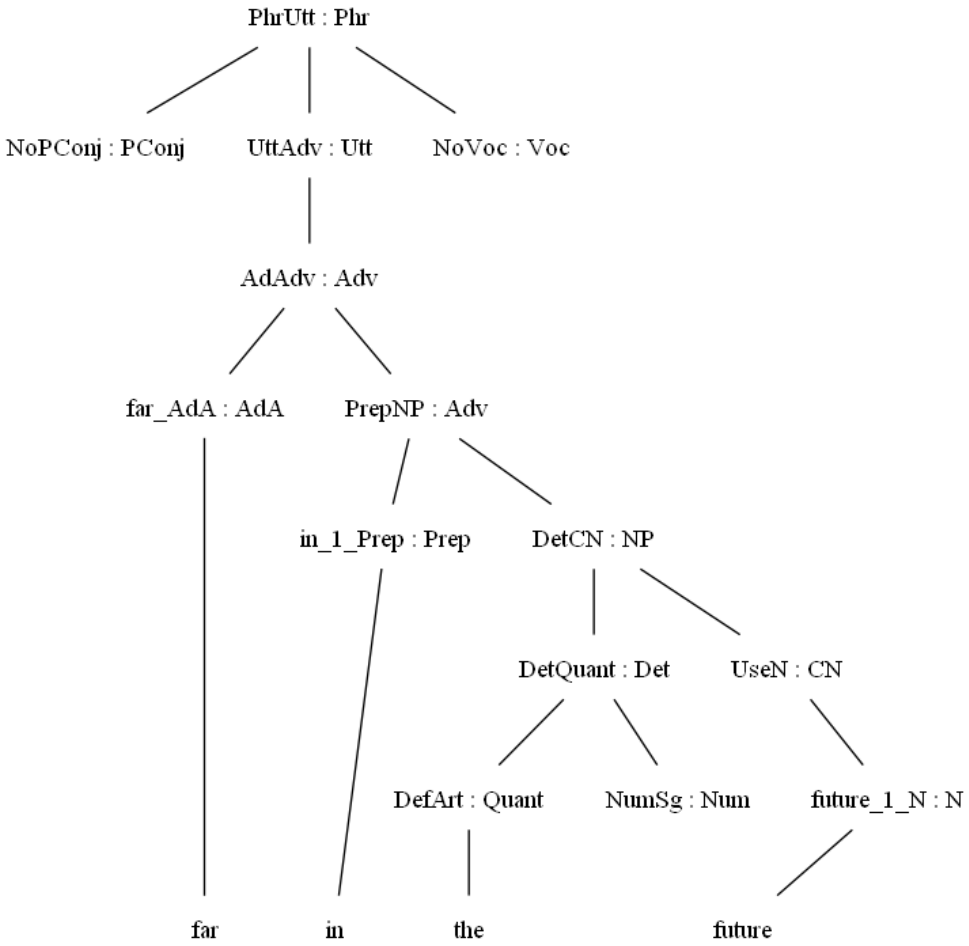
CHALMERS
UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG



A transformer-based parser for Grammatical Framework

Master's thesis in Computer Science - Algorithms, Languages and Logic

Alvaro Vazquez Velasco, Anthon Lenander

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Master's thesis 2024

A transformer-based parser for Grammatical Framework

Alvaro Vazquez Velasco, Anthon Lenander

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

A transformer-based parser for Grammatical Framework
Alvaro Vazquez Velasco, Anthon Lenander

Supervisor: Krasimir Angelov, Department of Computer Science and Engineering
Examiner: Jean-Philippe Bernardy, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Depicts the parse tree seen in Figure 2.1 in Section 2.2.1.

Typeset in LaTeX
Gothenburg, Sweden 2024

A transformer-based parser for Grammatical Framework
Alvaro Vazquez Velasco, Anthon Lenander
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Grammatical Framework (GF) is a programming language, the purpose of which, among many other things, is to describe natural languages through a common abstract syntax. There is already an algorithm available, to turn natural language into abstract syntax. This algorithm runs in polynomial time, but to produce any output, the input sentence needs to strictly follow grammatical rules. For example, misspelled sentences and sentences containing abbreviations will not be able to be parsed using the existing parsing algorithm. Thus, the objective of this thesis is to develop a transformer-based parser for Grammatical Framework that, given an input sentence, predicts a corresponding abstract syntax that when linearized, resembles the original sentence as closely as possible.

Keywords: Grammatical Framework, Machine learning, Machine translation, Natural language processing, Neural Network, Python, Transformer

Acknowledgements

A special thanks goes to our supervisor Krasimir Angelov, who has provided us with valuable knowledge in general, but in particular knowledge related to Grammatical Framework. He has also given insightful feedback and supported us throughout the entire process. Thanks also to our examiner Jean-Philippe Bernardy, for providing feedback on our report.

We would also like to take the opportunity to thank some of our fellow students, namely Hannes Häggander, Huitong Gao and Tianshuo Xiao, for peer reviewing our work and suggesting changes to make our thesis more easily readable.

Lastly, we would like to thank Chalmers University of Technology and University of Gothenburg for giving us the opportunity of writing our thesis in collaboration with them.

Alvaro Vazquez Velasco, Anthon Lenander, Gothenburg, September 2024

List of Acronyms

Below, a list of acronyms that have been used throughout this thesis, listed in alphabetical order, can be found:

BLEU Bilingual Evaluation Understudy
BP Brevity Penalty
BPE Byte Pair Encoding
DML Deep Machine Learning
FFNN Feed Forward Neural Network
GF Grammatical Framework
GPU Graphics Processing Unit
LAS Labelled Attachment Score
ML Machine Learning
MT Machine Translation
NLP Natural Language Processing
NN Neural Network
NPN Normal Polish Notation
RGL Resource Grammar Library

Contents

List of Figures

List of Tables

1
Introduction

This chapter aims to introduce the relevant background to this thesis project, the presentation of which can be found in Subsection 1.1. We will also explain the purpose of this thesis in Subsection 1.2. Our goals and challenges will be explored in further detail in Subsection 1.3. Last but not least, we would also like to present some boundaries for this thesis project, the discussion of which can be found in Subsection 1.4.

1.1 Background

Grammatical Framework (GF) is, amongst many things, a programming language, the purpose of which is to describe natural languages. GF does this through a common abstract syntax, more commonly known as an *interlingua* in the world of **Machine Translation** (MT). GF also provides functionality to *linearize*, that is, transform this abstract syntax, into concrete sentences in multiple natural languages. Apart from linearizing abstract expressions into natural language, GF is also able to *parse*, that is, transform natural language, into abstract expressions. Apart from parsing and linearizing, it should also be added that the framework provides random generation of trees (in the form of abstract expressions), type checking and conversion to constituency and dependency trees, among many other features.

For GF to be able to provide any results when parsing natural language given a concrete sentence, said sentence needs to strictly follow the grammatical rules of the natural language. This presents a particular kind of problem to be solved.

1.2 Purpose

The problem mentioned above could be stated as follows:

Given a concrete sentence as input, predict an abstract expression that, when linearized, resembles the original input as closely as possible.

This is the sole purpose of this master's thesis, to research and propose a machine learning model for parsing, that is able to solve the aforementioned problem and that can contribute to further development of GF.

1.3 Challenges

One of the main challenges for a project within **Machine Learning** (ML) such as this one, is the well known fact, that **Deep Machine Learning** (DML) models typically need a huge number of training examples to achieve an acceptable accuracy. For this specific project, an initial set of roughly 10,000 examples were provided to us by our supervisor.

The size of the initial set of training examples presents the risk of being to small in order for us to achieve a good enough accuracy in our resulting transformer-based model. A possible solution to this is to generate additional training examples. Luckily for us, GF offers generation of random sentences, which can and will be utilized. Another approach to generating more training data is to take already predefined sentence structures and switch out the words in the sentences. A final option would be to experiment with the combination of the two.

There are advantages and disadvantages to both approaches. Using Grammatical Framework to generate random sentences offers more flexibility in terms of syntactic construction. However, the choice of words will not be in our control, so the resulting sentences will not necessarily be naturally occurring ones. Taking the other approach will allow for more flexibility in the choice of words, whereas here, the sentence structures will overall be more rigid. Combining the two approaches might prove a valid option.

1.4 Thesis Boundaries

While this thesis project could in theory be extended to other languages, we have decided to limit our parser to parsing only the English language. This decision was made partly because English is the language with the best developed language resources, but also because the syntax and the morphology are easier for English. Thus, English is a good candidate for a first experiment. Future development could include extending the transformer-based parser to other natural languages.

2
Theory

The aim of this chapter is to go through relevant background, necessary to understand the rest of this thesis. The discussion will begin with MT in Subsection 2.1, with a specific focus on the interlingual approach to MT.

Then, we will explain in detail what Grammatical Framework and the Transformer model are. These discussions will span Subsections 2.2 through 2.5. Understanding the intricacies of these two topics is of utmost importance for one to understand what the problem that this thesis aims to solve is. A somewhat related topic, namely subword tokenization, will also be presented in Subsection 2.6.1.

In Subsection 2.7, we will also go through a couple of evaluation metrics, some of which were used to evaluate the resulting transformer model. The evaluation metrics used to evaluate our resulting transformer models include Levenshtein distance and BLEU score. Apart from those two, an explanation of Labelled Attachment Score will also be given.

Lastly, related work will be brought up, to set the thesis in context. This will be done in Subsection 2.8.

2.1 Machine Translation

There exists classicaly speaking, mainly three approaches to translation within the field of MT. We will focus this discussion on the interlingual approach, mostly because that is what GF makes use of. According to AlAnsary (2014), the interlingual approach is based on the idea that, while languages differ somewhat on the surface, they all share a common abstract syntax.

Honing in more deeply on the interlingual machine translation approach, it can be divided into two main components. The first component is to analyze a text in a natural language and produce a universal language-independent representation of the text. The second component is to turn the universal language-independent representation into another natural language. This approach presents several benefits, such as being able to create multilingual translation systems. It is also beneficial within other areas of **Natural Language Processing** (NLP).

2.2 Grammatical Framework

Grammatical Framework (GF) features a *Resource Grammar Library* (RGL), which currently covers the basic concrete syntaxes for 38 different languages (Ranta, 2011). For the development of the transformer-based parser, the grammar of highest interest for our purposes is the English grammar. These grammars generally feature relatively small lexicons. Therefore, the English lexicon used for this thesis project comes from GF WordNet. The WordNet provided by GF currently covers 28 languages, amongst which English, Swedish and Bulgarian are the most developed lexicons. As described in Section 1.4, we limit this thesis project to English, seeing as though this is only a first experiment.

The purpose of GF is to facilitate machine translation through a shared common abstract syntax, more commonly known as an interlingua. Users can then specify grammars, or concrete syntaxes, that include instructions for how to go from an abstract expression, to human intelligible language (Ranta, 2011).

To clarify what these concrete- and abstract syntaxes may look like, and what they are comprised of, the following two subsections will delve deeper and provide more details.

2.2.1 Abstract Syntax

In GF, the abstract syntax is defined by categories and functions, where each function has a corresponding type describing the types of the arguments and return type of said function (Ranta, 2011). To exemplify, let us have a look at an abstract expression:

PhrUtt NoPConj (UttAdv (AdAdv far_AdA (PrepNP in_1_Prep (DetCN (DetQuant DefArt NumSg) (UseN future_1_N))))) NoVoc

At first glance, abstract expressions in GF may look a bit alien to say the least. This exemplified abstract expression in particular represents the English sentence:

"Far in the future"

Abstract expressions in GF can be looked at as trees. For example, the sentence "Far in the future" is a phrase, represented by the category "Phr". "PhrUtt" is the first function in the abstract expression above, and has the type:

PhrUtt: PConj -> Utt -> Voc -> Phr

That is, a phrase utterance takes as arguments a phrase conjuction (PConj), an utterance (Utt) and a vocative (Voc), and produces a phrase (Phr).
Looking at the exemplified abstract expression above, it isn't too hard to find the functions with the corresponding categories, that the function "PhrUtt" takes as

arguments. The function "NoPConj" has category PConj, "UttAdv" takes an argument of type Adv and returns a result of type Utt, and "NoVoc" has category Voc.

Each parenthesis in the exemplified abstract expression above could be seen as one level of recursion. It could in a similar fashion also be viewed as a tree, which can be expanded in a recursive fashion function by function, where each function represents a subtree. For visual understanding, we refer to Figure 2.1 down below:

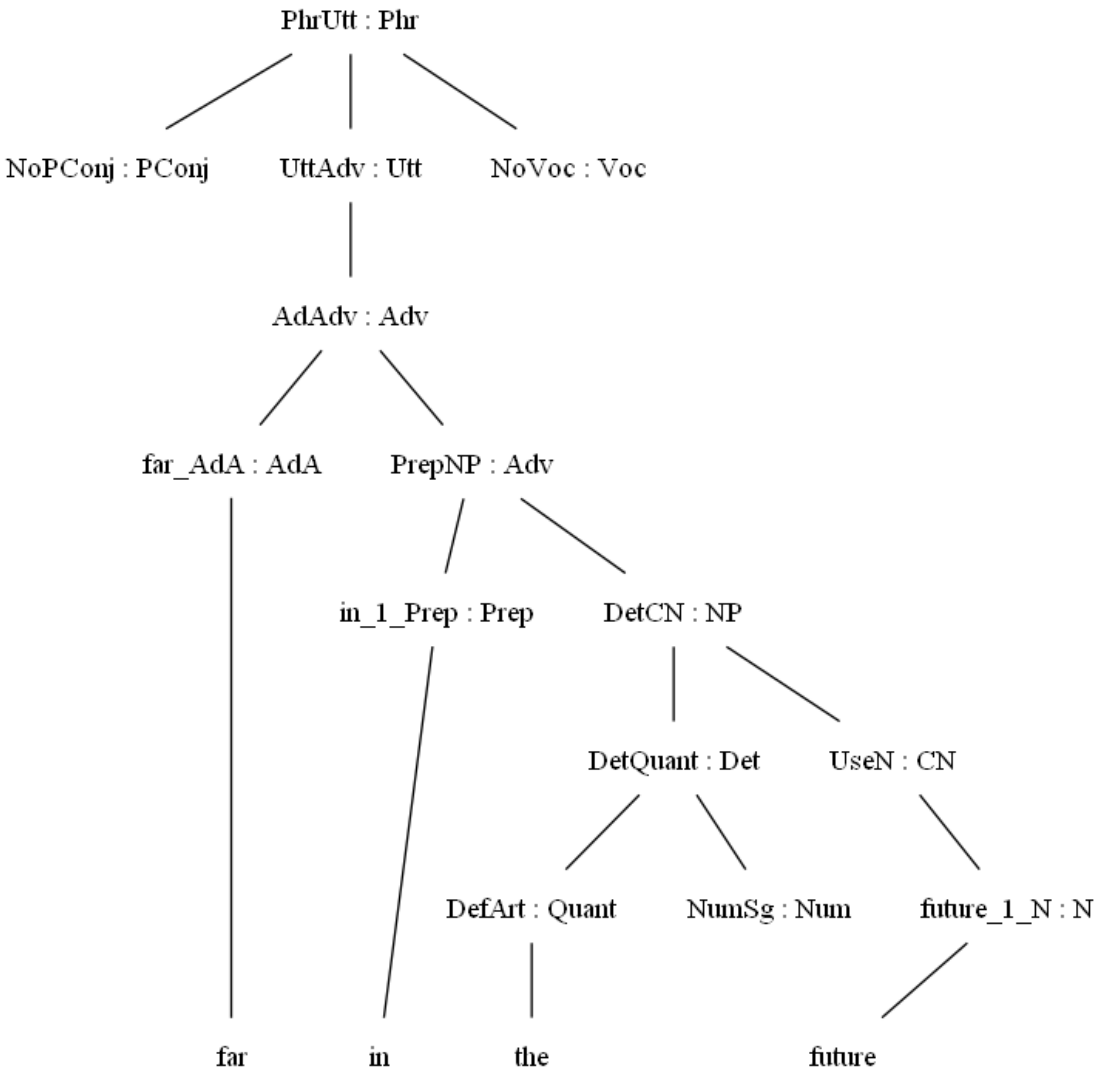


Figure 2.1: Illustrates a parse tree corresponding to the sentence "Far in the future".

For more information on different functions and categories in GF, we refer to the RGL API.

As a short notice, and in comparison with concrete syntaxes which will we explained in Section 2.2.2, the abstract syntax of GF does not feature any overloading. This fact is crucial for the tree building operation explained later on in Section 3.2.2.

2.2.2 Concrete Syntax

In the abstract syntax, only the type signatures of functions are defined. On a very high and abstract level of description, in a concrete syntax, these functions can then be implemented for each natural language. This is very much analogous to object-oriented programming, in the sense that in object-oriented programming, one can define an abstract class. This class can then have multiple implementations.

2.2.3 Linearization

The process of linearizing an abstract expression is in short, evaluating said abstract expression by using the implementation defined in a specific concrete syntax. Angelov (2011) describes in his PhD thesis, that this is done in a bottom up fashion.

2.2.4 Parsing

An important point to mention is that there is already a parser available for GF. The current method for parsing is outlined in Angelov (2009). It was then further developed in a collaborative effort in the seminal paper of Angelov and Ljunglöf (2014). The resulting parsing algorithm turned out to be several times faster than the previously proposed algorithm, among other benefits. The current parsing algorithm uses statistics to guess the best abstract expression given a sentence in a natural language.

The downside of this parsing method is that input sentences need to strictly follow the specified grammar for the used natural language, for GF to be able to parse said input sentence and return an abstract expression.

2.3 Training Corpus

A body of words, a corpus, is used in natural language processing to train machine learning systems. An entry in the corpus we use for this research project is depicted in the following example:

abs: PhrUtt NoPConj (UttNP (DetCN (DetQuant IndefArt NumSg) (AdjCN (PositA agile_2_A) (UseN mind_1_N)))) NoVoc
eng: an agile mind
swe: ett smidigt sinne
bul: гъвкав ум
key: 1 agile_2_A 01337785-a

Each entry has an abstract expression, a concrete expression in English, Swedish, and Bulgarian, and an identification key. The total amount of entries in the corpus is 9226.

2.4 Feed Forward Neural Networks

In the field of ML, a **Feed Forward Neural Network** (FFNN) is a type of neural network. The purpose of a neural network is to be able to stochastically interpolate between examples and approximate some function (Goodfellow, Bengio, and Courville, 2016). Let us denote this function by f^*. For example, given the context of MT, the function $y = f^*(x)$ could map an input sentence x in some natural language, to a sentence y in some other natural language.

According to Zou, Han, and So (2009), neural networks are structurally and functionally inspired by brains, and they are comprised of interconnected neurons. The function of a single neuron is to compute an output signal given a number of input signals. Mathematically, this process can be described with the following formula:

$$y = f(\sum_{i=0}^{n} w_i x_i - b_i) \tag{2.1}$$

The same process can be seen in visual format in Figure 2.2 down below:

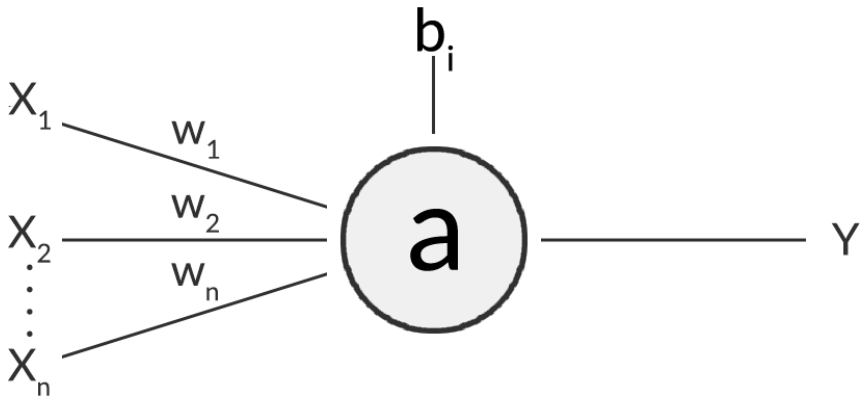


Figure 2.2: Illustrates a single neuron in an artificial neural network.

here, y is the computed output value of the neuron. Some input signals have a greater weight than other input signals, and part of the training of a neural network is to learn these weights. The values w_i refer to the weights associated with the input signals x_i. b_i is some threshold value, often referred to as a bias, hence the letter b. If the value of $\sum_{i=0}^{n} w_i x_i$ exceeds b_i, then we say that the neuron activates. Part of the training of a neural network also includes learning these biases for each neuron. Lastly, we have the activation function, denoted a. We will discuss some relevant activation functions more extensively in Section 2.4.1, and thus refer to said section.

According to the papers (Goodfellow, Bengio, and Courville, 2016) and (Zou, Han, and So, 2009), the expressiveness of neural networks in general does not lie within a single neuron. To the contrary, and as described earlier, a neural network is comprised of many interconnected neurons. These neurons are often organized into multiple layers. In a neural network, the types of layers that can be found include

input layers, output layers and potentially hidden layers. The potential hidden layers lay in between the input and output layer. As perhaps understood by the name, information in a FFNN travels forward, from left to right (recurrent neural networks also exist, in which information can flow both ways). All neurons in a previous layer are usually connected to all neurons in the following layer, like illustrated in Figure 2.3 down below:

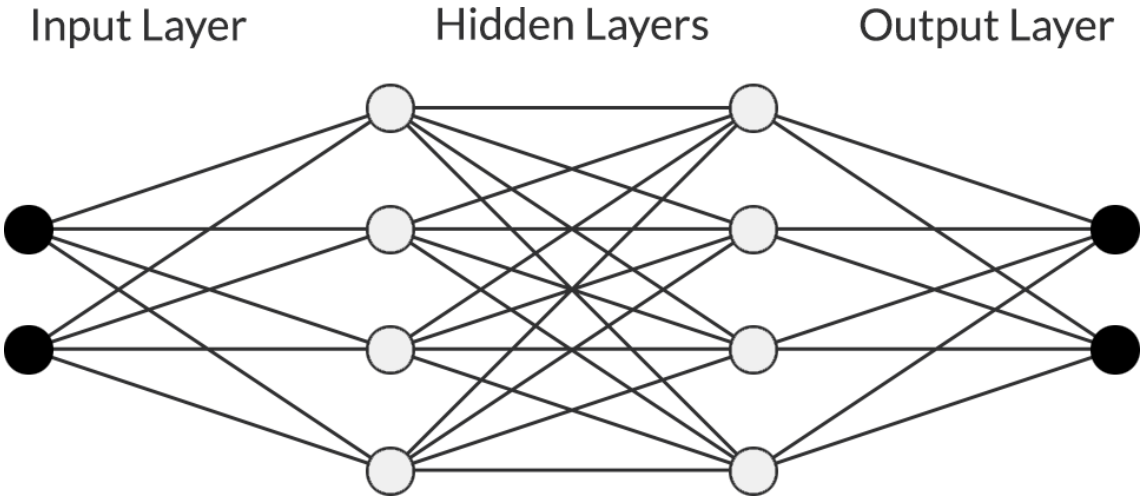


Figure 2.3: Illustrates a simple FFNN.

The example FFNN in Figure 2.3 above is comprised of four layers in total: An input layer containing two neurons, two hidden layers containing 4 neurons each, and a final output layer containing again two neurons. According to Zou, Han, and So (2009), a single hidden layer FFNN can only find an approximate solution to linearly separable problems. Therefore, in practice, the number of layers and the number of neurons per layer, is often much larger than portrayed in the example above. When the number of hidden layers in a FFNN exceeds that of 3 layers, it is said that the FFNN is **deep**.

According to Goodfellow, Bengio, and Courville (2016), Zou, Han, and So (2009) and Rumelhart, McClelland, Group, et al. (1986), training a FFNN often involves utilizing what is known as a back-propagation algorithm. In short, input data is propagated through the network, producing an output. An error can then be calculated by comparing the output of the network with some target output. This error is often referred to as the *cost function* (Zou, Han, and So, 2009). It is propagated backwards (from right to left) and is used to adjust the weights and biases mentioned earlier in this section. Referring again to Zou, Han, and So (2009), the goal is to minimize the cost function, which is often done using some gradient descent method (think finding the global minima in a two-dimensional graph, but for arbitrary dimensions). One thing to take into consideration regarding using gradient descent methods is the fact that they are not guaranteed to find a global minimum value.

2.4.1 Activation functions

The purpose of this subsection is to present activation functions relevant to this thesis. First, the softmax function will be discussed in detail in Section 2.4.1.1. We will then also explain how the ReLU activation function works, in Section 2.4.1.2.

2.4.1.1 Softmax

In simple terms, softmax is a function that turns a vector containing some k real values into a vector containing k real values all summing to 1. There are no restrictions to the values of the original vector, they can be positive, negative or even zero (Phillips, 2023). Thus, the function is often used at the end of a model, in order to calculate probabilities for each element in the output vector. The function itself is a generalization of logistic regression (two-class classification), made for multi-class classification (Phillips, 2023). Mathematically, the function can be described in the following way:

$$\sigma(\overrightarrow{Z})_i = \frac{e^{Z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{2.2}$$

where

σ = The softmax function
$\overrightarrow{Z}$ = The input vector
K = The number of elements in Z

As an example, let us take the vector $\overrightarrow{Z} = [2, 1, 15]$, where $z_0 = 2$, $z_1 = 1$ and $z_2 = 15$. Then we compute the softmax value for each element:

$$\sigma(\overrightarrow{Z})_0 = \frac{e^2}{e^2 + e^1 + e^{15}} = 2,26 * 10^-6$$
$$\sigma(\overrightarrow{Z})_1 = \frac{e^1}{e^2 + e^1 + e^{15}} = 8,31 * 10^-7$$
$$\sigma(\overrightarrow{Z})_2 = \frac{e^{15}}{e^2 + e^1 + e^{15}} = 0,99$$

The softmax value for each element in the vector is rounded to two decimal places, but it isn't hard to see that they all sum up to 1, as per the description of the softmax function.

2.4.1.2 ReLU - Rectified Linear Unit

This activation function in particular is very easy to understand. Mathematically speaking, it can be described as down below:

$$y = max(0, x) \tag{2.3}$$

That is, the output of a single neuron is either 0 if the output would have been negative, or just the original output value (tensorflow.org, 2024). It is widely used as activation function for all neurons in all layers but the output layer.

2.4.2 Layer Normalization

Internal covariate shift is a problem within the field of supervised machine learning, best described by Lei Ba et al. Ba, Kiros, and Hinton (2016):

> [...]the gradients with respect to the weights in one layer are highly dependent on the outputs of the neurons in the previous layer especially if these outputs change in a highly correlated way.
> (Ba, Kiros, and Hinton, 2016, p. 2).

In other words, certain circumstances make it difficult for a FFNN to learn effectively (Mudadla, 2023). To counteract this issue, a technique called **layer normalization** can be used. According to Ba, Kiros, and Hinton (2016) and Mudadla (2023), this technique involves calculating for each layer in a FFNN, the mean and the variance of the activations, and then for each layer, scale and shift said activations, to achieve a standard normal distribution (mean of 0 and variance of 1). Equation 2.4 down below showcases this procedure for a single neuron of a layer mathematically:

$$y_i = \gamma * \frac{(x_i - \mu)}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{2.4}$$

where y_i is the normalized output of the i-th neuron in the layer, x_i is the original output of the i-th neuron, μ is the calculated mean value, σ^2 is the calculated variance and both γ and β are learnable parameters. Last but not least we have ϵ, the only purpose of which is to avoid potential division by zero (Mudadla, 2023).

The benefits of using layer normalization are many. Ba, Kiros, and Hinton (2016) show in their paper some experimental results pointing to the fact that layer normalization allows for shorter training time and higher validation accuracy in certain tasks. Mudadla (2023) also points out that layer normalization allows for less careful initialization of the weights and biases in a FFNN. There are also a fair set of drawbacks to using layer normalization, some of which include an increased computational cost and reduced expressiveness. The increased computational cost is obvious, but the reduced expressiveness is due to the fact that certain tasks involve training data with a not so standard normal distribution. Enforcing a standard normal distribution may limit the ability of a FFNN to learn certain representations (Mudadla, 2023).

2.4.3 Dropout

The problem that dropout tries to address is that of overfitting. When we say that a machine learning model is overfit, we refer to the fact that is has learnt the training data *all too well*. In other words, the machine learning model is learning the statistical noise within the training dataset, so that when in practice faced with before unseen data, it performs very poorly (Yadav, 2023).

As described by Srivastava et al. (2014), utilizing droput, during training time, each neuron in a neural network is kept with probability p. Likewise, the neuron and all of its incoming and outgoing connections are removed with probability $(1 - p)$. Here, p is a tunable hyperparameter, and 0.8 seems to be the best value according to both Yadav (2023) and Srivastava et al. (2014). Figure 2.4 down below illustrates the effect of dropout during training:

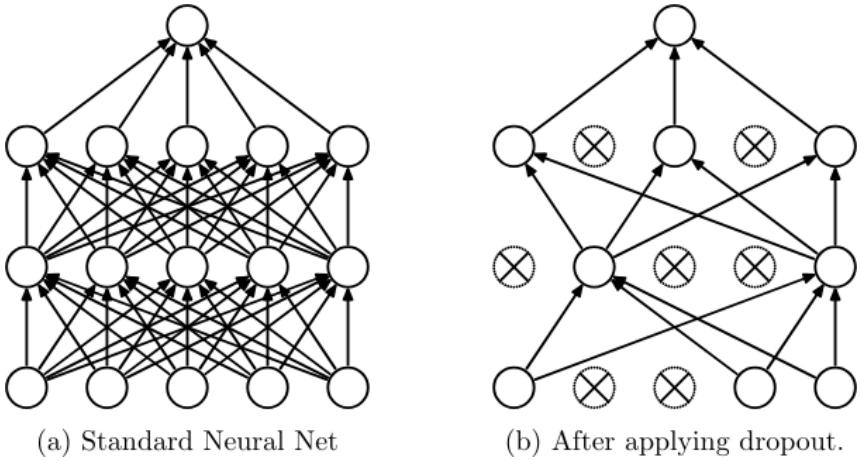


Figure 2.4: Illustrates a neural net a) with, and b) without dropout.
Source: Figure 1 in Srivastava et al. (2014).

When using the neural network, all neurons and their incoming and outgoing connections are kept. The weights on the edges going out from a neuron that was retained during training, are multiplied by the same probability of keeping the neuron during training time p. This is made to ensure that the expected output during training is the same as the actual output when using the neural network (Srivastava et al., 2014). This can be understood visually by looking at Figure 2.5 down below:

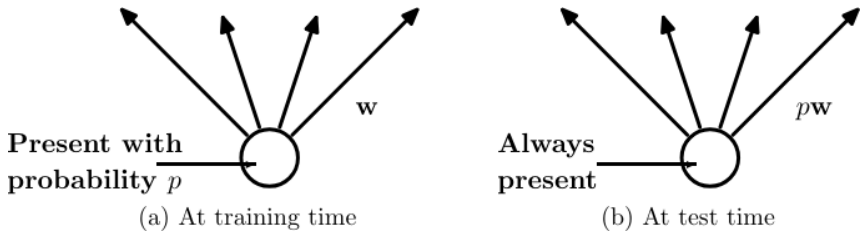


Figure 2.5: Illustrates dropout for a single neuron.
Source: Figure 2 in Srivastava et al. (2014).

2.4.4 Early stopping

Much like with dropout, early stopping also aims to address overfitting. As outlined by Prechelt (2002), the error generally decreases for each epoch (Recall that error refers to the difference between the target output and the actual output). Typically, a validation set is also used during training, the purpose of which is to measure the current accuracy of the model against unseen data at the end of each epoch. At some point, it is often noticeable that the error on the training set decreases, while the error on the validation set starts to increase again. Provided in Prechelt (2002) is a textbook example of this scenario, shown in Figure 2.6 down below:

Figure 2.6: Illustrates training vs. validation error.
Source: Figure 2.1 in Prechelt (2002).

Oftentimes, the error curves do not look as smooth as in Figure 2.6 up above. Prechelt (2002) also offers a more realistic validation error curve, which we reproduce in Figure 2.7 down below:

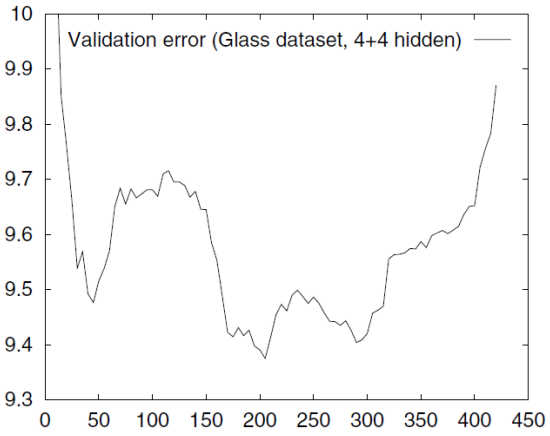


Figure 2.7: Illustrates an example of
validation error (y-axis) vs. time measured in epochs (x-axis).
Source: Figure 2.2 in Prechelt (2002).

We see in Figure 2.7 up above, that validation error reaches a local minimum after around 50 training epochs. The global minimum seems to be achieved after around 200 epochs, with a noticeable bump in between the local minimum encountered after around 50 training epochs. This introduces an uncertainty, because the validation error curves can never be known ahead of time (Prechelt, 2002). A question arises: when does one stop training?

Early stopping can be implemented using different classes of stopping criteria. One such class in particular is described by Prechelt (2002). The basic idea is that some metric is measured on the validation set after each epoch. The result of said measurement is then compared to the results from previous epochs. If the metric does not improve after some *patience* number of epochs (where *patience* is a tunable parameter), training is terminated. Oftentimes, the chosen metric to measure is the validation error. The procedure as a whole is described down below:

1. Split the dataset into a training- and a validation set.
2. For each training epoch:
    - Train using the training set
    - Evaluate using the validation set.
3. Terminate if chosen metric does not improve after *patience* number of epochs.

Another question which may arise at this point is: How does one split the original dataset into a training- and a validation set? This can be done in a couple of different ways, and may vary depending on the size of the original dataset. Therefore, we leave this question unanswered for now, and come back to it in Section 3.6.

2.5 The Transformer Architecture

In the paper *Attention is all you need* by Vaswani et al. (2017), the transformer deep learning architecture was proposed. The architecture became hugely popular in the following years, and is still very much popular to this day. It is a sequence-to-sequence model, and can be trained to perform tasks such as MT, which is what we will use it for in this thesis. The model is comprised of an encoder (see Subsection 2.5.3) and a decoder (see Subsection 2.5.4), and the architecture as a whole can be seen in Figure 2.8 down below:

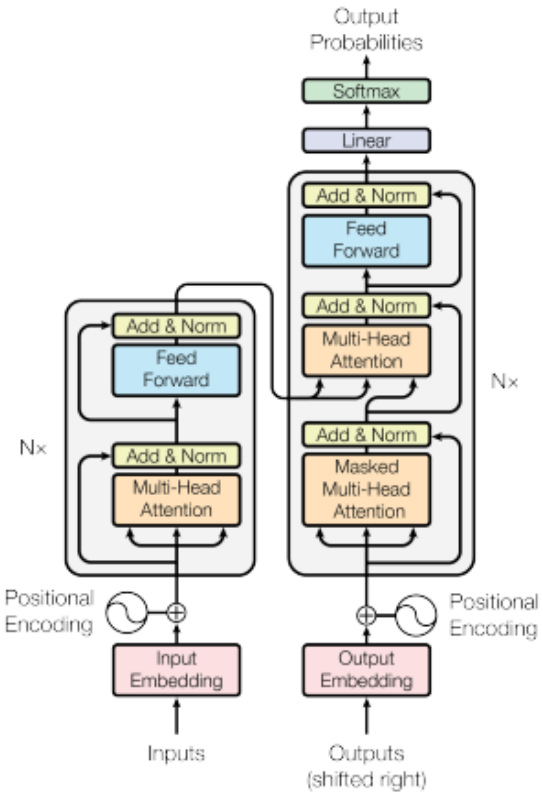


Figure 2.8: Illustrates the Transformer architecture.
Source: Figure 1 in Vaswani et al. (2017).

As can be seen in Figure 2.8 above, the input to the transformer model gets embedded using some form of input embedding (more on that in Section 2.6.1 and Chapter 3). Then, the input gets positionally encoded using the method described in Section 2.5.1.

Following the positional encoding of the input is an encoder comprised of N encoder layers, where N is a tunable parameter representing the number of layers of both the encoder and decoder stacks. The output of the encoder goes as input in the form of keys and values, to the cross attention part of the decoder. It is described in Vaswani et al. (2017), that the model is auto-regressive in that the decoder consumes all previously generated tokens as extra input to generate the next token.

The output of the entire architecture is then run through a softmax function to get output probabilities. For an explanation of the softmax function, see Subsection 2.4.1.1. We will in Chapter 3 see how we can use these output probabilities to, at every step, get the next (type correct) abstract function.

The decision was made to force outputs from each sub-layer inside the model, and the embedding layers, to be of dimension d_{model}. This is a tunable parameter used to facilitate the residual connections in the model, as described in Vaswani et al. (2017). These residual connections will be further explored, mainly in Sections 2.5.3 and 2.5.4. As will be seen later in Chapter 4, we experiment with different values of this parameter, to see how it affects the model accuracy.

The output of each sub-layer within both the encoder and the decoder is:

$$\text{LayerNormalization}(x + \text{SubLayer}(x)) \tag{2.5}$$

where SubLayer(x) is the function the sub-layer implements. This function could either be some global attention function or a FFNN, as seen in Figure 2.8 up above.

A FFNN is placed at the end of both the encoder and the decoder. Both of these FFNNs share the same design, and are comprised of an input layer with d_{model} neurons, a hidden layer with d_{ff} neurons and an output layer with d_{model} neurons Vaswani et al. (2017). Here, and much like with d_{model}, d_{ff} is also a tunable hyper-parameter, the purpose of which is to control the number of neurons in the hidden layer of the FFNNs. In more general terms, they both consist of two linear transformations. The connections between neurons in the input- and hidden layer also feature ReLU activations (Vaswani et al., 2017). The output of the FFNNs could thus be formulated as in Equation 2.6 down below:

$$\text{FFNN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{2.6}$$

where x is the input to the FFNN, W_1 and b_1 are the weight and bias matrices for the connections from the input layer to the hidden layer, and finally, W_2 and b_2 are the weight and bias matrices for the connections from the hidden layer to the output layer.

2.5.1 Positional Encoding

To add information about the relative and absolute positions of tokens in an input sequence, Vaswani et al. (2017) decided to add positional encoding to the input embeddings, which can be seen in Figure 2.8 in Section 2.5. To easily be able to sum up the input embeddings and the positional encodings for each token in an input sequence, the positional encodings are given the same dimensions (d_{model}) as the input embeddings.

When it comes to positional encoding, two options are generally available. Those options are learned- and fixed positional encoding. Inspired by Gehring et al. (2017), Vaswani et al. (2017) experimented with learned positional encodings, but found that learned and fixed positional encodings gave nearly identical results.

Input tokens are positionally encoded using sine and cosine functions with different frequencies, see Equation 2.7 down below:

$$\begin{aligned} PE_{pos,2i} &= sin(pos/10000^{2i/d_{model}}) \\ PE_{pos,2i+1} &= cos(pos/10000^{2i/d_{model}}) \end{aligned} \tag{2.7}$$

In Equation 2.7 above, pos is the position of the token in the input sentence, and i is the dimension. In other words, each dimension of the positional encoding corresponds to a sinusoid. This function was chosen in particular because it was hypothesized by Vaswani et al. (2017) that the model would easily be able to attend to relative positions this way. The reason being is that for any offset k, PE_{pos+k} can be expressed as a linear function of PE_{pos}.

2.5.2 Attention

As described by Vaswani et al. (2017), a formal definition of an attention function is:

> An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.
> (Vaswani et al., 2017, p. 3).

There exist a bunch of different attention functions, some of which will be discussed further in subsections 2.5.2.1-2.5.2.5. We will first and foremost have a look at *scaled dot-product attention*, before we move on to *multi-head attention*. There will also be a discussion about *cross attention*, before we finish of with two different flavors of *self attention*, namely *global self attention* and *causal self attention*. All of the different attention functions mentioned above are used in this thesis, and thus some amount of understanding is required to follow along.

2.5.2.1 Scaled Dot-Product Attention

This attention function in particular was developed by Vaswani et al. (2017), and can easily be understood visually by looking at Figure 2.9 below:

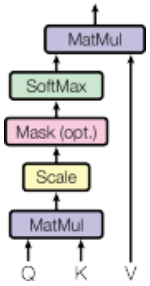


Figure 2.9: Illustrates the scaled dot-product attention function. Source: Excerpt from Figure 2 in Vaswani et al. (2017).

Figure 2.9 above contains input to the attention function in the form of three variables Q, K and V. These variables correspond to the queries, keys and values, as described in the definition of an attention function earlier in Section 2.5.2. The queries and keys share the same dimensionality, which is denoted d_k, whereas the values are of their own dimensionality, denoted d_v (Vaswani et al., 2017).

In practice, the attention is calculated on a set of multiple query vectors simultaneously. These vectors are gathered into a matrix. The same applies to the key and value vectors (Vaswani et al., 2017). The attention can then be calculated using the following formula:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.8}$$

As can be seen in Equation 2.8 above, the value QK^T given to the softmax function is divided by $\sqrt{d_k}$, hence *scaled* dot-product attention. The explanation as to why the dot product is scaled by $\sqrt{d_k}$ given by Vaswani et al. (2017), is that, for large values of d_k, the dot product QK^T grows very large magnitude-wise. This in turn may push the softmax function into areas of extremely small gradients. It should be noted that no further research was done by Vaswani et al. (2017) to confirm this theory. However, scaling by a factor of $\frac{1}{\sqrt{d_k}}$ seemed to have improved the performance of dot-product attention.

2.5.2.2 Multi-Head Attention

Before our discussion about multi-head attention can begin, something needs clarification. When we say multi-head attention, we refer to the fact that we perform scaled dot-product attention on our input multiple times. Another way to word it, is that we utilize multiple scaled dot-product attention layers in parallel. These layers are often referred to as heads, hence the name multi-head attention. The number of heads will hereon be denoted by the variable h in our discussion about multi-head attention.

Building on top of scaled dot-product attention, multi-head attention works by taking the queries, keys and values, and linearly projecting them h times into some learned linear projections of dimensionality d_k, d_k and d_v respectively. To clarify, the learned projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{model}}$ (Vaswani et al., 2017). The definition of d_{model} can be found in the beginning of this entire Section.

Scaled dot-product attention is then performed before the results are finally concatenated and linearly projected back into a d_{model}-dimensional output (Vaswani et al., 2017). For visual understanding we refer to Figure 2.10 below:

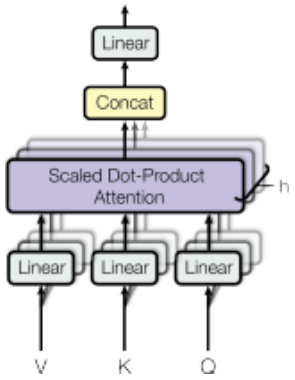


Figure 2.10: Illustrates the multi-head attention function.
Source: Excerpt from Figure 2 in Vaswani et al. (2017).

Mathematically speaking, multi-head attention can be described with the following equation:

$$\begin{aligned} &MultiHeadAttention(Q, K, V) = Concat(head_1, ..., head_h)W^O, \\ &where\ head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{2.9}$$

The benefit of multi-head attention in the context of NLP is that, unlike with a single attention head, the different heads in the multi-head attention layer can learn to attend to separate things completely. Take the following sentence as an example:

"My bike is broken, but it can be restored."

Using multi-head attention, one head could for example learn to attend from verbs to their direct objects. Thus, it could learn to understand that what is being restored

is "it". Imagine also that another head has learnt to attend from pronouns to nouns, so that the transformer now also understands that "it" refers to the "bike". This is simply not possible using single-head attention. Thus, hopefully the benefit of multi-head attention is even more apparent now.

2.5.2.3 Cross Attention

Another name for this attention function is encoder-decoder attention, and it really is just multi-head attention, but with a little twist to it. As perhaps already understood by its alternative name, this attention function involves both the encoder and the decoder of the transformer architecture. In cross attention, the queries come from the previous layer in the decoder, whereas the keys and values are just the encoder output (Vaswani et al., 2017). This form of attention is not uncommon in sequence-to-sequence models, such as the transformer architecture.

2.5.2.4 Global Self Attention

This particular attention function is, in accordance with cross attention, also just multi-head attention, but with a twist. The queries, keys and values are all the same, namely the output from the previous layer of the encoder (which would be the original input with input embedding and positional encoding applied). It is a mechanism used by the encoder part of the transformer architecture, which allows it to attend to all positions of the output from the previous layer in the encoder (Vaswani et al., 2017). In the context of NLP, this in turn allows the encoder to capture relationships between words in an input sentence, before translation.

2.5.2.5 Causal Self Attention

The difference between global- and causal self attention is very small, and in fact, they differ only in one small detail: Causal self attention only takes into account keys K_i for a query Q_j if i $<$ j. That is, only preceding tokens are considered. Contrary to global self attention, causal self attention is instead used by the decoder part of the transformer architecture.

2.5.3 Encoder

The purpose of the encoder is to map an input sequence of symbol representations, or tokens, $x = (x_1, ..., x_n)$, to a sequence of continuous representations $z = (z_1, ..., z_n)$, where z_i are vectors. As can be seen in Figure 2.11 below, a single encoder layer consists of two sub-layers. The first sub-layer is a global self attention layer, described previously in Subsection 2.5.2.4. The second layer is a simple FFNN. Relevant for both of these two sub-layers is the fact that their outputs are both residually connected to their inputs through the *Add* function. The output of the *Add* function then gets sent through a *Layer Normalization* function, as described in the beginning of Section 2.5.

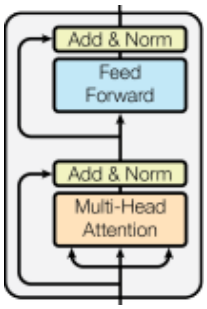


Figure 2.11: Illustrates the encoder in the transformer architecture. Source: Excerpt from Figure 1 in Vaswani et al. (2017).

2.5.4 Decoder

The decoder differs only slightly from the encoder. It is used to generate an output $y = (y_1, ..., y_m)$ given the sequence of continuous representations $z = (z_1, ..., z_n)$ from the encoder. Note that the length of the output $y = (y_1, ..., y_m)$ may differ from the length of the given continuous representation $z = (z_1, ..., z_n)$.

It employs causal self attention (see Section 2.5.2.5) for its first sub-layer, and then cross attention (see Subsection 2.5.2.3) for its second, but it keeps a FFNN as its final sub-layer. As we described in the beginning of this Section, the output from the encoder is used as input to the cross attention function of the decoder. For visual understanding, we refer to Figure 2.12 below:

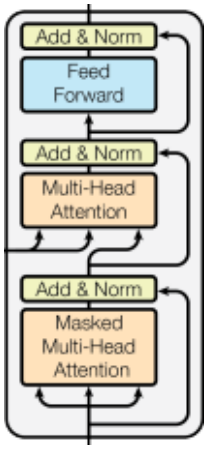


Figure 2.12: Illustrates the decoder in the transformer architecture. Source: Excerpt from Figure 1 in Vaswani et al. (2017).

2.6 Subword tokenization

In the field of NLP and MT, we can define tokenization as the task of segmenting a text into smaller units called tokens. Such tokens can take the form of words, utterances, or characters.

In a modern NLP context, and as described by Mielke et al. (2021), tokenization typically involves segmenting a sentence into non-linguistically motivated units, such units or "sub-words", are normally smaller than the classical "word-form" tokens. Sub-word tokenization helps the model understand new words by splitting rare occurring words into smaller meaningful units, while frequently occurring words form part of the vocabulary. For example, the tokenization of words can correspond to the syllabus or be randomly assigned.

The word "necessarily" can be split into "nec-es-sa-ri-ly", in this way the model could identify adverbs by the sub-unit "ly" as in easily or sadly. This showcases that while sub-words are non-linguistically motivated they possess a linguistic intuition.

An added feature of tokenization is that it can help identify the structure of the text, and it enables the model to better handle the variation of rare-occurring or unknown words.

2.6.1 Byte Pair Encoding

Byte Pair Encoding (BPE) is a compression algorithm that replaces common pairs of bytes with single bytes (Gage, 1994). The compression algorithm works by finding the most frequently occurring pairs of neighboring bytes and replacing them with a byte that is not in the original data. This process is repeated until no further replacement is possible, either because there are no more frequently occurring pairs or there are no more unused bytes to represent the pairs (Gage, 1994).

To illustrate how BPE works, let us have a string **abaaabcdaaab**. The most often occurring byte pair is **aa**, so it is replaced by **Z**, a non-occurring byte in the original data. The resulting string is **abZabcdZab**. In this resulting string, the next most common byte pair is **ab**, so it gets replaced by **Y**, which results in the string **YZYcdZY**. To clarify, **Z** = **aa** and **Y** = **ab**. The next most common byte pair is **ZY**, which we can replace with **X**. Thus, the final string becomes **YXcdX**. From this point onward, no more byte pairs can be exchanged with another non-occuring byte, and thus we terminate the procedure. In practice, the number of iterations can potentially be a very large number, and thus a limit to the number of iterations may have to be set.

2.7 Machine translation evaluation methods

To measure the quality of a machine generated translation, several automatic evaluation metrics have been proposed, all of which follow the same principle. An automatic evaluation method seeks to score an output from a machine translated system with respect to a small gold standard corpus of reference translations (Finch, Hwang, and Sumita, 2005). A good output translation can be defined by its closeness to an element of the reference corpus based on the input sentence. This closeness can also be seen as a notion of semantic equivalence, where the output and the reference are semantically equivalent if they achieve the same meaning.

2.7.1 BLEU

Bilingual evaluation understudy (BLEU) score is a technique that compares n-grams of a candidate sentence with n-grams of a reference translation and counts the number of matches (Papineni et al., 2002). On a baseline, the more matches, the better the candidate translation will be.

2.7.1.1 Unigram precision

Unigram precision is computed by looking at each word of a candidate translation and seeing if it appears in any of the translation reference sentences Papineni et al. (2002).

Let us take as an example the following candidate and reference sentences :

Candidate: the the the the
Reference: The cat is on the floor

The candidate reference has four elements, "the", each appearing on the reference sentences. Therefore it can be concluded that the precision for such a sentence will be $\frac{4}{4} = 1$. This simple measurement yields a high precision over a very improbable sentence. Machine translated systems tend to over generate "reasonable" words and such precision measurement would be unreliable.

2.7.1.2 Modified n-gram precision

As described in its original paper by Papineni et al. (2002), modified unigram precision starts by counting the maximum number of times a word occurs in any of the translation references. Afterward, clip the total count of each candidate word by its maximum reference count, add the clipped counts up, and divide by the total number of candidate words.

Precision is computed similarly for any candidate n-gram. The modified precision score, p_n, for the entire corpus will look as in equation 2.10:

$$p_n = \frac{\sum_{C \in Candidates} \sum_{n-gram \in C} Count_{clip}(n-gram)}{\sum_{C' \in Candidates} \sum_{n-gram' \in C'} Count_{clip}(n-gram')} \tag{2.10}$$

2.7.1.3 Brevity penalty

Brevity penalty (BP) is a factor that ensures high-scoring candidate translations match the reference translations in length, word choice, and word order. Let c be the length of a candidate translation and r be the reference corpus length.

$$BP = \begin{cases} 1, & \text{if}, c > r, \\ \exp^{(1-r/c)}, & \text{if}, c \leq r, \end{cases} \tag{2.11}$$

BP is computed as in equation 2.11.

2.7.1.4 Full *BLEU* score

The computation of the BLEU score is made by taking the geometric mean of the corpus modified n-gram precision scores and then multiplying by an exponential brevity factor. The modified n-gram precision is computed up to length N and is multiplied by a positive weight w_n. The complete equation looks as in 2.12:

$$\text{BLEU SCORE} = BP * \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{2.12}$$

For equation 2.12, Papineni et al. (2002) used a baseline of $N = 4$ and $w_n = 1/N$ which in modern practice is the standard.

2.7.2 Levenshtein distance

As described in Levenshtein (1965), Levenshtein distance is a method to measure the similarity between two strings. Similar to other edit distance methods, the Levensthein distance is computed as the minimum number of single character edits, which have the form of insertion, deletion, or substitution.

The distance between two strings, a and b, can be represented as in equation 2.13:

$$lev(a,b) = \begin{cases} length(a) & \text{if length(b)} = 0, \\ length(b) & \text{if length(a)} = 0, \\ lev(tail(a), tail(b)) & \text{if head(a)} = \text{head(b)}, \\ 1 + \min \begin{cases} \text{lev(tail(a), b)} \\ \text{lev(a, tail(b))} \\ lev(tail(a), tail(b)) \end{cases} & \text{otherwise.} \end{cases} \tag{2.13}$$

In equation 2.13 *tail(x)* refers to all but the first character of a string x, while *head(x)* refers to the first character of string x.

To exemplify, a functional style implementation of the Levenshtein distance between two words is as shown below:

$$
\begin{aligned}
&= \text{lev}(\text{kitten, kite}) \\
&= \text{lev}(\text{itten, ite}) \\
&= \text{lev}(\text{tten,te}) \\
&= \text{lev}(\text{ten, e}) \\
&= 1 + \min(\text{lev}(\text{en,e}), \text{lev}(\text{ten,""}), \text{lev}(\text{en, ""}))
\end{aligned}
$$

2.7.2.1 Average Levenshtein accuracy

The Levensthein distance follows equation 2.13 and is solved using a matrix to calculate the distance between strings A and B. Such a matrix is filled out from the upper-left corner, where the bottom-right corner will yield the Levenshtein distance. An example of a Levensthein distance matrix can be seen in the image below.

	#	k	i	t	t	e	n
#	0	0	0	0	0	0	0
k	1	0	1	1	1	1	1
i	2	1	0	1	2	2	2
t	3	2	1	0	1	2	3
e	4	3	2	1	1	1	2

Figure 2.13: Illustrates Levenshtein distance matrix to compute the similarity between two strings

The bottom-right corner yields a value of two, which is the computed distance.

The average Levenshtein accuracy is then computed by mapping the Levenshtein distance into a value between zero and one. The overall accuracy is calculated as in equation 2.14

$$
\text{accuracy} = 1 - \frac{\textbf{levenshtein distance(input, output)}}{\max(\text{length(input)}, \text{length(output)})} \tag{2.14}
$$

The code snippet below presents the average Levenshtein accuracy for the gold standard corpus.

```
for r, t in test_corpus:
  similarity = 1-levenshtein_distance(w1=r,w2=t)/max(len(r),len(t))
  accuracy += similarity
return accuracy / len(test_corpus)
```

2.7.3 Labelled Attachment Score (LAS)

In the context of parse trees, label attachment refers to the correct assignment of a word to its head and the correct dependency relation. Dependency relations between words, or dependency grammar, depict the connection between words in a sentence. A dependency structure can be determined by the relation between a word, which becomes the head of the structure, and all its dependents.

A simple dependency structure is shown in the example below.

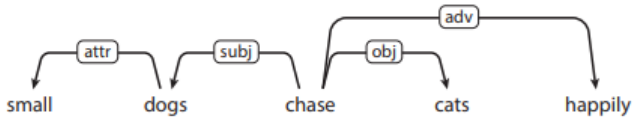


Figure 2.14: Illustrates a dependency structure.
Source: Figure 1 in De Marneffe and Nivre (2019).

Label attachment scores compute the accuracy of a parser by measuring how many words have been assigned both to the correct syntactic head and the correct label.

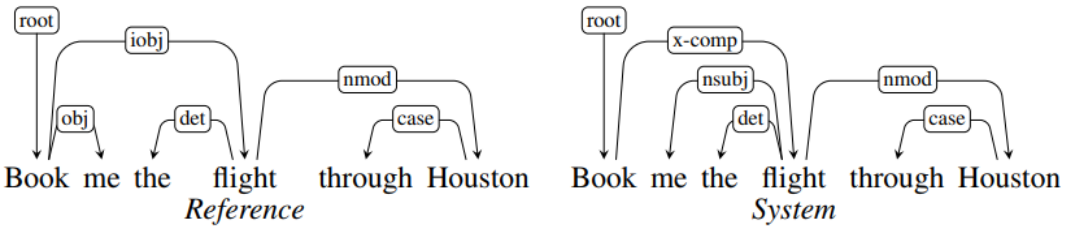


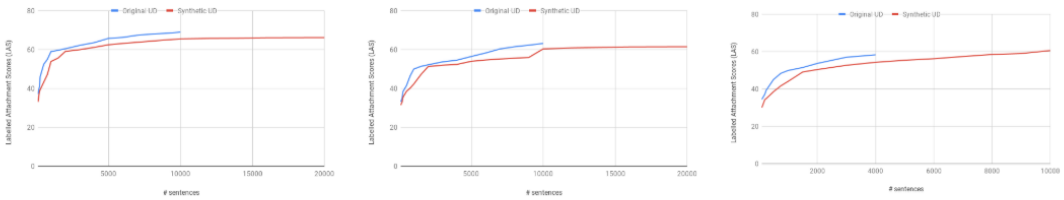
Figure 2.15: Illustrates comparison of system output and a gold standard tree with LAS of 2/3.
Source: Figure 15.15 in Jurafsky et al. (2014).

2.8 Related Work

2.8.1 Bootstrapping UD treebanks for delexicalized parsing

Experiments training different (non transformer-based) parsing models have been conducted by Kolachina and Ranta (2019) already back in 2019. In their work, they generated trees using GF, from which dependency trees were produced. These dependency trees were then used to train a dependency parser. We also do automatic tree generation in this thesis to produce a training corpus.

Kolachina and Ranta (2019) trained three parsing models for English, Finnish and Swedish respectively, the results of which, in terms of LAS, can be seen in Figure 2.16 down below:



(a) Learning curves for English (b) Learning curves for Finnish (c) Learning curves for Swedish

Figure 2.16: Illustrates parser training results.
Source: Figure 4 in Kolachina and Ranta (2019).

What can be seen from the results of said experiments, featuered in Figure 2.16 up above, is the fact that the measured score seems to be reaching some maximum, after which it simply flattens out. This is in proportion to how many training examples, in the form of sentences in each respective language, the parsing models were given. The point at which the measured score seems to flatten out is around 10,000 training examples.

2.8.2 Algebraic Positional Encodings

Since the introduction of the transformer model, several efforts have been made to improve upon the often ad-hoc positional encoding methods. Because abstract expressions in GF can be seen as trees, one relevant alternative to the sinusoidal positional encoding method described in Vaswani et al. (2017) is the special tree-positional encoding proposed by Kogkalidis, Bernardy, and Garg (2023). The work of the authors is guided by Montague's perspective, which follows:

> Syntax is an algebra, semantics is an algebra, and meaning is a homomorphism between them.
> (Janssen, 2014) (Kogkalidis, Bernardy, and Garg, 2023)

In the paper, a syntax and semantics for trees are proposed. The authors then elaborate on how these ingredients can be combined, creating a special tree-positional encoding. For more details in regards to the implementation, we refer to Kogkalidis, Bernardy, and Garg (2023).

3

Methodology

3.1 Training Corpus

The corpus, presented in this research project, used for training our transformer model was originally put together by our thesis supervisor. Using the parsing algorithm depicted in 2.2.4, our supervisor provided us with initial training data for the transformer-based parser. Sentences were taken from Princeton WordNet (Miller, 1995) (Fellbaum, 1998) and VerbNet (Schuler, 2005). These were then parsed using the current parsing algorithm, to produce corresponding abstract expressions. In some instances the abstract expressions were manually type-corrected, to account for wrongful predictions by the existing algorithm.

3.2 Tree operations

We will in Section 3.2.1 explain how and why we flatten GF trees into sequences. We will also in Section 3.2.2 make it clear how the operation is reversible, and why we need it to be.

3.2.1 Tree flattening

The reason for flattening GF abstract expressions into sequences is the fact that transformer models do not operate on trees. Instead, they work on sequences of tokens. As discussed in Section 2.2.1, each abstract expression can also be thought of as a tree. As such, each abstract expression needs to be flattened, in order to convert it to a sequence of tokens.

The flattening itself works by recursively going down a tree, and appending the names of the abstract functions to a list if the type of the expression is not a function application. Should the type of the expression be a function application, then we recursively flatten the function itself, and its arguments. If the length of the list of function names is less than the length that the transformer model expects, we append some padding tokens to make the list the desired length. This whole procedure is outlined in the form of a code-listing in Appendix A.

3.2.2 Tree building - the reverse operation

The resulting sequence of abstract function names produced during inference is unusable by GF, and needs to be converted back into a tree. The reverse operation of tree flattening is made possible due to the fact that the arity of each function is known. Also, as explained in Section 2.2.1, the abstract syntax of GF does not feature overloading. We can therefore recursively build a tree using this knowledge.

The operation utilizes Normal Polish Notation (NPN) (Łukasiewicz, 1970). An example of NPN is "+ 3 4" vs. "3 + 4". In NPN, first the operation is defined, and then the arguments are. Abstract expressions in GF are written in the same way. Take the toy example above with "+ 3 4"; the 3 could be the result of some other operation, and so too could 4. "+ 3 4" = "+ (- 5 2) (+ 3 1)" and so on and so forth. We build trees from abstract function names in the same way, and the arity of each abstract function helps us know how many arguments an operation takes.

To avoid confusion in regards to the discussion about this topic (*function* in programming vs. abstract *function* name), we instead refer to Appendix B. There, a code-listing of the above outlined procedure can be found.

3.3 Type Checking

During inference, given an English sentence, our transformer-based model returns a sequence of abstract function names. These abstract function names need to be type-correct for GF to be able to parse them into a tree. Take the abstract function "PhrUtt" for example. It has type "PhrUtt: PConj -> Utt -> Voc -> Phr", meaning that GF expects the next abstract function to be of type "PConj". If all abstract functions in the resulting sequence during inference adhere to these rules, then GF is able to correctly linearize the sequence.

In the production of this sequence of abstract function names during inference, we help the transformer return type-correct predictions by keeping a stack of the next expected function type, given the type of the previously generated token. Using again the same example as in the previous paragraph, say that the first prediction by a model is "PhrUtt". This function has type "PhrUtt: PConj -> Utt -> Voc -> Phr". That is, we expect the next prediction to be of type PConj. The model can be helped by filtering out all predictions that do not have the correct type. Remember that the transformer outputs d_{model} predictions for each token generation iteration, but we are only interested in the highest probability type-correct token. Thus, the prediction with highest probability among the remaining type-correct predictions is picked. Lastly, the stack needs to be updated with the next expected type given the currently predicted abstract function.

In each iteration of this described sequence generation procedure the model consumes all previously generated tokens to generate the next token, as described in Section 2.5. The generation of tokens continues until either the maximum number

of tokens have been generated, or an "[END]" token is generated.

3.4 Implementing the Transformer Model

The implementation of the transformer model is based on the parameters proposed by Vaswani et al. (2017), and follows the same structure as in Figure 2.8. It is built using the TensorFlow Python package.

The transformer-based parser follows the traditional encoder-decoder structure. The feed-forward network follows a traditional structure with a ReLu activation function, a normalization layer, and a dropout rate of 0.1.

At a high level the model counts with three main functions; train, test, and use.

3.4.0.1 Train function

Training consists of extracting the training corpus and shuffling it into smaller batches. Given batches are split into training and validation sets. In the following step, we initialize the transformer model and set an early stopping callback to limit the number of epochs. Finally, the transformer is fitted with the training dataset and the resulting model is saved for later testing and use.

Section 3.3 goes deeper into the training process and includes a description of adding a byte pair encoding to the pipeline.

3.4.0.2 Testing function

Using the resulting transformer-based parser we run through it a "golden standard" corpus composed of five hundred and one, never seen before, sentences. Each abstract syntax sentence the transformer yielded is linearized into its concrete syntax. Using average Levenshtein accuracy, described in section 2.7.2.1, the accuracy of the sentence is obtained by comparing the original input against the output obtained through the transformer.

3.4.0.3 Use function

Use function test individual sentences fed by the user. This becomes useful for comparing one-to-one sentences.

3.5 Data Generation

As described in Section 1.3, we were given by our supervisor an initial set of roughly 10,000 training examples. The general consensus in the DML community seems to be that the ideal number of training examples cannot be known ahead of time. Typically though, DML models require a lot more data in order to achieve an acceptable accuracy. In this thesis, we compare the outcome of three different means of data

generation, described in Subsections 3.5.1 - 3.5.3. Common for all three approaches, is the fact that we append the original training examples to the resulting text files.

3.5.1 Random Data Generation

As described in Section 1.1, GF features random generation of trees, in the form of abstract expressions. Using the Python package pgf mantained by, among others, Angelov (2023), which provides Python bindings to the GF runtime, a random tree can be generated and linearized into English with the following python code:

```
import pgf

grammar = pgf.readNGF("ParseEng.ngf")
english = grammar.languages["ParseEng"]

abstract_expression = grammar.generateRandom(grammar.startCat)[0]
english_sentence = english.linearize(abstract_expression)
```

The abstract expression and the English sentence can then be written to a text file and later used for training the model. To generate multiple training examples, one can simply loop until enough training examples have been generated.

3.5.2 Data Generation through Text Variation

Different from the random data generation method described in the previous subsection, data generation through text variation involves changing tokens in abstract expressions in a meaningful and type correct fashion. We do this by first flattening the abstract expression, as described in Section 3.2.1. Then, for each token in the resulting sequence, we check if its corresponding category is of interest. To exemplify, consider the following abstract expression:

PhrUtt NoPConj (UttS (UseCl (TTAnt TPastSimple ASimul) PPos (PredVP (UsePron he_Pron) (AdvVP (UseV look_around_V) (PrepNP for_Prep (DetCN (DetQuant DefArt NumSg) (PossNP (UseN source_4_N) (DetCN (DetQuant DefArt NumSg) (UseN disturbance_1_N)))))))) NoVoc

This abstract expression represents the English sentence:

"He looked around for the source of the disturbance"

In Python, we can define a list of sought after categories, and check each token for its category. If the token's category matches any of the sought after categories, then we can record the token and its index in the original list of tokens, like so:

```
sought_after_categories = [
    'A', 'N', 'Pol', 'Pron', 'Tense', 'V'
]
```

```
    modifiable_tokens = []
    for index, token in enumerate(tokens):
        category = grammar.functionType(token).cat
        if category not in self.sought_after_categories:
            continue

        modifiable_tokens.append(
            (token, index)
        )
```

To clarify, the sought after categories seen in the code-listing above are:

- A - Adjective
- N - Noun
- Pol - Polarity
- Pron - Pronoun
- Tense - Grammatical tense
- V - Verb

Tokens of interest in the example abstract expression above include:

- TPastSimple - Past simple grammatical tense
- PPos - Positive polarity
- he_Pron - The pronoun he
- look_around_V - The verb "look around"
- source_4_N - The noun "source"
- disturbance_1_N - The noun "disturbance"

The token TPastSimple represents a function with category Tense. Other tenses include present tense, past tense, future tense and conditional tense. There are 5 tenses in total. The token PPos represents a function with category Pol. There are two polarities in total, positive and negative. Exchanging the token PPos in the sentence above would result in negating the sentence. For clarity:

"He did not look around for the source of the disturbance"

The token he_Pron represents a function with category Pron. There are a total of 9 pronouns in GF Wordnet, including I, you, he and she.

The rest of the tokens are mainly of category N and V. For these tokens in particular, we utilize a random walk method for retrieving synonyms. GF Wordnet features functionality to filter out synonyms to a word of a specific category. We can thus construct a list of synonyms by first getting the synonyms to the input word, and then loop through the list of synonyms. For each iteration of this loop, we pick a random unseen word from the list of synonyms, and append its synonyms to the list. We then mark it as seen. We do this until we are either satisfied with the number of synonyms, or they are exhausted. For all intents and purposes to the following discussion, let us say that each noun and verb is able to produce 10

synonyms in total, be this in practice true or not.

For the example abstract expression above, we would then be able to generate $5 * 2 * 9 * 10 * 10 * 10 = 90,000$ variations in total. This is more than enough, and we hypothesize that for some shorter sentences, the number of variations will be a lot smaller. This in turn would create a very unbalanced training dataset if all variations for each sentence were to be considered. We thus shuffle the resulting list of variations for each abstract expression, and then pick some amount of variations uniformly at random.

Running our "text-variator" on the example abstract expression above with an amount of 10 resulting variations could produce sentences like the following ones:

"She did not look around for the source of the disturbance"
"He looks around for the source of the disturbance"
"She would not look around for the origin of the disturbance"
"They are looking around for the source of the turmoil"
"I was searching for the source of the noise"
"We did not look around for the source of the disturbance"
"I did not look for the origin of the commotion"
"You do not look for the origin of the disturbance"
"She does not look around for the source of the nuisance"
"It could not look around for the origin of the disturbance"

In this way, we are able to re-purpose the original training examples. As previously discussed, this approach gives us more flexibility in the choice of words. However, sentence structure will be more rigid compared to that of generating random data.

3.5.3 The Combined Approach

We also hypothesize, that perhaps generating some amount of random data, appending said random data to the original data and then generating some amount of variations on the resulting dataset, would improve the accuracy of the model. For the combined approach, this is exactly what we do.

3.6 Training a Transformer model

Before training of a transformer model can commence, certain preprocessing of the dataset is required. The preprocessing steps are as follows:

1. Tokenize the English sentences and the abstract expressions
2. Construct a corpus containing pairs of English sentences and abstract expressions

As a simple example of the above described process, consider the following pair of an English sentence and an abstract expression:

Abstract: PhrUtt NoPConj (UttS (UseCl (TTAnt TPastSimple ASimul) PPos (PredVP (DetCN (DetQuant DefArt NumSg) (UseN enemy_1_N)) (UseV fall_back_3_V)))) NoVoc

English: The enemy fell back

The English sentence is fed into a pre-trained byte pair encoder, which will yield a list of numbers representing the sentence. The transformation of the abstract expression on the other hand, is a lot easier to showcase:

['PhrUtt', 'NoPConj', 'UttS', 'UseCl', 'TTAnt', 'TPastSimple', 'ASimul', 'PPos', 'PredVP', 'DetCN', 'DetQuant', 'DefArt', 'NumSg', 'UseN', 'enemy_1_N', 'UseV', 'fall_back_3_V', 'NoVoc']

This list of abstract function names is then converted into a list of numbers, using a rather simple tokenization technique called string lookup. After having gone through all pairs of English sentences and abstract expressions, one can construct a vocabulary for both English and abstract. The string lookup method is then just a dictionary, mapping each token in the vocabulary to a number. Say 'NoVoc' is the 300th entry in the dictionary. Then, tokenizing 'NoVoc' using string lookup would yield 300. This is done on the entire list of abstract tokens, to yield a list of numbers. (It should be noted that this operation can also be done in reverse, to convert the transformer output back into abstract tokens).

We then create a dataset out of the resulting corpus, that we shuffle and turn into smaller batches. The dataset is then split into a training- and validation set, using a rather simple mathematical formula to determine the size of the validation set. The formula in question is outlined in Equation 3.1 down below:

$$\text{Size} = \min(\text{Percentage} \times \text{NumDatasetBatches}, \text{MaxBatches}) \tag{3.1}$$

Where *percentage* in the above Formula is a tunable hyperparameter. So too is the maximum number of validation batches.

To justify the use of the above formula, consider a dataset with 1M examples. Say also that a batch size of 64 is used. That would result in 15,625 batches. Having *percentage* in the above formula set to, say, 5 percent, would result in roughly 780 validation batches. Remember also that every batch contains 64 examples each. That is, the validation set would have a total of about 50k examples, which is more than enough. Therefore, we decided to also add a tunable upper bound on the number of validation batches.

We then specify a learning rate schedule for the optimization algorithm used, namely Adam, which is a stochastic gradient descent algorithm (as briefly described in Section 2.4 when discussing the training of a FFNN). The learning rate schedule can be visualized for easier visual understanding. Therefore, we provide the learning rate schedule in graphical format, in Figure 3.1 down below:

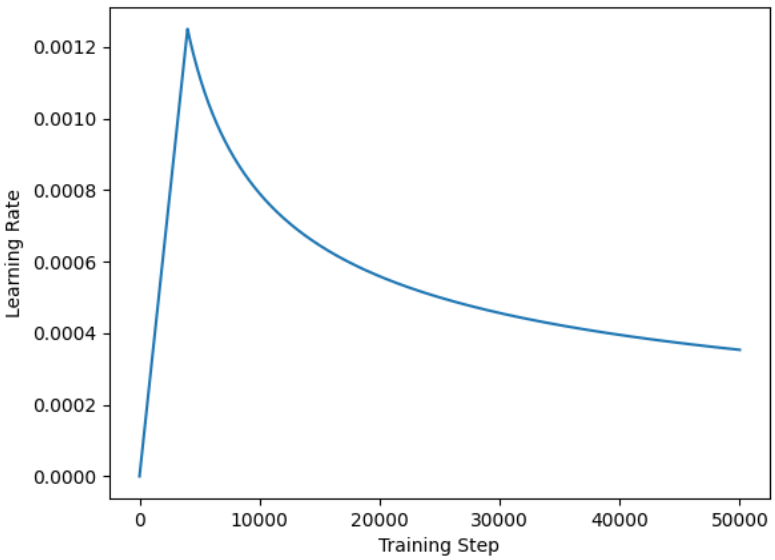


Figure 3.1: Illustrates the employed learning rate schedule.

As can be seen in Figure 3.1 above, the initial learning rate is rather slow, until about 5,000 training examples in, at which point it reaches its maximum value. Then, after having reached its maximum value, the learning rate slowly diminishes over time.

As loss function, *Sparse Categorial Cross-entropy* loss is used. Because padding tokens are used, we need to mask them when computing the loss.

The last piece of the puzzle is early stopping. We make use of early stopping by monitoring the validation loss, and terminating early if it doesn't improve after some number of training epochs.

3.6.1 Training Environment

This sections serves the purpose of explaining the different training environments utilized during this thesis. GPUs are oftentimes used to train machine learning models, because the underlying algorithms can be expressed in terms of linear algebra. Therefore, one can leverage the parallelism that GPUs provide. First, we will discuss the cloud computing environment, to then draw some parallels between said environment and one of our local machines, which will also be leveraged.

3.6.1.1 Cloud computing

Cloud computing served the purpose of optimizing the training speeds for the different models. Using Google Cloud services we set up two virtual machines with one NVIDIA T4 GPU each which allowed the training of two different models parallel. We provide the full configuration of the virtual machine in the table below 3.1

Technical Specification for the virtual machine		
GPU	Booting image	Size of the vm
NVIDIA T4	Debian 11 with python 3.10	100GB

Table 3.1: Showcases the technical specification for the virtual machine.

We provide the specifications of the NVIDIA T4 in Table 3.2 down below:

Technical Specification for NVIDIA T4			
Architecture	Compute Capability	Clock Speed	Memory
Turing	CUDA 10	10 Gbps	16GB GDDR6 256-bit 300.0 GBps

Table 3.2: Showcases the technical specifications for the NVIDIA T4 GPU.

NVIDIA type GPU's were choosen due to the compatibility with CUDA drivers, which the transformer model implements through the tensorflow python library.

3.6.1.2 Local machine

One alternative to cloud computing is using a local machine. To our disposal, we have a computer equipped with an NVIDIA GeForce RTX 2060. We provide the specifications of the NVIDIA GeForce RTX 2060 in Table 3.3 down below:

Technical Specification for NVIDIA GeForce RTX 2060			
Architecture	Compute Capability	Clock Speed	Memory
Turing	CUDA 7.5	14 Gbps	6GB GDDR6 192-bit 336.0 GBps

Table 3.3: Showcases the technical specifications for the NVIDIA GeForce RTX 2060 GPU.

While said GPU does not provide the same compute capability as the above mentioned cloud computing solution, it does serve a good backup. Having access to the NVIDIA GeForce RTX 2060 also enables us to train a couple of models in parallel at any given time.

3.7 Experimentation

In this section, we will present all the different experiments performed. First of, model architectural experiments will be explained. We will also outline some experiments to compare accuracy on different corpus sizes and types. After that, an experiment on the number of variations generated will be described in more detail. We will also describe the general procedure of training without early stopping, and comparing the resulting accuracy to that of models trained using early stopping. Lastly, we will discuss a final experiment to determine whether the accuracy measured by using average Levenshtein accuracy may be misleading.

3.7.1 Model Architecture

We refer to model architecture as the conjunction of functions that compose our transformer-based parser. The base model for all experiments is the one described in section 3.1. In addition to the base model we have two other architectures; BPE architecture and BPE + Type checker architecture.

On a BPE Architecture, we experiment with adding a Byte Pair Encoding layer on top of the base model architecture, as input encoding. The motivation behind adding a layer of byte pair encoding is for the transformer model to be able to yield better predictions when encountering unknown words in the training corpus.

In some cases, the predicted abstract functions yielded by the transformer would not be type correct. This in turn results in the fact that GF cannot linearize the resulting abstract expression. On a type checker architecture, we experiment with using the process described earlier in Section 3.3 at the output of the transformer. This is done to make sure that the predicted abstract functions are type-correct, and that

the resulting sequence of tokens can be linearized into an abstract expression by GF.

By experimenting with the different model architectures, we seek to take the most successful model in terms of accuracy and set this to be the model for which we will carry the rest of the experiments from this point forward.

3.7.2 Corpus comparison

We will be carrying out two different experiments, to determine how the type- and size of the corpus respectively, affect the model accuracy.

3.7.2.1 Corpus type

In this experiment, we will be training four different models. The first model will be trained on the base corpus (given to us by our supervisor). For the remaining models, the corpus size will be 20,000. The second model will be trained using the variation data generation method. The third model will be trained on a corpus generated using the random data generation method. Last but not least, we will also be training a model using the combination data generation method. They will train for a total of 40 epochs, although early stopping will be used. We will then compare the accuracy of the models on a validation dataset, and the number of sentences that the models are able to correctly linearize.

3.7.2.2 Corpus size

We will in this experiment take the most successful data generation method in terms of model accuracy from the previous experiment. This data generation method will then be used to generate corpora of increasing size, and then train models for the same number of epochs as in the previous experiment, using said corpora. In a similar vain to the previous experiment, we will then compare the accuracy and the number of sentences that the models are able to correctly linearize.

3.7.3 Number of variations

This experiment consists of generating datasets of the same sizes but with different number of variations in them. As an example of this, one dataset may be generated, only generating variations to nouns in the form of synonyms. Another dataset could then be generated, generating variations to both nouns and adjectives. We will then be able to compare the effect of each variation added to the dataset in terms of the resulting model accuracy and the number of sentences that each model is able to correctly linearize.

3.7.4 Model size

Another interesting thing to compare is how the model parameters affect the resulting models. We will therefore experiment on the effect of increasing the model parameters. The parameters in question are d_{model}, d_{ff} and the number of layers in

the encoder and decoder stacks N. These parameters are the same ones as described in section 2.5.

3.7.5 No early stopping

A quite interesting experiment that we will be conducting is that of comparing models trained using the same corpus and model parameters, where one model utilizes early stopping whereas the other does not. We will be making this comparison on different corpus sizes, where corpora are generated using the variational data generation method.

3.8 Evaluation methods

To evaluate the accuracy of the model two different metrics will be tested; BLEU Score and average Levensthein accuracy. In both cases, the abstract syntax sentence yielded by the transformer is linearized to obtain its concrete representation. It is intended to compare the input sentence with the obtained concrete sentence to measure how similar the input is to the predicted output from the transformer model.

The implementation of the BLEU Score follows the equation presented in 2.12, where we will test from one to four n-gram representations and $w_n = 1/N$, proposed by Papineni et al., 2002, is being used.

To exemplify, we obtained a unigram precision of 0.6764 from the following sentence.

Prediction: we live in a careless society
Reference: we live in an accumulative society

4
Results

The purpose of this chapter is to present the results of different experiments that we carried out. First and foremost, we will be presenting the results regarding the position of the first type-correct prediction by a model. We will also be providing results in relation to how different types of model architectures affect the model accuracy. Then, we will demonstrate the results of the experiments on type- and size of corpora. Furthermore, the results of the experiments on the number of variations will be laid out. We will then show how model size affected the resulting accuracy. Moreover, the results of training models without early stopping will be showcased. Lastly, we will also lay out some information concerning whether average Levenshtein accuracy may be misleading.

In the following presentation of our results, unless *explicitly* stated, we have been using a model with the *default parameters*. When we say the *default parameters*, we refer to the parameters in Table 4.1 down below:

Transformer Model Parameters								
num_layers	d_{model}	d_{ff}	h	d_k	d_v	p_{drop}	num_epochs	patience
4	128	512	8	16	16	0.1	40	3

Table 4.1: Showcases the default model parameters used in this thesis.

To clarify, the parameter *num_ layers* refers to the number of layers in the encoder and decoder stacks, as mentioned in the discussion about the transformer architecture in Section 2.5.2. Likewise, d_{model} and d_{ff} correspond to the variables with the same names, as discussed in Section 2.5.2. Next up we have h, which refers to the number of heads in the multi-head attention sections of the transformer architecture. d_k and d_v refer again to the variables with the same names in Section 2.5.2. p_{drop} is the dropout rate, for which a value of 0.1 was used. We trained for a total of 40 epochs, and used early stopping with a patience set to 3.

4.1 Type checking of the transformer output

One of the problems we encountered multiple times was the fact that our models would not always predict type-correct tokens. To clarify, this problem is a per-token occurrence when trying to parse a concrete sentence. For example, maybe the previously predicted token was a *UseN* token. This token expects the next token to have the category N (noun). It was clear that it wasn't always the case that our models predicted as next token, a token with category N in this example.

We thus decided to carry out another experiment, in which for each token generated, we go through the output of a model, and check at which position among the d_{model} number of predictions a type-correct token appears. Figure 4.1 down below showcases the result of said experiment:

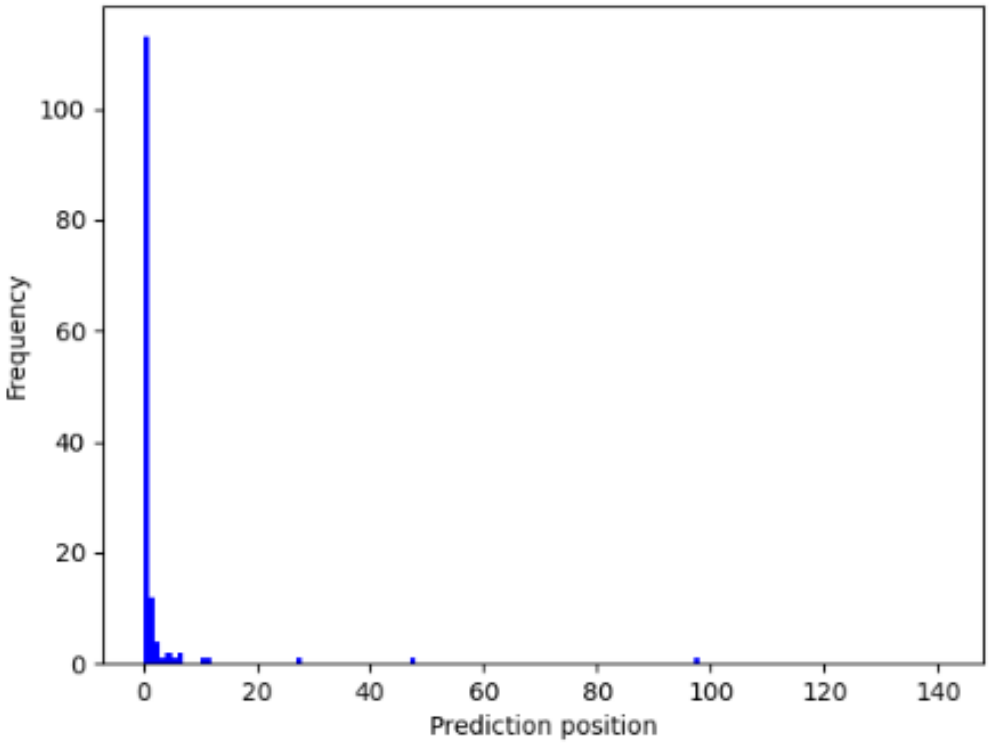


Figure 4.1: Illustrates a histogram over the first type-correct prediction position of a model output

.

4.2 Model Architecture

We trained all models with the base corpus provided to us by our supervisor, containing approximately 9,000 training examples. We tested the three models on the golden corpus containing five hundred and one unseen examples and measured both the overall accuracy of the model and the number of correctly linearized sentences each one produced.

In figure 4.2 we can observe the results from this experiment.

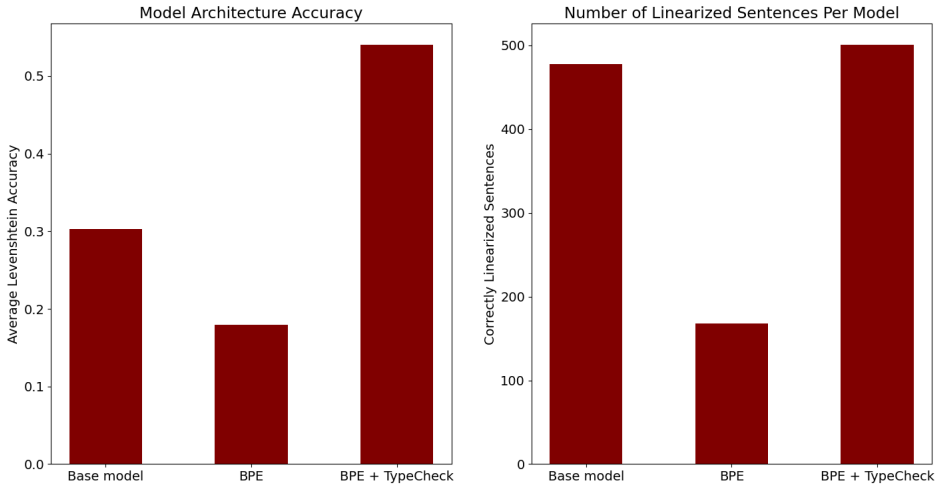


Figure 4.2: Illustrate the results between the different model architectures.

By implementing only a byte pair encoder, the yielded abstract expression tends to be type incorrect, meaning it cannot be linearized into a proper concrete representation. It can be observed how the addition of a type checker improves both the number of correctly linearized concrete sentences as well as the overall accuracy of the model.

4.3 Corpus comparison

In this section, the results of the experiments on corpus type- and size will be presented. First and foremost, we will be providing the results of the conducted experiment on corpus type. The result of said experiment was taken into account when later experimenting on corpus size, which is why we will be presenting that experiment lastly.

4.3.1 Corpus type

For this experiment, we trained four different models on different training datasets. The *default parameters* were used for all four models, and the first model was trained on the base corpus provided to us by our supervisor, containing roughly 9,200 training examples. The other three models were trained on data generated from the three different data generation methods described in Section 3.5. For those three models, a total of 20,000 training examples were generated. As discussed in Section 3.5, for all three data generation methods, the base corpus is appended to the generated data, and is therefore included in the total 20,000 training examples. Provided below in Figure 4.3 are the results of this experiment.

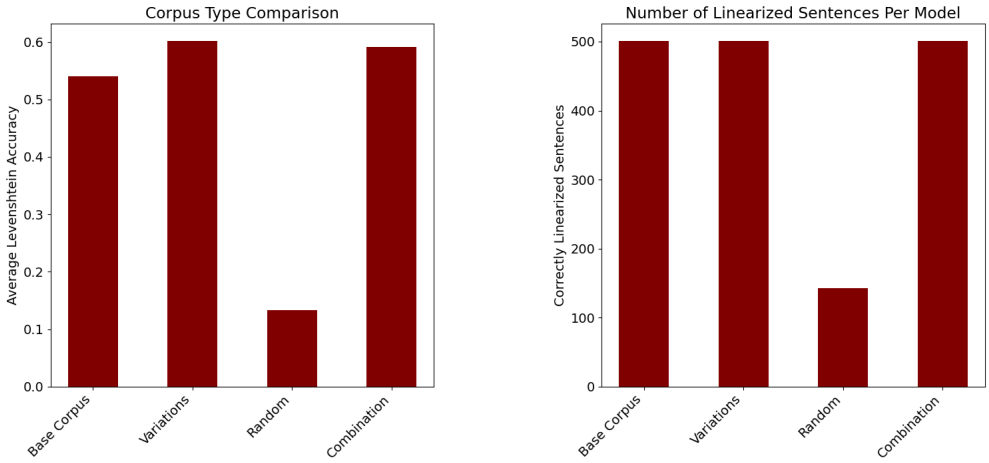


Figure 4.3: Illustrates two histogram, of which the left displays the average levenshtein accuracy, and the right displays the number of correctly linearized sentences, for the different models.

.

4.3.2 Corpus size

For this experiment, we trained different transformer models with corpora of increasing size, to see what happens to the average Levenshtein accuracy. The training data was generated using the variational data generation method described earlier in Section 3.5.2. This was done because using the variational data generation method seemed superior, judging from the results of the previous experiment.

The *default parameters* were used to train and evaluate our resulting models in terms of average Levenshtein accuracy for this experiment. Figure 4.4 down below illustrates our results in the form of a graph.

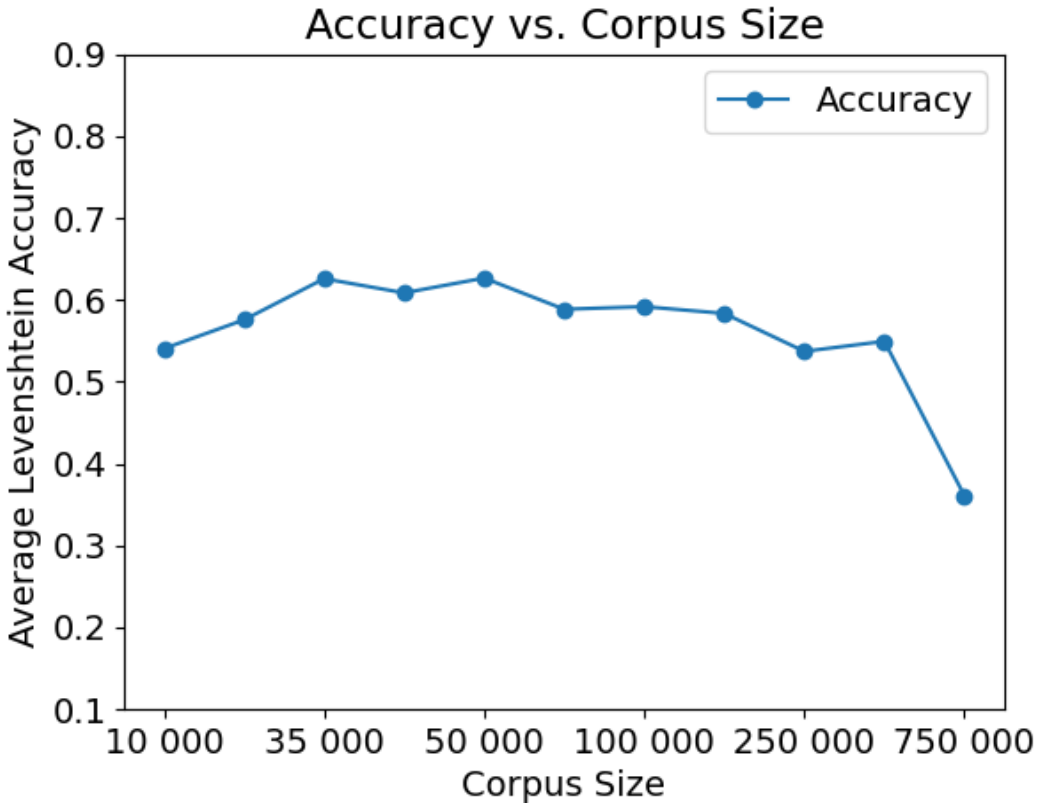


Figure 4.4: Illustrates corpus size vs. average Levenshtein accuracy.

4.4 Number of variations

Initially, we only generated variations in the form of negating a sentence, exchanging tense and pronouns, and adding an adjective to a noun. We hypothesized that perhaps introducing more variations would improve model accuracy in terms of Levenshtein accuracy. We thus generated 4 different corpora to test our hypothesis. The first corpus contained a total of 50,000 training examples, including synonyms to nouns. The second one contained again 50,000 training examples, including synonyms to both nouns and adjectives. The final two corpora were generated the same way, but containing 100,000 training examples instead of 50,000.

Much like before, we trained and evaluated or models using the *default parameters* described in the beginning of this Chapter. We provide the results of this experiment in Figures 4.5 and 4.6 down below:

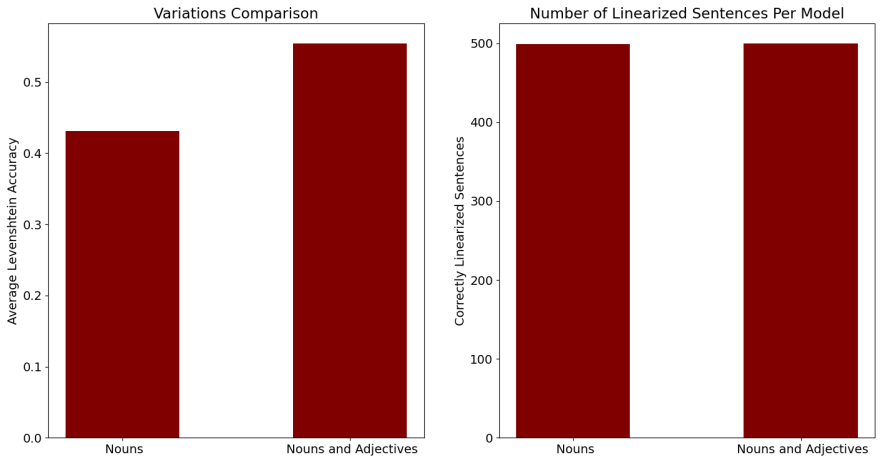


Figure 4.5: Illustrates the number of variations comparison between the 50,000 training datasets.

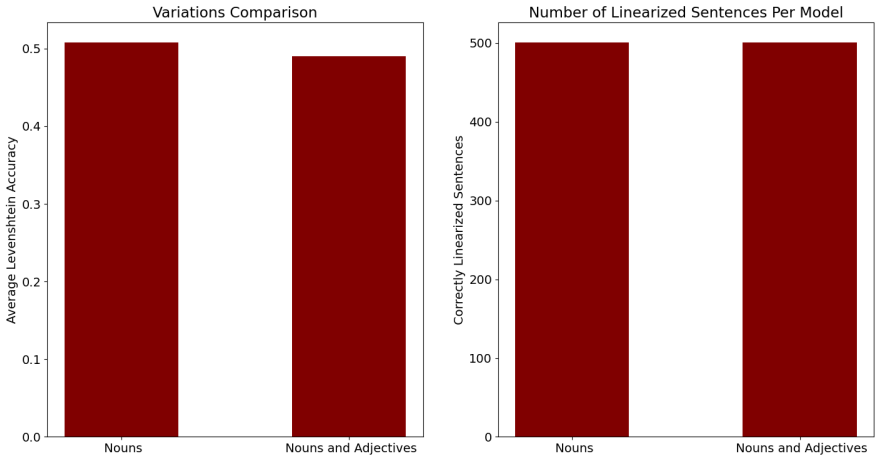


Figure 4.6: Illustrates the number of variations comparison between the 100,000 training datasets.

We then decided to add variations in the form of synonyms to verbs and adjectives as well. We trained models using the *default parameters* on corpora with sizes ranging from 35,000 to 750,000 training examples. Down below In Figure 4.7 the result of this addition to the number of variations can be found. We also provide statistics in the form of the number of correctly linearized sentences for each model, in Figure 4.8 down below:

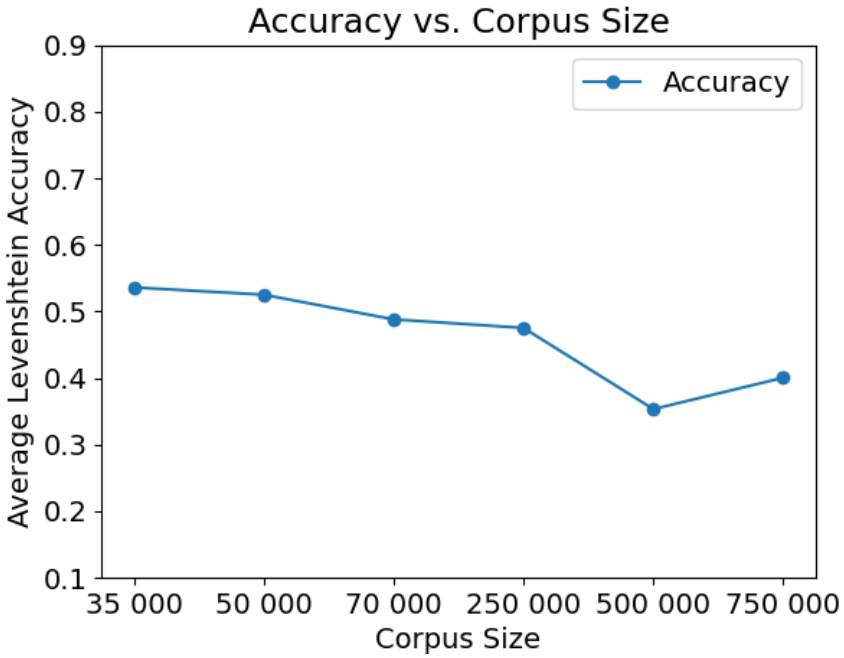


Figure 4.7: Illustrates the average Levenshtein accuracy with additional variations.

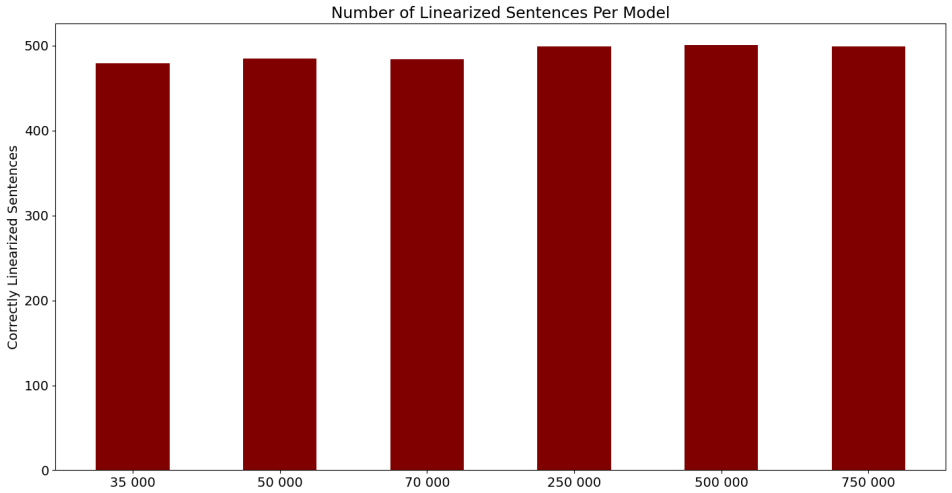


Figure 4.8: Illustrates the number of correctly linearized sentences per model.

4.5 Model size

When experimenting with how the model size would impact average Levenshtein accuracy, we began with the *default parameters*. We then set the number of layers in the encoder and decoder stacks N to 5, and made additive 5% increases to d_{model} and d_{ff} for each iteration of this experiment. To make it more explicit, Table 4.2 down below displays the new parameters for the first iteration of this experiment:

5% Increased Model Parameters								
num_layers	d_{model}	d_{ff}	h	d_k	d_v	p_{drop}	num_epochs	patience
5	135	538	8	16	16	0.1	40	3

Table 4.2: Showcases the 5% model parameter increase.

For this iteration of the experiment, we trained three models with increasing corpus sizes. The corpora were generated using the variational data generation method. Figure 4.9 puts on display the results of this iteration of the experiment:

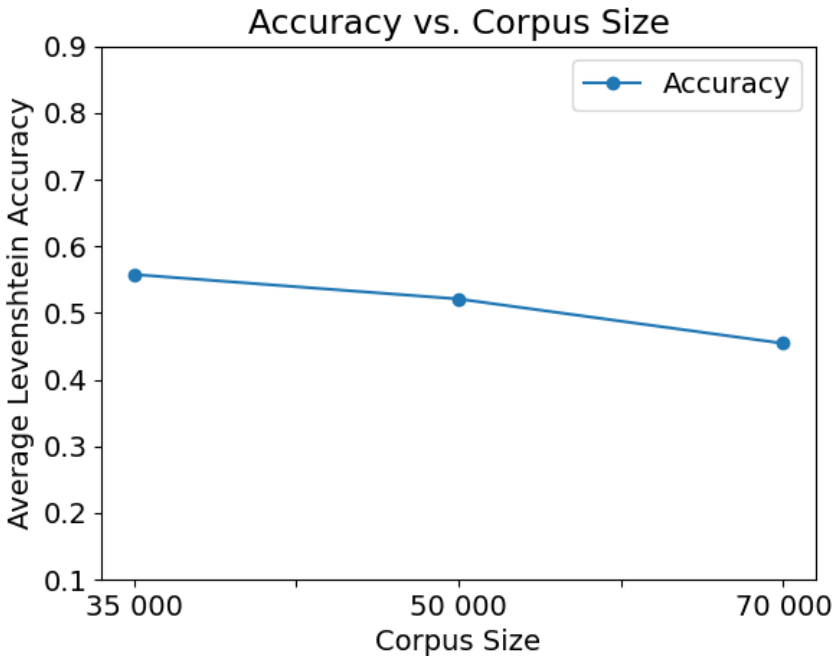


Figure 4.9: Illustrates the average Levenshtein accuracy with a 5% model parameter increase.

We then made another 5% additive model parameter size increase, yielding the model parameters found in Table 4.3 down below:

10% Increased Model Parameters								
num_layers	d_{model}	d_{ff}	h	d_k	d_v	p_{drop}	num_epochs	patience
5	141	563	8	16	16	0.1	40	3

Table 4.3: Showcases the 10% model parameter size increase.

Much like with the previous iteration of this experiment, we again trained three models with increasing corpus sizes. The corpora were generated using the variational data generation method. Figure 4.10 showcases the results of this iteration of the experiment:

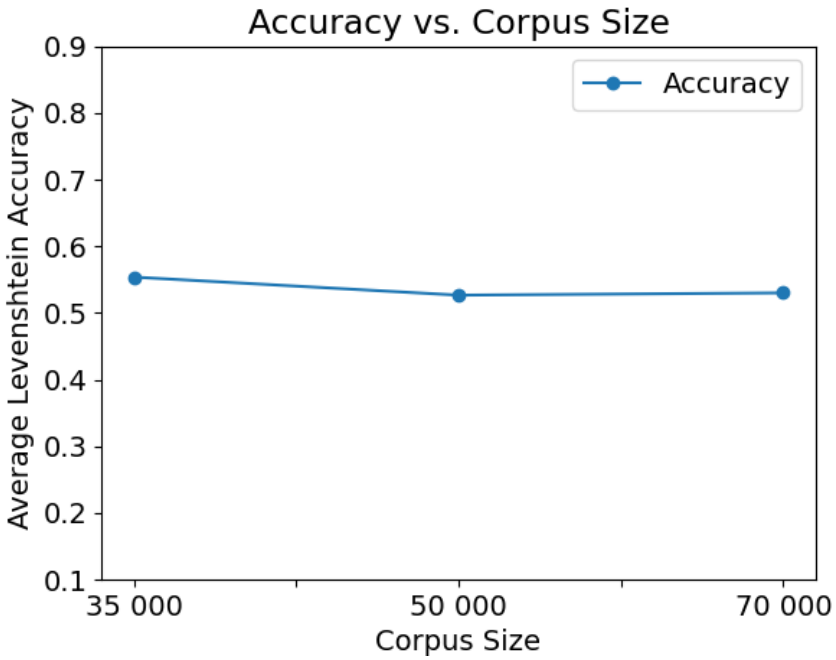


Figure 4.10: Illustrates the average Levenshtein accuracy with a 10% model parameter increase.

For the final iteration of the experiment, we made another 5% additive model parameter size increase, resulting in the model parameters found in Table 4.4 down below:

15% Increased Model Parameters								
num_layers	d_{model}	d_{ff}	h	d_k	d_v	p_{drop}	num_epochs	patience
5	147	589	8	16	16	0.1	40	3

Table 4.4: Showcases the 15% model parameter size increase.

Unlike with the previous iterations of this experiment, we decided to experiment with corpus sizes beyond that of 70,000. We hypothesized that perhaps the resulting models would learn better when having greater model parameter sizes, in combination with larger corpora. We therefore generated corpora using the variational data generation method, ranging from 50,000 to 750,000 training examples. The results of this final iteration of this experiment can be found in Figure 4.11 down below:

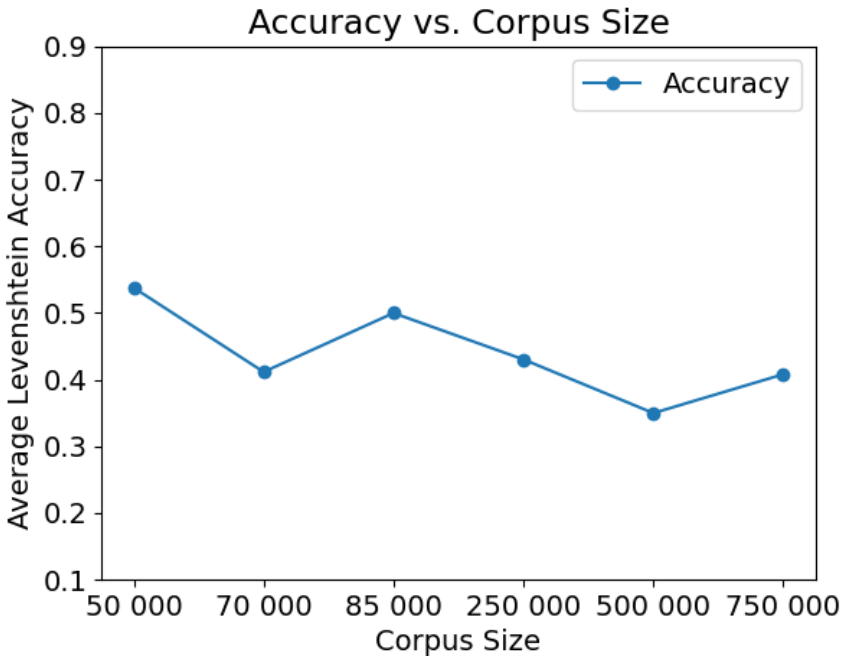


Figure 4.11: Illustrates the average Levenshtein accuracy with a 15% model parameter increase.

4.6 No early stopping

For this experiments, models were trained using the *default parameters*, but without any early stopping. Comparisons between this experiment and the experiment on corpus size can then be drawn. This is because the models from both experiments were trained on roughly the same corpora sizes. For this experiment in particular, models were trained on corpora with sizes ranging from 35,000 to 300,000 training examples. Provided in Figure 4.12 down below are the results of this experiment:

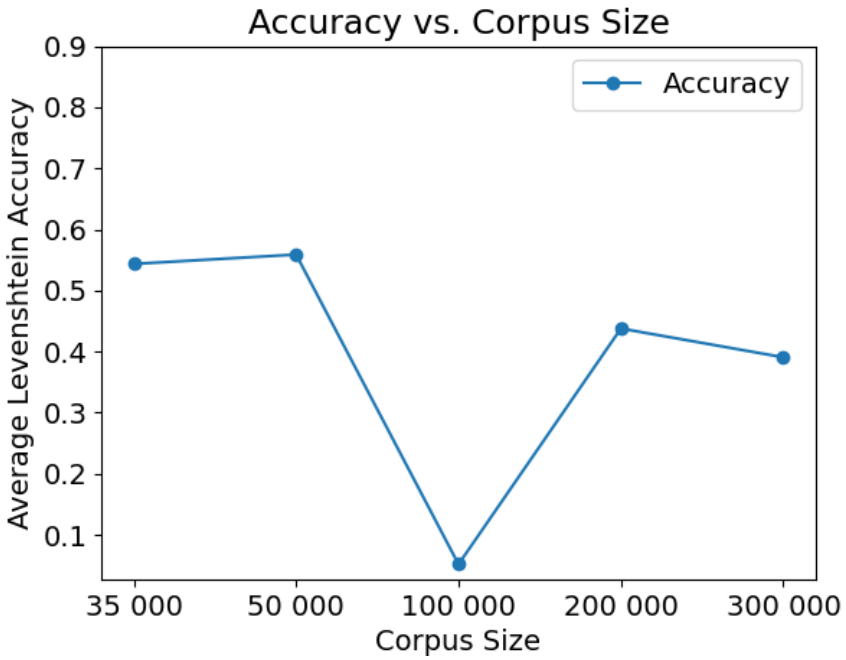


Figure 4.12: Illustrates the average Levenshtein accuracy without early stopping.

4.7 BLEU Score and Average Levenshtein Accuracy

Typical implementations of BLEU Score rely on having more than one candidate translation, to capture semantics.

Translation: men are able to walk on two feet
Candidate 1: human beings are able to walk on two feet
Candidate 2: man can walk on their feet
Candidate 3: humans have the ability to walk on two feet

As in the example above, BLEU benefits from the meaning of different candidates to be able to produce a score that entails semantics.

The training and testing corpora lack reference sentences, which made BLEU score not a viable implementation. At the same time, the average length of the sentences is not able to accommodate more than unigram or biagram comparisons in most cases.

In the following example, we can observe the modified precision for the unigram, bigram, trigram and tetragram for the prediction-reference pair taken from the testing corpus.

Prediction: a black star turns matter alleys apparently differently
Reference: A black star absorbs all matter
0.5, 0.37796, 0.2876, $4.797e^{-78}$

Plugin such low values, as in the tetragram, will render the full BLUE score to approximate a value of zero. In many scenarios, the formula presents a division error. Therefore having one-to-one comparisons using the BLUE score as an evaluation metric will, in many cases, through a computation error due to the low n-gram match between sentences.

Due to this reason, it was chosen to rely solely on the average Levenshtein accuracy. Due to its easy nature, and character-level comparison, we can obtain an accuracy value for every entry in the corpus.

5
Discussion

We will in this chapter discuss the results presented in Chapter 4. First of all, we will be discussing the position of the first type-correct prediction by a model for each generated token. We will also be discussing how different types of model architectures affect the model accuracy. Then, the corpora used to train the models will be discussed. In particular, we will mention the results of the experiments on type- and size of corpora. Furthermore, the results of the experiments on the number of variations will be elaborated upon. We will then discuss the results regarding how model size affected the resulting accuracy. Moreover, we will discuss the results of training models without early stopping. It will also be discussed whether average Levenshtein accuracy as an accuracy measurement may be misleading. Lastly, we would like to illuminate the fact that more training data does not necessarily mean better accuracy, as was seen in the results presented in Chapter 4.

5.1 Type checking of the transformer output

We see in Figure 4.1 that most type-correct tokens lie within the first 4 or so predictions of the model. A smaller amount of type-correct tokens can be found further down the line, in the form of outliers. It seems to be the case that the first type-correct token always lies within the first 100 predictions. Ideally, the first type-correct token should always be the first prediction though. This does not always seem to be the case. Thus, the result of this experiment further validates our choice to algorithmically select type-correct tokens during inference.

5.2 Model Architecture

It can be observed in figure 4.2 how BPE harmed the production of type-correct predictions, and in extension also the accuracy of trained models. By adding a type checker, it is ensured that resulting sequences of abstract function names can be turned into abstract expressions and later linearized into concrete representations. However, it is also important to note that the number of correctly linearized sentences does not always mean an improvement in accuracy.

In the example below, it can be observed that a sentence that was parsed type-correctly deviates from the original input when linearized back into English:

Type correct prediction: she could leave her programme
Reference: she was able to program her computer

From the example, we can observe how the transformer yields a sentence that contains similar words to the input, but semantically they differ from each other. Despite this, we decided to continue using the type checker model architecture. It allows for the most correctly produced sentences, whether they are accurate or not.

5.3 Corpus comparison

In this section, we will discuss the results of the experiments on corpus type- and size. We will begin by discussing the results of the experiment on corpus type, and lastly we will be discussing the results of the experiment on corpus size. The latter mentioned experiment was based on the results of the former, which is why we discuss the experiments in said order.

5.3.1 Corpus type

As can be seen in the histogram to the left in Figure 4.3, the type of corpus to achieve the highest accuracy in terms of average Levenshtein accuracy was the dataset generated using the variational data generation method. It even surpassed that of using just the base corpus. Training on randomly generated data reduced the accuracy drastically. Judging from the resulting histogram, introducing randomness brought down the accuracy of the combination data generation method. This can be seen by comparing the accuracy to that of the achieved accuracy using the variation data generation method.

In terms of the number of correctly linearized sentences, models trained on all corpus types except for randomly generated data are able to correctly linearize all sentences. The model trained on randomly generated data was only able to linearize 143 sentences out of 501.

These results indicate that abandoning randomly generated data, and thus also the combination data generation method, is a smart move. Randomly generated data brought down the overall accuracy, both in terms of average Levenshtein accuracy, and in the number of correctly linearized sentences. To strengthen this claim, we decided to plot the attention weights produced by parsing. We randomly selected and ran the sentence from the base corpus "Add insult to injury", through the transformer. The transformer output was "doubt cuts in injury to injuries", which somewhat resembles the original sentence.

However, we see that the model hallucinates quite a bit. Also, most of the time the output produced by the model could not be linearized by GF. This is because the model is not often able to produce type-correct predictions. It seems the different heads have learnt to attend to the same/similar things within the context of a sentence. Thus, the advantage of multi-head attention is not properly leveraged by the model.

Due to implementation shortcomings, it is only possible to inspect the attention scores from the last decoder layer of the transformer model. It would help to further strengthen our argument, had we made it possible to inspect other layers of the transformer model for attention as well. Despite this, we provide two different attention plots that enables the comparison of the random- and the variation data generation methods. On the y-axes, the in-tokens can be seen (before byte pair-

and positional encoding). On the x-axes, the transformer output is shown.
Looking at Figure 5.1 down below, we see that the attention patterns for most if not all of the heads are very similar in nature. For example, there is this prevalent vertical straight line in most of the heads, going from the top left corner down to the middle.

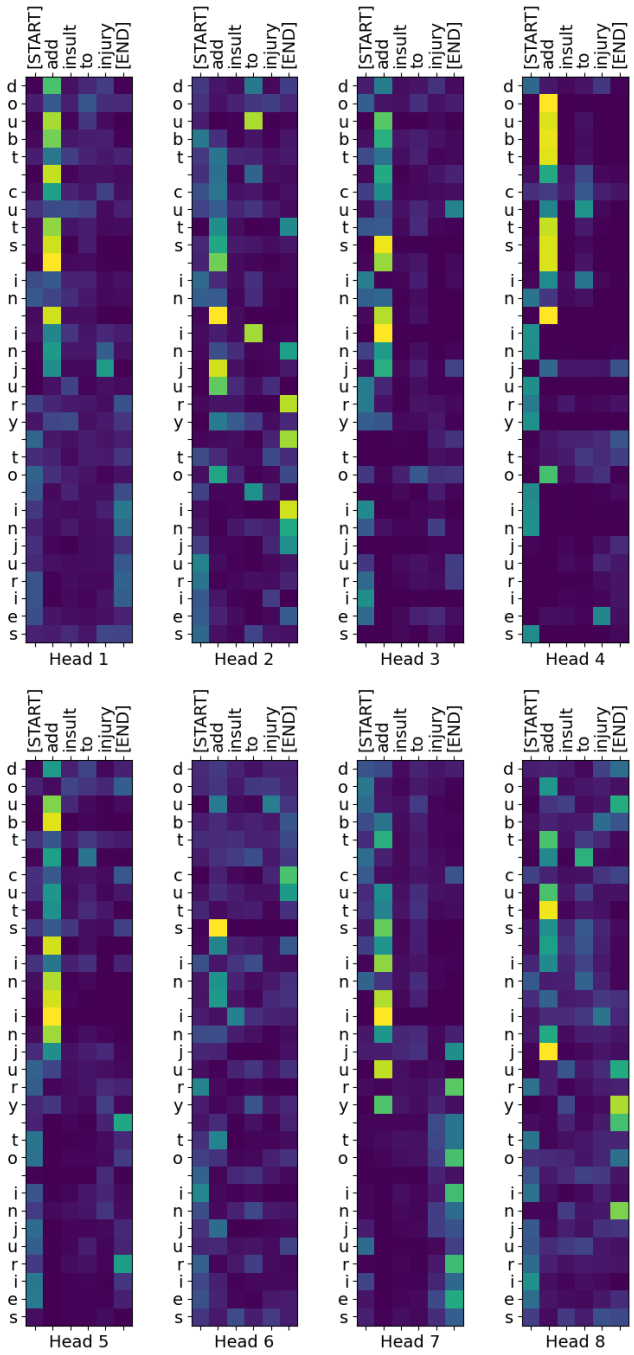


Figure 5.1: The attention weights produced by parsing the sentence "Add insult to injury" using a model trained on 20,000 randomly generated training examples

As a comparison, Figure 5.2 features the attention weights of the model trained on 20,000 training examples generated using the variation data generation method. The model wasn't able to produce any output for the sentence "Add insult to injury", so another sentence was instead selected. Despite this, it can be seen that the attention weights for the different heads differ a lot more in comparison. This means that the heads are in fact attending to a larger pool of information than that of the model trained on random data.

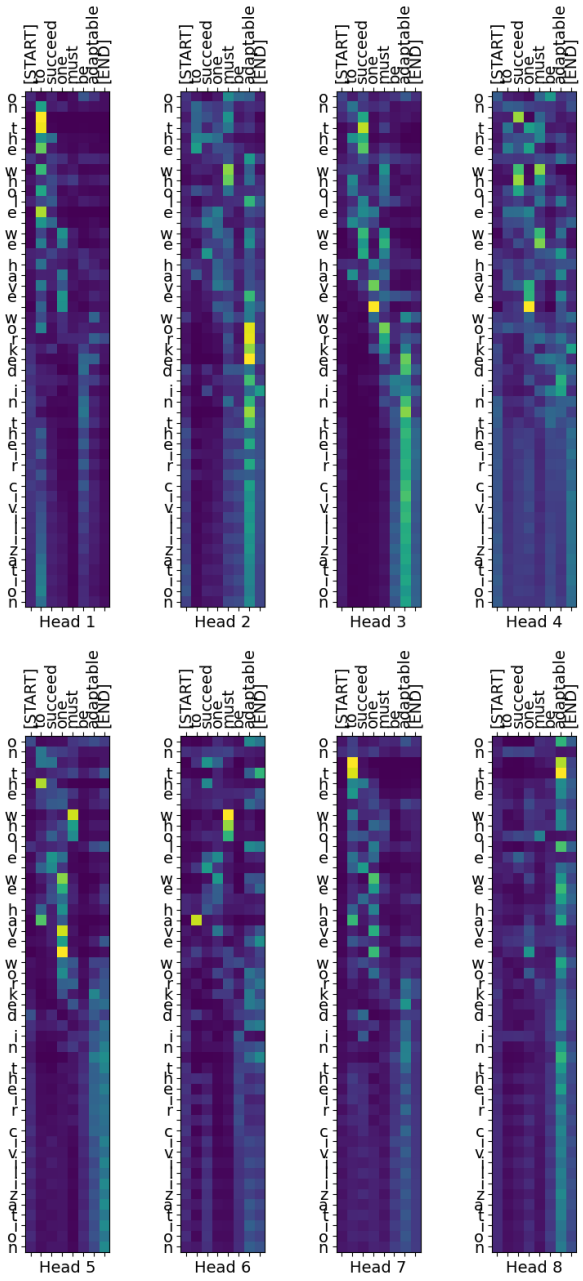


Figure 5.2: Illustrates the attention weights produced by parsing the sentence "to succeed one must be adaptable" using a model trained on 20,000 variationally generated training examples

5.3.2 Corpus size

What can be seen in Figure 4.4 is that achieved average Levenshtein accuracy peaks for the model trained on a corpus size of around 50.000, with a value of 62.66%. We hypothesized that perhaps a higher average Levenshtein accuracy cannot be achieved with the *default parameters* seen in Table 4.1. Thus, we conducted the same experiment for increasing corpus sizes beyond that of 50.000, to confirm our hypothesis. When training models with datasets containing more than 50.000 training examples, the average Levenshtein accuracy happens to be on a downhill trend.

Even though we use a different measure of accuracy, our results seem to be in accordance with the results presented by Kolachina and Ranta (2019), up to a certain point. They measure accuracy in terms of LAS (see Section 2.7.3), with corpora containing a maximum of 20,000 training examples. The observed accuracy flattens out at a certain point, which also happened for us until it eventually dropped again.

5.4 Number of variations

Looking at the left histogram in Figure 4.5, it can clearly be seen that increasing the number of variations had a positive effect on the resulting average Levenshtein accuracy. Including only synonyms to nouns in the training dataset, an accuracy of 43.12% was achieved. Compare that to also including synonyms to adjectives, which resulted in an accuracy of 55,45%.

In terms of the number of correctly linearized sentences, both models performed remarkably well in this aspect. The model trained on a corpus containing synonyms only to nouns was able to correctly linearize 499 out of 501 sentences. The model trained on a corpus containing synonyms to both nouns and adjectives was able to correctly linearize 500 out of 501 sentences.

Turning our attention to the left histogram in Figure 4.6, it can be seen that increasing the number of training examples from 50,000 to 100,000 had an effect on the resulting average Levenshtein accuracy. In this case, the model trained on a corpus containing synonyms only to nouns achieved an accuracy higher than that of the model trained on a corpus containing synonyms to both nouns and adjectives (50,81% vs. 49,02%). The difference is very small, but what can be observed is the fact that the average Levenshtein accuracy levels out in accordance with the number of training examples.

A positive note is that both models were able to correctly linearize 501 out of 501 sentences. Thus, it seems that increasing the number of training examples allows the model to more easily produce type-correct sentences at the cost of potentially choosing the wrong words. If this is the case, then that would explain why the average Levenshtein accuracy goes down (there are more synonyms for each noun and adjective that the model may have learnt, and has to choose from).

Adding variations to verbs and adverbs in the corpus, in addition to that of nouns and adjectives, resulted in a model scoring the initially good average Levenshtein accuracy of 53,61%. This can be seen in Figure 4.7. Unfortunately, the accuracy is on a downhill trend from there. Interestingly enough, if we take a look at the model trained on the dataset containing 750,000 training examples, the accuracy seems to be on a rise again.

As observed previously, and perhaps even expected at this point, the number of correctly linearized sentences goes up in accordance with the corpus size. The model trained on the dataset containing 35,000 training examples was only able to correctly linearize 479 out of 501 sentences. The model trained on the dataset containing 750,000 training examples was however able to correctly linearize all of the 501 sentences.

5.5 Model size

During the first iteration of the experiment of increasing the model parameter size (5% increase), an initial 55,77% average Levenshtein accuracy was achieved on a model trained on a training dataset containing 35,000 examples. The accuracy then went down in proportion to the increasing dataset size.

For the second iteration (10% model parameter size increase) of the experiment, it started out in a similar vein. An initial accuracy of 55,38% was achieved on a model trained on a training dataset containing 35,000 examples. However, there is now an observable increase between the models trained on datasets containing 50,000 and 70,000 respectively. The accuracy went up from 52,67% to 53,01%. This sparked some amount of interest, which is partly why we in the last iteration of this experiment decided to experiment with training datasets the sizes of which extend beyond that of 70,000 examples.

The final iteration (15% model parameter size increase) went much the same as with the previous iterations of the experiment. An initial accuracy of 53,78% was achieved on a model trained on a dataset containing 50,000 examples (for some reason we found it unneccesary to train a model on a dataset containing 35,000 training examples). An interesting observation is that when looking at Figure 4.11, the achieved accuracy first goes down, before it hits a local maximum. The accuracy then goes down for models trained beyond that of 85,000 training examples. There seems to be an uphill trend from 500,000 to 750,000 training examples though. Thus, future work could include training and evaluating models on training datasets the sizes of which exceed a million examples.

5.6 No early stopping

The results of this experiment in terms of average Levenshtein accuracy look rather random. In Figure 4.12 in Chapter 4, it can be seen that the average accuracy goes on an uphill trend between the model trained on a dataset containing 35,000 training examples, and the model trained on a dataset containing 50,000 training examples. Thus far, this is in accordance with the results of our initial experiments on corpus size, with a small difference of 0,3 percentile units.

For the model trained on a dataset containing 100,000 training examples, the average Levenshtein accuracy actually went down to 5,17%, which is a really bad result. This is however what can happen when validation error is not measured during training. This model seems to have hit or approached some local maximum when it comes to the validation loss, resulting in a very poorly performing model. This situation could have been avoided if early stopping had been put into use, which justifies our use of the method.

For the remaining two models, namely the ones trained on datasets containing 200,000 and 300,000 training examples respectively, the accuracy seems to again be on a downhill trend. For these models, it does not seem like the validation loss hit or approached some local maximum. Rather, it seems like the validation loss was relatively low at the end of the 40 training epochs. The average Levenshtein accuracies achieved look a lot like the achieved accuracies for models trained on corpora of similar sizes from previous experiments.

5.7 Data generation

As was prevalent throughout most of our experiments, more data generally does not mean greater accuracy. We often saw a peak accuracy of models trained on datasets containing around 50,000 training examples. Something that would be interesting in terms of future work, would be to figure out if going beyond the million mark when it comes to the amount of training examples has any impact. We saw for example in Figure 4.11 that accuracy seemed to be on an uphill trend again after surpassing 500,000 training examples.

Another thing that is important to take into account when it comes to the variation data generation method, is to not make certain sentences over-represented in the training dataset. For example, we had quite a lot of few word sentences present in the base corpus. For these sentences in particular, not many variations can be had. In comparison, we also have longer sentences present, of which many variations can be generated. This creates an imbalance in the resulting dataset, where variations of longer sentences can be over-represented. This in turn makes the models trained on said datasets overfit on certain sentences.

Perhaps creating a bigger base corpus would solve this problem of over-representation. However, for now, our workaround has been to limit the amount of variations for each sentence in the base corpus. This limit was made possible to manually set during data generation.

5.8 average Levenshtein accuracy

By working with the Levensthein distance we are measuring our accuracy to a character level. The more similar the input and output strings are, the better accuracy we will obtain, however, this evaluation method comes with the cost of not taking into consideration the semantics of the sentences.

This can cause the evaluations to be misleading. Consider the following example.

Prediction: I advise a copy of my lawyer's letter
Reference: I adjoin a copy of my lawyer's letter
Accuracy: 0.8918

From this comparison, we obtained an accuracy of 89.18% based on the character-level similarity between the two sentences. However, by changing the word adjoin to advise, it is clear that the whole sentence takes a different meaning. Therefore, it can be argued that the quality of the prediction is lower than the number reflected by the average Levenshtein accuracy.

On the other hand, we can have the opposite effect.

Prediction: we could buy a car
Reference: we were at least able to buy one car
Accuracy: 0.3611

From the example above, it can be argued that an accuracy of 36.11% can be misleading since the meaning of both sentences is similar and the model can produce an accurate prediction using a different set of words.

The lack of a reference sentence corpus made the BLEU Score not a viable implementation, which otherwise would have been able to accommodate the semantics of the sentences on the evaluation. Therefore, future work would benefit from accommodating BLEU score, or any other metric, that could measure the quality of the prediction based on semantics.

6
Conclusion

The goal of this thesis has been to develop a transformer-based parser for Grammatical Framework. Specifically, the goal of the parser was to, given a concrete sentence, predict an abstract expression that when linearized resembles the original sentence as closely as possible. To attempt to achieve this goal, a series of experiments were conducted.

Our first experiment was to see how byte pair encoding would affect the accuracy of a model compared to that of a model trained on the same dataset without byte pair encoding. We also experimented on aiding with type correct selection of tokens in combination with byte pair encoding, and observed an increase in accuracy with said combination.

Another experiment was performed to compare different training corpora. We conducted one experiment to determine what kind of data generation would yield the best results. It turned out that the variational data generation method was the best one, and so this method was used in the following experiments. The second experiment in relation to comparison of different training corpora was one in which we compared the average Levenshtein accuracy on models trained on copora of increasing sizes. There was a clear peak in accuracy around training datasets containing 50,000 training examples.

Something interesting that we investigated was how the number of variations for each sentence would affect the average Levenshtein accuracy. We initially trained two models, one in which the training dataset contained synonyms to nouns. The other model was trained on a training dataset containing synonyms to both nouns and adjectives. The addition of variations to adjectives resulted in an improvement of accuracy in terms of average Levenshtein accuracy. Both datasets contained 50,000 training examples, and so we were also interested in seeing what would happen if the dataset size increased to 100,000 training examples. This second iteration of the experiment instead yielded worse results for the model trained on a dataset containing synonyms to both nouns and adjectives. Meanwhile, the accuracy for the model trained on a dataset containing synonyms to just nouns instead increased. A final iteration of the experiment was performed in which variations in the form of synonyms to verbs and adverbs were also added (on top of synonyms to nouns and adjectives). A general downhill trend was observed in terms of accuracy, until the dataset size went beyond 500,000. At this point, an uphill trend was instead observed.

The second to last experiment conducted was one in which we increased the number of encoder and decoder layers to 5, and then for each iteration of the experiment we made an addition of 5% to d_{model} and d_{ff}, starting from the first iteration. In each iteration, we then trained models on increasing corpus sizes to see what happened to the accuracy in terms of average Levenshtein accuracy. There was an observed downhill trend for all iterations of this experiment. However, for the third and final iteration of the experiment, we decided to train on bigger and bigger corpora, and saw an uphill trend again on models trained on corpora the sizes of which extend beyond that of 500,000 training examples.

We wanted to justify our use of early stopping, and therefore we conducted an experiment in which models were trained without early stopping, on corpora of increasing sizes. The observed results were very much so expected. The average Levenshtein accuracy after 40 training epochs was very unstable. For one model in particular, training on a corpus containing 100,000 training examples, an average Levenshtein accuracy of 5,17% was achieved. This was very poor results indeed, but again, to be expected. For other models, however, the accuracy was closer to what was achieved with early stopping in place.

Semantics play an important role in the evaluation of a parser, and while character-level comparison was an overall good solution to measure the accuracy of our model, it also proved that relying solely on average Levenshtein accuracy could be misleading at times.

A general notion regarding data generation is the fact that while variational data generation proved to be the better option, oftentimes what we observed was still mediocre average Levenshtein accuracies. We believe the base corpus needs an expansion to provide a more diverse collection of training sentences from which variations can be generated.

There are a lot of things that can be improved, should this project be continued in the future. One thing in particular is having access to better hardware. With more computationally capable hardware in place, bigger models could be trained, they could be trained on bigger corpora, and they could finish training in a reasonable amount of time.

Another thing that could be done is experimenting with the special tree-positional encoding presented in Section 2.8.2. It would be interesting to see how utilizing the special tree-positional encoding would impact model performance.

Yet another enhancement would be to train the model to select type correct tokens already during training, as opposed to filtering out type-incorrect predictions during inference. For each training example, and for each token in a training example, one could for example provide the model with the expected type of the next token. Then the type-incorrect predictions could be filtered out already

during training, and this way the model would learn faster to predict type-correct tokens. The described procedure above would likely make training a bit more time consuming though, which again strengthens the need for better hardware.

Bibliography

[1] Sameh AlAnsary. "Interlingua-based machine translation systems: UNL versus other interlinguas". In: *The Egyptian Journal of Language Engineering* 1.1 (2014), pp. 42–54.

[2] Krasimir Angelov. "Incremental parsing with parallel multiple context-free grammars". In: *Proceedings of the 12th conference of the European chapter of the ACL (EACL 2009)*. 2009, pp. 69–76.

[3] Krasimir Angelov. *pgf PyPI*. Jan. 2023. URL: https://pypi.org/project/pgf/2.1/.

[4] Krasimir Angelov. *The mechanics of the Grammatical Framework*. Chalmers Tekniska Högskola (Sweden), 2011.

[5] Krasimir Angelov and Peter Ljunglöf. "Fast statistical parsing with parallel multiple context-free grammars". In: *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*. 2014, pp. 368–376.

[6] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML].

[7] Marie-Catherine De Marneffe and Joakim Nivre. "Dependency grammar". In: *Annual Review of Linguistics* 5 (2019), pp. 197–218.

[8] Christiane Fellbaum. *WordNet: An electronic lexical database*. MIT press, 1998.

[9] Andrew Finch, Young-Sook Hwang, and Eiichiro Sumita. "Using machine translation evaluation techniques to determine sentence-level semantic equivalence". In: *Proceedings of the third international workshop on paraphrasing (IWP2005)*. 2005.

[10] Philip Gage. "A new algorithm for data compression". In: *C Users J.* 12.2 (Feb. 1994), pp. 23–38. ISSN: 0898-9788.

[11] Jonas Gehring et al. "Convolutional sequence to sequence learning". In: *International conference on machine learning*. PMLR. 2017, pp. 1243–1252.

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

[13] Theo Janssen. "Foundations and applications of Montague grammar". PhD thesis. University of Amsterdam, 2014.

[14] D. Jurafsky et al. *Speech and Language Processing*. Pearson Education, 2014. ISBN: 9780133252934. URL: https://books.google.com.mx/books?id=Cq2gBwAAQBAJ.

[15] Konstantinos Kogkalidis, Jean-Philippe Bernardy, and Vikas Garg. *Algebraic Positional Encodings*. 2023. arXiv: 2312.16045 [cs.LG]. URL: https://arxiv.org/abs/2312.16045.

[16] Prasanth Kolachina and Aarne Ranta. "Bootstrapping UD treebanks for delexicalized parsing". In: *Proceedings of the 22nd Nordic Conference on Computational Linguistics*. 2019, pp. 15–24.

[17] Vladimir I. Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals". In: *Soviet physics. Doklady* 10 (1965), pp. 707–710. URL: https://api.semanticscholar.org/CorpusID:60827152.

[18] J Łukasiewicz. *Comments on Nicod's axiom and on" generalizing deduction"(1931). Reprinted in: Łukasiewicz, J.* 1970.

[19] Sabrina J. Mielke et al. *Between words and characters: A Brief History of Open-Vocabulary Modeling and Tokenization in NLP*. 2021. arXiv: 2112.10508 [cs.CL].

[20] George A Miller. "WordNet: a lexical database for English". In: *Communications of the ACM* 38.11 (1995), pp. 39–41.

[21] Sujatha Mudadla. *Layer normalization*. Dec. 2023. URL: https://medium.com/@sujathamudadla1213/layer-normalization-48ee115a14a4.

[22] Kishore Papineni et al. "Bleu: a method for automatic evaluation of machine translation". In: *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 2002, pp. 311–318.

[23] Hunter Phillips. *A Simple Introduction to Softmax. Softmax normalizes an input vector...* May 2023. URL: https://medium.com/@hunter-j-phillips/a-simple-introduction-to-softmax-287712d69bac.

[24] Lutz Prechelt. "Early stopping-but when?" In: *Neural Networks: Tricks of the trade*. Springer, 2002, pp. 55–69.

[25] Aarne Ranta. *Grammatical framework: Programming with multilingual grammars*. Vol. 173. CSLI Publications, Center for the Study of Language and Information Stanford, 2011.

[26] David E Rumelhart, James L McClelland, PDP Research Group, et al. *Parallel distributed processing, volume 1: Explorations in the microstructure of cognition: Foundations*. The MIT press, 1986.

[27] Karin Kipper Schuler. *VerbNet: A broad-coverage, comprehensive verb lexicon*. University of Pennsylvania, 2005.

[28] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[29] tensorflow.org. Apr. 2024. URL: https://www.tensorflow.org/api_docs/python/tf/keras/layers/ReLU.

[30] Ashish Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[31] Harsh Yadav. *Dropout in neural networks*. May 2023. URL: https://towardsdatascience.com/dropout-in-neural-networks-47a162d621d9.

[32] Jinming Zou, Yi Han, and Sung-Sau So. “Overview of artificial neural networks”. In: *Artificial neural networks: methods and applications* (2009), pp. 14–22.

A

Tree flattening

In this appendix, we provide the code-listing for the procedure for tree flattening, as outlined in Section 3.2.1:

```
import pgf

def flatten(expression, padding):
    functions = ["[START]"]

    def flatten(expression, arity):
        if type(expression) == pgf.ExprApp:
            flatten(expression.fun, arity + 1)
            flatten(expression.arg, 0)
        elif type(expression) == pgf.ExprLit:
            functions.append(str(expression.val))
        else:
            functions.append(expression.name)

    flatten(expression, 0)
    functions.append("[END]")

    while len(functions) < padding:
        functions.append("[PAD]")

    return functions
```

B

Tree building

In this appendix, we provide the code-listing for the procedure for building trees, as outlined in Section 3.2.2:

```
import pgf

grammar = pgf.readPGF("ParseEng.pgf")

def convert_tokens_to_abstract(*, tokens):
    class TokensToAbstractConverter:
        def __init__(self) -> None:
            super().__init__()

        def __fetch_next_token(self):
            token = tokens[self.token_index]
            self.token_index += 1

            return token

        def __extract_abstract_expression(self) -> pgf.Expr:
            token = self.__fetch_next_token()

            function_arity = 0
            if not str(token).isnumeric():
                # Error occurs when trying to
                # get the function type of an integer,
                # just continue.
                function_type = grammar.functionType(token)
                function_arity = len(function_type.hypos)

            tree = pgf.ExprFun(token)

            for _ in range(function_arity):
                argument = self.__extract_abstract_expression()
                tree = pgf.ExprApp(tree, argument)

            return tree
```

```
            def __call__(self, *, tokens) -> str:
                self.tokens = tokens
                self.token_index = 0

                abstract_expression =
                    self.__extract_abstract_expression()
                return abstract_expression

        tokens_to_abstract_converter = TokensToAbstractConverter()
        return tokens_to_abstract_converter(tokens = tokens)
```

The resulting abstract expression can then easily be converted into English using the following code:

```
    import pgf

    grammar = pgf.readPGF("ParseEng.pgf")
    english = grammar.languages["ParseEng"]

    def linearize_transformer_output(*, tokens):
        abstract_expression = convert_tokens_to_abstract(
            tokens = tokens
        )
        return english.linearize(abstract_expression)
```