).

## 6.1 Summary

The work that we have completed can be summarised as follows:

- We devised and implemented a component-wise system for recognising images of writing, based on the recognition pipeline.
- We tried and tested a number of algorithms at each stage of the system, and selected the best ones to use in our final implementation.
- We trained a number of CNN classifiers to act as likeness functions, upon which we could build our hypothesis-based approach.
- We designed and implemented an algorithm for efficiently selecting hypothesis words.
- We built a GUI that allowed the system to be transparent in its operations - allowing us to identify root issue with certain standard algorithms.
- We provided qualitative and quantitative data for and against the various methods implemented throughout the system.

- We ascertained the effectiveness of the hypothesis-based approach.

We have proposed and shown the power of an alternative method to the state of the art (HMMs) for developing document recognition systems. However, such an approach (as well as all current approaches to the problem) are still some way off of being turned into an application.

## 6.2 Future Work

In order to develop this approach so that it is a serious contender for solving problems in the field of document recognition, there are several areas that require significant developments.

### Improving The Underlying Classifier

The most pressing matter for improving the accuracy of a recognition system such as ours would be to build it upon a more powerful classifier. LeNet-5 is simple, and was effective for our needs in demonstrating the power of this approach, but to make the system one day usable in a commercial setting, it would likely require a more modern and powerful classifier such as AlexNet or ConvNet [18]. Our system in particular would have no trouble swapping in one of these more powerful networks to do its classification step, because of its pipeline-oriented design.

### Extending to Connected Characters

Work has/is already being done on the difficult problem of semantic segmentation, some of which we encountered in 4.4.1. This problem becomes even more difficult when we consider the prospect of at some point having to internally segment connected regions to find any individual characters that have been grouped together. There already exist certain recognition algorithms that can account for massive levels of noise, and say, pick out an image of a font-printed letter against a natural background. Ideally, we would like to be able to do this for handwritten text, i.e. pick out an 'a' from a string of otherwise connected characters. One particular approach to handling the connected case is proposed in [11] (p. 19). This approach particularly lends itself to being complemented by our hypothesis-based approach. We can extend our algorithm presented in 4.5.6 to being able to not only find the best hypothesis word for a given `FuzzyString`, but in fact to efficiently select the most likely word given multiple possible segmentations of a word's image. This method for connected character segmentation begins by performing heuristic over-segmentation on the isolated word image. This involves selecting a heuristic to generate segments that is slightly too weak - that is we choose a heuristic that is likely to generate a set of segmentations within which the correct set of segmentations is contained. This is demonstrated in Figure 48.

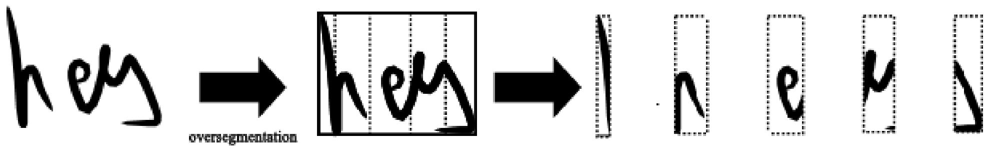


Figure 48: A possible over-segmentation of a connected word.

Possible segmentations can then be described by choosing the edges in a directed acyclic graph describing the result of this over-segmentation as seen in Figure 49.

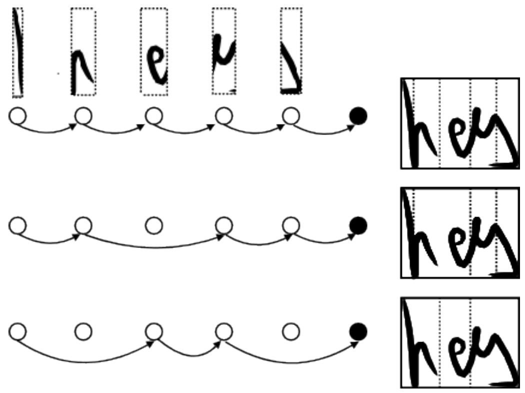


Figure 49: Each node corresponds to one of the vertical lines in the segmentation - skipping a node corresponds to skipping the line as part of the segmentation.

Now, for each possible segmentation, we can generate the `M, P` and `S` matrices to be used in our morph matrix algorithm. Two such segmentations, along with their `M, P` and `S` matrices are displayed in Figure 50.

$$S = \begin{bmatrix} h & e & g \\ b & \times & y \\ \times & \times & \times \end{bmatrix} \qquad S = \begin{bmatrix} h & a & j \\ b & \times & l \\ \times & \times & r \end{bmatrix}$$

$$P = \begin{bmatrix} 0.9 & 1.0 & 0.6 \\ 0.1 & \times & 0.4 \\ \times & \times & \times \end{bmatrix} \qquad P = \begin{bmatrix} 0.9 & 1.0 & 0.5 \\ 0.1 & \times & 0.3 \\ \times & \times & 0.2 \end{bmatrix}$$

$$M = \begin{bmatrix} 0.111.. & \times & 0.666.. \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \qquad M = \begin{bmatrix} 0.111.. & \times & 0.6 \\ \times & \times & 0.666.. \\ \times & \times & \times \end{bmatrix}$$

Figure 50

The morph matrix algorithm presented in 4.5.6 can now be modified slightly in order to select the most likely word taking into account multiple possible segmentations. In our original algorithm, we had a `queue` variable that kept a queue of moves (ordered by the probability of the resultant string) - polling this queue and making the returned move would result in an efficient way to get the next most likely string, and then the new moves that could be made from the newly generated string were added to the queue. To achieve this, elements of `queue` had the form `(stringPos, charPos, p)` where `stringPos` referred to the string that the move was being made from, `charPos` referred to the position in this string whose character was going to be incremented according to `P` and `S` matrices, and `p` was the probability of the resulting string according to the fuzzy classifier's output distribution. Hence, the only required modification to handle multiple segmentations is that we now record the tuple `(segmentation, stringPos, charPos, p)` as our moves in `queue` and we have an `nMostLikely` list for each segmentation, which is initialised with the most likely string for each possible segmentation. The algorithm can then operate exactly as before - with the `segmentation` variable being used to tell us which `M`, `P` and `S` matrices to refer to in the context of the algorithm.

This approach then allows us to select the best word from multiple possible segmentations - effectively solving the connected case. In order to properly compare a CNN recogniser that utilises the approach we have described in this document with other modern-day approaches such as HMMs, it would be useful to implement this and test on one of the competition databases provided by ICDAR, or the IAM database [20].

### Extending to Include Punctuation

During the course of our project, we did in fact intend to allow the classifier to recognise punctuation, so that writers could write more natural sentences, with full stops and commas and colons, etc. such that, again, perhaps the system could be usable. In practice, there does not appear to exist a dataset of punctuation. The handwriting recognition systems we have come across (including HMMs) do not attempt to recognise punctuation. However, if combined with a hypothesis-based approach, punctuation would give a whole new dimension of information to the recogniser. The problem is that, currently, and especially with an older architecture like LeNet-5, punctuation would be difficult to recognise in the same network as an alphabet, due to its smaller noise-like size. Smaller punctuation marks also tend to lack strong features when compared to most alphabetical characters. Extending to punctuation within a hypothesis-based method would also cause problems with the dictionary referencing procedure - the system would need to clearly define which pieces of punctuation are allowed where. This is fairly easy to do in all cases except for the apostrophe, which can occur within words (all other cases can be handled simply by asserting that they are legal either at the beginning or end of words, or both). The obvious solution to the apostrophe is just to make sure that all possible contractions are also terms in the reference dictionary, but this may not be practical. It is likely that in order to allow for punctuation to be recognised, a system would have to incorporate a more sophisticated language model.

## Using a Language Model

The reason we did not give priority to attempting to use a frequency-based language model such as a n-gram in our project, despite these models being fairly easy to construct, maintain and use, effectively comes down to an argument based on a famous heuristic called Zipf's law. Zipf's law states that, given a large enough sample of words sorted by frequency, the $n^{th}$ word in the list has a frequency proportional to $\frac{1}{n}$. This means that the first word is about twice as likely to occur as the second word, and three times as likely as the first word, etc. What this means for frequency-based-language models is that almost half of the time, we should be expecting one of only about 200 words. This means that they will always be prone to making mistakes whenever anyone decides to say something unexpected - real world examples of this include autocorrect on a mobile phone, which are typically implemented using an n-gram model. It will frequently 'correct' the user without their permission, in some cases leading to an increase in the time it takes for the user to communicate. A more sophisticated natural language model would have an idea of the semantics of what has been written, but getting algorithms to 'understand' the semantics of almost anything is a hard problem. Nevertheless, it is likely a necessary step in the progression towards total document recognition, whether via a hypothesis-based approach, or not.

## Wider Applications to Document Recognition

Throughout we have, on occasion, mentioned the potential application of this approach to recognising mathematics. Going even further, if we have an item to be recognised, and a structural distribution over the classes that the item could belong to, then we will be able to build a hypothesis-based approach to recognising that item. Of course, this is only useful if the search space is large enough that we cannot simply exhaust it - but this is very much the case in a number of written items, biological diagrams, chemical equations - perhaps even arbitrary blue-prints - all these things would be amenable to a hypothesis-based attack. Such an attack could take the form of using the morph matrix algorithm that we have described here (in 4.5.6) - this algorithm easily generalises to be usable on written glyphs that have an alphabet of symbols and some well-defined structure.

# 7 Bibliography

## References

[1] Hiromichi Fujisawa, "Forty years of research in character and document recognition—an industrial perspective," Pattern Recognition, vol. 41, no. 1, March, 2008.

[2] https://www.canadapost.ca/cpotools/apps/fpc/personal/findAnAddress?execution=e1s1. Accessed: 2017-05-14.

[3] Sébastien Eskenazi, Petra Gomez-Krämer, Jean-Marc Ogier, "A comprehensive survey of mostly textual document segmentation algorithms since 2008," Pattern Recognition, vol. 64, no. 1, April 2017.

[4] Fei Yin, Qiu-Feng Wang, Xu-Yao Zhang, Cheng-Lin Liu. "ICDAR 2013 Chinese Handwriting Recognition Competition," *International Conf. on Document Analysis and Recognition*, Washington DC, 2013, p.5

[5] Thomas Plötz, Gernot A. Fink. "Markov models for offline handwriting recognition: a survey," International Journal on Document Analysis and Recognition, vol. 12, no. 4, December 2009.

[6] Alex Graves, Jürgen Schmidhuber. "Offline Handwriting Recognition with Multidimensional Recurrent Neural Networks," *Advances in Neural Information Processing Systems 21*, Curran Associates, Inc. 2009, pp.545-552.

[7] Veronica Romero, Joan Andreu Sánchez, Nicolás Serrano, Enrique Vidal. "Handwritten Text Recognition for Marriage Register Books," *International Conf. on Document Analysis and Recognition*, Beijing, China, September, 2011.

[8] A. Antonacopoulos, C. Clausner, C. Papadopoulos, S. Pletschacher."ICDAR2013 Competition on Historical Book Recognition", *International Conf. on Document Analysis and Recognition*, Washington DC, 2013, p.1459.

[9] Derek Bradley, Gerhard Roth. "Adaptive Thresholding Using the Integral Image," *Journal of Graphics Tools*, vol. 12, no. 2, pp. 13-21, 2007.

[10] Franck Mamalet, Christophe Garcia. "Simplifying ConvNets for Fast Learning," *22nd International Conference on Artificial Neural Networks*, Lausanne, Switzerland, September, 2012, Proceedings, Part 2 pp.64.

[11] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner. "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no.11, November, 1998.

[12] Richard G. Casey, Eric Lecolinet. "A Survey of Methods and Strategies in Character Segmentation," *IEEE Transactions on Patter Analysis and Machine Intelligence*, vol. 18, no. 7, July, 1996.

[13] `https://www.codeproject.com/Articles/13525/Fast-memory-efficient-Levenshtein-algorithm`. Accessed: 2017-06-20.

[14] `https://ibug.doc.ic.ac.uk/`. Accessed: 2017-06-20.

[15] `http://www.heatonresearch.com/encog/`. Accessed: 2017-04-14.

[16] `https://deeplearning4j.org/`. Accessed: 2017-06-04.

[17] `https://www.nist.gov/srd/nist-special-database-19` Accessed: 2017-05-14.

[18] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks," *Advances in neural informations processing systems*, 2012.

[19] Jonathan Long, Evan Shelhamer, Trevor Darrell. "Fully convolutional networks for semantic segmentation," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[20] U.V. Marti, H. Bunke. "The IAM-database: an English sentence database for offline handwriting recognition", *International Journal on Document Analysis and Recognition*, vol. 5, no. 1, November, 2002, pp. 39–46.

[21] George Cybenko."Approximations by superpositions of sigmoidal functions", *Mathematics of Control, Signals, and Systems*, vol. 2, no. 4, December, 1989, pp. 303-314.

# 8 Appendix

## 8.1 Result Tables

The tables below describe our results on our tested methods. Recall that we have 3 versions of the testing data to test on: raw, denoised and segmentation-corrected. In the tables below, denoising preprocesses are tested on the 20 noisy images in the raw set. Line and word segmentation are tested on the denoised set, and classification accuracies are computed on segmentation-corrected for two of the hypothesis-scoring methods, and for both the segmentation-corrected and raw datasets for our best hypothesis-scoring method. These accuracies are calculated for three neural networks whose confusion matrices are given in the appendix. “Reported network accuracy” is the accuracy reported at the end of the training phase, while “real network accuracy” is the accuracy the network was found to have on the set of testing examples. Naturally, the real accuracy is less than reported, since the examples are not as clean as the datasets used for training. Note that `net3` is a network which is not trained to recognise ‘i’ and ‘j’ - as such, its reported accuracy refers to its accuracy at the end of the training procedure, i.e. not including ‘i’ and ‘j’. Word accuracy refers to the number of whole words correctly identified by the recogniser. Note that all percentage results have been rounded to the nearest integer.

| Denoising Preprocesses | | | |
|---|---|---|---|
| Preprocess Name | Success | Failure | TOTAL |
| `ContrastifyByAverage` | 8 | 12 | 20 |
| `ContrastifyByAdptvThrsh` | 20 | 0 | 20 |

| Line Segmentation Algorithms | | | |
|---|---|---|---|
| Algorithm | Success | Failure | TOTAL |
| By horizontal histogram | 29 | 11 | 40 |
| `FrontRunnerLineParser` | 36 | 4 | 40 |

| Word Segmentation Algorithms | | | |
|---|---|---|---|
| Algorithm | Success | Failure | TOTAL |
| `WordParserByLineAverage` | 11 | 29 | 40 |
| `WordParserViaSmudge` | 21 | 19 | 40 |
| `WordParserViaSmudge (interval)` | 35 | 5 | 40 |

| Discrete Accuracy (seg-corrected) | | | |
|---|---|---|---|
| Network | Reported Char. Acc. (%) | Real Char Acc. (%) | Word Acc. (%) |
| net1 | 91 | 63 | 19 |
| net2 | 84 | 71 | 25 |
| net3 | 95 | 72 | 26 |

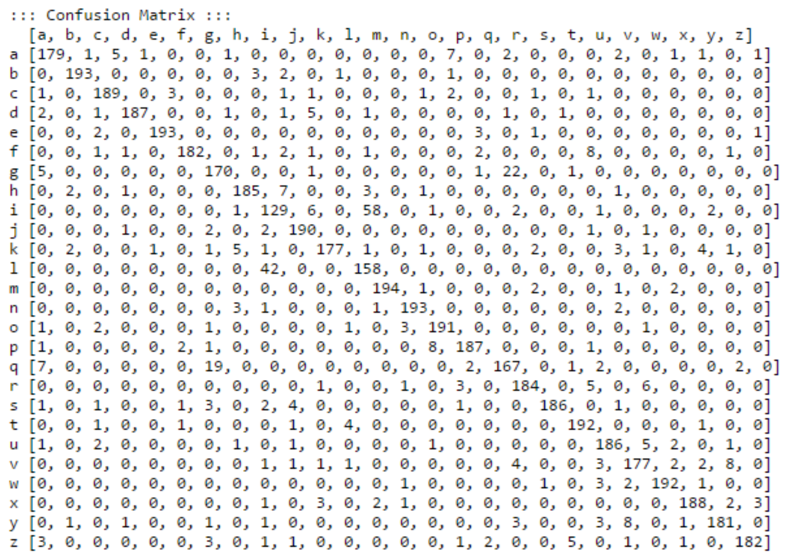
| Hyp-based Approach Accuracy (By Source Position)(seg-corrected) | | |
|---|---|---|
| Network | Char Accuracy (%) | Word Accuracy (%) |
| net1 | 67 | 31 |
| net2 | 76 | 38 |
| net3 | 77 | 36 |

| Hyp-based Approach Accuracy (By Occurrence Count)(seg-corrected) | | |
|---|---|---|
| Network | Char Accuracy (%) | Word Accuracy (%) |
| net1 | 69 | 39 |
| net2 | 81 | 54 |
| net3 | 78 | 51 |

| Hyp-based Approach Accuracy (By Additive P.Score)(seg-corrected) | | |
|---|---|---|
| Network | Char Accuracy (%) | Word Accuracy (%) |
| net1 | 68 | 41 |
| net2 | 84 | 71 |
| net3 | 81 | 64 |

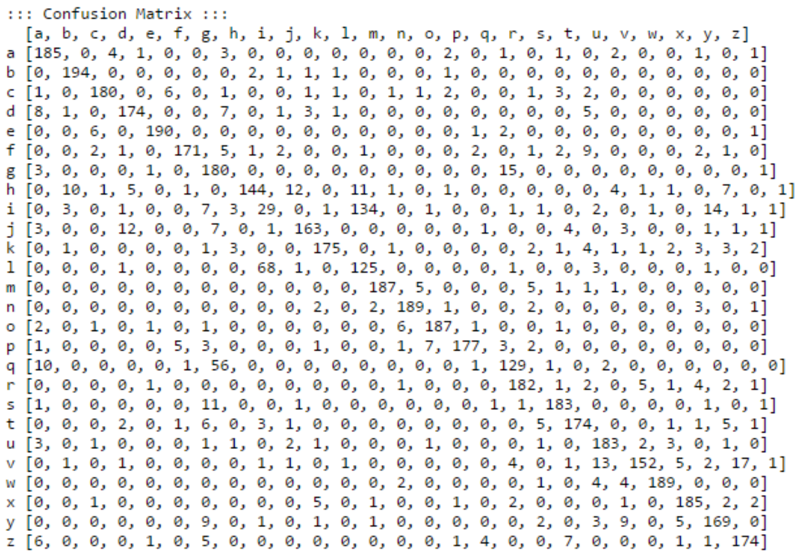
| Hyp-based Approach Accuracy (By Additive P.Score)(raw) | | |
|---|---|---|
| Network | Char Accuracy (%) | Word Accuracy (%) |
| net1 | 61 | 39 |
| net2 | 81 | 68 |
| net3 | 79 | 60 |

## 8.2 Confusion Matrices for Test Networks



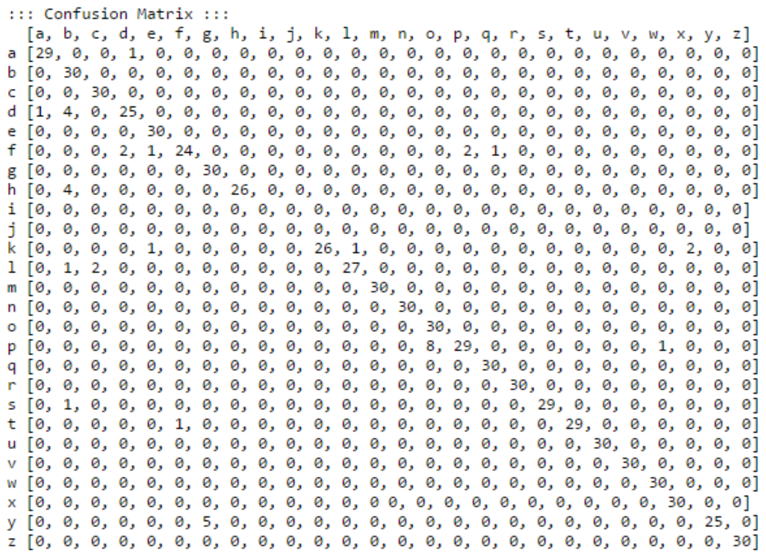
```
::: Confusion Matrix :::
  [a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z]
a [179, 1, 5, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 7, 0, 2, 0, 0, 0, 2, 0, 1, 1, 0, 1]
b [0, 193, 0, 0, 0, 0, 0, 3, 2, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
c [1, 0, 189, 0, 3, 0, 0, 0, 1, 1, 0, 0, 0, 1, 2, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0]
d [2, 0, 1, 187, 0, 0, 1, 0, 1, 5, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0]
e [0, 0, 2, 0, 193, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1]
f [0, 0, 1, 1, 0, 182, 0, 1, 2, 1, 0, 1, 0, 0, 0, 2, 0, 0, 0, 8, 0, 0, 0, 0, 1, 0]
g [5, 0, 0, 0, 0, 0, 170, 0, 0, 1, 0, 0, 0, 0, 0, 1, 22, 0, 1, 0, 0, 0, 0, 0, 0, 0]
h [0, 2, 0, 1, 0, 0, 0, 185, 7, 0, 0, 3, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
i [0, 0, 0, 0, 0, 0, 0, 1, 129, 6, 0, 58, 0, 1, 0, 0, 2, 0, 0, 1, 0, 0, 0, 2, 0, 0]
j [0, 0, 0, 1, 0, 0, 2, 0, 2, 190, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0]
k [0, 2, 0, 0, 1, 0, 1, 5, 1, 0, 177, 1, 0, 1, 0, 0, 0, 2, 0, 0, 3, 1, 0, 4, 1, 0]
l [0, 0, 0, 0, 0, 0, 0, 0, 42, 0, 0, 158, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
m [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 194, 1, 0, 0, 0, 2, 0, 0, 1, 0, 2, 0, 0, 0]
n [0, 0, 0, 0, 0, 0, 0, 3, 1, 0, 0, 0, 1, 193, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0]
o [1, 0, 2, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 3, 191, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
p [1, 0, 0, 0, 0, 2, 1, 0, 0, 0, 0, 0, 0, 0, 8, 187, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
q [7, 0, 0, 0, 0, 0, 19, 0, 0, 0, 0, 0, 0, 0, 0, 2, 167, 0, 1, 2, 0, 0, 0, 0, 2, 0]
r [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 3, 0, 184, 0, 5, 0, 6, 0, 0, 0, 0]
s [1, 0, 1, 0, 0, 1, 3, 0, 2, 4, 0, 0, 0, 0, 0, 1, 0, 0, 186, 0, 1, 0, 0, 0, 0, 0]
t [0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 4, 0, 0, 0, 0, 0, 0, 0, 192, 0, 0, 0, 1, 0, 0]
u [1, 0, 2, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 186, 5, 2, 0, 1, 0]
v [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 4, 0, 0, 3, 177, 2, 2, 8, 0]
w [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 3, 2, 192, 1, 0, 0]
x [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 3, 0, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 188, 2, 3]
y [0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 3, 8, 0, 1, 181, 0]
z [3, 0, 0, 0, 0, 0, 3, 0, 1, 1, 0, 0, 0, 0, 0, 1, 2, 0, 0, 5, 0, 1, 0, 1, 0, 182]
```

Figure 51: The confusion matrix for `net1`.



```
::: Confusion Matrix :::
  [a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z]
a [185, 0, 4, 1, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 2, 0, 1, 0, 1, 0, 2, 0, 0, 1, 0, 1]
b [0, 194, 0, 0, 0, 0, 0, 2, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
c [1, 0, 180, 0, 6, 0, 1, 0, 0, 1, 1, 0, 1, 1, 2, 0, 0, 1, 3, 2, 0, 0, 0, 0, 0, 0]
d [8, 1, 0, 174, 0, 0, 7, 0, 1, 3, 1, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0]
e [0, 0, 6, 0, 190, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 1]
f [0, 0, 2, 1, 0, 171, 5, 1, 2, 0, 0, 1, 0, 0, 0, 2, 0, 1, 2, 9, 0, 0, 0, 2, 1, 0]
g [3, 0, 0, 0, 1, 0, 180, 0, 0, 0, 0, 0, 0, 0, 0, 0, 15, 0, 0, 0, 0, 0, 0, 0, 0, 1]
h [0, 10, 1, 5, 0, 1, 0, 144, 12, 0, 11, 1, 0, 1, 0, 0, 0, 0, 0, 4, 1, 1, 0, 7, 0, 1]
i [0, 3, 0, 1, 0, 0, 7, 3, 29, 0, 1, 134, 0, 1, 0, 0, 1, 1, 0, 2, 0, 1, 0, 14, 1, 1]
j [3, 0, 0, 12, 0, 0, 7, 0, 1, 163, 0, 0, 0, 0, 0, 1, 0, 0, 4, 0, 3, 0, 0, 1, 1, 1]
k [0, 1, 0, 0, 0, 0, 1, 3, 0, 0, 175, 0, 1, 0, 0, 0, 0, 2, 1, 4, 1, 1, 2, 3, 3, 2]
l [0, 0, 0, 1, 0, 0, 0, 0, 68, 1, 0, 125, 0, 0, 0, 0, 1, 0, 0, 3, 0, 0, 0, 1, 0, 0]
m [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 187, 5, 0, 0, 0, 5, 1, 1, 1, 0, 0, 0, 0, 0]
n [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 2, 189, 1, 0, 0, 2, 0, 0, 0, 0, 0, 3, 0, 1]
o [2, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 6, 187, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
p [1, 0, 0, 0, 0, 5, 3, 0, 0, 0, 1, 0, 0, 1, 7, 177, 3, 2, 0, 0, 0, 0, 0, 0, 0, 0]
q [10, 0, 0, 0, 0, 1, 56, 0, 0, 0, 0, 0, 0, 0, 0, 1, 129, 1, 0, 2, 0, 0, 0, 0, 0, 0]
r [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 182, 1, 2, 0, 5, 1, 4, 2, 1]
s [1, 0, 0, 0, 0, 0, 11, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 183, 0, 0, 0, 0, 1, 0, 1]
t [0, 0, 0, 2, 0, 1, 6, 0, 3, 1, 0, 0, 0, 0, 0, 0, 0, 0, 5, 174, 0, 0, 1, 1, 5, 1]
u [3, 0, 1, 0, 0, 0, 1, 1, 0, 2, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 183, 2, 3, 0, 1, 0]
v [0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 4, 0, 1, 13, 152, 5, 2, 17, 1]
w [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 1, 0, 4, 4, 189, 0, 0, 0]
x [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 5, 0, 1, 0, 0, 1, 0, 2, 0, 0, 0, 1, 0, 185, 2, 2]
y [0, 0, 0, 0, 0, 0, 9, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 2, 0, 3, 9, 0, 5, 169, 0]
z [6, 0, 0, 0, 1, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 1, 4, 0, 0, 7, 0, 0, 0, 1, 1, 174]
```

Figure 52: The confusion matrix for `net2`.



```
::: Confusion Matrix :::
  [a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z]
a [29, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
b [0, 30, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
c [0, 0, 30, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
d [1, 4, 0, 25, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
e [0, 0, 0, 0, 30, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
f [0, 0, 0, 2, 1, 24, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
g [0, 0, 0, 0, 0, 0, 30, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
h [0, 4, 0, 0, 0, 0, 0, 26, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
i [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
j [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
k [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 26, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0]
l [0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 27, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
m [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 30, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
n [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 30, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
o [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 30, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
p [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8, 29, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
q [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 30, 0, 0, 0, 0, 0, 0, 0, 0, 0]
r [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 30, 0, 0, 0, 0, 0, 0, 0, 0]
s [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 29, 0, 0, 0, 0, 0, 0, 0]
t [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 29, 0, 0, 0, 0, 0, 0]
u [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 30, 0, 0, 0, 0, 0]
v [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 30, 0, 0, 0, 0]
w [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 30, 0, 0, 0]
x [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 30, 0, 0]
y [0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 25, 0]
z [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 30]
```

Figure 53: The confusion matrix for `net3`.

## 8.3 Sketch Proof of MPS Algorithm's Properties

1. Define a list `nMostLikely` where we will store (the column positions corresponding to) our strings, and a list `queue` where we will build a queue of best possible string modifications, ordered by the probabilities of the resulting string.

2. add the column positions corresponding to the most likely string to the list of `nMostLikely`. i.e. add $(1, 1, ..., 1)$ to `nMostLikely`. Calculate $p_0 = p_{11} \times p_{12} ... \times p_{1k}$ and add each of the tuples $(0, 1, m_{11} * p_0), (0, 2, m_{12} * p_0), ..., (0, k, m_{1k} * p_0)$ to `queue`.

3. Obtain and remove the first tuple in `queue`, `(stringPos, charPos, p)`.

4. Get the string in position `stringPos` of the list `nMostLikely`, i.e. let $(a_1, a_2, ..., a_k)$ `= nMostLikely[stringPos]`.

5. Increment the `charPos` character (indexed from 1) of this string, so that we have the string `new` $= (a_1, a_2, ..., a_{\texttt{charPos - 1}}, a_{\texttt{charPos}}+1, a_{\texttt{charPos + 1}}, ..., a_k)$. This is the vector of column positions corresponding to the next most likely realisation of the string. (which has probability p).

6. Add `new` to `nMostLikely`.

7. (For clarity) rename the column positions of `new` as $(u_1, u_2, ...u_k)$.

8. Add any of the following tuples that are not already present in `queue` to `queue`: $(\texttt{nMostLikely.size} - 1, 1, m_{u_1 1}) \times p), (\texttt{nMostLikely.size} - 1, 2, m_{u_2 2}) \times p), ..., (\texttt{nMostLikely.size} - 1, k, m_{u_k k}) \times p)$.

9. If `nMostLikely.size` is $n$ then terminate, otherwise go to 3.

For the purposes of this algorithm a realisation of a given `FuzzyString` is a tuple of column positions $(i_1, i_2, ..., i_k)$ indexed from 1. This tuple tells us how to read from the columns of the $S$ matrix to find the string, and the columns of the $P$ matrix to find the probability for each individual character (thus, assuming independence and taking their product results in the probability of the string).

**Definition**
We call a sequence of such tuples an **incremental chain** if each consecutive term has the same form as the previous term, except one component has been incremented. e.g. $(1,1,1),(2,1,1),(2,2,1),(2,3,1),(2,3,2),....$

Thus, to demonstrate that the algorithm we proposed actually computes the $n$ most likely string realisations of a given `FuzzyString` it suffices to show:

1. The strings resulting from a move `(stringPos, charPos, p)` in queue have probability corresponding to $p$.
2. Consecutive terms in an incremental chain have (weakly) decreasing probability.

Then, we can simply observe that the algorithm keeps track of *every* incremental chain beginning from the most probable string $(1,1,1,...,1)$ and maintains a list of moves corresponding to each of the next terms in each chain in `queue`. The string with highest `p` from this list must then be the next most likely string, since it has highest probability in the `queue`, and every string not in `queue` is further down in at least one of the incremental chains, and hence has lower probability than some element of `queue`.
So, we prove the two claims.

1. We proceed by induction on the number $k$ of entrances into the loop 3-9.

    $k = 0$

    The string $(i_1, i_2, ..., i_k)$ has probability $P_{i_1\,1} \times P_{i_2\,2} \times ... \times P_{i_k\,k}$.

    When we are at step 3, but before we have begun to execute the loop,

    we have moves in `queue` that correspond to the strings...

    $(2, 1, ..., 1)$ with probability $p_0 * m_{11}$,

    $(1, 2, ..., 1)$ with probability $p_0 * m_{12}$,

    ...,

    $(1, 1, ..., 2)$ with probability $p_0 * m_{1k}$.

    Now, $p_0 = P_{11} \times P_{12} \times ... \times P_{1k}, m_{ij} = M_{ij} = P_{(i+1)\,j}/P_{i\,j}$

    Hence... $p_0 * m_{11} = P_{21} \times P_{12} \times ... \times P_{1k}$

    ...

    $p_0 * m1k = P_{11} \times P_{12} \times ... \times P_{2k}$.

    So these elements of queue have `p` corresponding to the probabilities of the resulting strings.

    Now assume we have the proposition at our $k^{th}$ entry into the loop

    Then every `p` in `queue` corresponds to the probability of the resulting string.

    We pick a tuple from `queue`, call it `new`, add it to `nMostLikely` and add all of the possible consecutive terms in an incremental chain beginning with `new` to `queue`.

    Suppose `new` $= (i_1, i_2, i_3, .., i_k)$, then `new` has `p` $= P_{i_1 1} \times P_{i_2 2} \times ... \times P_{i_k k}$.

    Then we add all of its increments, each with probability `p` $\times M_{i_t t}$, where $t$ is the incremented column position. Hence, the incremented strings corresponding

    `p` $= P_{i_1 1} \times P_{i_2 2} \times ... \times P_{i_k k} \times P_{(i_t+1)t}/P_{i_t t}$

    $= P_{i_1 1} \times P_{i_2 2} \times ... \times P_{(i_{(t-1)}(t-1)} \times P_{(i_t+1)t} \times P_{(i_{(t+1)}(t+1)} \times ... \times P_{i_k k}$,

    which is the corresponding probability for the incremented string.

2. Suppose we have $(i_1, i_2, ..., i_k)$ and WLOG suppose we increment the first character to obtain $(i_1 + 1, i_2, ..., i_k)$ the first string has probability $P_{i_1\,1} \times P_{i_2\,2} \times ... \times P_{i_k\,k}$ , the second string has probability $P_{(i_1+1)\,1} \times P_{i_2\,2} \times ... \times P_{i_k\,k}$ and $P_{(i_1+1)\,1} \leq P_{(i_1)\,1}$ so the proposition holds.