Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Whiteboard Captioning Using Bézier Curves & Neural Networks

Author:
Alexander Goncharov

Supervisor:
Dr. Mark J. Wheelhouse

Second Marker:
Mr. Bozhidar Stevanoski

June 19, 2023

Abstract

This study explores an approach to handwriting recognition that leverages dynamic data from stylus or fingertip movements, and introduces two Bézier curve feature sets for the transformation of handwritten stroke points. The IAM-OnDB dataset was utilised to develop and evaluate several models, with the LSTM model outperforming the others, achieving a beam search decoding Character Error Rate (CER) and Word Error Rate (WER) of 42.3% and 88.1% on the test set, respectively. An interactive web application was also developed to facilitate real-time user engagement with the models and visualise Bézier curves. Unlike some previous related studies, the entire codebase was made open-source to promote further research and innovation in this field.

Acknowledgements

Firstly, I would like to hugely thank my supervisor, Dr. Mark J. Wheelhouse. His guidance and insights have been invaluable throughout this project. I greatly appreciate all the time he invested, and the enjoyable and enlightening discussions that shaped this research.

Additionally, I would like to extend my gratitude to Mr. Bozhidar Stevanoski for the all the time he invested and the invaluable feedback he provided during the interim report milestone.

Also, I am extremely grateful to my personal tutor, Dr. Jackie Bell, for her unwavering support throughout my degree. Her encouragement and support has been a significant factor in my academic journey.

Lastly, I would like to thank my mum, brother, and friends for always encouraging me and keeping me going throughout these four years. I would also like to thank my girlfriend, Rosemary Ng, for supporting and understanding of me working constantly throughout this project, and always keeping a smile on my face.

Contents

Chapter 1

Introduction

1.1 Motivation

Handwriting is a means of communication that allows individuals to express themselves in a unique and personal way. Handwriting recognition has often been based on optical character recognition (OCR), which uses images to identify text and has been already heavily researched. As technology changes, we need to explore new methods. One promising approach is to track the movement of a pen or stylus to recognise handwriting, with several studies having shown that this approach offers a potential higher classification rate compared to using OCR [1], and deep learning methods having outperformed "traditional" pattern recognition methods [2].

More and more devices, like Apple iPads, come with a stylus, and are used in many industries. With the rise of Virtual Reality (VR) and Augmented Reality (AR) technologies, even the movements of a fingertip can be tracked in a way similar to a pen tip. This new data can be used to predict what someone is writing, opening up exciting possibilities for handwriting recognition. This research will explore some of these opportunities.

1.2 Contributions

Some of the following contributions were made utilising pen-tip movement data:

- **Overcoming 'Vanishing Gradients':** Used the ISGL dataset to address the 'vanishing gradient' problem encountered during the processing of long (x, y) stroke point sequences, highlighting the need for refined feature extraction techniques.

- **Bézier Curve Feature Sets:** Created two unique feature sets that are used to convert handwritten (x, y) stroke points into Bézier curve features, offering a new perspective for the interpretation and recognition of handwritten data.

- **Handwriting Recognition with IAM-OnDB Dataset:** Used the IAM-OnDB dataset to develop handwriting recognition models, with the LSTM model being the best performing by achieving 42.3% and 88.1% for beam search decoding Character Error Rate (CER) and Word Error Rate (WER), respectively, on the test set.

- **Interactive Web Application:** Developed a user-friendly web application, providing an interface for real-time interaction with the models, adjustment of model parameters, and visualisation of Bézier curves for each written stroke.

- **Open-Source Contribution:** The codebase for this project was developed from scratch and made open-source, unlike previous related studies, helping to drive further innovation in the field of handwriting recognition using Bézier curves.

Chapter 2

Background

2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are computational models consisting of interconnected nodes, organised into layers: an input layer, one or more hidden layers, and an output layer (e.g. see Figure 2.1).

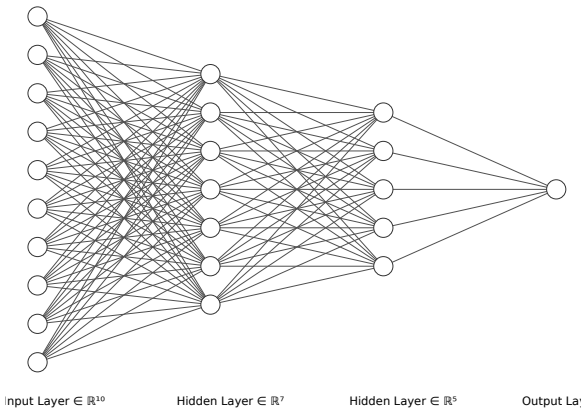


Figure 2.1: An example of a neural network, consisting of one input layer with 10 neurons, two hidden layers with 7 and 5 neurons respectively, and an output layer.

Each neuron in a layer connects to every neuron in the next layer, with each connection having an associated weight. These weights are adjusted during the learning process, allowing the network to learn from data and improve over time. The computation in each neuron can be expressed as:

$$y = f\left(\sum_i w_i x_i + b\right) \tag{2.1}$$

where x_i are the inputs, w_i are the weights, b is the bias and f is the activation function. The output y is then sent to the neurons in the next layer.

The input layer receives initial data for processing, which can be in various forms such as pixel data from images, numerical data, etc (in our case, (x, y) points of a stroke, or a Bézier curve fitted to one). The hidden layers perform computations on the inputs, applying a transformation function to the weighted sum of the inputs. The output layer produces the final output of the network, which can be used for tasks such as classification or regression. Their ability to learn from data and adapt their internal parameters makes them a powerful tool in the field of machine learning [3].

2.1.1 Cost Functions

The cost function, also known as the loss function or objective function, is a measure of the network's performance. It quantifies the difference between the predicted output and the actual output for the given input data. The goal of training an ANN is to minimise the cost function. The cost function J for a neural network can be expressed as:

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(y^{(i)}, \hat{y}^{(i)}) \tag{2.2}$$

where L is the loss function, $y^{(i)}$ is the actual output, $\hat{y}^{(i)}$ is the predicted output, m is the number of training examples, w are the weights, and b is the bias [4].

In the context of neural networks, the cost function plays a crucial role in the learning process. It provides a measure of the error made by the network in its predictions, and the goal of learning is to adjust the parameters of the network (weights and biases) to minimise this error, which is typically done by using gradient descent [5].

Moreover, the choice of cost function can significantly influence the learning process. Different cost functions are suitable for different tasks. For instance, Mean Squared Error (MSE) is often used for regression tasks, while Cross-Entropy loss is commonly used for multi-class classification tasks.

It's also worth noting that the cost function should be chosen carefully to avoid issues such as overfitting, where the model learns the training data too well and performs poorly on unseen data. Techniques such as regularisation, which adds a penalty term to the cost function to limit the complexity of the model, can be used to mitigate this issue [6].

2.1.2 Activation Functions

Activation functions, denoted as f in Equation 2.1, play a crucial role in Artificial Neural Networks (ANNs). They introduce non-linearity into the network, allowing it to learn and model complex patterns in the data. Without activation functions, ANNs would be limited to linear transformations, significantly reducing their expressive power [4].

As shown in Equation 2.1, the activation function is applied to the weighted sum of the inputs plus the bias, transforming the output of each neuron. This output is then passed on to the neurons in the next layer. Hence, the choice of activation function can significantly impact the performance of the network, and different types of activation functions are suitable for different tasks.

The sigmoid activation function, also known as the logistic function, is defined as:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

The sigmoid function maps any input value into the range between 0 and 1. This characteristic makes it particularly useful for binary classification problems, where the output can be interpreted as a probability [7]. However, the sigmoid function suffers from the 'vanishing gradient' problem, where the gradients become very small for large positive or negative inputs. This can slow down the learning process during backpropagation, as the updates to the weights become negligible [8].

The hyperbolic tangent (tanh) function is another commonly used activation function. It is defined as:

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.4}$$

The tanh function maps any input value into the range between -1 and 1. This means that the output is zero-centered, which can make the learning process more efficient [9]. Like the sigmoid function, the tanh function also suffers from the 'vanishing gradient' problem for large positive or negative inputs.

Another option is the Rectified Linear Unit (ReLU) activation function, which is defined as:

$$f(x) = \max(0, x) \tag{2.5}$$

ReLU introduces non-linearity into the network and is computationally efficient, as it allows the network to converge very quickly. It also helps to alleviate the 'vanishing gradient' problem, as it does not saturate in the positive region [10].

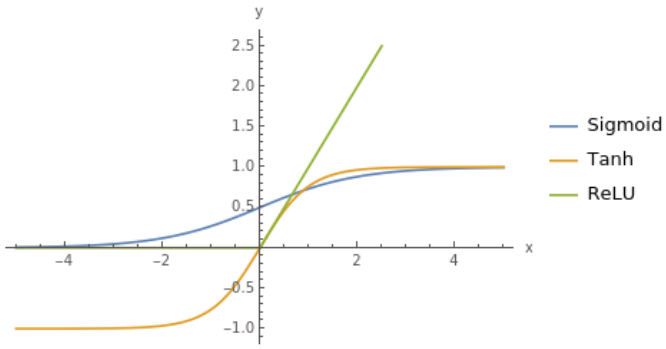


Figure 2.2: A visual comparison of Sigmoid, Tanh, and ReLU activation functions.

For a visual comparison between these different activation functions, see Figure 2.2.

Despite ReLU's advantages, it is typically not used in Recurrent Neural Networks (RNNs). This is primarily due to the nature of these networks, which are designed to handle sequential data and need to propagate information through time. The ReLU activation function can lead to 'exploding gradients', where the value of the gradient can become very large, causing the weights in the network to update in such a way that the network becomes unstable and unable to learn effectively.

Additionally, ReLU units can sometimes become inactive during training such that they no longer adjust their weights or contribute to the model, a problem known as 'dying ReLU'. This issue can be particularly problematic in RNNs, where the activation function is applied across many time steps, increasing the likelihood of a ReLU unit becoming inactive.

In contrast, activation functions like sigmoid and tanh, which have bounded outputs, are more commonly used in RNNs as they help maintain stability over time and prevent the gradients from exploding [11]. Hence, while ReLU has proven to be very effective in feed-forward neural networks, its characteristics make it less suitable for use in RNNs.

2.1.3 Gradient Descent and Backpropagation

Gradient descent and backpropagation are two fundamental algorithms used in training Artificial Neural Networks (ANNs).

Gradient descent is an optimisation algorithm used to minimise the cost function. It is an iterative method that moves in the direction of steepest descent as defined by the negative of the gradient, which is used to update the weights in an ANN and minimise the error function. The learning rate determines how big the steps are on the descent. Too high of a learning rate and the descent may overshoot the minimum, whereas setting it too low may result in it taking too long to converge [4]. Hence, it's important to set a learning rate that is a balance between both of these cases.

The update rule for the weights in gradient descent can be expressed as:

$$w = w - \eta \frac{\partial J}{\partial w} \tag{2.6}$$

where w are the weights, η is the learning rate, and $\frac{\partial J}{\partial w}$ is the gradient of the cost function J with respect to the weights [4].

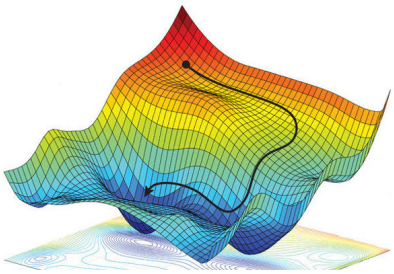


Figure 2.3: A visualisation of gradient descent being used to find a local optimum [12].

This equation represents the iterative process which is repeated until the algorithm converges to a minimum of the cost function (e.g. see Figure 2.3).

Gradient descent can be categorised into three types:

1. Batch Gradient Descent
2. Stochastic Gradient Descent (SGD)
3. Mini-batch Gradient Descent

Each type differs in how much data we use to compute the gradient of the objective function. Batch gradient descent uses the whole training data at every step whereas stochastic gradient descent uses 1 sample from the training data at every step. Mini-batch gradient descent is a compromise between the two extremes - it uses a mini-batch of n training samples at each step [5].

Batch gradient descent, while computationally intensive, has the benefit of producing a stable error gradient and a stable convergence [13]. It uses the entire dataset to compute the gradient of the loss function at each step of the training iteration. This means that each step of the training iteration considers all training examples in the dataset before making an update, leading to a thorough examination of the solution space. However, this method can be slow for large datasets and does not allow for online updates.

On the other hand, SGD uses only a single sample, i.e., a batch size of 1, at each iteration. The benefit of SGD is its speed and efficiency, making it particularly useful for large-scale datasets. SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily. While this fluctuation can help to navigate rough solution spaces and find the global minimum in non-convex tasks, it can also lead to less stable convergence with the final parameters potentially oscillating around the true values [13].

Mini-batch gradient descent strikes a balance between the efficiency of SGD and the stability of batch gradient descent. Adjusting the mini-batch size allows us to control the learning process - a smaller mini-batch size leads to a learning process closer to SGD where noise in the gradient estimation can help escape shallow local minima, while a larger mini-batch size leads to a learning process closer to batch gradient descent where the trajectory of the optimisation is smoother [13]. This is a method we will utilise and investigate within this research.

In addition to gradient descent, backpropagation is an algorithm used for training ANNs. It calculates the gradient of the loss function with respect to the weights of the network for a single input–output example efficiently, unlike a naive direct computation [14, 15].

In a naive direct computation, you would have to compute the effect of changing each weight on the loss function independently. This would involve a number of computations that is proportional to the number of weights in the network, which can be very large in a deep neural network.

However, backpropagation takes advantage of the fact that the weights in earlier layers of the network affect the loss function indirectly, through their effects on the later layers. By computing the gradient of the loss function with respect to the outputs of each layer (starting from the final layer and working backwards), and then using these to compute the gradient with respect to the weights, backpropagation reduces the amount of computation needed [15].

The combination of gradient descent and backpropagation allows ANNs to learn from the data by iteratively adjusting the weights of the connections in the network. The weights are adjusted to minimise the difference between the actual output and the desired output. This iterative process of weight adjustment (learning) continues until the network performs the task accurately enough [15].

2.1.4 Adam Optimiser

Building upon the concepts of gradient descent and backpropagation, and just to give a high-level overview, an optimiser is an algorithm or method used to adjust the attributes of the neural network such as weights and learning rate in order to reduce the losses, i.e. optimisers help to get results faster.

Within this research, the Adam optimiser is used - Adam provides an optimisation algorithm that can handle sparse gradients on noisy problems, is computationally efficient and has very little memory requirement, and is a method that computes adaptive learning rates for each parameter [5].

Adam works well in practice and compares favorably to other adaptive learning-method algorithms as it converges very fast and the learning speed of the Model is quite high and efficient. It is one of the most popular optimisation algorithms and has been used to achieve state-of-the-art results in many deep learning tasks [16].

2.1.5 Early Stopping and Dropout

To prevent overfitting, early stopping and dropout are two regularisation techniques used in training neural networks. Regularisation methods often involve modifying the network architecture or the learning process to prevent the network from relying too much on any single pattern or feature in the training data. This helps the network to generalise better to unseen data.

Overfitting occurs when a model learns the training data too well, including the noise and outliers, and performs poorly on unseen data. These techniques add a regularisation effect to the model (i.e. adding a penalty on the complexity of the model to reduce the risk of overfitting), improving its generalisation capability.

Early stopping works by stopping training when the performance on a validation dataset which the model does not learn from, stops improving, i.e., the error starts to increase.

The idea is to monitor the model's performance on a separate validation dataset during the training phase and stop training when the validation error begins to increase. This point indicates that the model is starting to memorise the training data and lose its ability to generalise to new data. By stopping the training at this point, early stopping helps to ensure that the model retains its generalisation ability [17].

Dropout is another regularisation technique for reducing overfitting in neural networks.

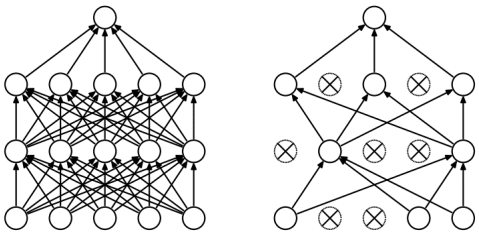


Figure 2.4: An example of a neural network with and without dropout. On the left, a standard neural network with 2 hidden layers. On the right, the same network with dropout applied. Crossed units have been dropped out [18].

It works by randomly setting a fraction of input units to 0 at each update during training time, as shown in Figure 2.4, which helps prevent overfitting. The fraction of dropping out is usually set between 0.2 and 0.5; at test time, no units are dropped out, but the same architecture is used with smaller weight values.

2.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a type of artificial neural network specifically designed to recognise patterns in sequences of data, such as handwriting [11].

RNNs are characterised by a loop in their architecture that allows information to be passed from one step in the sequence to the next. This means that when making a decision, an RNN considers not only the current input but also what it has learned from previous inputs. This ability to carry information across many time steps essentially creates a form of memory, which is a key advantage of RNNs over other types of neural networks. It allows RNNs to connect previous information to the present task, such as predicting the next word in a sentence or recognising a spoken word. This makes RNNs particularly well-suited to handling sequential data.

2.2.1 Simple Recurrent Neural Networks (RNN)

In a simple RNN, the state of a recurrent neuron at a given time step is a function of the input at that time step and its state at the previous time step:

$$h_t = f(h_{t-1}, x_t) \tag{2.7}$$

Here, h_t is the hidden state at time t, f is the RNN function, h_{t-1} is the hidden state at time $t-1$, and x_t is the input at time t.

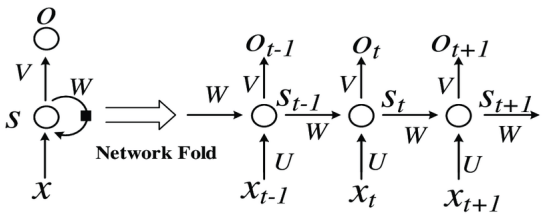


Figure 2.5: Basic recurrent neural network (RNN) structure. The unrolled RNN is on the left side and the network structure of the expanded RNN is on the right side; t represents the time, x is input data, O is output data, S is the network state, W is the update weight, V is the weight between the cell and the output, and U is the weight between the input and cell [19].

The RNN function, f, can be described as follows:

$$h_t = \sigma(W_{hh}h_{t-1} + W_{xh}x_t + b_h) \tag{2.8}$$

$$y_t = W_{hy}h_t + b_y \tag{2.9}$$

In these equations, W_{hh}, W_{xh}, and W_{hy} are weight matrices, b_h and b_y are bias vectors, and σ is the activation function (e.g., sigmoid or hyperbolic tangent). Similarly, x_t is the input at time t, h_t is the hidden state at time t, and y_t is the output at time t [11]. For a visualisation of the structure of a basic RNN, see Figure 2.5.

RNNs have found wide application in various fields. However, despite their simplicity, simple RNNs can be challenging to train effectively due to problems like 'vanishing' or 'exploding gradients'. These problems can occur due to the nature of backpropagation in RNNs, which involves propagating gradients through each time step of the sequence - if the sequence is long, the gradients can become extremely small (vanish), or extremely large (explode), due to the repeated multiplication of gradients in the backpropagation process. This makes the network hard to train, as small gradients slow down learning ('vanishing gradient' problem), and large gradients can cause learning to diverge ('exploding gradient' problem) [11]. Despite these challenges, simple RNNs remain a powerful tool for sequence-based problems, especially for short sequences.

2.2.2 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) is a type of Recurrent Neural Network (RNN) that was introduced to overcome the challenges of learning long-term dependencies in sequence prediction problems. Unlike traditional RNNs, LSTM has a unique design that enables it to forget or remember information for long periods of time, making it particularly effective for many complex tasks [20].

LSTM was designed to combat the vanishing gradient problem in standard RNNs. This problem arises during the training process, where the contribution of information decays geometrically over time, making it difficult for the RNN to learn to connect information over long gaps in time. LSTM addresses this issue by introducing a memory cell that can maintain information in memory for long periods of time. A key feature of the LSTM cell is that it can remove or add information to the cell state with special structures called gates. Gates control whether information can pass through or not. They are composed out of a sigmoid activation function and a pointwise multiplication operation [20].

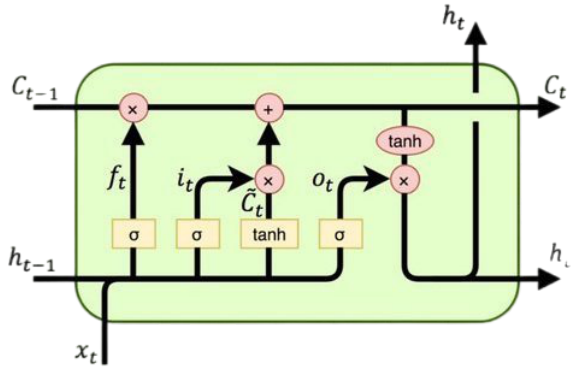


Figure 2.6: Long Short-Term Memory (LSTM) Cell Structure [21].

There are three types of gates within an LSTM cell:

1. **Forget Gate:** This gate decides what information should be thrown away or kept. Input data and the output of the last LSTM cell are passed through this gate, and after applying the sigmoid function, values between 0 and 1 are obtained. If a value is close to 0, it means

that the LSTM cell is going to forget all the information, and if it is close to 1, it is going to remember all the information. The forget gate is computed as:

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \tag{2.10}$$

2. **Input Gate:** This gate updates the cell state with new information. The input gate takes the current input and the output of the last LSTM cell, applies a sigmoid function to decide what values to update, and then a tanh function to create a vector of new candidate values that could be added to the state. The input gate is computed as:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \tag{2.11}$$

$$\tilde{C}_t = \tanh(W_{xC}x_t + W_{hC}h_{t-1} + b_C) \tag{2.12}$$

3. **Output Gate:** This gate decides what the next hidden state should be. The hidden state holds information from earlier inputs. The hidden state can also be used for predictions. The output gate is computed as:

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \tag{2.13}$$

$$h_t = o_t \odot \tanh(C_t) \tag{2.14}$$

In the equations above, f_t, i_t, $\tilde{C}_t$, C_t, and o_t are the forget gate, input gate, cell input, cell state, and output gate values at time t, respectively. W and b denote the weight matrices and bias vectors of each gate, and σ and tanh represent the sigmoid and hyperbolic tangent activation functions, respectively. The symbol $\odot$ denotes element-wise multiplication.

For see a visualisation of how these equations work together, see Figure 2.6.

Despite the additional complexity of LSTM networks compared to simple RNNs, they have proven to be a powerful tool for handling sequence-based problems, especially those involving long-term dependencies.

2.2.3 Gated Recurrent Units (GRU)

Gated Recurrent Units (GRUs) are a variant of RNN that aim to solve the vanishing gradient problem which can occur in traditional RNNs. GRUs use gating mechanisms that modulate the flow of information inside the unit, allowing the model to capture dependencies of various time scales effectively with fewer parameters than traditional RNNs.

The GRU has two gates, a reset gate and an update gate. The reset gate determines how to combine the new input with the previous memory, and the update gate defines how much of the previous memory to keep. If we denote the reset gate by r and the update gate by z, their computation can be formulated as follows:

$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \tag{2.15}$$

$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \tag{2.16}$$

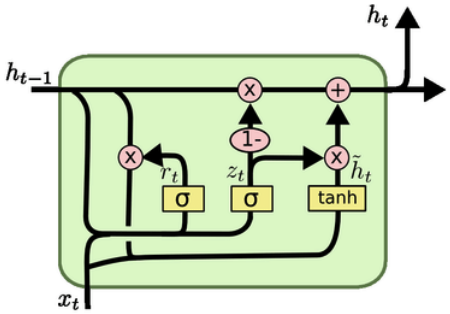


Figure 2.7: Gated Recurrent Unit (GRU) Structure [22].

Here, W_{xr}, W_{hr}, W_{xz}, and W_{hz} are weight matrices, b_r and b_z are bias vectors, and σ is the sigmoid activation function [23].

The candidate hidden state $\tilde{h}_t$ is computed as:

$$\tilde{h}_t = \tanh(W_{xh}x_t + W_{rh}(r_t \odot h_{t-1}) + b_h) \tag{2.17}$$

where, $\odot$ denotes element-wise multiplication.

Finally, the actual hidden state h_t is a linear interpolation between the previous hidden state h_{t-1} and the candidate hidden state $\tilde{h}_t$, with the update gate z_t acting as the interpolator:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{2.18}$$

Hence, the update gate z_t allows the GRU to capture long-term dependencies, as it can learn to keep the hidden state from changing, effectively allowing information to be carried across many time steps without vanishing or exploding. Moreover, the reset gate r_t allows the GRU to forget the previously computed state, providing a way to discard information that is no longer needed. For a better visualisation of how these equations work together, see Figure 2.7.

The main benefit between GRUs over LSTMs it that they have the advantage of being easier to modify, as they have fewer parameters, as well as having been shown to perform comparably to LSTMs on a variety of tasks [24]. Regarding this, investigations and analysis of the performance using these different types of RNN models for handwriting recognition will be discussed in later sections of this report.

2.2.4 Stacked Recurrent Neural Networks

Stacked Recurrent Neural Networks, also known as Deep RNNs, are a type of neural network architecture where multiple layers of RNNs are stacked on top of each other. This allows for the modeling of more complex patterns in the data by adding more layers of abstraction. Each layer in the stack processes the sequence with its own hidden state and passes its outputs sequence to the next layer [25].

For example, in the context of Long Short-Term Memory (LSTM) networks, a stacked LSTM model consists of multiple LSTM layers where each layer contains multiple LSTM cells. The output sequence of one layer of LSTM cells is used as the input sequence for the next layer. This allows the model to learn to represent the input data at various levels of abstraction, which can be beneficial for complex sequence prediction tasks.

Stacked RNN models can be trained using the same techniques as regular RNN models, but they are usually more computationally intensive to train due to the increased number of parameters, and the need to propagate gradients through multiple layers.

Despite the additional computational cost, stacked RNN models can provide improved performance on complex sequence prediction task, as they can capture hierarchical representations of the data by processing it at multiple levels of abstraction, which can lead to improved performance on complex sequence prediction tasks [25].

2.2.5 Bidirectional Recurrent Neural Networks

Bidirectional Recurrent Neural Networks (Bi-RNNs) are an extension of traditional RNNs. They are designed to capture information from both past (backward) and future (forward) states in the sequence. This is accomplished by implementing two separate layers for the input sequence: one layer processes the sequence in a forward direction, while the other processes the sequence in a backward direction. The outputs of these two layers are then combined to form the final output [26]. This design allows the model to capture both past and future context, which can be beneficial for sequence prediction tasks where future context is important.

Similar to stacked RNNs, bidirectional RNNs can be more computationally intensive to train due to the increased number of parameters and the need to propagate gradients in both directions. However, they can offer improved performance on tasks where both past and future context are important. The choice on whether to use a Bi-RNN model for handwriting recognition will be investigated further within this report.

2.3 Bézier Curves

A Bézier curve is a type of parametric curve that is defined by a set of control points, with a minimum of two but no upper limit. The curve shape is influenced by the position of these control points, where the curve begins at the first control point and ends at the last one, with the other control points guiding the curve's path by pulling or pushing the curve towards them. With this, the degree of a Bézier curve is given by the number of control points minus one [27].

2.3.1 Least Squares Fitting

The Bernstein matrix is a matrix representation of the Bernstein polynomials, which are used to define a Bézier curve. Each row of the matrix corresponds to a different value of the parameter t, which ranges from 0 to 1, and each column corresponds to a different term in the Bernstein polynomial. The Bernstein polynomial of degree n is given by:

$$B_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i} \tag{2.19}$$

where i is the current term of the polynomial. The Bernstein matrix is computed by evaluating this polynomial for each value of t and each term i [27].

Once the Bernstein matrix is computed, the least squares fitting is performed to find the control points of the Bézier curve that best fit the given data points, by minimising the sum of the squared differences between the actual data points and the points on the Bézier curve.

This is performed by first computing the pseudoinverse of the Bernstein matrix. The pseudoinverse is a generalisation of the matrix inverse that can be used even when the matrix is not square or when it is not of full rank; it has the property that when it is multiplied with the matrix, it gives the identity matrix.

The system of equations to solve is:

$$M\mathbf{P} = \mathbf{X} \tag{2.20}$$

where M is the Bernstein matrix, $\mathbf{P}$ is the vector of control points, and $\mathbf{X}$ is the vector of data points [27]. This system is usually overdetermined, meaning there are more equations than unknowns,

because the number of data points is typically greater than the number of control points. Therefore, an exact solution may not exist.

In this case, we can find the least squares solution, which minimises the sum of the squares of the residuals (the differences between the data points and the corresponding points on the Bézier curve). The least squares solution can be found by computing the pseudoinverse of the Bernstein matrix and multiplying it with the vector of data points [27]:

$$\mathbf{P} = M^{+}\mathbf{X} \tag{2.21}$$

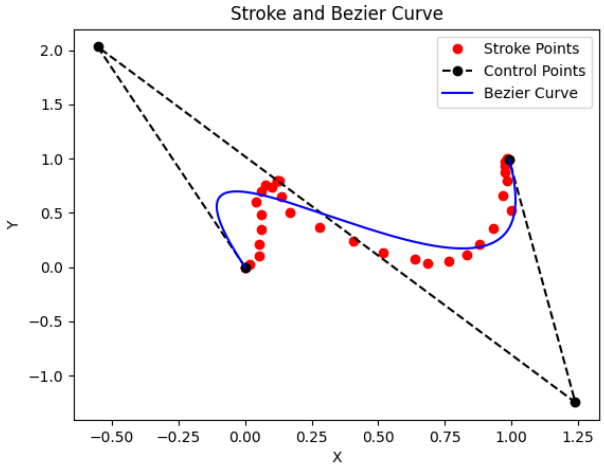


Figure 2.8: A Bézier curve of degree 3 fitted to a set of (x, y) stroke points.

This gives the control points that define the Bézier curve that best fits the data points in the least squares sense. Following this, to get the (x, y) coordinates of the points on the Bézier curve, the control points is multiplied with the Bernstein polynomial matrix. For an example of a Bézier curve fitted to (x, y) stroke points, see Figure 2.8.

2.3.2 De Casteljau's Algorithm

De Casteljau's algorithm is a recursive method to evaluate Bézier curves at a given value of t, and is numerically stable. The algorithm begins by linearly interpolating between each pair of consecutive control points, then between those resulting points, and so on, until only one point is left, which lies on the Bézier curve [28]. The algorithm can also be used to split a Bézier curve at a given value of t into two separate Bézier curves [29].

Within this research, the Least Squares Fitting approach is used instead of De Casteljau's algorithm for several reasons. Firstly, it directly follows from the mathematical definition of Bézier curves, which makes it conceptually straightforward. Secondly, it is primarily because it is computationally efficient, especially for high-degree curves and for a large number of points on the curve, as it involves matrix operations that can be efficiently performed using numerical libraries, such as *NumPy*.

However, the Least Squares Fitting approach is not as numerically stable as De Casteljau's algorithm, especially for very high-degree curves and for extreme values of t. De Casteljau's algorithm, on the other hand, is more numerically stable and is particularly useful for high-degree curves and for constructing subcurves [28], but it is more computationally intensive and does not directly give the points on the curve for all values of t.

2.3.3 Velocity and Acceleration

The first derivative of a Bézier curve, which represents the velocity of the stroke, is given by:

$$V(t) = n \sum_{i=0}^{n-1} (P_{i+1} - P_i) B_{i,n-1}(t) \tag{2.22}$$

where n is the degree of the Bézier curve, P_i are the control points, and $B_{i,n-1}(t)$ is the Bernstein polynomial of degree $n-1$ [30, 31].

The second derivative of a Bézier curve, which represents the acceleration of the stroke, is given by [30, 31]:

$$A(t) = n(n-1) \sum_{i=0}^{n-2} (P_{i+2} - 2P_{i+1} + P_i) B_{i,n-2}(t) \tag{2.23}$$

The velocity equation provides the rate of change of the position along the curve, while the acceleration equation gives the rate of change of the velocity [30]: the velocity vector $V(t)$ points in the direction of the curve at parameter t and its magnitude is the speed of the motion at that point. The acceleration vector $A(t)$, on the other hand, points in the direction of the change of the velocity vector. If the acceleration is in the same direction as the velocity, the speed is increasing. If the acceleration is in the opposite direction, the speed is decreasing. If the acceleration is perpendicular to the velocity, the speed is constant, but the direction is changing, which is the case in circular motion.

These quantities offer insights into both the geometric and dynamic characteristics of the curve. They provide understanding of the rate of change in position and direction of the curve, its curvature, and the behaviour of motion along the curve [30]. In subsequent sections, we will explore the potential of these values as input features for a Bézier curve that is fitted to a stroke.

2.4 Neural Machine Translation (NMT)

Neural Machine Translation (NMT) is a subfield of natural language processing (NLP) that utilises deep learning models, particularly Recurrent Neural Networks (RNNs), to perform machine translation. It is an approach to machine translation that employs neural network models to predict the likelihood of a sequence of words, typically modeling entire sentences in a single integrated model [32]. In contrast to traditional statistical machine translation (SMT), which involves separate translation and language models, NMT models the entire process through one unified neural network, which reads a sentence and outputs a correct translation [33].

A common structure of an NMT model is the sequence-to-sequence (seq2seq) model with an encoder-decoder architecture [33]. The encoder takes in the input sequence and compresses the information into a context vector, a fixed-length target sequence. For example, in our case, given a sequence of stroke data (consisting of (x, y) points), we can fit Bézier curves to represent each stroke. These curves effectively encapsulate the complexity and variability of handwriting strokes in a mathematically manageable form, where the resultant Bézier curves serve as a form of 'embedding', effectively transforming the raw handwriting data into a more manageable and expressive format. The decoder then outputs the translation based on the context vector.

While initially the context vector was represented as the final hidden state of the encoder, this approach has its limitations, particularly when dealing with long sentences - the fixed-length context vector might not be sufficient to accurately represent the semantic information for longer sentences, which could reduce the quality of translations for these.

Regarding training, NMT models are trained end-to-end using stochastic gradient descent (SGD) and backpropagation [33]. The objective is to minimise the negative log-likelihood of the correct translation, given the input sentence.

2.5 Connectionist Temporal Classification (CTC)

Connectionist Temporal Classification (CTC) is an approach utilised in neural networks, particularly with Recurrent Neural Networks (RNNs), to tackle sequence-to-sequence problems where the alignment between the input and output sequences isn't directly observable or known. This method introduces an efficient pathway to train sequence models without the need for pre-segmented training data [34, 35].

In the CTC framework, for an input sequence X of length T, the RNN provides a sequence of output probability distributions $Y = y_1, y_2, ..., y_T$. This sequence is over an extended alphabet that includes the original label set plus an additional blank label. The probability $y_t(k)$ of outputting label k at time t is computed by the softmax function [34]:

$$y_t(k) = \frac{\exp(a_t(k))}{\sum_{k'} \exp(a_t(k'))} \tag{2.24}$$

where $a_t(k)$ is the activation of label k at time t. The softmax function is a way to convert a set of K real numbers into a probabilistic distribution. It does this by taking the exponential of each input value, then normalising these values so they sum to 1, thus producing valid probabilities. This allows the model to provide a confidence level for each possible output label.

Alignments between X and the target sequence l are represented by all possible label sequences π of length T that, when blank symbols and repeated labels are removed, map to l. The probability of a given label sequence π is:

$$P(\pi|X) = \prod_{t=1}^{T} y_t(\pi_t) \tag{2.25}$$

CTC's objective is to maximise the total probability of the correct label sequence l, summed over all its valid alignments π:

$$P(l|X) = \sum_{\pi \in \Pi(l)} P(\pi|X) \tag{2.26}$$

where $\Pi(l)$ represents the set of all valid alignments of l [34].

The core principle behind CTC involves using the forward-backward algorithm, which is a dynamic programming technique. This method calculates the sum of probabilities of all potential alignments, thereby providing a mechanism to bypass the need for explicit alignment data. This is achieved by computing forward and backward variables, $\alpha_t(s)$ and $\beta_t(s)$, that represent total probabilities of observing prefixes and suffixes of the alignments at time t in state s. This computation is performed efficiently through the forward-backward algorithm [35].

$$\alpha_t(s) = \sum_{\pi \in \Pi_t(s)} \prod_{t'=1}^{t} y_{t'}(\pi_{t'}) \tag{2.27}$$

Where $\Pi_t(s)$ represents the set of all alignments of the prefix $l_1, l_2, ..., l_s$ of length t. The backward variable $\beta_t(s)$ is similarly defined, but it considers all paths starting from state s at time t. For a better visualisation of how CTC works, see Figure 2.9.

The CTC loss is then defined as the negative logarithm of the total probability of the correct label sequence, summed over all its alignments:

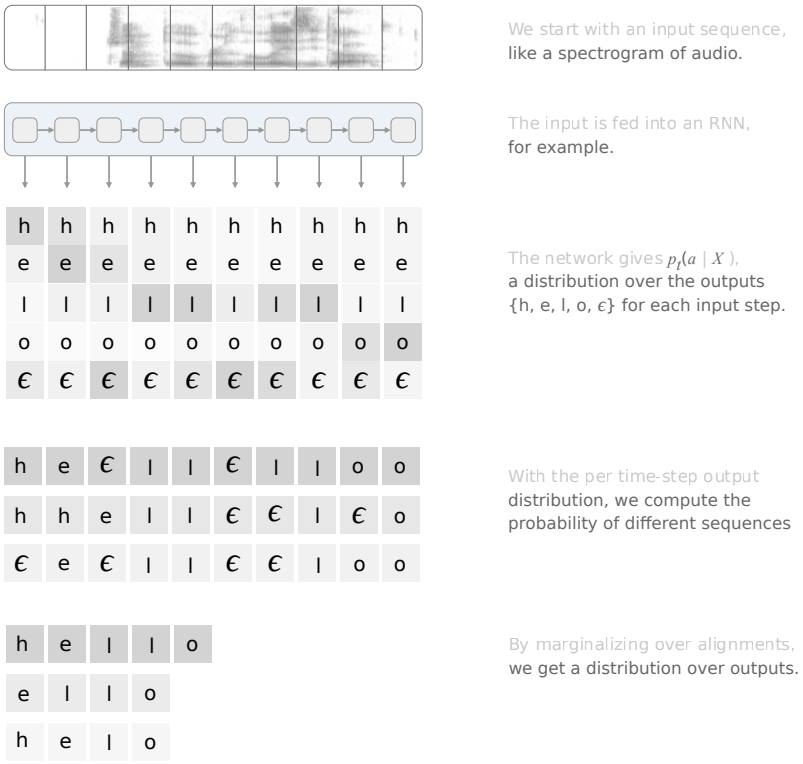


Figure 2.9: Visualising CTC alignments: from time-step probabilities to output sequence probability [36].

$$L(X, l) = -\log P(l|X) \tag{2.28}$$

This loss is minimised using gradient-based optimisation methods such as stochastic gradient descent (SGD) [35, 34].

2.5.1 Greedy Decoding

Greedy decoding, also known as "best path decoding", is the simplest and fastest method for obtaining the output sequence from the probabilities generated by the model. At each time step t, the label k with the maximum probability $y_t(k)$ is selected. Adjacent identical labels and blank labels are then removed to obtain the final output sequence [34]:

$$l_{greedy} = k_1, k_2, ..., k_T \quad \text{where} \quad k_t = \arg\max_k y_t(k) \tag{2.29}$$

While being computationally efficient, greedy decoding might not always yield the most probable sequence, since it doesn't account for the joint probabilities of sequences, but instead independently selects the most probable label at each time step. However, it's very simple to implement.

2.5.2 Beam Search Decoding

To overcome the limitations of greedy decoding and find a sequence that is closer to the globally most probable sequence, we can use beam search decoding. Beam search is a heuristic search algorithm that explores the space of possible output sequences in a breadth-first manner, but it maintains only a limited "beam" of the most promising candidates at each time step [33].

For CTC, beam search decoding can be implemented in a way that extends the candidates in the beam with all possible labels (including the blank label), and computes the total log-probability

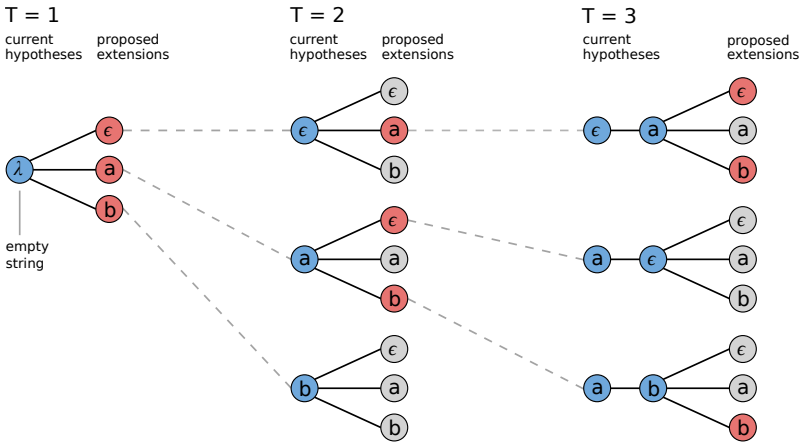


Figure 2.10: Visualising beam search decoding with an alphabet of $\{\epsilon, a, b\}$ and a beam size of three [36].

of each extended candidate. It then selects the top B candidates (where B is the beam width) to include in the beam for the next time step. Finally, the candidate with the highest total log-probability that ends with a non-blank label or with a blank but had a non-blank label before it, is selected as the output sequence [34, 35]:

$$l_{beam} = \arg\max_{l} P(l|X) \quad \text{where} \quad P(l|X) \approx \max_{l \in B} P(l|X) \tag{2.30}$$

For a better visualisation of the beam search decoding algorithm, see Figure 2.10.

Although beam search decoding is more computationally demanding than greedy decoding, it typically yields better results, as it considers a larger portion of the output sequence space and takes into account the joint probabilities of label sequences. However, the choice of the beam width B can significantly affect the performance and computational requirements of beam search decoding, with a larger B leading to better results but higher computational cost.

2.5.3 Applying CTC in Handwriting Recognition

In the context of online handwriting recognition, where we aim to predict the correct characters given a sequence of stroke data represented as Bézier curves, CTC offers several benefits. Given the inherent variability in handwriting, where the number of strokes and the time taken to write a particular character can vary significantly between different instances and individuals, the use of CTC allows the model to handle these variations by learning to predict the blank token when necessary, thereby improving the robustness of the model.

In our setting, the RNN model with CTC loss would take as input a sequence of Bézier curves (each curve representing a stroke) and output a sequence of characters, which can then be decoded by either using greedy or beam search decoding. During training, the CTC loss would be computed between the model's output and the target sequence of characters, and the model's parameters would be updated to minimise this loss.

This approach has the advantage of being end-to-end trainable, meaning that all components of the system can be trained jointly to optimise the overall performance. This also allows the model to learn to extract useful features directly from the Bézier curves.

CTC also enables the model to handle variable-length input and output sequences, which is essential in handwriting recognition, as both the number of strokes and the length of the written text can vary between different instances. Furthermore, by summing over all possible alignments, CTC allows the model to be robust to small variations in the stroke timing and sequence, which is a common characteristic of handwriting. Overall, the use of CTC in conjunction with RNN type

models and Bézier curve representations provides a powerful and flexible framework for handwriting recognition [36].

2.5.4 Limitations

Despite its strengths, CTC has some limitations - as CTC is fundamentally designed to tackle sequence prediction problems where explicit alignment between input and output sequences is not available [34], it imposes an assumption of conditional independence between output labels. In simpler terms, it presumes that each label in the output sequence is independent of others, or that the probability of each output label is unaffected by its preceding or subsequent labels [35].

This assumption can be limiting in numerous sequence prediction tasks, including natural language processing and handwriting recognition, where the occurrence of a label is often influenced by those surrounding it [37]. For example, in English text, the letter 'u' typically follows 'q'. In the case of handwriting recognition, a certain stroke can be influenced by the preceding stroke when forming a specific character. Hence, the inherent limitation in CTC's design, that is, its disregard for these dependencies while making predictions, can hinder its effectiveness in accurately capturing patterns in language or handwriting strokes [38].

To overcome this limitation, one common approach is to utilise language models or implement beam search during decoding to better consider the context of the sequence and recognise the dependencies between labels [32]. Alternatively, attention mechanisms can be employed by using for example a transformer model, allowing the model to "refer back" to the input and previously predicted output labels while generating the current label, thus improving the model's prediction accuracy [39].

2.6 Accuracy Metrics

2.6.1 Character Error Rate (CER)

Character Error Rate (CER) is a widely used metric for evaluating the performance of handwriting recognition models. CER measures the dissimilarity between the predicted character sequence and the ground truth sequence, by counting the number of character-level operations (insertions, deletions, and substitutions) required to convert the predicted sequence into the target sequence, and then normalising this count by the length of the target sequence [40]:

$$CER = \frac{S + D + I}{N} \tag{2.31}$$

Where:

- S is the number of substitutions
- D is the number of deletions
- I is the number of insertions
- N is the total number of characters in the target sequence

CER offers a granular level of evaluation as it considers each individual character when assessing the performance. However, this can be a disadvantage in certain scenarios where the focus is more on the accuracy of recognised words rather than individual characters.

2.6.2 Word Error Rate (WER)

While CER offers a detailed measure of model performance, Word Error Rate (WER) provides a more overarching view of the model's ability to correctly predict sequences. WER is often used in

conjunction with CER to provide a comprehensive view of model performance. Similar to CER, WER is computed by counting the number of word-level operations (insertions, deletions, and substitutions) required to transform the predicted sequence into the target sequence, and then normalising this count by the total number of words in the target sequence [40]:

$$WER = \frac{S + D + I}{N} \tag{2.32}$$

Where:

- S is the number of word substitutions
- D is the number of word deletions
- I is the number of word insertions
- N is the total number of words in the target sequence

By focusing on words, WER provides a more human-centric view of the model's performance, as it aligns more closely with how humans perceive errors in language recognition tasks. However, it's important to note that WER can sometimes overlook fine-grained errors that CER can detect, which is why these two metrics are often used together for a comprehensive evaluation of handwriting recognition models.

Chapter 3

Related Work

The field of handwriting recognition has seen significant advancements in recent years, with researchers exploring various innovative approaches to improve accuracy and efficiency. This section reviews some notable studies that have contributed to the current state of the field.

3.1 Stroke-Based Cursive Character Recognition

The paper titled 'Stroke-Based Cursive Character Recognition' by K.C. Santosh and Eizaburo Iwata [1] presents a different approach to handwriting recognition, focusing on cursive scripts like Devanagari. The paper's approach to handwriting recognition is based on stroke analysis rather than the use of CTC loss or Bézier curves. The authors propose a character recognition framework that includes learning and testing modules. In the learning module, handwritten strokes are learnt or stored using stroke pre-processing, feature selection, and clustering to form a template. The testing module then matches each test stroke with the templates to identify the most similar one.

The authors also emphasise the importance of using stroke spatial descriptions in their approach. They argue that the location of strokes is a crucial feature in distinguishing between characters, especially in cursive scripts like Devanagari. The paper proposes a method for analysing handwritten characters based on both the number of strokes and their spatial information.

This approach is different from the use of CTC loss and Bézier curves in that it focuses on the analysis of individual strokes and their spatial relationships rather than the overall shape of the handwriting or the prediction of character sequences. This could potentially offer a more nuanced and detailed analysis of handwriting, especially for complex and cursive scripts, where the number and arrangement of strokes can vary significantly between characters, and the overall shape of the handwriting might not be as informative.

In contrast, the use of CTC loss and Bézier curves in handwriting recognition has shown to be particularly effective for languages with simpler scripts, such as English, as Bézier curves provide a mathematical means to represent smooth and continuous curves, which are common in handwriting. By fitting Bézier curves to handwriting strokes, the model can capture the overall shape and flow of the handwriting, which can be crucial for recognising characters and words. This is primarily due to the nature of these techniques and the characteristics of these languages, where characters and words often have distinct and recognisable shapes.

However, it's important to note that the effectiveness of these techniques can depend on various factors, including the quality and quantity of the training data, the complexity of the handwriting, and the specific characteristics of the language script. Therefore, while CTC loss and Bézier curves might generally be more suitable for simpler languages like English, they might not always be the best choice for every scenario or dataset.

3.2 Google's LSTM-based Online Handwriting Recognition

The initial study, 'Fast Multi-language LSTM-based Online Handwriting Recognition' by Victor Carbune and his team at Google [2], introduces a unique system for online handwriting recognition. This system supports 102 languages and employs a deep neural network architecture. It has superseded their previous Segment-and-Decode-based system, leading to a reduction in the error rate by 20-40% for the majority of languages. The system utilises sequence recognition methodologies and a novel input encoding using Bézier curves, resulting in recognition times that are up to 10 times faster than their previous system. The team conducted a range of experiments to ascertain the optimal configuration of their models and reported the results of their setup on multiple public datasets.

However, it is worth noting that Google did not open-source any of the code in this study. As handwriting recognition is becoming increasingly important, particularly approaches that are not reliant on Optical Character Recognition (OCR) and learn from actual stroke points to recognise, making such implementations available for further research and development would be beneficial for the research community. Open-sourcing this work could foster further innovation in the field and enable other researchers to build upon this foundational work.

Regarding potential areas for further exploration and improvement in this study, the researchers employed a degree-3 Bézier curve and an algorithm involving curve splitting for their experiments. Investigating the use of Bézier curves of different degrees without curve splitting could potentially yield different results and insights into the handwriting recognition process.

The feature vectors used in this study are unique and fewer in number. They include the vector between the endpoints, the distance between the control points and the endpoints relative to the distance between the endpoints, the angles between control points and endpoints in radians, the time coefficients, and a Boolean value indicating whether this is a pen-up or pen-down curve. Future research could investigate the use of different feature vectors, such as acceleration, velocity, directness, direction, measure of curvature and directness, and among others.

In terms of the model architecture, the authors used bidirectional Long Short-Term Memory (LSTM) networks. However, exploring other Recurrent Neural Network (RNN) models, such as standard RNNs and Gated Recurrent Units (GRUs), could be worthwhile, especially given that GRUs might potentially outperform LSTMs. Additionally, the use of non-bidirectional models and potentially transformer models could also be investigated for their performance in this task.

In conclusion, while the paper presents a novel approach to multi-language online handwriting recognition, there are several potential areas for further exploration and improvement, including the use of different Bézier curve degrees, feature vectors, and RNN models. The open-sourcing of such work could significantly contribute to the advancement of the field.

3.3 Pentelligence

The study titled 'Pentelligence: Combining Pen Tip Motion and Writing Sounds for Handwritten Digit Recognition' by Schrapel, Stadler, and Rohs [41] introduces a unique approach to handwritten digit recognition. The authors propose a system called Pentelligence, which captures the pen tip's motions and sound emissions during writing. The system operates on regular paper without requiring a separate tracking device. The results, based on a dataset of 9408 handwritten digits from 26 individuals, showed that the combined motion and sound approach outperformed single-sensor approaches, achieving an accuracy of 78.4% for 10 test users.

However, the study has certain limitations that could be addressed in future research. One of the main limitations is that users are required to use a specific pen, which may not be an inclusive design and scalable approach to research. Users should ideally be able to use any type of pen. Therefore, it would be beneficial to research into building a universal API that people could use to predict characters of strokes, just by knowing the (x, y) points of the strokes. This would also allow other researchers in the future to build upon this research more easily.

Another limitation is that the system was not tested very rigorously, with only 10 users being tested. Given the variability in handwriting styles among individuals (such as cursive writing, different writing speeds, etc.), it would be worthwhile to build a model and test it on more data. This could potentially lead to more robust and generalisable results.

In summary, while the Pentelligence system represents a significant advancement in the field of handwriting recognition, offering a practical solution that operates on regular paper without the need for specialised tracking devices, there are several potential areas for further exploration and improvement.

3.4 Summary

Some of these studies we have looked at show how handwriting recognition research is evolving, with advancements in deep learning and novel approaches offering promising directions for future work. In this research, I aim to address some of the identified limitations and explore the potential research directions suggested in the previous subsections. I believe it's important to have research designs that work for everyone, to test these designs thoroughly, and to open-source any code. This will help to drive further innovation in the field of handwriting recognition.

Chapter 4

ISGL Dataset Character Recognition

This research began with the application of recurrent neural network models to the task of handwritten character recognition. The aim was to establish a basic baseline and potentially investigate the issue of 'vanishing gradients', a problem that RNNs are susceptible to. For this purpose, the Intelligent System Group Lab (ISGL) online dataset [42] was utilised. This dataset comprises independent online handwritten English characters collected from 64 writers. The data was gathered using a digitised tablet coupled with a custom-developed acquisition software, which recorded key information during the writing process, including the number of strokes, pen-up and pen-down events, (x, y) pixel positions (stroke points), and the time taken to write each character.

In terms of data usage rights, the ISGL online dataset is licensed under the Creative Commons Attribution 4.0 International license. This license permits sharing, copying, and modifying the dataset, provided appropriate credit is given, a link to the license is included, and any changes made to the dataset are indicated. However, it does not allow implying endorsement by the rights holder for any use of the dataset [42].

4.1 Model Architecture

The ISGL dataset character recognition system is designed to process a list of (x, y) stroke points as input. These points are fed into a two-layer recurrent neural network model. The model then generates a softmax output, which is decoded to produce a character prediction. The entire process is designed to be efficient, aiming to deliver the prediction in minimal time. For a visual representation of the system design, refer to Figure 4.1.

4.2 Data Extraction & Preprocessing

The dataset is structured into directories of text files, with distinct files for capital letters, lowercase letters, and numbers. Each text file encapsulates the written character and associated information, such as pen lift events, (x, y) stroke points, the time taken to write each stroke, and the number of strokes. To see some of the samples in the ISGL dataset, see Figure 4.2.

For model training, the data was preprocessed by extracting all stroke points and labels from the text files. All stroke points were normalised to ensure that all (x, y) coordinates fall within the range of 0 to 1, inclusive. Additionally, the points were shifted so that the minimum x and y coordinates align at 0. This normalisation process is crucial as it allows the model to effectively learn from the input data without any particular feature overshadowing others due to its scale. Moreover, normalising the stroke points aids in the efficient convergence of the gradient descent optimisation during the training phase. It also ensures consistency in the input data, which is particularly important when dealing with a variety of handwriting styles and sizes present in the

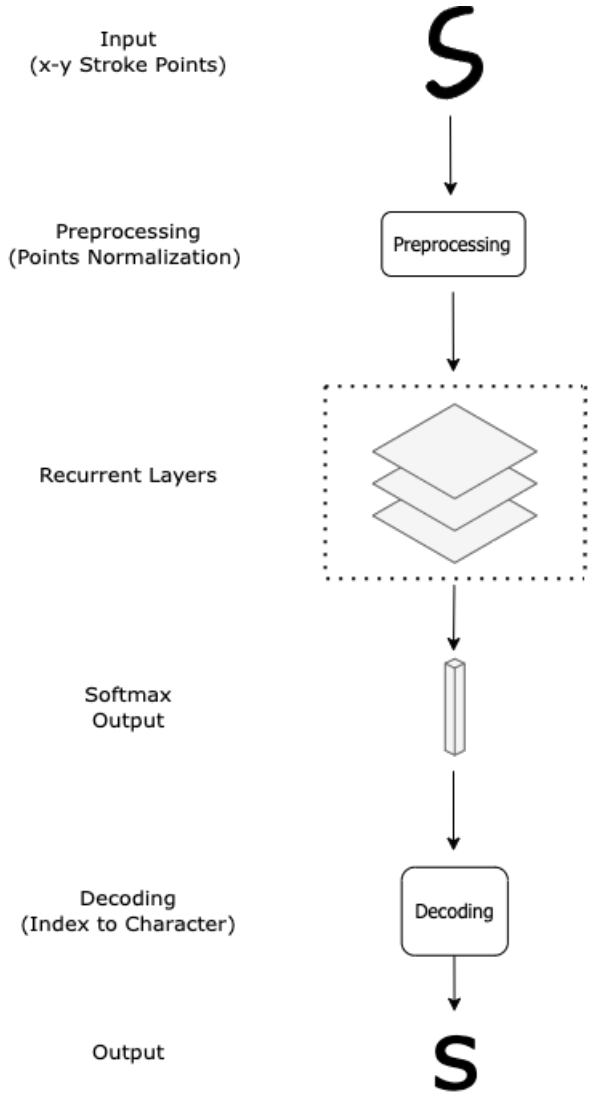


Figure 4.1: Architecture diagram of the ISGL dataset character recognition pipeline.

dataset. This step in data preprocessing plays a vital role in to ensure the model is able to learn effectively.

Also, it's important to note that some of the extracted data had to be discarded because certain text files contained no stroke points. This was likely due to occasional errors in the recording of pen nib tracking by ISGL. After preprocessing and filtering the data, I divided it into training, validation, and test sets, following an 80-10-10 split. This setup enabled efficient model training and testing.

Moreover, the pixel positions were the primary focus during data extraction, as no other information directly related to each pixel position was available. For instance, the dataset did not provide time stamps for each pixel position (stroke point), only the time taken to write the entire stroke. Consequently, extracting the time taken was not particularly beneficial, as it would only yield the average speed for the entire stroke, which varies greatly for each writer, and not for example the velocity normalised at each pixel position.

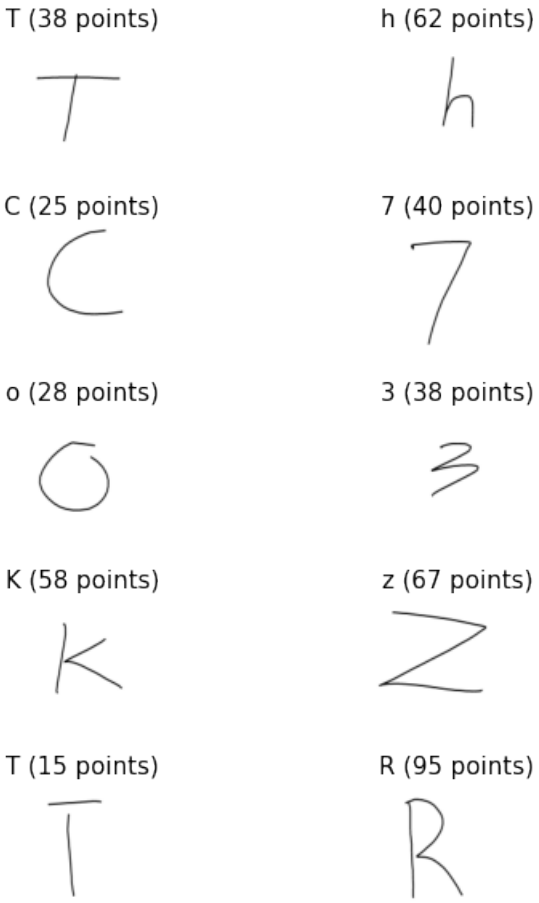


Figure 4.2: 10 random samples from the ISGL dataset with label and number of stroke (x, y) points made to write it above each written character.

4.3 Initial Model Training & Evaluation

The Recurrent Neural Network (RNN), Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU) models were all trained over a span of 8 epochs using cross-entropy loss and an Adam optimiser. During training, a batch size of 32 was used, along with a hidden layer size of 32. The learning rate for the training process was set at 3×10^{-4}.

However, the performance of the models was low, which was anticipated due to the 'vanishing gradient' problem. This issue arises when dealing with excessively long (x, y) stroke point sequences, with the maximum sequence length in the extracted data being 377 and the average being 49. However, other factors could also have contributed to the suboptimal performance, including the absence of feature engineering on the extracted data and the scarcity of training data (only 7985 total samples available for training, validation, and testing across all lowercase and uppercase letters, and digits).

Figures 4.3 and 4.4 illustrate a divergence between the training and validation loss. While the training loss is decreasing, the validation loss is on the rise. This trend suggests that the model may be overfitting on the training data, which implies that the model is learning the specifics of the training data too well, but is having difficulty generalising effectively to unseen data.

Figures 4.5 and 4.6, reveal that the accuracy percentages for all models are notably low, averaging around 1.05%.

A deeper look into Figure 4.7 shows that the distribution of gradients for the learnable hidden-hidden weights of the simple RNN model is quite sparse, with the majority of gradients being close

Figure 4.3: Training loss plots for the RNN, LSTM and GRU models trained on the ISGL dataset.

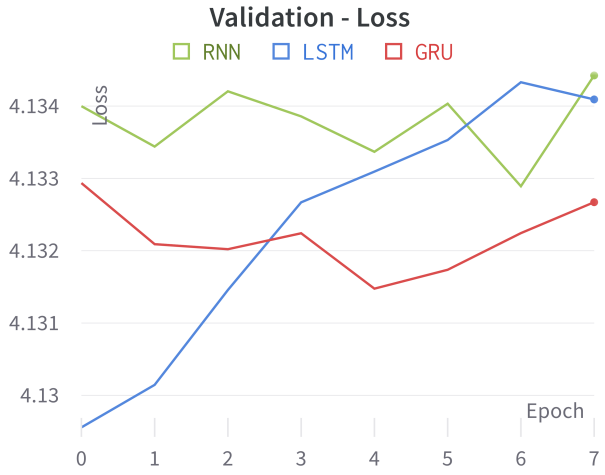


Figure 4.4: Validation loss plots for the RNN, LSTM and GRU models trained on the ISGL dataset.

to 0, as indicated by the blue line in the figure. This is as a result of the 'vanishing gradient' problem (discussed above and also in more detail in Subsection 2.2.1). Consequently, the accuracy percentages remain low across all models, including GRUs and LSTMs, as these models can also be affected by the 'vanishing gradient' problem when dealing with long sequence lengths.

Despite the challenges encountered and the lack of sufficient training data, this initial implementation served as a valuable baseline for further research and development. Also, based on the results, it was clear that additional data and more sophisticated methods of feature extraction were needed to build more powerful models capable of recognising handwriting. The next chapter will discuss these approaches and the subsequent research conducted to improve model performance.

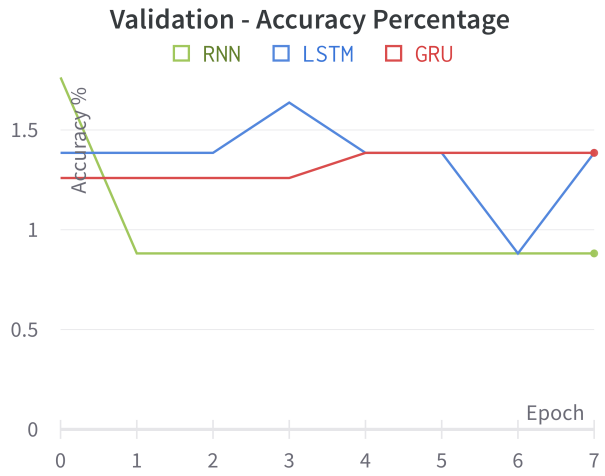


Figure 4.5: Validation accuracy (percentage correct) plots for RNN, LSTM and GRU models.

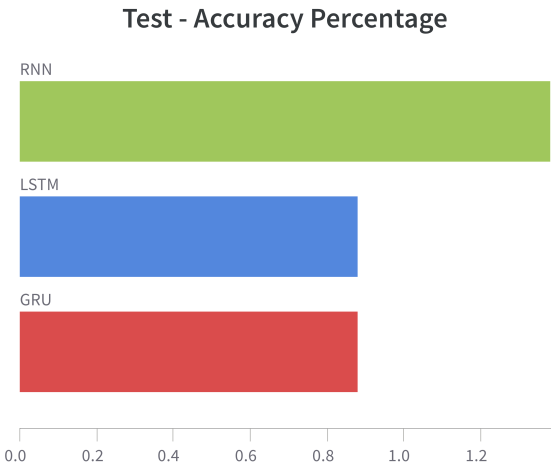


Figure 4.6: Test accuracy (percentage correct) plots for RNN, LSTM and GRU models.

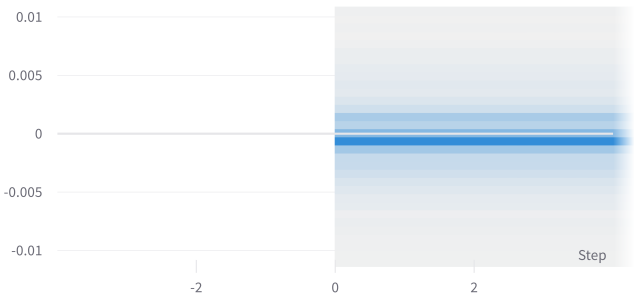


Figure 4.7: Distribution of gradients of the learnable hidden-hidden weights of the 2nd layer in the RNN model, based on the ISGL character recognition dataset.

Chapter 5

IAM-OnDB Dataset Handwriting Recognition

The IAM On-Line Handwriting Database (IAM-OnDB) [43] is a rich source of handwritten English text samples, collected on a whiteboard. The database comprises unconstrained handwritten text, captured using an E-beam system and stores the stroke data in XML format. The dataset is a result of contributions from 221 writers, and it contains over 1,700 acquired forms, 13,049 isolated and labeled text lines, and 86,272 word instances drawn from an 11,059 word dictionary.

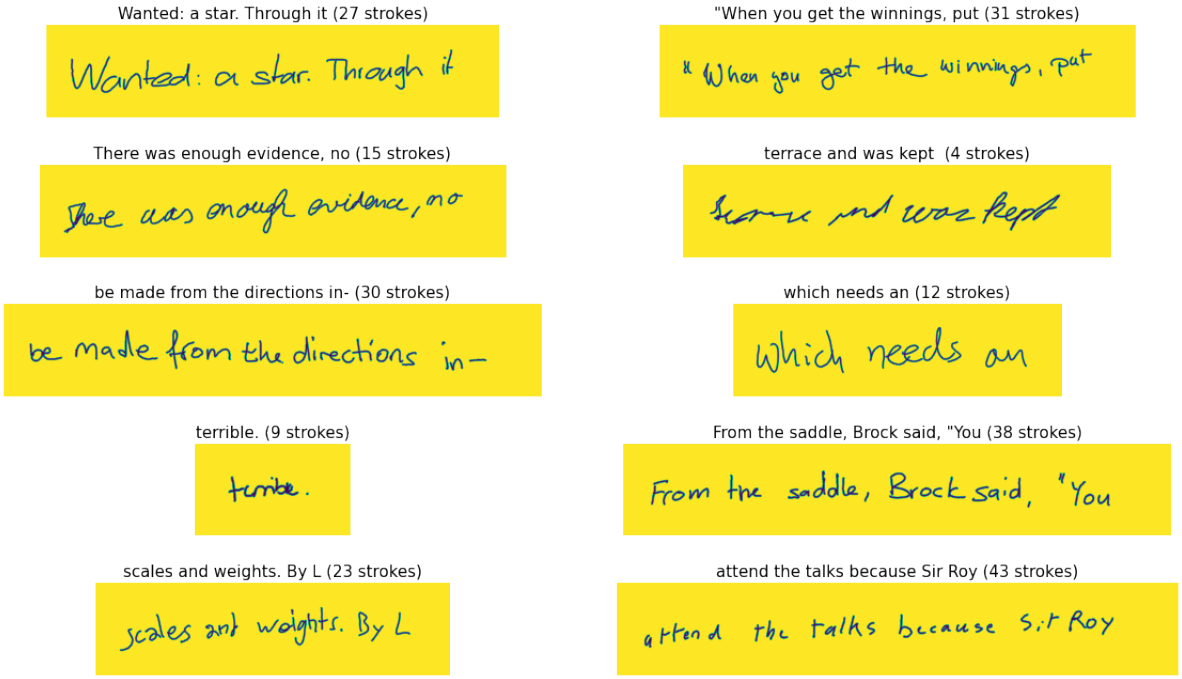


Figure 5.1: 10 random samples from the IAM-OnDB dataset with label and number of strokes taken to write it above each written label.

The IAM-OnDB dataset provides a more extensive set of handwriting samples compared to the ISGL dataset. This increased volume and variety of data is expected to improve the performance of the models by providing a broader range of handwriting styles and patterns for the models to learn from - to see some of the samples in the dataset, see Figure 5.1.

Regarding the usage rights, the IAM-OnDB dataset is open to the public and can be freely used for non-commercial research purposes. The creators of the dataset have a simple request: if you

use any data from the IAM-OnDB dataset, you should register on their website. This allows them to keep track of who is using their data. Also, if you are publishing any scientific work that utilises the IAM-OnDB dataset, you should include a reference to their database in your publication [43].

5.1 System Design

The handwriting recognition model for the IAM-OnDB dataset is designed to process a list of strokes as input, where each stroke is a sequence of (x, y) points with associated timestamps. These strokes undergo preprocessing, and a Bézier curve is fitted to each stroke. The features derived from these Bézier curves are then fed into a recurrent neural network model.

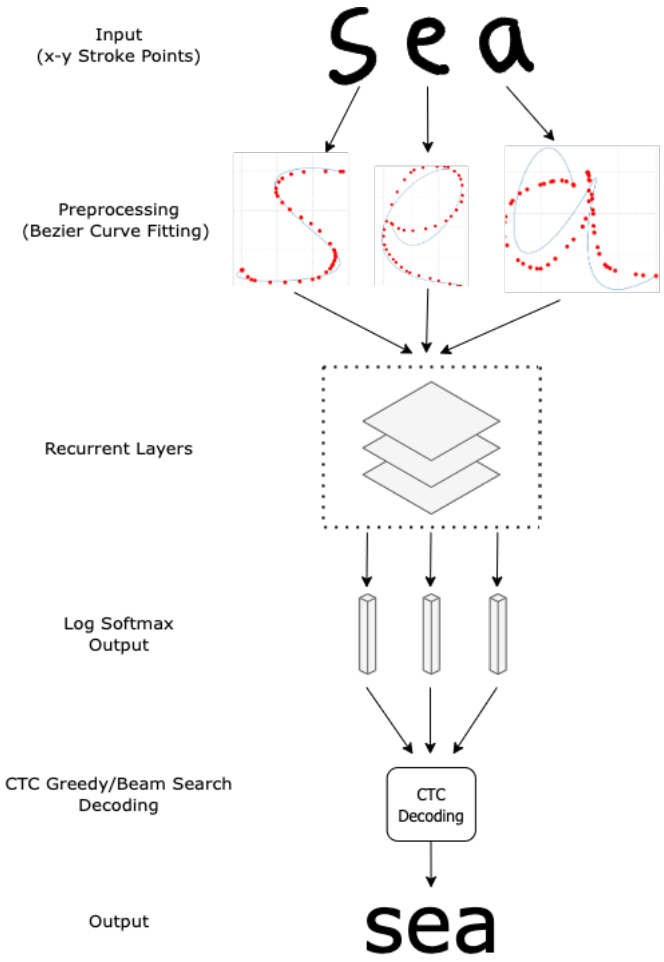


Figure 5.2: Model architecture diagram for IAM-OnDB dataset handwriting recognition.

The model generates a log softmax output, which is then decoded using either a Connectionist Temporal Classification (CTC) greedy or beam search decoding method to produce a prediction. The entire process is designed to be efficient, with the aim of delivering the prediction in minimal time. For a visual representation of the model architecture, refer to Figure 5.2.

This model is used within the backend of a web application that allows users to experiment with different parameters of the model and have their handwriting recognised by the model. Figure 5.3 provides a simple system design of the web application.

In this setup, the frontend sends a HTTP POST request to the backend in a JSON format. This request includes the strokes to be recognised, the name of the model to be used, the degree of the Bézier curve, and the points per second (downsampling rate) that should be used during preprocessing.

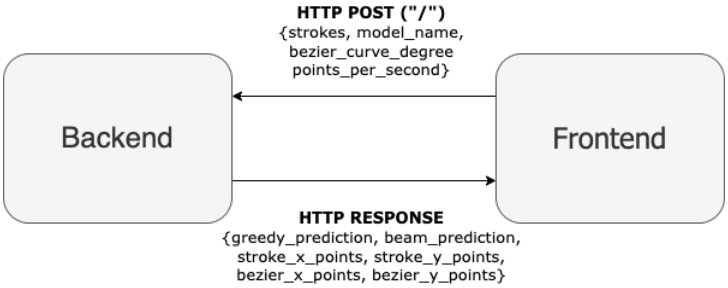


Figure 5.3: System design of the frontend and backend of the IAM-OnDB dataset handwriting recognition web application.

Upon receiving the request, the backend processes the strokes using the specified parameters and generates a prediction. The backend then sends a response back to the frontend in a JSON format. This response includes the predictions obtained from both the CTC greedy and beam search decoding methods, the original stroke points, and the points of the fitted Bézier curve. The frontend uses this information to create visualisations for the user.

5.2 Data Extraction & Preprocessing

The IAM-OnDB dataset is organised into separate directories. One directory contains the transcriptions in plain text files. Another directory houses multiple sub-directories, each corresponding to a transcription. These sub-directories contain XML files that store the stroke data for each line in a transcription, including (x, y) points, timestamps, and coordinates that describe the size of the whiteboard used.

The extracted data was stored in two *NumPy* arrays: one for the line labels from all transcriptions and another for the corresponding stroke data for each line label. For example, if a line label was written with two strokes, the stroke data for both strokes, including the (x, y) points and timestamps for each point, were stored in the array.

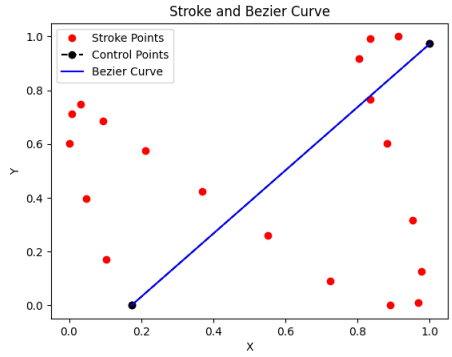
The labels were encoded by assigning an index to each character in the label. The stroke data was normalised by shifting each point so that the minimum x and y coordinates were at 0, and scaling the points to ensure all (x, y) coordinates fell within the range of 0 to 1, inclusive. The timestamps for all stroke data were also adjusted so that the first stroke point for each stroke started at timestamp 0 seconds.

This normalisation process is essential for the efficient convergence of the gradient descent optimisation during the training phase. It also ensures consistency in the input data, which is particularly important when dealing with a variety of handwriting styles and sizes present in the dataset. This step in data preprocessing plays a crucial role in ensuring the model can learn effectively. It's important to note that some of the extracted data had to be discarded because certain XML files contained no stroke data, likely due to occasional errors in the recording of pen nib tracking during the data collection process of the IAM dataset.

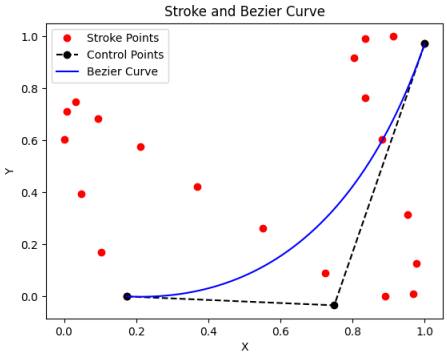
The extraction of this data required careful handling to ensure the correct association between the line label, extracted from one of the text files, and the corresponding stroke data parsed from a specific XML file. This was crucial to maintain the integrity of the data and to ensure that the subsequent model training was based on accurate and consistent data.

5.2.1 Initial Bézier Curve Features

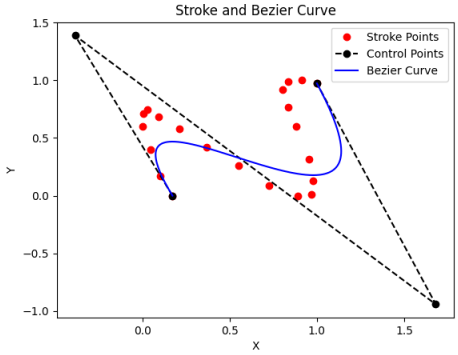
Once the data was extracted and normalised, a Bézier curve of degree N (any integer greater than or equal to 2), was fitted to each stroke in each set of stroke data. For a visualisation of different Bézier degree curves, see Figure 5.4.



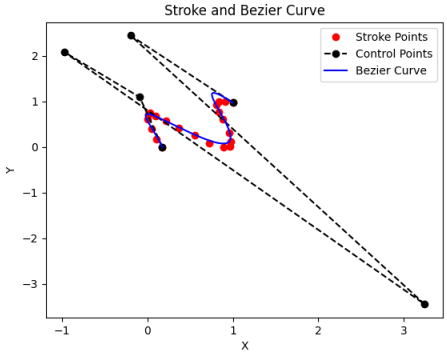
(a) A Bézier curve of *degree 1* fitted to a set of (x, y) stroke points.



(b) A Bézier curve of *degree 2* fitted to a set of (x, y) stroke points.



(c) A Bézier curve of *degree 3* fitted to a set of (x, y) stroke points.



(d) A Bézier curve of *degree 5* fitted to a set of (x, y) stroke points.

Figure 5.4: Bézier curves of degree 1, 2, 3, and 5 fitted to a set of (x, y) stroke points.

Two types of feature sets were investigated to describe the features of the Bézier curve. The first feature set includes the following features:

- **Total Length:** This feature represents the total length of the stroke, calculated as the sum of the Euclidean distances between consecutive points in the stroke.

- **Directness:** Directness is a measure of how straight the stroke is. It is calculated as the direct distance (Euclidean distance) between the first and last points of the stroke divided by the total length of the stroke. A value close to 1 indicates a straight stroke, while a value significantly less than 1 indicates a more curved or complex stroke.

- **Total Curvature:** This feature is the sum of the absolute differences in direction between consecutive segments of the stroke. It provides a measure of how much the stroke direction changes throughout its course. The curvature is calculated using the interpolated stroke data. To find the change in direction between two points on the curve, you can subtract the direction (angle) at the first point from the direction at the second point [28]. This gives you the change in direction as you move from the first point to the second point.

  To demonstrate this, let's consider a simple example where we have a stroke with three points: A, B, and C. The coordinates of these points are as follows:

  - A: $(0, 0)$

  - B: $(1, 1)$

– C: $(2, 0)$

First, we need to calculate the direction of the segments AB and BC. The direction (angle) is calculated using the arctan2 function:

– Direction AB $= \arctan 2(B_y - A_y, B_x - A_x) = \arctan 2(1-0, 1-0) = \arctan 2(1, 1) = \frac{\pi}{4}$

– Direction BC $= \arctan 2(C_y - B_y, C_x - B_x) = \arctan 2(0-1, 2-1) = \arctan 2(-1, 1) = -\frac{\pi}{4}$

Next, we find the change in direction between the two segments by subtracting the direction at the first point from the direction at the second point:

$$\text{Change in direction} = \text{Direction BC} - \text{Direction AB} = -\frac{\pi}{4} - \frac{\pi}{4} = -\frac{\pi}{2} \tag{5.1}$$

We take the absolute value of the change in direction to get the curvature at point B:

$$\text{Curvature at B} = |\text{Change in direction}| = \left|-\frac{\pi}{2}\right| = \frac{\pi}{2} \tag{5.2}$$

Since we only have one change in direction (at point B), the total curvature of the stroke is simply the curvature at B, which is $\frac{\pi}{2}$.

- **Average Direction Values:** This feature is the average sine and cosine of the direction of the stroke. The direction of a segment is defined as the angle it makes with the x-axis. This angle is calculated for each segment of the interpolated stroke data, and the sine and cosine of these angles are averaged to give the average sin and cos direction values [28]. This value provides a measure of the average orientation of the stroke, capturing whether the stroke tends to move more horizontally or vertically.

  To build upon the previous example, to calculate the average direction values, we first compute the direction of each segment AB and BC as we did before. We then compute the sine and cosine of these direction values:

  – Sine of Direction AB $= \sin(\frac{\pi}{4}) = \frac{\sqrt{2}}{2}$

  – Cosine of Direction AB $= \cos(\frac{\pi}{4}) = \frac{\sqrt{2}}{2}$

  – Sine of Direction BC $= \sin(-\frac{\pi}{4}) = -\frac{\sqrt{2}}{2}$

  – Cosine of Direction BC $= \cos(-\frac{\pi}{4}) = \frac{\sqrt{2}}{2}$

  The average sin/direction and cos/direction values are then computed by averaging the sine and cosine values respectively:

$$\text{Average sin/direction} = \frac{\sin(\frac{\pi}{4}) + \sin(-\frac{\pi}{4})}{2} = 0 \tag{5.3}$$

$$\text{Average cos/direction} = \frac{\cos(\frac{\pi}{4}) + \cos(-\frac{\pi}{4})}{2} = \frac{\sqrt{2}}{2} \tag{5.4}$$

- **Average Curvature Values:** To calculate the average curvature values, we first compute the curvature at each point along the stroke. In this case, we only have one point of curvature

(at point B), which we computed before as $\frac{\pi}{2}$. We then compute the sine and cosine of this curvature value:

- Sine of Curvature at B = $\sin(\frac{\pi}{2}) = 1$
- Cosine of Curvature at B = $\cos(\frac{\pi}{2}) = 0$

The average sin/curvature and cos/curvature values are then computed by averaging the sine and cosine values respectively. In this case, since we only have one point of curvature, the average sin/curvature and cos/curvature values are simply the sine and cosine of the curvature at B:

$$\text{Average sin/curvature} = \sin\left(\frac{\pi}{2}\right) = 1 \tag{5.5}$$

$$\text{Average cos/curvature} = \cos\left(\frac{\pi}{2}\right) = 0 \tag{5.6}$$

In a more complex stroke with multiple points of curvature, we would compute the sine and cosine of the curvature at each point, and then average these values to get the average sin/curvature and cos/curvature values. These average curvature values provide a measure of the average curvature of the stroke, capturing how much the stroke tends to curve or change direction.

- **End Point Difference:** This feature is a 2D vector representing the difference in the x and y coordinates between the first and last points of the stroke [2]. It provides a measure of the overall change in position from the start to the end of the stroke.

- **Control Point Distributions:** These features represent the normalised distances from the first control point to each of the other control points of the Bézier curve [2]. They provide a measure of how the stroke is distributed along the curve.

- **Angles:** These features are the angles between the line joining the first control point and each of the other control points, and the x-axis. They provide a measure of the orientation of the stroke segments defined by the control points.

- **Time Coefficients:** These features represent the relative times at which each control point is reached [2], normalised to the range $[0, 1]$. They provide a measure of the timing and speed of the stroke.

- **Pen-up Flag:** This binary feature indicates whether the pen was lifted (pen-up) or remained in contact with the writing surface (pen-down) after the completion of this stroke [2]. It provides information about the segmentation of the handwriting into distinct strokes. In this current approach, the pen is always lifted at the end of each stroke, rendering this feature less informative for the model's learning process. However, it is included for potential future investigations. For instance, we may explore a different approach in the future where multiple Bézier curves are fitted to various segments within a single stroke, rather than fitting one Bézier curve to each entire stroke. These segmented curves could then be combined to form a more complex representation of the stroke. This approach is discussed further in Section 3.2.

These features provide a comprehensive description of the Bézier curve fitted to the stroke data, capturing various aspects of the stroke such as its length, curvature, and direction. This should allow the model to learn effectively from the data, enabling it to capture the nuances of different handwriting styles.

It's important to note that these features are calculated from the Bézier curve fitted to the stroke data, rather than the raw stroke data itself. This means that they capture the properties of the stroke as represented by the Bézier curve, which may be a smoothed or simplified version of the original stroke. This can help to reduce the impact of noise or small variations in the stroke data, making the features more robust and reliable. Furthermore, the mathematical process of calculating the tangent vectors at each point along the curve and determining the change in direction at each point along the curve, provides a precise measure of the curvature of the curve [28], which is a crucial aspect of the stroke's characteristics.

5.2.2 Extended Bézier Curve Features

In addition to the initial set of features, an extended feature set was also investigated that includes additional characteristics derived from the velocity and acceleration of the stroke. These features are computed from the first and second derivatives of the Bézier curve (as discussed in Subsection 2.3.3), which represent the velocity and acceleration of the stroke, respectively.

- **Velocity:** The first derivative of the Bézier curve is computed, which represents the velocity of the stroke. The velocity is a vector quantity that has both magnitude (speed) and direction. From this velocity, the following features are computed:

  - *Mean Velocity:* This is the average speed of the stroke over its course. It is computed as the mean of the magnitudes of the velocity vectors.

  - *Standard Deviation of Velocity:* This is a measure of the variability in the speed of the stroke. It is computed as the standard deviation of the magnitudes of the velocity vectors.

  - *Mean Velocity Angle:* This is the average direction of the stroke. It is computed as the circular mean of the angles of the velocity vectors. The circular mean, also known as the mean angle, is a method used to calculate the average of a set of angles. This method is particularly useful because it takes into account the cyclical nature of angles. Angles are cyclical because they repeat every 360 degrees (or 2π radians). The circular mean treats the angles as points on a unit circle (a circle with a radius of 1), rather than as numbers on a number line. Each angle is converted to a point on the unit circle using the sine and cosine functions. The circular mean is then calculated as the angle of the average point on the unit circle, providing a more representative average direction of the stroke [44].

  - *Velocity Angular Dispersion:* This is a measure of the variability in the direction of the stroke. It is computed as the circular standard deviation of the angles of the velocity vectors. The circular standard deviation is a measure of the dispersion of a set of angles, providing an indication of how spread out the stroke directions are from the mean direction [44].

  To demonstrate an example, let's consider a Bézier curve of degree 2 defined by three control points: A, B, and C. The coordinates of these points are as follows:

  - A: $(0, 0)$

  - B: $(1, 1)$

  - C: $(2, 0)$

  The Bézier curve defined by these points is given by the equation:

$$B(t) = (1-t)^2 A + 2t(1-t)B + t^2 C \tag{5.7}$$

where t ranges from 0 to 1.

The first derivative of this Bézier curve, which represents the velocity of the stroke, is given by:

$$B'(t) = 2(1-t)(B-A) + 2t(C-B) \tag{5.8}$$

We can compute the velocity vectors at various points along the curve by substituting different values of t into this equation. For example, at the start of the stroke ($t = 0$), the velocity vector is:

$$B'(0) = 2(1-0)(B-A) = 2B - 2A = 2(1,1) - 2(0,0) = (2,2) \tag{5.9}$$

At the midpoint of the stroke ($t = 0.5$), the velocity vector is:

$$B'(0.5) = 2(1-0.5)(B-A) + 2 \cdot 0.5(C-B) = B - A + C - B = C - A = (2,0) - (0,0) = (2,0) \tag{5.10}$$

At the end of the stroke ($t = 1$), the velocity vector is:

$$B'(1) = 2(1-1)(B-A) + 2 \cdot 1(C-B) = 2C - 2B = 2(2,0) - 2(1,1) = (2,-2) \tag{5.11}$$

The magnitudes (speeds) of these velocity vectors are:

- Speed at $t = 0$: $\sqrt{(2)^2 + (2)^2} = 2\sqrt{2}$
- Speed at $t = 0.5$: $\sqrt{(2)^2 + (0)^2} = 2$
- Speed at $t = 1$: $\sqrt{(2)^2 + (-2)^2} = 2\sqrt{2}$

The Mean Velocity is the average of these speeds:

$$\text{Mean Velocity} = \frac{2\sqrt{2} + 2 + 2\sqrt{2}}{3} = \frac{4\sqrt{2} + 2}{3} \tag{5.12}$$

The angles of the velocity vectors are:

- Angle at $t = 0$: $\arctan 2(2,2) = \frac{\pi}{4}$
- Angle at $t = 0.5$: $\arctan 2(0,2) = 0$
- Angle at $t = 1$: $\arctan 2(-2,2) = -\frac{\pi}{4}$

The Mean Velocity Angle is the circular mean of these angles:

$$\text{Mean Velocity Angle} = \text{circmean}\left(\frac{\pi}{4}, 0, -\frac{\pi}{4}\right) = 0 \tag{5.13}$$

The Velocity Angular Dispersion, calculated approximately, is:

$$\text{Velocity Angular Dispersion} = \text{circstd}\left(\frac{\pi}{4}, 0, -\frac{\pi}{4}\right) \approx 0.66 \text{ radians} \tag{5.14}$$

This shows that the stroke has a high average speed of $\frac{4\sqrt{2}+2}{3}$, and it changes direction by an average of 0 radians from one segment to the next. The Velocity Angular Dispersion of approximately 0.66 radians indicates that there is a moderate variability in the direction of the stroke.

- **Acceleration:** The second derivative of the Bézier curve is computed, which represents the acceleration of the stroke. The acceleration is also a vector quantity that has both magnitude (rate of change of speed) and direction (rate of change of direction). From this acceleration, the following features are computed:
  - *Mean Acceleration:* This is the average rate of change of speed of the stroke over its course. It is computed as the mean of the magnitudes of the acceleration vectors.
  - *Standard Deviation of Acceleration:* This is a measure of the variability in the rate of change of speed of the stroke. It is computed as the standard deviation of the magnitudes of the acceleration vectors.
  - *Mean Acceleration Angle:* This is the average rate of change of direction of the stroke. It is computed as the circular mean of the angles of the acceleration vectors. Similar to the mean velocity angle, the circular mean here provides an average direction of change in velocity that is more representative of the actual changes in direction [44].
  - *Acceleration Angular Dispersion:* This is a measure of the variability in the rate of change of direction of the stroke. It is computed as the circular standard deviation of the angles of the acceleration vectors. The circular standard deviation, like the circular mean, is a method that takes into account the cyclical nature of angles. It is a measure of the dispersion of a set of angles around the circular mean, providing an indication of how spread out the changes in stroke direction are from the mean change in direction [44].

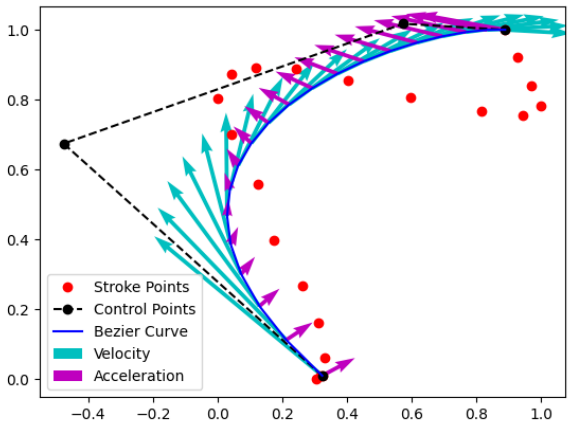


Figure 5.5: A Bézier curve of degree 3 fitted to a set of (x, y) stroke points, also displaying the velocity and acceleration at curve points (with arrows representing the directions and lengths being the magnitudes).

These features provide a description of the dynamics of the stroke, capturing not only its shape and orientation, but also its speed, changes in speed, direction, and changes in direction. For a visualisation of velocity and acceleration being measured at Bézier curve points, see Figure 5.5. We will investigate the performance of both of these feature sets.

5.3 Final Model Training & Evaluation

The first step in the training and evaluation process was to partition the preprocessed data. It was divided into training, validation, and test sets, following a 70-10-10-10 split. This division created two validation sets, which allowed us to explore the impact of using cross-validation versus a single validation set. If we opted for a single validation set, we would combine the first validation set with the original training set, resulting in an 80-10-10 split for training, validation, and test sets, respectively.

Label	**Greedy Decoding Prediction**	**CER (Greedy)**	**WER (Greedy)**	**Beam Decoding Prediction**	**CER (Beam)**	**WER (Beam)**
Then he lifted his head and asked	Then he elifted his hea a and ked	15.15%	57.14%	Then he elifted his hea a and sked	12.12%	57.14%
But these have been successfully	But these whave been Mucersfuly	12.5%	40%	But these have been Muccersfully	6.25%	20%

Table 5.1: Examples of decoding predictions during model training.

With the data sets prepared, I proceeded to train various types of recurrent neural network models, including Recurrent Neural Network (RNN), Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU) models. The training process utilised the Connectionist Temporal Classification (CTC) loss function and the Adam optimiser. To see some examples of decoded predictions during model training, see Table 5.1.

During the training phase, different hyperparameters were experimented with to optimise the performance of the models. These hyperparameters included batch size, hidden size, the number of recurrent layers, the use of bidirectional models, learning rate, and beam size for beam search decoding. The impact of using different Bézier curve feature sets as input data to the models after finding the optimal hyperparameters were also evaluated.

To assess the performance of the models during validation and testing, the Character Error Rate (CER) and Word Error Rate (WER) was measured for both greedy and beam search decodings. The following section will discuss some of these investigations.

5.3.1 Investigations

Hyperparameter	**Value**
Beam Size	5
Bézier Curve Degree	5
Bézier Feature Set (refer to Subsections 5.2.1, 5.2.2)	Extended
Bidirectional RNNs Used	True
Cross Validation (if False, Single Validation set used)	False
Dropout Rate	0.3
Epochs	200
Hidden Size	256
Batch Size	16
Learning Rate	3×10^{-4}
Number of Recurrent Layers	3
Dropout Patience	10

Table 5.2: Default hyperparameters used for most investigations.

An exploration began by comparing the performance of a recurrent neural network model trained

using two distinct feature sets: the initial and extended Bézier curve feature sets, as discussed in Subsections 5.2.1 and 5.2.2. For most of these investigations, an LSTM model and the hyperparameters outlined in Table 5.2 were used, except when the focus was on a specific hyperparameter, in which case the value was adjusted accordingly.

Feature Set	**Avg. Beam CER**	**Avg. Greedy CER**	**Avg. Beam WER**	**Avg. Greedy WER**
Initial Set	52.12%	52.53%	100.1%	100.1%
Extended Set	49.84%	50.23%	97.78%	98.05%

Table 5.3: Test set average CERs & WERs of greedy and beam search decodings of an LSTM model trained using initial and extended Bézier curve **feature sets**.

Table 5.3 reveals that the LSTM model trained using the extended Bézier curve feature set outperformed the one trained with the initial set. The model trained with the extended feature set achieved both a beam search decoding CER and WER approximately 2.3% lower than the model trained with the initial feature set. This suggests that the additional features in the extended set, which capture the velocity and acceleration characteristics of the strokes, provide valuable information that enhances the model's ability to understand the data and make more accurate predictions.

Validation	**Avg. Beam CER**	**Avg. Greedy CER**	**Avg. Beam WER**	**Avg. Greedy WER**
Single Validation Set	52.13%	52.48%	99.69%	99.54%
Cross Validation	54.05%	56.41%	100.9%	98.86%

Table 5.4: Test set average CERs & WERs of greedy and beam search decodings of an LSTM model trained using a **single validation set** and **cross validation**.

Next, the performance of an LSTM model trained using cross-validation versus a single validation set was examined. As shown in Table 5.4, the LSTM model trained using a single validation set (which results in a larger training set) outperformed the same model trained using cross-validation. This was true even when the same hyperparameters were used. The model trained with a single validation set achieved a beam search decoding CER approximately 1.9% lower, and a WER approximately 1.2% lower than the model trained using cross-validation. This suggests that using a single validation set, which results in a larger training set, leads to better model performance. This is likely because the larger training set provides more examples for the model to learn from, leading to better generalisation and prediction accuracy.

Batch Size	**Avg. Beam CER**	**Avg. Greedy CER**	**Avg. Beam WER**	**Avg. Greedy WER**
16	48.26%	48.68%	94.89%	94.51%
32	50.54%	50.72%	96.55%	96.37%
64	51.53%	51.94%	97.78%	97.86%
128	52.01%	52.95%	99.76%	98.99%

Table 5.5: Test set average CERs & WERs of greedy and beam search decodings of an LSTM model trained using different **batch sizes**.

Moving on to batch size, as shown in Table 5.5, using a batch size of 16 yielded the best results. The LSTM model trained with a batch size of 16 achieved a beam search decoding CER and WER of approximately 48.3% and 94.9% respectively. These results suggest that a smaller batch size may be more effective for this specific task, potentially due to the increased frequency of weight updates during training.

In terms of hidden sizes, as shown in Table 5.6, an LSTM model trained with hidden sizes of 256 or 512 yielded the best results. The model trained with a hidden size of 256 achieved a beam search decoding CER and WER of approximately 49.6% and 97.9% respectively, and the model trained

Hidden Size	Avg. Beam CER	Avg. Greedy CER	Avg. Beam WER	Avg. Greedy WER
64	50.2%	51.48%	100.8%	98.13%
128	48.92%	49.85%	99.3%	97.86%
256	49.56%	49.87%	97.92%	97.6%
512	49.36%	49.58%	98.11%	98.21%

Table 5.6: Test set average CERs & WERs of greedy and beam search decodings of an LSTM model trained using different **hidden sizes**.

with a hidden size of 512 achieved a beam search decoding CER and WER of approximately 49.4% and 98.1% respectively. These results suggest that a larger hidden size can improve the performance of the model, likely by allowing it to capture more complex patterns in the Bézier curve data. A hidden size of 256 will be used, as a model with this hidden size is less complex and therefore, trains faster compared to a model with a hidden size of 512.

Number of Layers	Avg. Beam CER	Avg. Greedy CER	Avg. Beam WER	Avg. Greedy WER
1	63.37%	63.62%	114.3%	113.3%
2	52.71%	53.2%	101.3%	101.1%
3	48.51%	48.78%	97.53%	97.44%
4	45.26%	45.83%	91.45%	91.92%
5	71.58%	72.01%	107.7%	106%

Table 5.7: Test set average CERs & WERs of greedy and beam search decodings of an LSTM model trained using different **number of recurrent layers**.

In addition, Table 5.7 indicates that the LSTM model trained with 4 recurrent layers yields the best results. Specifically, this configuration resulted in a beam search decoding CER and WER of approximately 45.3% and 91.5% respectively. This suggests that increasing the number of layers can enhance the model's performance, but only to a certain extent. For instance, a model trained with 5 layers did not perform as well, possibly due to increased complexity and associated training difficulties.

Bidirectional	Avg. Beam CER	Avg. Greedy CER	Avg. Beam WER	Avg. Greedy WER
False	48.71%	49.24%	101.9%	102.1%
True	48.1%	48.57%	95.74%	95.71%

Table 5.8: Test set average CERs & WERs of greedy and beam search decodings of an LSTM model trained with and without **bidirectional** layers.

Furthermore, Table 5.8 reveals that the LSTM model trained with bidirectional layers outperformed its unidirectional counterpart. In particular, the bidirectional model achieved a beam search decoding CER that was approximately 0.6% lower. While this might not seem like a substantial difference, it's still a noteworthy improvement. Also, the WER was approximately 6.2% lower for the model with bidirectional layers compared to the model without them. This is a significant improvement, showing that using bidirectional layers can greatly enhance the model's ability to accurately recognise handwriting.

Also, Table 5.9 shows that dropout rates of 0.3 and 0.4 produced the best results. Specifically, using a dropout rate of 0.3 leads to a beam search decoding CER and WER of approximately 42.3% and 88.1% respectively, and a dropout rate of 0.4 leads to a beam search beam search decoding CER and WER of approximately 42.1% and 88.6% respectively. However, since a dropout rate of 0.3 trains the model faster than a rate of 0.4, we chose to use 0.3.

Moreover, Table 5.10 indicates that the LSTM model trained using Bézier curve degree 4 input data yields the best results. The model trained with Bézier curve degree 4 input data achieved a

Dropout Rate	Avg. Beam CER	Avg. Greedy CER	Avg. Beam WER	Avg. Greedy WER
0.2	43.4%	43.76%	90.89%	90.61%
0.3	42.33%	42.77%	88.12%	87.81%
0.4	42.1%	42.56%	88.63%	88.08%
0.5	45.53%	46.21%	94.3%	93.48%

Table 5.9: Test set average CERs & WERs of greedy and beam search decodings of an LSTM model trained using different **dropout rates**.

Bézier Curve Degree	Avg. Beam CER	Avg. Greedy CER	Avg. Beam WER	Avg. Greedy WER
2	44.9%	45.15%	94.28%	94.39%
3	43.54%	44.07%	91.75%	90.99%
4	42.33%	42.77%	88.12%	87.81%
5	43.75%	44.1%	91.01%	91.04%

Table 5.10: Test set average CERs & WERs of greedy and beam search decodings of an LSTM model trained using different **Bézier curve degrees**.

beam search decoding CER and WER of approximately 42.3% and 88.1% respectively. This shows that increasing the Bézier curve degree improves the performance up until a certain degree. For example, using a Bézier curve degree of 5 as input data gave worse results. Beyond this point, the model likely starts to overfit.

Learning Rate	Avg. Beam CER	Avg. Greedy CER	Avg. Beam WER	Avg. Greedy WER
3×10^{-4}	48.69%	48.89%	96.91%	96.83%
5×10^{-4}	46.37%	46.74%	94.01%	94.1%
7×10^{-4}	45.48%	45.84%	93.26%	93.07%
1×10^{-3}	47.12%	47.7%	96.56%	96.04%
2×10^{-3}	47.57%	48.18%	96.51%	95.55%

Table 5.11: Test set average CERs & WERs of greedy and beam search decodings of an LSTM model trained using different **learning rates**.

Lastly, Table 5.11 shows that the LSTM model trained using a learning rate of 7×10^{-4} gives the best results. Specifically, this configuration resulted in a beam search decoding CER and WER of approximately 45.5% and 93.3% respectively. It suggests that a learning rate of 7×10^{-4} provides a good balance between learning speed and model accuracy.

Also, Table 5.12 shows that using a beam size of 3, 8, or 30 doesn't change the beam search decoding CER and WER values much. So, we'll use a beam size of 3 as it makes the model train faster compared to using a large beam size, and has the lowest beam decoding CER and WER values.

In summary, these investigations have helped to identify a set of optimal hyperparameters for training the recurrent neural network models, as shown in Table 5.13.

5.3.2 Final Results

Following the investigations, the RNN, LSTM, and GRU models were trained using the optimal hyperparameters specified in Table 5.13. For this final training round, the dropout patience was set to 10 and the number of epochs to 200.

From the data in Table 5.14, it's clear that the LSTM model obtained the best results. It achieved roughly 42.3% and 88.1% average beam search decoding CER and WER respectively on the test set. Although these values are higher than one might prefer, it is important to take into account

Beam Size	Avg. Beam CER	Avg. Beam WER
3	51.14%	97.49%
8	51.43%	98.38%
30	51.15%	97.82%

Table 5.12: Test set average CERs & WERs of beam search decodings using different **beam sizes** after training a model.

Hyperparameter	Optimal Value
Bézier Curve Feature Set	Extended
Bézier Curve Degree	4
Validation Set	Single validation set
Layer Directionality	Bidirectional
Number of Recurrent Layers	4
Batch Size	16
Hidden Size	256
Learning Rate	7×10^{-4}
Dropout Rate	0.3
Beam Size	3

Table 5.13: Optimal hyperparameters for model training based on the investigations.

the considerable size and diversity of the dataset, which can contribute to these higher rates, as illustrated in Figure 5.1.

Moreover, the LSTM model achieved roughly 21% and 12.2% lower average beam search decoding CER and WER respectively, compared to the RNN model. This likely stems from the RNN model's absence of memory components, as discussed in Section 2.2. Furthermore, the LSTM model achieved roughly 7.6% and 7.2% lower average beam search decoding CER and WER, respectively, compared to the GRU model. This is likely because the LSTM model, being more complex and having more parameters, can detect complex patterns more effectively.

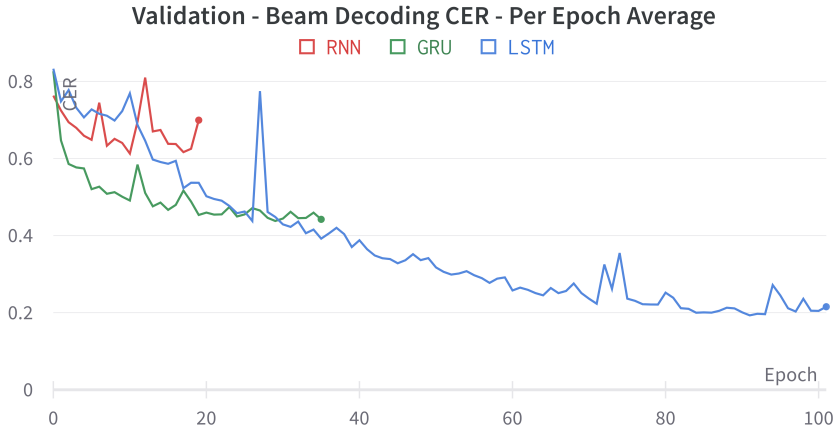


Figure 5.6: Validation beam search decoding CER plots for RNN, LSTM and GRU models.

For a visualisation of how the average validation greedy and beam search decoding CERs decrease over time during training for all models, refer to Figures 5.6 and 5.7.

Similarly, for greedy and beam search decoding WERs, see Figures 5.8 and 5.9.

Also, we can see that both the validation and training loss converge as shown in Figures 5.10 and 5.11, indicating that the models are effectively learning from the training data and generalising well to unseen validation data. The convergence of these losses suggests that the models have

Model	Avg. Beam CER	Avg. Greedy CER	Avg. Beam WER	Avg. Greedy WER
RNN	63.36%	66.28%	100.3%	99.23%
LSTM	42.33%	42.66%	88.12%	87.81%
GRU	49.94%	50.92%	95.31%	94.25%

Table 5.14: Final test set average CERs & WERs of greedy and beam search decodings for the RNN, LSTM and GRU models trained on the IAM-OnDB dataset.

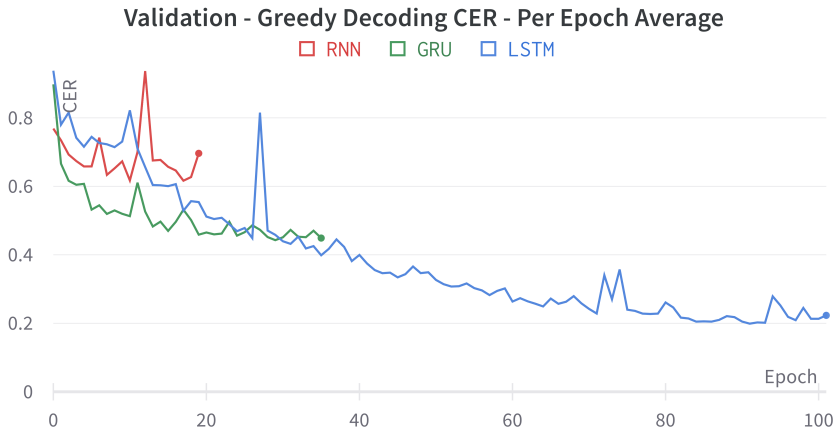


Figure 5.7: Validation greedy decoding CER plots for RNN, LSTM and GRU models.

reached a point where further training would not significantly improve performance, thus avoiding potential overfitting.

Additionally, Figure 5.12 shows that the gradients of the fully connected layer for the LSTM model become more stable and less sparse as the number of epochs increase during training, which is an indicator of convergence.

On average, using an NVIDIA A30 Tensor Core GPU from the Imperial Department of Computing's Slurm GPU cluster, we can extract all data and train each model in approximately 30-40 minutes. This speed helps us update the models with new data often. It also lets us make fast changes to improve them. This quick turnaround could be very important for future research and work in this area.

5.4 Web Application

The web application provides an interactive platform for users to input their handwriting and receive immediate recognition results. The frontend was developed using React and the backend using Flask. For a visualisation of the main UI on the frontend, see Figure 5.13.

The application is designed to capture and process stroke points from the user's input, and then use these points to generate a prediction using a backend model. The model parameters can be adjusted by the user, allowing for a personalised and interactive experience. Additionally, the application provides a visual display of the Bézier curves for each stroke that the user writes, as shown in Figure 5.14. This feature offers users a unique insight into how their handwriting is processed.

The application tracks the user's mouse movements to capture the dynamic nature of handwriting. Specifically, when the user presses the left mouse button, a new stroke begins. As the user moves the mouse while keeping the button pressed, the application draws a line from the position of the mouse in the previous frame to its position in the current frame. This process occurs at a

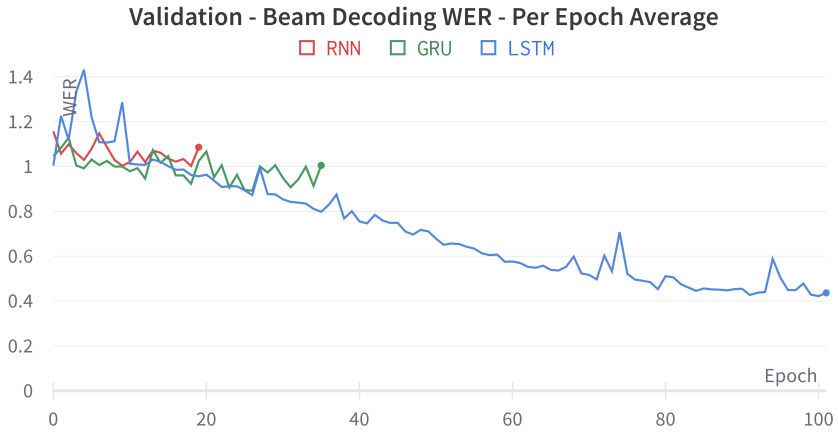


Figure 5.8: Validation beam search decoding WER plots for RNN, LSTM and GRU models.

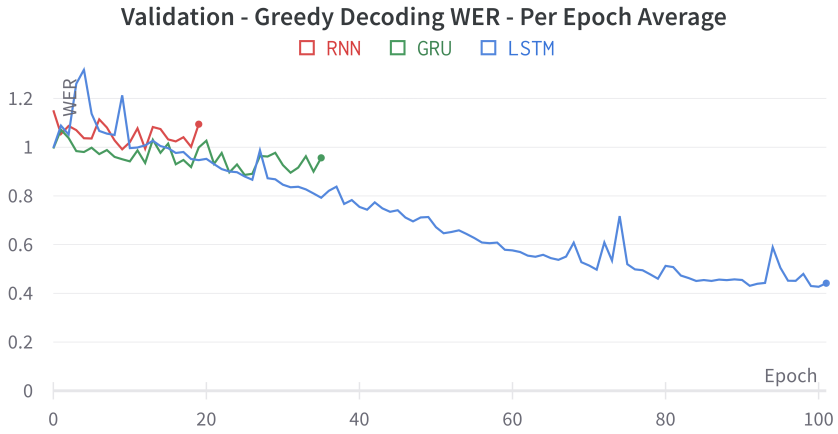


Figure 5.9: Validation greedy decoding WER plots for RNN, LSTM and GRU models.

high frequency, creating the impression of smooth, non-linear strokes that mimic the natural flow of handwriting. The application continues to track the mouse movements and add to the current stroke as long as the button remains pressed. When the user releases the mouse button, the current stroke is considered complete.

This implementation enables the application to capture the start and end points of each stroke, as well as the path followed by the user's hand.

5.4.1 Challenges

During the testing phase, I encountered an issue where the model's predictions were inverted. For instance, when the input was the character 'm', the model predicted it as 'w'. This issue arose because the frontend used a coordinate system where the origin was at the top left corner of the canvas. To rectify this, I implemented a preprocessing step on the backend that flips the y-coordinates of the received stroke points. This ensured that the model's predictions were correctly oriented.

Additionally, I observed that the model's predictions were less accurate when each stroke contained too many points. To enhance the model's performance, I introduced a downsampling process. I added a slider on the frontend that allows the user to adjust the number of points recorded per second. By reducing the number of points, the model's predictions became more accurate. This is because the downsampled data more closely resembled the samples in the IAM-OnDB dataset,

Figure 5.10: Training loss plots for the RNN, LSTM and GRU models trained using the IAM-OnDB dataset.

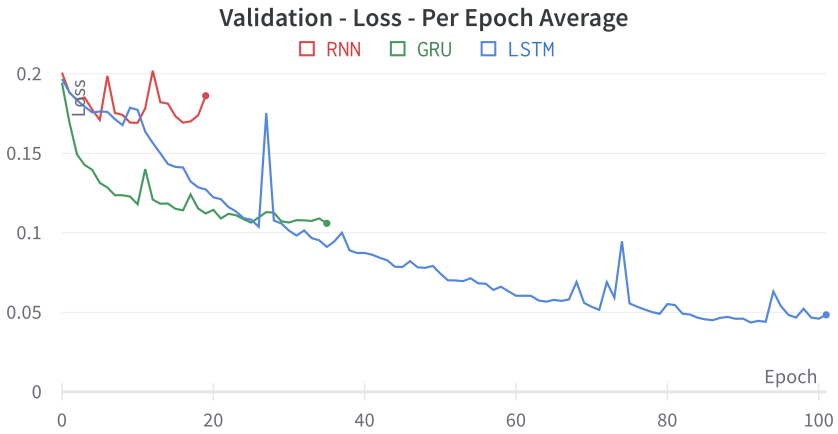


Figure 5.11: Validation loss plots for the RNN, LSTM and GRU models trained using the IAM-OnDB dataset.

which the model was trained on. To see a visualisation of the impact of varying downsampling rates, refer to Figure 5.15.

Furthermore, to optimise the response time from user input to model prediction, I added a Least Recently Used (LRU) cache to the backend. This cache stores all the loaded models during the initial startup. As a result, the system doesn't need to load the models again for each request received from the frontend. This caching mechanism significantly speeds up the response time, providing a better user experience.

5.4.2 User Experience (UX) Evaluation

In summary, the web application provides an interactive platform where users can handwrite and immediately receive recognition results. Real-time visualisations of the Bézier curves fitted to each stroke of the user's handwriting offer an intuitive understanding of how the backend models process strokes and make predictions. Furthermore, the application offers users the flexibility to modify various parameters and test out different models. A key feature is the instantaneous return of handwriting predictions, ensuring a quick response time for the user.

It's important to note that the models encounter difficulties when attempting to predict labels for cursive handwriting, in which characters are typically interconnected, as shown in Figure 5.16.

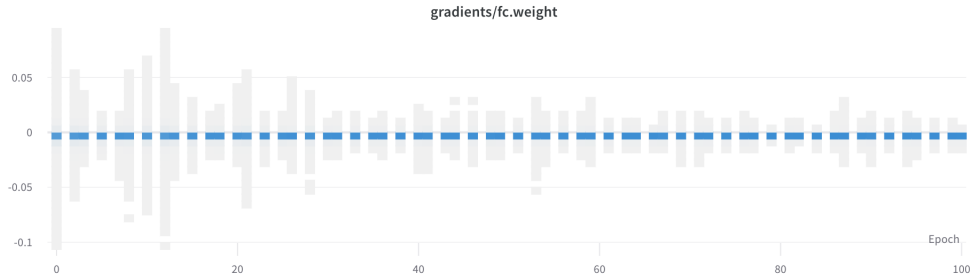


Figure 5.12: Distribution of gradients of the fully connected layer in the final LSTM model trained on the IAM-OnDB dataset.

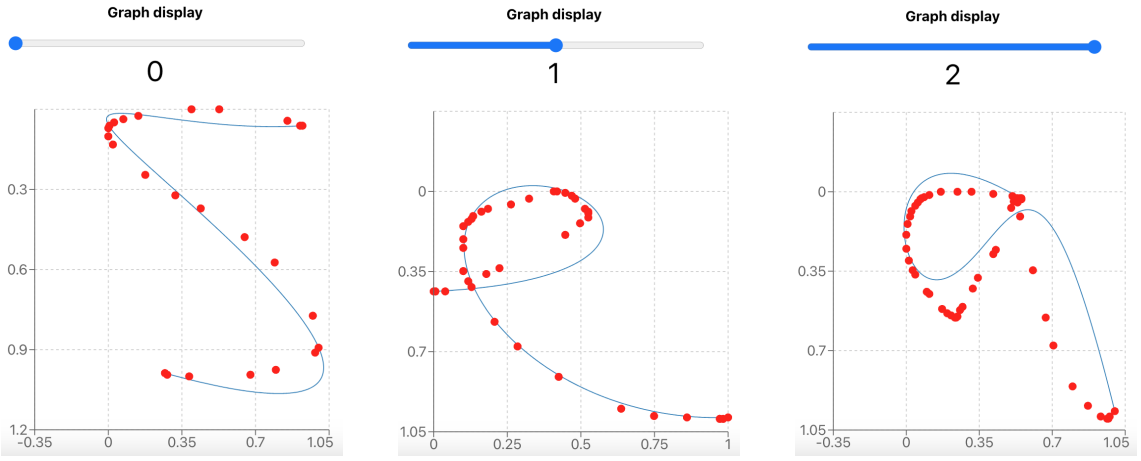
Figure 5.13: Main UI of the IAM-OnDB dataset handwriting recognition web application.

This difficulty could potentially be due to two reasons: firstly, the IAM-OnDB dataset may not provide enough examples of cursive writing that cover a broad range of word types. Secondly, the models are trained to predict a character for each Bézier curve. Given that a single Bézier curve is fitted to each complete stroke without any splitting, this approach proves problematic when cursive handwriting is encountered, where often an entire word is formed within a single stroke.

Moreover, the models have difficulty accurately predicting characters that are typically formed with two strokes (such as the characters ‘t’, ‘i’, and ‘f’), as illustrated in Figure 5.17. As a probable solution, employing a Bézier curve splitting approach as discussed in Carbune et al. [2], could enhance the models’ capabilities to handle and accurately predict labels for cursive handwriting and characters typically written with two strokes.

Also, characters with similar appearance and Bézier curve shapes, like ‘2’ and ‘Z’, can sometimes be misidentified as identical. For instance, as shown in Figure 5.18, the model predicts the label ‘C’ for some ‘e’ characters in a handwritten word. This is especially true when characters are poorly written or too small, as it becomes challenging for the models to accurately recognise specific stroke curvatures and bends. These stroke features are vital for accurate recognition.

Despite these challenges, the system performs well with larger handwriting, similar to what might be seen in a whiteboard lecture environment.

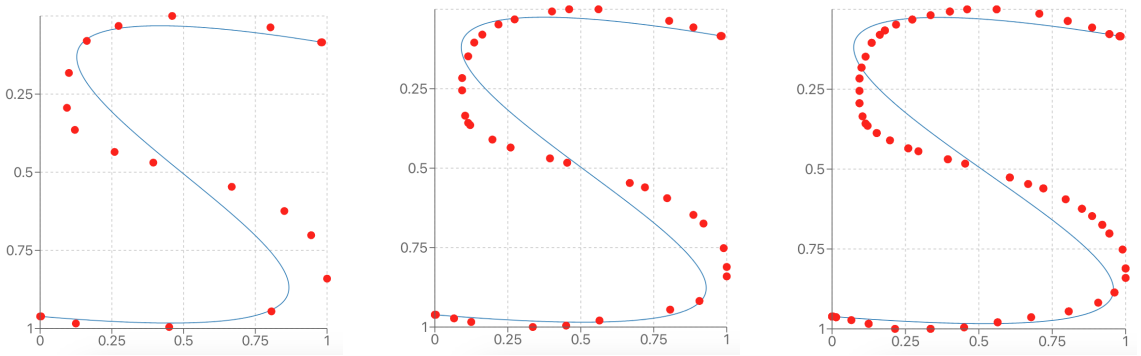


(a) Bézier curve fitted to character 'S' stroke.

(b) Bézier curve fitted to character 'e' stroke.

(c) Bézier curve fitted to character 'a' stroke.

Figure 5.14: Web UI visualisation of Bézier curves of degree 4 fitted to handwritten word "Sea".



(a) Bézier curve of degree 4 fitted to handwritten character 'S' using a *25 points per second* rate.

(b) Bézier curve of degree 4 fitted to handwritten character 'S' using a *50 points per second* rate.

(c) Bézier curve of degree 4 fitted to handwritten character 'S' using a *100 points per second* rate.

Figure 5.15: Web UI visualisation of Bézier curves of degree 4 fitted to handwritten character 'S' using different points per second downsampling rates.

Additionally, the user-adjustable downsampling rate (points per second), depends on the user's writing speed. For instance, if a character is written quickly, the points per second rate should be reduced, while for slower writing, this rate should be increased. This helps the models provide the most accurate predictions and ensures the handwriting appears as close as possible to the data the models were trained on.

All of this code was developed from scratch and open-sourced (unlike previous related studies by some companies), which will help to drive further innovation in the field of handwriting recognition using Bézier curves.

Figure 5.16: Web UI displaying a beam search (bottom left in image) and greedy (bottom right in image) decoding prediction, '**h**', for handwritten word "**hello**".

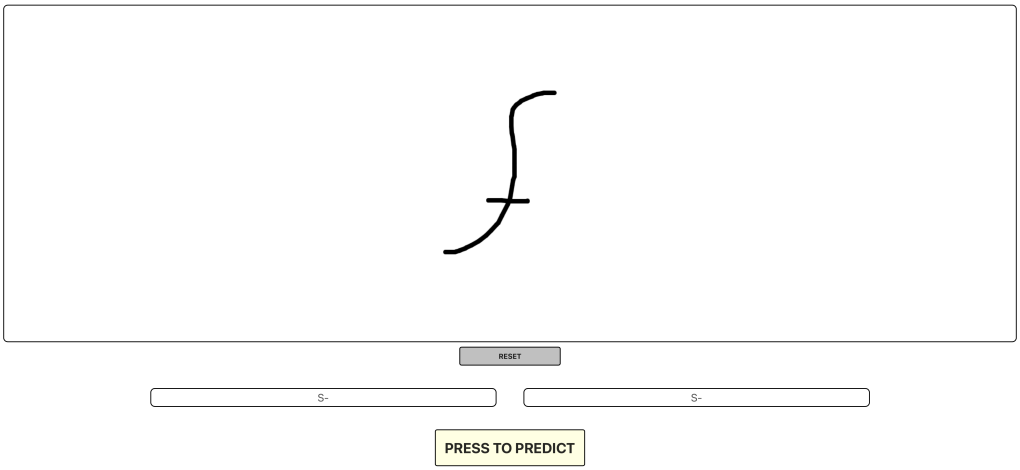


Figure 5.17: Web UI displaying a beam search (bottom left in image) and greedy (bottom right in image) decoding prediction, "**S-**", for handwritten character '**f**'.

Figure 5.18: Web UI displaying a beam search (bottom left in image) and greedy (bottom right in image) decoding prediction, "**SomeWhCrC**", for handwritten word "**somewhere**".

Chapter 6

Conclusion

In this research, a variety of models were developed and analysed for handwriting recognition, utilising two distinct datasets.

The ISGL dataset was used to recognise characters, which highlighted the challenges of the 'vanishing gradient' problem, particularly during the processing of long (x, y) stroke point sequences. This led to a low accuracy of approximately 1.05% across all models. Despite this, the research provided a crucial starting point for more advanced studies, highlighting the need for a larger dataset and refined feature extraction techniques.

For the next phase, the IAM-OnDB dataset was used to create models for handwriting recognition, leveraging the properties of Bézier curves. Of the models tested, the LSTM model outperformed the RNN and GRU models. The LSTM model achieved an average of 42.3% and 88.1% for beam search decoding Character Error Rate (CER) and Word Error Rate (WER), respectively, on the test set. Despite these figures being higher than ideal, it's important to take into account the considerable size and diversity of the dataset, which can contribute to these higher rates, as illustrated in Figure 5.1. The LSTM model demonstrated its ability to handle intricate patterns and diverse data effectively.

A web application was also developed to allow users to interact with the handwriting recognition models. This application allows for adjustment of model parameters and visualisation of the Bézier curves for each stroke written, offering an in-depth look into the handwriting recognition process. The models had some difficulties accurately predicting cursive handwriting and characters usually formed with two strokes, such as 't', 'i', and 'f' - using a Bézier curve splitting approach may potentially enhance the capability of the models. Instead, the application is more adept at recognising larger handwriting, and the user-adjustable downsampling rate ensures lower error rates relative to the user's writing speed.

Finally, the codebase for this project, built from scratch and open-sourced, stands ready to promote further research in the domain of handwriting recognition using Bézier curves.

6.1 Future Work

Looking ahead, the progress made offers opportunities for future work. A potential next step could involve creating a backend API that would allow quick handwriting predictions using (x, y) stroke points and timestamps supplied by users. There's also potential for using computer vision to capture stroke points and timestamps from real-life writing scenarios, like writing on a whiteboard, broadening the system's practical applications.

Moreover, an improved downsampling rate algorithm could be devised to take into account the writing speed and the number of captured stroke points. This would automatically adjust the rate

for the user, reducing the need for manual rate changes. Additionally, it would be useful to adopt a dynamic time window that continuously sends all stroke points written by the user to the backend. This would allow the system to update the frontend prediction in "real-time" as the user writes, eliminating the need for the user to manually click the 'predict' button.

Investigating a Bézier curve splitting approach, such as one utilised by Google in Carbune et al. [2], could also be beneficial. Studying the performance of this algorithm across various models, not merely LSTMs, would further contribute to the existing research.

Another promising direction could be implementing a transformer model for handwriting recognition based on Bézier curves. By comparing the performance of this model with recurrent neural network models, it could provide meaningful insights, particularly given the potential performance boost from the self-attention mechanism of transformers.

6.2 Ethical Considerations

This work does not involve the collection of personal data. Both the IAM-OnDB and ISGL datasets were used for training and evaluating the performance of the models, which anonymise the writer IDs contributing to the data. Moreover, the handwriting recognition models utilising the IAM-OnDB dataset go through a preprocessing stage where data is transformed into a list of Bézier curve features. When users interact with the web application, only (x, y) stroke points are recorded. Importantly, no personal identifiers are collected or stored by the application, further ensuring the privacy of users. Bézier curves are fitted to the strokes to generate predictions, and no images are processed or stored. This design choice mitigates the risk of inference attacks on the models. Even if an attacker tried to reconstruct the training data, they could only get access to anonymised Bézier curve features, preventing any linkage to personal identifiers.

Additionally, it's important to consider the environmental implications of this work. The extraction of different data sets and the training of multiple models required significant computational resources, particularly as the Imperial Department of Computing's Slurm GPU cluster was heavily used for development. Future work in this area should consider strategies to reduce environmental impact while maintaining research and development progress.

Bibliography

[1] K. C. Santosh and E. Iwata. Stroke-based cursive character recognition, 2013.

[2] Victor Carbune, Pedro Gonnet, Thomas Deselaers, Henry A. Rowley, Alexander Daryin, Marcos Calvo, Li-Lun Wang, Daniel Keysers, Sandro Feuz, and Philippe Gervais. Fast multi-language lstm-based online handwriting recognition, 2020.

[3] Bing Cheng and D. M. Titterington. Neural Networks: A Review from a Statistical Perspective. *Statistical Science*, 9(1):2 – 30, 1994. doi: 10.1214/ss/1177010638. URL https://doi.org/10.1214/ss/1177010638.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[5] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[6] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

[7] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.

[8] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735.

[9] Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. *Efficient BackProp*, pages 9–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35289-8. doi: 10.1007/978-3-642-35289-8_3. URL https://doi.org/10.1007/978-3-642-35289-8_3.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[11] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks, 2013.

[12] Alexander Amini, Ava Soleimany, Sertac Karaman, and Daniela Rus. Spatial uncertainty sampling for end-to-end control, 2019.

[13] Daksh Trehan. *Gradient Descent Explained*. [Online]. Available from: https://towardsdatascience.com/gradient-descent-explained-9b953fc0d2c. [Accessed 5th May 2023].

[14] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436, 2015.

[15] Simeon Kostadinov. *Understanding Backpropagation Algorithm*. [Online]. Available from: https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd. [Accessed 5th May 2023].

[16] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[17] Lutz Prechelt. Automatic early stopping using cross validation: quantifying the criteria. *Neural Networks*, 11(4):761–767, 1998. ISSN 0893-6080. doi: https://doi.org/10.1016/S0893-6080(98)00010-0. URL https://www.sciencedirect.com/science/article/pii/S0893608098000100.

[18] Jiayan Qiu. *Convolutional Neural Network based Age Estimation from Facial Image and Depth Prediction from Single Image*. PhD thesis, 01 2016.

[19] Miao Gao, Guoyou Shi, and Shuang Li. Online prediction of ship behavior with automatic identification system sensor data using bidirectional long short-term memory recurrent neural network. *Sensors*, 18(12), 2018. doi: 10.3390/s18124211. URL https://www.mdpi.com/1424-8220/18/12/4211.

[20] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. A review of recurrent neural networks: Lstm cells and network architectures. *Neural Computation*, 31:1–36, 05 2019. doi: 10.1162/neco_a_01199.

[21] Savvas Varsamopoulos, Koen Bertels, and Carmen Almudever. Designing neural network based decoders for surface codes, 11 2018.

[22] Abien Fred M. Agarap. A neural network architecture combining gated recurrent unit (GRU) and support vector machine (SVM) for intrusion detection in network traffic data. In *Proceedings of the 2018 10th International Conference on Machine Learning and Computing*. ACM, feb 2018. doi: 10.1145/3195106.3195117. URL https://doi.org/10.1145%2F3195106.3195117.

[23] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014. URL http://arxiv.org/abs/1406.1078.

[24] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.

[25] Alex Graves, Abdel rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks, 2013.

[26] M. Schuster and K.K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997. doi: 10.1109/78.650093.

[27] Tim A Pastva. Bezier curve fitting. Master's thesis, Naval Postgraduate School, 1998.

[28] Gerald E. Farin and Dianne Hansford. *The Essentials of CAGD*. A. K. Peters, Ltd., USA, 1st edition, 2000. ISBN 1568811233.

[29] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice (2nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., USA, 1990. ISBN 0201121107.

[30] Les Piegl and Wayne Tiller. *The NURBS Book*. Springer-Verlag, New York, NY, USA, second edition, 1996.

[31] CK Shene. Derivatives of a bézier curve. [Online]. Available from: https://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/Bezier/bezier-der.html. [Accessed 25th May 2023].

[32] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[33] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks, 2014.

[34] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, page 369–376, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933832. doi: 10.1145/1143844.1143891. URL https://doi.org/10.1145/1143844.1143891.

[35] Alex Graves. Sequence transduction with recurrent neural networks, 2012.

[36] Awni Hannun. Sequence modeling with ctc. *Distill*, 2017. doi: 10.23915/distill.00008. URL https://distill.pub/2017/ctc.

[37] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality, 2013.

[38] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches, 2014.

[39] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation, 2015.

[40] Andrew Morris, Viktoria Maier, and Phil Green. From wer and ril to mer and wil: improved evaluation measures for connected speech recognition. 10 2004. doi: 10.21437/Interspeech.2004-668.

[41] Maximilian Schrapel, Max-Ludwig Stadler, and Michael Rohs. Pentelligence: Combining pen tip motion and writing sounds for handwritten digit recognition. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, page 1–11, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356206. doi: 10.1145/3173574.3173705. URL https://doi.org/10.1145/3173574.3173705.

[42] Ibrahim Adeyanju, Elijah Omidiora, Olusayo Fenwa, Omodolapo Babalola, and Opeyemi Durodola. Isgl online and offline character recognition dataset. Mendeley Data, 2019. doi: 10.17632/n7kmd7t7yx.1.

[43] Marcus Liwicki and Horst Bunke. Iam-ondb - an on-line english sentence database acquired from handwritten text on a whiteboard. In *Proceedings of the Eighth International Conference on Document Analysis and Recognition*, ICDAR '05, page 956–961, USA, 2005. IEEE Computer Society. ISBN 0769524206. doi: 10.1109/ICDAR.2005.132. URL https://doi.org/10.1109/ICDAR.2005.132.

[44] Kanti Mardia and Peter Jupp. *Directional Statistics*. 01 2000. ISBN 9780471953333. doi: 10.1002/9780470316979.ch11.