# DIGITAL FONT GENERATION USING LONG SHORT-TERM MEMORY NETWORKS

**A Thesis Submitted to**
**the Graduate School of Engineering and Sciences of**
**İzmir Institute of Technology**
**in Partial Fulfillment of the Requirements for the Degree of**

**MASTER OF SCIENCE**

**in Computer Engineering**

**by**
**Onur TEMİZKAN**

**July 2019**
**İZMİR**

We approve the thesis of **Onur TEMİZKAN**

**Examining Committee Members:**

**Asst. Prof. Dr. Mustafa ÖZUYSAL**
Department of Computer Engineering, İzmir Institute of Technology

**Asst. Prof. Dr. Nesli ERDOĞMUŞ**
Department of Computer Engineering, İzmir Institute of Technology

**Asst. Prof. Dr. Zerrin IŞIK**
Department of Computer Engineering, Dokuz Eylül University

**05 July 2019**

**Asst. Prof. Dr. Mustafa ÖZUYSAL**
Supervisor, Department of Computer Engineering,
İzmir Institute of Technology

**Assoc. Prof. Dr. Tolga AYAV**
Head of the Department of
Computer Engineering

**Prof. Dr. Aysun SOFUOĞLU**
Dean of the Graduate School of
Engineering and Sciences

# ACKNOWLEDGMENTS

I would like to thank my supervisor Mustafa Özuysal for his invaluable support to my ideas and guidance throughout this research. Without his encouragement and contribution, this work could not be completed.

I would also express my gratitude to my beloved family for their love, endless support and patience. This thesis is dedicated to them.

# ABSTRACT

DIGITAL FONT GENERATION USING LONG SHORT-TERM MEMORY NETWORKS

Long Short-Term Memory (LSTM) Networks are powerful models to solve sequential problems in machine learning. Apart from their use on sequence classification, LSTMs are also used for sequence prediction. Predictive features of LSTMs have been used extensively to generate handwriting, music and several other types of sequences. Configuration and training of LSTM networks are relatively more arduous than non-sequential models, especially when input data is complex. In this research, the aim is to train LSTM networks and its different variations, use their generative features on a relatively obscure and complex type of sequences in machine learning; digital fonts. Controlled experiments have been performed to find the effects of different model parameters, input encodings or network architectures on learning font based sequences. All in all, in this document; the procedure of creating a dataset from digital fonts are provided, training strategies are demonstrated and the generative results are discussed.

**Keywords**: long short-term memory, sequence generation, digital fonts

# ÖZET

UZUN KISA VADELİ BELLEK AĞLARI İLE SAYISAL YAZI TİPİ ÜRETİMİ

Uzun Kısa Vadeli Bellek Ağları, makine öğrenmesi alanında, dizisel veri içeren problemlerde başarıyla kullanılmaktadır. Dizi sınıflandırma alanındaki yaygın kullanımlarına ek olarak, Uzun Kısa Vadeli Bellek Ağları'ndan, dizi öngörüsü alanında da yararlanılmaktadır. Bu ağların tahmin yetenekleri, el yazısı üretimi, müzik üretimi, ve diğer diziler üzerinde üretim için de geniş çapta tercih edilmektedir. Ancak, diğer makine öğrenmesi yöntemleri ile karşılaştırıldıklarında; Uzun Kısa Vadeli Bellek Ağlarının konfigürasyonları ve eğitim aşamaları, eğitilecek veri karmaşıklaştıkça daha fazla zorlaşmaktadır. Bu araştırmanın hedefi, Uzun Kısa Vadeli Bellek Ağlarının ve türevlerinin, göreceli olarak karmaşık bir veri olan sayısal yazı tipleri üzerinde denemektir. Bu amaçla kontrollü deneyler yapılmış, Uzun Kısa Vadeli Bellek Ağlarının farklı konfigürasyonlardaki başarıları ölçülmüş ve karşılaştırılmıştır. Bu dokümanda, sayısal yazı tipleri kullanılarak bir makine öğrenmesi veri tabanı oluşturulma süreci, makine eğitimi aşamaları ve stratejileri açıklanmış, sayısal yazı tipi üretim sonuçları gösterilmiş ve incelenmiştir.

**Anahtar Kelimeler**: uzun kısa vadeli bellek, dizi üretimi, sayısal yazı tipleri

# TABLE OF CONTENTS

LIST OF FIGURES ........ ix

LIST OF TABLES ........ xi

LIST OF ABBREVIATIONS ........ xii

CHAPTER 1. INTRODUCTION ........ 1
- 1.1. Motivation ........ 1
- 1.2. Aim and Objectives ........ 2
- 1.3. Disposition ........ 3

CHAPTER 2. RELATED WORK ........ 5

CHAPTER 3. BACKGROUND ........ 7
- 3.1. Material ........ 7
  - 3.1.1. Fonts and Typefaces ........ 7
    - 3.1.1.1. Early History ........ 7
    - 3.1.1.2. Analogue Era ........ 8
    - 3.1.1.3. Digital Era ........ 8
    - 3.1.1.4. Categorization ........ 9
  - 3.1.2. Digital Fonts ........ 10
    - 3.1.2.1. Vector Formats ........ 11
    - 3.1.2.2. Bézier Curves and Splines ........ 12
    - 3.1.2.3. Bézier Control and Anchor Points ........ 13
    - 3.1.2.4. Point Classification Tokens ........ 14
    - 3.1.2.5. Glyph Labels ........ 14
  - 3.1.3. Dataset ........ 14
- 3.2. Sequential Deep Learning ........ 15
  - 3.2.1. Recurrent Neural Networks ........ 15
  - 3.2.2. Long Short-Term Memory Networks ........ 16
  - 3.2.3. Bidirectional Long Short-Term Memory Networks ........ 17
  - 3.2.4. Optimization Methods ........ 18

CHAPTER 4. METHODOLOGY ........ 20
4.1. Preprocessing ........ 20
4.1.1. Parsing ........ 20
4.1.2. Subsetting ........ 21
4.1.3. Encoding and Feature Definitions ........ 22
4.1.4. Adjusting Starting Point ........ 23
4.2. Training ........ 23
4.2.1. Input Representation ........ 24
4.2.2. Foundational Neural Network Model ........ 25
4.2.3. Batching ........ 25
4.2.3.1. Padded and Masked Sequential Training ........ 26
4.2.3.2. Point by Point Training ........ 27
4.2.3.3. On-line Training ........ 28
4.2.4. Learning Rate Strategies ........ 29
4.2.4.1. Static Learning Rate ........ 30
4.2.4.2. Decaying Learning Rate ........ 30
4.2.5. Data Encoding ........ 30
4.2.5.1. Coordinates Encoding ........ 30
4.2.5.2. Point Flag Encoding ........ 33
4.2.5.3. Glyph Index Encoding ........ 33
4.2.6. Alternative LSTM Architectures ........ 33
4.3. Digital Font Generation ........ 35
4.4. Software Infrastructure ........ 35

CHAPTER 5. EXPERIMENTS ........ 37
5.1. Criterion on Generative Performance ........ 37
5.1.1. Overlaps ........ 37
5.1.2. Closing Shapes ........ 37
5.1.3. Contour Count ........ 37
5.2. Experimental Results ........ 38
5.2.1. Per Glyph Training ........ 39
5.2.2. Per Glyph Group Training ........ 40
5.3. LSTM and BLSTM Layers ........ 40
5.3.1. LSTM Layers ........ 41
5.3.2. BLSTM Layers ........ 43

5.4. The Effect of Learning Rates ........ 44

CHAPTER 6. CONCLUSION ........ 49
6.1. Future Work ........ 49

REFERENCES ........ 51

INDEX ........ 55

# LIST OF FIGURES


| Figure | Page |
|---|---|
| Figure 1.1. Comparison of a handwritten character and a font glyph. | 2 |
| Figure 2.1. Online handwriting generation with LSTM networks. | 5 |
| Figure 3.1. Textura font in Gutenberg Bible | 8 |
| Figure 3.2. Construction of letter 'B' with geometric shape guides. | 9 |
| Figure 3.3. Categories of fonts and typefaces | 10 |
| Figure 3.4. Comparison of a raster font and a vector font. | 11 |
| Figure 3.5. Example of a composite glyph. | 12 |
| Figure 3.6. A cubic Bézier curve | 13 |
| Figure 3.7. Different Bézier representations. | 14 |
| Figure 3.8. Comparison of FFNNs and RNNs. | 16 |
| Figure 3.9. RNN nodes (unrolled). | 17 |
| Figure 3.10. Comparison of RNN node and LSTM node | 17 |
| Figure 4.1. Overall system architecture. | 20 |
| Figure 4.2. Flowchart of preprocessing module | 21 |
| Figure 4.3. Starting point adjustment | 23 |
| Figure 4.4. Flowchart of training module. | 24 |
| Figure 4.5. Foundational LSTM model for experiments | 26 |
| Figure 4.6. Data padding strategies. | 27 |
| Figure 4.7. Flowchart of point-by-point training. | 28 |
| Figure 4.8. Flowchart of dynamic batch-sized training | 29 |
| Figure 4.9. Coordinate encodings. | 31 |
| Figure 4.10. Input layer and architecture of foundational model | 34 |
| Figure 4.11. Separated input architecture - alternative 1 | 34 |
| Figure 4.12. Separated input Architecture - alternative 2 | 35 |
| Figure 4.13. Flowchart of font generation module | 36 |
| Figure 5.1. Overlap condition on glyph B | 38 |
| Figure 5.2. Non-closing condition on glyph t | 39 |
| Figure 5.3. Contour count mismatch on glyph A | 40 |
| Figure 5.4. Generative variance of N. | 41 |
| Figure 5.5. Generative results of [E, H, I, W, X, Z]. | 43 |
| Figure 5.6. Generative results of [C, G, J, S, U] | 44 |

Figure 5.7. Generative results of A through epochs. .................................... 46
Figure 5.8. Generative results of e through epochs. .................................... 47
Figure 5.9. LSTM and BLSTM generation results of E. .............................. 48

# LIST OF TABLES

**Table** **Page**

Table 4.1. Input representation of font dataset. .......................................... 25

Table 5.1. Model parameters for single-glyph training. ................................. 41

Table 5.2. Performance comparison between LSTM and BLSTM layers. ........... 42

Table 5.3. Performance comparison between static and decaying learning rates. .... 45

# LIST OF ABBREVIATIONS

RNN . . . . . . . . . . Recurrent Neural Network
LSTM . . . . . . . . . . Long Short-Term Memory
BRNN . . . . . . . . . . Bidirectional Recurrent Neural Network
BLSTM . . . . . . . . . . Bidirectional Long Short-Term Memory
OTF . . . . . . . . . . OpenType Font
TTF . . . . . . . . . . TrueType Font

# CHAPTER 1

# INTRODUCTION

Fonts are almost always the medium of textual communication today. Apart from their primary use case, representing text; the styles and properties of different fonts affect readers' perception on the information that is written. Having a long history, font design is a very established but still an active branch of visual arts. This research aims to apply the generative abilities of Long Short-Term Memory networks on sequential data from digital fonts and to create machine-generated fonts.

Sequence generation is a very popular use case of Recurrent Neural Networks and Long Short-Term Memory Networks [1, 2]. There is plenty of research on generating handwriting [3], music [4] and text [5] with RNNs. They are also used for computer vision based tasks such as image recognition [6], classification [7] and captioning [8].

Furthermore, a few number of studies have been conducted on font generation topic. The major approach on this purpose is using example based methods to create composite fonts. This approach requires user input and a separate dataset for blended generation of fonts [9, 10]. With these requirements, mentioned studies are mainly established on artistic style learning and transfer learning concepts. Studies on font generation are based on several methods such as numerical methods [11], genetic algorithms [12], feed-forward neural networks [10] and generative adversarial networks [13, 14]. The main drawback on these approaches are their representation of font glyphs in training. Font glyphs and optionally user input are generally extracted from rasterized image data in most studies. Even the expected generative outputs are either image data or outline vectors; the concept of not using vector data directly from native digital font definition vectors introduces impurity and a need for data inference during training.

This study uses sequential vector data that is defined in modern digital fonts as both input and expected output. With this approach, we strive to preserve the purity of native font data and create high-quality output that is in native sequential form and encodable into commercially distributed fonts without data loss.

In this research, the endeavour is to generate glyph outlines from digital fonts using Long Short-Term Memory Networks and discuss the effects of different training strategies on generative performances of LSTM models on digital font data.

## 1.1. Motivation

Sequence generation on 2-dimensional coordinate systems is a well researched subject, because of popular topics such as *"Online Handwriting Generation"*. Handwriting generation is in a partly similar context with the domain of this dissertation. The main difference is, while online handwriting data is basically a set of independent *pen strokes*, vector font glyphs are sets of closed shapes called *contours* or *outlines*.

In some aspects, generating glyph outlines is a more complex problem than generating strokes. The primary reason is that, handwritten characters have single contours, while to form closed shapes, glyph outlines have at least two contours. In other words, while drawing a border of a handwritten character, one draw in a single direction is required. Contrarily, to draw a border of a character in a digital font, at least two lines or curves in two opposite directions are needed. Figure 1.1 visualizes the differences between a handwritten character and a font glyph.

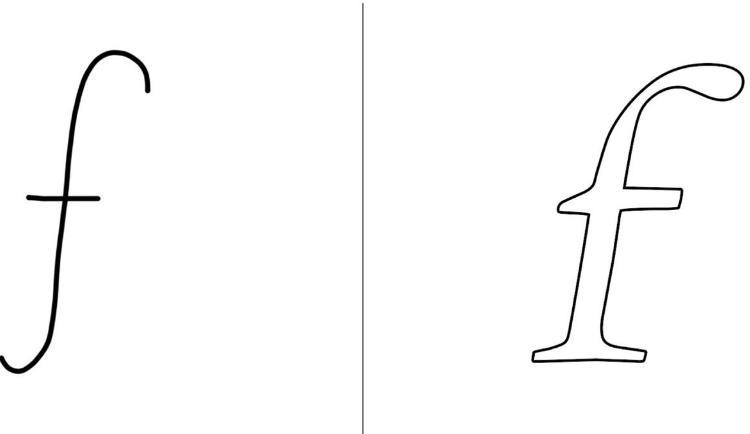


Figure 1.1. The handwritten *f* character (left) is formed of two strokes, while the font glyph for the same letter (right) is represented with its outline which is a complex closed shape.

Another challenging situation in digital font generation is the necessity of closing shapes. By definition, font glyphs are defined as sets of closed shapes and to assure the validity of generated glyphs, the generative model should end each outline at its starting point.

The main motivation of this research is to tackle the challenge of generating this relatively complex type of sequences with Long Short-Term Memory Networks.

## 1.2. Aim and Objectives

The aim of this project is to create a dataset from digital font definitions, to train an LSTM network with the dataset and to discuss the generation results. The experiment results are used to analyse the capabilities of LSTM networks in variable sized, multi-feature, complex sequences.

Besides, the objectives of this research are:

- To create a sequential machine learning dataset from digital font data,
- To create LSTM deep learning models and train them with the dataset,
- To generate glyph outline representations from trained models,
- To benchmark and analyse the experiment results.

## 1.3. Disposition

The first chapter of this document is the Introduction. The brief definition of the problem and data are explained in it, the motivation, aim and objectives of this study are declared.

Chapter 2: Related Work is an overview of previous studies on sequential learning in 2-dimensional coordinate systems.

Chapter 3: Background prefaces the history and characteristics of font data, reviews learning based methods such as Recurrent Neural Networks and Long Short-Term Memory Networks, their variants and optimization methods.

Chapter 4: Methodology is a step by step explanation of the proposed system including dataset creation, data preprocessing, training strategies and generation. Section 4.1: Preprocessing; contains details about dataset creation and feature engineering work on font data. Section 4.2: Training, is a comprehensive background of training the proposed system with different strategies and parameters. In this section, the variations on training approaches are elaborated in groups for each variant. Section 4.3: Digital Font Generation, summarizes the workflow of the generation logic in this study.

Chapter 5: Experiments, demonstrates the generative results of experiments that are performed through this research. The results are grouped by control points that are mentioned in Section 4.2.

Chapter 6: Results, contains remarks on this research process and discussion about its output. Subjective opinions about possible enhancements and modifications on this work are given.

# CHAPTER 2

# RELATED WORK

There are plenty of research in literature on sequence generation with deep learning models. As mentioned in Chapter 1, these researches are made in numerous data domains such as text, video and music and more [3–5].

In perspective of research domain, handwriting generation is the most engaging area of research to font generation and these two terms are frequently used at close quarters [13, 15]. Apart from their relation of representing the same information in different structures, font data and digital handwriting data have similar data features and characteristics. Both data domains are defined in 2-dimensional coordinate systems. Also, the primitives of both data domains are sets of curves.

There are several studies in literature on handwriting generation using Bayesian Networks [16], Beta-Elliptic Models [17], Hidden Markov Models [18] and Variational Auto-Encoders [19]. Having a close relation to the methodology of our research, *"Generating Sequences with Recurrent Neural Networks"* by Graves [3] contains an influential study on handwriting generation with LSTM models. An output sample from that paper is shown in Figure 2.1.

Figure 2.1. Generative results of handwriting from LSTM networks in different writing styles (Source: [3]).

Besides, as mentioned in Chapter 1, previous studies on font generation were

mainly based on systems that require user interaction to perform artistic style learning in combination with font datasets. Suveeranont and Igarashi [9] proposed a skeletal glyph representation to apply styles of handwritten characters to outline fonts. Also, Lian et. al. [10] used vectorization of raster font images and handwritten character strokes for font generation and experimented sequential deep learning models for handwritten glyph strokes in their proposed style learning system. Yoshida et. al. [12] proposed a genetic algorithm based system for generating blends of multiple vector fonts as raster images. Using Bézier Curves for Chinese character strokes, Li et. al. [15] introduced a procedural font generation algorithm. Finally, Hayashi et. al [14] and Jiang et. al. [13] used generative adversarial networks for font style learning and generation.

This study proposes LSTM networks for learning and generating vector font glyph outlines. In the proposed system, glyph outlines are defined as sequential input and output, aiming minimal loss in purity of original font definitions.

# CHAPTER 3

# BACKGROUND

## 3.1. Material

The material and the expected output of this study is digital fonts. Fonts are very diverse and well structured data sources having a long historical background. This section contains brief information about terminology, history and mathematical background of digital fonts.

### 3.1.1. Fonts and Typefaces

Fonts are defined as *"an assortment or set of type or characters all of one style and sometimes one size"* according to Merriam-Webster Online Dictionary [20]. They are used in almost every printed document and digital devices today, in order to provide any information in textual format.

On the other hand, typefaces are families of fonts that have common design. Typefaces can contain multiple fonts in several styles, weights and postures. For instance; *Helvetica* is a typeface and *Helvetica Bold* is a font.

Fonts have a long history starting with the Medieval Age and have been evolving more rapidly than ever after the digitalization of the world. This section provides a concise history of fonts and their evolution.

#### 3.1.1.1. Early History

The history of fonts has started with the invention of the printing press. At that time, letters and punctuation marks were being smelted into metal blocks and aligned together as page blocks to create page layouts. That representation helped to keep an entire document in a uniform visual language. The first printed document in history: *"The Gutenberg Bible (1455)"* (Figure 3.1) is printed with the first font of the world,

*"Textura"*. This font had been designed by Johannes Gutenberg who is also the inventor of printing press. *Textura* was designed as a uniform representation of gothic handwritten letters in that era.

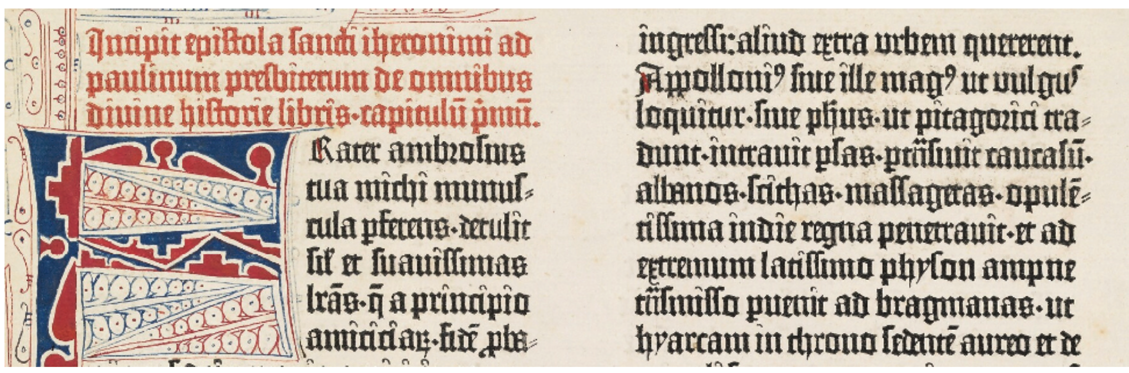

Incipit epistola sancti iheronimi ad
paulinum presbiterum de omnibus
divine historie libris-capitulū pmū.
Frater ambrosius
tua michi munus-
cula pferens-detulit
sil et suavissimas
lrās-q̄ a principio
amicicias-fidē pba-
ingressi: aliud extra urbem quererent.
Apollonius sive ille magus ut vulgus
loquitur-sive phus-ut pitagorici tra-
dunt-intravit psas-ptrāsivit caucasū-
albanos-scithas-massagetas-opulē-
tissima indie regna penetravit-et ad
extremum latissimo physon amne
trāsmisso pvenit ad bragmanas-ut
hyarcam in throno sedentē aureo et de

Figure 3.1. Excerpt from *Gutenberg Bible (1455)* featuring *Textura* font (Source: [21]).

### 3.1.1.2. Analogue Era

After Gutenberg, until the invention of computers, typography had become a prevalent domain of Renaissance art. A large number of typefaces had been designed and used in press and architecture. Thereupon, typefaces were being designed in an analogue way. The glyph designs were usually being drawn on plain or graph papers. The glyph proportions were either not defined explicitly or defined with the help of basic geometrical shapes [22].

Geometric shape based typefaces were applicable to any scale, as a result of having ratio based definitions. The printing heads in different scales were able to be created keeping the original design specifications [22].

### 3.1.1.3. Digital Era

The invention of computers and devices with electronic displays led to a need for digital font rendering. For this purpose, digital fonts were defined in a digital data schema.

A typical digital screen is basically a matrix of pixels. Considering the number of available pixels is varied between devices, correctly rendering text in various sizes on

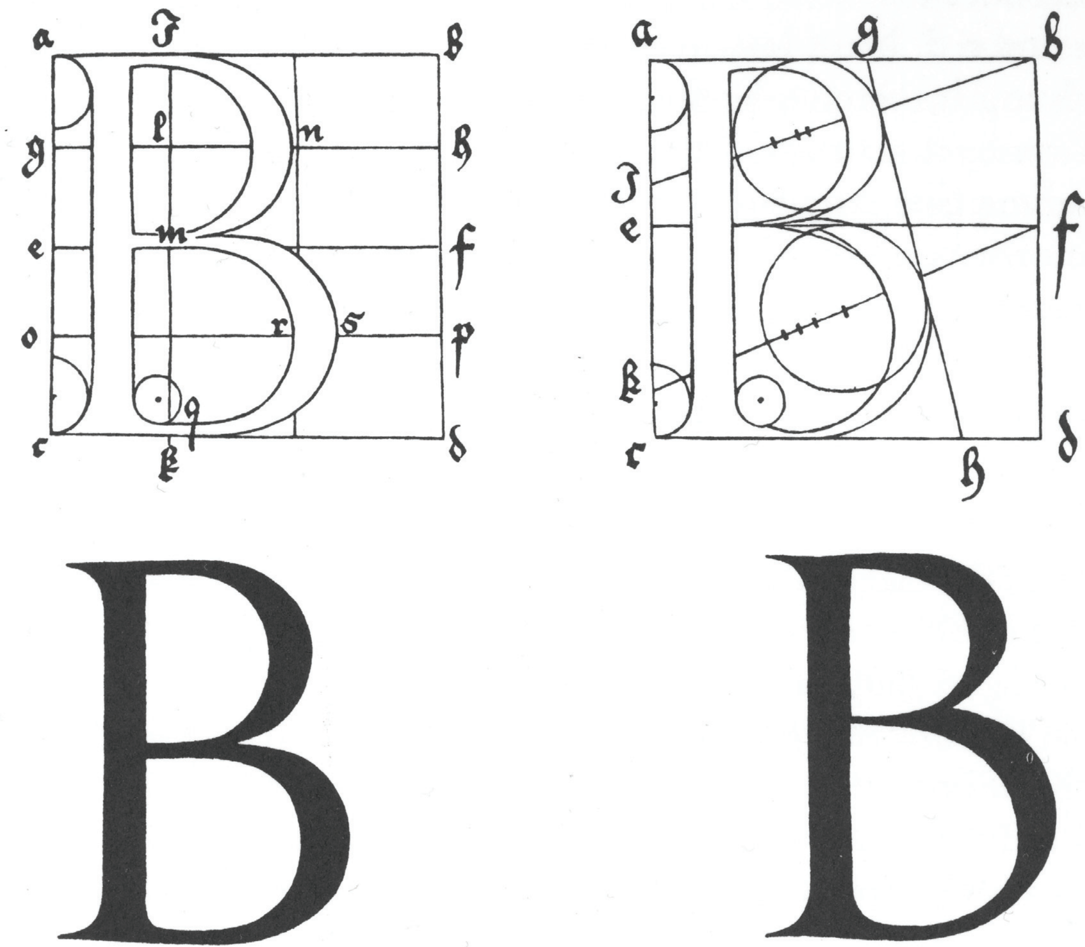


Figure 3.2. Standardization attempts in early era of fonts were based on several geometric guides. The design specification of letter *B* is an example of a combination of grid based design (left), and circular shape based design (right) (Source: [22]).

screens had been critical. To render fonts on pixel areas in different scales correctly, a scalable definition like the geometric definition mentioned in Section 3.1.1.2: Analogue Era, should had been defined.

Several industrial pioneers of that time created their own standards for that purpose, yet the basic idea was similar: Keeping font definitions in vector representation. The details about the standards are given in Section 3.1.2.1 and the most widely used vector representation is elaborated in Section 3.1.2.2.

### 3.1.1.4. Categorization

While typefaces are sets of letters in visual consistency, they have various characteristics and psychological effects on readers. For example, *"Garamond"* is known as a good typeface for paragraph content, while *"Futura"* is more appropriate for headings such as news article titles [23]. These versatile characteristics created a need for categorization. Some typeface categories and categorization choices are opinionated, yet the widespread categorization is accepted and used in this study.

Fonts and typefaces are categorized in several ways and there are plenty of categories in each criterion. For simplicity, most common criteria and categories are mentioned in this report. Figure 3.3 is a visual example of different font and typeface characteristics.

Normal *Italic* *Oblique*

Light Normal **Bold**

Serif Sans-Serif **Slab-Serif** Monospace

Roman **Grotesque** Humanist **Geometric**

Figure 3.3. Fonts are categorized by their posture (first row), weight (second row), by several design approaches (third row), and by artistic movements (bottom row). Texts denote categories of each example.

### 3.1.2. Digital Fonts

Modern Digital fonts can be described as sets of glyph definitions that contain external features such as hinting, ligatures, baselines and offsets, alongside glyph outline definitions. Not every digital font is vector-based, a part of fonts are known as *raster fonts*

or *bitmap fonts* by reason of being defined in bitmap image matrices. The comparison of *raster fonts* and *vector fonts* are shown in Figure 3.4.

Figure 3.4. A raster font (left) is represented in an image matrix, which does not scale to different sizes. A vector font (right) is represented in a 2-dimensional coordinate system which can be scaled without loss of readability.

Bitmap fonts had been widely preferred on primitive computers and gaming consoles such as *Commodore 64* and *Nintendo Entertainment System*, because the advantage of sparing computational force and memory to parse and render vector fonts were critical. The drawback was that these fonts would lose their legibility on different scales.

Almost all modern fonts are represented by vectors today. The main advantage of using vector fonts are their scalability on different sizes and adaptability to different screens and print environments. [24]

In this research, the aim is to generate vector fonts. Raster fonts are out of the scope of this work.

#### 3.1.2.1. Vector Formats

There are several standards for representation of vector fonts on digital devices. Though, the main idea is almost the same. Adobe, Microsoft, Apple and other industrial pioneers created different standards. *Adobe*'s *PostScript* and *Type 1* formats, *Apple*'s and *Microsoft*'s *TrueType* and *Adobe and Microsoft*'s *OpenType* formats are very similar to each other. The differences between them are mostly rendering or encoding extensions. In this research, the main interest is on *Glyph Outlines*, which are almost the same in all of these formats.

For description of a font format, *OpenType* standard is used in this section. However, almost all of the specification applies also to other formats.

*OpenType* fonts are described as a set of tables. Those tables represent different mappings that are needed to render a font, including; *(eg: character-to-glyph, index-to-location)*, *metadata*, *glyph representations*, *scaling*, *hinting*, *rendering guides* and more.

For the purpose of this study, glyph representations are primarily needed. In *OpenType* and *TrueType* formats, that information is stored in `GLYF` table. `GLYF` table contains one or more outline shapes for each glyph in *Bézier-Spline* format.

The second table that is needed for this study is `CMAP` table. This table maps characters to corresponding glyphs. One important thing to point out about character-glyph mapping is that, a character may be mapped to multiple glyph definitions at once. Or a glyph can consist of multiple glyphs. Those types of glyphs are called *Composite Glyphs*. Figure 3.5 shows an example of this situation.

Figure 3.5. Glyph "`e`" (Left), is combined with glyph "`acute`" (Middle), to form a composite glyph "`eacute`" (Right). With this concept, it's possible to create accented letters or composite symbols without using extra space.

As using composite glyphs do not contribute to the generative purposes of this research. They are discarded for simplicity.

### 3.1.2.2. Bézier Curves and Splines

Bézier curves are parametric curves that are used extensively to represent high-order 2-dimensional curves. These curves use multiple control and anchor points to define a curve. Details about control and anchor points are given in Section 3.1.2.3

Since font outlines are basically closed shapes, to model them, multiple con-

secutive Bézier curves are needed. Bézier-Splines, which are also called as *Composite Bézier Curves* are effective representations of multiple connected cubic or quadratic Bézier curves.

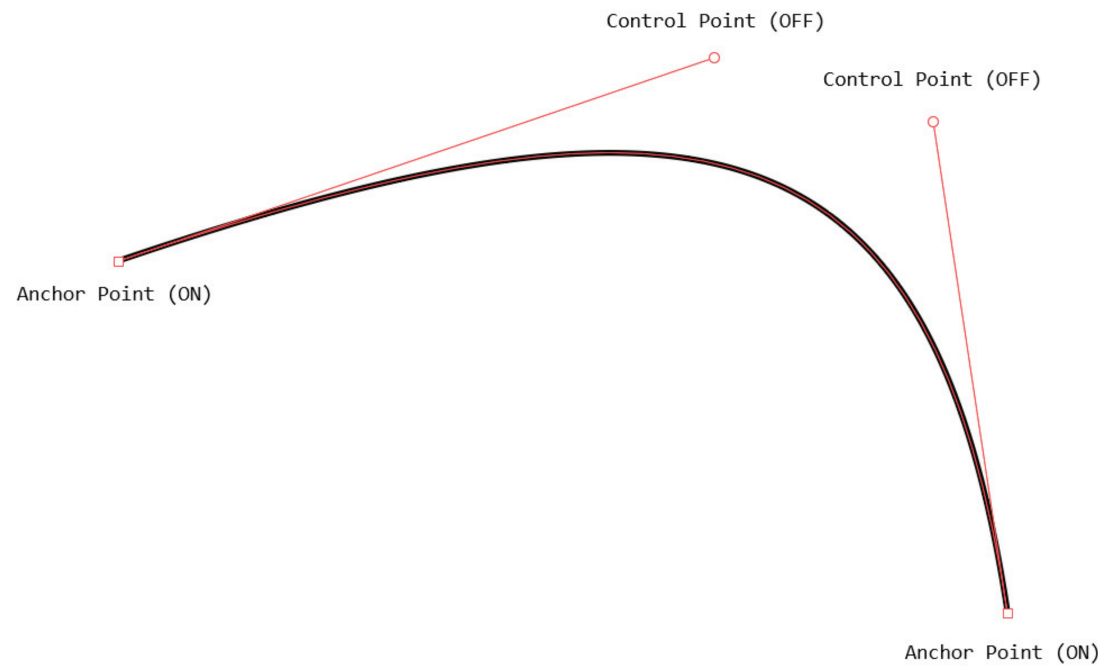



Figure 3.6. This cubic Bézier curve has two anchor points, one at the beginning and one at the end and two consecutive control points between anchors. The distance between these two control points and their relative distances to their neighbouring anchor points, define the direction and orientation of the curve.

#### 3.1.2.3. Bézier Control and Anchor Points

In *Bézier Curves and Splines*, there are two types of points: `ON` and `OFF` points. `ON` points are named as *on-the-curve* or *anchor* points, and `OFF` points are named as *off-the-curve* or *control* points. This classification is necessary for all point time-steps. Relevant definitions of Bézier format for this study are,

- Two consecutive `ON` points represent a line.
- One `OFF` point following an `ON` point represents a single *Quadratic Bézier Curve*.

- Two consecutive `OFF` points following an `ON` point represents a single *Cubic Bézier Curve*.

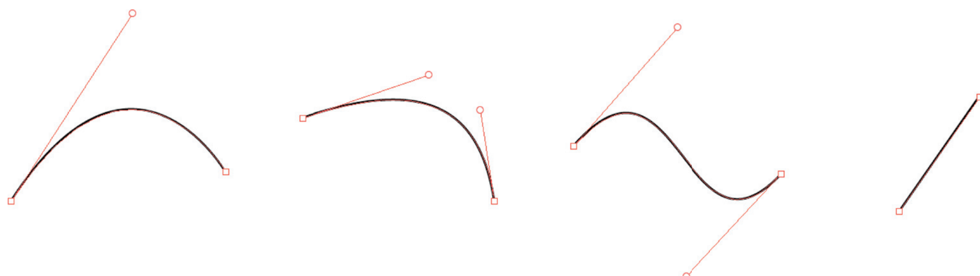


Figure 3.7. Quadratic Bézier curve parabola (left) has one control point. Cubic Bézier curve parabola (middle-left) forms a parabola with two control points. Cubic Bézier curve hyperbola (middle-right) also has two control points. Bézier line (right) is defined by two consecutive anchor points.

#### 3.1.2.4. Point Classification Tokens

A glyph representation can contain one or more outlines. Each outline has a starting point, a sequence of control or anchor points, and an ending point. Those signals are important for any generative approach in order to learn when to stop or alter state. This data is not defined explicitly in either *OpenType* or *TrueType* formats. Nevertheless it is trivial to extract that information using raw glyph outline data. The details on gathering that data are given in Section 4.1.

#### 3.1.2.5. Glyph Labels

To learn which outline correspond to which glyph, labelling is needed. This data is extracted from `CMAP` table as *glyph indexes* and used as a non-temporal feature. Since this feature is non-temporal, different encoding schemes to feed it into the generative model are tested. Details on extraction of this feature and encoding strategies are given in Section 4.1.

### 3.1.3. Dataset

In this research, *Google Fonts Archive* [25] is used, This archive contains almost 3000 free and open-source *OpenType* and *TrueType* fonts, and their metadata at the time of this writing. While the choice of dataset in this project is *Google Fonts Archive*, any set of *OpenType* or *TrueType* fonts is compatible to be used while creating the dataset.

The main reason why *Google Fonts Archive* used in this project, is all the fonts have properly formatted metadata files. This metadata are used to filter and group different types of fonts. For example: While creating dataset, only Bold, Italic and Sans-Serif fonts can be selected and processed.

The relevant information that are defined in metadata files are:

- **Weight**: One from [100 ,200, 400, 600, 800]
- **Posture**: One from [Normal, Italic]
- **Style**: One from [Sans-Serif, Serif, Display, Handwriting, Symbol]

This dataset is not usable as-is for our sequential models. Preprocessing should be done on this data to form a LSTM-compatible input from raw data.

## 3.2. Sequential Deep Learning

As mentioned before, the purpose of this research is to use font data as sequences of points. To learn and generate structures of consecutive points, sequential models are needed instead of feed-forward models. This section is a concise review on sequential learning methods with deep learning, their internals and variants. Furthermore the optimization methods for training these models are mentioned and compared in this section.

### 3.2.1. Recurrent Neural Networks

RNN is a powerful model to solve machine learning problems where the data is sequential. The main idea behind RNNs is to enhance feed-forward neural networks with circular data flow, to learn temporal structure in data.

Nodes in RNNs have more than one input and output gates unlike FFNN nodes which have one for each. Those gates connect nodes in multiple directions. With the help

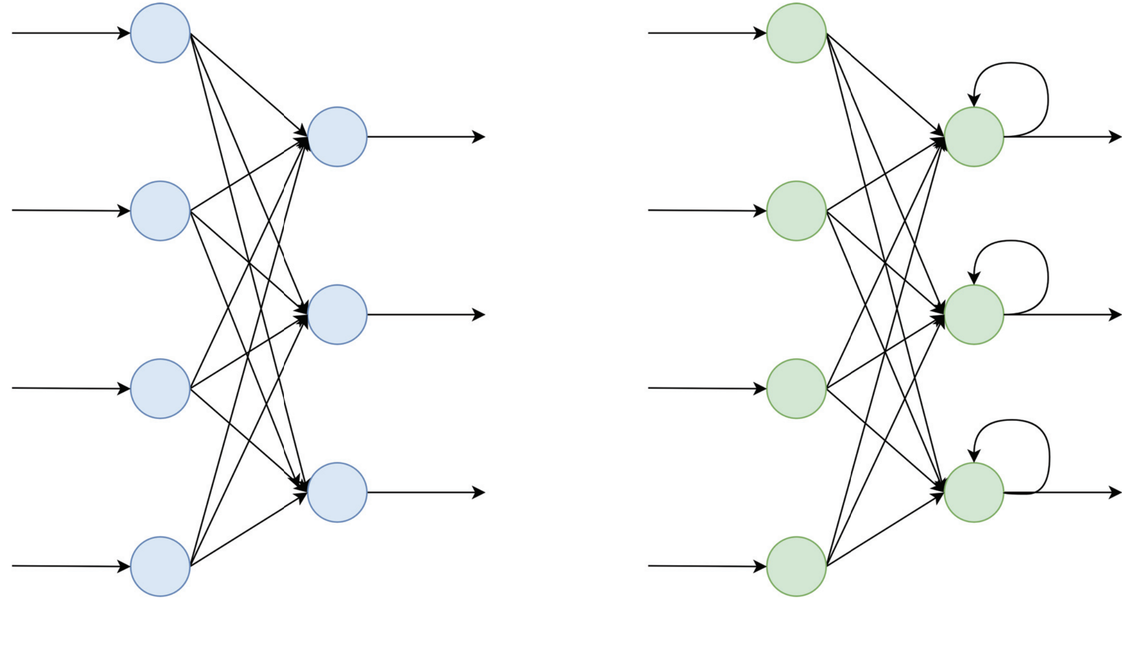



Figure 3.8. In a feed-forward neural network (left), the data flows in a single direction throughout the network, while in a recurrent neural network (right), nodes of hidden layers recirculate data and learn temporal structures in consecutive input timesteps.

of this modification, when sequential data is fed to a RNN, previous data remains for a while in the network.

On back-propagation, the gradient is also calculated with errors from multiple directions. Figure 3.9 shows forward flow in blue and backward flow in yellow.

One problem that arises in RNN training is *"Vanishing / Exploding Gradient Problem"*[26]. When sequences get longer and training time increases, the earlier nodes in RNN gets very small weight updates because the value of error gets smaller at back-propagation of each layer. While this problem is not specific to RNNs and may apply to every *deep* neural network, RNNs should be deep and have longer data flow by definition. Because of it, this problem hurts almost all RNNs in general.

### 3.2.2. Long Short-Term Memory Networks

Among a large number of trials to solve the flaws of RNNs, one successful attempt is Long Short-Term Memory Networks [1]. This approach enhances RNNs at node

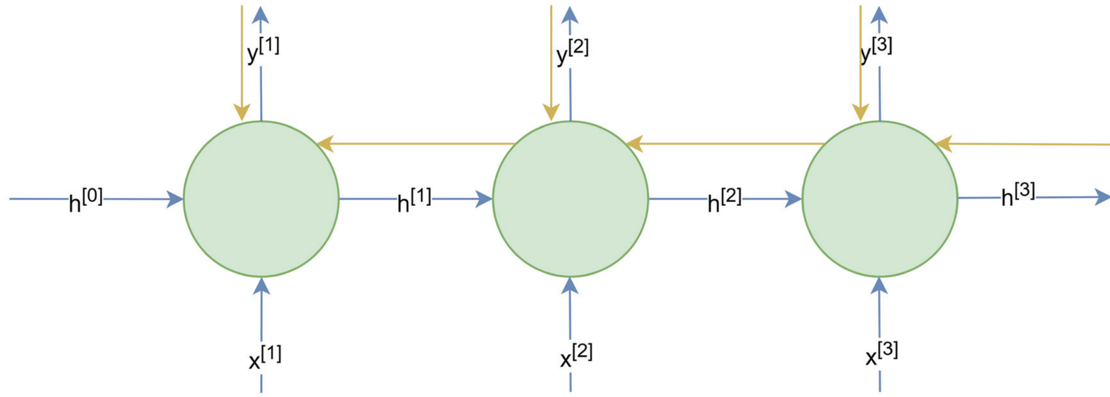



Figure 3.9. RNN Nodes (Unrolled). *h* represent the temporal state between nodes, while *x* and *y* represent inputs and outputs of nodes.

level. In LSTM specification, node definition in RNN is enhanced with two main additions, a *memory cell* and a *forget gate*. The first addition enables the node to remember longer temporal dependencies with maintaining state over time. The second addition is instrumental to clear internal state when the window of interest ends. This feature is a powerful tool to learn from variable-length sequences and manually clean temporal window of interest before starting to learn from a new sequence. Figure 3.10 shows the main differences between a naïve RNN node and an LSTM node.

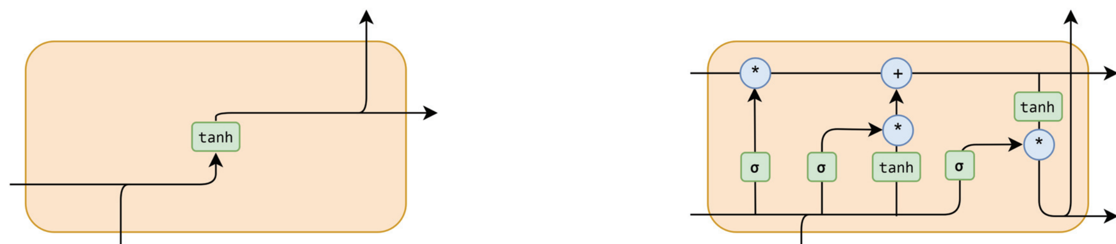



Figure 3.10. While RNN node (left) has only a *tanh* activation gate in its data flow, LSTM node (right) also has a cell-state which can be updated or flushed when needed.

### 3.2.3. Bidirectional Long Short-Term Memory Networks

Bidirectional Recurrent Neural Network (BRNN) [27] is a modification of RNN which can be trained simultaneously in positive and negative time directions. In this approach, half of the state neurons are assigned to forward data and the other half to backward data. This modification strives to enhance the learning ability of models with complex data patterns. This approach is also applied to LSTM networks (BLSTM). Bidirectional Long Short-Term Memory Networks have been used successfully in different problem domains such as; speech recognition [28], handwriting recognition [29], sequence tagging [30] and many more.

### 3.2.4. Optimization Methods

As font generation problem is a supervised regression problem, there are a number of choices for optimization. The experiments in this research are performed with several optimization methods.

*Stochastic Gradient Descent (SGD)* (3.1) is a simple method to minimize error on almost any topic in science and engineering. [31] This approach is an iterative and probabilistic method to minimize loss after each batch of data. Almost all deep learning focused optimization methods that are used today are based on this mathematical model. The weakness of the naïve usage of this method is high risk of finding local minima instead of global minima.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}) \tag{3.1}$$

*RMSProp* (3.2) is an extension on *SGD* which is proposed by Geoffrey Hinton in his Neural Networks course [32]. This approach uses a moving average of squared gradients instead of using raw gradient. With this, the step size adapts itself using previous gradients, prevents both exploding or vanishing gradient problem and also decreases risk of settling in a local minima.

$$\begin{aligned} E[g^2]_t &= 0.9E[g^2]_{t-1} + 0.1g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \end{aligned} \tag{3.2}$$

*Momentum* (3.3) is another extension on *SGD*. This extension updates step sizes

with momentum, using previous gradient value, and lets the learning gets faster as it goes. [33]

$$
\begin{aligned}
v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \\
\theta &= \theta - v_t
\end{aligned} \tag{3.3}
$$

*Nesterov Accelerated Gradient* (3.4) peeks ahead in to update step sizes with momentum instead of backward direction. [34]

$$
\begin{aligned}
v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \\
\theta &= \theta - v_t
\end{aligned} \tag{3.4}
$$

*ADAM (Adaptive Moment Optimization)* 3.5 is a combination of *RMSProp* and *Momentum* optimizers, keeps using moving average of squared gradients while adding momentum to have faster training. [35]

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t
\end{aligned} \tag{3.5}
$$

*NADAM* 3.6 a version of *ADAM* optimizer that uses *Nesterov Momentum* instead of *Momentum*. [35]

$$
\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} (\beta_1 \hat{m}_t + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t}) \tag{3.6}
$$

# CHAPTER 4

# METHODOLOGY

This chapter is a summary of computational methods and approaches that are taken in this study. Overall system architecture and internal steps are described, mathematical backgrounds of methods are mentioned. Figure 4.1 is an overview of the suggested system.

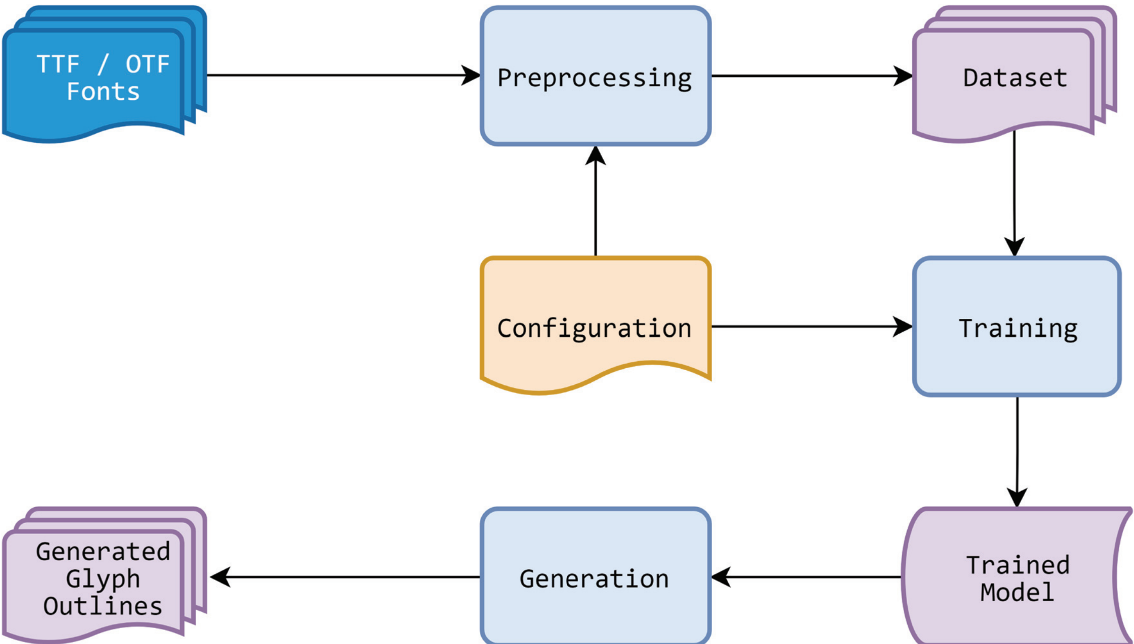



Figure 4.1. The proposed system in this study consist of three modules; Preprocessing, Training and Generation. The input of the system is a set of TTF or OTF fonts and the expected output of the system is an LSTM-generated set of glyph outlines.

## 4.1. Preprocessing

To form deep learning data from *OpenType* and *TrueType* font files, preprocessing steps such as *Parsing*, *Subsetting* and *Formatting* have defined. The primary aim in those steps is to prune all non-relevant data and create learnable sequences. To parse and subset raw font files, *FontTools Open Source Font Manipulation Library* [36] is used.

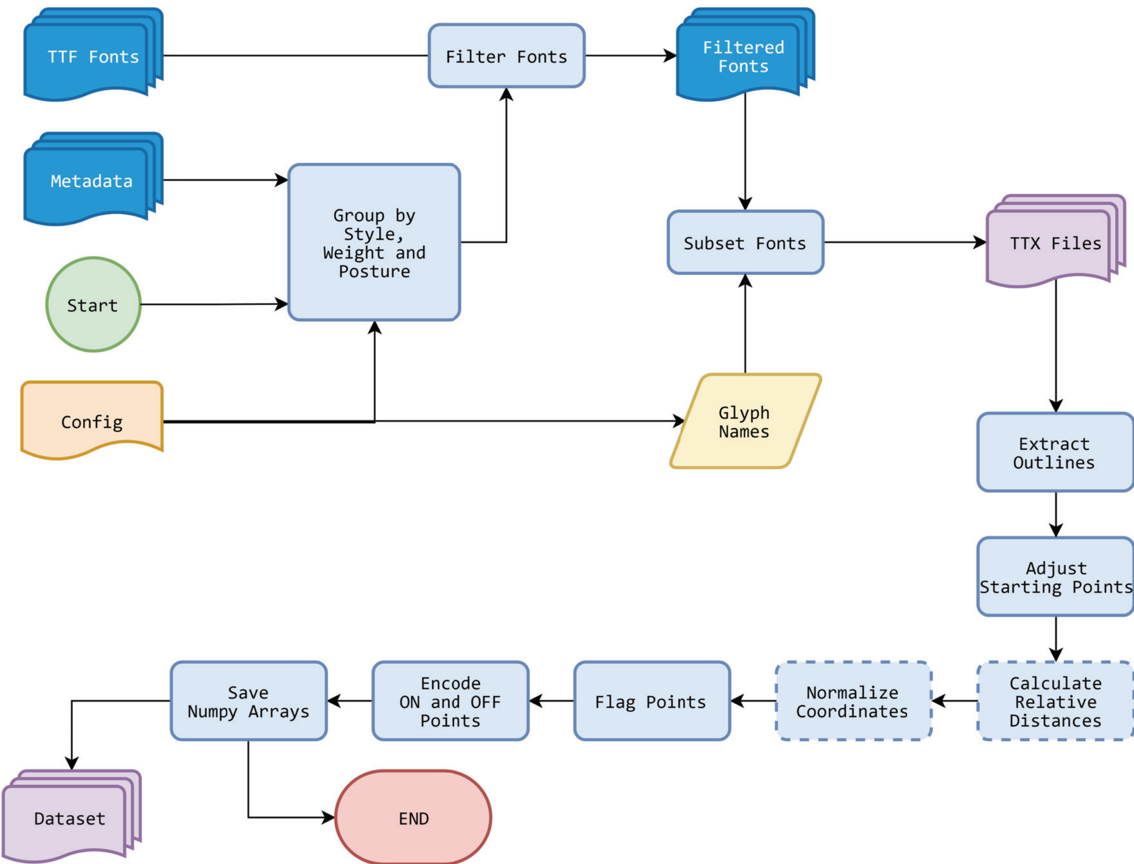



Figure 4.2. In preprocessing, binary TTF fonts are encoded into learnable glyph sequences. The raw fonts are filtered by their characteristics, and their required parts such as specific groups of glyphs are extracted. After filtering, fonts are converted to TTX which is a traversable representation for font data. Their learning features are extracted from TTX and encoded into numeric sequential dataset. Additionally, optional phases such as normalization are also performed in this module.

### 4.1.1. Parsing

*TrueType* and *OpenType* fonts are binary file formats. To be able to extract and use data from them, there is a need for conversion to a non-binary format. *TTX* (*"TT"* for *TrueType* and *"X"* for *XML*) is a human-readable format based on *XML*, that can be encoded from or decoded to *TrueType* binary formats.

*FontTools* provides an internal tool to convert *TTF* files to *TTX* with subsetting options. Created *TTX* files can be parsed and traversed easily to form the dataset.

### 4.1.2. Subsetting

*TrueType* and *OpenType* fonts contain several tables to keep extensive definitions about fonts such as encoding, rendering parameters, hinting and antialiasing guides. The scope of this thesis is limited to glyph outline definitions, so all the definitions except glyph outlines can be excluded from training as they are obsolete.

Furthermore, apart from table subsetting; the glyphs that are not to be trained should also be excluded. `CMAP` table is used to get glyph mappings from `GLYF` table, for each glyph outline definition. Only these two tables are used in preprocessing phase to form a dataset from a single glyph or a group of glyphs.

Another phase of subsetting is filtering by font metadata. *Google Fonts Archive* includes metadata definitions for each font file. Before forming the dataset, fonts files are filtered by their style, weight and posture. This allows focused experiments to be performed.

For experiments in this research, several subsetting parameters are used. In experiments, the model is trained with either multiple glyphs or a single glyph, The filtering of specific glyphs from fonts have been done in subsetting phase. Metadata based subsetting parameters are also changed in different experiments but most of the experiments are made with *Sans-Serif*, *Regular Postured*, *Medium Weight* fonts. The reason of this choice is that, these parameters represent the most common sense fonts.

### 4.1.3. Encoding and Feature Definitions

It is mentioned that all glyph outline definitions are formed from one or more closed shape definitions. These closed shapes have sequences of control and anchor points.

The primary feature of a point in 2-dimensional space is its $x$ and $y$ coordinates. These coordinates are defined in range between 0 and 1000. Scaling and normalization are optionally used for coordinates in this study.

Another feature of a point is the flag if the point is an anchor or a control point. This feature is also explicitly defined in font files and used as an integer-coded boolean in this study.

An important aspect in sequential learning is the knowledge of when a sequence starts and ends. Though, the data is defined for each glyph, glyph outlines have vari-

able lengths and all outlines should be flattened into a single sequence to be useful as LSTM input. Therefore, all points should had been flagged with their state. 4 point states are defined for this purpose; `START_CONTOUR`, `END_CONTOUR`, `DRAW_TO` and `END_GLYPH`. This states are also integer-coded in input sequences.

Though it is not a temporal feature, glyph name is also included in each time-step. Unique integer values are assigned to each glyph and added as a feature of sequences.

As an alternative of integer encoding, one-hot encoding is also experimented for point states and glyph indexes.

### 4.1.4. Adjusting Starting Point

Glyph outline definitions in digital font representations are closed shapes and their starting points are ambiguous. This creates noise in data and affects learned patterns negatively in training. At preprocessing phase, all glyph outlines are adjusted to start from their bottom-left points to assure consistency. Figure 4.3 is an example of this adjustment on uppercase E glyph.

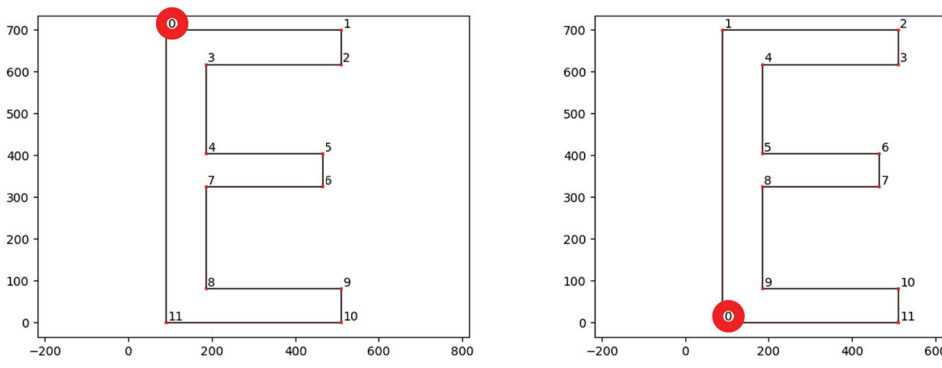



Figure 4.3. Raw glyph with point indexes (Left),are rolled to start from their closest point to origin for consistency (Right).

## 4.2. Training

This section details the training processes of this study and mentions the strategies that are taken to cope with different limitations of learning glyph sequences. An overview of the training module is shown in Figure 4.4.

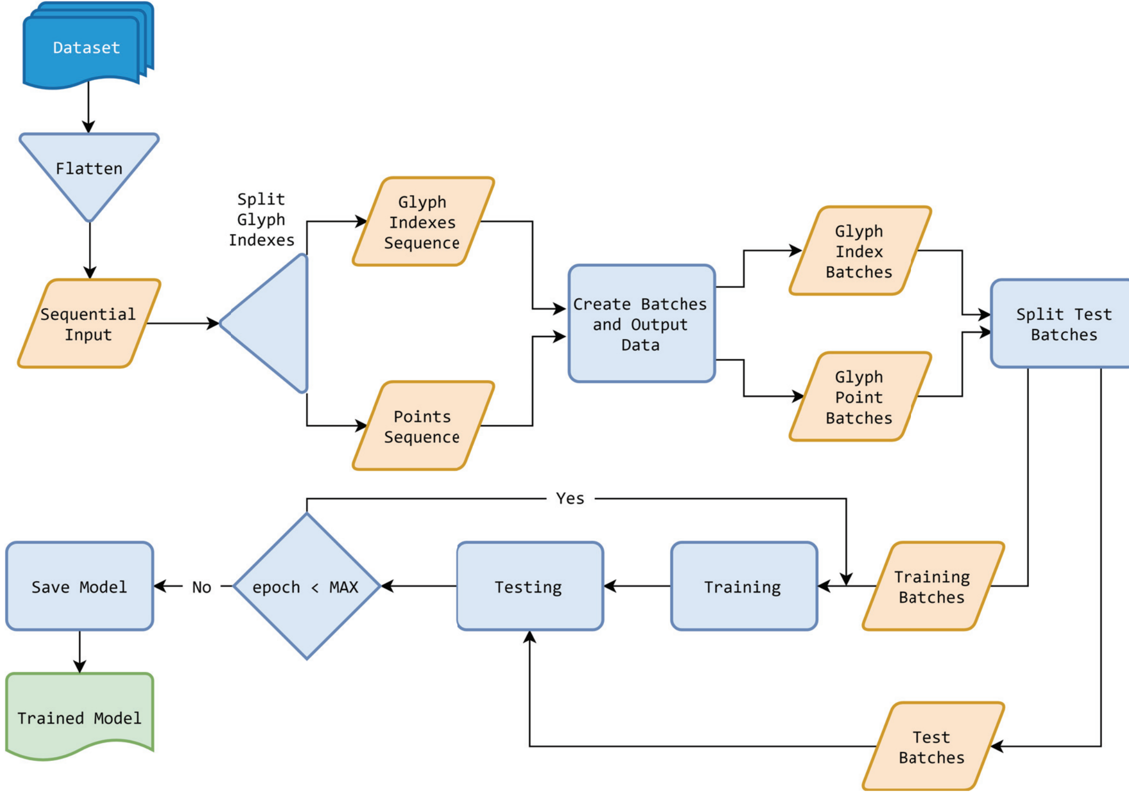



Figure 4.4. The pre-formed dataset is fed into the network after a flattening. Glyph indexes and temporal point sequences from flattened sequence are separated to created batches of data, a part of batches are used for testing, and rest of the batches are fed into the network for training. After the predefined epoch count is reached, model weights are saved to be used in generation module.

### 4.2.1. Input Representation

As mentioned in Section 4.1.3, each point in a glyph sequence represent a single time-step in the sequential model, so there is a need to represent temporal data about the point and also non-temporal data about the glyph in overall.

Table 4.1.: Input representation of font dataset.

| Feature | Type | Values | Description |
|---|---|---|---|
| x-coordinate | int | [0, 1000] | Raw x-coordinate of point in 2D space. |
| y-coordinate | int | [0, 1000] | Raw y-coordinate of point in 2D space. |
| on | bool | {on, off} | Bézier point flag, indicating if the point is *on-curve* or *off-curve* |
| flag | int | {StartContour, DrawTo, EndContour, EndGlyph} | Point flag indicating the role of point for the glyph outline. |
| glyph-index | int | [0, 255] | Non-temporal glyph indicator. Same throughout each glyph outline representation |

5 features for each point are selected to be included in inputs. These features are shown in Table 4.1:

The generated time-steps can be fed into LSTM networks one by one or in batches. Considering that almost all machine learning models work with less features [37], the input representation is designed to have minimum features while keeping all relevant data. This representation provides all the information to define a glyph outline in a simplistic way.

### 4.2.2. Foundational Neural Network Model

A foundational neural network model is defined to make independent modifications for controlled experiments. This model is designed to be as simplistic as possible, and based on models that created successful output to related works in literature. [3]. This model can be defined as a generalized sequence learning and prediction workflow.

Figure 4.5 represents the architecture of the foundational model. Though it is selected as the main architecture for all experiments; in Section 4.2.6, different network architectures are proposed and discussed.

Figure 4.5. The foundational model is a simple 2-layer LSTM network followed by a single dense layer having size as the number of output features.

### 4.2.3. Batching

Batch size is an important hyper-parameter of neural networks. Especially the training and prediction performances of sequential models such as RNNs and LSTMs are highly effected by batch size. Considering sequential models are supposed to learn temporal structure inside data, the clarity and unity of sequence batches is critical.

Glyph outline definitions have unequal lengths. Even for the same glyph, various fonts define different number of points in a single glyph. As the primary intent is to train the model for the purpose of generating complete glyph outlines after training; in training it is crucial to isolate different glyph definitions from each other. Then all internal states in the LSTM model should be cleared after each glyph, to prune all non-relevant temporal information inside model.

In this research, 3 different batching approaches are evaluated.

#### 4.2.3.1. Padded and Masked Sequential Training

In this approach, all the input sequences are pre-padded or post-padded with a specific value to create equal-sized batches. This method guarantees that all batches contain exactly one glyph and all glyphs are contained inside only one batch.

One drawback of this approach is that there is a requirement to define a static batch size. The defined batch size should not be very large for two reasons. The first reason is; when batch size is too big in an LSTM, training time and memory consumption increases dramatically. The other reason is, if a batch contains little information about the sequence and most of the batch size is filled with padded values, learning ability of the model decreases.

Figure 4.6 is an example of pre-padding and post-padding data for static batch size of 20.

To prevent the second issue mentioned in last paragraph, masking logic is used. Getting a batch input, a *Keras* model can ignore specific time-step inputs with a layer called *Masking Layer*. This layer does not let specific time-steps to pass into next layers and keeps LSTM model clean if when is placed before it. In this batching approach, *Masking Layer* is placed between input layer and the first LSTM layer.

```
[321, 65,  1, 1, 5]  [0,   0,   0, 0, 5]  [-1, -1,  -1,-1,-1]
[3,   427, 0, 2, 5]  [123, 123, 0, 1, 5]  [-1, -1,  -1,-1,-1]
[635, 18,  1, 3, 5]  [423, 23,  1, 2, 5]  [-1, -1,  -1,-1,-1]
[0,   0,   0, 0, 5]  [3,   423, 0, 2, 5]  [-1, -1,  -1,-1,-1]
[123, 123, 0, 1, 5]  [575, 48,  1, 2, 5]  [-1, -1,  -1,-1,-1]
[423, 23,  1, 2, 5]  [126, 573, 1, 2, 5]  [-1, -1,  -1,-1,-1]
[3,   423, 0, 2, 5]  [123, 123, 1, 2, 5]  [-1, -1,  -1,-1,-1]
[575, 48,  1, 2, 5]  [123, 123, 0, 3, 5]  [-1, -1,  -1,-1,-1]
[126, 573, 1, 2, 5]  [-1, -1,  -1,-1,-1]  [-1, -1,  -1,-1,-1]
[123, 123, 1, 2, 5]  [-1, -1,  -1,-1,-1]  [-1, -1,  -1,-1,-1]
[123, 123, 0, 3, 5]  [-1, -1,  -1,-1,-1]  [-1, -1,  -1,-1,-1]
[423, 26,  1, 0, 5]  [-1, -1,  -1,-1,-1]  [-1, -1,  -1,-1,-1]
[3,   433, 0, 1, 5]  [-1, -1,  -1,-1,-1]  [0,   0,   0, 0, 5]
[559, 48,  1, 1, 5]  [-1, -1,  -1,-1,-1]  [123, 123, 0, 1, 5]
[553, 23,  1, 1, 5]  [-1, -1,  -1,-1,-1]  [423, 23,  1, 2, 5]
[1,   413, 0, 1, 5]  [-1, -1,  -1,-1,-1]  [3,   423, 0, 2, 5]
[525, 48,  1, 1, 5]  [-1, -1,  -1,-1,-1]  [575, 48,  1, 2, 5]
[423, 23,  1, 1, 5]  [-1, -1,  -1,-1,-1]  [126, 573, 1, 2, 5]
[3,   473, 0, 2, 5]  [-1, -1,  -1,-1,-1]  [123, 123, 1, 2, 5]
[835, 18,  1, 2, 5]  [-1, -1,  -1,-1,-1]  [123, 123, 0, 3, 5]
```

(a) Original Data (b) Post-Padded Data (c) Pre-Padded Data

Figure 4.6. Separate glyphs are originally consecutively located inside data (a). To apply padding-masking logic, the data is either post-padded (b), or pre-padded (c) with a predefined timestep.

#### 4.2.3.2. Point by Point Training

This approach takes inputs in sequences one-by-one, having batch size exactly 1.

The strength of this method is keeping original dataset clean, without needing any padding or masking logic, feeding data into network as-is. All glyphs are concatenated to form a single sequence and are fed into the network.

The flaw of this approach is having a big number of gradient updates, as the model performs back-propagation for each batch of data. This increases the computational time needed for each epoch dramatically. Also having large number of gradient updates may lead to overfitting in network. Furthermore, in consequence of feeding temporal data point by point, the network does not learn long temporal structures in data. Instead it learns point to point mappings which is inferior for the generative aims of this research.

To make the network to learn long temporal structure in data, internal states of LSTM nodes should be cleared manually in this approach. To perform it, when a point that is flagged as `END_GLYPH` is fed into the network, the implementation should reset LSTM internal temporal states. *Keras* has a feature called *Stateful LSTM* for this purpose. This LSTM nodes, does not clear their internal states after each batch, instead they wait for a method called `reset_states` to perform internal state clearing. In the implementation, only after a point with `END_GLYPH` flag is processed, `reset_states` method is called. Though, this feature prevents point-by-point learning, the computational burden is still high.

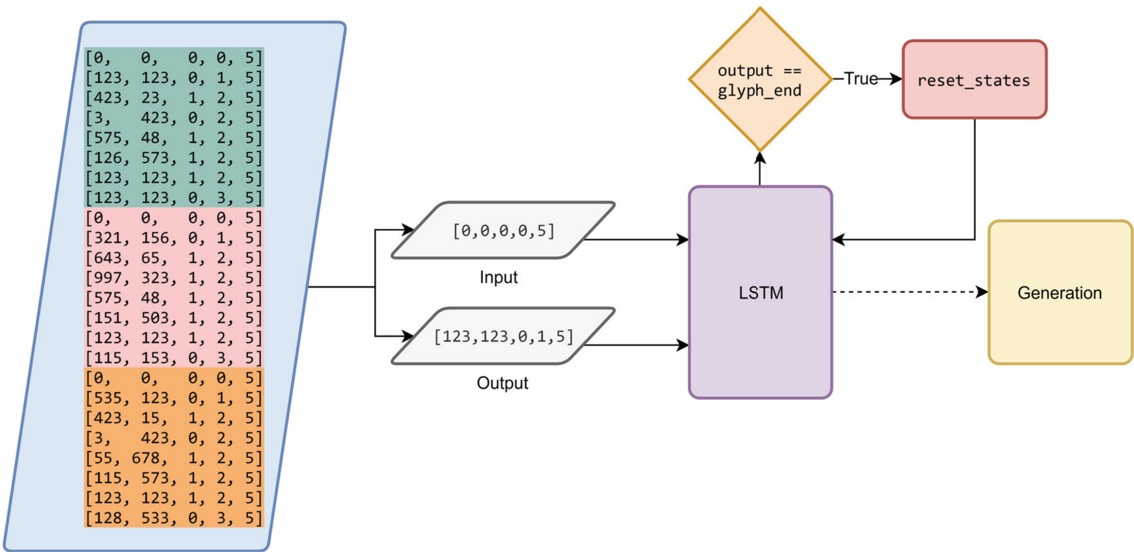



Figure 4.7. Flowchart of point-by-point training. With this approach, the network processes every point as the input and expects a single output for each gradient update.

#### 4.2.3.3. On-line Training

A more sophisticated approach for batching problem is using dynamic batch sizes [38]. This idea is mainly preferred for *On-line Learning* with LSTM networks, since *on-line data* characteristically have arbitrary sizes. Having similar properties, glyph outlines are also compatible with this approach. Using this idea for glyph outlines, the input can remain as-is and the data can be fed into networks in self-contained full sequence batches in various lengths.

In the implementation for on-line training, LSTM models are created without specifying batch sizes. Glyph outline sequences are kept as-is and each batch is defined as a sequence of points for one glyph. In training, each input and output sequence is fed into the network variable-sized batches.

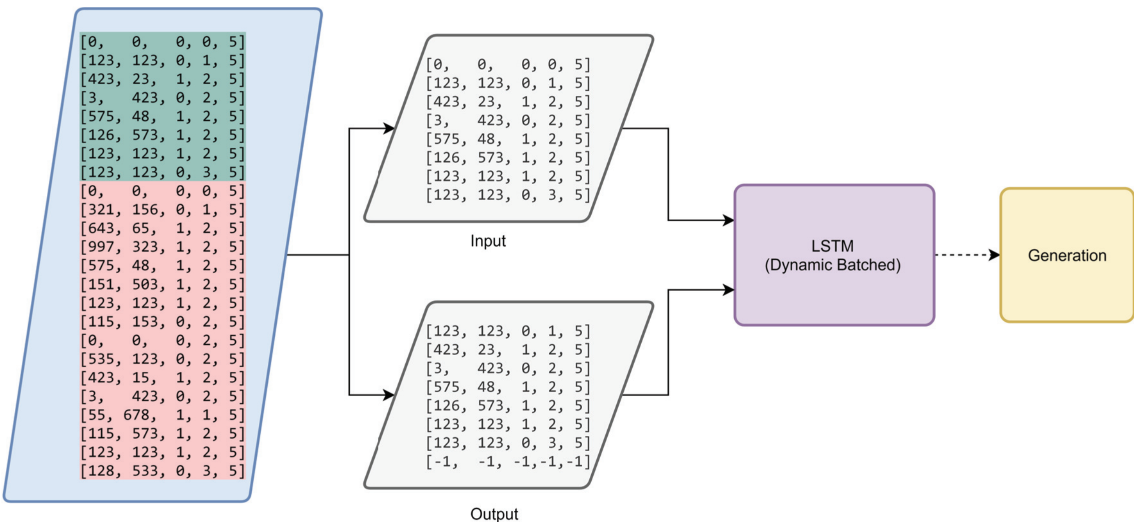



Figure 4.8. Flowchart of dynamic batch-sized training. Having batches per glyph, the input and output are self contained with only required information to be learned by the model.

### 4.2.4. Learning Rate Strategies

Learning rate is another important parameter for LSTM training. The optimal value of learning rate primarily depends on the network size, data size and feature count for each time-step. The learning ability of LSTM model is very sensitive to learning rate

while training. To find optimal learning rate to generate typefaces, several approaches have experimented in this study.

#### 4.2.4.1. Static Learning Rate

The typical way to define learning rate in a deep learning model is to specify it as a constant value. The selection of learning rate is done manually for each training experiment and, cumulative losses after each epoch is tracked to measure performance of the selected learning rate. In this experiment, using high learning rates such as `0.1` or `0.01` are performed well in initial epochs of training but the losses are converged to relatively high values in later epochs. These experiments are failed because of not finding acceptable loss values and generative results. Contrarily, low learning rates such as `0.00001` have converged to lower loss values on the long run, but the number of epochs, hence the need of time to train the network became higher.

#### 4.2.4.2. Decaying Learning Rate

To prevent both early convergence and long learning times, a combined approach is performed. In this approach, an initial learning rate is defined at the beginning of training and it decreased periodically after a specific number of epochs. This approach requires two more parameters to be defined alongside learning rate; *decay rate* and *decaying period*. Using dynamic learning rate by decaying logic, training times are optimized and final loss values have stayed low.

### 4.2.5. Data Encoding

In preprocessing phase (Section 4.1), several ways to create input data are defined. To find an optimal way of representing glyph outline data, different adjustments have been made in preprocessing phase. The adjustments are grouped per feature in this section.

#### 4.2.5.1. Coordinates Encoding

Glyph anchor/control point coordinates are the most important feature of glyph outlines that should be trained to LSTM in this research. In the raw TrueType dataset that is used, this feature is generally in range: $(x : [0, 1000], y : [0, 1000])$ and both $x$ and $y$ are represented as integers. Since the range of coordinate features are relatively very large compared to other features in input data that is mentioned at Table 4.1, it may dominate other features. Interestingly, because the importance of coordinate features are also relatively high, this may have positive effect to expected training and generation performance in this research. To find the most optimal coordinate representation for research, 6 different encodings are experimented.

```
[0,     0,     0, 0, 5]
[123,   123,   0, 1, 5]
[300, -100,   1, 2, 5]
[-420, 400,   0, 2, 5]
[572, -375,   1, 2, 5]
[-449, 525,   1, 2, 5]
[-3,   -450,  1, 2, 5]
[682,  20,    0, 3, 5]
```

(a) Raw Coordinates

```
[0,     0,     0, 0, 5]
[12.3, 12.3, 0, 1, 5]
[42.3, 2.3,   1, 2, 5]
[0.3,  42.3, 0, 2, 5]
[57.5, 4.8,   1, 2, 5]
[12.6, 57.3, 1, 2, 5]
[12.3, 12.3, 1, 2, 5]
[80.5, 14.3, 0, 3, 5]
```

(b) Min-Max Scaled Coordinates [0, 100]

```
[-0.98,-0.94, 0, 0, 5]
[-0.54,-0.31, 0, 1, 5]
[0.55, -0.82, 1, 2, 5]
[-0.97, 1.25, 0, 2, 5]
[1.09, -0.69, 1, 2, 5]
[-0.53, 2.03, 1, 2, 5]
[-0.54,-0.31, 1, 2, 5]
[1.93, -0.20, 0, 3, 5]
```

(c) Z-Score Normalized Coordinates

```
[0,     0,     0, 0, 5]
[123,  123,  0,  1,  5]
[423,  23,    1,  2,  5]
[3,    423,  0,  2,  5]
[575,  48,    1,  2,  5]
[126,  573,  1,  2,  5]
[123,  123,  1,  2,  5]
[805,  143,  0,  3,  5]
```

(d) Relative Distances

Figure 4.9. Point coordinates are used as-is (a), Min-Max scaled (b), Z-Score Normalized (c), or as relative distances from previous point (d).

**Raw Coordinates** Keeping coordinates as-is and training the model is the simplest approach without any requirement of data preprocessing.

The downside of this approach is, when raw data is fed into the model, model becomes very responsive to outliers. Since in this case an outlier is not necessarily a single noisy point, it can be a full sequence too. To eliminate those outliers, or at least reduce their effects on the learning, preprocessing should be done on coordinate features.

**Scaled Raw Coordinates** A simple modification on raw glyph coordinates is scaling their value in a specified range. Scaling stands for downscaling in this context, because the aim is to reduce the range of coordinate data. While this approach narrows the range of coordinates, the scaled data still have the outliers, only in a different scale.

$$X_{sc} = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{4.1}$$

Experiments have been made using *Min-Max Scaling* Equation (4.1), and the generation results were similar to the results of raw coordinates.

**Normalized Raw Coordinates** Normalization is a powerful method to both downscale coordinate data and also to eliminate outliers. To normalize coordinate values in experiments, *Z-Score Normalization* Equation (4.2) has been used. In this method, the expected range is also defined but the scaling logic is more sophisticated. Z-Score uses standard deviations and means to adaptively scale data. This helps to reduce the effects of outliers in data, also downscaling can be done at the same time.

$$z = \frac{x - \mu}{\sigma}. \tag{4.2}$$

**Relative Coordinate Differences** One other approach that is preferred in related researches [3], [39] is altering raw coordinates to relative differences. In this approach, a glyph starts at a predefined point in coordinate system (e.g. $x = 0, y = 0$), and for all the following points, coordinate features are their relative distances from their predecessors. This approach is widely used because it is a more natural representation of *drawing* data, also the range is smaller and zero centred.

**Scaled Relative Coordinate Differences** Relative coordinates have a smaller range than raw coordinates. But still, though they are not very common, long jumps between points have to large values in data. Similarly as it is done in raw coordinates; to test the

effectiveness of range scaling on learning performance, scaling also applied to relative coordinate differences in experiments.

**Normalized Relative Coordinate Differences** To equalize the relative point distances from different glyphs to the same range, *Z-Score* normalization is also applied to relative coordinate differences. One situation to remark here is; the relative coordinates of glyph outlines are either positive or negative and it is vital to keep those properties of data not to deform training and generation.

By definition, *Z-Score* normalization can convert a negative value to positive or vice versa because it scales every data into a zero-centred range. However, relative distances from glyph outlines are not exposed to this risk. Since all the relative distances form a closed shape, the mean of all relative distances in each glyph is exactly zero.

#### 4.2.5.2. Point Flag Encoding

To specify behavioural flags of points, 4 different states: (`START_CONTOUR`, `DRAW_TO`, `END_CONTOUR`, `END_GLYPH`) are defined. Considering the number of these flags are relatively low, integer encoding is selected as the encoding of this feature, at the first sight. Although, as this feature is categorical, one-hot encoding is also tried in the experiments.

#### 4.2.5.3. Glyph Index Encoding

Glyph indexes are non-temporal categorical features of the typeface dataset in this research. Glyph index feature is only meaningful while training networks with multiple glyphs. For per-glyph trainings, this feature is obsolete.

To encode glyph indexes; both integer encoding and one-hot encoding is used. However, unlike point flags, glyph indexes have a relatively big range. Integer encoding has a potential of underperformance in big ranges of data. Thus, different network architectures to use one-hot encoding are defined in Section 4.2.6.

### 4.2.6. Alternative LSTM Architectures

Though most of the experiments are performed using the foundational model declared in Chapter 5, several network architectures are also tried. Most of the adjustments on the foundational network architecture is made on input layers.

Foundational model is not compatible with one-hot encoded categorical data. To represent categorical glyph index feature, integer encoding should had been chosen. Input layer and the architecture of the foundational model is shown in Figure 4.10.

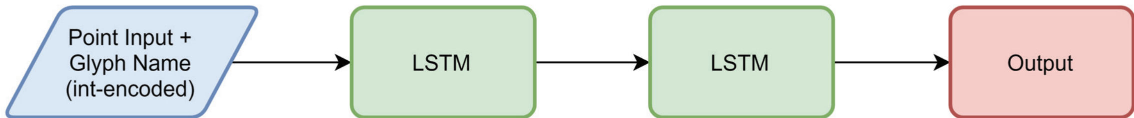



Figure 4.10. Input layer and architecture of foundational model

As an alternative; input layer of the foundational model is modified to feed different features of sequential time-steps in separate input layers. Input layer modifications are primarily developed to feed one-hot encoded features alongside continuous features in a disjoint structure. The main motivation to separate one-hot encoded data from other data is to prevent domination of one-hot data which require large number of input slots when they are fed into the network. Having separate input gates for continuous data and one-hot vectors, enables the capability of weighting the effect of individual features and preventing domination. Two alternative LSTM architectures are shown in Figure 4.11 and Figure 4.12.

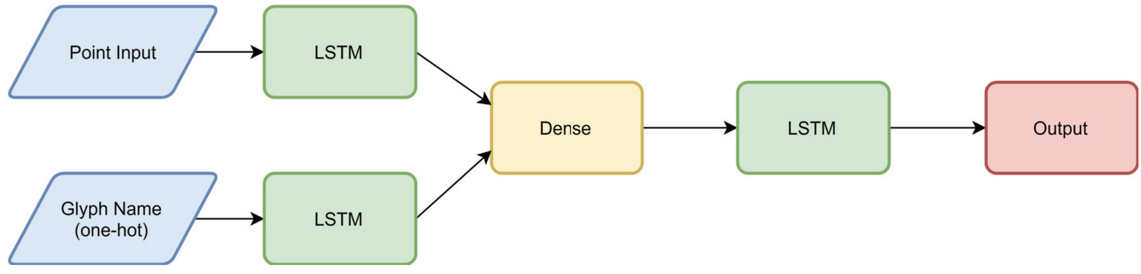



Figure 4.11. The first alternative architecture separates temporal data from categorical glyph name data using separate input layers and dedicated LSTM nodes for each input. Outputs of these two LSTM nodes are concatenated into a dense layer, then fed into a common LSTM node.

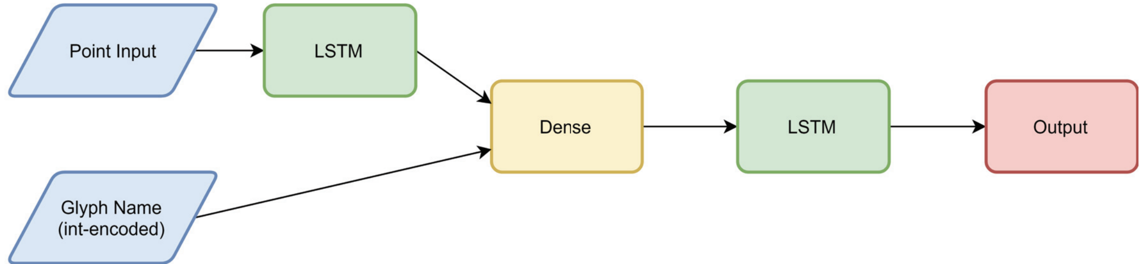



Figure 4.12. In second alternative architecture, only temporal features have a dedicated LSTM node. Glyph name feature is merged with that LSTM node with a dense layer, then fed into common LSTM node.

## 4.3. Digital Font Generation

Predictive features of LSTM networks are extensively used for generating sequences. To generate glyph outlines with LSTM networks, training with digital font dataset that is defined in Section 4.1 is used. After the model learns the temporal structure inside glyph representations, generation has performed using predictive features of the network using the trained model. Figure 4.13 is a visual summary of the generation module and its components.

## 4.4. Software Infrastructure

Long Short-Term Memory networks are used in different compositions in this research. Several hyper-parameters such as *learning-rate*, *hidden layer size*, *dropout*, different optimizers and loss functions are used. Also, experimental network architectures are tried in this project. To prototype models with these large number of variations, additional libraries and frameworks are used in this study.

To create models and and run the experiments, *"Keras: Python Deep Learning Library"* [40] is used. This library is a deep learning frontend that uses several backend frameworks for computation. For computations on *Nvidia* GPUs, *"Tensorflow Open Source Machine Learning Platform"* [41] and *"Nvidia CUDA Parallel Computing Platform"* [42] is used. For computations on other GPU and CPUs, *"PlaidML Portable Tensor Compiler"* [43] is used. The code in this study is written merely using *Keras*. The two backends mentioned, provided same results with a single Keras implementation.

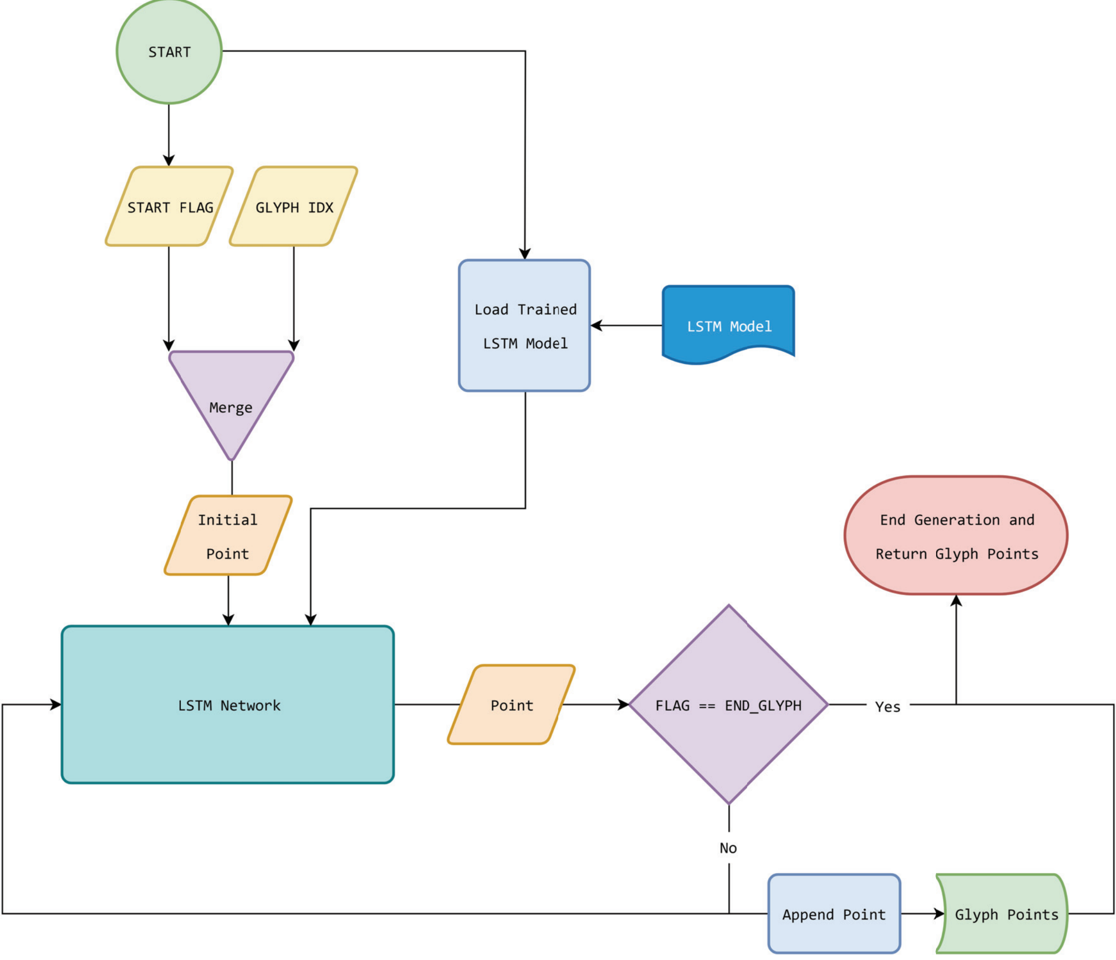



Figure 4.13. A static START_CONTOUR input with no specified point coordinates is fed to the network alongside a GLYPH_INDEX, then the network starts predicting next possible points of generated glyphs. All predicted points are passed through a sampling process, then fed again into the network together with the sampled point to predict the next point in glyph. While training data is designed to contain point states such as START_CONTOUR, END_CONTOUR or END_GLYPH, the prediction ends at a point when it predicts a point with END_GLYPH flag. At this point, the shape that is created is closed by drawing a line or a curve (depending on the ON state) to the starting point of the current contour.

# CHAPTER 5

# EXPERIMENTS

This chapter presents the results of the experiments in this research and discussions on performances of different parameters in controlled experiments. The performance criterion and their examples are also detailed in this chapter.

## 5.1. Criterion on Generative Performance

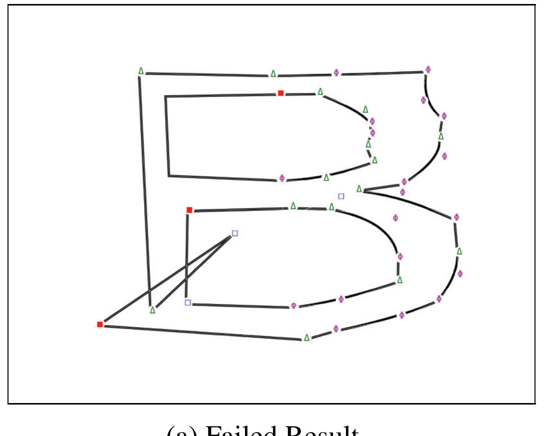
Generation results are classified as successful or failed by several qualitative and quantitative conditions. Even when a generation result passes all the quantitative conditions of success, it should be visually seen to check if it represents the expected glyph properties.

### 5.1.1. Overlaps

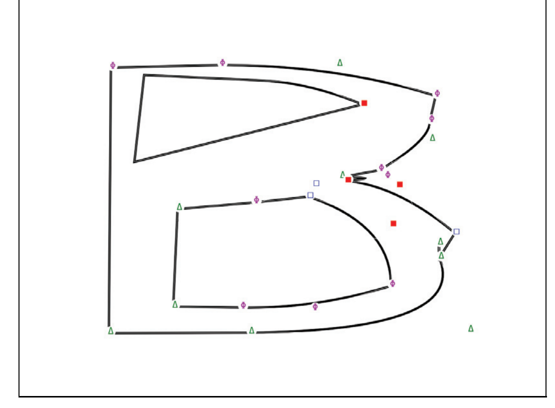
The first condition to success of a generation result is the inexistence of overlaps. An overlap of two curves or lines cannot exist on a font glyph by definition of being formed of closed-shapes. A generative result with any number of overlaps is classified as a failed result. Figure 5.1 is an example of the usage of this criteria.

### 5.1.2. Closing Shapes

A font glyph is a set of closed shapes, thus if a generation result has any non-closed shapes it is classified as a failed result. This criteria eliminates most of premature generation results especially in undertrained models. An example of this criteria is shown in Figure 5.2.



(a) Failed Result



(b) Successful Result

Figure 5.1. Overlap criteria is the most common way of filtering failed results in this study. The failed result (a) is a qualitatively good representation of glyph B, however, having a single wrong point prediction on its bottom-left, it becomes a failed experiment having an overlapping line. Contrarily, while the successful result (b) is a more primitive representation of glyph B, it does not contain any overlaps, thus it's a successful experimental result for this criteria.

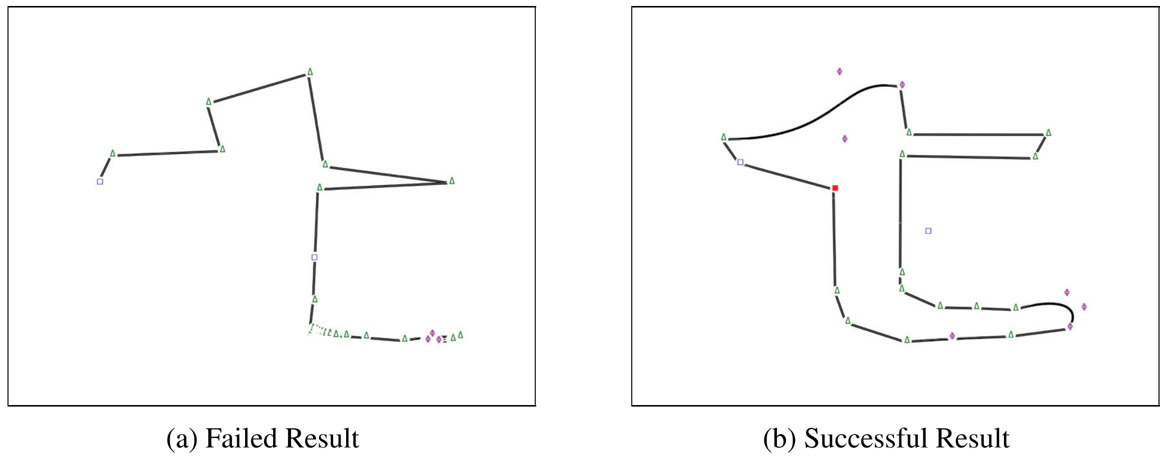
### 5.1.3. Contour Count

For almost every glyph, the number of contours are static. For example C has only one contour, A has two contours and B has three contours. Any generation result that does not contain that specified number of contours are classified as failed. Figure 5.3 is an example of this criteria on glyph A.

## 5.2. Experimental Results

The experiments are performed per glyph and per glyph groups in this study. In per glyph training, only one glyph is used to train the network. Contrarily in per group training, a pre-defined set of glyphs are fed to the same network and that group of glyphs are expected as output. Glyph groups are can be a full character-set such as an alphabet, or can be a pre-defined group of glyphs that are formed by their characteristics.

Two main criteria are selected for characteristic glyph grouping. The first criterion is *Outline Count*. Learning outline finishing points is an important indicator of generative



(a) Failed Result

(b) Successful Result

Figure 5.2. The failed result (a) is a premature generation result from glyph `t`. In later epochs, the successful result (b) is generated from the same model.

performance of *point state* feature. The other criterion is the *presence of curves* in glyphs. This feature is an indicator of generative performance of *anchor point* and *control point* learning.

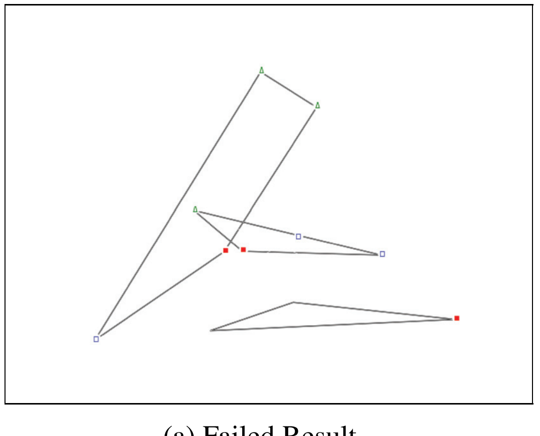
In per glyph training, the fifth timestep feature *glyph index* becomes obsolete, thus the dimensionality of input data is reduced. Also, since the variation of data is decreased, the network becomes more focused and specialized to generate a specific glyph. In contrast, in per group training the network becomes more generalized and generates glyphs in similar styles.

### 5.2.1. Per Glyph Training

Training a single network for one glyph is a simple approach that reduces dimensionality and the variance of both input and output data of the network. Samples from results of several per-glyph experiments are shown in this section.

Figure 5.7 shows the training results of glyph `A` with generative evolution through epochs. In this experiment, a 2-layer LSTM network is trained with 10 *monospace* fonts. The coordinates are normalized with *Z-Score* normalization and dynamic batching is used.

Figure 5.8 shows the training results of glyph `e` with generative evolution through epochs. This experiment is done in exactly same environment with the previous experiment, trained by 10 *monospace* fonts using a 2-layer LSTM network. The coordinates are


(a) Failed Result

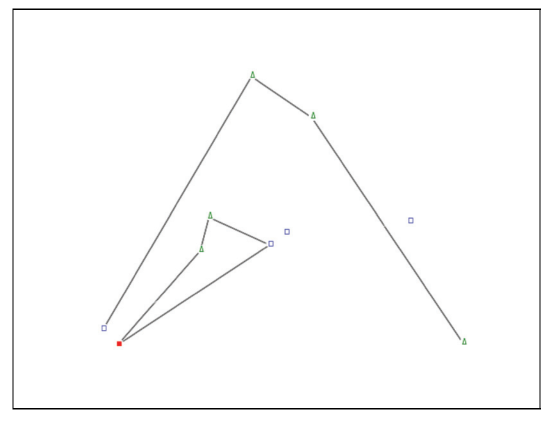

(b) Successful Result

Figure 5.3. The failed result (a) has 3 contours which is not the expected contour count for glyph `A`. However, having 2 contours the successful result (b) passes this condition for glyph `A`, regardless of its dissimilarity with the expected glyph shape.

also normalized with *Z-Score* normalization and dynamic batching is used.

As mentioned in Section 4.3, in generation, every predicted point is passed through a sampling phase and fed again into the network. With this approach, the generated glyphs from the same network slightly vary in style. An example for generative variance is given in Figure 5.4.

The details of the model that is used to generate single glyphs are given in Table 5.1.

### 5.2.2. Per Glyph Group Training

To be able to generate multiple glyphs from a single model, training with groups of glyphs is needed. Unlike single glyph training, `GLYPH_INDEX` feature which denotes different glyphs is instrumental for this approach. In this section, samples from generative results of different groups are shown. The groups are either formed by glyph characteristics or randomly. A complete character set or an alphabet can also be selected as a group, however in that case, the variance of data becomes very large and training requires a larger dataset. Figure 5.5 and Figure 5.6 are generation results of two characteristically merged groups of glyph training.

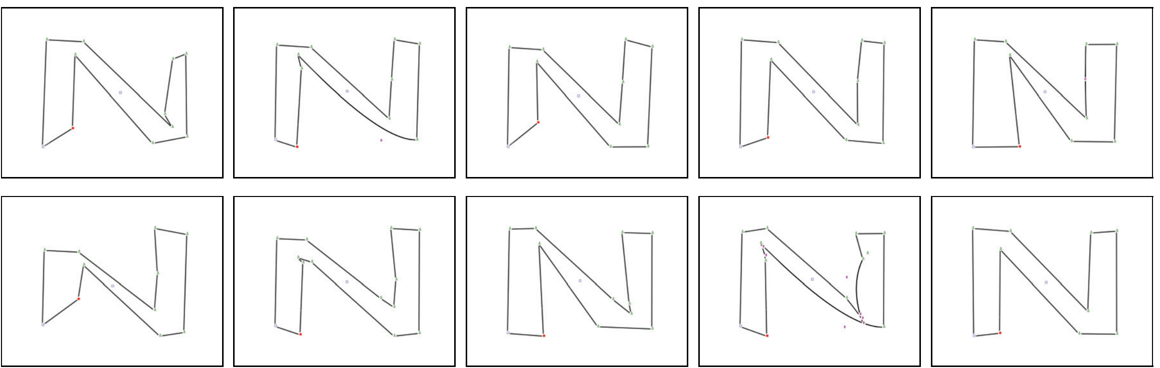


Figure 5.4. Generative outputs of glyph N from a single model. The styles of the glyphs vary in different generations.

Table 5.1.: Model parameters for single-glyph training.

| Parameter | Value |
|---|---|
| Glyph Style | Monospace |
| Glyph Weight | 400 |
| Glyph Posture | Regular |
| Number of Glyph Samples | 10 |
| Model Architecture | Foundational Model |
| Optimizer | RMSProp |
| Learning Rate Strategy | Decaying (%10 in every 50 epoch) |
| Initial Learning Rate | 0.001 |
| Layer Type | Bidirectional LSTM |
| Coordinate Encoding | Normalized Raw Coordinates (Z-Score) |
| Point State Encoding | Integer Coded |
| Batching Strategy | Dynamic Batching |

## 5.3. LSTM and BLSTM Layers

As mentioned in Section 3.2.3, Bidirectional LSTMs create favourable results when successive time-steps of sequences are available in training data. This requirement is a root of incompatibility of BLSTMs with several problems such as on-line learning or real-time forecasting. Unlikely, glyph outline data in this study contains full sequences of trainable data and is compatible with BLSTM model. In experiments, BLSTM layers are used for comparison with LSTM layers. The results of these two approaches have meaningful contrast with different superiorities in dissimilar setups. A sampled performance comparison between LSTM and BLSTM layers is given in Table 5.2.

Table 5.2.: Performance comparison between LSTM and BLSTM layers. In the first sample h; Despite both generating visually almost identical results, the selection of LSTM layer have a significant superiority over BLSTM layer for all criterion and also in converged loss values. On the other hand, in the second glyph A, BLSTM layers have significantly better ratios in every criterion and also in loss values. Furthermore the placement of the inner contour is visually better in the BLSTM sample. In the last example a, the performances of both preferences are similar with a slight superiority of BLSTM layers.

| Glyph | LSTM | | BLSTM | |
|---|---|---|---|---|
| h | Success: 40/100<br>Contour: 65/100<br>Overlap: 40/100<br>Closing: 94/100<br>Loss: 0.0799 | | Success: 22/100<br>Contour: 42/100<br>Overlap: 22/100<br>Closing: 53/100<br>Loss: 0.0825 | |
| A | Success: 23/100<br>Contour: 34/100<br>Overlap: 24/100<br>Closing: 65/100<br>Loss: 0.1053 | | Success: 65/100<br>Contour: 72/100<br>Overlap: 67/100<br>Closing: 83/100<br>Loss: 0.0692 | |
| a | Success: 13/100<br>Contour: 24/100<br>Overlap: 32/100<br>Closing: 69/100<br>Loss: 0.0514 | | Success: 16/100<br>Contour: 56/100<br>Overlap: 44/100<br>Closing: 61/100<br>Loss: 0.0546 | |

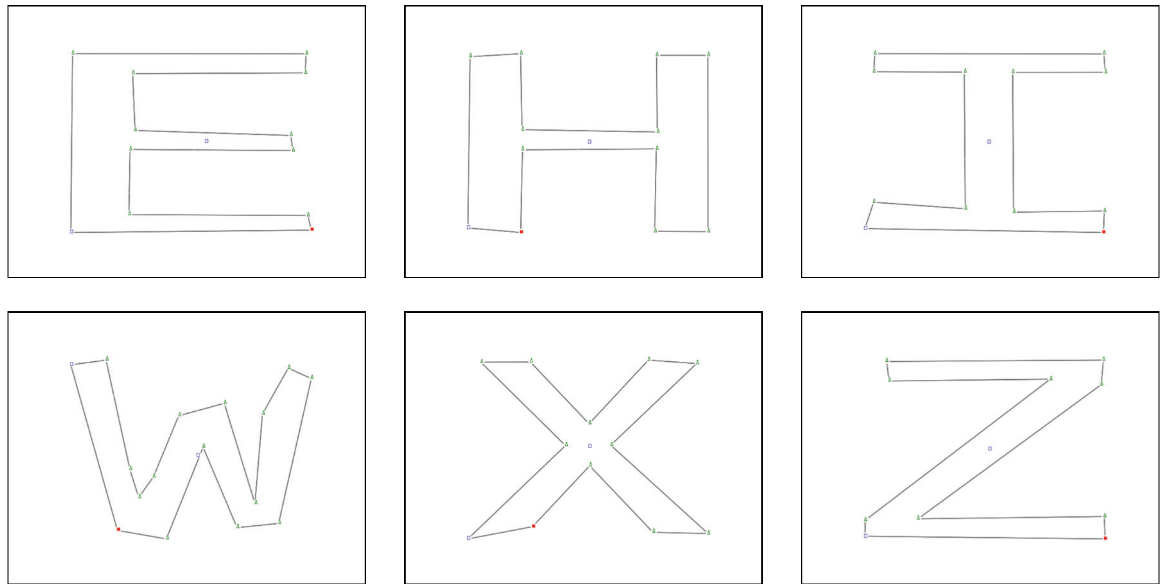


Figure 5.5. Generative results of `[E, H, I, W, X, Z]`. This set of glyphs represent the *non-curved* and *single outline* glyphs from Latin uppercase letters. These 6 glyphs are used to train a model with their integer-coded glyph index features. The model succeeded to discriminate these glyphs and created these results from different starting glyph flags in generation module.

### 5.3.1. LSTM Layers

In experiments, as shown in Table 5.2, the choice of using LSTM layers have created generally high-quality results on large sets of data and low learning rates.

Additionally; compared to BLSTM layers, LSTM layers have inferior ability of learning relatively longer sequences. Yet, LSTM layers work well in short sequences; in this case for simpler glyphs such as **I** and **C**. As LSTM layers learn data in single direction, these layers have failed learning and generating more complex glyphs such as **E**. Figure 5.9a shows an example of a generative result of glyph **E** from a model with LSTM layers. As can be seen, the model has failed to generate the whole glyph. Instead it failed at a point where the glyph have a similar pattern as the previously learned part of the glyph. Instead of continuing generation of remaining parts of the closed shape, it predicts an identical pattern as that previous pattern repeatedly. The awareness of the forthcoming time-steps in the glyph may prevent this situation.

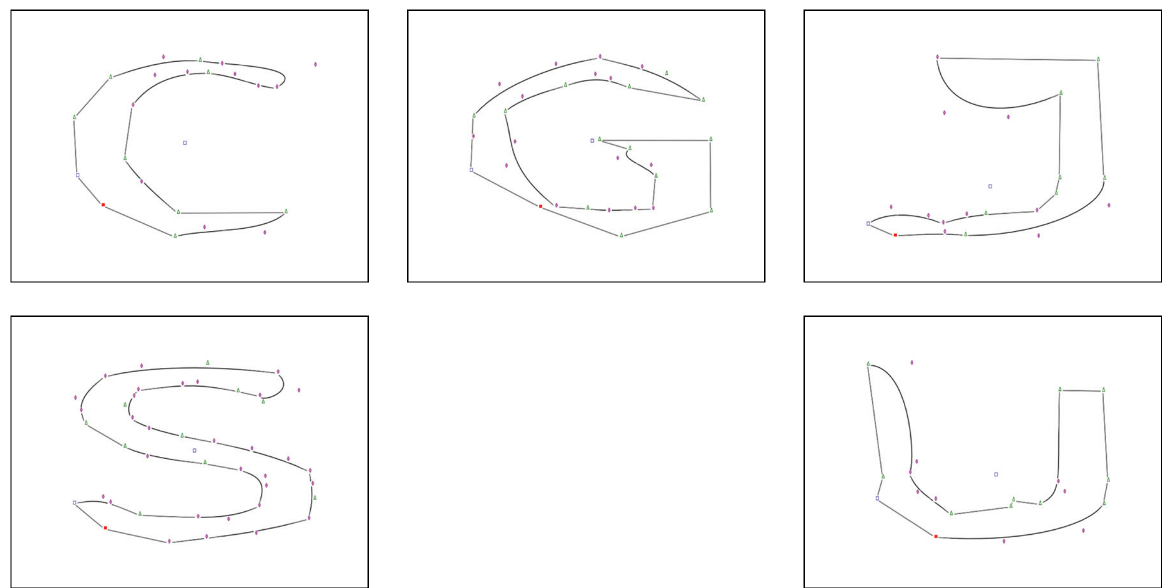


Figure 5.6. Generative results of [C, G, J, S, U]. This set of glyphs represent the *curved* and *single outline* glyphs from Latin uppercase letters. This model also succeeded to discriminate glyphs and created diverse results from different starting glyph flags in generation module.

### 5.3.2. BLSTM Layers

As mentioned in Section 3.2.3, BLSTM layers learn patterns in data from two directions. This ability of BLSTM layers, allows the model to make predictions of a specific time-step; not only in consideration of previous patterns, but also forthcoming patterns.

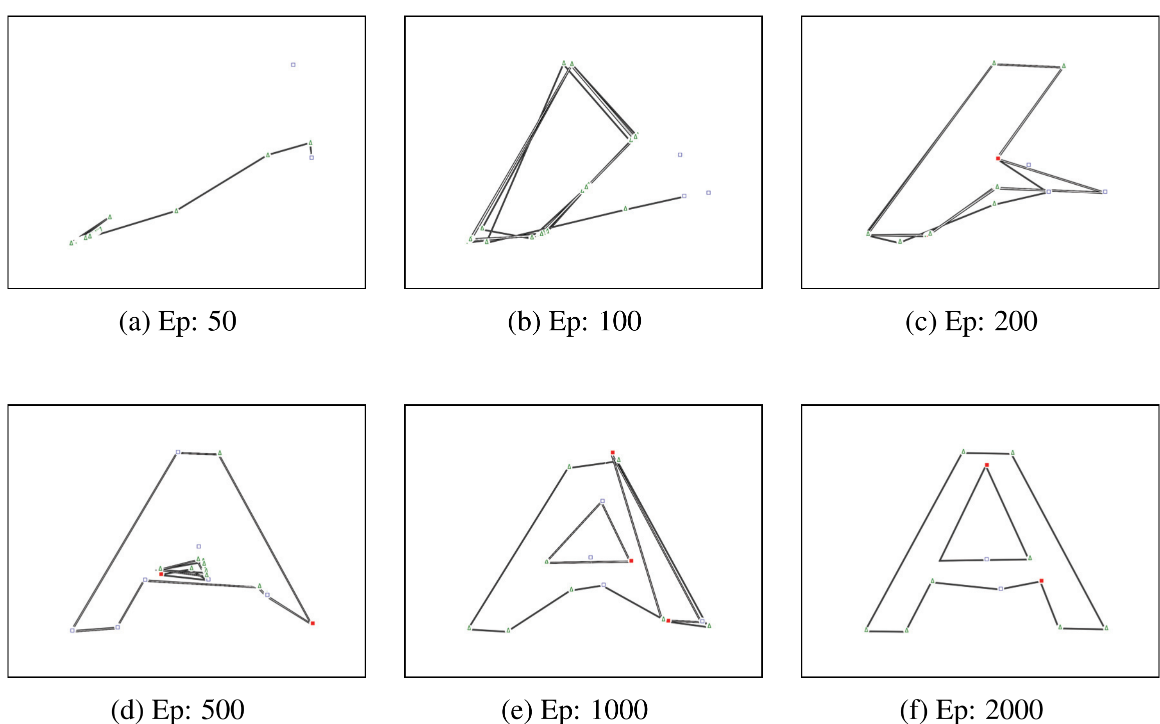
For glyph generation task, BLSTM layers are especially eligible in the case of repetitive patterns in trained glyphs. Nevertheless, the experiments have demonstrated that, the use of BLSTM layers does not contribute to a significant improvement when training glyphs that do not have non-repetitive patterns, such as **h** and **a** which are shown in Table 5.2. Still, while BLSTM networks have not significantly underachieved, compared to LSTM layers in general, they become favourable in a generic model for the purpose of consistency and broadness.

Table 5.3.: Performance comparison between static and decaying learning rates. Experiment results on 3 glyphs are provided in this table. The first glyph f is a single-contour curved glyph and the results of controlled experiments on this glyph shows that; while the preference of using decaying learning rate increased the ratio of successful generations, it also reduced the success rate of samples with correct number of contours. Also the visual comparison of samples can be interpreted that, decaying learning rate created an over-generalized version of that glyph. Contrarily; for the second glyph N, decaying learning rate created higher quality generation results with also visually better representation of the glyph. In the last experimental sample E; Decaying learning rate had significantly improved the quality of the generation results, both in criterion rates and also visually.

| Glyph | Static Learning Rate | | Decaying Learning Rate | |
|---|---|---|---|---|
| f | Success: 3/100<br>Contour: 65/100<br>Overlap: 8/100<br>Closing: 72/100<br>Loss: 0.1072 | | Success: 7/100<br>Contour: 34/100<br>Overlap: 16/100<br>Closing: 75/100<br>Loss: 0.0963 | |
| N | Success: 71/100<br>Contour: 74/100<br>Overlap: 72/100<br>Closing: 92/100<br>Loss: 0.0625 | | Success: 82/100<br>Contour: 87/100<br>Overlap: 82/100<br>Closing: 98/100<br>Loss: 0.0205 | |
| E | Success: 59/100<br>Contour: 75/100<br>Overlap: 54/100<br>Closing: 74/100<br>Loss: 0.0749 | | Success: 97/100<br>Contour: 97/100<br>Overlap: 97/100<br>Closing: 100/100<br>Loss: 0.692 | |

## 5.4. The Effect of Learning Rates

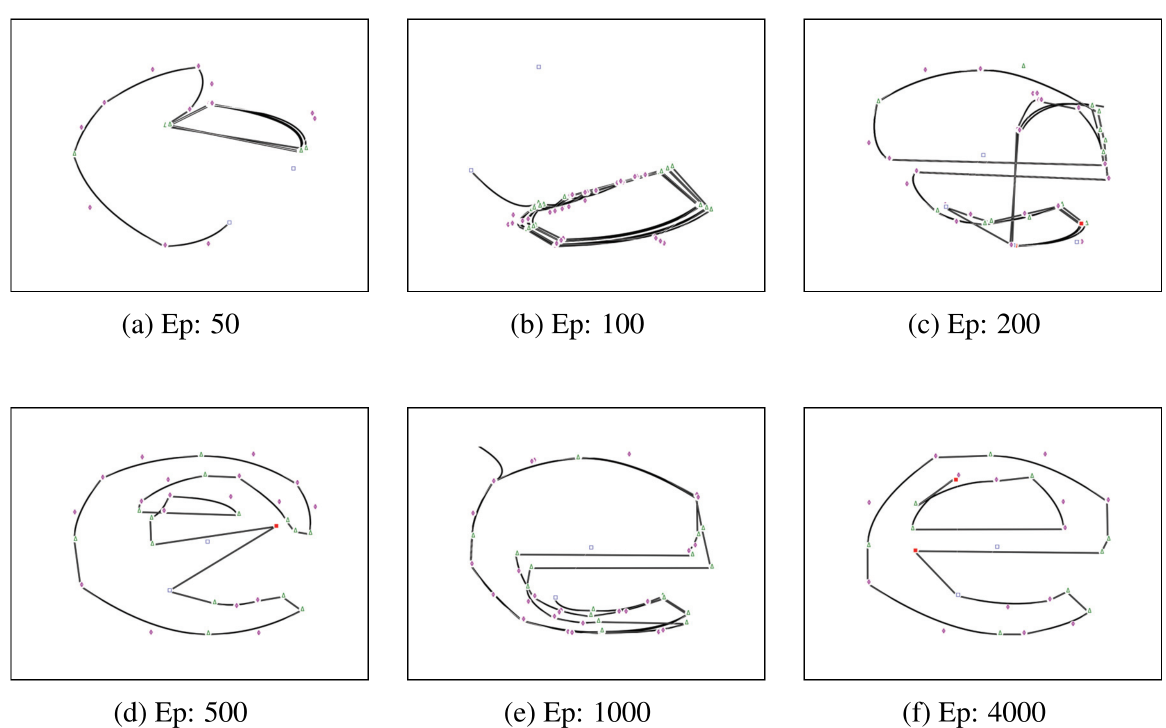
Apart from the effects of switching LSTM nodes with BLSTM nodes, the strategy of defining learning rates had significant effects on generative performances of models. As seen in Table 5.3, usage of decaying learning rates had positive effects on learning performances, compared to static learning rates. Models that are trained with decaying learning rates had converged to lower loss values than models with static learning rates. Also, decaying learning rates reduced existence of overlaps which have been the most common failing criteria among all criterion that are described in Section 5.1. Nevertheless, for several glyphs such as f, decaying learning rates also led to over-generalization and dropped success rates of *contour count* criteria, despite increasing overall success rates.



(a) Ep: 50 (b) Ep: 100 (c) Ep: 200

(d) Ep: 500 (e) Ep: 1000 (f) Ep: 2000

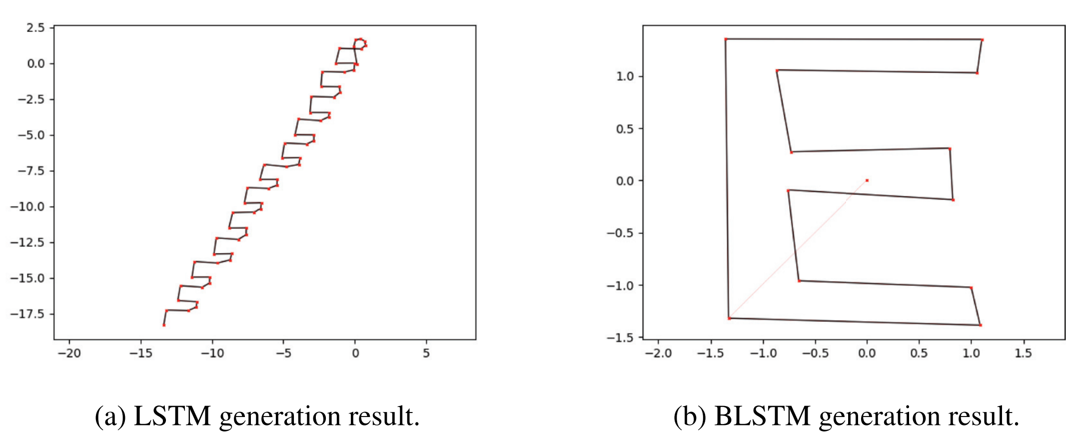
Figure 5.7. Samples from generative results of training glyph A through epochs. Uppercase A is a non-curved multi-contour glyph and so forth, the network is expected to learn to end a contour and start another without the requirement of learning how to draw curves. At epoch 50 (a), the network learned how to draw a line but could not form a closed-shape. At epoch 100 (b), it drew a sequence of lines to form a closed shape but could not end glyph and created loops on that area. However at 200 (c), it closed the shape but did not draw the outer contour of A completely. At epoch 500 (d), network successfully drew the outer contour and started the inner contour. At, 1000 (e) it can be seen that the inner and outer contours are created by the network but the edges of the generated glyph had smooth edges which did not express the style of the input data. Finally at epoch 2000 (f), the network had learned the style of the data and created a sharp edged glyph.



(a) Ep: 50 (b) Ep: 100 (c) Ep: 200

(d) Ep: 500 (e) Ep: 1000 (f) Ep: 4000

Figure 5.8. Samples from generative results of training glyph e through epochs. Lowercase e is a curved multi-contour glyph and so forth, the network is expected to learn both to draw multiple contours and of to learn how to draw curves. At an early epoch 50 (a), the network learned how to draw a curve without being able to close shapes, at epoch 100 (b), it could not create a closed shape but created loops of curves on the same area. At 200 (c), it closed the shape but ended early without forming the outer contour of e. At epoch 500 (d), network still could not finished the outer contour and could not start drawing the inner contour. At, epoch 1000 (e) the outer contour had successfully created however the network did not end the contour and drew an extra contour on the same path. Eventually at epoch 4000 (f), the network had drew both inner and outer contours.




(a) LSTM generation result.

(b) BLSTM generation result.

Figure 5.9. LSTM layers do not have any projection about the further data and make decisions only relying on previous timesteps. For glyphs having repetitive patterns such as `E`, this characteristic of LSTM creates a repetition issue (a). Contrarily BLSTM layers have projections of both past and future of the data and do not create repetitive patterns in the same environment (b). These two generative results are created from the identical learning parameters in two models that contain LSTM nodes and BLSTM nodes respectively.

# CHAPTER 6

# CONCLUSION

This study is the pioneering study on Long Short-Term Memory Networks using vector fonts as data domain. As its primary aim, this thesis proposes a model and guideline for font generation with Long Short-Term Memory networks and provides samples from generative results of several experiments. Additionally, a method of forming a sequential deep learning dataset from fonts is proposed and detailed in this dissertation.

LSTMs have demonstrated great advancements on sequence learning tasks. They achieved impressive results on several sequential problems in speech and handwriting domains. However, complex patterns and high dimensionality on sequential data harms performances of LSTMs dramatically. As its secondary aim, this study also has also strived to assess the effect of different variations of LSTM models and data encoding approaches on learning and generative performances, corresponding to solve a relatively complex problem as generating digital fonts that are both high dimensional and have perplexing sequential patterns.

As a discussion of performance, experimental results show that appropriate data encoding is one of the most important factors in LSTMs. Especially batch consistency and selection of normalization strategy had significant effects on generative abilities of LSTMs. Additionally in experiments, the choice of generative network architecture is proven to have great effects on learning complex patterns in sequences. Lastly, decaying learning rates with correct parameters have exceptionally improved learning abilities of models.

## 6.1. Future Work

This experimental research is a starting point for font generation using sequential deep learning models. While the results of this study is not close to commercial-quality fonts, the methods and the results in this study can be extended to create distributable high quality machine generated vector typefaces. Generated glyph outlines can be encoded into *OpenType* format with additional features such as *hinting*, *spacing* and *ligatures*.

Apart from font generation, this study can also be a foundation to create generative

models for data in several domains inside 2-dimensional space. It can also be extended to multidimensional spaces or can be evolved to support several data features.

It is possible to move this study further by implementing a domain oriented network architecture for digital font generation and using generative adversarial models.

# REFERENCES


[1] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[2] F. A. Gers, J. Schmidhuber, and F. A. Cummins, "Learning to forget: Continual prediction with LSTM," *Neural Computation*, vol. 12, no. 10, pp. 2451–2471, 2000.

[3] A. Graves, "Generating sequences with recurrent neural networks," *CoRR*, vol. abs/1308.0850, 2013.

[4] N. Boulanger-Lewandowski, Y. Bengio, and P. Vincent, "Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription," *Proceedings of the 29th International Conference on Machine Learning, ICML 2012*, vol. 2, 2012.

[5] I. Sutskever, J. Martens, and G. E. Hinton, "Generating text with recurrent neural networks," pp. 1017–1024, 2011.

[6] D. Zhang, "Image recognition using scale recurrent neural networks," *CoRR*, vol. abs/1803.09218, 2018.

[7] J. Wang, Y. Yang, J. Mao, Z. Huang, C. Huang, and W. Xu, "Cnn-rnn: A unified framework for multi-label image classification," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[8] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and tell: A neural image caption generator," in *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pp. 3156–3164, 2015.

[9] R. Suveeranont and T. Igarashi, "Example-based automatic font generation," in *Smart Graphics* (R. Taylor, P. Boulanger, A. Krüger, and P. Olivier, eds.), (Berlin, Heidelberg), pp. 127–138, Springer Berlin Heidelberg, 2010.

[10] Z. Lian, B. Zhao, and J. Xiao, "Automatic generation of large-scale handwriting fonts via style learning," in *SIGGRAPH ASIA 2016 Technical Briefs*, SA '16, (New York, NY, USA), pp. 12:1–12:4, ACM, 2016.

[11] C.-W. Liao and J. S. Huang, "Font generation by beta-spline curve," *Computers & Graphics*, vol. 15, no. 4, pp. 527 – 534, 1991.

[12] K. Yoshida, Y. Nakagawa, and M. Köppen, "Interactive genetic algorithm for font generation system," in *2010 World Automation Congress*, pp. 1–6, 2010.

[13] Y. Jiang, Z. Lian, Y. Tang, and J. Xiao, "Dcfont: an end-to-end deep chinese font generation system," in *SIGGRAPH Asia Technical Briefs*, 2017.

[14] H. Hayashi, K. Abe, and S. Uchida, "Glyphgan: Style-consistent font generation based on generative adversarial networks," *CoRR*, vol. abs/1905.12502, 2019.

[15] Q. Li, J. Li, and L. Chen, "A bezier curve-based font generation algorithm for character fonts," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 1156–1159, 2018.

[16] Hyunil Choi, Sung-Jung Cho, and J. H. Kim, "Generation of handwritten characters with bayesian network based on-line handwriting recognizers," in *Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings.*, pp. 995–999, 2003.

[17] H. Bezine, A. M. Alimi, and N. Sherkat, "Generation and analysis of handwriting script with the beta-elliptic model," in *Ninth International Workshop on Frontiers in Handwriting Recognition*, pp. 515–520, 2004.

[18] T. Varga, D. Kilchhofer, and H. Bunke, "Template-based synthetic handwriting generation for the training of recognition systems," in *Proceedings of the 12th Conference of the International Graphonomics Society*, pp. 206–211, 2005.

[19] T. Yamada, M. Hosoe, K. Kato, and K. Yamamoto, "The character generation in handwriting feature extraction using variational autoencoder," in *2017 14th*

*IAPR International Conference on Document Analysis and Recognition (ICDAR)*, vol. 01, pp. 1019–1024, 2017.

[20] Merriam-Webster.com, “Font.” https://www.merriam-webster.com/dictionary/font, 2019.

[21] J. Gutenberg (printer), “Biblia Latina: The Gutenberg Bible. Volume 1 of 2,” 1455.

[22] A. Dürer, *Underweysung der Messung*. 1538.

[23] D. Fox, A. D. Shaikh, and B. S. Chaparro, “The effect of typeface on the perception of onscreen documents,” *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 51, no. 5, pp. 464–468, 2007.

[24] T. Wright, “History and technology of computer fonts,” *IEEE Annals of the History of Computing*, vol. 20, no. 2, pp. 30–34, 1998.

[25] Google, “Google Fonts.” https://developers.google.com/fonts/, 2016.

[26] S. Hochreiter, “The vanishing gradient problem during learning recurrent neural nets and problem solutions,” *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, vol. 6, no. 2, pp. 107–116, 1998.

[27] M. Schuster and K. K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE Trans. Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[28] A. Graves, N. Jaitly, and A. Mohamed, “Hybrid speech recognition with deep bidirectional lstm,” in *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, pp. 273–278, 2013.

[29] M. Liwicki, A. Graves, H. Bunke, and J. Schmidhuber, “A novel approach to online handwriting recognition based on bidirectional long short-term memory networks,” in *To appear in Proceedings of the 9th International Conference on Document Analysis and Recognition*, 2007.

[30] Z. Huang, W. Xu, and K. Yu, “Bidirectional LSTM-CRF models for sequence tagging,” *CoRR*, vol. abs/1508.01991, 2015.

[31] H. Robbins and S. Monro, “A stochastic approximation method,” *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, 1951.

[32] G. Hinton and T. Tieleman, “Lecture 6.5- RmsProp: Divide the gradient by a running average of its recent magnitude.” Coursera: Neural Networks for Machine Learning, 2012.

[33] N. Qian, “On the momentum term in gradient descent learning algorithms,” *Neural Networks*, vol. 12, no. 1, pp. 145–151, 1999.

[34] Y. E. Nesterov, “A Method of Solving A Convex Programming Problem with Convergence Rate O(1/k2),” *Soviet Mathematics Doklady*, 1983.

[35] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[36] F. Contributors, “fonttools 3.25.0.” https://github.com/fonttools/fonttools/, 2018.

[37] J. H. Friedman, “On bias, variance, 0/1-loss, and the curse-of-dimensionality,” *Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 55–77, 1997.

[38] J. Press, “Lstm online training and prediction: Non-stationary real time data stream forecasting,” 2018.

[39] Z. C. Lipton, “A critical review of recurrent neural networks for sequence learning,” *CoRR*, vol. abs/1506.00019, 2015.

[40] F. Chollet *et al.*, “Keras.” https://keras.io, 2015.

[41] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, “Tensorflow: A system for large-scale machine learning,” in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI’16, pp. 265–283, 2016.

[42] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with CUDA,” *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[43] Intel, “PlaidML 0.3.5.” https://github.com/plaidml/plaidml, 2018.