Norwegian University of
Science and Technology

Recognizing Text Signatures Using Neural Machine Translation

Thomas Gautvedt

Master of Science in Informatics
Submission date: June 2017
Supervisor: Helge Langseth, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

Optical character recognition (OCR) is a technology used to convert scanned text into searchable data. OCR systems have achieved up to 99% recognition rates when working with clean and well-formatted documents under optimal conditions. However, the results are less promising under suboptimal conditions, for example when faced with damaged or obfuscated text.

We propose a new method for recognizing words that are obfuscated in a particular way. This recognition is accomplished by utilizing their "signature," a small portion of the original text. Our approach to this problem is to consider it as a translation problem, and we attempt to solve it by using state-of-the-art methods in the field of machine translation. Three models were developed as a result of the research conducted in this thesis. Two of these were based on the encoder-decoder framework for sequence-to-sequence prediction. The best performing model had an accuracy of over 98% when recognizing text written in a single font and close to 90% when recognizing text written in five different fonts under 10% noise.

Sammendrag

Optisk tegngjenkjenning er en teknologi som brukes for å konvertere skannet tekst til søkbar data. Optisk tegngjenkjenningssystemer har oppnådd nærmere 99% gjennkjenningsrate på rene og velformaterte dokumenter under optimale forhold. Resultatene er derimot mindre lovende under suboptimale forhold, for eksempel når teksten er skadet eller tilslørt.

Vi foreslår en ny metode for gjenkjenning av ord som er tilslørt på en bestemt måte. Denne gjenkjenningen gjøres ved å bruke "signaturen" til ordet, som er en liten del av den originale teksten. Vi betrakter problemet som et oversettingsproblem, og vi forsøker å løse det ved å bruke "state-of-the-art" metoder innen maskinoversettelse. Tre modeller ble utvikler som et resultat av forskningen utført i denne masteroppgaven. To av disse var basert på encoder-decoder-rammeverket for sekvens-til-sekvens-prediksjon. Den beste modellen gjenkjente med en treffsikkerhet på over 98% på tekst skrevet i en skrifttype, og nærmere 90% på tekst skrevet i fem forskjellige skrifttyper under 10% støy.

Preface

This thesis was written the autumn of 2016 and the spring of 2017 at the Department of Computer Science (IDI) at the Norwegian University of Science and Technology (NTNU).

I would first like to express my gratitude to my supervisor Helge Langseth for valuable input, assistance, and feedback throughout the project. I would also like to thank family and friends for their moral support.

Thomas Gautvedt
Trondheim, May 31, 2017

Contents

Abstract i

Sammendrag iii

Preface v

Contents vii

List of Figures xi

List of Tables xiii

I Introduction and Methodology 1

1 Introduction 3
- 1.1 Background 3
- 1.2 Problem and Motivation 4
  - 1.2.1 Real World Example 5
- 1.3 Goals and Research Questions 6
- 1.4 Contributions 7
- 1.5 Thesis Structure 8

2 Methodology 11
- 2.1 Evaluating Research Questions 11
- 2.2 Research Approaches 12
- 2.3 Research Strategy 13
  - 2.3.1 Design and Creation 13
  - 2.3.2 Experiment 14
  - 2.3.3 Combining the Two Strategies 15
- 2.4 Data Generation Method and Data Analysis 15

II Background and Establishing the State-of-the-Art 17

3 Problem Elaboration 19
- 3.1 Tuning Input Data 19

3.2 Typefaces and Typography Techniques . . . 20
3.2.1 Typefaces . . . 20
3.2.2 Typography Techniques . . . 21
3.3 Ambiguous Input . . . 22
3.4 Evaluating Problem Input and Output . . . 23
3.4.1 Input Format . . . 24
3.4.2 Output Format . . . 24
3.5 Translation . . . 24
3.5.1 Dissimilarities to Typical Translation Problems . . . 25
3.6 Method . . . 26

4 Background Theory **29**
4.1 Feedforward Neural Network . . . 29
4.2 Recurrent Neural Network . . . 30
4.2.1 Input and Output Shapes . . . 31
4.2.2 Long-Short Term Memory . . . 31
4.2.3 Gated Recurrent Unit . . . 33
4.3 Encoder-Decoder Framework . . . 33
4.3.1 Attention Mechanism . . . 35
4.4 Vocabulary Encoding . . . 36
4.4.1 One-Hot Vector Encoding . . . 36
4.4.2 Word Embeddings . . . 36

5 Related Work **39**
5.1 Evolution of Machine Translation . . . 39
5.2 Statistical Machine Translation . . . 40
5.3 Neural Machine Translation . . . 41

III System Design, Models, and Experiments **43**

6 System Design **45**
6.1 General Design . . . 45
6.2 Preprocessor . . . 46
6.2.1 Multifont and Casing . . . 47
6.3 Transformator . . . 47
6.3.1 Noise . . . 49

7 Models **51**
7.1 Repeat Vector . . . 51
7.2 Regular Encoder-Decoder . . . 53
7.3 Attention Encoder-Decoder . . . 54

8 Experiments **55**
8.1 Approach 55
8.2 Setup 55
8.2.1 Accuracy Metric 57
8.2.2 Loss Optimizer 57
8.3 Dataset Details 58
8.4 Experiments 58
8.4.1 Accuracy on Datasets 58
8.4.2 Handling of Two Fonts 59
8.4.3 Noise Handling 59
8.4.4 Stress Test 60

IV Results, Discussion, and Conclusion **61**

9 Results and Discussion **63**
9.1 Accuracy on Datasets 63
9.1.1 VecRep 64
9.1.2 EncDecReg 68
9.1.3 EncDecAtt 72
9.2 Handling of Two Fonts 76
9.2.1 Accuracy and Loss for Each Model 76
9.3 Noise Handling 79
9.4 Stress Test 80
9.4.1 EncDecReg Results 81
9.4.2 EncDecAtt Results 82
9.5 Result Discussion 83
9.5.1 General Discussion 83
9.5.2 Research Questions 83
9.5.3 Discussing the Research Goal 84
9.6 Analysis 84
9.6.1 Encoding 84
9.6.2 Decoding 85
9.6.3 Use of Attention 87

10 Conclusion and Future Work **89**
10.1 Conclusion 89
10.2 Contributions 90
10.3 Future Work 90

Appendices **93**

A Image and Figure References **95**

B Example Words **97**

Bibliography **99**

List of Figures

1.1 A book scanner at the Internet Archive headquarters in San Francisco, California . . . 4
1.2 Tweet from @notch . . . 5
1.3 Blurred image of @notch's changelog for Minecraft version 1.7 . . . 6
1.4 Decoding of the blurred image from @notch's Tweet done by Reddit user tmcaffeine . . . 6
1.5 Illustration of a word with a signature with a height of one pixel . . . 7

2.1 Model of the research process . . . 12

3.1 The word "CAT" with two different signatures . . . 19
3.2 The word "CAT" written in Arial and Times New Roman with highlighted serifs . . . 20
3.3 Text with a variable-width font on the left, and a monospace type font on the right . . . 21
3.4 Kerning adjusted text on the left, no kerning on the right . . . 21
3.5 Input and output example . . . 23
3.6 Input with stop words . . . 24
3.7 Illustrative translation between our two languages . . . 25
3.8 Encoding and decoding between our two languages . . . 26

4.1 Illustration of a mathematical model of a neuron . . . 29
4.2 A compact and an unfolded recurrent neural network . . . 30
4.3 Flow of a basic recurrent neural network . . . 32
4.4 Flow of an LSTM cell . . . 32
4.5 An illustration of the proposed RNN encoder-decoder . . . 34
4.6 Encoder-decoder with an attention mechanism . . . 35

5.1 The Google Translate app and its image translation feature . . . 40
5.2 Phrase-based translation of a sentence in German to English . . . 41
5.3 General translation approach in neural machine translation from an English sentence to Italian . . . 42

6.1 Simplified module interaction overview . . . 46

6.2 Illustration of the pipeline process from canvas to signature . . . 47
6.3 Illustration of handlers transforming data . . . 48

7.1 An LSTM outputting its final output after reading a sequence . . . 51
7.2 Repeating the output from an LSTM and feeding it to another LSTM . . . 52
7.3 Simplified illustration of the `VecRep` model . . . 52
7.4 Simplified illustration of the `EncDecReg` model . . . 53
7.5 Simplified illustration of the `EncDecAtt` model . . . 54

8.1 Plotted hyperbolic tangent function . . . 57
8.2 Plotted sigmoid function . . . 57
8.3 The word “SHORTCUTS” written in Arial (monospaced). Size 186x31 . . . 59
8.4 The word “UPLIFTER” written in Times New Roman. Size 180x31 . . . 59

9.1 Accuracy and loss for `VecRep` on small dataset . . . 64
9.2 Accuracy and loss for `VecRep` on medium dataset . . . 65
9.3 Accuracy and loss for `VecRep` on big dataset . . . 66
9.4 Confusion matrix for the best `VecRep` model on the big dataset . . . 67
9.5 Accuracy and loss for `EncDecReg` on small dataset . . . 68
9.6 Accuracy and loss for `EncDecReg` on medium dataset . . . 69
9.7 Accuracy and loss for `EncDecReg` on big dataset . . . 70
9.8 Confusion matrix for the best `EncDecReg` model on the big dataset . . . 71
9.9 Accuracy and loss for `EncDecAtt` on small dataset . . . 72
9.10 Accuracy and loss for `EncDecAtt` on medium dataset . . . 73
9.11 Accuracy and loss for `EncDecAtt` on big dataset . . . 74
9.12 Confusion matrix for the best `EncDecAtt` model on the big dataset . . . 75
9.13 Accuracy and loss for `VecRep` handling two fonts . . . 76
9.14 Accuracy and loss for `EncDecReg` handling two fonts . . . 77
9.15 Accuracy and loss for `EncDecAtt` handling two fonts . . . 78
9.16 Change of accuracy as the amount of noise is increased . . . 79
9.17 Confusion matrix for the best `EncDecReg` model on the stress test . . . 81
9.18 Confusion matrix for the best `EncDecAtt` model on the stress test . . . 82
9.19 Visualization of context vectors for three different words with different noise seeds using t-SNE . . . 85
9.20 Decoding the word “KITTENS” . . . 86
9.21 Decoding the word “KITTENS” by feeding back wrong label . . . 86
9.22 Attention heatmap . . . 87

B.1 Two words written in Arial Mono. Sizes 102x41 and 143x41 . . . 97
B.2 Two words written in Times New Roman. Sizes 198x41 and 269x41 . . . 97
B.3 Two words written in Courier. Sizes 168x41 and 271x41 . . . 97
B.4 Two words written in Georgia. Sizes 276x41 and 96x41 . . . 97
B.5 Two words written in Verdana. Sizes 88x41 and 142x41 . . . 97

List of Tables

3.1 Example of signatures and sequences of the upper-case letters in the English alphabet . . . 23

4.1 Weather data over time for Trondheim in Norway . . . 31
4.2 One-hot encoding of the sentence "is the machine working" . . . 36

9.1 Test accuracy for each model on each test set, with the best results for each test set in bold . . . 63
9.2 Accuracy for each model on a dataset with two fonts . . . 76
9.3 Accuracy for the `EncDecAtt` model . . . 79
9.4 Accuracy for each model on the stress test . . . 80

Part I

Introduction and Methodology

Chapter 1

Introduction

In this chapter, we give an introduction to our thesis. We first present the background to the problem. Section 1.2 presents the problem and gives a real world example. We present our goals and research questions in Section 1.3. Contributions are summarized in Section 1.4, and the final section of the chapter presents the structure of the rest of the thesis.

1.1 Background

The digital revolution marked the onset of the digital age, just like the industrial revolution marked the beginning of the industrial age. Similarly to how the industrial revolution gave us new manufacturing processes, the digital revolution gave us digital electronics, most noticeable the computer (Freeman and Louçã, 2001, chap. 9). With the arrival of high-performance computers and high-speed networks, use of digital technologies have increased rapidly. Digital technologies have enabled information to be created, manipulated, disseminated, relocated, and stored with increasing ease (Lee et al., 2002). This, in addition to increased storage capacities, as well as cheaper storage units, has made digital formats suitable for preservation and storage (Morris and Truskowski, 2003). Data that used to be stored in analog heaps finds new life in digital formats. Photos, audio, video, and books are just a few types of data that are commonly stored, or preserved, digitally.

Oxford English Dictionary (2010) defines the action or process of digitizing, the task of converting analog data into digital data, as "digitization." Google Inc.[1] and their service Google Books[2] is one example of mass digitization. Google Books is a service that provides full-text search of books and magazines online. Other similar projects like Google Books includes JSTOR[3], which provides online access to millions of academic journal articles, and Bokhylla.no[4] (The Bookshelf). Bokhylla.no is a project initiated

[1] `https://www.google.com/intl/en/about/`
[2] `https://books.google.com`
[3] `https://www.jstor.org`
[4] `http://bokhylla.no`

by the National Library of Norway. It was launched in 2009 and aims to provide online access to literature published in Norwegian. The service will contain about 250,000 books when it is completed in 2017 (Nasjonalbiblioteket, 2016). Google's primary goal with Google Books is to provide a search and index service (Coyle, 2006), whereas the goal with Bookhylla.no is to provide an enhanced reading environment where visitors can read entire books from cover to cover. Bookhylla.no also provides a complete in-text search of its entire library.

Figure 1.1: A book scanner at the Internet Archive headquarters in San Francisco, California

Despite their different goals with digitization, they rely on the same types of technologies to achieve them. While simply scanning (see Figure 1.1) will suffice to make the literature available online, other technologies are needed to index the content. Indexing is the process of capturing the scanned text and converting it into searchable data. This capturing is done with a technology called optical character recognition, or OCR for short. OCR has many applications, and is in use in many areas today, such as book scanning, number plate recognition, handling of checks and passports, as well as assistive technologies for blind and visually impaired users (Mori et al., 1999; Kurzweil et al., 2000).

1.2 Problem and Motivation

OCR is, in a broad sense, a branch of artificial intelligence, and a research branch in pattern recognition and computer vision (Mori et al., 1999). It was first believed that it would be easy to develop an OCR, and researchers estimated that an accurate reading machine would be introduced in the 1950s. During the 1950s and the early 1960s, researchers were still struggling with an ideal OCR model. Their early work has since

laid the foundation for modern research in the field (Mori et al., 1992). Things have progressed since the 1950s. Ye and Doermann (2015) states:

> While many researchers view optical character recognition (OCR) as a solved problem, text detection and recognition in imagery possess many of the same hurdles as computer vision and pattern recognition problems driven by lower quality or degraded data.

OCR systems have achieved up to 99% recognition rates when working with clean and well-formatted documents under optimal conditions. However, under suboptimal conditions, where the OCR system faces variants of text layouts and fonts, uneven illuminations, and other obstacles, the systems typically have lower rates of detection and recognition (Ye and Doermann, 2015). Pasting of paper, ink spreading, fading, or dirt, are just a few ways text can be damaged or obfuscated (Bhardwaj and Agarwal, 2014). This leads to obstacles and difficulties for OCR systems. In this thesis, we take on a particular kind of obfuscated text and present a new way to recognize it by utilizing their "signature."

1.2.1 Real World Example

An example of the type of obfuscated text we will attempt to handle in this thesis can be found in a Tweet from Markus "Notch" Persson. Persson is a Swedish video game programmer and designer and is most famous for creating the highly acclaimed video game Minecraft. He was an inveterate user of Twitter and used to hint or tease upcoming features of the game. Figure 1.2 contains a Tweet from Persson, which was Tweeted on June 12th, 2011.

Figure 1.2: Tweet from @notch

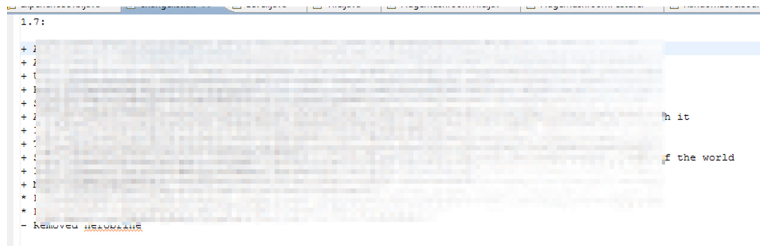

Figure 1.3: Blurred image of @notch's changelog for Minecraft version 1.7

Persson's Tweet is referring to an image which can be seen in Figure 1.3. The image is a print screen of what looks like a text editor or an IDE[5]. Although the image is heavily blurred and cropped, some of the text is partially visible, like the lower portion of the file names in the tab pane at the top of the image. When this Tweet was posted, one user who called himself tmcaffeine on Reddit[6], was able to identify the text in the tabs. He installed the same IDE Persson used and matched the default font in the program against the letters that were partially visible in the image. His decoded image can be seen in Figure 1.4. More than four months later, when version 1.7 of Minecraft was released, it was clear that the decoding was correct as big mushrooms started to appear in the game[7] (Official Minecraft Wiki, 2017).

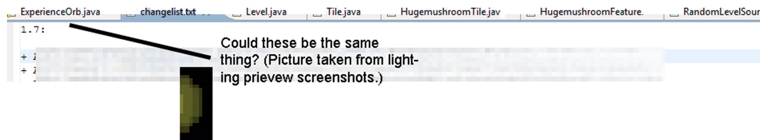

Figure 1.4: Decoding of the blurred image from @notch's Tweet done by Reddit user tmcaffeine

The decoding that was done to match the letters poses an interesting problem. Can characters and words be recognized given only a small portion of the glyphs? The decoding in Figure 1.4 was done manually by comparing letters one-by-one, but could such recognition be done automatically? Could this type of pattern recognition be learned? These questions were the basis for this thesis.

1.3 Goals and Research Questions

Our goal is to use the "signature" of letters in a word to recognize it. We can imagine a "signature" by applying two masks to an image with some written text. These masks span the total width of the image, and also cover most of the image from top to bottom,

[5]Integrated Development Environment
[6]`https://www.reddit.com`
[7]The experience orbs were postponed, and introduced in the 1.8 update instead.

and from the bottom to the top. The only area that is not covered is the narrow gap between the two masks. The pixels in this gap defines the "signature" for our image and the text written in it.

Figure 1.5: Illustration of a word with a signature with a height of one pixel

Figure 1.5 contains the written word "THESIS." The original text in this image has a height of 50 pixels. Our masks are applied at both the top and bottom, exposing only a single line that has a height of one pixel. This line defines the "signature" for this word and is highlighted in red. The areas that are covered by the masks are translucent and only shown for illustrative purposes. This approach was used to extract signature sequences from words in this thesis.

Our overarching research goal is stated below. This was the main goal we focused on archiving throughout the work on this thesis. In addition to the research goal, we also created three research questions that we wanted to answer. Our research goal is somewhat open and mainly explores whether or not an approach like the one we propose is possible, while our research questions are more specific and concrete. The research questions are asked to get more insight and knowledge related to the approach we propose. The importance of the research questions is explained in greater detail in Section 2.1.

Research Goal *Develop a model that is able to use signature sequences to recognize words and letters*

Research Question 1 *Does ambiguity in character signature sequences impact recognition rates?*

Research Question 2 *Can a signature sequence based model handle multiple fonts at the same time?*

Research Question 3 *Can a signature sequence based model be robust to noise?*

1.4 Contributions

The contribution of this thesis is mainly the exploration of how to use machine learning and signature sequences to recognize words. The results of the experiments and

conclusions done in this thesis may lay the foundation for new ways to use machine learning to recognize patterns in data using sequences. Although the ideas presented in this thesis are not new, they may prove that certain problems can be solved using data in different ways. Detailed analysis and exploration of the underlying concepts and implementations of the models, in the context of our results, will also serve as contributions. This study may also give better insight into how the proposed models work and how they handle a problem with our characteristics.

1.5 Thesis Structure

The remainder of this thesis is structured as follows:

Part I – Introduction and Methodology

- **Chapter 2 – Methodology** establishes the methodology used throughout the thesis. It explains how the research was carried out and how we chose our research approach.

Part II – Background and Establishing the State-of-the-Art

- **Chapter 3 – Problem Elaboration** explains the problem in greater detail. It explains the various factors that define our problem. We look into how we can interpret our problem in the context of established fields of machine learning. The method and approach to solving the problem are also established.
- **Chapter 4 – Background Theory** introduces some of the background information relevant to the proceeding chapter.
- **Chapter 5 – Related Work** presents related work done in the area of machine translation. We present some of the earlier work as well as establishing state-of-the-art methods.

Part III – System Design, Models, and Experiments

- **Chapter 6 – System Design** introduces the system we developed to run our experiments. This brief introduction includes overall design and general approach, as well as describing key components.
- **Chapter 7 – Models** presents the three models we developed as a part of our research based on state-of-the-art technologies.
- **Chapter 8 – Experiments** establishes how we conducted our experiments, as well as presenting the structure of each experiment and what our goals and expectations were. This chapter also presents the configuration of the models and choice of hyper-parameters.

Part IV – Results, Discussion, and Conclusion

- **Chapter 9 – Results and Discussion** present the result of each experiment presented in the previous chapter. We discuss the results and evaluate the models. An analysis of the models in the context of the results from the experiments is also carried out.

- **Chapter 10 – Conclusion and Future Work** draws the conclusion for our research. We state our research contribution and present thoughts on possible paths for future work.

Chapter 2

Methodology

This chapter presents the research methodology that was used in the process of writing this thesis. Section 2.1 looks closer at the research questions which we defined in the previous chapter. Section 2.2 presents the research approach, and Section 2.3 presents our choice of research strategies. In Section 2.4 we present our data generation methods and how we did our data analysis.

2.1 Evaluating Research Questions

We have already presented our research goal and questions as:

Research Goal *Develop a model that is able to use signature sequences to recognize words and letters*

Research Question 1 *Does ambiguity in character signature sequences impact recognition rates?*

Research Question 2 *Can a signature sequence based model handle multiple fonts at the same time?*

Research Question 3 *Can a signature sequence based model be robust to noise?*

In this section we will look closer at how we may give an answer to our research questions in a meaningful way:

RQ1: It was made clear that ambiguity could be an issue early in our testing. How and why it could impact our problem is explained thoroughly in Section 3.3. Measuring the potential impact of ambiguity could be done by conducting tests and experiments to see how the recognition rates were affected by ambiguous data.

RQ2: While learning to recognize words written in one font may be challenging, introducing additional fonts will almost certainly increase the difficulty, resulting in a more complex problem. Whether or not a model is capable of handling this can,

similarly to **RQ1**, be answered by running tests with text written in multiple fonts and see how the recognition rates were affected. Sections 3.1 and 3.2 explains how we can tune the input and how typography techniques may affect recognition.

RQ3: While clean and well-formatted data is preferable, we may not always be so lucky that our data is completely noise free. Noise-induced data is also more "realistic," in a real world perspective than 100% noise-free data. Investigating how well a model can handle certain degrees of noise is interesting, as it could say something about its robustness. Investigating this could be done by running tests and introduce noise increasingly until the recognition rates deteriorate beyond a certain point.

2.2 Research Approaches

We chose to follow the research process presented by Oates (2005, chap. 3). Figure 2.1 gives an overview of this process and its components. The overview illustrates how to form different research approaches by mixing and matching various types of strategies, data generation methods, and types of data analysis.

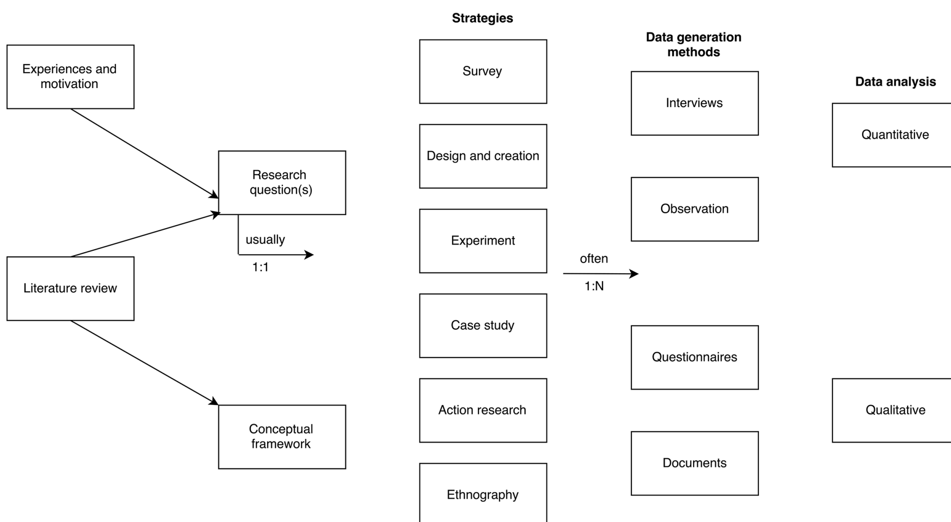

Figure 2.1: Model of the research process

Our literature review is presented in Chapter 5. The research questions were presented in Section 1.3 and was explained further in the previous section. The conceptual framework is presented throughout this chapter. Our choice of research strategy is presented in the next section, and our choice of data generation methods, as well as our data analysis approach, is presented in Section 2.4.

2.3 Research Strategy

Our choice of research strategy was based on our research questions and overarching research goal, and how they could best be answered and achieved. We decided to use the research strategy of design and creation combined with the research strategy of experiments. Oates (2005, pp. 35) states that one research question typically has one research strategy, but that it is also possible to combine one or more. Our combination of research strategies allowed us to build one or more models iteratively, and to test various approaches to find optimal solutions. We followed the strategy of design and creation for the most part, but utilized experiments to investigate cause and effect relationships. We also used experiments to answer the research questions, which would be an important part of our research.

2.3.1 Design and Creation

Development of a new IT product is the primary focus of the design and creation research strategy. IT products are also called artefacts, and there are four of these (Oates, 2005, pp. 108–109):

- **Constructs:** the concepts or vocabulary used in a particular IT-related domain. For example, notions of entities, objects, or data flow.
- **Models**[1]**:** combinations of constructs that represent a situation and are used to aid problem understanding and solution development. For example, a data flow diagram, a use case scenario, or a storyboard.
- **Methods:** guidance on the models to be produced and process stages to be followed to solve problems using IT. For example, formal, mathematical algorithms, or commercialized and published methodologies.
- **Instantiations:** a working system that demonstrates that constructs, models, methods, ideas, genres or theories can be implemented in a computer-based system.

In our case, the artefact we wanted to develop as a part of our research would fall into the category of instantiations. This artefact would be a fully functional system that would be capable of achieving our research goal, as well as answering our research questions. As this system would be an essential part of the research process, it would be important that it could be considered as research, and not just as a demonstration of technical powers. The process and the development of the system had to demonstrate academical qualities such as analysis, explanation, argument, justification, and critical evaluation (Oates, 2005, pp. 109–111).

Approach

The approach in design and creation revolves around a problem-solving strategy. It utilizes an iterative process over five steps (Oates, 2005, pp. 111–112):

[1] Note that Oates (2005) uses this term for something else than what the "industry" term implies. Outside this chapter, we use the term "model" as the term "instantiations" defined by Oates (2005).

- **Awareness:** involves recognizing a problem. This step is necessary to find what problem we are trying to solve.
- **Suggestion:** is the step where we create a tentative idea of how the problem might be addressed.
- **Development:** is where we implement an idea from the previous step.
- **Evaluation:** involves examining the artefact. Evaluations are done to estimate its worth and deviations from the expectations.
- **Conclusion:** is the final step in the cycle where results are collected and written down. Gained knowledge is identified, and any unexpected or unexplainable results could lay the ground for further research.

It is important to understand that these steps are not necessarily followed strictly. Instead, they work as guidelines, and the process is more of an iterative fluid cycle where the approach may shift depending on problem or situation. Oates (2005, pp. 112) explains how these cycles work and what you as a researcher achieves by using this research method as follows:

> Thinking about a suggested tentative solution leads to greater awareness of the nature of the problem; development of a design idea leads to increased understating of the problem and new, alternative tentative solution; discovering that a design doesn't work according to the researcher's expectations leads to new insights and theories about the nature of the problem, and so on.

The goal is to work out a prototype that is gradually modified until a satisfactory implementation is produced. One of the biggest advantages of this approach is that it is not necessary to fully understand a problem before developing prototypes and exploring tentative solutions. This research strategy also opens up the possibilities of testing prototypes often and comparing results along the way to see if one direction or approach works better than others.

2.3.2 Experiment

Experiments are, as already mentioned, a research strategy that focuses on investigating cause and effect, and the relationship between the two. The research strategy is structured around hypotheses. With a given hypothesis, an experiment is designed to prove or disprove the hypothesis. For example, a hypothesis may be:

Hypothesis: *if I go outside in the rain, I am going to get wet.*

Testing this hypothesis could be done by going outside in the rain to see if the subject got wet. According to Oates (2005, pp. 126–129), research strategies that are based on experiments may, among others, be characterized by:

- Observation and measurement. Here the researchers make a precise and detailed observation of outcome and changes that occur when a particular factor is introduced.
- Proving or disproving a relationship between two or more factors.
- Explanation and prediction. The researchers can explain the casual link between two factors.
- Repetition, where experiments are carried out multiple times. This repetition is done under varying conditions, to be certain that the observed and measured outcomes are not caused by some other factor.

2.3.3 Combining the Two Strategies

Neither strategy was used in its complete form throughout the entire process. Instead, we leaned on the fluid nature of design and creation, and we used concepts from experiments to help us move forward.

We created meaningful hypotheses related to the current tentative solution for a problem. While implementing the solution, we also made sure to construct it in such a way that we could test our hypotheses when it was finished. We evaluated the results of the development process and cross-examined them with the results from the experiments. The combined knowledge gained though this approach helped us in the next cycle of the iteration.

2.4 Data Generation Method and Data Analysis

Data generation method is the means by which we produced the empirical data. This data was used to evaluate our research. Many researchers who chose to use the design and creation strategy pay little attention to properly use the data generation methods as presented in the research approach model (Oates, 2005, pp. 116–117). This is usually because the artefact that is developed needs to be tested in a specific way. Evaluation of a system like ours is often done by training and testing it on sets of data. Such data generation methods fall outside the overview presented in Figure 2.1. We chose, for our data generation method, to construct our own datasets and tested the system on these sets.

Data analysis based on the data from our tests was done quantitatively. Quantitative data analysis means that our data and evidence were based on numbers. The data was compared and analyzed using tables, charts, and graphs (Oates, 2005, chap. 17). Developing a system iteratively, as we did, this type of analysis made it easier to compare results, behavior, and progression.

Part II

Background and Establishing the State-of-the-Art

Chapter 3

Problem Elaboration

In this chapter, we will expand on the problem definition given in the introduction. In Section 3.1 we explain how the use of signatures allows us to tune the input data. We explain how different typefaces and typography techniques may affect our problem in Section 3.2. How our problem has to deal with ambiguous data is explained in Section 3.3. Input and output are evaluated in the context of each other in Section 3.4, and in Section 3.5 we evaluate our problem as a type of translation task. Finally, in Section 3.6, we present our method for solving the problem.

3.1 Tuning Input Data

With the use of signatures, we can alter how we capture the data from the original image in various ways. These alterations can be done by configuring text sizes, signature positions, or signature heights in different ways. Altering the signature capturing configurations would result in completely new sequences for words.

Figure 3.1: The word “CAT” with two different signatures

Let us consider the word “CAT” in upper-case letters, written in Arial with a font height of 50 pixels, as illustrated in Figure 3.1. In this figure, we have highlighted two signatures, both with a height of one pixel. The first signature is 16 pixels from the bottom, while the other is five pixels from the top. As we can see from the figure, the

signatures we capture from the text varies significantly depending on where we chose to place it. For example, the top signature defines a T as 38 black pixels, while the bottom defines it as seven black pixels.

Figure 3.1 also illustrates how our system needs to differentiate between letter spacing and characters that consist of multiple strokes. In the bottom signature, the letter C is defined as a sequence of eight black, 27 white, and seven black pixels. However, the sequence of seven black, seven white and 35 black pixels is just the final stroke of the letter C as well as the letter A. This sequence should not be classified as a letter, as it contains fragments of two separate ones.

3.2 Typefaces and Typography Techniques

In this section, we look closer at how typefaces (also known as font family), and techniques in typography may affect our problem.

3.2.1 Typefaces

In addition to tuning the capturing of the input data, choice of fonts may also play a role in the ambiguity of the problem. In Figure 3.1 we used the monoline sans-serif font Arial. Monoline means that all the strokes in the typeface have the same widths (Felici, 2012, pp. 315). Sans-serif means that the font does not have serifs; a crossing feature at the end of the principal character strokes (Felici, 2012, pp. 33–36). In Figure 3.2 we have written the word “CAT” first in Arial, and then in Times New Roman. Times New Roman is a font with serifs which is not monoline. The serifs are highlighted in red in the figure. The visual difference between the two words illustrates how the choice of font affects the captured sequences.

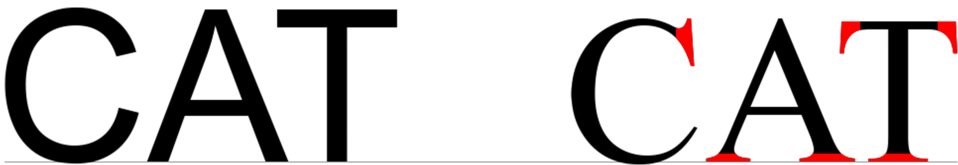

Figure 3.2: The word “CAT” written in Arial and Times New Roman with highlighted serifs

Fonts are also spaced differently. A monospaced font, also called fixed-width font, uses the same width for all characters. This is in contrast to variable-width fonts, where each character may have different widths (Felici, 2012, pp. 8–11). Illustrated in Figure 3.3, the text on the left is written in regular Arial which has variable width characters, while the text on the right is written in an Arial variant with monospacing. The two words written in monospaced font have equal width, whereas the two words that are written in a variable-width font have different widths. With a monospaced font, each glyph is placed inside a maximum width constraint. The glyph does not need to take up the entire width, but monospacing guarantees that there are no strokes outside the

constraint. In our problem, we do not define where a character "starts" or "stops," so we do not know the location of these constraints. However, the use of a variable-width font may give increased variance in the distances between two glyphs.

Figure 3.3: Text with a variable-width font on the left, and a monospace type font on the right

3.2.2 Typography Techniques

There are additional factors that affects how text looks, which may, in turn, affect the characteristics of the captured sequences. Kerning is one such factor. Kerning is the process of adjusting the spacing between two characters to compensate for their relative shapes. This is done to increase the readability and create a more visually pleasing result (Felici, 2012, chapt. 11). An illustrative example of kerning can be seen in Figure 3.4, where the letters on the left side have applied kerning, bringing the two letters closer. The letters on the right side have no kerning. On the left side, the letters more naturally fit against each other, while the letters on the right side have a conspicuous spacing between them. As for our problem, kerning may make it more difficult to separate the characters from each other. Because kerning eliminates long sequences of spacing, by shifting characters closer to each other, it eliminates "hints" that could otherwise be used to identify where one letter ends and another begins.

Figure 3.4: Kerning adjusted text on the left, no kerning on the right

Other techniques exist in typography that are applied to text to achieve more readable and visually pleasing arrangement of characters. As these alter how the text looks, it may alter our input data. Some of these are (Felici, 2012, pp. 320, 299, 302, 298, 297):

- Typographic alignments, such as justified text or ragged right.

- Font weights, which defines the degree of boldness and widths of strokes.
- Slanted forms such as italic.
- Location of the baseline, which defines the line that most of the characters appear to be sitting on.
- Anti-aliasing or font smoothing, which is a technique for smoothing the appearance of characters on a computer screen by adding gray pixels around their edges.

3.3 Ambiguous Input

There is a chance our input data may end up being ambiguous. A character signature can consist of a single sequence of black pixels or a series of alternating black and white pixels. Because our system does not know what a character looks like, there is no way for it to know what sequences are “rests” of characters, spacing, or valid characters. It may also happen that a character consisting of a single sequence of black pixels is also a subset of another character.

Consider the bottom signature in Figure 3.1. Because Arial is a monolinear font, we know that the strokes on the C and the strokes on the T have equal width. The T could be represented as a series of some white pixels, three black pixels, and optionally more white pixels. We can not know for sure just how many white pixels is before or after the three consecutive black pixels, because this may vary depending on what character is before and after the T. If our system learns this mapping, it will incorrectly recognize various other characters, such as parts of the C as a potential T. This means that our system have to recognize a huge variety of sequences and correctly map them to an output character while ignoring invalid sequences.

Table 3.1 holds the actual signatures and sequences for monospaced Arial given the settings specified in Section 1.3. This table illustrates how the input is encoded and what our system has to recognize and learn. We can see examples of why the problem may be difficult in the sequences for the letters C, I, J, L, T, and Y. These six letters all share the same identical sequence. With an identical sequence, the only way we can differentiate between these letters is to consider the white pixels before and after its signature. Many of the other character sequences besides the ones for C, I, J, L, T, and Y also contains three consecutive black pixels. In total, the subsequence of three consecutive black pixels is found in eleven of the seventeen unique character sequences. Some of them, like W and N, also contains three subsequent black pixels multiple times in their sequences.

Similarity, sequences for complete words will also consist of repeated subsequences. Some of these subsequences may be individual letters, while others are parts of multiple ones. This substantiates that our problem requires a system that can learn beyond simply mapping parts of sequences to sets of labels.

Character(s)	**Signature**	**Sequence**
A	[0,0,0,1,1,1,1,1,1,1,0,0,0]	$3B,7W,3B$
B	[0,0,0,1,1,1,1,1,1,1,0,0,0,0,0]	$3B,7W,5B$
C, I, J, L, T and Y	[0,0,0]	$3B$
D	[0,0,0,1,1,1,1,1,1,1,1,1,1,0,0,0]	$3B,10W,3B$
E and F	[0,0,0,0,0,0,0,0,0,0,0,0,0,0]	$14B$
G	[0,0,0,1,1,1,1,1,1,0,0,0,0,0,0,0]	$3B,6W,7B$
H and U	[0,0,0,1,1,1,1,1,1,1,1,1,0,0,0]	$3B,9W,3B$
K	[0,0,0,0,0,1,0,0,0,0]	$5B,1W,4B$
M	[0,0,0,1,1,1,1,0,1,1,1,1,0,0,0]	$3B,4W,1B,4W,3B$
N	[0,0,0,1,1,1,1,0,0,0,1,1,0,0,0]	$3B,4W,3B,2W,3B$
O and Q	[0,0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,0]	$3B,11W,3B$
P	[0,0,0,0,0,0,0,0,0,0,0,0,0]	$13B$
R	[0,0,0,0,0,0,0,0,0,0]	$10B$
S and X	[0,0,0,0,0,0,0]	$7B$
V	[0,0,0,0,1,1,1,1,0,0,0]	$4B,4W,3B$
W	[0,0,0,1,1,1,0,0,1,0,0,0,1,1,0,0,0]	$3B,3W,2B,1W,3B,2W,3B$
Z	[0,0,0,0]	$4B$

Table 3.1: Example of signatures and sequences of the upper-case letters in the English alphabet

3.4 Evaluating Problem Input and Output

In our problem, we have an input that is a matrix or a vector of binary data. The binary input denotes the color of a particular pixel at the given location in our "signature." The output, or the correct labels, will correspond to the correct letter in the word. For convenience, the letters will be given a unique integer value. For example, if the words are written in upper-case letters of the English language, we could use the numbers in the range 1 to 26, denoting A through Z.

$$\begin{aligned} \vec{\text{inputRaw}} &= [4W,3B,7W,3B,8W,3B,18W,3B,23W,3B,13W,14B,6W,3B,10W,3B] \\ \vec{\text{inputEncoded}} &= [4,-3,7,-3,8,-3,18,-3,23,-3,13,-14,6,-3,10,-3] \\ \vec{\text{outputEncoded}} &= [1,12,12,9,5,4] \\ \text{output} &= \text{ALLIED} \end{aligned}$$

Example 3.5: Input and output example

Example 3.5 contains actual input and output for the word "ALLIED." Note that in this example, we have encoded the input to integers. This was done by counting the length of consecutive pixels of the same color. Consecutive black pixels are negated, to differentiate between sequences of black pixels and sequences of white pixels. With

this encoding, four black consecutive pixels are encoded as -4. Similarity, a sequence of 18 consecutive white pixels is encoded as 18.

3.4.1 Input Format

Our input has the feature that they form a sequence. Both the values in the sequence, as well as the ordering of the sequence, is crucial for the prediction. This feature is fundamentally different from other problems such as traditional image recognition, where the exact location of a pattern may be irrelevant.

Because the input forms a sequence, it is important that the entire sequence is read, and that we do not cut the sequence off at the end. Truncating or ignoring values in the input sequence would result in mislearning. Instead of our model correctly identify subsequences, the model would attempt to find patters in the data that is not there. This mislearning would cripple the model, and the overall accuracy may suffer due to contradicting sequences. Keeping the input data unaltered and complete is therefore essential.

$$
\begin{aligned}
\text{input}\vec{\text{E}}\text{ncoded} &= [-3, \pi, 23, -3, \pi, 13, -14, \pi] \\
\text{output}\vec{\text{E}}\text{ncoded} &= [12, 9, 5] \\
\text{output} &= \text{LIE}
\end{aligned}
$$

Example 3.6: Input with stop words

We lack the concept of "stop words" in our problem. Example 3.6 illustrates an input with stop words, denoted as π. Stop words may make the problem easier to solve, as we would know within which boundaries each character resides. In Example 3.6, we have placed the stop word right after the end of each character, instead of just the barriers of the character itself. These barriers could potentially reduce ambiguity as we would know for a fact that the letter I if followed by an E, would always be the subsequence $[23, -3]$. However, instead of relying on stop words, we want our model to find a pattern in the input that makes sense based on the corresponding output. This pattern, if correctly predicted, would not need explicit stop words, as the model would be able to find them implicitly.

3.4.2 Output Format

As with the input, our output also forms a sequence. Both the values in the sequence and the ordering of the sequence itself is important. The words "HELLO" and "HLLOE" contains the same letters, but have different meanings.

3.5 Translation

We have now considered the input and output separately. Considering the input and output in the context of each other, we can see the relationship between the two. Se-

quences and subsequences in the input result in either an output sequence or a single output value. This process can be seen as a type of translation. We “translate” the input sequences of language α into an output sequence of language β. It is irrelevant that the actual input and output are not defined languages with linguistic properties. As long as there is a relationship between the source and target languages, the task may be considered as a translation problem. Figure 3.7 illustrates the translation between our source and target languages, reusing the data from Example 3.6. The values in the translated language are the same as the encoded output in Example 3.6, that is, the index value for each letter in the English language, where A is 1, B is 2 and so on.

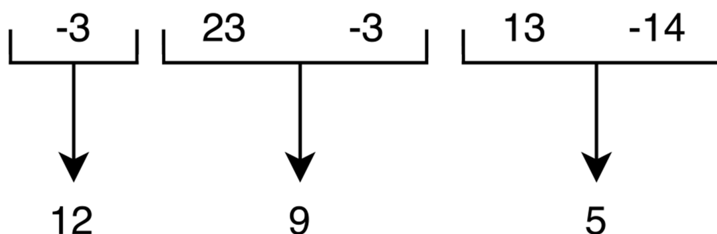

Figure 3.7: Illustrative translation between our two languages

3.5.1 Dissimilarities to Typical Translation Problems

While our problem may be similar to that of translation, it also differs in several ways. Common for almost all of these differences is that our languages are very simple. This difference means that we can simplify, or completely ignore, tasks that are otherwise important in more complex translation problems. For example, the task of “part-of-speech tagging” is to determine the part of speech for each word in a sentence. This means to determine if a word is a noun, verb, adjective, and so on. “Part-of-speech tagging” is a complex task and a task that is important, as it may influence how words are used in different contexts. In our constructed languages, we have no concept of such speech groups. Our “words” have one, and only one, meaning, regardless of context.

Our problem also differs from translation between two spoken words when it comes to arrangement and order. We know that with our two languages, the ordering is always the same, going from left to right. Translating spoken languages can never give the same guarantee, as different languages have different sentence structures. Words that can be directly translated from one language to another do not necessarily have the same arrangement in both of them.

These observations make it clear that although our problem is related to translation, it also differs from more traditional problems.

3.6 Method

We established in the previous section that the problem can be considered as a translation problem between two languages α and β. In language β we have some word W. This word is encoded by running it through some function $f_1(x) = y$, resulting some output U in our other language α. This output is the signature sequences. The process of encoding the original word W to the sequence U can be considered as a translation, from language β to α. Our goal is to take the output U, run it through some other function $f_2(x) = y$ which gives us the original word W. This process is the translation in the opposite direction and can also be considered as decoding the encoded sequence. Both the encoding and decoding processes are illustrated in Figure 3.8.

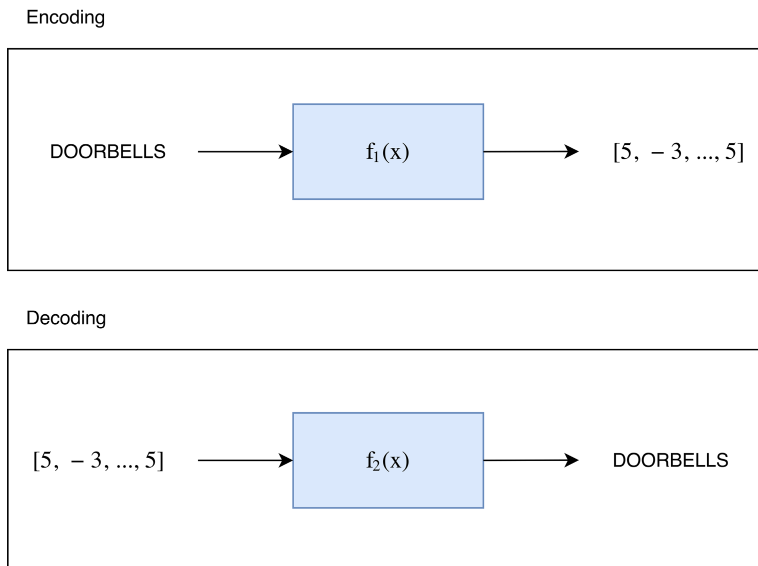

Figure 3.8: Encoding and decoding between our two languages

Our encoding function is a deterministic algorithm, that is, given the same input, it will always produce the same output. The fact that the problem is deterministic makes it possible for us to verify a solution with absolute certainty, which is usually not the case with traditional translation problems. The nature of this encoding also makes the problem solvable in many ways.

One way we could potentially solve the problem is by applying exhaustive search, also known as brute-force search. This technique involves enumerating all possible candidates for the solution and is guaranteed always to find a solution if it exists. The problem with this approach is the number of calculations necessary to find a solution. If we have a total of 26 classes, one class for each letter in the English alphabet, and we want to decode the signatures of a word that is 16 letters long, we end up with 26^{16} possible combinations. This number makes it clear that exhaustive search approaches are not applicable to a problem like ours.

Instead of utilizing exhaustive search and brute-force methods, we want to find a way to solve the problem in a reasonable amount of time. A trade-off from this is that an optimal solution in many cases can not be guaranteed. We have focused on more general areas of machine learning to solve this problem, particularly the field of natural language processing and machine translation. The next two chapters present background theory and related work done in this and closely related fields.

Chapter 4

Background Theory

In this chapter, we introduce the background theory relevant to the related work presented in the next chapter. Feedforward neural networks are introduced in Section 4.1. Recurrent neural networks, in addition to some common recurrent neural network architectures, are introduced in Section 4.2. The encoder-decoder framework is introduced in Section 4.3, and two types of vocabulary encoding methods are presented in Section 4.4.

4.1 Feedforward Neural Network

Artificial neural network is a computational model used in machine learning and computer science. The idea behind neural networks lies in the use of artificial neurons, an idea that can be traced back to the 1940s. These artificial neurons are loosely analogous to axons in a biological brain, and artificial neural network is an attempt at modeling the information process capabilities of nervous systems (Stuart Russell, 2010, pp. 727–728).

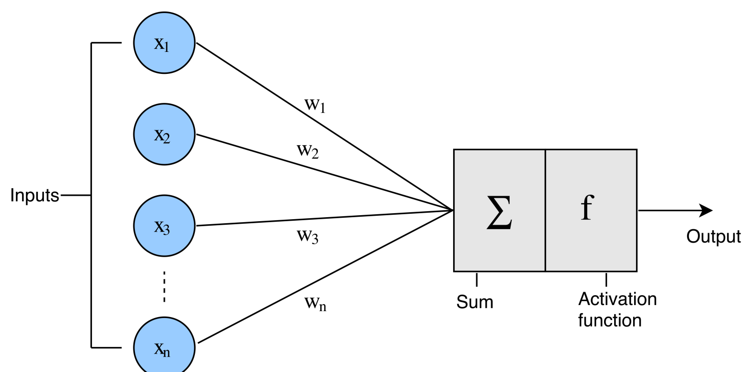

Figure 4.1: Illustration of a mathematical model of a neuron

Figure 4.1 illustrates a simple mathematical model for a neuron, often called a unit or a node. This unit "fires" when a linear combination of its inputs exceeds some threshold. A neural network is a collection of many such units. The properties of a network are determined by its topology, as well as the properties of the units. Networks are constructed by directly linking nodes with each other. A link from unit i to unit j serves to propagate the activation a_i from i to j. The strength and sign of the signal are determined by the numeric weight that is associated with the unit.

A feedforward network consists of units which have connections that only goes in one direction. These nodes receive input from the "upstream" nodes, and delivers output to the "downstream" nodes, forming a directed acyclic graph (Stuart Russell, 2010, pp. 728–729).

4.2 Recurrent Neural Network

A recurrent neural network (RNN) is a type of artificial neural network. This type of neural network creates an internal state of the network which allows it to express dynamic temporal behavior. The structure of a recurrent neural network is much like a feedforward network, but in addition to feeding "downstream" nodes, nodes also feed its output back into its own inputs (Stuart Russell, 2010, pp. 729). This type of architecture can support a short-term memory, a feature that is necessary for problems where data depends on previous data, for example in areas such as speech recognition.

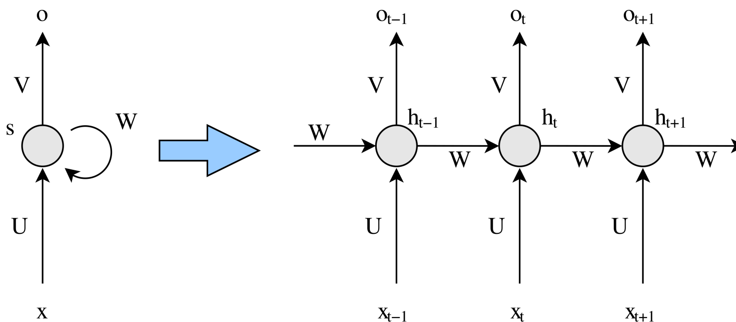

Figure 4.2: A compact and an unfolded recurrent neural network

We can consider recurrent neural networks as a loop, and we can unfold it into a complete sequence, as illustrated in Figure 4.2. In the figure, X_t is the input, and h_t is the hidden state of the recurrent network at timestep t. The hidden state function as the unit's memory and the hidden state from timestep t is reused in the next timestep $t+1$, along with the input of the current timestep. It is also important to note that the network shares the same weights W across timesteps. The recurrent neural network has input to hidden connections parametrized by a weight matrix U, as well as hidden-to-hidden recurrent connections parametrized by a weight matrix W. In addition, the network has hidden-to-output connection parametrized by a weight matrix V (Goodfellow et al., 2016, pp. 378–379).

$$h_t = \sigma(b + Wh_{t-1} + Ux_t) \tag{4.1}$$

$$\hat{y}_t = \sigma(c + Vh_t) \tag{4.2}$$

Equations 4.1 and 4.2 are slightly simplified from Goodfellow et al. (2016, pp. 378–381), and defines the forward propagation of the model illustrated in Figure 4.2. The parameters b and c are the bias vectors. Computing the gradient in a recurrent neural network can be done with an iterative gradient descent back-propagation algorithm, such as back-propagation through time (Werbos, 1990; Rumelhart et al., 1988).

4.2.1 Input and Output Shapes

Recurrent neural networks are usually fed data that has an input shape of `(timesteps, features)`, where the first dimension defines how many timesteps our data consists of, and the last dimension defines the number of features for each timestep. Table 4.1 contains weather data for Trondheim in Norway in the period June 2016 to September 2016[1]. This example has a total of four timesteps, one for each month in the period, and a total of three separate features: temperature, rain, and wind. This data has an input shape of `(4, 3)`. We can consider this data as recording a total of three features over the course of four timesteps.

Features	**Timesteps**			
	1	**2**	**3**	**4**
Temperature (average)	12.2°	14.8°	13.0°	12.2°
Rain (total)	31.7 mm	77.5 mm	87.1 mm	77.6 mm
Wind (average)	2.3 m/s	2.0 m/s	2.1 m/s	1.9 m/s

Table 4.1: Weather data over time for Trondheim in Norway

The output of a typical recurrent neural network has a shape of `(units)`, where the value of `units` is the number of output units in the network. Considering the compact network in Figure 4.2, the output of a recurrent neural network is the output of the last iteration of the loop. One may also use the output of every iteration in the loop, which results in an output shape of `(timesteps, units)`, where the dimensionality of the timesteps in the output is equal to the dimensionality of the timesteps in the input.

4.2.2 Long-Short Term Memory

Long-Short Term Memory (LSTM) is a recurrent neural network architecture. It was first proposed by Hochreiter and Schmidhuber (1997) and was meant to address some of the shortcomings of more basic recurrent neural network architectures. Bengio et al. (1994) showed that recurrent neural networks faced an increasingly difficult problem as the duration of the dependencies to be captured were increased. While the architecture

[1] `https://www.yr.no/sted/Norge/S\T1\or-Tr\T1\ondelag/Trondheim/Trondheim/statistikk.html`

could take into account short-term dependencies rather well, long-term dependencies were increasingly difficult to learn. The LSTMs were explicitly designed to avoid the long-term dependency problem.

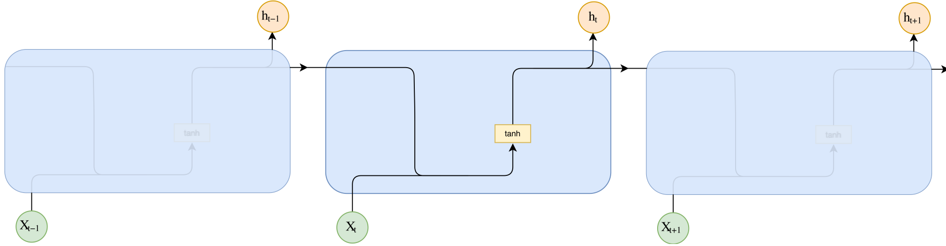

Figure 4.3: Flow of a basic recurrent neural network

Figure 4.3 illustrates the chain like structure of a basic recurrent neural network. Its architecture is relatively simple, containing only one layer. In this illustration, the layer uses the hyperbolic tangent function (tanh) as its activation function.

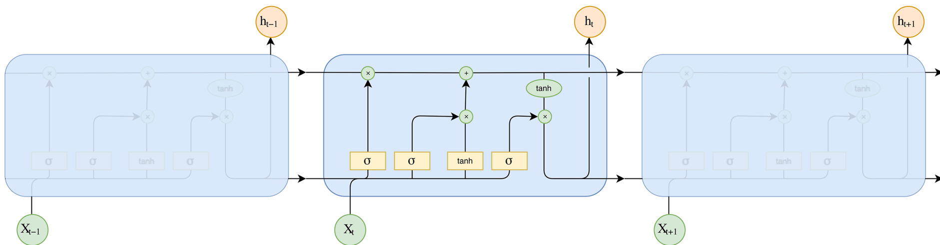

Figure 4.4: Flow of an LSTM cell

Figure 4.4 illustrates a similar chain structure, but that of an LSTM cell. The LSTM module has a total of four neural network layers. The topmost horizontal line carries the cell state of the unit, while the horizontal line at the bottom carries the hidden state. The LSTM unit is enriched by several so-called gating units. These gates regulate what information is remembered by the cell state, and what is forgotten. The leftmost sigmoid layer is called the "forget gate layer." This gate decides what the state should forget from the existing information. The next sigmoid gate is called the "input gate layer" and determines which values should be updated. The hyperbolic tangent gate layer creates a vector of candidate values, which could be added to the cell state. After the state is updated, or the candidate values are thrown away, the final sigmoid gate, the "output gate layer," decides which parts of the cell state should be outputted (Hochreiter and Schmidhuber, 1997; Goodfellow et al., 2016; Olah, 2015; Gers et al., 2002). The first proposed version of the LSTM did not have a forget gate. It was introduced by Gers et al. (2000) and allowed the LSTM to reset its state. This version has since become one of the most common variants of the LSTM.

Variants of LSTM

The LSTM implementation described in the previous section is one of the traditional LSTMs. There exist other variants of the architecture with other characteristics. One popular LSTM variant was introduced by Gers and Schmidhuber (2001). Their variant added "peephole" connections. These connections allow the gate layer to look at the cell state. The idea behind this variant was to have an LSTM that could learn to reset its memory contents selectively, and in turn, produce stable results in the presence of never-ending input streams. Gers and Schmidhuber (2001) stated that their LSTM variant with peephole connections and forget gates was superior to the traditional LSTM.

Another variant is the "Convolutional LSTM," proposed by Xingjian et al. (2015). Their variant extended the traditional LSTM and added convolutional structures to both the input-to-state and state-to-state transitions. Their conclusion was that their proposed "ConvLSTM" layer was suitable for spatiotemporal data due to its inherent convolutional structure.

A comparison of various LSTM variants was carried out by Greff et al. (2016). They concluded that the traditional, vanilla LSTM, performed reasonable well on various datasets. They investigated a total of eight variants of the LSTM, and in their experiments, none of the eight modifications significantly improved performance. However, certain modifications simplified the LSTMs without significantly decreasing performance.

4.2.3 Gated Recurrent Unit

Another popular modification of the LSTM was proposed by Chung et al. (2014). Their simplified variant called the Gated Recurrent Unit, or GRU for short, uses neither peepholes connections nor output activation functions. Instead, the GRU couples the input and the forget gate into an update gate. The GRU also merges the hidden states and the cell states (Greff et al., 2016; Chung et al., 2014). Comparisons between the LSTM and the GRU units have shown mixed results, and experiments have concluded that there is no clear winner between the two (Greff et al., 2016; Chung et al., 2014). Jozefowicz et al. (2015a) compared various LSTM and GRU units and concluded that GRUs outperformed the LSTM on all tasks except language modeling.

4.3 Encoder-Decoder Framework

The encoder-decoder framework is a concept centralized around two recurrent neural networks. The idea is to encode the input in the first neural network and decode it in the second neural network. The first recurrent neural network, also called the encoder, reads the input sentence, a sequence of vectors $X = (x_1, x_2, \dots, x_n)$. This sequence is then encoded into a vector c, which may or may not be of fixed length (Sutskever et al., 2014; Cho et al., 2014b).

The decoder is often trained to predict the next word $y_{t'}$ given the context vector c and all the previously predicted words $y_1, \dots, y_{t'-1}$. Bahdanau et al. (2014) summarize the architecture with:

The decoder defines a probability over the translation y by decomposing the joint probability into the ordered conditionals:

$$p(y) = \prod_{t=1}^{T} p(y_t \mid \{y_1, \ldots, y_{t-1}\}, c), \tag{4.3}$$

where $y = (y_1, \ldots, yT_y)$. With an RNN, each conditional probability is modeled as

$$p(y_t \mid \{y_1, \ldots, y_{t-1}\}, c) = g(y_{t-1}, s_t, c), \tag{4.4}$$

where g is a nonlinear, potentially multi-layered, function that outputs the probability of y_t, and s_t is the hidden state of the RNN.

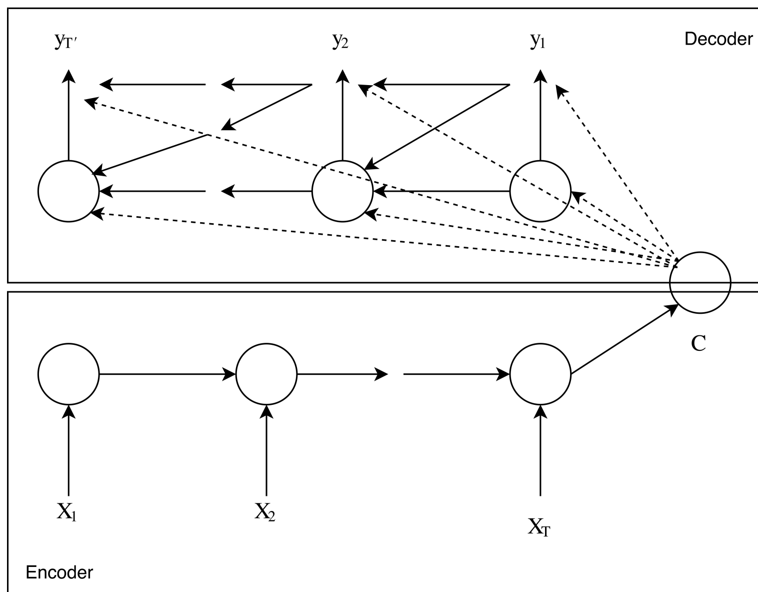

Figure 4.5: An illustration of the proposed RNN encoder-decoder

This model, as introduced by Cho et al. (2014b), has been used in sequence-to-sequence problems with great results. Their proposed recurrent neural network encoder-decoder model is illustrated in Figure 4.5. Sutskever et al. (2014) proposed a similar encoder-decoder model, but used multi-layer cells in their sequence-to-sequence model. In the description of their proposed model, Sutskever et al. (2014) states:

> The RNN can easily map sequences to sequences whenever the alignment between the inputs the outputs is known ahead of time. However, it is not clear how to apply an RNN to problems whose input and the output sequences have different lengths with complicated and non-monotonic relationships.

The encoder-decoder model has eliminated the problem of unknown alignment between input and output. This model has therefore been found suitable for various problems with these characteristics, such as natural language processing, speech recognition, and computer vision.

4.3.1 Attention Mechanism

In the encoder-decoder framework, the first neural network has to compress all the necessary information of a source input into a fixed-length vector. Bahdanau et al. (2014) conjectured that the use of a fixed-vector was a bottleneck in improving the performance of the basic encoder-decoder model. Their proposed model extended the vector encoding by allowing the model to automatically soft-search for parts of the input to attend during decoding. With the attention mechanism, the decoder does not have to rely solely on the information in the encoded context vector, as it can supplement with information directly from the input data.

Tests of the proposed model on the task of English-to-French translation revealed that the model outperformed conventional encoder-decoder models significantly regardless of sentence length (Bahdanau et al., 2014). Bahdanau et al. (2014) also concluded that the attention mechanism was more robust to the length of a source sentence. Similar attention mechanisms have since then been applied to other models with improved results (Hsu et al., 2016; Sankaran et al., 2016).

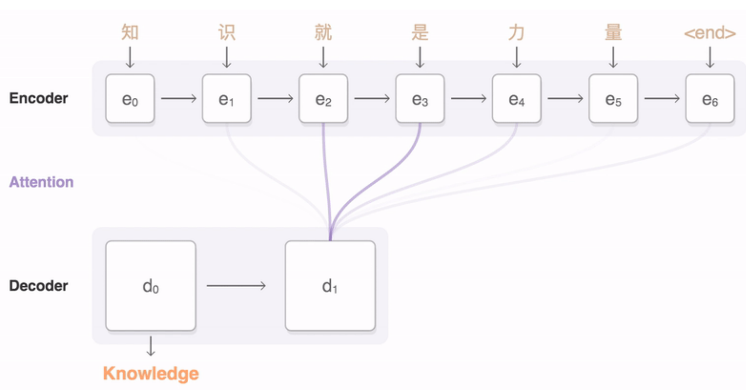

Figure 4.6: Encoder-decoder with an attention mechanism

Figure 4.6 illustrates a translation task between Chinese and English with an attention mechanism. The lines between the encoder and decoder indicate the degree of "focus" the attention mechanism pays to each part of the input. As the decoding process progresses, the area the attention mechanism attends to shifts.

4.4 Vocabulary Encoding

One-hot vectors and word embeddings are two different ways of representing a vocabulary mathematically. Vocabulary representation is necessary when working with textual data, for example in translation tasks. In this section, we look closer at pros and cons of these two encoding methods.

4.4.1 One-Hot Vector Encoding

One-hot vector encoding is a common way to represent a vocabulary. In this encoding, we create a binary column for each unique word in the vocabulary. To represent a word we set all the binary values to 0 except the column that corresponds to the unique word. Table 4.2 illustrates a simple one-hot encoding of a short sentence with a minimal vocabulary.

However, this encoding method becomes troublesome when the vocabulary is big. If the vocabulary has a total of 90,000 unique words, the one-hot vector would correspondingly need to have a size of 90,000. This vector would also consist almost exclusively of zeroes. Besides, one-hot encoded vectors neither define relationships between nearby or related words nor define any notion of semantic.

Sample	**is**	**the**	**machine**	**working**
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

Table 4.2: One-hot encoding of the sentence “is the machine working”

4.4.2 Word Embeddings

An alternative to one-hot encoding is word embedding. Word embedding, also known as a type of vector space model, is a way to map vocabulary to vectors of real numbers. The concept of distributed representation for symbols dates back several decades (Hinton, 1986), although the most common approaches usually follow a more modern model (Bengio et al., 2003). The goal of word embedding is to find a representation that does not suffer from the *curse of dimensionality*, problems that arise when organizing data in high-dimension space.

The embedding is done with some parameterized function which maps the word to high-dimensional vectors. For example:

$$W(\text{"home"}) = (0.1, 0.3, 0.0, \ldots)$$
$$W(\text{"sun"}) = (-0.7, 0.6, 0.1, \ldots)$$

Usually, the W is initialized randomly, which places the words randomly in the vector space. During training of the model, the embeddings are usually trainable, and the model learns to have meaningful placement of the vectors.

Word embedding itself is a collective name for the encoding technique, and there exist different predictive models for learning word embeddings from raw data. One such model is Global Vectors for Word Presentation[2] or `GloVe` for short. With `GloVe` embeddings, one can measure the linguistic or semantic similarity by calculating the Euclidean distance between two words.

[2] https://nlp.stanford.edu/projects/glove/

Chapter 5

Related Work

This chapter looks closer at related work carried out in the field of machine translation. We present the history and progress of machine translation in Section 5.1. In section 5.2 we present statistical machine translation, and neural machine translation is presented in Section 5.3. We both present some early work and establish the current state-of-the-art in the field.

5.1 Evolution of Machine Translation

In 1954, a translation system developed at Georgetown University was demonstrated for the first time. The system did a completely automatic translation of more than sixty sentences from Russian to English. One of the creators, Léon Dostert, predicted that automatic text-reading translation machines would be finished within three to five years (Hutchins, 1997). As research continued, the complexity of the linguistic problems became more and more apparent. Critics argued that the concept of fully automatic high-quality translator that could produce translations indistinguishable from those of humans translators were impossible in principle (Hutchins, 2007).

National Science Foundation established the Automatic Language Processing Advisory Committee (ALPAC) in 1964 to carry out a study of the realities of machine translation. In 1966 they published their report that concluded that the use of machine translation was slower, less accurate, and twice as expensive as human translation. The report also stated that there were no immediate or predictable prospect of useful machine translation (Hutchins, 2007; Council et al., 1966; Koehn, 2010). The ALPAC report led to the U.S government reducing their funding in the field dramatically.

It was first in the later half of the 1970s, and the early 1980s, that machine translation again saw a rise in popularity. The latter half of the 1980s also saw a general revival in interest in Interlingua systems. This interest was motivated in part by artificial intelligence, which was also a research field that attracted much attention. Since the 1980s, new methods such as corpus-based approaches and statistical machine translation based systems have emerged. Speech translation has also seen growing interest since the late 1980s (Hutchins, 2007).

Figure 5.1: The Google Translate app and its image translation feature

In more recent years, online translation services such as Google Translate[1] and Yahoo!'s Babelfish[2][3] has gained much popularity. Both services offer on-demand translation for free (Mike Schuster and Thorat, 2016; Hutchins, 2007). Google reported on their blog in 2016 that their service supported over 100 languages, had more than 500 million users, and translated more than 100 billion words a day (Turovsky, 2016b). The native app for Google Translate has also become very popular. It offers the same functionality as their online counterpart, in addition to other features, such as "Word Lens" which can translate images in place (see Figure 5.1).

5.2 Statistical Machine Translation

For the past three decades or so, machine translation has taken a new direction. Instead of pre-defined rule-based systems, many modern machine translation systems attack the problem with statistical methods and ideas from information theory (Brown et al., 1990). Statistical machine translation (SMT) was born as an idea in the 1980s in the labs of IBM Research. The idea came in the wake of the success of statistical methods in speech recognition. The idea was to model the translation task as a statistical optimization problem. Some of the best performing SMT systems today are phrase-based, an approach where the input sequence is broken up into a sequence of phrases, and these phrases are mapped one-to-one to output phrases, which may be reordered (Koehn, 2010, pp. 127–128). Figure 5.2 illustrates a phrase-based translation of a sentence in German to English.

Statistical machine translation has been the dominant translation paradigm for decades (Wu et al., 2016). Variants of SMT-based systems have achieved state-of-the-art performance in machine translation (Watanabe et al., 2007). SMT systems also exist on the

[1] https://translate.google.com
[2] https://www.babelfish.com
[3] The service now relies on translations from the Bing Translator, but the service is otherwise unaltered.

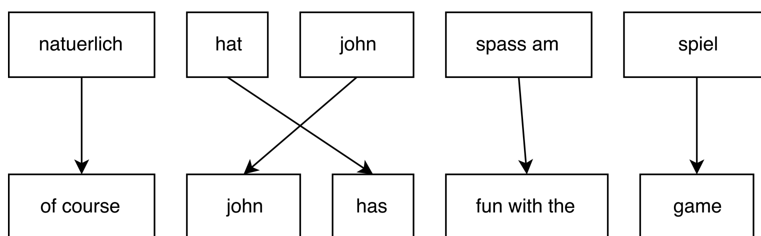

Figure 5.2: Phrase-based translation of a sentence in German to English

commercial market, a market long dominated by the well-established, and older, rule-based methods (Hutchins, 2007).

5.3 Neural Machine Translation

Neural machine translation (NMT) is another approach to machine translation that has emerged recently. The concept of neural machine translation is to build a jointly-tuned single neural network which is trained to maximize translation performance. This approach is a different from traditional statistical machine translation systems, which usually consist of sub-components that are optimized separately (Wołk and Marasek, 2015). The benefit of neural machine translation systems is its ability to learn directly, in an end-to-end fashion.

In 2016, Google published their work on Google's Neural Machine Translation system, or GNMT for short. This system replaced their older statistical machine translation system that previously ran Google Translate (Turovsky, 2016a). GNMT uses the common sequence-to-sequence learning framework as proposed by Sutskever et al. (2014); Wu et al. (2016). Their implementation is closely related to the work of Kalchbrenner and Blunsom (2013) who were the first to map a input sentence into a vector, and then back into a sentence using the encoder-decoder framework. It also builds on the neural network architecture presented by Cho et al. (2014b). Cho et al. (2014b) did their translation using LSTM-like RNN architecture, although their primary focus was to integrate their neural network into an SMT system (Cho et al., 2014b; Sutskever et al., 2014). The proposed model of Sutskever et al. (2014) did the entire translation end-to-end and was not integrated with any other frameworks or systems. Their implementation achieved close-to-best results in an English to French translation task and outperformed various SMT-based systems.

The encoder-decoder approach, as shown in Figure 5.3 has also been used as the foundation for various other NMT architectures. Chung et al. (2016) proposed an encoder-decoder based model dubbed *bi-scale recurrent neural network*, which handled multiple timescales in a sequence better than previous models. Their model did the translation on character-level instead of the more common approach of translating on word-level. They proved that their decoder, which was fed sequences of characters without any explicit word segmentation, was able to translate at the level of characters, and that the model benefited from the approach.

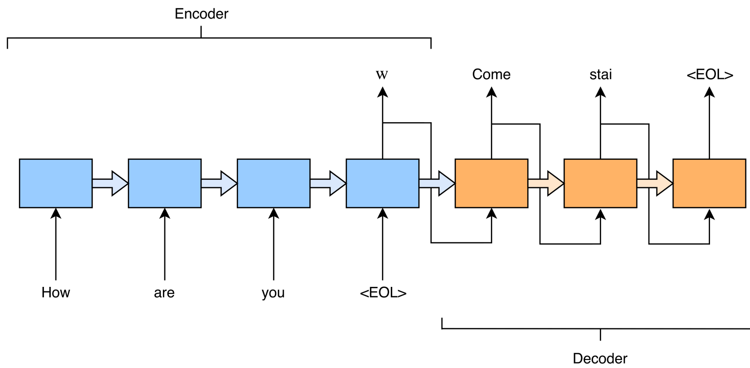

Figure 5.3: General translation approach in neural machine translation from an English sentence to Italian

Bahdanau et al. (2014) proposed another model, as already introduced in Section 4.3.1, which tried to solve the performance bottleneck related to the encoded vector. Cho et al. (2014a) had already shown that the encoder-decoder model had problems with longer dependencies. Their analysis also introduced a novel network dubbed *gated recursive convolutional neural network*, which was evaluated alongside more traditional encoder-decoder models. Their evaluation showed that both architectures performed relatively well on short sentences, but suffered significantly as the length of the sentences increased. The model proposed by Bahdanau et al. (2014) instead learns to align and translate jointly. It does this by encoding the input sequence into a sequence of vectors and chooses a subset of these vectors adaptively while decoding the translation. This model outperformed the basic encoder-decoder significantly in their experiments.

Despite being a relatively new approach, neural machine translation has achieved impressive results. In many machine translation tasks, they have obtained state-of-the-art performance, outperforming statistical machine translation systems that have matured over several decades. Bahdanau et al. (2014) stated that most of the proposed neural network translation models belonged to a family of encoder-decoders, making it the key factor for this success in many models. Recently, new work has been published that addresses other problems and bottlenecks with the encoder-decoder architecture, such as the rare word problem, increasing performance even further (Sennrich et al., 2015). The attention mechanism has also shown promising results since it was first introduced, achieving new state-of-the-art results in certain translation tasks (Luong et al., 2015).

Part III

System Design, Models, and Experiments

Chapter 6

System Design

The main objective of this thesis is to recognize signature sequences by creating a fully functional system, as established by our choice of research strategy in Chapter 2 and method in Chapter 3. In this chapter, we present the design of the system we made. We present the general design in Section 6.1, while Sections 6.2 and 6.3 presents the most essential components, namely the `Preprocessor` and the `Transformator`.

6.1 General Design

Our models use supervised learning, a task which revolves around two phases: training and testing. We designed our system specifically with this in mind.

The system consists of several separate modules, each responsible for its separate task. The two first modules are the `Trainer` and the `Tester`. These modules are responsible for starting whatever other modules are required to run before the actual training or testing can take place, as well as invoking the action on the correct model. The `Preprocessor` is responsible for creating the datasets we use in both the training and testing phases. The `Transformator` module "transform" the input and label data stored from the `Preprocessor` into a format that the models expect. Finally, the system calls the `Predictor` which loads the correct model and either starts training or testing it.

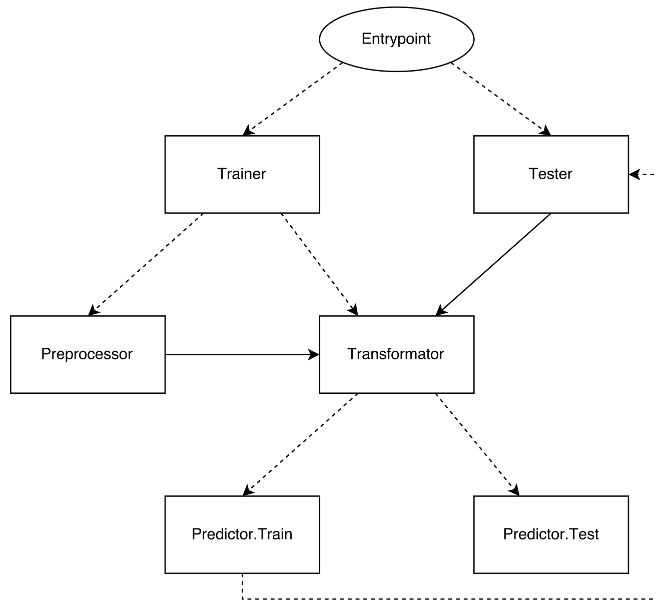

Figure 6.1: Simplified module interaction overview

Figure 6.1 presents a simplified overview of the system design, and how the modules fit together. A solid line indicates a direct route from one module to another, and a dashed line indicates that the module can take several routes. For example, the `Trainer` module can either create a new dataset by calling the `Preprocessor` or reuse an existing dataset and call the `Transformator` instead.

6.2 Preprocessor

As there were no existing datasets suitable for our particular problem, we decided to create these ourselves. Creation of these datasets was done by implementing a configurable pipeline system that involved a process in several steps:

1. Build a complete dictionary of words, make sure they only include allowed characters.

2. Select random words from the list and construct training, validation, and test sets. Make sure no words exceeds the maximum word length configuration. Remove duplicated words both across lists and within the same list.

3. Write the text from the lists on own (empty) canvases, with the specified font type and font size.

4. Find the boundaries of the characters and crop the text to remove the excessive space on the canvas.

5. Apply the masks and extract the signature values.

6. Save the final output.

Figure 6.2: Illustration of the pipeline process from canvas to signature

Figure 6.2 illustrates the steps done when writing out, cropping, and applying our signature to a word. First, we write out the word on a big, white canvas. This canvas has a predefined size, as we are unable to predict the actual size of the text before it is written out. The canvas is then cropped based on the boundaries calculated from the individual characters. Finally, we apply the masks to the cropped image and extract the pixels in the signature. In Figure 6.2, the rightmost subfigure illustrates the extracted signature from the middle subfigure, which has a height of one pixel and is masked fourteen pixels from the bottom. The signature was expanded vertically for increased visibility. The final signature sequence is extracted by iterating over the pixels in the image and is stored for later. In the illustrated example, the word "AMPLITUDES" results in a signature sequence that is 207 pixels long. This signature is stored as a binary vector with the same length.

6.2.1 Multifont and Casing

Special functionality was implemented in the pipeline process to create datasets with multiple fonts. The pipeline chooses one of N fonts, with an equal probability distribution ($\frac{1}{N}$) for each font.

Another method was implemented to change the casing of the words picked from the datasets. If the system is configured to include both upper-case and lower-case letters, the system will capitalize all the letters with probability 0.5.

6.3 Transformator

The `Transformator` module was created to transform the sequences of binary data from the `Preprocessor` into a format that the models expected. During development, especially in the early stages, the formats the models expected their inputs and output in varied. It, therefore, made sense to split the `Preprocessor` and the `Transformator` into separate modules which are executed in sequence. This approach allowed us to reuse the same dataset on multiple models that had different input and output format expectations.

The `Transformator` is given a sequence of "handlers," depending on which model is to be loaded in the `Predictor`. These handlers are executed in sequence, and each handler does a modification to either the input or the output format, and the data is

propagated to the next handler. This way, the handlers could easily be reordered or swapped if a model expects another format. Typical tasks include padding the input to a given width, re-scaling the input to unsigned integers, or turning the output into a one-hot matrix.

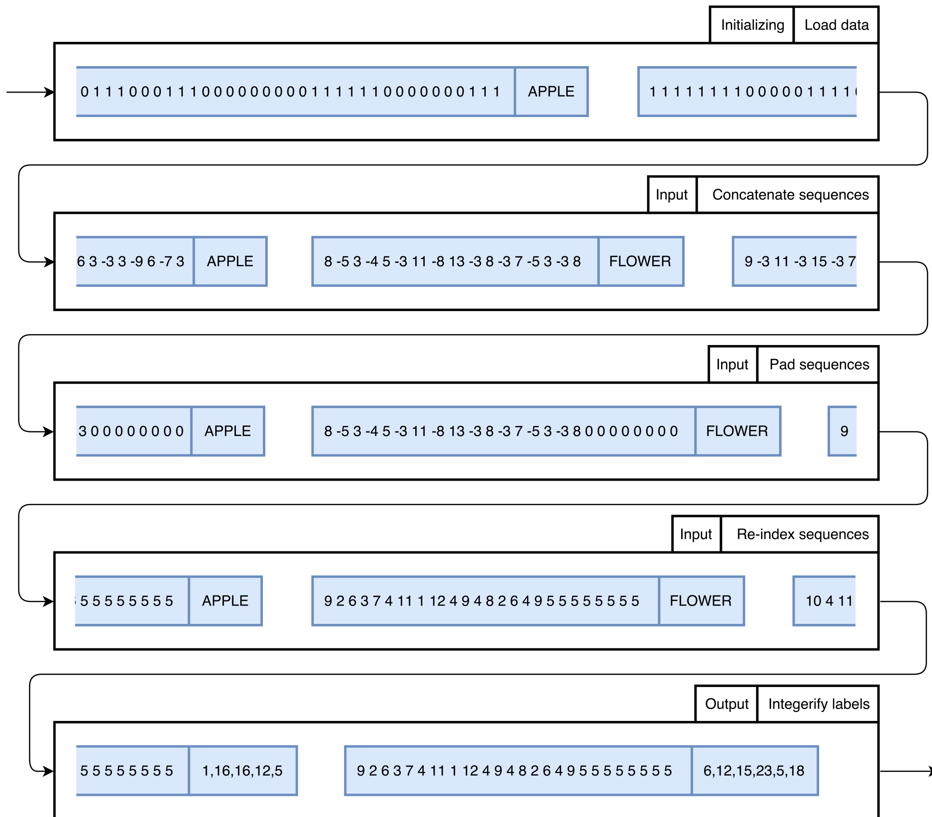

Figure 6.3: Illustration of handlers transforming data

Figure 6.3 illustrates a sequence of handlers, each transforming the data in one specific way. The first handler initiates the process by loading the binary data from the `Preprocessor`. The second handler concatenates the binary data into sequences, where positive integers indicate a series of white pixels, and a negative integer indicates a series of black pixels. The third handler applies padding, to ensure that all the data has the same width before feeding it to the models. The padding is done by appending zeros at the end of the original sequence. The second last handler re-indexes the data, as our models expect unsigned integer values. The final handler transforms the labels from letters to an integer value, here we have given each letter the value that corresponds to their number in the English alphabet.

6.3.1 Noise

A special handler was implemented to add noise to the data. This handler iterates over the binary bits from the raw pixel data, and with a given probability sets a random bit value 1 or 0. The handler accepts a threshold in percent, which governs how often the bits can be randomized. Because the handler does not take into account the original value of the bit, a handler with a threshold T will on average change a bit-value with probability approximately $\frac{T}{2}$.

Chapter 7

Models

In this chapter, we present the three models we created during our iterative development process. We present the `VecRep` model in Section 7.1, and our two encoder-decoder based models `EncDecReg` and `EncDecAtt` are presented in Sections 7.2 and 7.3.

7.1 Repeat Vector

The first model, called `VecRep` for short, has a similar structure to that of the encoder-decoder framework. This model consists of two groups of LSTMs. The first group reads the entire input and outputs its value from the last iteration. This output is then repeated in the dimension of time and inputted into a new group of LSTMs.

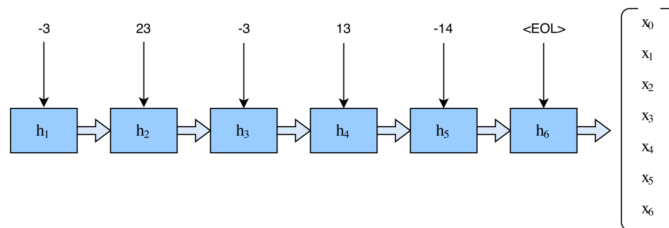

Figure 7.1: An LSTM outputting its final output after reading a sequence

Figure 7.1 illustrates a regular LSTM that reads an input sequence and outputs the values from the last iteration. The output vector has a length equal to the number of units in the LSTM. As illustrated in Figure 7.2, the output from the LSTM is repeated N times, where N is the length of the output sequence. This means we take an output shape of `(units)` from the first group of LSTMs, and turn it into a shape of `(N, units)`.

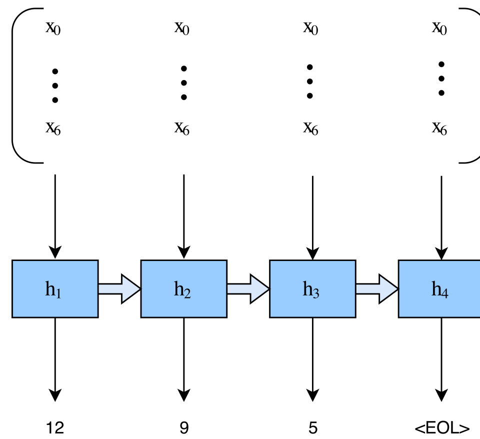

Figure 7.2: Repeating the output from an LSTM and feeding it to another LSTM

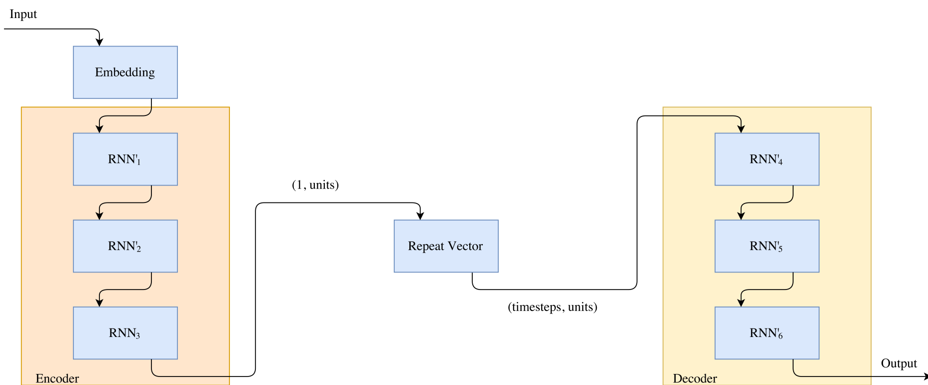

Figure 7.3: Simplified illustration of the `VecRep` model

The `VecRep` model is illustrated in Figure 7.3. The “encoder” and “decoder” both consists of three recurrent units. These units are stacked sequentially, which means that the first cell, RNN'$_1$ completed all its iterations, and the output from this cell is then fed to the next RNN cell which does the same. The recurrent cells that are marked, such as RNN'$_1$, indicates that the cell returns its output for every iteration, whereas a cell that is not marked only outputs from the last iteration. The “encoding” in this model is done by the RNN$_3$ cell, which only outputs in the last iteration. This approach is somewhat similar to the encoded context vector in the encoder-decoder framework. Between the two stacks of recurrent units is the layer which repeats the final output of RNN$_3$ in the dimension of time. Embedding is used to represent the input data, whereas one-hot vectors represent the output.

7.2 Regular Encoder-Decoder

The second model implements the encoder-decoder framework. It is called `EncDecReg` as it is the "regular" encoder-decoder, unlike the last model. This model has certain similarities to the `VecRep` model; also this one consists of two groups of LSTMs, where the first group encodes the input and the last group decodes.

However, in this model, we use the last hidden state of the encoder, instead of the output as with the `VecRep` model. During testing and validation, the decoder feed its output back in as input. However, feeding the output back as input is not used during training, where we use the actual labels instead. This is similar to the approach taken by Bengio et al. (2015).

This model also uses two embeddings, one for the input and one for the output. The input is embedded before the encoder reads the input, and the output embedding is applied on the fly when the decoder is reusing its output as input. This representation is different from the `VecRep` model which only used embedding for the input data. Both the embeddings in this model is fully trainable.

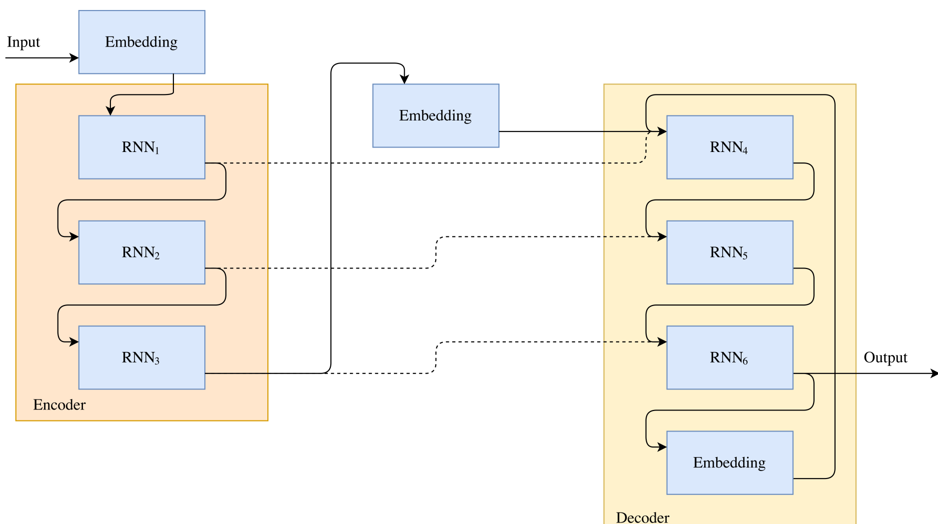

Figure 7.4: Simplified illustration of the `EncDecReg` model

The `EncDecReg` model is illustrated in Figure 7.4. The stacked recurrent units are handled differently in this model compared to the `VecRep` model. In the `VecRep` model, each RNN cell iterates all the timesteps in the input and passes its output to the next RNN cell which does the same, whereas in this model the RNNs are called sequentially for each timestep. With this sequentiality, the input for every timestep is first fed to RNN_1, then to RNN_2, and so on. This approach results in a cell that consists of multiple layers, whereas `VecRep` has multiple cells in sequence. The last hidden state of the encoder RNN layers is passed as the initial hidden states for the RNN layers in the decoder, illustrated with a dashed line. In the decoder, we both output from RNN_6, while also passing the output to an embedding layer and feeding it back into the first RNN

layer in the decoder. Note that the embedding between the encoder and decoder, and the embedding below the RNN_6 layer is the same embedding cell.

7.3 Attention Encoder-Decoder

The last model also implements the encoder-decoder framework, similarly to the `EncDecReg` model. However, this model also implements the attention mechanism and is named `EncDecAtt` for short. As explained in Section 4.3.1, the attention mechanism allows the decoder to peek into the input by providing a list of values it can attend to. The implementation of the attention mechanism used in this model is based on Vinyals et al. (2015).

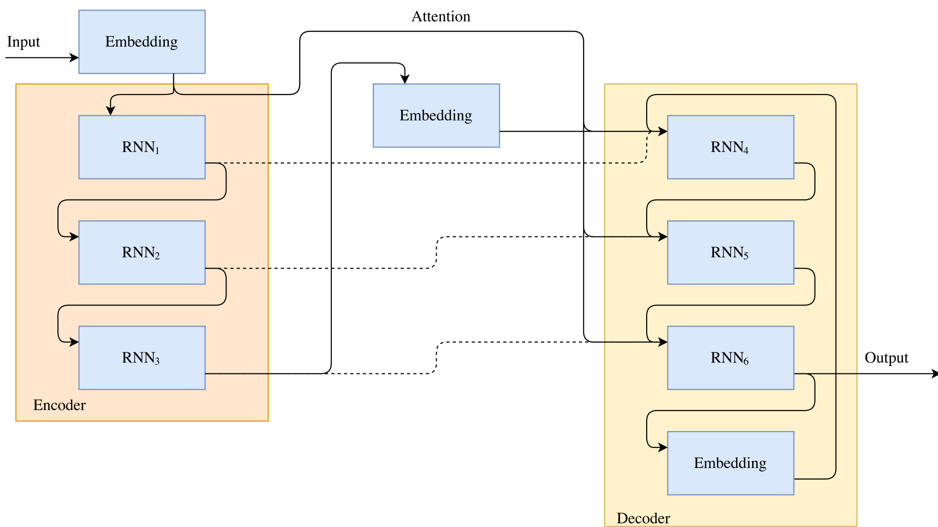

Figure 7.5: Simplified illustration of the `EncDecAtt` model

The `EncDecAtt` model is illustrated in Figure 7.5. The difference between this model and the `EncDecReg` model is the additional input for the RNN layers in the decoder cell. This data is fetched directly from the input embedding and allows the attention mechanism to attend to this data, in addition to the hidden states from the encoder cell.

Chapter 8

Experiments

In this chapter, we present the final experiments that were carried out as a part of this thesis. Section 8.1 presents the general experiment approach. We present the setup of our models in Section 8.2. Section 8.3 presents details about the datasets, and Section 8.4 presents the experiment details.

8.1 Approach

Our experiments followed a traditional pattern with individual training, validation, and testing datasets. Our models were first fed the entire training set with the correct labels. After iterating the entire training set, we let the models predict on the validation set, recording their calculated accuracy and loss values. This alternating process went on until the model had reached a convergence where the loss for validation no longer made significant changes. As this process went on, we saved the weights for the model with the lowest validation loss value. When the alternating training and validation process was stopped, we loaded the weights for the best model, and tested that model on the testing dataset, recording the final accuracy.

8.2 Setup

In this section, we present the setup, configurations, and choice of hyper-parameters in our three models.

- All three of our models used embedding to represent the input. This embedding was represented in a vector space with dimension 128. Using a higher dimension representation, like 1024 gave tiny improvements in results, but increased the computational time tremendously. A dimension size of 128 was a good trade-off, and made sense for our relatively small vocabulary. Britz et al. (2017) concluded that using high embedding dimensions, such as 2048 achieved the best results, but only by a small margin. They also concluded that embeddings with smaller margins, such as 128, seem to have sufficient capacity to capture most of

the necessary semantic information in their vocabulary. This vocabulary was significantly larger than ours. The encoder-decoder models also used embedding to represent the output.

- The embeddings were randomly initialized with an uniform distribution over the half-open interval $[-\sqrt{3}, \sqrt{3})$. As the embeddings were trainable, their placement in the vector space would change, which meant that the original placement was less important.
- Both LSTMs and GRUs have been used in neural machine translation architectures. We decided to use the LSTM cell in all our networks.
- The LSTM units were based on the implementation of Hochreiter and Schmidhuber (1997). The inner activation function was hyperbolic tangent (see Equation 8.1 and plot in Figure 8.1), and the activation applied over the current timestep was a regular sigmoid function (see Equation 8.2 and plot in Figure 8.2). These configurations are pseudo-standard for the LSTM, although some variations exist.
- The weights in the LSTMs were initialized with "Glorot uniform initializer," also called "Xavier uniform initializer" (Glorot and Bengio, 2010).
- All three models used groups of LSTMs. These had a depth of three. Smaller and bigger stacks affected results slightly, but a depth of three was a good trade-off between computational time and results.
- The forget gate in the LSTM module was initialized to 1.0. This was done to reduce the scale of forgetting at the beginning of training and is something that is recommended by Jozefowicz et al. (2015b).
- After each LSTM cell, we had a dropout layer, which randomly dropped units, along with their weights, by setting their values to 0 (Srivastava et al., 2014). The dropout rate after *each* LSTM was 0.2, which meant that around 20% of all output was ignored after each LSTM unit. This rate was a result of trial-and-error, but the rate did not significantly affect results in the range of ± 0.1.

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (8.1)$$

$$\frac{1}{1 + e^{-x}} \quad (8.2)$$

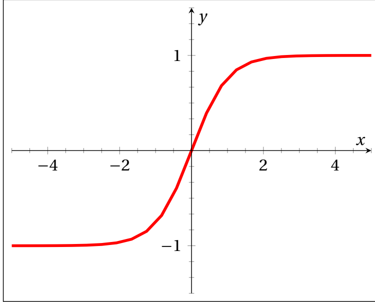

Figure 8.1: Plotted hyperbolic tangent function

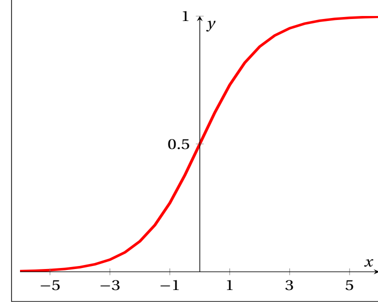

Figure 8.2: Plotted sigmoid function

8.2.1 Accuracy Metric

Accuracy was calculated by computing the categorical accuracy for each label. This computation was done by comparing the categorical output of the prediction with the one-hot vector representing the correct label in the dataset.

8.2.2 Loss Optimizer

We tested various loss optimizers for our three models. Adaptive Moment Estimation (Adam) (Kingma and Ba, 2014) and Adadelta (Zeiler, 2012) are two loss optimizers, among many, that have been applied to encoder-decoder problems earlier (Cho et al., 2014a; Arik et al., 2017). Our choice of optimizer was decided by trial-and-error and ultimately fell on Adam.

The Adam optimizer is a method that computes adaptive learning rates for each parameter. We kept the proposed default values of $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$. We used a learning rate of 10^{-2}, instead of the more common rate of 10^{-3}. We increased the learning rate as all models had a tendency to use many epochs to reach satisfactory progression.

8.3 Dataset Details

Datasets were created using the pipeline process explained in Section 6.2. Our word list had a total of 356,719 unique words, populated by three open sourced word lists:

1. sil.org[1] (109,582 words)
2. Public GitHub repository dwyl/english-words[2], only the file "words.txt" (354,985 words)
3. The English dictionary (only en_US) for Apache OpenOffice[3]. Preprocessed (39,908 words)

Only words consisting of actual letters were added, and the words had to have a minimal length of two characters. Duplicates were removed.

8.4 Experiments

This section presents the various experiments we conducted to evaluate if we reached our research goal and to answer our research questions.

8.4.1 Accuracy on Datasets

The first, and simplest experiment we did was to check how well our models could do prediction on datasets of various sizes. For this experiment, three datasets were created. These sets had the following sizes and configurations:

Small: 1,000 training, 100 validation, 100 testing, max 10 chars

Medium: 5,000 training, 500 validation, 500 testing, max 15 chars

Big: 25,000 training, 2,500 validation, 2,500 testing, max 20 chars

Our goal with this experiment was to record how the recognition rates for each model were affected by the size of the datasets. This experiment could potentially say something about the robustness of the models and how well they were able to learn from either a limited or big dataset. Our expectations for this experiment was that all three models should have relativity good accuracy across all datasets. We suspected that the accuracy would increase with the size of the datasets.

[1] http://www-01.sil.org/linguistics/wordlists/english/
[2] https://github.com/dwyl/english-words
[3] http://extensions.openoffice.org/en/project/english-dictionaries-apache-openoffice

8.4.2 Handling of Two Fonts

In this experiment, we used text written in two fonts. The previous experiment had text written in only monospaced Arial (example in Figure 8.3), while this experiment also used the font Times New Roman (example in Figure 8.4). The dataset used in this experiment had a training set of size 10,000 and validation and test sets of size 1000. The sets had a maximum length of 15 characters. The distribution between the two fonts are listed below:

Training: Arial: 51.1% (5,110), Times New Roman 49.0% (4,900)

Validation: Arial: 51.7% (517), Times New Roman 48.3% (483)

Test: Arial: 51.2% (512), Times New Roman: 48.8% (488)

Figure 8.3: The word “SHORTCUTS” written in Arial (monospaced). Size 186x31

Figure 8.4: The word “UPLIFTER” written in Times New Roman. Size 180x31

Our goal with this experiment was to record how the individual models were able to handle input written in more than one font. We expected the two encoder-decoder models to perform better in this experiment than the `VecRep` model. However, we did not exclude that the `VecRep` model could reach satisfactory results either.

8.4.3 Noise Handling

In the third experiment, we added more noise increasingly to the input data. This experiment was conducted to record how resilient and robust the model was to noise in the data. For this experiment, we only used the best performing model, based on the results of the previous experiments. The amount of noise added for each test was based on results underway, as we did not know how the model would respond to the noise beforehand.

The goal of this experiment was to record how the accuracy decreased while the amount of noise increased, and we expected the model to handle noise to a degree we found satisfactory.

8.4.4 Stress Test

The final experiment was a "stress test" where we increased the complexity of the task in various ways. For this experiment, we used a total of five fonts:

- Arial (monospaced)
- Times New Roman
- Courier
- Georgia
- Verdana

These fonts are variants of serif fonts (Georgia and Times New Roman), sans-serif fonts (Arial monospaced, Verdana), and monospaced fonts (Arial monospaced, Courier).

This experiment had words written in both upper-case and lower-case letters, in contrast to the other experiments which only had upper-case letters. The other experiments, having exclusively upper-case letters, had a total of 27 classes (A-Z plus an "empty" class), whereas this experiment would have a total of 53 classes (A-Z, a-z, and the "empty" class). The "empty" class was used to pad the values as the words had different lengths. The datasets had a size of 50,000, 5,000, and 5,000 for the training, testing, and validation sets respectively. We also added a noise factor of 10%. Some of the generated words are listed in Appendix B.

The goal of this experiment was primarily to see just how far we could go with the models. We had little information on how the two encoder-decoder models would perform under these conditions. Our general expectations for this experiment was for the `EncDecAtt` model to outperform the `EncDecReg` model.

Part IV

Results, Discussion, and Conclusion

Chapter 9

Results and Discussion

This chapter presents the results of the experiments we conducted. Section 9.1 presents the results of the experiment with three datasets of various sizes. The results of the experiment with two fonts are presented in Section 9.2. In Section 9.3 we present the results of the `EncDecAtt` model's robustness to noise. The final "stress test" results are presented in Section 9.4, and all the results are discussed in Section 9.5. Finally, in Section 9.6 we analyze the models in the context of our results.

9.1 Accuracy on Datasets

Table 9.1 contains the accuracy for each model on each of the three datasets of various sizes. The accuracy and loss plots for each test are presented next, showing the progression over the epochs.

	Small dataset	**Medium dataset**	**Big dataset**
`VecRep`	16.80%	25.14%	55.01%
`EncDecReg`	41.20%	55.52%	95.49%
`EncDecAtt`	**92.00%**	**97.16%**	**98.75%**

Table 9.1: Test accuracy for each model on each test set, with the best results for each test set in bold

The `EncDecAtt` model had the best results on all three datasets. The lowest accuracy this model had was 92% on the smallest dataset. Across all three datasets, the difference between the best and worst results for the `EncDecAtt` model was less than 7%. These results are in contrast to the `EncDecReg` model, which had low accuracy results for both the small and medium datasets, but high accuracy on the big dataset. The `EncDecReg` model had an accuracy of almost 95.5% on the big dataset, which is less than 4% worse than the results for the `EncDecAtt` model. The difference in the accuracy between the `EncDecReg` and `EncDecAtt` model on the medium dataset was more than 40% and almost 50% on the smallest dataset. The `VecRep` model had consistently

lower accuracy than the two other but almost doubled its accuracy from the medium to the big dataset.

9.1.1 VecRep

Accuracy and Loss

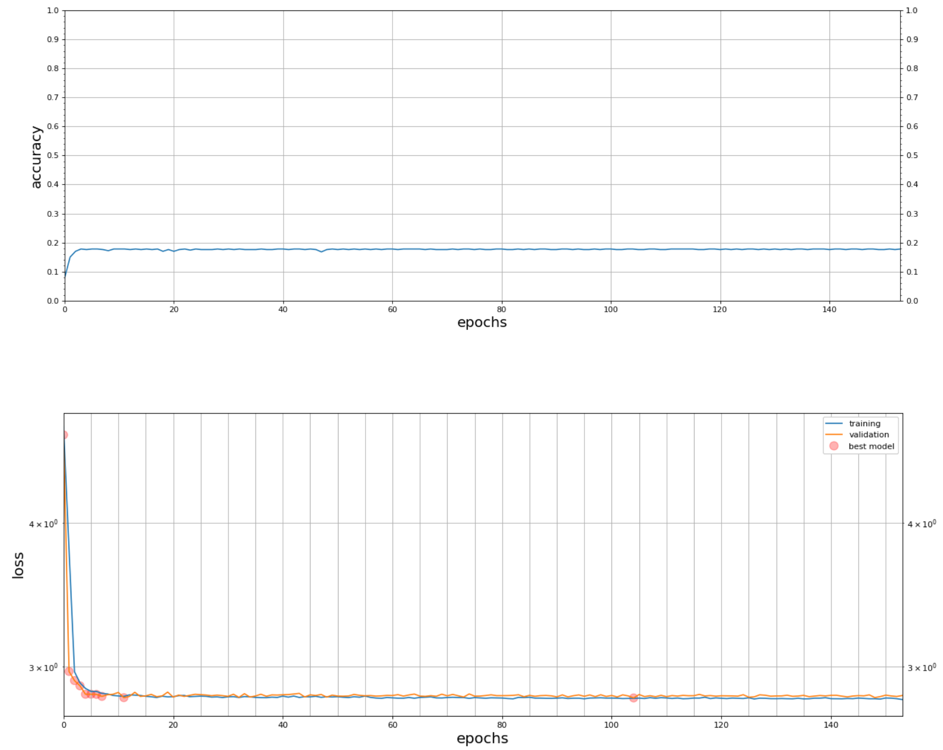

Figure 9.1: Accuracy and loss for `VecRep` on small dataset

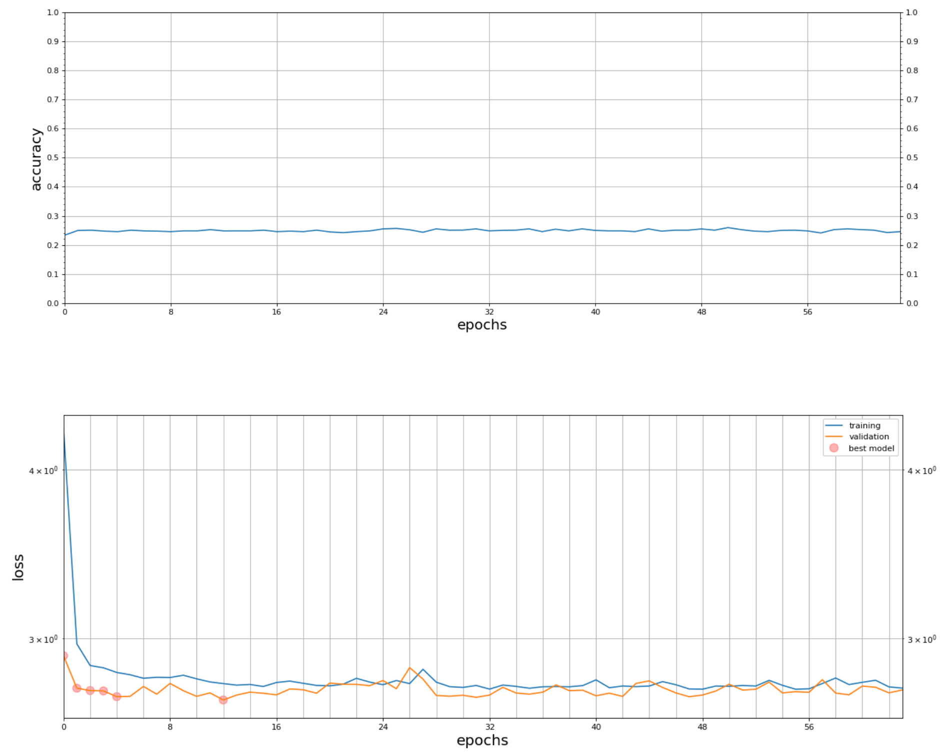

Figure 9.2: Accuracy and loss for `VecRep` on medium dataset

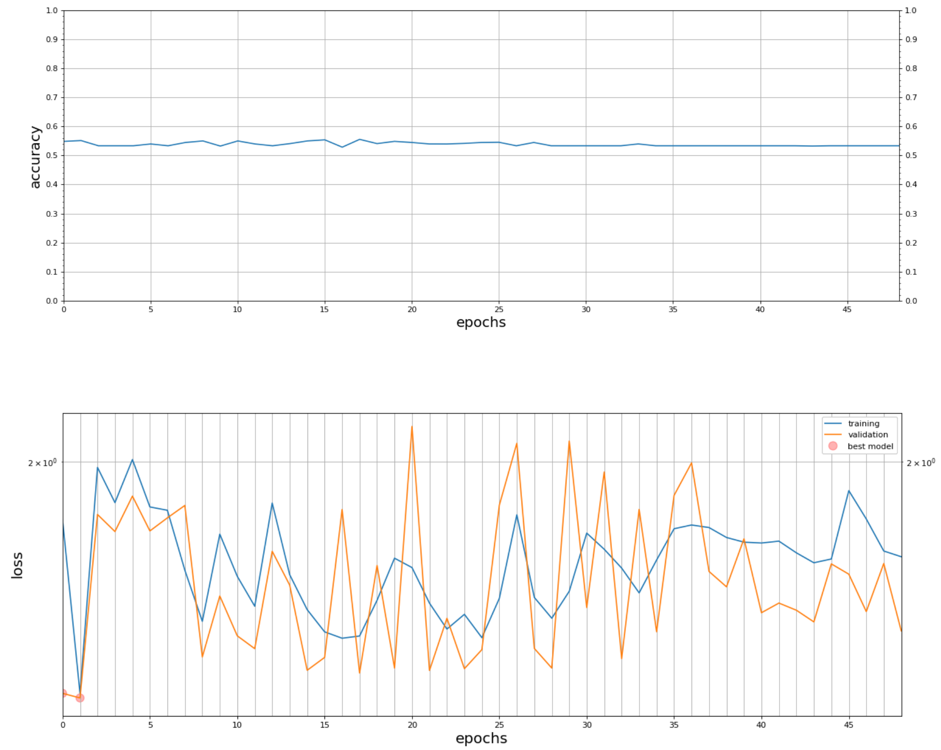

Figure 9.3: Accuracy and loss for `VecRep` on big dataset

The minimal increase in accuracy for all three `VecRep` models across the epochs indicates that the model was unable to learn. The loss values for the tests on the small and medium datasets seem to decrease during the first couple of epochs, while the loss value for the test on the big dataset appears highly erratic.

Confusion Matrix

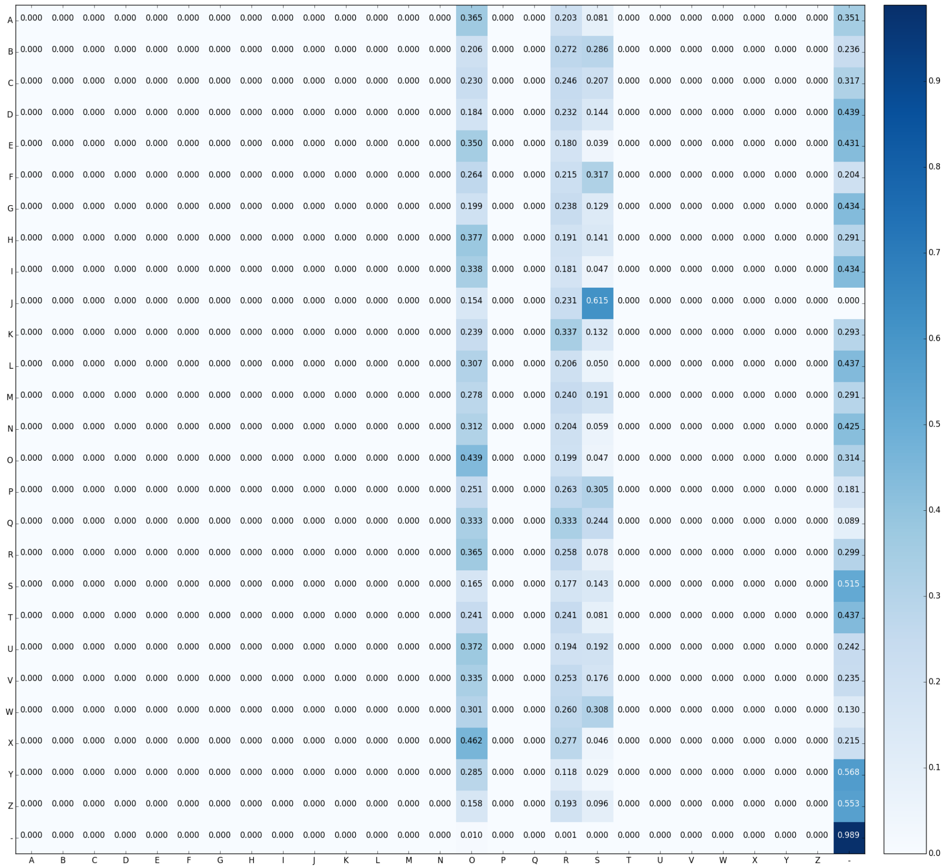

Figure 9.4: Confusion matrix for the best `VecRep` model on the big dataset

The confusion matrix illustrated in Figure 9.4 explains how the `VecRep` model was able to achieve an accuracy of over 50% on the big dataset. The big dataset had a maximum word length of 20 letters, while most words in the English language, as well as in our datasets, are shorter than this. Shorter words were padded to compensate for the different word-lengths. This padding was done by filling the empty labels with a special padding symbol. The `VecRep` model seem to base its predictions on the most common labels, which would be the padding symbol. It also seems to prefer other common labels such as O, R, and S. Every other cell in the confusion matrix is zero, meaning the model did not predict on more than four of the total 27 classes. The confusion matrix clearly indicates that the `VecRep` model was not able to translate between our two constructed languages.

9.1.2 EncDecReg

Accuracy and Loss

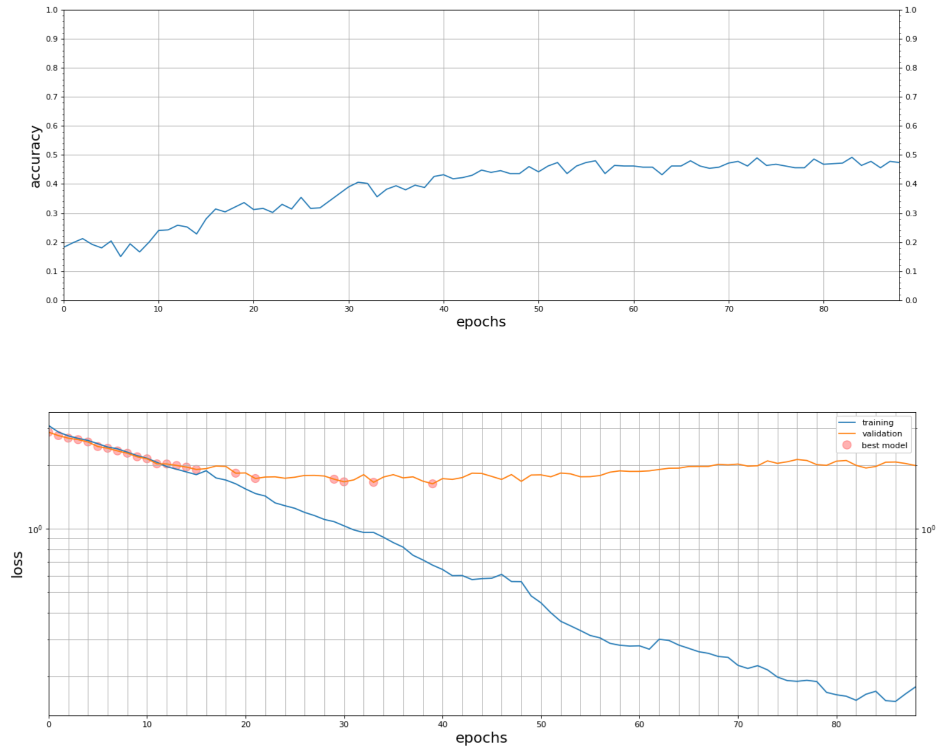

Figure 9.5: Accuracy and loss for `EncDecReg` on small dataset

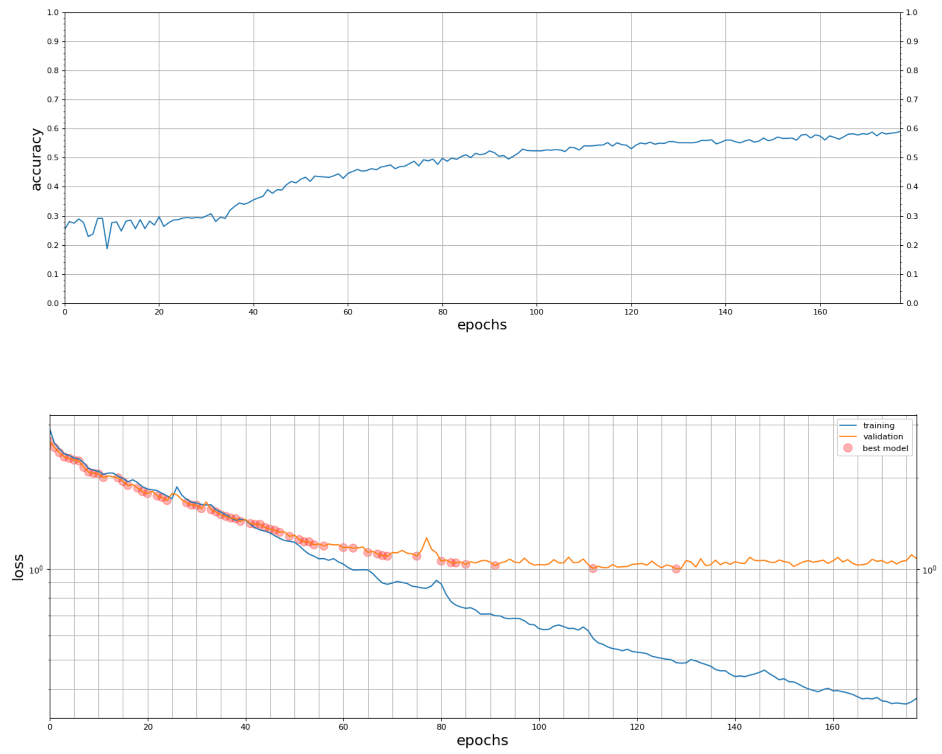

Figure 9.6: Accuracy and loss for EncDecReg on medium dataset

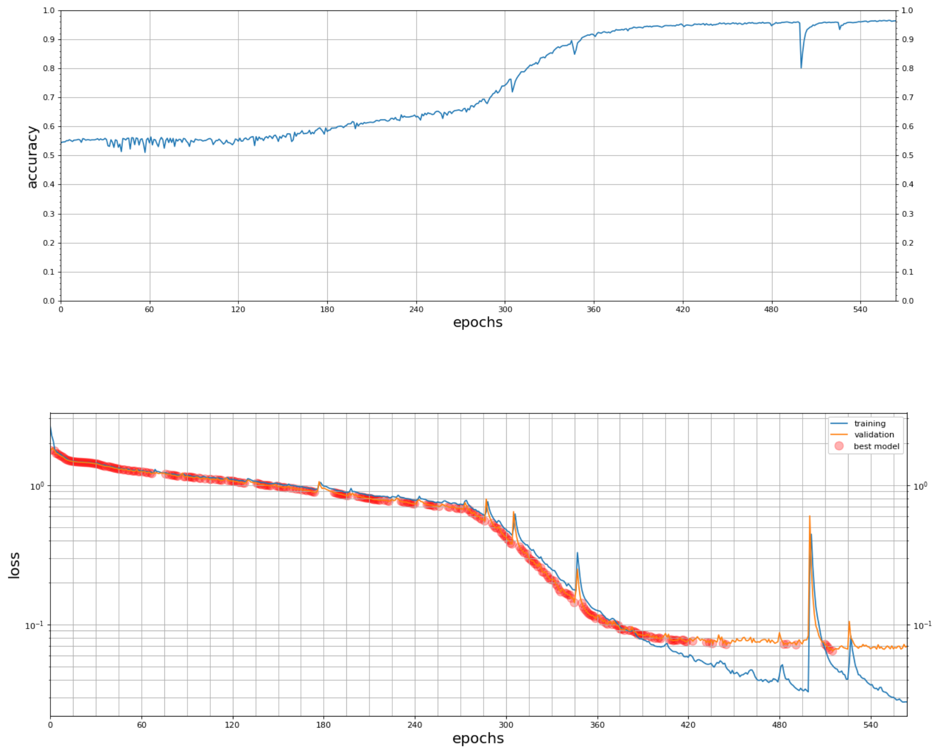

Figure 9.7: Accuracy and loss for EncDecReg on big dataset

The plots for the three best EncDecReg models indicates that the models were able to learn. The accuracy for all three of them indicates improvement, although the accuracy for the small and medium datasets never improves beyond 60% accuracy. The loss plots also indicate overfitting, especially on the smallest dataset. The model ran on the biggest dataset seem to improve almost continuously for 400 epochs. This model also had a big spike in the loss values at around epoch 500 but was able to recover after this. These spikes seem to appear with the Adam optimizer when a model is stuck on a plateau, and the average of past squared gradients becomes small, although we are not entirely sure this is the cause.

Confusion Matrix

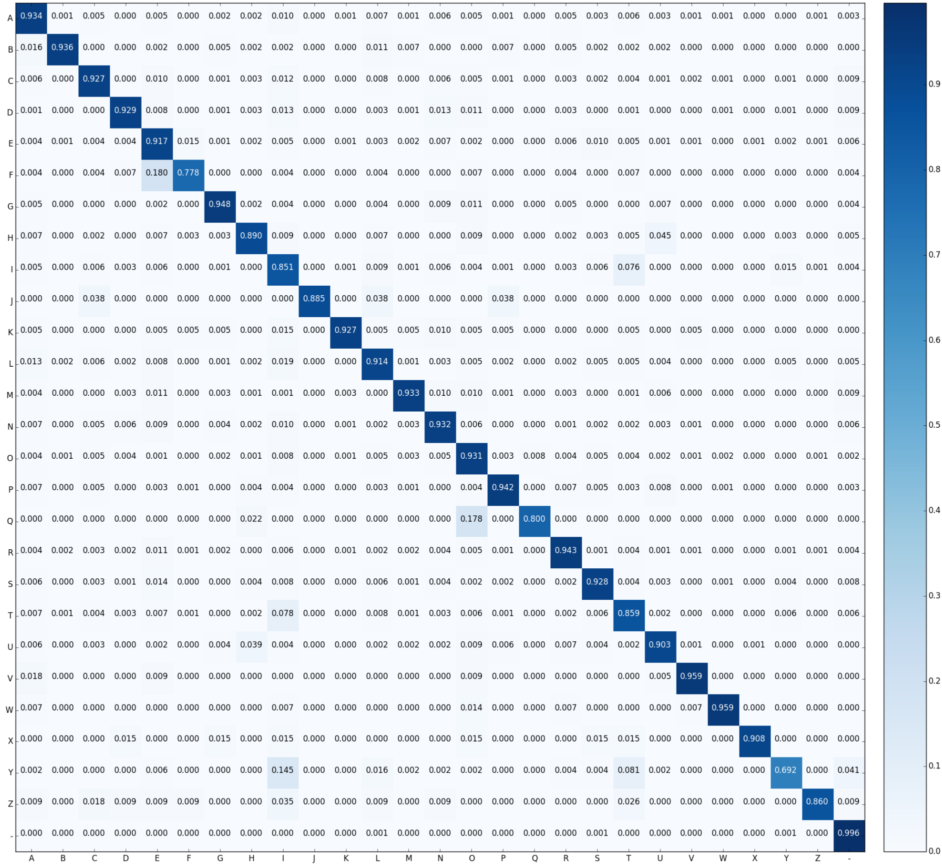

Figure 9.8: Confusion matrix for the best EncDecReg model on the big dataset

The EncDecReg had an accuracy of almost 95.5% on the big dataset, which is reflected in the confusion matrix in Figure 9.8. The confusion matrix has a defined diagonal line with a very high individual accuracy. Most of the labels have an accuracy of over 0.9 with a few exceptions. Y is wrongly labeled as an I in 14.5% of the instances, and as a T in 8.5% of the instances. The most wrongly labeled classes are F as an E (18%) and Q as O (17.8%).

9.1.3 EncDecAtt

Accuracy and Loss

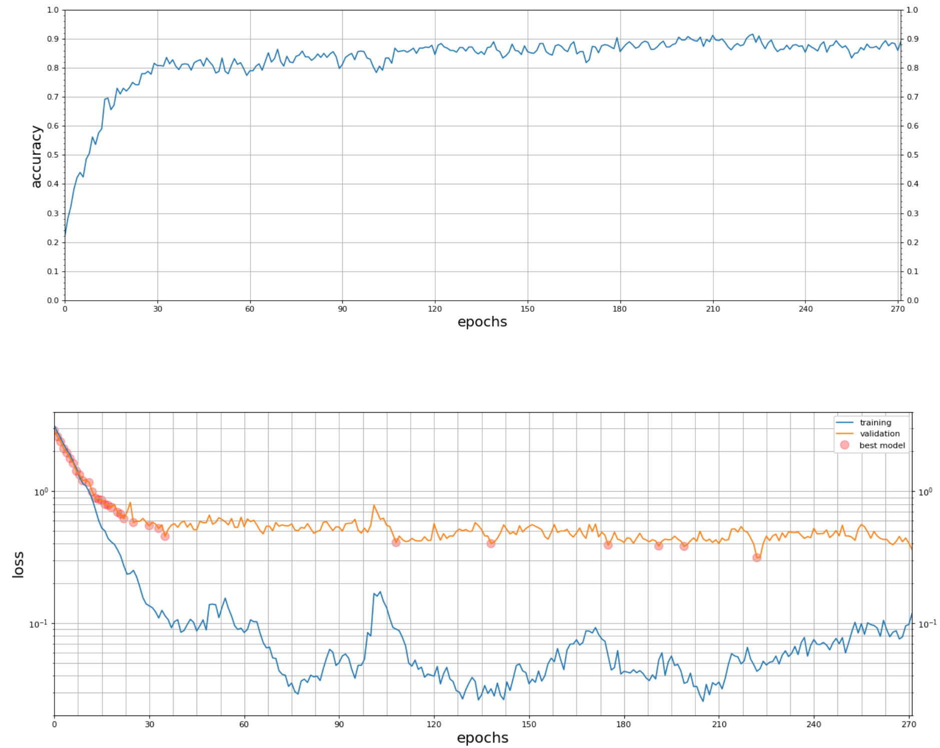

Figure 9.9: Accuracy and loss for `EncDecAtt` on small dataset

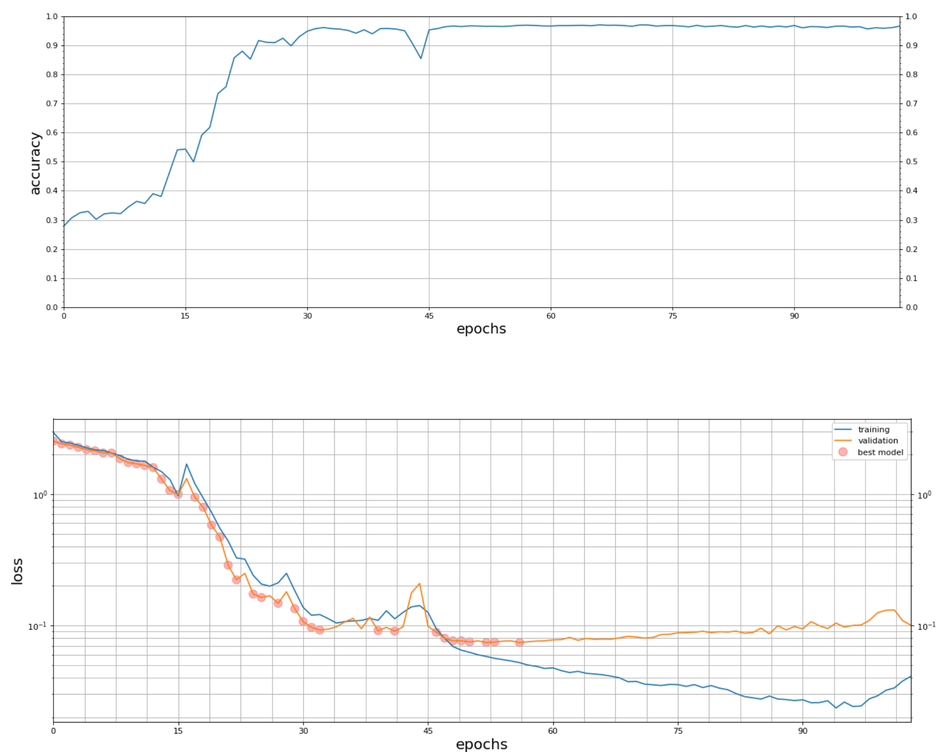

Figure 9.10: Accuracy and loss for `EncDecAtt` on medium dataset

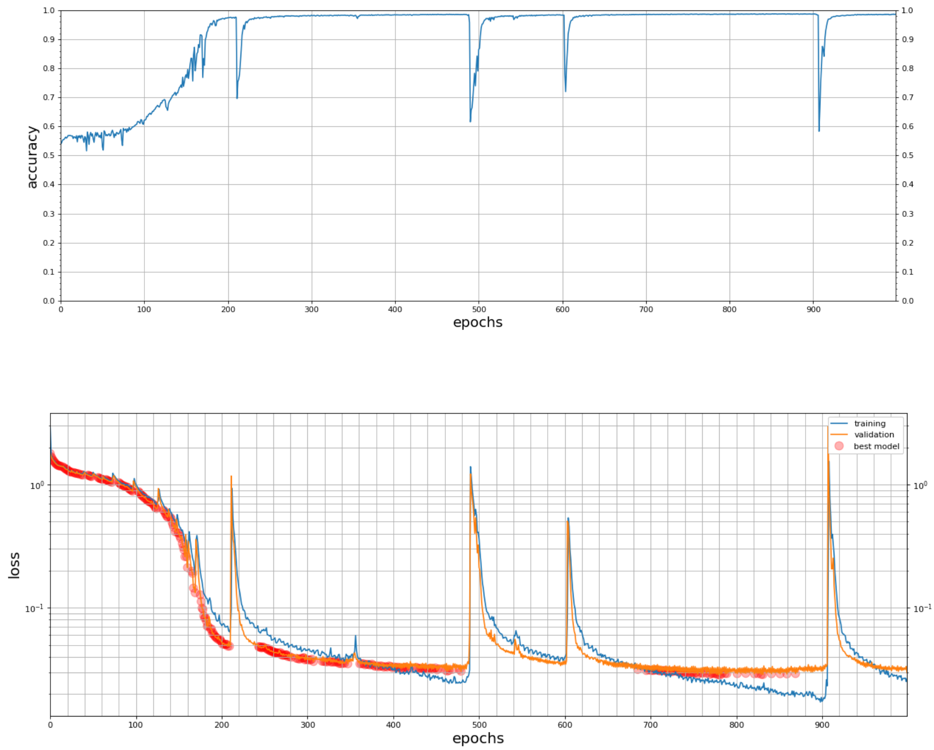

Figure 9.11: Accuracy and loss for `EncDecAtt` on big dataset

These plots show much of the same as the plots for the `EncDecReg` model. The overfitting is less apparent than with the `EncDecReg` model, although the `EncDecAtt` model also seems to overfit on both the small and medium datasets. The model for the big dataset also has multiple spikes, similar to the `EncDecReg` model. This model also seems to improve for many epochs on the biggest dataset.

Confusion Matrix

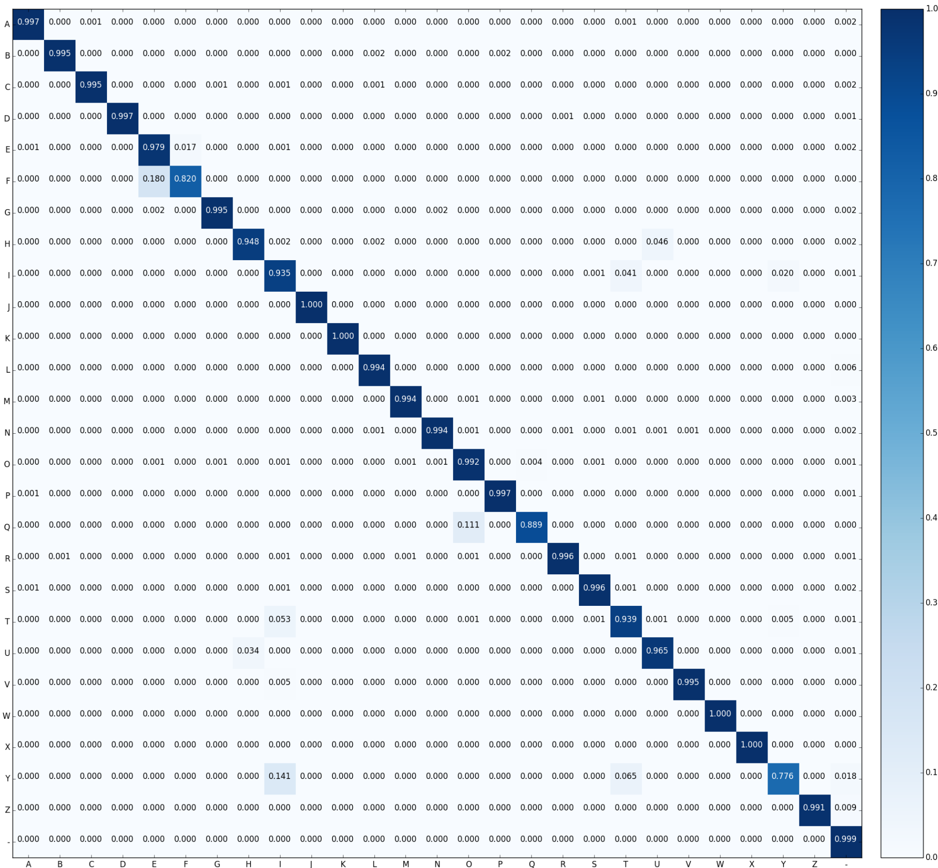

Figure 9.12: Confusion matrix for the best `EncDecAtt` model on the big dataset

The confusion matrix for the `EncDecAtt` model (Figure 9.12) has high individual accuracy, similar to the results of the `EncDegReg` model, although this model has improved even further. The `EncDegAtt` model successfully classifies four classes without a single error, and only three classes have a lower accuracy than 0.9. However, also this model seem to struggle with some of the same misclassifications as the `EncDegReg` model, namely F as E, Q as O, and Y as I or T.

9.2 Handling of Two Fonts

Table 9.2 contains the results for each model on the dataset with two fonts. As seen in this table, the accuracy of the `EncDecAtt` is more or less unaffected by the introduction of a second font, whereas the `EncDecReg` and `VecRep` models have reduced accuracy compared to previous experiments.

Model	Accuracy
`VecRep`	40.49%
`EncDecReg`	88.21%
`EncDecAtt`	98.93%

Table 9.2: Accuracy for each model on a dataset with two fonts

9.2.1 Accuracy and Loss for Each Model

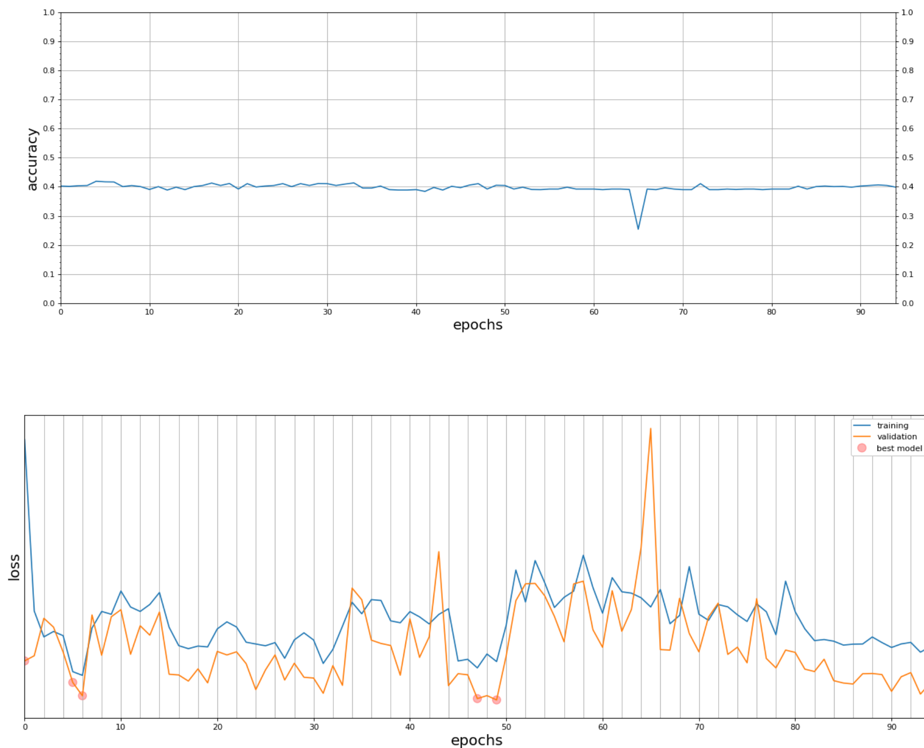

Figure 9.13: Accuracy and loss for `VecRep` handling two fonts

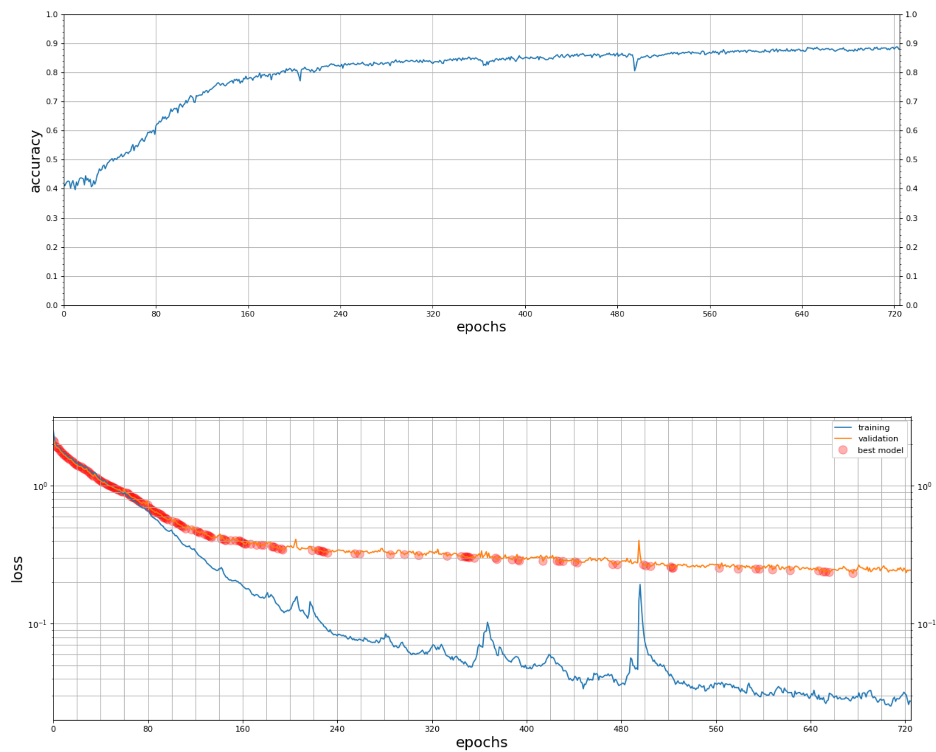

Figure 9.14: Accuracy and loss for EncDecReg handling two fonts

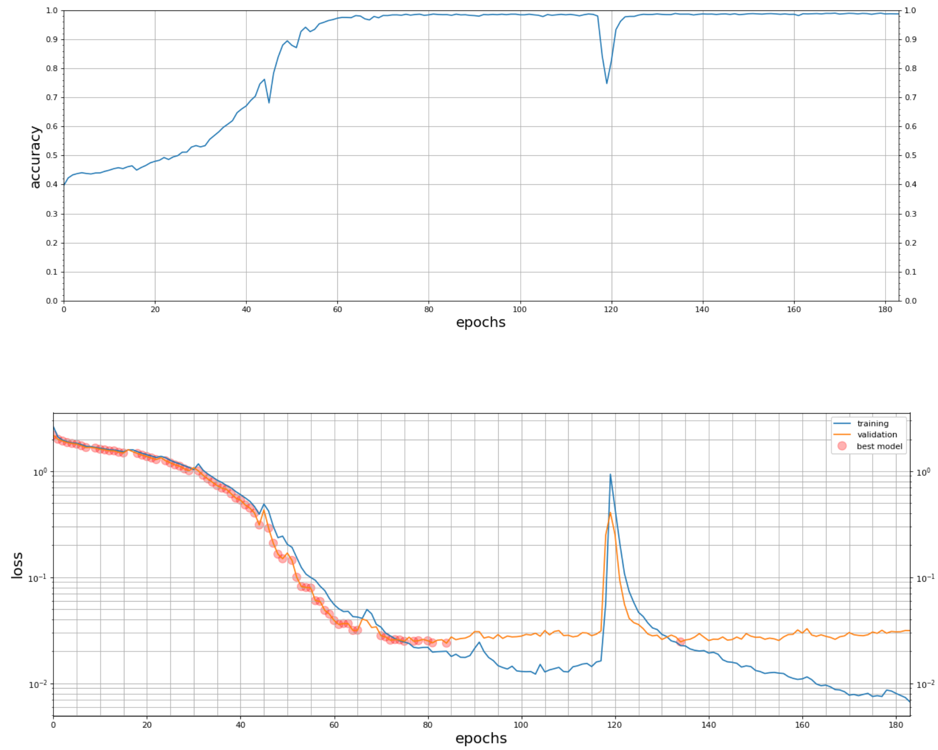

Figure 9.15: Accuracy and loss for `EncDecAtt` handling two fonts

Again the loss plot for the `VecRep` indicates that this model is unable to learn properly, and the loss value seems to worsen after a few couple of epochs. Both the encoder-decoder models seem to overfit, although the `EncDecReg` model overfits significantly after about 130 epochs.

9.3 Noise Handling

Table 9.3 contains the accuracy of the `EncDegAtt` model as the amount of noise was increased. The "Noise alterations," e.i. the actual amount of bits altered from a correct 1 to an incorrect 0, or vice versa, is also listed.

Noise factor	Noise alterations	Accuracy
0%	0%	98.06%
2%	1.49%	95.83%
5%	2.98%	93.35%
6%	3.46%	91.40%
8%	4.45%	90.93%
9%	4.96%	72.25%
10%	5.46%	69.36%
15%	7.94%	66.77%
20%	10.41%	59.51%
40%	20.32%	47.67%
50%	25.28%	46.20%
60%	30.18%	45.29%

Table 9.3: Accuracy for the `EncDecAtt` model

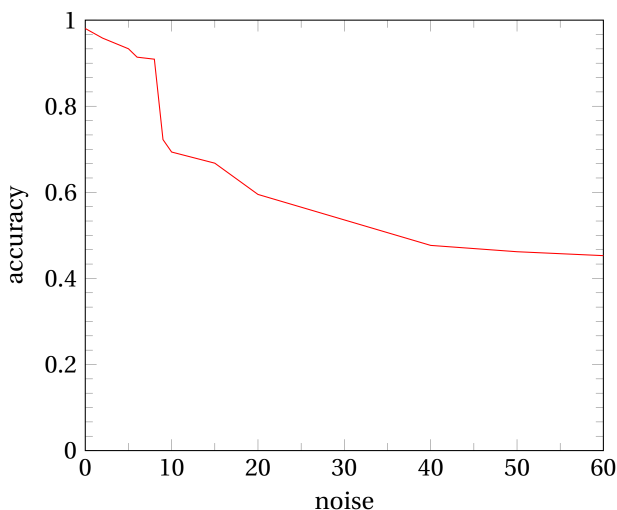

Figure 9.16: Change of accuracy as the amount of noise is increased

The accuracy is also plotted in Figure 9.16, which illustrates how the accuracy deteriorates as the amount of noise was increased. This graph illustrates how the deterioration of the accuracy first falls fast, then flattens out once the amount of noise gets more and more dominant. As shown in both the table and the graph, the accuracy decreased

by less than 1.5% when the noise was increased from 40% to 50%. Similarly, the accuracy decreased with less than a percent when the noise rose to 60%. This is in contrast to how the accuracy decreased by almost 5% when the noise was introduced to a perfect dataset, compared to a dataset with 5% noise. Further, the accuracy decreased by 24% when the noise was increased from 5% to 10%.

This shows the difference in accuracy when learning and predicting on datasets that are perfect, near-perfect, and datasets with significant amounts of noise. It also shows that increasing noise on datasets that already have much noise in them have a smaller effect on the accuracy.

9.4 Stress Test

This experiment was carried out on the models `EncDecReg` and `EncDecAtt`, as the `VecRep` model already had poor results on the experiments that were significantly simpler. The results are presented in Table 9.4.

Model	**Accuracy**
`EncDecReg`	55.02%
`EncDecAtt`	88.44%

Table 9.4: Accuracy for each model on the stress test

9.4.1 EncDecReg Results

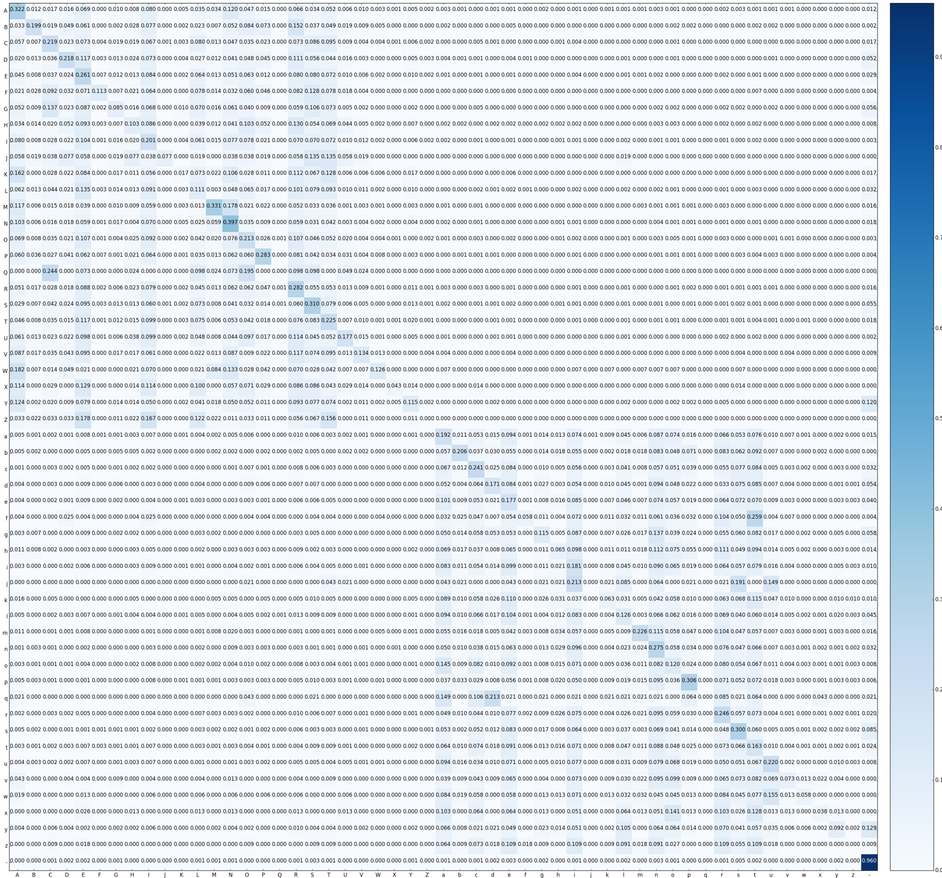

Figure 9.17: Confusion matrix for the best `EncDecReg` model on the stress test

Shown in Figure 9.17 is the confusion matrix for the best `EncDecReg` model on the stress test. As depicted in this matrix, the model had a hard time classifying the labels correctly, which corresponds to its accuracy of 55%. However, the confusion matrix shows traces of a faint diagonal line going from corner-to-corner, indicating correct classifications. In general, the model seems capable of separating the upper-case and lower-case letters from each other, having almost no incorrect classifications shared between the two halves of the matrix. For both groups of upper-case and lower-case letters, the model seems to favor a few letters in each, wrongly classifying them across a wide spectrum of other letters.

9.4.2 EncDecAtt Results

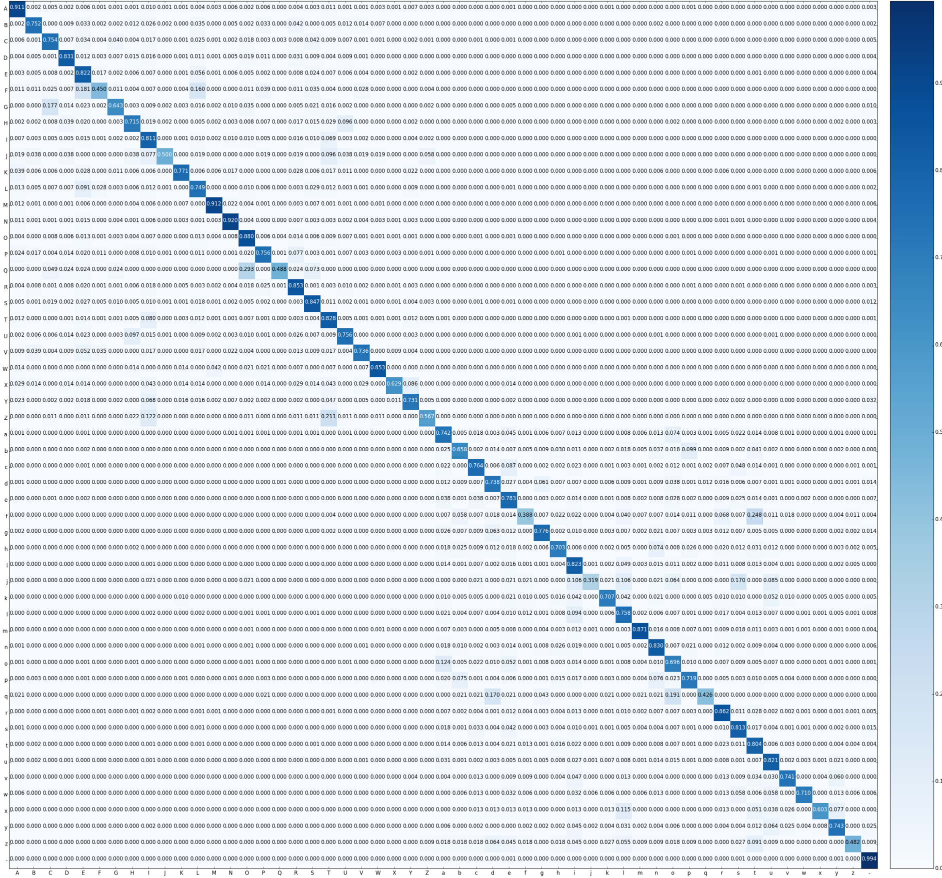

Figure 9.18: Confusion matrix for the best `EncDecAtt` model on the stress test

Figure 9.18 shows the classification matrix of the best `EncDecAtt` model on the stress test. This matrix depicts a clear diagonal line from corner-to-corner, indicating that the model has been able to, for the most part, classifying labels correctly. Although there are certain labels that are misclassified, the vast majority of the labels are correctly classified, which corresponds to the accuracy of over 88% for this model. The confusion matrix also shows almost no overlap between the labels in the upper-case and lower-case classes, indicating that also this model was able to separate these groups from each other. Some of the labels are repeatably misclassified, and many of these are the same misclassifications we have seen in other tests for the same model.

9.5 Result Discussion

In this section, we discuss and compare the various results and models against each other. We also answer the research questions we defined earlier and discuss our results in the context of our research goal.

9.5.1 General Discussion

The first experiment indicated that the `VecRep` model was unable to learn the problem to a satisfactory degree. The confusion matrix indicates that the model is unable to translate anything, and relies on classifying the padded values. `EncDecReg` and `EncDecAtt` both show indications that they learned to translate between our two languages, especially when the size of the datasets was increased. The `EncDecAtt` model outperformed the `EncDecReg` model in every single experiment. The latter model also reached impressive results for the most complicated experiments we carried out. While the only difference between these two encoder-decoder based models is the attention mechanism, there is a big difference in their performance. We see this both when the datasets are small, and in more complex experiments.

9.5.2 Research Questions

Ambiguity in Character Signature Sequences

In our first research question, **RQ1**, we asked if the models were able to deal with ambiguity in the input data. In our first experiment, we used the same system configurations as shown in Table 3.1, which we used to illustrate how some of the letter signatures were either identical or subsequences of another signature. In this table, we showed that the letters C, I, J, L, T, and Y all shared the same unique signature of three black pixels. The other letters that also shared identical signatures were E and F, H and U, O and Q, and S and X.

Comparing these sets to the confusion matrices, we can see that some of these characters were indeed those the models struggled to classify correctly. Specifically, U, I and T, F and E, as well as Q and O were problematic for both the encoder-decoder models, although their accuracy was no lower than 0.692 for the `EncDegReg` model, and 0.776 for the `EncDegAtt` model on any of these labels. The most commonly wrongly classified label was an E as F, with both the models having an equal confusion of 0.18.

These numbers indicate that there is still room for improvement, and that ambiguity may be a concern. Nevertheless, we have concluded that in the context of our experiments and results, the models were able to handle ambiguity to a degree we found satisfactory.

Handling of Multiple Fonts

Experiments have indicated that both the encoder-decoder models were able to handle more than one font, as questioned in **RQ2**. The model without the attention mechanism had visibly lower accuracy than experiments carried out on datasets consisting of one font. The `EncDecAtt` model was more or less unaffected by the introduction of an additional font in the second experiment. The same model also yielded good results in the stress test, where the input consisted of five different fonts. On the same test, the `EncDecReg` model had lower accuracy, indicating that this model may be unsuitable for input with such high variance in the sequences.

Handling of Noise

Lastly, **RQ3** asked if the model(s) were able to adapt to noise and imperfect input data. Experiments have shown that the `EncDecAtt` model was able to handle increasing amounts of noise, although the accuracy decreased as the noise factor increased. For a noise factor of 5%, the accuracy decreased by a little less than 5%, while a noise factor of 10% decreased the accuracy by more than 28%. As there is no real answer to how much noise a model should handle, or how robust a model should be to noise, there is no way to define a hard threshold between reasonable amounts of noise, and excessive amounts of noise. We have decided to conclude this question by stating that the `EncDecAtt` model was able to handle noise in a satisfactory manner.

9.5.3 Discussing the Research Goal

Our goal in this thesis was to create a model that was able to use signature sequences to recognize letters and words. The results from our experiments, which has been presented in this chapter, indicates that we were successful in reaching this goal. The two encoder-decoder models, especially the `EncDecAtt` model, has been successful in recognizing words with a high accuracy, and we have found these results satisfactory for our evaluation.

9.6 Analysis

In this section, we look closer at how the encoder-decoder models work, and we analyze the models in the context of our results. The goal of this analysis is to investigate the cause and the affect for various parts of the encoder-decoder framework.

9.6.1 Encoding

We first look at the encoding mechanism of the encoder-decoder framework and how it works. We do this by training a model on a dataset with 1% noise and then feeding the same words multiple times with the same amount of noise, but different random seeds. We fed the model three words one thousand times each and plotted the encoded

context vector in a low dimensional space using t-distributed Stochastic Neighbor Embedding (t-SNE). t-SNE is a tool for visualizing high-dimensional data. It converts similarities between data points to joint probabilities and tries to minimize the Kullback-Leibler divergence between the joint probability distribution between low-dimensional and high-dimensional data (Maaten and Hinton, 2008).

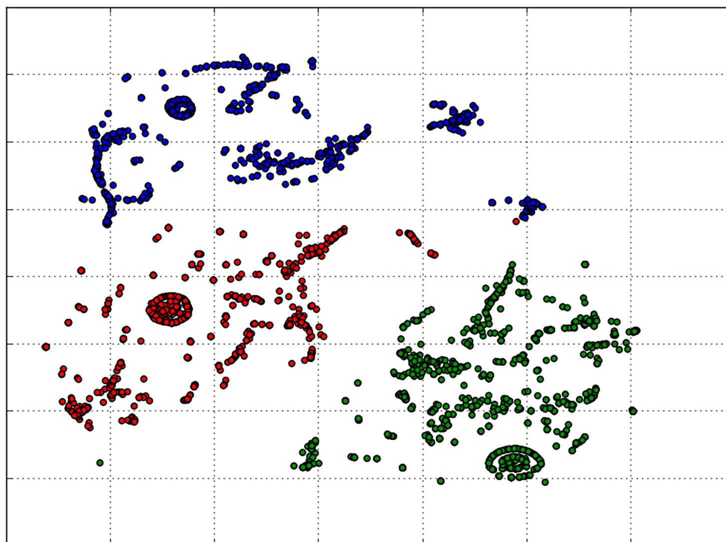

Figure 9.19: Visualization of context vectors for three different words with different noise seeds using t-SNE

The plot can be seen in Figure 9.19, where the three clusters are the words “PYRAMIDS” (red), “MYTHIC” (blue), and “SNOWMAN” (green). We observe that t-SNE places many of the instances of each word in a very focused cluster. However, each word also has instances that are scattered outside their focus areas, indicating a variance in the context vector. We can also see a few examples of extreme outliers for all three words. The low-dimensional visualization indicates that the context vector can retain some of the information in the word itself, despite the applied noise. This is an interesting observation, as it illustrates the robustness of the compression done in the context vector, and how the encoder-decoder can take advantage of this.

9.6.2 Decoding

The decoder and its ability to feed the previous output back as input is one of the corner-stones in the encoder-decoder framework. Allowing the decoder to read the previous output helps the decoder know something about the current state of the “translation” process from the vector passed on from the encoder. If the decoder did not feed the output back as input, it would have to rely solely on the hidden states and cell states for the decoding process.

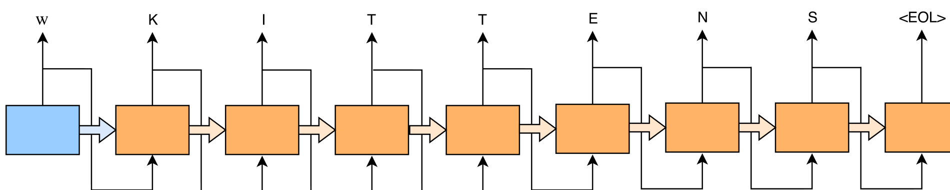

Figure 9.20: Decoding the word “KITTENS”

During decoding, the decoder outputs a vector for each timestep, ran through a softmax function. The framework uses `argmax` to find the element in the vector with the highest value, e.i. the predicted value, and feeds this value back to the decoder for the next timestep. We have altered this functionality, and instead of feeding back the predicted value, we send something else back and see how the decoder reacts. Without any modification to the behavior, the model we used is perfectly able to classify each label in the word and produces the correct output “KITTENS,” as illustrated in Figure 9.20.

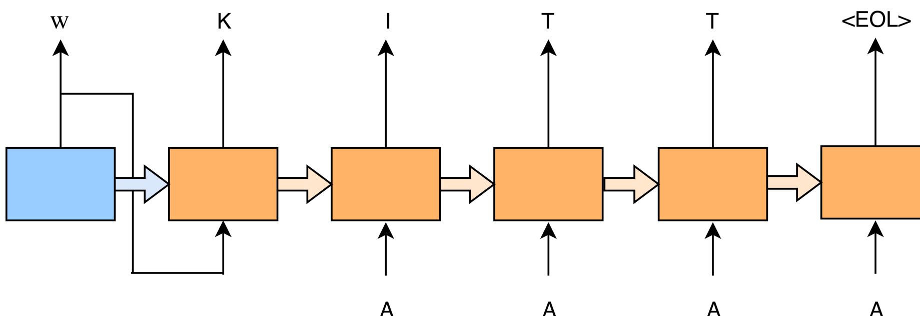

Figure 9.21: Decoding the word “KITTENS” by feeding back wrong label

If we instead change the logic to use the `argmin` function, which returns the element we predicted the “least,” the word becomes “KIIIEES.” Similarly, if we only feed back the first label, which in this case would be the letter A, the word becomes “KITTT,” as illustrated in Figure 9.21. Note that the first letter in the word is predicted based on the context vector from the encoder, so it is likely this is correct. Except for the first letter, we see some of the other letters in the word have changed. However, despite our trickery, the decoder is still able to predict labels that are actually in the word, indicating the some of the “knowledge” in the decoding process is also stored and shared within the recurrent unit. We also observe that the output is not that far off, with both the misled decoders outputting the second letter correctly. Lastly, if we only “fool” the decoder for the first and second letter, telling it we outputted an A, the decoder was able to predict every letter correctly, outputting “KITTENS.” The correct prediction also indicates that the knowledge in the cell states may correctly override the fed back input, instead of creating a “domino” effect as we expected, where the output would progressively get

more and more incorrect.

The size of the context vector correlates to the size of the RNN cell, as the context vector is the RNN cell's last hidden state. In the experiments above, the RNN cells had a size of 128. We changed this size to see if this affected the results when we tricked the decoder. The decoder was fed the letter A as the first and second label, similarly to the experiment already carried out, in which the RNN cell with a depth of 128 was able to classify every label correctly. With an RNN cell size of 64, the decoder was still able to classify correctly. However, with a size of 32, the decoder started to misclassify after the first label.

These observations indicate that the encoded context vector contains partial knowledge about which labels to output, especially the first labels. As the decoding progresses, the decoder seems to shift its attention more towards the previous output and relies less on the information from the context vector. Our experiments also indicates that the size of the context vector correlates to how long the decoder can rely on the information in it.

9.6.3 Use of Attention

The experiments carried out in this thesis have shown that both the models based on the encoder-decoder framework were powerful, but the model that utilized the attention mechanism was significantly better. Figure 9.22 shows the attention heatmap, in other words, the areas in which the attention mechanism focused while classifying each label. In this example, the model correctly classifies the word "IMAGINED." The signature configurations for this experiment was reused from Table 3.1.

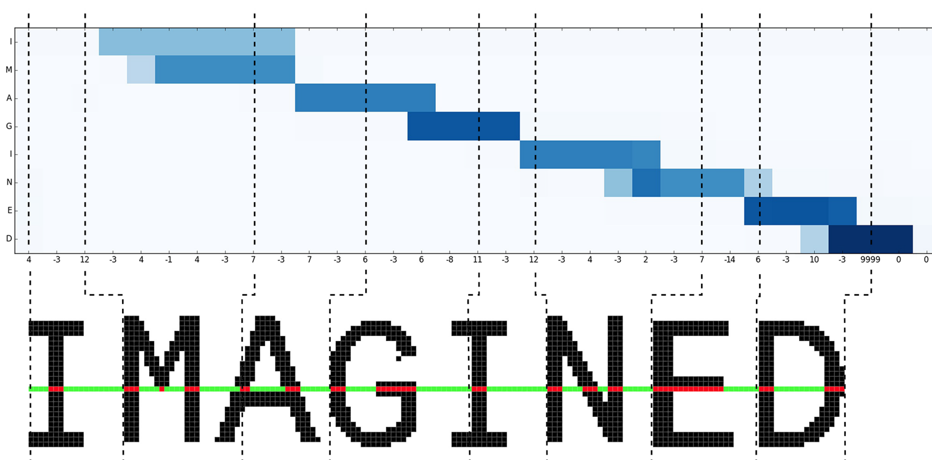

Figure 9.22: Attention heatmap

Some interesting observations can be done with the information from the heatmap and the aligned input text. One interesting observation is that while classifying the very first label, the attention mechanism pays no interest in the values for that label, and only focuses on later values. Table 3.1 indicates that the signature for the letter I is

shared among five other letters, and it may seem like the attention mechanism is smart enough to consider later input to classify the label correctly. We can observe the same behavior when the second I is classified. The signature for the letter E is shared with the letter F, and we observe that the attention mechanism focuses on information beyond its own signature. The attention mechanism most likely does this because the only way to differentiate between and an F and an E is with the sequences after its own signature. The signatures for the letter M and N both begin with the same subsequence $[-3, 4]$. This is reflected in the attention heatmap where we observe that the mechanism pay extensive attention to values after this subsequence while classifying both these letters.

The additional information provided by the attention mechanism gives the `EncDecAtt` model a significant advantage over the `EncDecReg` model, which can only rely on the information in the context vector. We see that this benefit pays off in the consistently better results the `EncDegAtt` model has compared to the `EncDecReg` model. The heatmap also shows how the attention mechanism was able to keep attention fixed on longer subsequences, accounting for the different width ratio between input and output.

Chapter 10

Conclusion and Future Work

Section 10.1 presents the final conclusions drawn from the results presented and discussed in the previous chapter. We evaluate our research contributions in Section 10.2, and in Section 10.3 we present ideas for potential future work and improvements.

10.1 Conclusion

As a result of research, we have developed two models based on the encoder-decoder framework that gave satisfying results on the experiments we carried out. The `EncoDecAtt` model was able to classify and differentiate between 27 classes with an accuracy of 92%, 97%, and almost 99% for experiments carried out on increasingly larger datasets. The model was also able to classify a problem with 53 classes with an accuracy of 88% under challenging conditions.

We have shown that both the encoder-decoder models were able to handle signature sequence ambiguity, although some letters were repeatedly misclassified as other letters. We have also shown that the `EncDecAtt` model was able to handle input with two fonts without nearly any reduced accuracy. The `EncDecAtt` model had an accuracy of nearly 99%, while the `EncDecReg` model had an accuracy of about 88%. Lastly, we have shown that the `EncDecAtt` model was robust to noise. For reasonable low amounts of noise, the accuracy remained high but quickly deteriorated once the amount of noise increased. With 5% noise, the model had an accuracy of over 93%.

The two models built and presented in this thesis are built on state-of-the-art technologies and uses state-of-the-arts approaches in the area of (neural) machine translation. These models illustrate the power of the encoder-decoder framework and prove how this framework was able to handle input and output with unknown alignments with high precision.

We have concluded that we met the research goal of developing a model that uses signature sequences to recognize letters and words based on the experiments we conducted and the results they gave.

10.2 Contributions

The encoder-decoder framework has attracted much attention, and the approach has almost become "de facto standard" in various sequence-to-sequence mapping related problems, despite it only dating back to 2014. The model has, since its recent inception, been used in countless models, and the framework has played a major role in achieving various state-of-the-art results in areas such as machine translation, computer vision, and speech recognition. These achievements may undermine some of the results we have achieved in this thesis to some degree, as the framework has already been proven countless times to handle much more complex problems. The research contribution this thesis does not revolve around proving just how powerful the encoder-decoder framework is. Instead, we have demonstrated how the framework was able to handle a problem with characteristics like ours. We have already stated how our problem is both similar and dissimilar to traditional translation problems in Section 3.5.

The main contribution in therefore the exploration in using the encoder-decoder framework and the attention mechanism. We have shown how that the framework handles input and output that have different widths. Some of the datasets used in our experiments had a ratio between input and output of nearly 7 : 1, which is more than most traditional translation tasks. We have also illustrated how the attention mechanism aligns with the same ratio between input and output. We have additionally shown how the attention mechanism acts differently depending on the ambiguity of the input data.

Our data have a strict ordering from start to end, with no alteration in the ordering of neither the input values nor the output values. We have shown that the encoder-decoder models have successfully learned not to swap or reorder output values, something that is common when translating between two spoken languages. Furthermore, we have also illustrated how the encoder-decoder framework encodes the input information, and how noise affects this process. Lastly, we have explored how the mechanism of feeding back previous output affects the decoder module. We have experimented with this mechanism and investigated how the decoder shifts its attention from the context vector to the input as the decoding progresses.

10.3 Future Work

The results archived by the `EncDecAtt` model is, under certain conditions, nearing perfect. It would be interesting to see if the model was able to improve even further by using more than one signature sequence. Using multiple signatures could be done by capturing two sequences at different heights, for example as illustrated in Figure 3.1. Doing this could also have the potential to eliminate ambiguity. Without ambiguity, the model could be able to recognize the input perfectly. Flawless recognition opens up possibilities to use the approach as a possible lossless compression algorithm.

We have focused our experiments in this thesis on recognizing single words. It would be interesting to see if a similar approach could be applied to multiple words or even full sentences. Recognizing either multiple words or whole sentences would require the model to handle much longer input sequences, as well as sequences that are

broken up into separate groups of words.

There are also possibilities left untouched related to both the encoder-decoder framework, and the attention mechanism. Numerous variants and improvements have been proposed for both of them, and the two encoder-decoder models implemented in this thesis barely scratches the surface for what is possible.

Appendices

Appendix A

Image and Figure References

- Image 1.1 is from https://commons.wikimedia.org/wiki/File:Internet_Archive_book_scanner_1.jpg and distributed under a CC-BY 2.0 license.
- Images 1.2 and 1.3 were used with the consent of Markus Persson.
- Image 1.4 was used with the consent of user tmcaffeine on https://www.reddit.com.
- Figure 2.1 was recreated from Oates (2005).
- Inspiration for Figure 4.1 was taken from http://www.theprojectspot.com/tutorial-post/i/7 and http://homepages.gold.ac.uk/nikolaev/311perc.htm.
- Figure 4.2 was inspired by http://www.wildml.com/2015/09/recurrent-neural-networks/ and the illustrations in Goodfellow et al. (2016).
- Figure 4.3 and 4.4 was inspired by the blog post http://colah.github.io/posts/2015-08-Understanding-LSTMs/
- Figure 4.5 was recreated from the original illustration in Cho et al. (2014b).
- Image 4.6 is a print screen from https://google.github.io/seq2seq/.
- Image 5.1 is a print screen from Android's video "Android: 100 Billion Words." https://www.youtube.com/watch?v=wIK0JKTQcI8.
- Figure 5.2 was recreated from Koehn (2010).
- Inspiration for Figure 5.3 was taken from https://github.com/farizrahman4u/seq2seq and http://mogren.one/talks/2016/09/29/nmt.html.

Appendix B

Example Words

ABUSH zamarra

Figure B.1: Two words written in Arial Mono. Sizes 102x41 and 143x41

KIRMESSES paleoceanography

Figure B.2: Two words written in Times New Roman. Sizes 198x41 and 269x41

ATALANTA circumrotated

Figure B.3: Two words written in Courier. Sizes 168x41 and 271x41

RESUBSCRIBER bishop

Figure B.4: Two words written in Georgia. Sizes 276x41 and 96x41

LOVE captains

Figure B.5: Two words written in Verdana. Sizes 88x41 and 142x41

Bibliography

Arik, S. O., Chrzanowski, M., Coates, A., Diamos, G., Gibiansky, A., Kang, Y., Li, X., Miller, J., Raiman, J., Sengupta, S., et al. (2017). Deep voice: Real-time neural text-to-speech. *arXiv preprint arXiv:1702.07825*.

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bengio, S., Vinyals, O., Jaitly, N., and Shazeer, N. (2015). Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 1171–1179.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166.

Bhardwaj, N. and Agarwal, S. (2014). An imaging technique for retrieval of lost content in damaged documents. *International Journal of Computer Applications*, 104(5).

Britz, D., Goldie, A., Luong, T., and Le, Q. (2017). Massive exploration of neural machine translation architectures. *arXiv preprint arXiv:1703.03906*.

Brown, P. F., Cocke, J., Pietra, S. A. D., Pietra, V. J. D., Jelinek, F., Lafferty, J. D., Mercer, R. L., and Roossin, P. S. (1990). A statistical approach to machine translation. *Computational linguistics*, 16(2):79–85.

Cho, K., Van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014a). On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014b). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Chung, J., Cho, K., and Bengio, Y. (2016). A character-level decoder without explicit segmentation for neural machine translation. *arXiv preprint arXiv:1603.06147*.

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555.*

Council, N. R., Committee, A. L. P. A., et al. (1966). *Language and Machines: Computers in Translation and Linguistics; A Report.* National Academy of Sciences, National Research Council.

Coyle, K. (2006). Mass digitization of books. *The Journal of Academic Librarianship*, 32(6):641–645.

Felici, J. W. (2012). *The complete manual of typography: a guide to setting perfect type.* Peachpit Press, 2nd edition.

Freeman, C. and Louçã, F. (2001). *As time goes by: from the industrial revolutions to the information revolution.* Oxford University Press.

Gers, F. A. and Schmidhuber, E. (2001). Lstm recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340.

Gers, F. A., Schmidhuber, J., and Cummins, F. (2000). Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471.

Gers, F. A., Schraudolph, N. N., and Schmidhuber, J. (2002). Learning precise timing with lstm recurrent networks. *Journal of machine learning research*, 3(Aug):115–143.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning.* MIT Press. `http://www.deeplearningbook.org`.

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., and Schmidhuber, J. (2016). Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems.*

Hinton, G. E. (1986). Learning distributed representations of concepts. In *Proceedings of the eighth annual conference of the cognitive science society*, volume 1, page 12. Amherst, MA.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Hsu, W.-N., Zhang, Y., and Glass, J. (2016). Recurrent neural network encoder with attention for community question answering. *arXiv preprint arXiv:1603.07044.*

Hutchins, J. (1997). From first conception to first demonstration: the nascent years of machine translation, 1947–1954. a chronology. *Machine Translation*, 12(3):195–252.

Hutchins, J. (2007). Machine translation: A concise history. *Computer aided translation: Theory and practice.*

Jozefowicz, R., Zaremba, W., and Sutskever, I. (2015a). An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 2342–2350.

Jozefowicz, R., Zaremba, W., and Sutskever, I. (2015b). An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 2342–2350.

Kalchbrenner, N. and Blunsom, P. (2013). Recurrent continuous translation models. In *EMNLP*, volume 3, page 413.

Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Koehn, P. (2010). *Statistical machine translation*. Cambridge University Press.

Kurzweil, R. C., Bhathena, F., and Baum, S. R. (2000). Reading machine system for the blind having a dictionary. US Patent 6,033,224.

Lee, K.-H., Slattery, O., Lu, R., Tang, X., and McCrary, V. (2002). The state of the art and practice in digital preservation. *Journal of research of the National institute of standards and technology*, 107(1):93.

Luong, M.-T., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

Maaten, L. v. d. and Hinton, G. (2008). Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605.

Mike Schuster, M. J. and Thorat, N. (2016). Zero-shot translation with google's multilingual neural machine translation system. `https://research.googleblog.com/2016/11/zero-shot-translation-with-googles.html`. [Accessed: 2017-05-12].

Mori, S., Nishida, H., and Yamada, H. (1999). *Optical character recognition*. John Wiley & Sons, Inc.

Mori, S., Suen, C. Y., and Yamamoto, K. (1992). Historical review of ocr research and development. *Proceedings of the IEEE*, 80(7):1029–1058.

Morris, R. J. and Truskowski, B. J. (2003). The evolution of storage systems. *IBM systems Journal*, 42(2):205–217.

Nasjonalbiblioteket (2016). Collaboration projects. `http://www.nb.no/English/The-Digital-Library/Collaboration-Projects`. [Accessed: 2016-10-25].

Oates, B. J. (2005). *Researching information systems and computing*. Sage.

Official Minecraft Wiki (2017). 1.7.2 - official minecraft wiki. `http://minecraft.gamepedia.com/1.7.2`. [Accessed: 2017-04-20].

Olah, C. (2015). Understanding lstm networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/. [Accessed: 2017-03-30.

Oxford English Dictionary (2010). digitization. http://www.oed.com/view/Entry/240886. [Accessed: 2016-10-24].

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1988). Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1.

Sankaran, B., Mi, H., Al-Onaizan, Y., and Ittycheriah, A. (2016). Temporal attention model for neural machine translation. *arXiv preprint arXiv:1608.02927*.

Sennrich, R., Haddow, B., and Birch, A. (2015). Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958.

Stuart Russell, P. N. (2010). *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Prentice Hall, 3rd edition.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Turovsky, B. (2016a). Found in translation: More accurate, fluent sentences in google translate. https://blog.google/products/translate/found-translation-more-accurate-fluent-sentences-google-translate/. [Accessed: 2017-03-28].

Turovsky, B. (2016b). Ten years of google translate. https://blog.google/products/translate/ten-years-of-google-translate/. [Accessed: 2017-03-28].

Vinyals, O., Kaiser, Ł., Koo, T., Petrov, S., Sutskever, I., and Hinton, G. (2015). Grammar as a foreign language. In *Advances in Neural Information Processing Systems*, pages 2773–2781.

Watanabe, T., Suzuki, J., Tsukada, H., and Isozaki, H. (2007). Online large-margin training for statistical machine translation. In *In Proc. of EMNLP*. Citeseer.

Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

Wołk, K. and Marasek, K. (2015). Neural-based machine translation for medical text domain. based on european medicines agency leaflet texts. *Procedia Computer Science*, 64:2–9.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

Xingjian, S., Chen, Z., Wang, H., Yeung, D.-Y., Wong, W.-K., and Woo, W.-c. (2015). Convolutional lstm network: A machine learning approach for precipitation nowcasting. In *Advances in Neural Information Processing Systems*, pages 802–810.

Ye, Q. and Doermann, D. (2015). Text detection and recognition in imagery: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 37(7):1480–1500.

Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.