


UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

CS/IT Honours Project Final Paper 2022

Title: Meaning Representation Parsing: The Edge Prediction Component of a Semantic Graph Parser

Author: Claudia Greenberg

Project Abbreviation: MRP

Supervisor(s): Dr Jan Buys

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	0
Theoretical Analysis	*0*	*25*	10
Experiment Design and Execution	*0*	*20*	15
System Development and Implementation	*0*	*20*	10
Results, Findings and Conclusions	*10*	*20*	10
Aim Formulation and Background Work	*10*	*15*	15
Quality of Paper Writing and Presentation	*10*		10
Quality of Deliverables	*10*		10
Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*)	*0*	*10*	0
Total marks		**80**	**80**

Meaning Representation Parsing: The Edge Prediction Component of a Semantic Graph Parser

Final Paper

Computer Science Honours Project 2022

Claudia Greenberg
grncla009@myuct.ac.za
Department of Computer Science
University of Cape Town
South Africa

ABSTRACT

Semantic Graph Parsing is a highly-researched and highly-demanded field of Natural Language Processing. Its usage reaches to other fields such as Artificial Intelligence. There have been recent developments within the area that changes how accurate and efficient a parser can become, particularly the introduction of Transformer Neural Networks. This research Honours Project Paper presents additional research into the effectiveness of these Transformers in the development of the Edge Prediction Component of a Semantic Graph Parser. Additionally, it delves briefly into the effectiveness of using Maximum Entropy Loss as a means of training a model. Limited results are provided by attempting to compare pretrained BERT Transformers, as opposed to non-trained LSTMs. Furthermore, limited results are presented comparing Maximum Entropy Loss to Maximum Margin Loss. However, no official conclusions may be presented. Given the paper's limitations and scope, this work provides a solid foundation for additional research and it covers only a finite number of subtopics, yielding potential for future work in its areas of research.

1 INTRODUCTION

The Computer Science field of Natural Language Processing is one that is vastly studied in more recent years, due to its wide range of uses and due to the introduction of *Deep Learning* and increasing computational power [30]. Its usage spans various other fields, such as robotics and data exploration [21] and it is present within various aspects of our everyday lives, such as language translation and virtual assistants [17]. It is an amalgam of Linguistics and Artificial Intelligence. The overall aim of this field is for computers or machines, of any kind, to be able to decode and understand the syntactical structure and semantic meaning of natural human sentences, no matter the complexity. They need to be able to understand these sentences on their own. This can be achieved by receiving formal representations of such sentences.

Under the field of Natural Language Processing is a task called Meaning Representation Parsing. This task aims to convert, or encode, natural human sentences into specific formal representations (depending on the context and machine) that the machine can understand [24]. One way that this can be done is in a graphical representation, such as a task called Semantic Graph Parsing. This is the focus of this paper.

A *Semantic Graph Parser* (SGP) creates said graphs. The parser needs to be as accurate and efficient as possible for every possible sentence within the language(s) it is trained for. Using this understanding, these models would be able to both predict other sentences and fill in omitted words in a sentence. This can be done using various techniques and components. Over recent years, many researchers, such as [3, 4, 11, 23, 33], have published papers describing their attempts at creating accurate SGPs. Some use preexisting approaches and components, such as [5, 26], whilst others introduce novel ones, such as [9, 32].

Graphs consist of two components: nodes, representing the words of the sentence, and edges, representing the relationships between the nodes. Both components have labels attached which depict additional information. An additional complication that is considered in this paper is the non 1:1 correspondence between the words (also known as tokens) and the nodes. Nodes may correspond to one token, several tokens (known as spans) or sub-tokens. More on this in Section 3.1.

Given the above, the two main components of a SGP are, therefore, the node prediction module and the edge prediction module. Some parsers are developed using integrated node and edge modules, such as [15, 18], while some are designed with separate components, such as [4, 37]. This paper focuses on the latter component which is designed to perform as a separate entity to node prediction.

There are a variety of possible graph formations. Because machines expect a specific graph structure and specification, this formation is determined by the computational framework. These frameworks depict the required node and edge formation, as well as the expected label types. The main frameworks are depicted in a paper by Oepen et. al [27]. Our paper uses the *Elementary Dependency Structures* (EDS) framework and is solely focused on the English language.

Due to its popularity and extensive research history, the SGP task contains more than one known strategy to create such parsers. These parsers are developed using various techniques and they are evaluated using consistent evaluation metrics, which are explained in Section 4.4. They aim for the highest possible accuracy and efficiency (known as "state-of-the-art").

One common factor between the parsers is the use of a *Neural Network* to train the models. There are multiple type of Neural Networks, many of which have been used in the design of SGPs. Recently, a newer type of Neural Network has been developed [32], called the *Transformer*, which addresses some of the issues pertaining to the other preexisting types. It is often paired with pretrained data for more accurate learning. Transformers are becoming a popular option within parser design and has yet to be fully explored

within this specific field of work. Neural Networks are explained in more detail in Section 2.2.

This paper proposes another attempt at accurately predicting the edges between the nodes, given the correct nodes. It aims to investigate some of the techniques that have been used to create accurate, efficient edge prediction modules.

1.1 Research Questions

There are two research questions, which were initially proposed in this Honours Project's Project Proposal. As previously explained, the main aim of the edge prediction module is to understand and predict the edges of a semantic graph with the highest possible accuracy. This forms the second pipelined stage of the graph-based Semantic Parser. The following questions are thus proposed:

(1) How does the accuracy of an edge prediction module, developed using pretrained BERT, compare to parsers' edge prediction modules developed with non-trained LSTMs?
(2) How does the accuracy of an edge prediction module with a Maximum Entropy training objective (loss) compare to edge prediction modules that utilise a Maximum Margin training objective (loss)?

We hypothesise that the BERT module using Cross Entropy Loss (Maximum Entropy) will be the optimal module for this particular task.

1.2 Ethical, Professional and Legal Issues

No user evaluation was necessary for this paper's research. The only ethical consideration taken is with the potential biases (in terms of aspects such as race and gender) that come with the data obtained from external sources.

The input data is sourced from the LinGo Redwoods Treebank [28, 29]. This treebank is publicly accessible and is a composition of differing sources such as the Wall Street Journal.

The data pretraining was conducted using a pretrained BERT model from Hugging Face called "SpanBERT/spanbert-base-cased" [19] [1]. This model is used to amplify the predictions of spans.

It is recognised that these sources are not completely unbiased. Therefore, the modules developed follow the same fate. This potential bias has been minimised as best as possible and what is unavoidable is thus formally declared in this paper.

1.3 Structure of the Paper

This Honours Project Final Paper investigates the topics outlined above and aims to answer the above questions.

The paper begins with some relevant background on Semantic Parsing and Neural Networks for context. It then proceeds with the architectural structure of the modules developed, particularly delving into the structure of the Biaffine Network which is the main component of the modules. Following this is the experimental setup, which describes how the architecture is used to create the modules. This includes the preliminary steps before training as well as any concluding steps. Subsequently, the findings are presented and discussed. The paper is concluded with sections on the conclusions, paper limitations and potential future work.

2 BACKGROUND AND RELATED WORK

2.1 Semantic Parsing

Semantic Parsing is an umbrella term for various sub-topics, such as *Semantic Dependency Parsing* (SDP) and *Semantic Graph Parsing*. Each sub-topic is slightly different, however they all have the same goal: to formally represent human language so that it is understandable by a computer. Parsers of this nature will be provided, as input, human sentences and will provide, as output, a formal, computer-interpretable representation of the input. This may be in various forms, such as a table, tree or (usually directed) graph.

SDP focuses on the dependency relations between tokens. A relation is comprised of a head node and a dependency node, where the relation points from the head to the dependent, as the dependent node relies on the head node. This relation is called an arc, or edge, and is labelled with the dependency relation. Semantic Graph Parsing, however, incorporates a non-1:1 correspondence between nodes and tokens. This is the parsing problem researched in this paper.

Transition- and Graph-Based Parsing are two main parsing techniques researched in this Honours Project, the latter of which is the focus of this paper. Transition-Based Parsing is based off of *shift-reduce parsing* whereby buffer and stack data structures are utilised. Graph-Based Parsing, on the other hand, utilises tree and graph data structures to depict sentences. These structures are created using assigned scores which are used to find the most optimal final structure [20]. The main, relevant difference between a graph and tree structure is the presence of cycles in graphs and absence of such in trees. In the context of Semantic Parsing, this yields the ability to add additional edges.

As explained in Section 1, a graph consists of nodes (the words of the sentence) and edges (relationships between the nodes). One whole graph represents one sentence's meaning. Figure 1 [2] depicts this using the EDS framework.

The words of the input sentence are split up (known as *tokenised*) into tokens. These tokens are mapped to nodes. Often, one node is mapped to a particular token. At other times, a node may span multiple tokens. When using the EDS framework, nodes may come in several forms, such as **TOP** (root/head node), **_at_p** (preposition "at") and **card** (number). They may have none, one or multiple incoming and outgoing edges. The root (top) node will never have an incoming edge. Spans are explained in detail in Section 3.1.

When using the EDS framework, edges may come in several forms, such as **ARG1**, **BV**, **L-INDEX** and **R-INDEX**. These edges connect nodes together, yielding information about how they relate to form the meaning of the sentence.

2.2 Neural Networks

The main task of a Semantic Parser is to *learn* how to understand and produce human sentences. A very effective way to do this is through the use of *Neural Networks* (NNs).

NNs fall under Machine Learning. Their idea and design were inspired by the neurons in the human brain [13]. They consist of many interconnected basic computing units, called nodes (ranging from only a few to billions), which turn a vector of values into a

[1] https://huggingface.co/SpanBERT/spanbert-base-cased

[2] https://repgraph.vercel.app/main

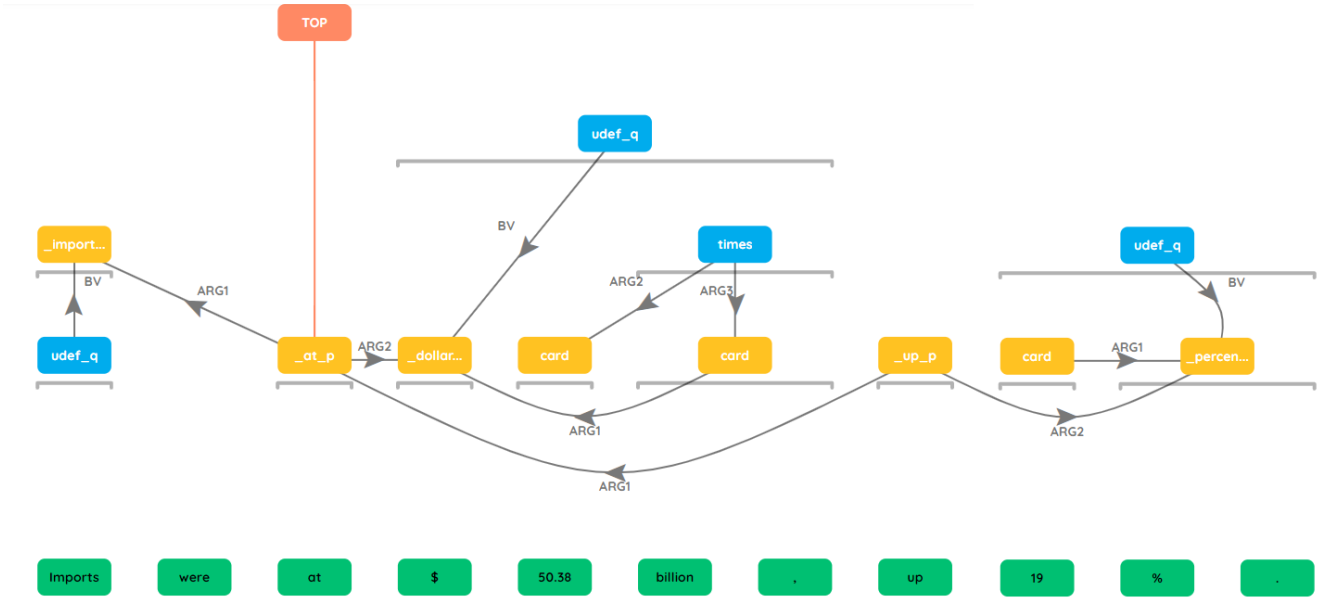

Figure 1: An Elementary Dependency Structures Graph for the Sentence: *Imports were at $50.38 billion, up 19%.* [7]

single-valued output. This output is then passed onto the remaining nodes in a particular way, depending on the structural requirements of the NN. The purpose of these nodes is to work together to compute complex calculations. Each node serves a unique purpose and these nodes are usually structured in a layered format. A NN will consist of an input layer, output layer, and a varying number of intermediate, hidden layers. Ordinarily, the layers are calculations, consisting of layer-specific biases and node-specific weights. Often, values are then put through a non-linear activation function to restrict the values to a certain range. An example of such a function is Sigmoid [20].

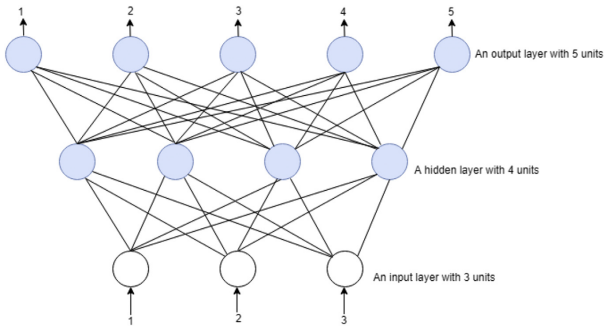

Figure 2: A Two-Layered Feedforward Neural Network [1]

Figure 2 depicts a simple NN example. The bottom layer is the input layer, consisting of three nodes which receive the raw data. They pass their outputs to the four nodes in the intermediate, hidden layer (which, in this case, is the only hidden layer). These four nodes pass their outputs to the output layer, which outputs the final values of the NN.

There are various different kinds of NNs, each of which serve different purposes. Some kinds are better suited for certain tasks than others. It is up to the researcher to choose which NN(s) suit(s) their models' components best.

The task presented in this paper is commonly designed using three distinct types.

The first type is the most straightforward: a *Feedforward* NN (FFNN). Within each layer of the network, every node receives input from every node in the previous layer, and feeds its output directly into every node in the following layer. This type is often suited for more basic calculations and is an appropriate introductory example for students. It fits into the Biaffine Network architecture, explained in Section 3.2.

The second type is the *Recurrent* NN (RNN). This type of network considers previously-processed nodes. This provides historical context (memory) so that the model can gain additional knowledge on the semantic meaning of the inputted sentence. Context refers to the understanding that words at the beginning of the sentence play a role in what the rest of the sentence may mean. For its input, a node of this type of NN relies both on the previous layers' outputs as well as the node's own output. This results in cycles, as nodes are not considered to be "independent units" [31], like they are in the FFNN. Papers, such as [3, 31, 34], support the use of RNNs within the design of Semantic Parsers.

A popular extension of the RNN, used as a baseline comparison in this paper's task, is a Long Short-Term Memory (LSTM) and its variation, a Bidirectional LSTM (BiLSTM), as used by [3, 22, 37]. The LSTM is designed to retain relevant context and remove what is irrelevant. The BiLSTM is a concatenation of two LSTMs which carry information in both directions [20].

There are certain disadvantages of the RNN, two of which are highlighted. Firstly, the longer the sentence, the less the NN remembers, resulting in a lower accuracy score. Secondly, due to the nature of its calculation process, the process must be completed sequentially, which can be very inefficient.

The final and most recent type is the *Transformer* NN. This was proposed by Vaswani et al. [32] in 2017. The main identifying feature is its ability to use positional encodings which are combined with additional embeddings. The structure is made up of *blocks*, whereby each block contains a combination of FFNNs, linear layers and self-attention layers. It takes, as input, a sequence of vectors and returns, as output, a sequence of vectors of the same size. More information on the design and architecture is described in Vaswani et. al's paper [32].

Despite how recent this Transformer NN is, it has been receiving a rapidly increasing amount of attention. Papers, such as [12, 32], have published papers proving the effectiveness of using Transformers as a means to train Semantic Parsers' components.

The Transformer's design combats some of the drawbacks of other NNs. The encoding is not completed sequentially. This allows for a greater retention of relevant contextual information. This also allows for parallelisation because each unit is treated independently of one another. This is an even more advantageous feature, given the expansion of computer architectures into using multiple cores.

A popular example of Transformers is the Bidirectional Encoder Representations from Transformers (BERT). This was developed by Devlin et. al in 2019 [8]. BERT is often used as a pretraining technique in Semantic Parsers and has been used in papers such as [16, 37].

3 ARCHITECTURAL STRUCTURE

Given an English sentence and its corresponding nodes, we want to be able to predict the edges that connect the nodes together, in order to form a complete representation of the sentence. We do this by training an edge prediction module of a SGP into learning a human language so that it may find a way to know which edges belong where. The module can do this, first, by assigning a score to every single possible edge (in other words, an edge from every node to every other node), whereby the higher the score, the more likely that the edge is correct. Subsequently, the module must then choose a combination of such edges to yield the highest-scoring graph. This will complete the entire formalism for the sentence at hand.

There are three main components to this edge prediction module, all of which are outlined in detail below.

3.1 Data Encoding

After the raw dataset is obtained from a treebank, it must be sent through an encoding layer, which prepares the data for the subsequent layer.

If required, the raw data must first be sent through a preprocessing stage which formats it into a more readable and extractable format, such as the CoNLL-U or CoNLL-X format [2]. This formatting is sent through the preprocessing encoding layer.

Preprocessing is conducted differently, depending on whether spans are required. Working with spans is an additional step within the process that frees the 1:1 token-node requirement. For example, it would be inappropriate to split up the term *South Africa* into two separate tokens, because they are not treated as two separate entities in human languages. This would be more appropriate as a single node, depicting the country it represents. In Figure 1, the second **card** node spans the tokens **billion** and **, (comma)**. Allowing spans in the structure frees up many restrictions placed on the sentence, which allows the structure to encompass more of the intricacies and exceptions of the language. Spans may consist of one token, multiple tokens, or even sub-tokens. Additionally, they may overlap. Whilst span prediction opens up the opportunity to capture more of the leniency and complexities of human language, they result in a more complicated development process and yield additional problems.

If no spans are required, the layer reads in the tokens from the data and encodes each token into a format readable by the subsequent layer.

However, if spans are required, the data is formatted in such a way that tokens and nodes are separate. The layer must read in each node's span information, dictating which tokens it spans. Each node now encompasses all token information of those which it spans.

3.2 Biaffine Network for Individual Edge Scoring

In 2016, Dozat and Manning [9] published a paper introducing an architecture, called the *Biaffine Network*, that can assign a score to every possible edge. This has received a lot of attention since publication due to its effectiveness, as papers such as [6, 22, 26] have used this in their design of a Semantic Parser.

There are several parts to this architecture, which are outlined below. Figure 3 depicts the overall architecture, which consists of four main layers. The BERT encoder of the left diagram replaces the Embedding-BiLSTM layers of the right diagram, as they perform the same task of creating embeddings to be fed into the subsequent layer. Most of the basic information described below can be found in Jurafsky and Martin [20].

The first layer (or stage) is the BERT encoder (or Embedding-BiLSTM encoder). BERT is a Transformer NN (explained briefly in Section 2.2) used to create contextual embeddings of the nodes. The weights used in the BERT model are usually fine-tuned during training to improve accuracy. BERT first encodes the sentence's tokens, creating a contextual embedding per token. These are then mapped to nodes based on the nodes' span information. This is done in the steps outlined below.

The tokens of the inputted sentence are first split into sub-tokens (see Section 4.1 for the example *finishing*).

Because the subsequent layer requires a single embedding per node, the token embeddings per node must be concatenated into a single embedding. He and Choi [14] explain two concatenation methods:

- Last Embedding: The node's embedding is its last sub-token.
- Average Embedding: The node's embedding is the average of all of its tokens.

A variation of the Average Embedding method is to subtract the first and last sub-token embeddings.

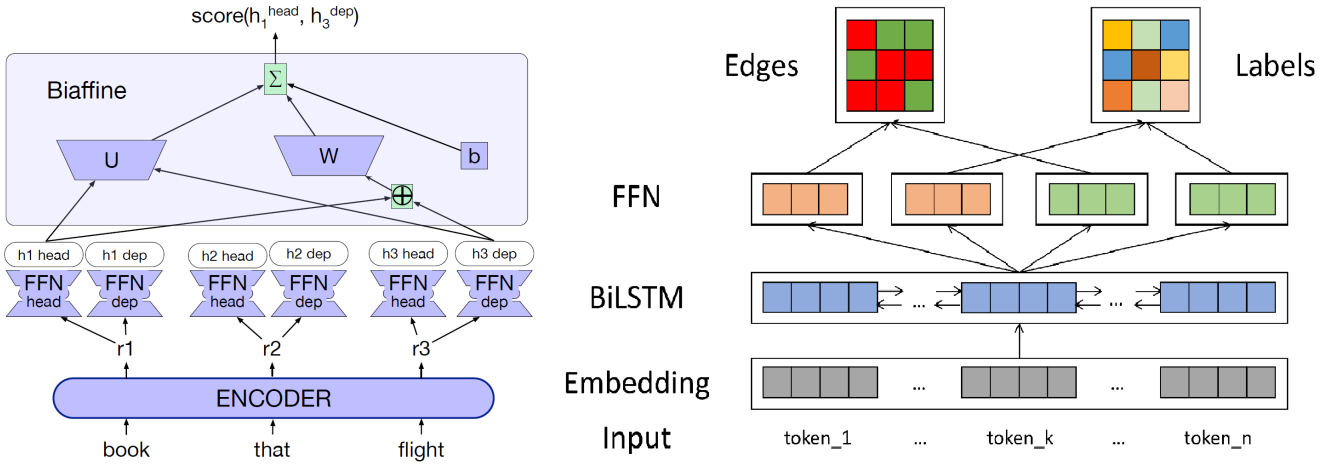

Figure 3: A Biaffine Network using BERT (left) [20] and using BiLSTM (right) [10, 35]

The node embeddings, $\mathbf{r}_i$ for node i, are sent to two FFNNs. These FFNNs consist of a linear, activation and - if required - dropout layer. The outputs of these two modules are representations for the head and dependent nodes and their corresponding labels. For context, an edge's representation is in the form: $\mathbf{h}_i^{head} \rightarrow \mathbf{h}_i^{dep}$, where i and j are nodes.

The representations are presented as follows:

$$\mathbf{h}_i^{(\text{edge-head})} = \text{FFNN}^{(\text{edge-head})}(\mathbf{r}_i) \quad (1)$$

$$\mathbf{h}_i^{(\text{edge-dep})} = \text{FFNN}^{(\text{edge-dep})}(\mathbf{r}_i) \quad (2)$$

$$\mathbf{h}_i^{(\text{label-head})} = \text{FFNN}^{(\text{label-head})}(\mathbf{r}_i) \quad (3)$$

$$\mathbf{h}_i^{(\text{label-dep})} = \text{FFNN}^{(\text{label-dep})}(\mathbf{r}_i) \quad (4)$$

where $\mathbf{h}_i^{head}$ is the head node's embedding representation and $\mathbf{h}_i^{dep}$ is the dependent node's embedding representation.

These representations are sent to the third, main layer: the *Biaffine scoring function*. This is a classifier network that uses trained weights and biases to assign a score to the edge between the inputted nodes. It uses the following equations:

$$\mathrm{s}_{ij}^{(\text{edge})} = \text{Biaff}^{(\text{edge})}(\mathbf{h}_i^{(\text{edge-head})}, \mathbf{h}_j^{(\text{edge-dep})}) \quad (5)$$

$$\mathrm{s}_{ij}^{(\text{label})} = \text{Biaff}^{(\text{label})}(\mathbf{h}_i^{(\text{label-head})}, \mathbf{h}_j^{(\text{label-dep})}) \quad (6)$$

where *Biaff* is the Biaffine scoring function, s_{ij} is the score, and

$$\text{Biaff}(\mathbf{x}, \mathbf{y}) = \mathbf{x}^{\mathrm{T}}\mathbf{U}\mathbf{y} + \mathbf{W}(\mathbf{x} \bigoplus \mathbf{y}) + b \quad (7)$$

where $\mathbf{x}$, $\mathbf{y}$ are the representations, $\mathbf{U}$ and $\mathbf{W}$ are the weight matrices, and b is the bias term.

A score is only valid when placed relative to other scores. For example, a lone score of **2** does not clarify whether it is high or not. It will hold a high ranking if the other scores range between **-2** and **3**, and it will hold a low ranking if the other scores range between **1** and **5**.

When using Cross (Maximum) Entropy Loss, once the Biaffine layer has calculated a score for an edge, the score is converted into a probability using a softmax function. This converts the score into a probability (between 0 and 1). The higher the probability score, the more likely the edge is correct and should be included in the final graph. This is necessary for the Cross Entropy Loss calculation, as it is based on probabilities. This calculation is as follows:

$$P(y_{ij}^{(\text{edge})}|\mathbf{w}) = \text{softmax}(\mathbf{s}_{ij}^{(\text{edge})}) \quad (8)$$

$$P(y_{ij}^{(\text{label})}|\mathbf{w}) = \text{softmax}(\mathbf{s}_{ij}^{(\text{label})}) \quad (9)$$

where $P(y_{ij}|\mathbf{w})$ is the new score in the form of a probability.

This concludes the Biaffine Network. The module now has a large number of edges and their corresponding scores. It must now find a way to choose the best combination to construct the semantic graph.

3.2.1 Loss Function. The loss function is an essential aspect of any trained model. This is calculated after every step (epoch) of training and is optimised in order to produce the most accurate model. The predicted probabilities of the edges are compared against the expected edges. The loss results then help with fine-tuning the model's parameters, as a lower loss results in a higher model accuracy score. It updates the model's weights by using a backpropagation algorithm [20].

There are different types of loss functions. One type of loss function is *Cross Entropy*. Entropy is loosely defined as the amount of knowledge a model has and Cross Entropy uses the principle of Maximum Entropy. This loss function aims to produce high-quality graphs by minimising the loss (error) [20]. It receives per-edge scores as input and compares them against which edges are expected.

Another type is *Maximum Margin*. This aims to optimise a specific loss value, which compares the expected (*gold*) graph and the current predicted graph [4]. Maximum Margin receives the highest-predicted graph as input. A value is calculated by comparing this to the expected graph. The term "margin" is based on maximising the margin between the graph's scores, by increasing the score of correct parts of the predicted graph and decreasing the score of the incorrect parts.

function MAXSPANNINGTREE(G=(V,E), $root$, $score$) **returns** *spanning tree*

$F \leftarrow []$
$T' \leftarrow []$
$score' \leftarrow []$
for each $v \in V$ **do**
  $bestInEdge \leftarrow \text{argmax}_{e=(u,v) \in E}\ score[e]$
  $F \leftarrow F \cup bestInEdge$
  for each $e=(u,v) \in E$ **do**
    $score'[e] \leftarrow score[e] - score[bestInEdge]$

if T=(V,F) is a spanning tree **then return** it
else
  $C \leftarrow$ a cycle in F
  $G' \leftarrow$ CONTRACT(G, C)
  $T' \leftarrow$ MAXSPANNINGTREE$(G', root, score')$
  T $\leftarrow$ EXPAND(T', C)
  return T

function CONTRACT(G, C) **returns** *contracted graph*

Figure 4: The Chu-Liu Edmonds Algorithm for Finding the Maximum Spanning Tree [20]

3.3 Maximum Spanning Tree Algorithm for Graph Scoring

Following the Biaffine Network component is an algorithmic component that takes all of the edges' scores, as input, and computes the highest-scoring combination of edges to create the final Semantic Graph.

An intuitive first algorithm is the *greedy* algorithm. This simply takes the highest-scoring incoming edge for each node. However, the greedy algorithm is not ideal for all cases for two main reasons. Firstly, the chosen edges may not create a fully-connected graph. Secondly, when while keeping the fully-connected graph restriction, the score may not be the highest-possible score. This happens because, if insisting on including the highest-scoring edges, the effect on the rest of graph may yield a lower combined score [20]. However, the greedy algorithm is an effective starting point for a more effective approach.

One of these more effective approaches is called the *Chu-Liu Edmonds Algorithm* for finding the *Maximum Spanning Tree*. A spanning tree is a tree-based data structure (an acyclic graph), consisting of nodes connecting with directed edges, where every single node is (not necessarily directly) connected to every other node. A maximum spanning tree is a spanning tree in which its chosen edges yield the highest combined score possible, given all possible edges.

Whilst ultimately aiming for a graph structure, beginning with this tree algorithm ensures that the final graph consists of a combination of edges that link together in a way that makes sense. Ensuring that a tree structure is present within the graph guarantees that all nodes are connected.

Figure 4 depicts this recursive algorithm. The algorithm's explanation can be found in Jurafsky and Martin [20]. Once the maximum spanning tree is found, additional non-tree edges are added which add to the semantic meaning of the sentence. This creates the SGP's fully-predicted Semantic Graph.

4 EXPERIMENTAL SETUP

As previously explained, the overall aim of this SGP is to learn the English language well enough to be able to formally represent its sentences in a graphical representation, dictated by the EDS framework. The edge prediction module is the parser which represents the relationships between the words (tokens) of the sentence. There are four main stages to this.

The first stage of the process is for the module to be **built**. The module parameters are set (such as the learning rate and number of epochs) and the data is preprocessed to be fed into the module.

Subsequent to successful building, the second stage is for the module to be **trained** by providing it with a large number of sentences and their correctly-mapped nodes, edges and labels. This data can be obtained through a Treebank (see Section 1.2 for information about this paper's data acquisition). The more data it is provided with, the more accurate the module can become and it must study the data repeatedly (dictated by the number of epochs). However, there is a risk of overfitting, whereby the module starts to "overstudy" the training data and is unable to understand new sentences.

After training is **validation** (or development). This is used on the module, after training, to tune the module so that its loss function is optimised. In other words, it is used to "validate" or "develop" the module. This can be used to both prevent and identify overfitting, as the accuracy results of the validation set would be significantly lower than that of the training set if overfitting was occurring.

The final stage of this process is **testing**. This, intuitively, is a dataset used to "test" the module's performance. It is expected to perform slightly worse than the other sets. Overfitting has occurred during training if the test set performs much worse than the training set.

The development process consisted of a heavy code aspect and all coding was developed using Python. The main package used was *PyTorch*.

Implementing a high-quality Biaffine Network is not a straightforward task with built-in functions. It requires a number of complex tasks integrated to produce a functioning network. Implementing this from scratch is out of scope for this paper. It requires additional experience and knowledge, both on the Biaffine Network and on the programming language chosen.

Due to this constraint, a Python codebase was chosen on the recommendation of this paper's supervisor, Dr Jan Buys. This codebase, called *SuPar* was created by Yu Zhang, a first-year PhD student from Soochow University in Taipei [3]. It is publicly accessible on GitHub [4] [36]. It is a program which encompasses different types of state-of-the-art Semantic Parsers for over nineteen languages.

It proved challenging to navigate through, and adapt, this code. Zhang has more experience both in this field and in the Python language. He designed the codebase using a modular, yet complex, structure and utilised Python's many complex functions. Only relevant parts of the code were used in this paper's implementation. This implementation is based, specifically, off of the project's Biaffine Semantic Dependency Parser. While all classes were left in

[3] https://yzhang.site/
[4] https://github.com/yzhangcs/parser

the submitted code, we only adapted what was required for our implementation. See Appendix A for the list of classes.

The training process is computationally-expensive, requiring a lot of time and computational power. Due to this heavy demand, the modules were sent to be trained on the Centre for High Performance Computing (CHPC). The modules were trained on the CHPC using at least ten central processing units (CPUs) and one graphics processing unit (GPU), which is significantly more computational power than the average dual-core laptop can provide. Even with this increased power, each module took hours to train.

The following subsections of this section outlines the detailed steps of the research implementation (development) process. It begins with the data preprocessing, followed by how the Biaffine Network and greedy algorithm were used. It continues with a description of how the slightly different modules were developed and ends off with a description of the evaluation process and additional hyperparameter tuning.

4.1 Parser Building and Data Preprocessing

As explained above, the first stage of the code is to build the SGP. Whilst the original codebase provides three different types of parsers to be developed, only the relevant type (*sdp.py*) was utilised and adapted for our SGP.

The code first requires an array of parameters dictating various aspects. For example, it requires information on where to find the train, validation and test data, where to store the model, which encoder to use and which features to include. We added several additional parameters which were required for our development.

The parser is then built by adjusting the structure, according to the inputted parameters, and reading in the data.

As explained in Section 1.2, the data source for this particular paper is from the (publicly-accessible) LinGo Redwoods Treebank [29]. We use 38,893 data units in total, each unit consisting of a sentence and its corresponding tokens, nodes and edges. This information is appropriately distributed into a training (91.53%), evaluation (4.66%) and test (3.81%) set.

We were provided a program (from supervisor, Dr Jan Buys) that receives, as input, the original raw data and produces, as output, eds files. These files are consistent with json files. These files place the data into a more readable and more easily-accessible format.

We then placed these files, as input, into a self-written data conversion Python file (*data_conversion.py*). Our file constructed the data in a way that is consistent with what the edge prediction implementation expects. Initially, the data was formatted into a *CoNLL-U/X* format [2]. We first minimally adapted the format to suit this parser's requirement. We then adapted it to accommodate for the spans.

During the initial implementation of the Biaffine Network, tokens were inputted. Subsequently, when the code was adapted for spans, the tokens and nodes were to be kept separately. This required the tokens to be placed as an additional layer on the initial CoNLL-U format and the format, itself, had to be completely re-designed.

The original codebase used objects *CoNLLs* to represent the tokens and the CoNLLs of each sentence were placed in an object *CoNLLSentence*. A CoNLL object (representing a token) consists of:

- **ID**
- **FORM** (word)
- **LEMMA**
- **CPOS** (coarse-grained part-of-speech tag)
- **POS** (fine-grained part-of-speech tag)
- **FEATS** (features)
- **HEAD** (token head)
- **DEPREL** (dependency relations to the HEAD)
- **PHEAD** (projective heads of the token)
- **PDEPREL** (dependency relations to the PHEAD)

During the first implementation, we only required the **ID**, **FORM** and **PHEAD** (which contained the edges in the form **<source node 1>:<label 1>|<source node 2>:<label 2>...**).

The second step, involving spans, required us to conduct a heavy adaptation of the code. The structure of the sentence objects proved difficult to adapt in order to allow for both tokens and nodes to be inputted as separate entities. The adaptation involved us adjusting the format required and the data structures used to store the data. This code adaptation was required until the embeddings were calculated and replaced the raw data input.

The new CoNLL object was adapted to the following form:

- **ID**
- **TAG**
- **START** (the span's start token)
- **END** (the span's end token)
- **EDGES** (in the form: **<source node 1>:<label 1>|<source node 2>:<label 2>...**)

The CoNLLSentence object still holds these CoNLL objects with an additional attribute called **FORM**. This takes the form of a tuple **(WORD, CHAR, ELMO, BERT)**, whereby **WORD** stores the tokens, **CHAR** stores the Character embeddings, and **ELMO** and **BERT** hold space for those appropriate features. We ensured that the **FORM** attribute was used in place of what was used in the original codebase.

Once the data had been successfully loaded into the code, we sent them to a BERT layer. The BERT model used is called *SpanBERT/spanbert-base-cased* from Hugging Face, developed by Joshi et. al [19]. This model is designed to be better-suited for span implementations. The tokens were first sent to a BERT tokeniser. This tokeniser splits each token into sub-tokens (for example, *finishing* is split up into *finish* and its suffix *##ing*). These sub-tokens are then converted into IDs, which correspond to the BERT model's fixed vocabulary (consisting of 28,996 words and their corresponding ID values).

These IDs are sent through the pretrained BERT model's many layers. Their corresponding embeddings are then outputted. These embeddings are what is sent to the subsequent Biaffine Network Layer. There were two implementations of this.

During the first implementation, where spans are not taken into account, we sent the sub-token IDs through the model without any intermediate steps.

Figure 5 depicts the second implementation. The data input includes span information whereby, for each node, the start and end tokens are documented (Step 1). To reduce complication for when a node only encompasses one token, we depict the end token as the first token after the span ends. We create two data structures, called

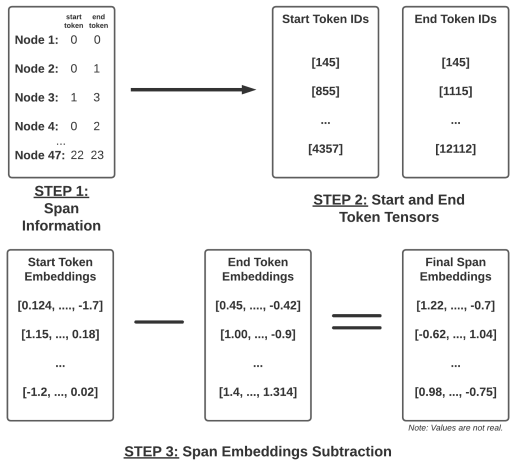

Figure 5: BERT Embeddings with Span Adaptation

tensors, storing the nodes' start token IDs and their end token IDs (Step 2). These tensors then go through the same process that calculates their embeddings. We then then subtract the embeddings from one another [(end token embedding) - (start token embedding)] (Step 3). This creates the final node embeddings. After dropout is conducted, they are sent through to the next layer.

After the successful training of the main module (using spans, a Biaffine Network, Cross Entropy [Maximum Entropy] Loss and BERT pretraining), the non-trained LSTM module was created using the same codebase. The module required the same data preprocessing as the BERT module. However, the BiLSTM NN replaces the BERT NN. GloVe [5] and Character embeddings are calculated and concatenated to create the final token embeddings which are sent to the BiLSTM. The output of the BiLSTM is the appropriate equivalent to the BERT token embeddings. The same subtraction process is then conducted. This required us to utilise and adapt additional classes and functions.

4.2 Biaffine Network: Individual Edge Scoring

Once the data is understood and is converted into embeddings, the embeddings are sent through four Multilayer Perceptrons (which are the FFNNs described in Section 3.2): two create the head and dependent representations for the edge and two create the head and dependent representations for the labels.

These representations are sent through two Biaffine layers, outputting scores for both the edges and the labels.

The loss is subsequently calculated when given the scored edges and labels, as well as the gold edges and labels.

Fortunately, the original code required very little adaptation after the embeddings were calculated. At this layer, we only needed to ensure that the required attributes, separated during the span adaptation, are not lost during the process.

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

4.3 Maximum Spanning Tree Algorithm for Graph Scoring

Given the edge and label scores, the edges are first investigated. Initially, the greedy algorithm was used (using the function *argmax*).

Using the greedy approach, the edges are first chosen. The design of this codebase's Biaffine Network resulted in two values being outputted per score entry, whereby one value is the negative of the other. If the second value is the positive, it is chosen to be included. A mask is created to depict which edges are chosen to be included or excluded.

The labels are also sent through the argmax function. While each edge score entry held two values, each label score entry held as many values as there are label types (such as ARG1, BV, etc.). The maximum score of each entry is chosen such that *if* that edge is chosen, this label would be attached to that edge.

The edge mask is mapped to the maximum label scores. The result of this is the predicted labels for each included edge. This is in the form of a tensor. This is sent to be decoded and printed out in the same format as the input. See the format in Section 4.6.

Due to time limitations, the Maximum Spanning Tree algorithm could not be integrated in time. The code remains in the submitted codebase and the place of integration is noted in the code. However, its successful integration is left as future work.

4.3.1 Loss Function. The Cross Entropy Function (torch.nn.CrossEntropyLoss) was used as the criterion to be optimised in the module. We appended an additional parameter which allows the user to choose between Cross Entropy and Maximum Margin (torch.nn.MarginRankingLoss) as the chosen loss function. Due to time limitations, the Maximum Margin adaptation was not completed. However, the code allows for the addition if future work is conducted.

4.4 Evaluation

In order to answer the research objectives posed, an evaluation process needs to take place. This evaluation calculates how accurate the parser module is by comparing the predicted edges of an inputted sentence to the gold (expected) edges of that sentence. The test sentences are obtained from the same corpus used to train the module. None of the sentences inputted for testing have been seen before by the module within the training or validation sets.

There are three common, main metrics used to evaluate this module: *Precision*, *Recall* and *F1 Score*. The F1 score is commonly used as an evaluation metric in the field of Semantic Parsing. All of these are used on both the edges themselves (which relationships exist between nodes) and the edges' labels (what type of relationship exists). The calculations are as followed [20]:

- **Precision**: the ratio of correct edges/labels to total predicted edges/labels.
- **Recall**: the ratio of correct edges/labels to total expected edges/labels.
- **F1 Score**: the "harmonic mean" between Precision and Recall.

A zero (0) score indicates that the module has failed to predict any correct edges or labels. A one (1) score indicates that the module has successfully predicted all of the correct edges and labels, and has

not predicted anything extra. The goal of any parser is to achieve an F1 score of as close to one as possible.

We developed an evaluation program in Python, lightly based off of a program developed by our fellow Honours Project partner, Chase Ting Chong. This reads in the predicted and gold graphs, calculates the three metrics, and writes this to a file. The results are depicted in the subsequent section, Section 5.

4.5 Hyperparameter Tuning

After the BERT and BiLSTM modules successfully trained using this specific dataset and with the code's span adaptation, we attempted to calculate the number of epochs that would maximise their training potentials.

Appendix B, includes a depiction of the Cross Entropy Loss results in a tabular (Table 4 and 5) and graphical (Figure 7 and 8) formation.

This depicts interesting results for the BERT module. All but one set of data follow the expected trend (whereby the loss decreases rapidly and then flattens out). The "Dev Label Loss" (validation) reaches a minimum much earlier than the other loss sets, increases, and then converges higher than the rest. Due to the paper's scope, this calls for additional research in a follow-up paper.

In addition to the above graph, F1 scores were calculated using the validation sets for the main module (BERT). Table 6 and Figure 9 are depicted in Appendix B. Taking the loss and evaluation score data into consideration, the BERT module with twenty (20) epochs is chosen, mainly because the additional epochs do not yield much benefit for the accuracy.

Our adapted codebase's implementation of BERT yielded high evaluation scores without any hyperparameter tuning. However, this is not the case for the BiLSTM module. Additional hyperparameter tuning could not be completed due to time limitations and this is *highly* recommended as future work. Due to this limitation, the module did not yield satisfactory preliminary results and this has caused limited results to be presented in this paper.

Table 5 and Figure 8 depict the experimentation completed on the BiLSTM module with fifteen epochs. Loss does decrease, however it is very minimal. Twenty (20) epochs are chosen as the final BiLSTM module, however, to use for the results but we emphasise that this is no way an optimal module to use. The codebase sets the initial epoch size to five thousand (5000), which indicates that the default parameters may not be optimal for such a low number of epochs. Training the module at such a high number of epochs would have been too high for our resources.

The final module parameters are depicted in Appendix C.

4.6 Final Edge Prediction Module Overview

Figure 6 depicts the typical input and output of our developed module's prediction. The input consists of the tokens (words) of the sentence, preceded by an exclamation mark indicator, and the nodes presented in the CoNLL object format (see Section 4.1).

The end token depicted is the first token after the end of the span. This is explained in Section 4.1. If the end token of a span is the final token in the sentence, we choose to use the automatic end-of-sentence (*EOS*) token for BERT (102) and the pad (*PAD*) token for the BiLSTM (0).

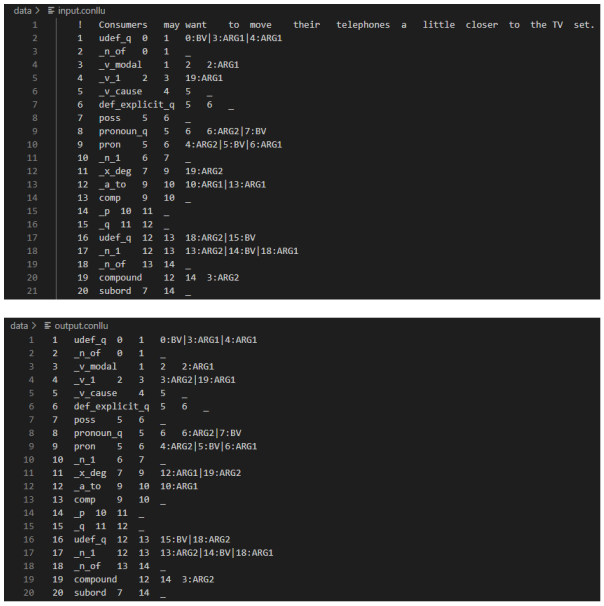

Figure 6: The Expected Input (Top) and Output (Bottom) of the Developed Edge Prediction Module

The output is presented very similarly to the input. The only difference is the omission of the tokens in the first row, as those are solely used to process the data.

5 RESULTS AND DISCUSSION

A working edge prediction module was developed successfully during the paper's implementation. Initially, the module used pretrained BERT and assumed that tokens corresponded to nodes in a 1:1 fashion. After this module trained and predicted successfully, we adapted the code to separate the nodes and tokens in order to allow for a non-1:1 correspondence. Subsequent to the success of this stage was the BiLSTM adaptation of the span module, which is used to answer both of the research objectives below.

A noteworthy comparison to make, before demonstrating the research objective results, is between the module with and without spans. Without the span adaptation, several nodes correspond to one token. This results in a rapid increase in predicted edges. Table 8 in Appendix D depicts the accuracy score comparison. The F1 scores are significantly different. While precision is considerably different, recall is surprisingly less so. To reiterate, precision is the ratio of correct edges to total predicted edges, while recall is the the ratio of correct edges to total expected edges. This is a envisioned observation because the correct edges are predicted within the bundle of additional edges that are outputted. The additional edges are incorrect, called *false positives*, therefore bringing down precision more than recall.

As stated in Section 1.1, our research questions relate to two main comparisons: (1) Pretrained BERT versus Non-Trained LSTM and (2) Maximum Entropy versus Maximum Margin Loss.

Due to the time limitations, only the BERT and BiLSTM Biaffine modules (both using Maximum Entropy Loss) were successfully

developed before the paper's deadline. However, despite the successful training, the BiLSTM module failed to provide satisfactory results. Therefore, no valid evaluation scores could be outputted to answer this paper's research questions below. These results, although impractical to answer the questions, are depicted in Tables 9 and 10 in Appendix D.

The BERT module results are depicted as followed:

5.1 Pretrained BERT vs. Non-Trained LSTM

Accuracy Scores (%)	Precision	Recall	F1 Score
Pretrained BERT	97.36	95.63	**96.42**

Table 1: F1 Results on Test Data for BERT (20 Epochs)

This module yielded very high accuracy scores, with both high precision and recall. This result was yielded without hyperparameter tuning.

5.2 Maximum Entropy Loss vs. Maximum Margin Loss

Because the BiLSTM module could not produce results, no comparison can be made. However, the Maximum Margin results used are presented:

Accuracy Scores (%)	Precision	Recall	F1 Score
Maximum Margin (Random)	94.25	95.57	94.98
Maximum Margin (W2V)	94.72	96.12	95.42
Maximum Margin (ELMo)	88.44	92.40	90.38
Maximum Margin (ELMo*)	95.80	96.79	96.29

Table 2: F1 Results on Validation Data for Max. Margin Loss

The Maximum Margin results shown are impressive, ranging from a F1 score of 90.38% to 96.29%. These results are extracted from Cao et. al's paper on Knowledge-Intensive and Data-Intensive Models [4]. The models depicted are:

- **Random:** BiLSTM and random embedding initialisation
- **W2V:** BiLSTM and word2vec features [25]
- **ELMo:** ELMo features and softmax layer for classification
- **ELMo*:** BiLSTM and ELMo features

These models were chosen because they align with the BiLSTM module developed. A model using BERT had not been found. Because the above results for the baseline Maximum Margin parser are extracted from the cited sources, this hinders the ability to form confident conclusions based on the comparison between that of our developed parser and that of the baseline results. This is because they do not share the same development environment and module parameters are likely to be different.

Due to the sub-optimal results presented in our developed BiLSTM module, no official conclusions can be made.

The results presented are severely limited. This paper provides a solid foundation for future work to be conducted and, with additional module tuning, the results can be updated and validated.

6 CONCLUSIONS

In this paper, we produced additional research within the topic of Semantic Graph Parsing. We attempted to develop and compare two edge prediction modules, one using BERT embeddings and one using a BiLSTM with Character and GloVe embeddings. Additionally, we briefly delve into the effectiveness of the Maximum Entropy Loss Principle, as a means to measure a model's loss, as opposed to Maximum Margin Loss. Using the results above, we cannot provide confident conclusions to answer our research questions. However, we have provided evidence that the modules developed are viable and, given additional time, additional hyperparameter tuning can improve the BiLSTM module. We believe that this paper provides a solid foundation for future work, which will aid the research and evolution of Semantic Graph Parsing techniques.

7 LIMITATIONS AND FUTURE WORK

7.1 Limitations

Three main limitations of this paper are outlined below. Firstly, the codebase developed is based off of a preexisting codebase. This codebase is designed at a level more advanced than we can fully understand at our level of knowledge on the topic and on the Python language. A deeper understanding of the codebase's functions and structure would have enabled a better adaptation and redesign of the necessary aspects. Secondly, due to time limitations, not everything that was initially planned could be achieved within the given time frame. Besides the main module, which utilised BERT and Maximum Entropy, only the non-trained LSTM module could also be successfully built and briefly analysed by the deadline. Therefore, the module using Maximum Margin was not able to be built within the time frame given. Finally, another limitation (explained in Section 4.5) is the lack of hyperparameter tuning of the BiLSTM module, which yielded poor evaluation scores. These limitations have restricted the validity of the results and conclusions. Future work is highly-recommended to improve these modules.

7.2 Future Work

This paper is based on a fast-paced, highly-demanded field with many uses. Therefore, there are many future paths this paper's work can take. Some, amongst many others, are outlined. Firstly, Section 4.5 explains how hyperparamter tuning was not conducted due to time limitations. This is highly recommended in a follow-up paper, as this will yield valid conclusions. Maximum Margin should be used to build a module, based off of this preexisting code, in order to provide additional validity (once the BiLSTM module is viable) to the above accuracy results between Maximum Entropy and Maximum Margin. *SpanBERT/spanbert-base-cased* was used as the BERT encoder. However, this is one of many other BERT models which could be used and a comparison between different BERT encoders could be a noteworthy comparison. Section 4.5 also explains a surprising loss result. This should be further investigated for a better understanding of its cause. A final future recommendation is to integrate the Maximum Spanning Tree algorithm, which could not be integrated due to time limitations, despite most of the code already being included in the codebase.

ACKNOWLEDGMENTS

I would like to acknowledge my project supervisor, Dr Jan Buys, for his continuous support and guidance.

REFERENCES

[1] Oludare Isaac Abiodun, Aman Jantan, Abiodun Esther Omolara, Kemi Victoria Dada, Nachaat AbdElatif Mohamed, and Humaira Arshad. 2018. State-of-the-art in artificial neural network applications: A survey. *Heliyon* 4, 11 (2018), e00938.

[2] Sabine Buchholz and Erwin Marsi. 2006. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the tenth conference on computational natural language learning (CoNLL-X)*. 149–164.

[3] Jan Buys and Phil Blunsom. 2017. Robust Incremental Neural Semantic Graph Parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vancouver, Canada, 1215–1226. https://doi.org/10.18653/v1/P17-1112

[4] Junjie Cao, Zi Lin, Weiwei Sun, and Xiaojun Wan. 2021. Comparing Knowledge-Intensive and Data-Intensive Models for English Resource Semantic Parsing. *Computational Linguistics* 47, 1 (2021), 43–68.

[5] Wanxiang Che, Longxu Dou, Yang Xu, Yuxuan Wang, Yijia Liu, and Ting Liu. 2019. HIT-SCIR at MRP 2019: A Unified Pipeline for Meaning Representation Parsing via Efficient Training and Effective Encoding. In *Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 Conference on Natural Language Learning*. Association for Computational Linguistics, Hong Kong, 76–85. https://doi.org/10.18653/v1/K19-2007

[6] Yufei Chen, Yajie Ye, and Weiwei Sun. 2019. Peking at MRP 2019: Factorization- and composition-based parsing for elementary dependency structures. In *Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 Conference on Natural Language Learning*. 166–176.

[7] Jaron Cohen, Roy Cohen, Edan Toledo, and Jan Buys. 2021. RepGraph: Visualising and Analysing Meaning Representation Graphs. (2021).

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[9] Timothy Dozat and Christopher D Manning. 2016. Deep biaffine attention for neural dependency parsing. *arXiv preprint arXiv:1611.01734* (2016).

[10] Timothy Dozat and Christopher D Manning. 2018. Simpler but more accurate semantic dependency parsing. *arXiv preprint arXiv:1807.01396* (2018).

[11] Yantao Du, Fan Zhang, Weiwei Sun, and Xiaojun Wan. 2014. Peking: Profiling syntactic tree parsing techniques for semantic graph parsing. In *Proceedings of the 8th international workshop on semantic evaluation (semeval 2014)*. 459–464.

[12] Gabriela Ferraro and Hanna Suominen. 2020. Transformer semantic parsing. In *Proceedings of the The 18th Annual Workshop of the Australasian Language Technology Association*. 121–126.

[13] Yoav Goldberg. 2016. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research* 57 (2016), 345–420.

[14] Han He and Jinho Choi. 2020. Establishing strong baselines for the new decade: Sequence tagging, syntactic and semantic parsing with BERT. In *The Thirty-Third International Flairs Conference*.

[15] Luheng He, Kenton Lee, Omer Levy, and Luke Zettlemoyer. 2018. Jointly Predicting Predicates and Arguments in Neural Semantic Role Labeling. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, Melbourne, Australia, 364–369. https://doi.org/10.18653/v1/P18-2058

[16] Daniel Hershcovich and Ofir Arviv. 2019. TUPA at MRP 2019: A Multi-Task Baseline System. In *Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 Conference on Natural Language Learning*. Association for Computational Linguistics, Hong Kong, 28–39. https://doi.org/10.18653/v1/K19-2002

[17] Julia Hirschberg and Christopher D Manning. 2015. Advances in natural language processing. *Science* 349, 6245 (2015), 261–266.

[18] Zhengbao Jiang, Wei Xu, Jun Araki, and Graham Neubig. 2020. Generalizing Natural Language Analysis through Span-relation Representations. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 2120–2133. https://doi.org/10.18653/v1/2020.acl-main.192

[19] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S Weld, Luke Zettlemoyer, and Omer Levy. 2020. Spanbert: Improving pre-training by representing and predicting spans. *Transactions of the Association for Computational Linguistics* 8 (2020), 64–77.

[20] Daniel Jurafsky and James H. Martin. 2021. *Speech & Language Processing*. Pearson Education India.

[21] Aishwarya Kamath and Rajarshi Das. 2018. A survey on semantic parsing. *arXiv preprint arXiv:1812.00978* (2018).

[22] Yuta Koreeda, Gaku Morio, Terufumi Morishita, Hiroaki Ozaki, and Kohsuke Yanai. 2019. Hitachi at MRP 2019: Unified Encoder-to-Biaffine Network for Cross-Framework Meaning Representation Parsing. In *Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 Conference on Natural Language Learning*. Association for Computational Linguistics, Hong Kong, 114–126. https://doi.org/10.18653/v1/K19-2011

[23] Sunny Lai, Chun Hei Lo, Kwong Sak Leung, and Yee Leung. 2019. CUHK at MRP 2019: Transition-Based Parser with Cross-Framework Variable-Arity Resolve Action. In *Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 Conference on Natural Language Learning*. Association for Computational Linguistics, Hong Kong, 104–113. https://doi.org/10.18653/v1/K19-2010

[24] Zi Li and Nianwen Xue. 2019. Parsing meaning representations: Is easier always better?. In *Proceedings of the first international workshop on designing meaning representations*.

[25] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* 26 (2013).

[26] Seung-Hoon Na, Jinwoon Min, Kwanghyeon Park, Jong-Hun Shin, and Young-Kil Kim. 2019. JBNU at MRP 2019: Multi-level Biaffine Attention for Semantic Dependency Parsing. In *Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 Conference on Natural Language Learning*. Association for Computational Linguistics, Hong Kong, 95–103. https://doi.org/10.18653/v1/K19-2009

[27] Stephan Oepen, Omri Abend, Jan Hajic, Daniel Hershcovich, Marco Kuhlmann, Tim O'Gorman, Nianwen Xue, Jayeol Chun, Milan Straka, and Zdenka Uresova. 2019. MRP 2019: Cross-Framework Meaning Representation Parsing. In *Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 Conference on Natural Language Learning*. Association for Computational Linguistics, Hong Kong, 1–27. https://doi.org/10.18653/v1/K19-2001

[28] Stephan Oepen and Dan Flickinger. 2019. The ERG at MRP 2019: Radically compositional semantic dependencies. In *Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 Conference on Natural Language Learning*. 40–44.

[29] Stephan Oepen, Dan Flickinger, Kristina Toutanova, and Christopher D Manning. 2004. Lingo redwoods. *Research on Language and Computation* 2, 4 (2004), 575–596.

[30] Daniel W Otter, Julian R Medina, and Jugal K Kalita. 2020. A survey of the usages of deep learning for natural language processing. *IEEE transactions on neural networks and learning systems* 32, 2 (2020), 604–624.

[31] Kanchan M Tarwani and Swathi Edem. 2017. Survey on recurrent neural network in natural language processing. *Int. J. Eng. Trends Technol* 48 (2017), 301–304.

[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[33] Xinyu Wang, Jingxian Huang, and Kewei Tu. 2019. Second-Order Semantic Dependency Parsing with End-to-End Neural Networks. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 4609–4618.

[34] Jiudong Yang and Jianping Li. 2017. Application of deep convolution neural network. In *2017 14th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)*. IEEE, 229–232.

[35] Yuxiao Ye and Simone Teufel. 2021. End-to-end argument mining as biaffine dependency parsing. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*. 669–678.

[36] Yu Zhang. 2020. *SuPar*. https://github.com/yzhangcs/parser

[37] Yue Zhang, Wei Jiang, Qingrong Xia, Junjie Cao, Rui Wang, Zhenghua Li, and Min Zhang. 2019. SUDA-Alibaba at MRP 2019: Graph-Based Models with BERT. In *Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 Conference on Natural Language Learning*. Association for Computational Linguistics, Hong Kong, 149–157. https://doi.org/10.18653/v1/K19-2014

A SUPPLEMENTARY INFORMATION FOR *SYSTEM DEVELOPMENT AND IMPLEMENTATION*: A LIST OF DEVELOPED CLASSES

See Section 4 for accompanying text.

This list depicts the codebase structure. The original codebase documentation can be found here [6]. Unused classes are retained in the codebase. Those marked with:

- * **(asterisk):** Original code, heavily adapted for our parser implementation.
- ** **(double asterisk):** Our developed files.
- *** **(triple asterisk):** Code provided by our supervisor.
- Those not marked are either unused or used but unedited.

General Classes	cmds	models	modules	parsers	structs	utils
__init__.py	*__init__.py*	*__init__.py*	*__init__.py*	*__init__.py*	*__init__.py*	*__init__.py*
*biaffine_sdp.py**	*aj_con.py*	*const.py*	*affine.py*	*const.py*	*chain.py*	*common.py*
*data_conversion.py***	*biaffine_dep.py*	*dep.py*	*dropout.py*	*dep.py*	*dist.py*	*config.py*
*epoch_eval.py***	*cmd.py**	*model.py**	*gnn.py*	*parser.py**	*fn.py*	*data.py**
*eval.py***	*crf_con.py*	*sdp.py**	*lstm.py*	*sdp.py**	*semiring.py*	*embed.py*
*extract − convert − mrs.py****	*crf_dep.py*		*mlp.py*		*tree.py*	*field.py**
*main_predict.py***	*crf2o_dep.py*		*pretrained.py**		*vi.py*	*fn.py*
*parse − convert − mrs.py****	*vi_con.py*		*transformer.py*			*logging.py*
*semantics.py****	*vi_dep.py*					*maxtree.py***
setup.py	*vi_sdp.py*					*metric.py*
*syntax.py****						*optim.py*
						parallel.py
						tokenizer.py
						*transform.py**
						vocab.py

Table 3: List of Classes

[6] https://parser.yzhang.site/en/latest/index.html

B SUPPLEMENTARY INFORMATION FOR *EXPERIMENT DESIGN AND EXECUTION*: EXPERIMENTATION RESULTS

See Section 4.5 for accompanying text.

Epochs	2	3	4	5	10	15	20	25	30	35	40
Training Edge Loss	0.0395	0.0183	0.0132	0.0105	0.0050	0.0030	0.0018	0.0011	0.0007	0.0004	0.0002
Validation Edge Loss	0.2229	0.0799	0.0508	0.0356	0.0120	0.0065	0.0035	0.0020	0.0010	0.0004	0.0002
Training Label Loss	0.0172	0.0110	0.0093	0.0085	0.0078	0.0079	0.0083	0.0090	0.0104	0.0111	0.0124
Validation Edge Loss	0.0857	0.0578	0.0593	0.0533	0.0544	0.0606	0.0596	0.0680	0.0720	0.0716	0.0737

Table 4: Cross Entropy Loss Results for the BERT Module

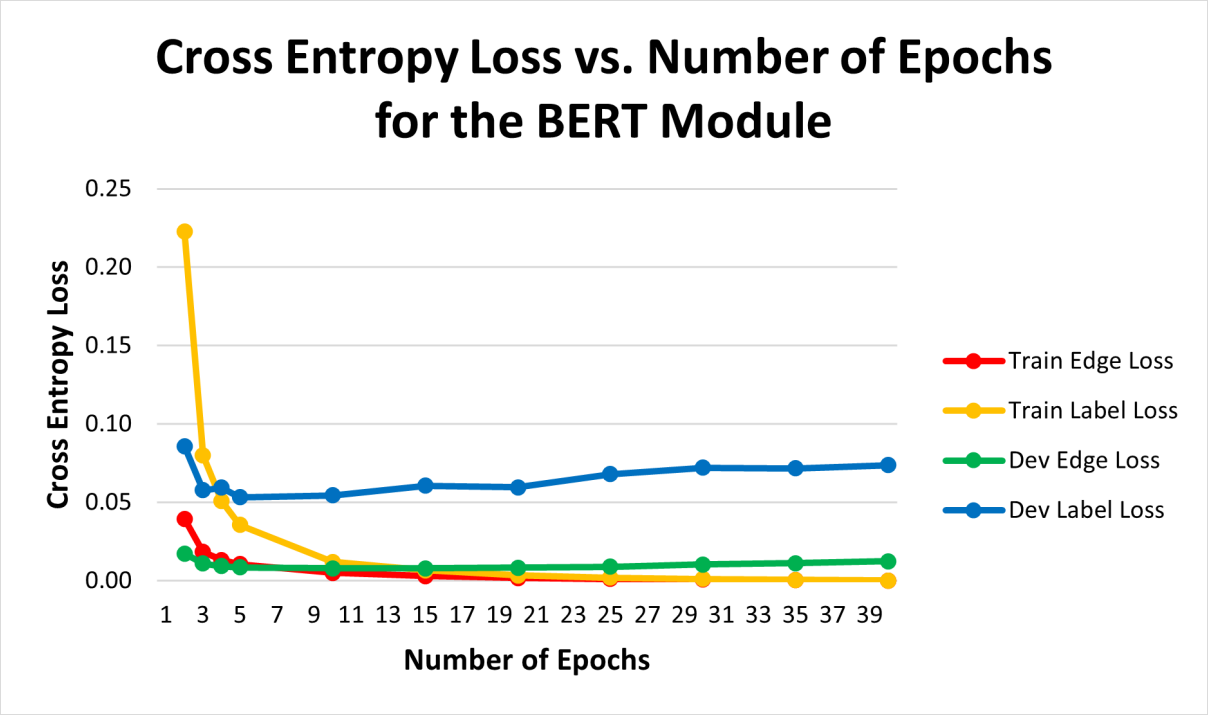

Figure 7: Loss vs. Number of Epochs for the BERT Module (BERT, Biaffine, Maximum Entropy Loss)

Epochs	2	3	4	5	10	15
Training Edge Loss	0.2432	0.2173	0.2155	0.2164	0.2172	0.2151
Validation Edge Loss	1.7840	1.7694	1.7689	1.7645	1.7705	1.7670
Training Label Loss	0.1717	0.1717	0.1717	0.1717	0.1717	0.1717
Validation Edge Loss	1.6194	1.6194	1.6194	1.6194	1.6194	1.6194

Table 5: Cross Entropy Loss Results for the BiLSTM Module

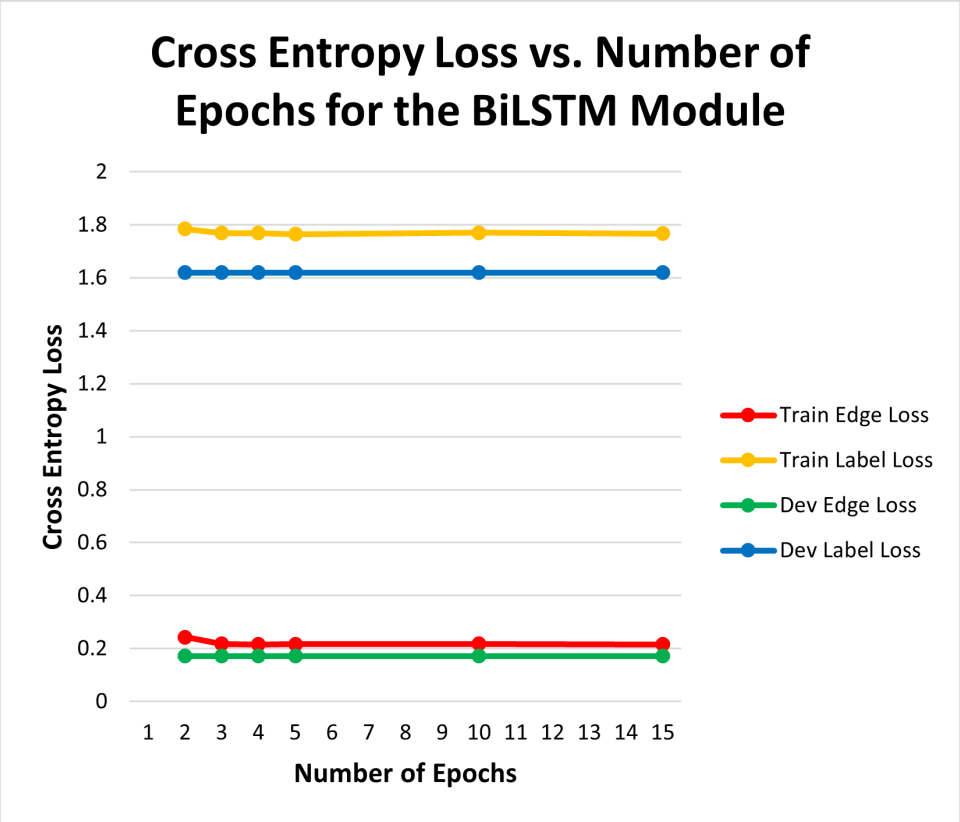

Figure 8: Loss vs. Number of Epochs for the BiLSTM Module (GloVe Embeddings, Character Embeddings, BiLSTM, Biaffine, Maximum Entropy Loss)

Epochs	2	3	4	5	10	15	20	25	30	35	40
Precision (%)	93.54	94.80	95.69	95.98	96.67	97.22	97.47	0.0057	97.45	97.56	97.76
Recall (%)	91.64	93.12	94.21	94.57	95.12	95.73	95.90	0.0206	96.03	95.99	96.16
F1 Score (%)	92.39	93.80	92.39	93.80	94.81	95.16	95.81	96.42	96.62	96.72	96.91

Table 6: F1 Scores on Validation Data for BERT Module per Number of Epochs

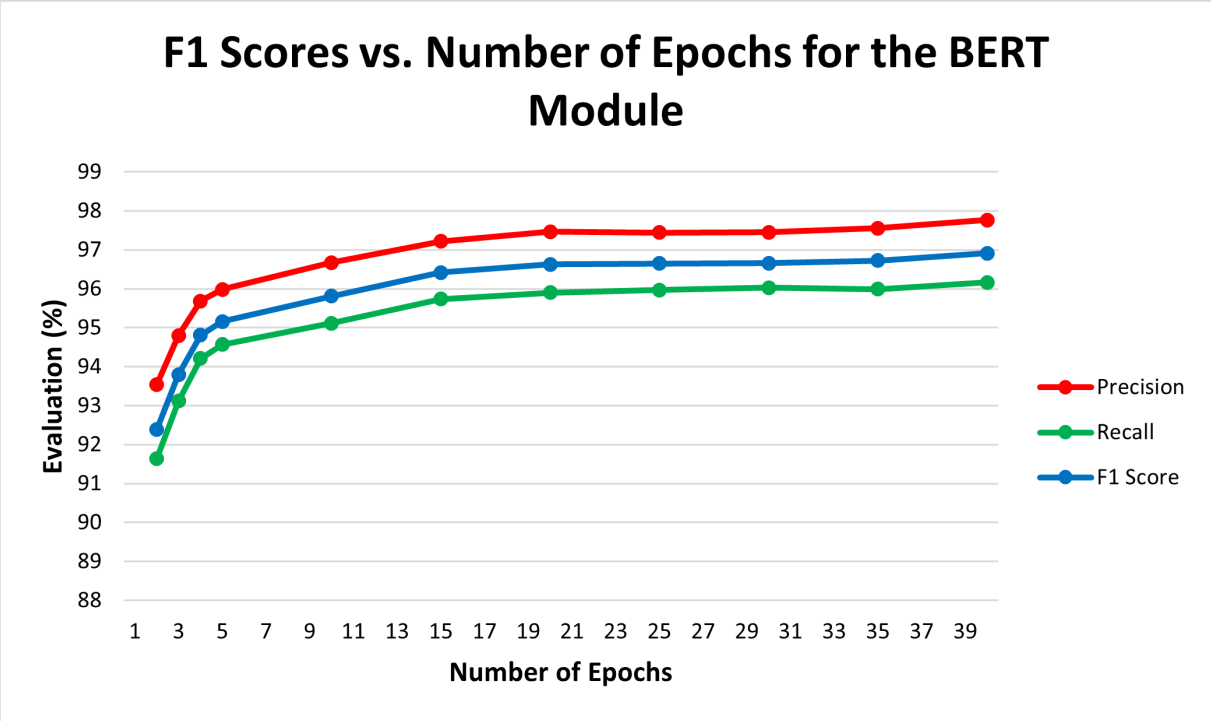

Figure 9: F1 Score vs. Number of Epochs for the BERT Module (BERT, Biaffine, Maximum Entropy Loss)

C SUPPLEMENTARY INFORMATION FOR *SYSTEM DEVELOPMENT AND IMPLEMENTATION:* MODULE PARAMETERS

See Section 4.5 for accompanying text.

Parameter	Value	Additional Parameters	Value
mode	train	n_words	28996 (BERT) ; 9060 (BiLSTM)
path	./bert_model_20 (BERT) ; ./lstm_model_20 (BiLSTM)	n_tags	None
device	-1	n_chars	None (BERT) ; 87 (BiLSTM)
seed	1	n_lemmas	None
threads	16	n_embed	100
workers	0	n_pretrained	125
cache	False	n_feat_embed	100
binarize	False	n_char_embed	50
amp	False	n_char_hidden	400
lr	5e-05 (BERT) ; 5e-03 (BiLSTM)	char_pad_index	None (BERT) ; 0 (BiLSTM)
lr_rate	20	char_dropout	0.33
warmup	0	elmo_bos_eos	(True, False)
warmup_steps	1	elmo_dropout	0.5
mu	0.9	n_bert_layers	4
nu	0.999	mix_dropout	0.0
eps	1e-08	bert_pooling	mean
weight_decay	0	bert_pad_index	None
decay	0.95	finetune	False
decay_steps	1	n_plm_0	0
num_epochs	20	embed_dropout	0.2
loss_type	entropy	encoder_dropout	0.33
feat	None (BERT) ; ['char'] (BiLSTM)	pad_index	0
build	True	n_labels	10
checkpoint	False	elmo	original_5b
encoder	bert (BERT) ; lstm (BiLSTM)	n_encoder_hidden	768 (BERT) ; 1200 (BiLSTM)
max_len	None	n_encoder_layers	3
buckets	32	n_edge_mlp	600
train	lustre/data/train.conllu	n_label_mlp	600
dev	lustre/data/dev.conllu	edge_mlp_dropout	0.25
test	lustre/data/test.conllu	label_mlp_dropout	0.33
embed	glove-6b-100	interpolation	0.1
n_embed_proj	125	unk_index	100 (BERT) ; 1 (BiLSTM)
bert	SpanBERT/spanbert-base-cased	min_freq	7
		fix_len	20
		local_rank	0
		bos_index	101 (BERT) ; 2 (BiLSTM)
		form	*
		reload	False
		src	github
		batch_size	5000
		update_steps	1
		clip	5.0
		patience	100
		verbose	True

Table 7: Full Parameter List for Both BERT and BiLSTM Modules (Including Original Codebase Defaults)

* ((words): SubwordField(vocab_size=28996, pad=[PAD], unk=[UNK], bos=[CLS]), None, None, None) (BERT) ;
((words): Field(vocab_size=402247, pad=<pad>, unk=<unk>, bos=<bos>, lower=True), (chars): SubwordField(vocab_size=87, pad=<pad>, unk=<unk>, bos=<bos>), None, None) (BiLSTM)

D SUPPLEMENTARY INFORMATION FOR *RESULTS, FINDINGS AND CONCLUSIONS*: ADDITIONAL RESULTS

These results do not aid in answering the research questions. However, they are necessary to include. See Section 5 for accompanying text.

Accuracy Scores (%)	Precision	Recall	F1 Score
With Spans	96.36	94.63	**95.50**
Without Spans	6.04	64.36	9.92

Table 8: F1 Results on Test Data for the BERT Module (10 Epochs) With and Without the Span Adaptation

Accuracy Scores (%)	Precision	Recall	F1 Score
Pretrained BERT	72.08	0.95	**1.11**

Table 9: F1 Results on Validation Data for BiLSTM (20 Epochs)

Accuracy Scores (%)	Precision	Recall	F1 Score
Pretrained BERT	74.01	0.81	**1.21**

Table 10: F1 Results on Test Data for BiLSTM (20 Epochs)