A Detailed Analysis of Semantic Dependency Parsing with Deep Neural Networks

En detaljerad analys av semantisk dependensparsning med djupa neuronnett

Daniel Roxbo

Supervisor: Robin Kurtz
Examiner: Marco Kuhlmann
Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår. Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art. Upphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydda mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart. För ytterligare information om Linköping University Electronic Press se förlagets hemsida [http://www.ep.liu.se/].

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page [http://www.ep.liu.se/].

© Daniel Roxbo
The use of Long Short Term Memory (LSTM) networks continues to yield better results in natural language processing tasks. One area which recently has seen significant improvements is semantic dependency parsing, where the current state-of-the-art model uses a multilayer LSTM combined with an attention-based scoring function to predict the dependencies.

In this thesis the state of the art model is first replicated and then extended to include features based on syntactical trees, which was found to be useful in a similar model. In addition, the effect of part-of-speech tags is studied.

The replicated model achieves a labeled $F_1$ score of 93.6 on the in-domain data and 89.2 on the out-of-domain data on the DM dataset, which shows that the model is indeed replicable. Using multiple features extracted from syntactic gold standard trees of the DELPH-IN Derivation Tree (DT) type increased the labeled scores to 97.1 and 94.1 respectively, while the use of predicted trees of the Stanford Basic (SB) type did not improve the results at all. The usefulness of part-of-speech tags was found to be diminished in the presence of other features.
Acknowledgments

I would like thank Marco Kuhlmann for providing the thesis opportunity and also for the valuable feedback received during the half time assessment. I would also like to thank my supervisor, Robin Kurtz, for the many enlightening discussions which took place during this thesis.
Contents

Abstract iii
Acknowledgments iv
Contents v
List of Figures vii
List of Tables viii

1 Introduction 1
  1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
  1.2 Aim . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
  1.3 Research Questions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
  1.4 Delimitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

2 Theory 4
  2.1 Dependency Parsing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
  2.2 Neural Networks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
  2.3 Recurrent Neural Networks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
  2.4 Word Representations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
  2.5 Regularization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
  2.6 Predicting Semantic Dependencies Using Syntactic Features . . . . . . . . . . 16
  2.7 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

3 Method 17
  3.1 Execution environment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
  3.2 Dataset . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
  3.3 Features . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
  3.4 Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
  3.5 Differences to the Dozat and Manning Model . . . . . . . . . . . . . . . . . . 25
  3.6 Analyzing Architecture Choices . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
  3.7 Extending the Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
  3.8 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

4 Results 31
  4.1 Replicated Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
  4.2 Extended Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

5 Discussion 35
  5.1 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
  5.2 Method . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
  5.3 The Work in a Wider Context . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
List of Figures

1.1 Semantic and syntactic differences ........................................ 2
2.1 Semantic Dependency Graph .................................................. 6
2.2 Feedforward Neural Network .................................................. 7
2.3 Activation Functions ............................................................. 8
2.4 Recurrent Neural Network ...................................................... 10
3.1 SDP Flavors ........................................................................ 18
3.2 Sentence Lengths .................................................................. 19
3.3 DM Label Frequency ............................................................. 20
3.4 Feature Illustrations .............................................................. 21
3.5 Model Overview .................................................................. 22
3.6 SimpleConcat ....................................................................... 27
3.7 FNNConcat ......................................................................... 28
3.8 InputConcat ......................................................................... 29
4.1 Training Progress .................................................................. 31
4.2 Performance on Different Sentence Lengths ......................... 32
List of Tables

3.1 Network Hidden Sizes .................................................. 23
3.2 Hyperparameters ..................................................... 24
3.3 Dropout Rates .......................................................... 25
3.4 Replication Deviations ............................................... 25
4.1 Replication Results ................................................... 32
4.2 Concatenation Method ............................................... 33
4.3 Addition of Syntactic Gold Dependencies ......................... 33
4.4 Addition of Syntactic Predicted Dependencies ..................... 33
Natural language processing (NLP) is currently one of the biggest, and perhaps most important fields in machine learning. Due to its success, a wide variety of technologies have been made available to the everyday consumer. Speech recognition, automated text translation, text summarization and recommender systems are all examples of useful NLP applications. One of the core tasks in NLP, which is used as a predecessor in many other tasks, is dependency parsing, where the relationships between the words in a sentence is modelled. This thesis will take a closer look at one form of dependency parsing, semantic dependency parsing (SDP) (Oepen et al., 2015) and evaluate ways to do it.

Semantic dependencies aim to model the semantic meaning between the words in a way which both humans and computers can understand. This can be done by forming a directed acyclic graph where the vertices represent the words and the edges represent the dependencies between them. The creation of these graphs has traditionally been done using a combination of different algorithms, while more recent approaches tend to be using end-to-end deep neural networks (DNNs). The types of DNNs used are commonly variants of recurrent neural networks (RNNs), which are specializing on sequence based data.

The accuracy of the semantic dependency parsers are constantly improving as new techniques emerge and older techniques are refined. Kiperwasser and Goldberg (2016) successfully applied a RNN-variant called bidirectional long short term memory to a problem similar to semantic dependency parsing, syntactic dependency parsing. Syntactic dependency parsing focuses on the grammatical structure of the sentence while semantic dependency parsing attempts to model the meaning of the sentence. While the grammatical structure can be modelled by a tree, the semantic requires a more relaxed graph structure as shown in Figure 1.1.

Inspired by the success of Kiperwasser and Goldberg (2016), Dozat and Manning (2017) extended this work by tuning the hyperparameters of the neural network used and including a technique called biaffine attention in the edge classification stage. This work has since been extended to semantic dependency parsing (Dozat and Manning, 2018), achieving new state of the art results. While not being quite as effective as Dozat and Manning (2018), Peng et al. (2018) found that incorporating features from syntactic dependency trees helped them predict semantic dependencies. In this thesis the parser by Dozat and Manning (2018) is replicated and it is investigated whether the use of these syntactic features can raise the state of the art even higher.
An unfortunate aspect of the concise format in which scientific studies are expressed is that some of the architecture choices are not always properly motivated. When Dozat and Manning (2018) followed up their syntactic dependency parsing study (Dozat et al., 2017) they changed the embedding dropout scheme without neither theoretical nor empirical justification. In (Dozat et al., 2017) they replaced their dropped embeddings with zero-vectors, which is a common approach suggested by Gal and Ghahramani (2016). In (Dozat and Manning, 2018) they instead replaced the dropped embedding with a new non-zero embedding which was learned during the training of their model. Another issue with their study is that while they are studying how some of the features such as character embeddings and lemma embeddings affect performance, they are ignoring the effect of the pretrained embeddings (GloVe) and part-of-speech embeddings. Since they are blindly including them, it remains unclear to what extent these features are affecting the performance. This thesis aims to bring clarity to these issues by studying the effects of both the dropout schemes, and the part-of-speech tags.

1.1 Motivation

Forms of dependency parsing are the basis of many downstream NLP-tasks like information extraction, machine translation and question answering systems. Thus, improvements in this area may in turn improve on the downstream tasks which are relying on this area. More accurate machine translations and text summarization is of great benefit, both to companies utilizing text analytics and the everyday internet user who wants to browse the internet in a native language.

Replication studies greatly increases the reliability of the previous studies. If the results can be closely replicated, it strongly suggests that the previous study has been conducted fairly and no essential details have been omitted in the paper. Studying some of the choices made in the model architecture more carefully is of scientific value as it is currently not established whether part-of-speech tags or learned drop-embeddings (Dozat and Manning, 2018) are useful in these DNN-based semantic dependency parsers. The addition of syntactic features has been found useful in similar models (Peng et al., 2018) which makes it interesting to
see whether the same features are also useful in the current state of the art semantic dependency parser by Dozat and Manning (2018).

1.2 Aim

The main goal of this thesis is to study how much a state of the art parser in semantic dependency parsing can be improved by adding syntactic information. In doing so the parser by Dozat and Manning (2018) needs to be replicated accurately enough to render similar performance in order to establish a reliable baseline. The replicated model can then be extended with syntactic information to determine its usefulness. In addition, the architecture choices made by Dozat and Manning (2018) regarding the use of part-of-speech tags and drop-embeddings are studied in detail.

1.3 Research Questions

1. How much can the $F_1$ score of Dozat and Manning’s architecture be improved by the integration of syntactic information?

2. How much do specific design decisions in Dozat and Manning’s work such as the choice of the dropout strategy and the integration of part-of-speech tags contribute to the $F_1$ score of the model?

1.4 Delimitations

Although there are a few, rather similar forms of dependency parsing, this study will focus on semantic dependency parsing. The semantic dependency dataset used is the DM dataset presented in (Oepen et al., 2015) which is publicly available\(^1\). The syntactic trees used as features comes from the HPSG-derived DT scheme (Ivanova et al., 2012) in the case of the gold trees, and the Stanford basic scheme as predicted by the Bohnet and Nivre (2012) parser for the predicted trees.

\(^1\)http://sdp.delph-in.net/
The theory chapter begins with an overview of dependency parsing and its historical approaches. It continues covering the current techniques in dependency parsing as well as the theoretical building blocks that are used.

2.1 Dependency Parsing

In dependency parsing, a sentence is mapped to a structure which more explicitly expresses the relationships between the words in the sentence. The most common way of doing this is syntactic dependency parsing which provides dependency relations between the words. A dependency relation is a binary relation where one word, the dependent, depends on another word, the head. These dependency relations can optionally be further divided into the type of dependency, thus making them labeled dependencies. Every word must have exactly one head and can have zero or more dependents. To satisfy this requirement, an artificial word called root is usually added to each sentence which then acts as the head to the real head of the sentence.

Dependency parsing transforms a sentence into a directed graph. The graph $G = (V, E)$ contains a set of vertices $V$, representing the words and a set of edges $E$, representing the dependency relations between the words. The tokens in $V$ are the same as the words in the source sentence. Given the set of dependency relation types $R$, an edge is defined as $E \subseteq V \times R \times V$.

The two most commonly used approaches are transition-based parsing which builds the tree piece by piece by doing local predictions, and graph-based parsing which generates the tree which scores the highest according to a scoring function.

Transition-based Parsing

Transition-based parsing is a deterministic, greedy search approach based on a shift-reduce procedure. The parsing is based on the idea of picking the best transition between different configurations in an abstract machine (Kübler et al., 2009). Each configuration represents the current state of the parser.

---

1. Vertex and node are used interchangeably in this thesis.
2. Edge and arc are used interchangeably in this thesis.
2.1. Dependency Parsing

Nivre (2003) applied transition-based parsing to the task of dependency parsing, with the goal of doing the parsing more efficiently while retaining competitive accuracy with the dynamic programming algorithms which were popular at the time. The algorithm initializes two containers, a stack and a buffer with the root node and the sentence words respectively. The words are then shifted to the stack where they are assigned dependency arcs, and then reduced from the stack. The choices of transitions are made by feeding relevant parts of the current configuration (words on the buffer and stack) to a classifier, which typically is trained using machine learning. Nivre et al. (2006) demonstrated the applicability of the Support Vector Machine classifier for this task although memory-based classifiers have also been frequently used (Kübler et al., 2009).

The use of neural networks in dependency parsing was popularized when Chen and Manning (2014) showed both accuracy and speed improvements using neural networks as the classifier in a transition-based parser. Kiperwasser and Goldberg (2016) simplified the feature engineering process by applying a variant of neural networks, bidirectional long short term memory (BiLSTM) (Graves and Schmidhuber, 2005) as the classifier in a transition-based parser. Instead of using a set of hand-crafted features, BiLSTMs are able to use all previous and future words in the sentence being parsed. Despite the simplicity they showed state of the art results on accuracy.

Graph-based Parsing

While transition-based parsers rely on locally greedy predictions of the edges, graph-based parsers take a combinatorial approach where they attempt to extract the best overall tree from the set of possible trees, \( T \). By using a global approach, graph-based parsers tend to perform better on long distance dependencies in comparison to transition-based parsers. The best tree, \( t \in T \), given sentence \( S \), is assumed to be the one scoring the highest using a scoring function. As can be seen in Equation (2.1), the score of a graph (or tree) can be calculated by summing over the scores of its subgraphs (denoted by \( p \)).

\[
\text{score}(S, t) = \sum_{p \in T} \text{score}(S, p)
\]  

(2.1)

The most common way of scoring a tree is called arc-factored parsing, in which the subgraphs are broken down into the edges between the nodes. The score of the tree thus becomes the sum of the scores of all dependencies in the tree. This can be seen in Equation (2.2) where \( e \) is an edge between two nodes.

\[
\text{score}(S, t) = \sum_{e \in t} \text{score}(S, e) = \sum_{e \in t} \omega \cdot f(S, e)
\]  

(2.2)

The function \( f(S, e) \) maps the dependency to a feature representation while \( \omega \) is a weight vector which is adjusted during training. This approach was successfully used by McDonald et al. (2005a) to parse unlabeled dependencies. They used the Eisner algorithm (Eisner, 1996; Eisner and Satta, 1999) to generate the set of all possible trees, an online algorithm to train the weights and a set of hand-crafted features for the feature mapping. The use of the Eisner algorithm introduced a constraint where only a subset of trees called projective trees were generated. McDonald et al. (2005b) removed this restriction by instead using a variant of the maximum spanning tree (MST) algorithm (Chu, 1965), which also reduced the asymptotic complexity.

Recently Kiperwasser and Goldberg (2016) applied BiLSTMs as the scorer in a graph-based dependency parser. Their parser was later improved upon by Dozat and Manning (2017) who tweaked the hyperparameters of the model and used an alternative way of estimating the scores for the tree candidates. Dozat and Manning (2018) used a simplified version of the graph-based parser in Dozat and Manning (2017) without the maximum spanning tree algorithm, to the problem of semantic dependency parsing and achieved state of the art results.
2.2. Neural Networks

The use of neural networks is the foundation of the deep learning paradigm which has emerged in recent years. The technique was inspired by how the brain processes the information it receives from the environment. As the name implies it consists of a network of artificial neurons which are divided into multiple layers. Between each of the layers are weighted connections and in plain feedforward neural networks (FNN), each neuron in the previous layers is connected to all neurons in the next layer.

The Neuron

A neuron is a function which receives a number of numerical input signals, and outputs a value based on these inputs. One of the oldest and simplest variants of the neuron is the perceptron (Rosenblatt, 1958), which sums its weighted binary inputs and outputs a binary value based on if the sum exceeds a threshold. Typically the threshold is replaced with a bias such that \( \text{bias} = -\text{threshold} \). For inputs \( x = x_1, \ldots, x_n \), weights \( w = w_1, \ldots, w_n \) and a bias \( b \), the output of the perceptron is described in Equation (2.3):

\[
f(x; w) = \begin{cases} 
0, & \text{if } w \cdot x + b \leq 0 \\
1, & \text{otherwise}
\end{cases}
\]  (2.3)

Due to the binary constraints the perceptron is very limited and today more relaxed variants of the neurons are used where inputs, weights and the output can be any real number as...
2.2. Neural Networks

Figure 2.2: A feedforward neural network with three layers.

\[ f(x; w) = w \cdot x + b \]  
(2.4)

Layers

A neural network consists of multiple layers with stacked neurons. FNNs usually contain at least three layers: an input layer, a hidden layer and an output layer. A small neural network of this type is illustrated in Figure 2.2. The input layer \( x \) is assigned the values of the function input, which is then mapped to the hidden layer \( h \) using an affine transformation. The hidden layer is then mapped to the output layer via yet another affine transformation. This can be seen mathematically in (2.5), where \( V \) and \( W \) are transformation matrices and \( b_1 \) and \( b_2 \) are biases.

\[
\begin{align*}
    h &= Vx + b_1 \\
    \text{output} &= Wh + b_2 \\
    x &\in \mathbb{R}^{d_{in}}, \ V &\in \mathbb{R}^{d_h \times d_{in}}, \ W &\in \mathbb{R}^{d_{out} \times d_h}, \ b_1 &\in \mathbb{R}^{d_h}, \ b_2 &\in \mathbb{R}^{d_{out}}
\end{align*}
\]  
(2.5)

The number of neurons in the hidden layer determines how capable the network is of learning the function. If there are a lot of neurons, the network will be able to learn more complex functions, but it will also require more computational power to train and it may allow the network to pick up random noise from the training data. When combined with a non-linear activation function, a neural network with a single hidden layer can closely replicate a wide range of continuous functions. However, instead of having a single hidden layer with a lot of neurons, a common strategy is to have multiple layers with fewer neurons. This allows for more effective usage of the neurons as the neurons in the deeper layers are able to use the calculations provided by the previous layers (Montúfar et al., [2014]).

The number of neurons in the output layer depends on the task at hand. In binary classification, the goal is to predict which one of two classes is most likely given the inputs. This can be represented by a single output neuron by letting all positive values indicate one class and the negative ones the other. In practice it is useful to model the prediction as a confidence estimation. This is useful not only in an interpretation aspect, but also when training the network to compute better predictions since it gives a better measurement of how wrong or right a prediction was. Rather than letting the output neuron become an arbitrary number, the logistic function (2.6) is commonly applied, restricting the output values to the range \([0, 1]\).

The predicted value for one class is \( f(x) \), and \( 1 - f(x) \) for the other class. As both outcomes
2.2. Neural Networks

(a) Logistic  
(b) Tanh  
(c) ReLU

Figure 2.3: Three common choices of activation functions used in the context of neural networks.

are in the range $[0, 1]$ and sum to 1, they form a valid probability distribution which means that the neural network is essentially estimating the function $p(y = 1|x)$.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$  \hspace{1cm} (2.6)

In the multiclass case, the preference for each class is usually represented by having one neuron per class in the output layer. Similarly to binary classification, the network is attempting to approximate the function $p(y|x)$. However, instead of using the logistic function (which would form a multivariate distribution rather than a multinomial), the softmax function (2.7) is applied to each of the output neurons. The softmax is named from the fact that by exponentiating the outputs, the separation between the largest value and the rest is enlarged, thus being a soft form of the max function. The denominator is the sum of all $K$ exponentiated neurons in the output layer, which ensures that their individual values are in the range $[0, 1]$, and that their combined sum is one.

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}}$$  \hspace{1cm} (2.7)

**Activation Function**

To be able to learn non-linear functions a non-linearity has to be added to the neural network. The activation function is a non-linear function applied to each of the neurons in the hidden layer as shown in (2.8). The logistic function (2.6) was commonly used in the past but is now usually replaced (Goldberg, 2016) with either the hyperbolic tangent ($\text{tanh}$, (2.9)) or variants of the rectified linear unit (ReLU) (2.10), where ReLU is the most common choice (LeCun et al., 2015). The three functions are plotted in Figure 2.3 for values in the range $[-10, 10]$. Both the logistic and tanh functions are forms of sigmoid functions, which are characterized by the S-shaped curve seen in both Figure 2.3a and Figure 2.3b. The tanh function, unlike the logistic function, is centered around zero. Figure 2.3c depicts the ReLU function which in addition to performing well, also is a more accurate model of biological neurons than the previous two (Glorot et al., 2011).

$$h = \sigma(Vx + b_1)$$

$$\text{output} = Wh + b_2$$  \hspace{1cm} (2.8)

$$\sigma(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$  \hspace{1cm} (2.9)

$$\sigma(x) = \text{max}(0, x)$$  \hspace{1cm} (2.10)
2.2. Neural Networks

Loss Function

Training the neural network is done by giving the network an input, looking at the output and then skewing the weights of the network in such a way that the desired output becomes more likely in the future. This requires some kind of metric which measures how far away from the desired outcome the current network outcome is. This metric is called a **loss function** or a **cost function**. The loss function used depends on the task at hand, but is usually based on a maximum likelihood estimation (Goodfellow et al., 2016). When the outputs form a probability distribution over $K$ classes as is the case when applying the softmax function, a loss describing the difference between the predicted distribution $\hat{y}$ and the true distribution $y$ is given by the categorical cross-entropy loss (2.11).

$$L(\hat{y}, y) = -\sum_k y_k \log (\hat{y}_k) \quad (2.11)$$

Similarly in binary classification where the logistic function has been applied to the output neuron, the binary cross-entropy loss (2.12) is commonly used. When the true and predicted distributions are similar, the loss will be close to zero but as they become more different, the loss increases.

$$L(\hat{y}, y) = -y \log (\hat{y}) - (1-y) \log (1-\hat{y}) \quad (2.12)$$

Training

During training the neural network is modified in such a way that desired outcomes become more likely. More specifically, the weight matrices connecting the network layers are changed as a function of how they affected the value produced by the loss function. This is typically done by applying gradient descent based optimizers to the network, with the objective of minimizing the loss. The gradient of the loss with respect to the weights and biases can be calculated through a two phased process called backpropagation (Rumelhart et al., 1986). In the first phase, a training sample is fed to the network which then produces an output and based on the difference between the output and the desired output, a loss is calculated. The second phase calculates the partial derivatives of the loss with respect to the parameters using the chain rule.

Optimization

The change in weights is performed by an optimizer whose objective is to minimize the expected loss over the set of training samples. In the context of neural networks, the type of optimizer most commonly used is from the family of **stochastic gradient descent** (SGD) optimizers (Goodfellow et al., 2016). The gradient is the vector pointing towards the steepest ascent from a point in a function. Since the loss function should be minimized rather than maximized, SGD moves with step length $\eta$ in the opposite direction as shown in (2.13) where the superscripts represent time steps, and $\nabla_W L(\hat{y}, y)$ is the gradient of the loss function $L$ with input vectors $\hat{y}$ and $y$, with respect to the weights $W$.

$$W^{(i)} = W^{(i-1)} - \eta \nabla_W L(\hat{y}, y) \quad (2.13)$$

In practice it is computationally more efficient to calculate the average gradient over a minibatch of $m$ samples (2.14).

$$W^{(i)} = W^{(i-1)} - \frac{\eta}{m} \nabla_W \sum_{j=1}^m L(\hat{y}_j, y_j) \quad (2.14)$$

As the name implies, the samples are chosen randomly and the process is repeated until the entire training set has been used at which point one epoch is said to have elapsed. Training usually requires multiple epochs before the loss stops decreasing, and if the network has
2.3. Recurrent Neural Networks

Recurrent neural networks (RNNs) (Elman, 1990) are a variant of neural networks which specialize in modeling sequential data. This has made them very popular in the context of natural language processing where there are many natural types of sequences. The most obvious examples are perhaps how a sequence of letters form a word and a sequence of words form a sentence. A frequently studied task in natural language processing is the mapping of the words in a sentence to their respective part of speech (PoS) classes. PoS classes like verbs and nouns describe the function of the word in a sentence. Given the task of mapping words $x_{1..n}$ to classes $y_{1..n}$, a RNN models the problem as a sequence of time steps. For each time step it calculates a hidden state based on the current input data which is then fed to the next time step. In addition to the output from the previous time step, new input data is also given to the current time step as shown in Equation (2.15) and Figure 2.4. There is one weight matrix for the inputs and one for the previous hidden state, and these weights are shared across all of the time steps. The function $f$ is an activation function. In basic RNNs, the outputs $y_{1..n}$ are simply the same as the corresponding hidden state outputs $h_{1..n}$.

$$h_t = f(W_h x_t + W_{hh} h_{t-1} + b)$$ (2.15)

There are a few different ways in which information could be extracted from a sequence. There is the sequence to sequence approach, in which the words in the input sequence is mapped to another sequence (which may be of another length). This approach is commonly used in machine translation. A sequence could also be mapped to a single summarizing vector, which is an approach commonly taken in sentiment analysis. Another common sequence method is using a single vector as a starting seed and from that generate a sequence, an approach useful in automatic image caption generation.
Bidirectional RNNs

When the entire sentence is known at the classification stage (as is often the case) it makes sense to also use the information regarding the words after the word when doing the prediction, instead of just the words before it. This is done using bidirectional RNNs and typically generates significant performance gains. In bidirectional RNNs, a set of outputs are generated as described above, and yet another set of outputs are generated by repeating the same process in reverse. The outputs corresponding to the same time step are then concatenated, forming a representation that is aware of both its left and right contexts.

Deep RNNs

Similarly to how neural networks with multiple hidden layers can more easily capture advanced relationships in the data, RNNs can be stacked into multiple layers. The outputs $y_{1..n}$ of the first layer are then used as the inputs to another RNN, rather than forming the final outputs themselves. This can be done in an arbitrary amount of layers, and the outputs of the top layer are then considered the final output.

Long Short Term Memory

Although RNNs can theoretically capture dependencies over many time steps through the forwarding of the hidden state, in practice it becomes difficult due to exploding and vanishing gradients during backpropagation. To overcome some of these issues the most commonly used methods are variants of gated RNNs where different gates decide which part of the hidden state is to be forgotten and what part of the input is to be learned. The most widely used gated RNN is Long Short Term Memory (LSTM), introduced by Hochreiter and Schmidhuber (1997). LSTMs use an extra memory vector $c$, input gate $\Gamma_i$, forget gate $\Gamma_f$ and output gate $\Gamma_o$ as shown in Equation (2.16). The logistic function $\sigma$ is applied element-wise, transforming the elements in the gate vectors to the range $[0, 1]$. The gates are later applied using the element wise product $\odot$, which means that gates with high values will leave the other component largely unchanged, while gates with low values will erase most of the information.

$$
\begin{align*}
\Gamma_i &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \\
\Gamma_f &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \\
\Gamma_o &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \\
g &= \tanh(W_{xg}x_t + W_{hg}h_{t-1} + b_g) \\
c_t &= c_{t-1} \odot \Gamma_f + g \odot \Gamma_i \\
h_t &= \tanh(c_t) \odot \Gamma_o
\end{align*}
$$

The vector $g$ combines the previous hidden state with the current input and applies a non-linearity, similarly to basic RNNs. The memory cell $c_t$ is then updated by erasing parts of the previous memory cell using the forget gate $\Gamma_f$, and then adding the (hopefully) relevant parts of $g$, filtered using the input gate $\Gamma_i$. The hidden state $h_t$ is then updated as a non-linear function of the current memory cell, filtered by the output gate $\Gamma_o$. Instead of just passing the hidden state between time steps, the LSTM passes both the hidden state and a memory cell.

Attention

The use of the attention technique in natural language processing has its origin in the field of machine translation where it bridged the gap between neural machine translation (NMT) and preexisting methods (Bahdanau et al., 2014). NMT models are usually built in an encoder-decoder fashion. In the encoding phase a LSTM is supplied with a sentence for which it in each time step calculates a hidden state. Before attention, a common way to proceed was to then use the hidden state of the last time step of the encoder as input to a decoder which consists of another LSTM. The decoding LSTM could then predict the first translated word
and then use that as input to the second time step which predicts the second word and so on until the end of the sentence has been predicted. The major problem of this approach is that all of the information regarding all the words in the source sentence is concentrated into a single fixed size vector, which becomes a bottleneck in regards to the amount of information which can be passed from the encoder to the decoder.

Bahdanau et al. (2014) similarly used an encoder-decoder approach, but rather than only using the last hidden state of the encoder, a weighted average of all the source hidden states of the encoder is used. At each time step $t$, the decoder uses context information $c_t$ based on a weighted average of the hidden states $h_{1..n}$ of the encoder as seen in (2.17) where $\alpha_t$ are the weights.

\[
c_t = \sum_j \alpha_t h_j
\]  

(2.17)

The weights are supposed to represent how well-aligned the decoder at time $t$ is with the encoder at time $j$ and are calculated as shown in (2.18) where $s_{t-1}$ is the previous hidden state of the decoder. Bahdanau et al. (2014) use a simple single hidden layer feed forward neural network to calculate the score of the alignments.

\[
a_{tj} = \frac{\exp(score(s_{t-1}, h_j))}{\sum_k \exp(score(s_{t-1}, h_k))}
\]  

(2.18)

Luong et al. (2015) further refine the use of attention in NMT by using the current hidden state $s_t$ rather than the previous and considers different scoring functions as can be seen in (2.19). They find that the \textit{dot} and \textit{general} (or \textit{bilinear}) scoring methods outperform the \textit{concat} method, which was used by Bahdanau et al. (2014).

\[
\begin{cases}
  s_t^T h_j & \text{dot} \\
  s_t^T Wh_j & \text{general} \\
  W[s_t; h_j] & \text{concat}
\end{cases}
\]  

(2.19)

Variants of attention are also used in the area of dependency parsing. Kiperwasser and Goldberg (2016) uses the concatenated alignment model from Bahdanau et al. (2014) when calculating the score between the dependencies of syntactic dependency graphs. Dozat and Manning (2017) extends Kiperwasser and Goldberg (2016) by replacing the concatenated attention with biaffine attention (2.20), and report state of the art results with it. As shown in (2.20), biaffine attention consists of a bilinear part ($v_1^T U v_2$), and an affine part ($W[v_1; v_2] + b$). Biaffine attention has also been used by Dozat and Manning (2018) where they achieved state-of-the-arts results in semantic dependency parsing.

\[
score(v_1, v_2) = v_1^T U v_2 + W[v_1; v_2] + b
\]  

(2.20)

**Gradient Clipping**

Exploding gradients are common in deep neural networks such as RNNs and can cause the optimizer to take too large steps in the direction of the gradient, even if the learning rate is small (Goodfellow et al., 2016). Gradient clipping deals with this issue by scaling down the gradient if the norm of the gradient exceeds a threshold (Pascanu et al., 2013), as shown in (2.21) where $g$ is the gradient and $\nu$ is the threshold.

\[
clip(g) = \begin{cases} 
  g, & \text{if } ||g|| \leq \nu \\
  \nu \frac{g}{||g||}, & \text{otherwise}
\end{cases}
\]  

(2.21)
2.4 Word Representations

While the use of words and language comes rather natural to humans, computers need some form of numerical representations to be able to work with them. One of the simplest ways to represent a word in a way which the computer can deal with is the one-hot vector. The one-hot vector has one index for each word in the vocabulary, and contains zeros in every element except for the one corresponding to the word which it represents, where the value is one. The one-hot vector allows the computer to distinguish between words, but it also models them as independent from each other. This clearly is not a fair model as there are many ways in which words can be considered related. Synonyms are perhaps the most obvious example as each synonym can represent the same semantic meaning as its counterparts. Although not carrying the same meaning, football and hockey are similarly related by both of them sharing the concept of sports. Describing the similarity between words can be done by letting a word be defined by the contexts in which it appears. Words occurring in similar contexts are thus considered to be similar.

Word representations may either be long and sparse like the one-hot encoding, or short and dense. Dense representations are limited to a fixed number of dimensions (typically in the range of 10-1000), regardless of the vocabulary size and the words are said to be embedded into this representation. Limiting the number of dimensions has computational advantages but it also improves on word representation generalization such that similar words can have similar vectors. These advantages have made dense word vectors the type of representation which is most commonly used in NLP applications.

Vector Algebra

A numerical, vector-based word representation which does a good job of describing the relationships between words gives similar words similar vectors. If two words are very similar, then the cosine similarity of their vectors should be close to one. Similarly, other forms of word relationships may be identified by adding and subtracting vectors. For example, Mikolov et al. (2013c) found that adding the vector representations of King and Woman and subtracting the one for Man resulted in a vector similar to the vector of Queen. Comparing the relations between words with vector algebra in this manner is one form of intrinsic evaluation which can be used to quickly estimate the quality of the word embeddings.

\[ \text{similarity}(a, b) = \cos \alpha = \frac{a \cdot b}{\|a\| \|b\|} \]  

(2.22)

Common Types of Word Embeddings

A simple, straightforward way to utilize word embeddings is to create a randomly initialized matrix of size \(|V| \times d\) where \(|V|\) is the number of words in the vocabulary and \(d\) the chosen dimension of the word embedding. Each row contains a word embedding for the corresponding word. These embeddings can then be fed to a neural network as features and later be updated using backpropagation during the training of the neural network, just like any other parameter.

There is a lot of text available on the internet in various contexts and it turns out that it can be used to pre-train word embeddings which can then be used as additional features in other NLP tasks. Two widely used ways of computing these embeddings are word2vec (Mikolov et al., 2013a; Mikolov et al., 2013b) and Global Vectors (GloVe) (Pennington et al., 2014). Although collecting texts and training the embeddings on them manually is a viable option, there are pre-trained embeddings publicly available for both word2vec and GloVe.
2.4. Word Representations

Word2vec

Word2vec (Mikolov et al., 2013a; Mikolov et al., 2013b) includes two ways of computing word vectors, and was the first method to compute word embeddings on a corpus consisting of more than a billion words. The algorithms use a moving window over the text, containing both a center word and a number of neighbouring words on both sides called context words. The window is moved one position at a time and at each step a simple neural network is trained on an artificial binary classification task. The major difference between the two algorithms in word2vec is the type of classification task conducted. In the skip-gram algorithm the network is asked to predict, for each context word, if the word actually is a context word given the current center word in the moving window. In the continuous bag-of-words (CBOW) algorithm, the task is instead to predict the center word given the context words.

GloVe

Pennington et al. (2014) argues that simply moving a window across the corpus and doing local predictions is an inefficient way of using the data. They suggest that a better method (GloVe) is to use a count-based approach where all co-occurrences are first recorded in a matrix $X_{p r} \in \mathbb{R}^{V \times V}$, and based on these global counts then derive the dense word embeddings. They randomly initialize two sets of word embeddings, $w_p \in \mathbb{R}^{V \times d}$ and $w_r \in \mathbb{R}^{V \times d}$, representing a word and context word respectively. These vectors are then trained using a least squares model, where the objective is to minimize the difference between the inner product of the two vectors and the log of their co-occurrence count, for each combination of words as shown in (2.23).

$$J = \sum_{i=1, j=1}^{V} f(X_{ij})(w_p^T w_r' - \log X_{ij})^2 \quad (2.23)$$

When trained, the word and context words vectors are summed to form the final embedding. Pennington et al. (2014) evaluates the GloVe model on three common tasks in NLP, named entity recognition (NER), word similarity and word analogy. For the former two, they manage to outperform the word2vec model while also requiring less time to train.

Character Embeddings

Character embeddings assign an embedding to each character, and then combine them to form word embeddings. This has two major benefits over word-level embeddings. The vocabulary of characters is likely completely known at training time which means that any word level embedding can be generated at testing time, even if the word which is being generated has never been seen before. Secondly, character embeddings can learn to exploit morphological language structures such as prefixes and suffixes. In English for example, prefixes like im can reverse the meaning of a word as in impossible or impractical. Similarly suffixes like ing determine the form of the word as in running or writing.

Ling et al. (2015) successfully used character embeddings in part of speech tagging by feeding the character embeddings to a bidirectional LSTM. The endstates of both the forward and backward outputs were then combined to form the word embedding. Cao and Rei (2016) used attention over all LSTM outputs rather than just using the end states and demonstrated its use in syntactic analogy answering. Dozat et al. (2017) used both the attention technique combined with the end state when they achieved state-of-the-art results in syntactic dependency parsing. In semantic dependency parsing, Dozat and Manning (2018) found that adding character-based embeddings to the previous model likewise improved their performance.
2.5 Regularization

Much of the success in deep learning is due to the fact that the deep layers can use arbitrarily complex combinations of the features. As touched on in Section 2.2, this flexibility may however lead to overfitting, where the network adjusts to the noise in the training set rather than learning the general function of the population data. In addition to early stopping, most recent studies in various fields of deep learning apply other forms of regularization, where restrictions are applied to the training process in order to learn more generalizable functions.

Weight Regularization

Weight regularization prevents the weights in the network from becoming too large by adding additional conditions to the minimization problem solved by the optimizer. The general form of the loss function using weight regularization in a neural network with a single hidden layer can be seen in (2.24). The weight $W_{ij}$ is the weight between input neuron $x_j$ and hidden neuron $h_i$, the constant $\lambda$ determines the amount of regularization and the constant $\alpha$ sets the type of regularization. This essentially gives the optimizer a budget regulating the maximum sum of the weight norms. When $\lambda$ is equal to zero, no regularization occurs, but as $\lambda$ grows, the weight budget shrinks.

$$L_{reg}(\hat{y}, y) = L(\hat{y}, y) + \lambda \sum_{i,j} |W_{ij}|^\alpha \quad (2.24)$$

The weight reduction occurring due to regularization is not necessarily evenly distributed over the weights. Some weights may increase in weight while others shrink. The nature of how the weights are changing is determined by the $\alpha$ constant. The case where $\alpha = 2$ is called $L_2$ or ridge regularization and it gradually shrinks some of weights towards zero in an asymptotic fashion. The other common case is called $L_1$ or lasso regularization and occurs when $\alpha = 1$. $L_1$ regularization shrinks a subset of the weights quickly to zero. $L_1$ thus performs a form of automatic feature selection as it effectively forgets irrelevant features.

Dropout

Dropout (Hinton et al., 2012; Srivastava et al., 2014) is a relatively new regularization technique which quickly became widely adopted in the deep learning community. It combines the ideas of adding noise to model uncertainty, and combining the predictions of multiple classifiers (ensembling) by randomly dropping neurons during training. The dropout can be applied to both the input data and the neurons in the hidden layers, thus forming a new sub-network in each training step. The model is thus forced to learn to predict the output using many different subsets of the network. At testing time however, no neurons are dropped which means that in a sense an average network is used to do the predictions. An alternative way of applying dropout to the network is DropConnect (Wan et al., 2013), which drops the weighted connections between the neurons rather than the neurons.

In the context of RNNs, researchers initially struggled with applying dropout appropriately and it was believed that it was only applicable to the input and output levels (Zaremba et al., 2014; Bluche et al., 2015). However, Gal and Ghahramani (2016) found that also using dropout on the hidden states between the time steps, variational dropout, improved performance. In order to do so, they applied the dropout in a variational manner, which means that a fixed dropout mask for both the inputs and hidden states is shared between all time steps. Merity et al. (2017) showed that DropConnect similarly could be used in RNNs by applying it to the hidden-to-hidden matrix.

Gal and Ghahramani (2016) argue that since the embedding matrices are learned during training, they should be subject to dropout just like the rest of the network. This is done by temporarily randomly replacing a fraction of the rows in the embedding matrix with vectors...
containing zeros. A variant of embedding replacement is to instead of using zeros, replace the rows with another token, which was used by Dozat and Manning (2018).

### 2.6 Predicting Semantic Dependencies Using Syntactic Features

The idea of using information from syntactic trees to predict semantic graphs was recently explored by Peng et al. (2018). They first ran a syntactic parser over the data to learn the syntactic dependencies, and then incorporated the syntactic information into the BiLSTM-based NeurboParser (Peng et al., 2017) when predicting the semantic dependencies. By concatenating the corresponding syntactic head representation to each word, they were able to improve the $F_1$ score achieved by their model.

### 2.7 Evaluation

The metric of interest in semantic dependency parsing is the $F_1$ score (Oepen et al., 2015) which is defined as the harmonic mean between the metrics recall and precision, as shown in (2.25)

$$F_1 = \left( \frac{\text{recall}^{-1} + \text{precision}^{-1}}{2} \right)^{-1} = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$  

(2.25)

Precision is defined in (2.26) and measures how often the made prediction is correct. That is, out of all the predictions made, how often is the predicted outcome the same as the true outcome. Recall, defined in (2.27), instead measures how often the prediction captures the true outcome. If in the true outcomes, a class appears 100 times and an algorithm is able to predict it correctly in 25 of these cases, then the recall is $25/100 = 0.25$. If the algorithm in total predicted the class 50 times, then the precision would be $25/50 = 0.5$.

$$\text{precision} = \frac{\text{TruePositive}}{\sum \text{PredictedPositive}}$$  

(2.26)

$$\text{recall} = \frac{\text{TruePositive}}{\sum \text{ConditionPositive}}$$  

(2.27)
This chapter describes the steps taken to replicate the model by Dozat and Manning (2018), and the extensions made. Section 3.1 describes the computational environment used during the experiments along with the tools being used. Section 3.2 covers the details regarding the dataset including the different subsets which are used during evaluation. The process of replicating the model used by Dozat and Manning (2018) is described in Sections 3.3-3.4. The known deviations from their model are described in 3.5. Section 3.7 details the attempts to improve the system. Section 3.8 covers the evaluation steps.

3.1 Execution environment

Most of the experiments are executed in virtual machines provided by Colaboratory. Colaboratory is a Jupyter Notebook based environment where users can execute their code remotely. Some experiments are executed on preemptive instances of virtual machines from Google Cloud Platform instead, as this enables the execution of multiple experiments simultaneously. The preemptive distinction implies that the training may be interrupted, but when that is the case it is later resumed by loading the most recently saved model. The models are saved after each completed epoch of training. All experiments are executed on a single Tesla K80 GPU regardless of the platform used.

The code is written using PyTorch version 1.0, a deep learning framework in Python supporting both CPU and GPU computations.

3.2 Dataset

The semantic dependency parsing dataset used in this thesis comes from SemEval 2015 Task 18 (Oepen et al., 2015), which is based on the Wall Street Journal section of the Penn Treebank. The data consists of three different sub-datasets, DELP-IN MRS-Derived Bi-Lexical Dependencies (DM), Enju Predicate-Argument Structures (PAS) and Prague Semantic Dependencies (PSD), where each sub-dataset has its own way of defining the semantic dependencies.

---

1https://colab.research.google.com
2https://jupyter.org/
3https://cloud.google.com/
4https://pytorch.org
3.2. Dataset

Each of the datasets contains for each sentence a semantic dependency graph, where a node corresponds to a word in a sentence. In addition to the word form, the node also contains additional information such as a part-of-speech (PoS) tag, a lemmatized version of the word and a binary indicator which denotes if the word is a top node. When a word is lemmatized it is reduced to its lemma, or dictionary form. The existence of a top indicator has different implications across the different datasets, but it implies that the node is either a head or a root. All subsets cover the same set of sentences, but the PoS tags, lemmatized versions of the words and top annotations are sometimes different. The lengths of the sentences are illustrated in Figure 3.2. Figure 3.1 shows the different ways the dependencies of the sentence “Imports were at $50.38 billion, up 19%.” are annotated.

**DELP-IN MRS-Derived Bi-Lexical Dependencies (DM)**

The DM dataset is based on an annotation of the Wall Street Journal (WSJ) called DeepBank (Flickinger et al., 2012), which was created using hand-written grammar rules and manual disambiguation. The semantic dependencies in DeepBank were converted to Elementary Dependency Structures (Oepen and Lønning, 2006) and then bi-lexical form (Ivanova et al., 2012). Top nodes in DM indicate the highest scoping head which is not a quantifier (Oepen...
3.2. Dataset

Figure 3.2: A histogram of the sentence lengths occurring in the training dataset. The distribution is centered around the mean length of 22.5 words with a standard deviation of 10.2.

et al., 2015). Figure 3.1a shows a sentence in the DM representation. The DM dataset contains 59 unique dependency types, although only a few are used frequently as shown in Figure 3.3. The DM dataset is the only SDP dataset publicly available, and will thus be the only SDP dataset used in this thesis.

Predicate-Argument Structures (PAS)

The PAS data is based on the Enju HPSG tree-bank. The top nodes in PAS indicate a semantic head. Figure 3.1b shows a sentence in the PAS representation.

Prague Semantic Dependencies (PSD)

PSD is based on the Prague dependency treebank, which contains semantic dependencies for both the WSJ and a Czech translation of WSJ. The tectogrammatical annotation layer was used to extract the bi-lexical dependencies. Top nodes indicate main verbs for the most part and each sentence may have multiple top nodes. Figure 3.1c shows a sentence in the PSD representation.

Data Handling

The dataset is split into training, development and test subsets. The training set is based on sections 00-19 of the Wall Street Journal corpus, while the development and test sets are based on the 20th and 21st sections respectively. The model is trained on the training set and

\[\text{http://sdp.delph-in.net/index.php}\]
evaluated on the development set. The test set is only used for the final evaluations of the experiments.

3.3 Features

To predict the intra-sentence semantic dependencies, each of the words are converted into numeric representation based on a number of features. The features used are the word form, the lemmatized version of the word, the PoS tag, the pretrained GloVe embedding and a character-based word embedding dynamically generated for the word. All features are encoded into dense 100-dimensional vector embeddings, except for the pretrained embeddings as they already are dense vectors. Once each individual feature vector has been obtained, they are concatenated to form a final word representation.

To be able to encode the features, an initial pass is ran through the training data. In this pass, vocabularies are created for each type of feature, where all discovered feature instances are registered. For each of the vocabularies an embedding matrix is then randomly initialized, where each row corresponds to a feature instance in the vocabulary.

Words

The words used in the feature representation goes through a very simple normalization procedure where very rare features are grouped together. Concretely, all words are transformed into lower case form and all numbers are replaced by a special NUM token. All word forms and lemmas occurring less than seven times in the training data are excluded from the dictionaries. The exclusion of words from the dictionaries contributes to a problem which is en-

Figure 3.3: The ten most frequent labels in the DM train dataset.
3.3. Features

Imports were at $50.38 billion, up 19%.

(a) The tokenized sentence from the dataset.

imports were at $NUM billion, up NUM %.

(b) The normalized sentence where all words are uncased and all numbers have been replaced by the NUM token.

import were at $F_NUM billion _ up NUM % _

(c) The lemmatized sentence. In addition to normalization, the words have also been transformed into their most basic form as is illustrated by the word imports being replaced by the word import. In the lemmatized tokens of the dataset, the number replacement tokens are _generic_card_ne_ and _generic_cd_ rather than NUM and F_NUM, which have been used here due to space restrictions.

NNS VBD IN $ CD CD , RB CD NN .

(d) The sentence represented by its part-of-speech tags.

Figure 3.4: A raw sentence (#20011008) along with its modified variants from the DM subset, which are used to create the features.

countered during the test phase; there are words which are not known at training time. The solution to this problem used in this thesis, is to represent all unknown words by an explicit, artificial unknown word token in the dictionary, and then treat this unknown word as any other word. Figure 3.4a displays the raw sentence while Figure 3.4b shows the normalized version.

Lemmatized Words

Lemmatization is the task of reducing all different forms of a word to its dictionary form, or lemma. For example, run, running, ran, runs are all based on the lemma run. In addition, some basic normalization steps are performed as illustrated in Figure 3.4c. All of the datasets provides lemmatizations of the words, although they may differ between datasets as there are different methods of lemmatization.

Part-of-Speech Tags

Similarly to the lemmatized words, the datasets also include part-of-speech (PoS) tags for the words. The PoS tags of the sentence in Figure 3.4a are depicted in Figure 3.4d.

Pretrained Embeddings

As mentioned in Section 2.4, pretrained embeddings of words can be learned in auxiliary classification tasks. Both word2vec and GloVe are methods of producing such embeddings, and there are pretrained embeddings of both versions trained on large datasets publicly available for download. In this thesis, 100-dimensional GloVe embeddings trained on 6 billion words are used. Unlike the other features, the GloVe-embeddings are kept frozen during the training of the model.

Character-based Word Embeddings

All of the characters encountered in the normalized word form are stored in a character embedding table. To generate a character-based word embedding, the individual characters of a word are first translated into their embedding representations, and then used as input to

---

6Available at https://nlp.stanford.edu/projects/glove/
3.4. Model

The dependency parsing system is an end-to-end neural network which is fed the features and outputs the predicted dependencies. The model, illustrated in Figure 3.5, predicts both labeled and unlabeled dependencies using a few layers of bidirectional LSTMs, a layer of a FNN and an attention mechanism on top of that. The FNN and attention parts are split into an arc subsystem and a label subsystem, which predicts the existence of dependency and type of dependency between two words respectively. The same, shared LSTM network is used by both subsystems. The hidden and embedding sizes are shown in Table 3.1.

LSTM

The feature representations for the words in the sentences are used as inputs to a three layer deep bidirectional LSTM. The outputs from the forward and backward LSTMs are concatenated and then used as input to the next layer. Each of the LSTM layers use 600-dimensional hidden states, and together they are thus producing a 1200-dimensional output vector at each time step.

Figure 3.5: An overview of the model. The words $w_{1,5}$ are embedded into their feature representations. The model predicts labeled dependencies between all pairs of words but only assigns the ones matching the edge predictions to the output prediction. The rows in the prediction matrices represent heads, the columns dependents and the colors the type of dependency.

a unidirectional single layer LSTM. The output from the last time step is then linearly transformed to a 100-dimensional vector and constitutes the word embedding. However, rather than feeding the LSTM one character embedding at each time step, convolutions over three consecutive character embeddings are used. The use of convolutions allow the LSTM network to work with higher level subword features such as prefixes and suffixes.

To reduce the overhead from the character-based word generation, a vocabulary is generated from the set of unique words in a mini-batch. The word embeddings for the vocabulary are then generated as described above, and can then be used as a lookup table for the words in the mini-batch.

3.4 Model

The dependency parsing system is an end-to-end neural network which is fed the features and outputs the predicted dependencies. The model, illustrated in Figure 3.5, predicts both labeled and unlabeled dependencies using a few layers of bidirectional LSTMs, a layer of a FNN and an attention mechanism on top of that. The FNN and attention parts are split into an arc subsystem and a label subsystem, which predicts the existence of dependency and type of dependency between two words respectively. The same, shared LSTM network is used by both subsystems. The hidden and embedding sizes are shown in Table 3.1.

LSTM

The feature representations for the words in the sentences are used as inputs to a three layer deep bidirectional LSTM. The outputs from the forward and backward LSTMs are concatenated and then used as input to the next layer. Each of the LSTM layers use 600-dimensional hidden states, and together they are thus producing a 1200-dimensional output vector at each time step.
### 3.4. Model

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Embeddings</td>
<td>100</td>
</tr>
<tr>
<td>Char LSTM</td>
<td>1 @ 400</td>
</tr>
<tr>
<td>Char linear</td>
<td>100</td>
</tr>
<tr>
<td>BiLSTM</td>
<td>3 @ 600</td>
</tr>
<tr>
<td>Arc/Label FNN</td>
<td>600</td>
</tr>
</tbody>
</table>

Table 3.1: The hidden sizes of the model.

**FNN**

Both the arc- and the label-predicting subsystems are using a simple single layer feedforward neural network to map the recurrent outputs of the LSTM to a lower dimensional vector of 600 dimensions. Further, instead of just reducing the dimension, they are also mapped to a dependent representation and a head representation. Each recurrent output state $r_i$ is thus used as input to four different FNNs, producing four different 600-dimensional vectors as shown in Equation (3.1).

$$
\begin{align*}
    h_{arc-dep}^{(i)} &= FNN_{arc-dep}(r_i) \\
    h_{arc-head}^{(i)} &= FNN_{arc-head}(r_i) \\
    h_{label-dep}^{(i)} &= FNN_{label-dep}(r_i) \\
    h_{label-head}^{(i)} &= FNN_{label-head}(r_i)
\end{align*}
\tag{3.1}
$$

By doing this, the less important parts of the vectors can be removed such as pass by information required by the previous or next time step. This both reduces the computational complexity of the model and also decreases the likelihood of the network overfitting (Dozat and Manning, 2017).

**Attention**

To generate scores indicating dependency and type of dependency respectively between a head representation and a dependent representation, the scoring function from biaffine attention is used. The use of attention has been shown to be useful in both syntactic and semantic dependency parsing tasks (Dozat and Manning, 2017; Dozat et al., 2017; Dozat and Manning, 2018). The fact that a FNN is used to transform the recurrent outputs before the attention is applied makes it a form of deep attention (Dozat and Manning, 2017). Equation (3.2) was presented in Section 2.3 but is repeated here for convenience. The score is calculated for all combinations of head $v_i$ and dependent $v_j$.

$$
\text{score}(v_i, v_j) = v_i^T U v_j + W[v_i; v_j] + b \tag{3.2}
$$

In the arc system, the tensors $U$ and $W$ are of dimensions $\mathbb{R}^{600 \times 1 \times 600}$ and $\mathbb{R}^{1 \times 1200}$ respectively. The tensors $U$ and $W$ in the label system are of dimensions $\mathbb{R}^{600 \times c \times 600}$ and $\mathbb{R}^{c \times 1200}$ respectively, where $c$ is the number of unique labels in the training data. Dozat and Manning (2018) found that the tensor $U$ could be reduced to a diagonal tensor without hurting the performance in the case of the labeled system, but not the arc system. This approach is also adopted during this thesis as it reduces the number of trained parameters.

**Predictions**

In the arc system, the task is to do binary predictions regarding the existence of a dependency between all combinations of heads and dependents. The scoring function in Equation (3.2) produces a scalar score between a pair of dependency candidates. By letting all positive scores indicate the presence of a dependency (and all negative scores the absence of a
dependency) predictions are generated for all candidate pairs as shown in Equation (3.3).

\[
prediction_{arc}(score_{i,j}) = \begin{cases} 
1, & \text{if } score_{i,j} \geq 0 \\
0, & \text{otherwise}
\end{cases}
\]  

(3.3)

The label system exploits the fact that there can only be labeled dependencies when there are arc dependencies. Thus, the score for labeled dependencies is calculated for all predicted arc dependencies. Here the score is a c-dimensional vector rather than a scalar. The predicted label for the dependency is the label corresponding to the element which contains the highest score, as emphasized in Equation (3.4).

\[
prediction_{label}(score_{i,j}) = \arg \max_c (score_{i,j,c})
\]  

(3.4)

### Training

The loss functions used are sigmoid cross entropy for the arc sub-system and softmax cross entropy for the labeled sub-system. The label loss is only calculated for pairs of words where a gold dependency exists. Both of the subsystems are trained simultaneously by summing the arc and label losses. The sum is weighted by an interpolation constant as shown in Equation (3.5) to keep the label loss from dominating the arc loss.

\[
loss_{total} = 0.025 * loss_{label} + (1 - 0.025) * loss_{arc}
\]  

(3.5)

To train the model, a variant of the Stochastic Gradient Decent optimizer called Adam (Kingma and Ba, 2014) is used with hyperparameters $\beta_1 = 0$ and $\beta_2 = 0.95$. In addition, $L_2$ regularization is used with a small constant of $3 \times 10^{-9}$. The model is trained in mini-batches of 50 sentences for 210 epochs, and the best performing epoch according to the development data is then used to predict the test data. The hyperparameters used are shown in Table 3.2.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Epochs</td>
<td>210</td>
</tr>
<tr>
<td>Mini-batch size</td>
<td>50</td>
</tr>
<tr>
<td>Adam $\beta_1$</td>
<td>0</td>
</tr>
<tr>
<td>Adam $\beta_2$</td>
<td>0.95</td>
</tr>
<tr>
<td>Learning rate</td>
<td>$1 \times 10^{-3}$</td>
</tr>
<tr>
<td>Gradient clipping</td>
<td>5</td>
</tr>
<tr>
<td>Interpolation constant</td>
<td>0.025</td>
</tr>
<tr>
<td>$L_2$ regularization</td>
<td>$3 \times 10^{-9}$</td>
</tr>
</tbody>
</table>

Table 3.2: The hyperparameters used during training.

### Dropout

Dropout is applied in various parts of the model during training. For the parts of the network where the dropout is applied with probability $\rho$, the entries are scaled up by a factor of $1 - \rho$ to keep the expected throughput the same as in the prediction phase where dropout does not occur. The amount of dropout applied to the different parts of the model is shown in Table 3.3.

All dropout is applied in a variational (Gal and Ghahramani, 2016) manner, that is the same dropout mask is shared between all time steps in a sequence. The dropout method used on the LSTM hidden states is DropConnect (Wan et al., 2013; Merity et al., 2017).

Dropout is applied to all of the embedded features except the GloVe embedding. When dropping embedding types, rather than zeroing out rows in the embedding matrices, they
3.5 Differences to the Dozat and Manning Model

<table>
<thead>
<tr>
<th>Location</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Embeddings</td>
<td>20%</td>
</tr>
<tr>
<td>Char LSTM feedforward</td>
<td>33%</td>
</tr>
<tr>
<td>Char LSTM recurrent</td>
<td>33%</td>
</tr>
<tr>
<td>Char Linear</td>
<td>33%</td>
</tr>
<tr>
<td>BiLSTM feedforward</td>
<td>45%</td>
</tr>
<tr>
<td>BiLSTM recurrent</td>
<td>25%</td>
</tr>
<tr>
<td>Arc FNN</td>
<td>25%</td>
</tr>
<tr>
<td>Arc attention</td>
<td>25%</td>
</tr>
<tr>
<td>Label FNN</td>
<td>33%</td>
</tr>
<tr>
<td>Label attention</td>
<td>33%</td>
</tr>
</tbody>
</table>

Table 3.3: The dropout percentages in various parts of the model.

are replaced with learnable <DROP>-tokens. Since replacing rather than zeroing out keeps the expected throughput the same, no scaling is required. To improve the training speed, embedding types are dropped per mini-batch rather than per sequence.

Following Dozat et al. (2017), the character embeddings are not dropped when generating the character-based word embeddings.

### 3.5 Differences to the Dozat and Manning Model

<table>
<thead>
<tr>
<th>Type</th>
<th>Dozat &amp; Manning</th>
<th>This thesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>GloVe</td>
<td>125-dimensional transformation</td>
<td>No transformation</td>
</tr>
<tr>
<td>GloVe dropout</td>
<td>20%</td>
<td>0%</td>
</tr>
<tr>
<td>Gradient clipping</td>
<td>Not specified</td>
<td>5</td>
</tr>
<tr>
<td>mini-batch size</td>
<td>3000 tokens</td>
<td>50 sentences</td>
</tr>
<tr>
<td>Training duration</td>
<td>75000 steps</td>
<td>210 epochs</td>
</tr>
<tr>
<td>Normalization</td>
<td>Not specified</td>
<td>Lowercase, number token</td>
</tr>
<tr>
<td>Variational technique</td>
<td>Dropout</td>
<td>DropConnect</td>
</tr>
</tbody>
</table>

Table 3.4: The known differences between the reference implementation by Dozat and Manning (2018) and the one used in this thesis.

In addition to the differences highlighted in Table 3.4, there are a few more implementation specific details which may or may not be different to the system by Dozat and Manning (2018). In this thesis, a set of embedding types are dropped for each mini-batch rather than for each sentence. Another possible difference concerning the generation of character-based word embeddings is that each unique word is only generated once per mini-batch, and this embedding is then used wherever the word is used.

The main model presented by Dozat and Manning (2018) uses the factorized approach described in Section 3.4 with biaffine attention, but they also tried using an unfactorized approach and bilinear attention, and found that these approaches work equally well. Since the bilinear, unfactorized approach is simpler than the biaffine, factorized approach, this thesis first validates that these results also hold true in the replicated model, and then continues using this simpler model when extending the model with syntactic features.

### Unfactorized Approach

The factored system suggested by Dozat and Manning (2018) consists of a label system and an edge system. During training, the losses of these systems are combined with an interpolation
constant and thereby trained simultaneously. During testing the intersection of the labels predicted by the label system, and the edges predicted by the edge system is used to form the final prediction of the dependencies. The unfactorized approach removes the edge system entirely and lets the label system do all of the work. To enable the label system to do this, it is extended to include a None-label which is used to describe the lack of a dependency arc between two nodes.

3.6 Analyzing Architecture Choices

Embedding Dropout

To test if the new embedding dropout scheme used in Dozat and Manning (2018) where a dropout-token is learned provides any benefit over the more common zero-dropout used by Dozat et al. (2017) and Dozat and Manning (2017), two experiments are conducted. In the first experiment, a bilinear, unfactorized model is used and whenever a word type is dropped, its embedding is replaced by a new embedding which is learned during training. Note that this baseline model is the same model which is used throughout most of the experiments. In the zero-dropout experiment a dropped embedding is always replaced by a zero-vector, while scaling up the remaining embeddings by dividing them with $1 - \rho$, where $\rho$ is the dropout probability.

Part-of-Speech

In the DM dataset the true PoS tags are made available, and they are used in the replicated model to predict the semantic dependencies. To study the effects of PoS tags in the replicated model, the replicated bilinear unfactorized model with gold PoS tags is used as baseline. From this setting a new model is trained with PoS tags completely disabled. Yet another experiment is then conducted where the gold PoS tags are replaced by new tags, as predicted by the (Bohnet and Nivre, 2012) parser. The parser is able to predict PoS tags with 97% accuracy according to the development set. To get a more complete picture, the same experiments are repeated on a more basic version of the replicated model where both the use of character-based and lemmatized embeddings are disabled.

3.7 Extending the Model

SemEval 2015 Task 18 (Oepen et al., 2015) consists of three tracks where each track was constrained to use only certain kinds of information to predict the dependencies. As part of the task, an external dataset called companion data was released which includes syntactic trees for the data. In the open task, syntactic trees in the Stanford Basic (SB) scheme (De Marneffe et al., 2006) were supplied as predicted by a syntactic parser (Bohnet and Nivre, 2012). In the gold task, the true syntactic trees as defined by the DELPH-IN Syntactic Derivation Tree (DT) (Ivanova et al., 2012) were supplied. The DT trees, similarly to the semantic dataset DM, are derived from the LinGO English Resource Grammar (ERG) (Flickinger, 2000).

Following the success of Peng et al. (2018), this thesis incorporates features derived from the syntactic trees into the semantic parser. The effects of using either predicted or gold trees are studied independently. The way the syntactic features are used is by concatenating a token representation with the token representation corresponding to a syntactic feature. The syntactic features studied in this thesis are the head, the head of the head and the sum of the dependents.
3.7. Extending the Model

Figure 3.6: The SimpleConcat method. To the right is the syntactic tree for a sentence consisting of words $w_{1..4}$, where $w_4$ is the artificial root-word which acts as the head for the true head of the sentence. Each recurrent output is then concatenated with the recurrent output corresponding to the its syntactic head.

Adding Syntactic Head

Integrating the features can be done in a few different ways and in this thesis three ways were attempted which will be referred to as SimpleConcat, FNNConcat and InputConcat.

The SimpleConcat method was used by Peng et al. (2018) and is performed by concatenating each recurrent output $r_i$ from the BiLSTM with the recurrent output $r_j$ corresponding to the syntactic head of $r_i$. This is shown in Equation (3.6) and illustrated in Figure 3.6. The concatenated output $\hat{r}_i$ is then used as input the the feedforward networks, in the same way as $r_i$ was used in the previous model.

$$\hat{r}_i = [r_i; r_{\text{HEAD}(i)}]$$ (3.6)

Instead of concatenating the recurrent outputs immediately, FNNConcat adds a fifth FNN whose purpose is to represent the output as a syntactic head. FNNConcat is a novel idea which is based on the assumption that only part of the vector is useful when being used as a syntactic head feature. By first mapping each recurrent output to an explicit syntactic representation before concatenating it with a original FNN-output, the network can learn to exclude redundant information. The FNNConcat network is illustrated in Figure 3.7. Mathematically, this replaces Equation (3.1) with Equation (3.7). The new representation $\tilde{h}_i$ is then
3.7. Extending the Model

Figure 3.7: The FNNConcat method. Each recurrent output is passed to the four previous FNNs and the new $FNN_{syn}$. Each FNN output is then concatenated with the corresponding head from the $FNN_{syn}$ output.

$$h_{(arc-dep)}^i = FNN_{arc-dep}(r_i)$$

$$h_{(arc-head)}^i = FNN_{arc-head}(r_i)$$

$$h_{(label-dep)}^i = FNN_{label-dep}(r_i)$$

$$h_{(label-head)}^i = FNN_{label-head}(r_i)$$

$$h_{(syn-head)}^i = FNN_{syn-head}(r_i)$$

$$\hat{h}_{(arc-dep)}^i = [h_{(arc-dep)}^i, h_{\text{HEAD}(i)}]$$

$$\hat{h}_{(arc-head)}^i = [h_{(arc-head)}^i, h_{\text{syn-head}}^i, h_{\text{HEAD}(i)}]$$

$$\hat{h}_{(label-dep)}^i = [h_{(label-dep)}^i, h_{\text{syn-head}}^i]$$

$$\hat{h}_{(label-head)}^i = [h_{(label-head)}^i, h_{\text{syn-head}}^i, h_{\text{HEAD}(i)}]$$

InputConcat is the least invasive method of concatenation since it is conducted on the input features rather than the internal states of the network. This novel idea is based on the assumption that providing the syntactic information earlier in the network will allow the network to better utilize it. The syntactic head feature is thus treated like any other feature as seen in Figure 3.8.

All mentioned methods concatenate a word vector with another vector based on its syntactic head. While all of words in a sentence have valid syntactic heads, the artificial root word does not. To deal with this issue, the syntactic head of the root is set to the root itself. This is illustrated in Figure 3.8, where the root word $w_4$ is concatenated with itself.

**Additional Syntactic Features**

As will be shown in Section 4.2, **InputConcat** is the most useful way of integrating the syntactic features according to the development data. To reduce the amount of model combinations, this is the only way in which features are added in the additional experiments. In addition to just concatenating the corresponding syntactic head to the token, the syntactic head of the syntactic head is concatenated. The use of the head of the head, or grandparent, has
3.8 Evaluation

There are a few interesting ways in which the performance of the system can be evaluated. To simplify comparisons with other systems, this thesis uses the same metrics used in Dozat and Manning (2018) which are labeled and unlabeled precision, recall and $F_1$ scores on both the in-domain and out-of-domain test data. These are calculated as micro averages over the data.

In each experiment a model is trained once using a predefined random seed, and then evaluated on the in-domain and out-of-domain test data. The choice of using a single training instance is based on the limited computational resources which are available during this thesis. However, this does introduce some uncertainty regarding the performance evaluation as each model will end up in a local optimum during optimization, which may not coincide with the global optimum. To measure this uncertainty, two additional models of the best performing setting are trained using different random seeds.

Significance tests are conducted using the null hypotheses and a bootstrap estimation of the p-value following Berg-Kirkpatrick et al. (2012). The difference between a baseline experiment and an additional experiment, $\delta(x)$, on the test set, $x$, is first measured. A new test set, $x'$, of the same size is then generated by randomly sampling $x$ with replacement. If the new difference between the baseline and the additional experiment, $\delta(x')$ is greater than $2\delta(x)$ a counter, $s$, is incremented. Once this process has been repeated 100000 times, the estimated p-value is calculated as $s/100000$. The null hypotheses assumes that the difference between the experiments is zero. The p-value measures the probability of observing the difference between the experiments, given that the null hypothesis is true. Since the true difference in the test set between the experiments is $\delta(x)$, this will also be the average difference between the experiments. Thus the difference between the experiments need to exceed $2\delta(x)$ rather than...
just $\delta(x)$ when estimating the p-value above in order to compensate for the skewed average. Differences with a corresponding p-value smaller than 0.05 are considered significant.
4Results

4.1 Replicated Model

The evaluations on in-domain and out-of-domain DM test data are shown in Table 4.1. The table consists of four sections where the first section shows the performance of the full, factorized variants of the replicated model. The second and third sections are both using the unfactorized approach with bilinear attention, which is the setting used in all of the following experiments as well. The third section presents the results acquired using a more basic variant of the model where both the character-based and lemmatized embeddings are disabled. In the second and third sections, -GoldPos indicates that the use of gold PoS tags was disabled, while in +PredPoS they were replaced by their predicted counterparts. The fourth section shows the performance of the basic and full reference models as reported by Dozat and Manning (2018).

The differences between bilinear and biaffine attention and factorized vs unfactorized are very minor and not statistically significant. Similarly, the differences between the learned drop-token and the more common zero-dropout are also small. However, the difference in

\[
\text{Figure 4.1: Development } F_1 \text{ score as a function of the number of epochs.}
\]
4.1. Replicated Model

(a) The performance of the biaffine model as a function of the number of words in the sentence on the in-domain data.

(b) The performance of the biaffine model as a function of the number of words in the sentence on the out-of-domain data.

Figure 4.2: The change for multiple metrics as a function of sentence length.

Table 4.1 shows how the presence of PoS tags affects the performance. In the full model, disabling the PoS tags (-GoldPos) entirely or replacing them by predicted tags (+PredPoS) both reduces the $F_1$ score significantly. The difference in using predicted tags rather than no tags is small and not significant. However, in the basic model the predicted tags outperform no tags, and the difference is significant.

The labeled $F_1$ score on the in-domain data between the learned drop-token and zero-dropout is marginally significant with a p-value of 0.049.

Table 4.1 shows how the presence of PoS tags affects the performance. In the full model, disabling the PoS tags (-GoldPos) entirely or replacing them by predicted tags (+PredPoS) both reduces the $F_1$ score significantly. The difference in using predicted tags rather than no tags is small and not significant. However, in the basic model the predicted tags outperform no tags, and the difference is significant.

Figure 4.3 shows the change in labeled $F_1$ score on the development data as a function of the number of epochs for the biaffine model. After about 130 epochs the curve plateaus but does not decrease substantially.

Figure 4.2 shows the performance of the biaffine model on different sentence lengths, on the in- and out-of-domain datasets respectively. Short sentences are easier to predict and the difference is more pronounced on the OOD data, but the difference is not major. Also shown in the figure is the exact match (EM) metric, which measures how often the model correctly classifies all of the dependencies in a sentence. As can be seen, EM decreases when the number of words in the sentence increases.

<table>
<thead>
<tr>
<th>Model</th>
<th>ID L</th>
<th>ID U</th>
<th>OOD L</th>
<th>OOD U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Biaffine Factorized</td>
<td>93.5</td>
<td>94.2</td>
<td>89.1</td>
<td>90.2</td>
</tr>
<tr>
<td>Bilinear Factorized</td>
<td>93.6</td>
<td>94.3</td>
<td>89.0</td>
<td>90.1</td>
</tr>
<tr>
<td>Bilinear</td>
<td>93.6</td>
<td>94.2</td>
<td>89.2</td>
<td>90.2</td>
</tr>
<tr>
<td>Bilinear zero-dropout</td>
<td>93.4</td>
<td>94.1</td>
<td>89.0</td>
<td>90.0</td>
</tr>
<tr>
<td>Bilinear -GoldPos</td>
<td>93.4</td>
<td>94.0</td>
<td>88.8</td>
<td>89.8</td>
</tr>
<tr>
<td>Bilinear +PredPoS</td>
<td>93.2</td>
<td>93.9</td>
<td>88.9</td>
<td>90.0</td>
</tr>
<tr>
<td>Bilinear -char -lemma</td>
<td>92.8</td>
<td>93.7</td>
<td>87.9</td>
<td>89.4</td>
</tr>
<tr>
<td>Bilinear -char -lemma -GoldPos</td>
<td>92.0</td>
<td>93.2</td>
<td>86.2</td>
<td>88.0</td>
</tr>
<tr>
<td>Bilinear -char -lemma +PredPoS</td>
<td>92.4</td>
<td>93.5</td>
<td>87.3</td>
<td>88.9</td>
</tr>
</tbody>
</table>

Table 4.1: The labeled (L) and unlabeled (U) $F_1$ scores evaluated on the in-domain (ID) and out-of-domain (OOD) test data.
4.2 Extended Model

<table>
<thead>
<tr>
<th>Model</th>
<th>ID L</th>
<th>ID U</th>
<th>OOD L</th>
<th>OOD U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>93.6</td>
<td>94.2</td>
<td>89.2</td>
<td>90.2</td>
</tr>
<tr>
<td>SimpleConcat Syntactic Head</td>
<td>96.3</td>
<td>96.8</td>
<td>92.5</td>
<td>93.3</td>
</tr>
<tr>
<td>FNNConcat Syntactic Head</td>
<td>95.7</td>
<td>96.2</td>
<td>91.5</td>
<td>92.4</td>
</tr>
<tr>
<td>InputConcat Syntactic Head</td>
<td>96.7</td>
<td>97.3</td>
<td>93.4</td>
<td>94.2</td>
</tr>
</tbody>
</table>

Table 4.2: The effect of concatenating token representations at different locations in the model. The concatenation made is with the token representation representing the syntactic head according to the gold trees in the DT annotation.

<table>
<thead>
<tr>
<th>Model</th>
<th>ID L</th>
<th>ID U</th>
<th>OOD L</th>
<th>OOD U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>93.6</td>
<td>94.2</td>
<td>89.2</td>
<td>90.2</td>
</tr>
<tr>
<td>Head</td>
<td>96.7</td>
<td>97.3</td>
<td>93.4</td>
<td>94.2</td>
</tr>
<tr>
<td>Head + HeadHead</td>
<td>96.9</td>
<td>97.5</td>
<td>93.7</td>
<td>94.7</td>
</tr>
<tr>
<td>Head + HeadHead + DepSum</td>
<td>97.1</td>
<td>97.7</td>
<td>94.1</td>
<td>95.1</td>
</tr>
<tr>
<td>Head + HeadHead + DepSum #2</td>
<td>97.0</td>
<td>97.7</td>
<td>94.1</td>
<td>95.1</td>
</tr>
<tr>
<td>Head + HeadHead + DepSum #3</td>
<td>97.2</td>
<td>97.8</td>
<td>94.0</td>
<td>95.0</td>
</tr>
</tbody>
</table>

Table 4.3: The effects of adding different amounts of syntactical features.

<table>
<thead>
<tr>
<th>Model</th>
<th>ID L</th>
<th>ID U</th>
<th>OOD L</th>
<th>OOD U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>93.6</td>
<td>94.2</td>
<td>89.2</td>
<td>90.2</td>
</tr>
<tr>
<td>InputConcat Head + HeadHead + DepSum</td>
<td>92.8</td>
<td>93.6</td>
<td>88.2</td>
<td>89.3</td>
</tr>
<tr>
<td>InputConcat Head</td>
<td>93.4</td>
<td>94.1</td>
<td>88.9</td>
<td>90.0</td>
</tr>
<tr>
<td>SimpleConcat Head</td>
<td>93.6</td>
<td>94.2</td>
<td>89.2</td>
<td>90.1</td>
</tr>
</tbody>
</table>

Table 4.4: The effects of using syntactic features based on predictions made using the Bohnet-Nivre parser in the Stanford basics scheme.

Using DT Gold Trees

The three methods of concatenation were tested by concatenating the tokens with their syntactic heads. While all three methods improved the F1 score on the test data, they did so to different extents. Concatenating the tokens at the input level (InputConcat) was found to be more effective than concatenating the outputs of the BiLSTM-network (SimpleConcat). First mapping the BiLSTM-network outputs too a new syntactic head representation using a feedforward neural network (FNNConcat) performed worse than directly concatenating the BiLSTM-network outputs. The strongest method, InputConcat, improved the labeled in-domain (IDL) score by 3.1% and labeled out-of-domain (OODL) score by 4.2%, which constitutes a relative reduction of the error rate by 48.4% and 38.9% respectively. Including more syntactic features, such as the head of the head and the sum of the dependents pushed the F1 score even higher to 97.1 IDL and 94.1 OODL. This is a reduction of the F1 error by 54.7% and 45.4% respectively. These changes are statistically significant with a p-value lower than 0.05.

The strongest setting (Head + HeadHead + DepSum) was trained using three different random seeds, resulting in slightly different performances. However, these differences are not significant.
4.2. Extended Model

Using SB Predicted Trees

When using the Stanford basic predicted trees rather than the DT gold trees, all score improvements go away as shown in Table 4.4. When using the settings which previously produced the strongest performance (InputConcat with head, head of head and dependent sum features) with the predicted trees instead, the performance regresses compared to the base line. Using InputConcat with only the syntactic head feature likewise lowers the $F_1$-score, although not as much as when using all of the features. Using SimpleConcat with only the head feature (which was the setting successfully tested in (Peng et al., 2018)) leaves the $F_1$-score mostly unchanged compared to the baseline model.
5 Discussion

5.1 Results

Replicated Model

The full replicated model scores slightly lower than the reference model presented by Dozat and Manning (2018) on the labeled in-domain DM data, while scoring slightly higher on the out-of-domain data. However, the differences, -0.2 and +0.3, are likely within the margin of error. This indicates that the model has been correctly implemented and that the differences between the models (presented in Table 3.4), such as frozen GloVe embeddings and the use of DropConnect rather than Dropout, do not have a major effect in the full model. However, the basic replicated model (without character and lemma embeddings) implemented in this thesis strongly outperforms the corresponding reference model on both the in- and out-of-domain data, with a difference of +1.4 and +1.0. The difference between the basic and the full model is that the full model is using, in addition to the features in the basic model, character based word embeddings and lemmatized embeddings. Given that the full replicated model gains less performance from using these extra features than the corresponding reference model, a likely explanation is that the reference model is using other, or none at all, preprocessing steps. In the replicated models, the words are lowercased and all numbers are converted to a special number-token. As was shown in Chapter 3.3, these steps had also been applied to the lemmatized version of the words, available in the dataset. Assuming the reference model does not perform lowercase transformation or number replacement, the lemmatized features would provide it much useful information by grouping differently cased words. In the replicated model however, there is much more overlap between the preprocessed word and the lemmatized word, reducing the amount of information which can be extracted. Thus the addition of the extra features are less useful in the replicated models.

Model Variants

When comparing the bilinear and unfactorized models to the biaffine factorized model, it seems that they are of similar quality in terms of their ability to accurately predict semantic dependencies. This result is in line with the findings of Dozat and Manning (2018) who did the same comparison. The bilinear, unfactorized model is however a less complex system as the arc subsystem can be removed completely. Further, the unfactorized model no longer
5.1. Results

Embedding Dropout

Dozat and Manning (2018) replaced their dropped embeddings in the reference model with a learned drop-word embedding rather than replacing them with zeros. Both options were tested in this thesis and they produced similar results, although the learned embeddings performed slightly stronger. The labeled in-domain $F_1$ score was the only metric where learned embedding significantly outperformed the zero-embedding. The corresponding p-value was only marginally significant with a value of 0.049. The inconclusive result reflects the lack of consensus in the NLP field in regards to how to conduct embedding dropout in neural networks. The idea of dropping entire word embeddings was proposed by Iyyer et al. (2015). Gal and Ghahramani (2016) suggests that when using recurrent neural networks, the same set of word embedding types should be dropped for an entire sentence, and they are doing this by replacing the embeddings with zero-filled vectors. When Kiperwasser and Goldberg (2016) drop their embeddings, they replace them with the unknown-word embedding rather than zeros. This makes intuitive sense as the model then learns to use this embedding during training rather than first encountering it at testing time. Dozat and Manning (2018) does not provide any justification to why they are using a learned drop-embedding, which is only seen at training time, rather than one of the two previously mentioned methods. It should be noted that they exclude words appearing less than 7 times in the training data from their embedding matrices, thus the model is still able learn how to deal with the unknown-word embedding during the training phase.

Part-of-Speech

The usefulness of PoS tags was studied using three different settings on both the full and basic variants of the replicated model. The baseline setting used the true PoS tag as given in the DM dataset. The alternative settings either replaced the tags by new tags, predicted by the (Bohnet and Nivre, 2012) parser (+PredPos) or removed them altogether (-GoldPos). In the full model (with character and lemma embeddings), using no tags reduced the labeled in-domain and out-of-domain $F_1$ scores by 0.2 and 0.4 respectively compared to the baseline. Surprisingly, using predicted tags rather than no tags decreased the labeled in-domain score even further. When running the same experiments on the basic model, contradictory results were observed. In the basic model, the usage of predicted PoS tags improves the in-domain labeled $F_1$ score from 92.0 to 92.4, compared to the $F_1$ score of 92.8 when using the true PoS tags. Another difference in the effects of PoS tags in the basic and full model is the magnitude of the impact. In the full model, the difference between using the true PoS tags and no PoS tags is merely 0.2 $F_1$ while in the basic model it is 0.8 $F_1$. The tendency of PoS tags contributing less when additional features are added was also observed in a recent study by Smith et al. (2018), who concludes that using two of the three features pretrained embeddings, character embeddings and PoS tags gives similar results to using all of them.

Using Syntactic Trees

The use of the DT gold trees as syntactic features was very successful and produced a result on the gold track of the SemEval-2015 task 18 which is unmatched in the literature. The Input-Concat method had the greatest effect on performance, and more syntactic features resulted in stronger models. There is clearly a strong relationship between these syntactic dependency trees and the semantic dependency graphs which were predicted. However, when using Stanford basic (SB) predicted trees the results either remain the same or degrade compared to the baseline, depending on the amount of features used and method of application. Given
the experiments conducted in this thesis, it is difficult to conclude how much of the degradation is due to the differences in schemes, DT and SB, and how much is due to the inaccuracy of the syntactic predictions. Unfortunately, the gold DT trees and the predicted SB trees are the only trees which are publicly available from SemEval 2015 Task 18. While the attempts in this study failed to extract any useful information from the predicted syntactic dependencies, Peng et al. (2018) show that it is indeed possible. Their system is using a pipelined approach where a slightly altered version of the syntactic dependency parser by Kiperwasser and Goldberg (2016) is trained jointly with the semantic dependency parser (Peng et al., 2017). The syntactic dependencies are then applied using the SimpleConcat method described in Section 3.7, and provide a 0.5 – 1.4% boost (from 89.4) in $F_1$ score compared to when only using the semantic parser. Their syntactic dependencies are similarly to SB, Stanford dependencies (De Marneffe and Manning, 2008). Although it is difficult to point out the specific reason to why they are able to utilize predicted syntactic features while the model used in this thesis fails to do so, there are at least two contributing factors. They are using a different, more modern syntactic parser while the syntactic predictions used in this thesis were generated by the Bohnet and Nivre (2012) parser. Another contributor which likely has an even greater effect is the fact that their semantic dependency parser is much weaker than the model used in this thesis. While the full unfactorized replicated model with bilinear attention which has been used as baseline throughout this thesis achieves a $F_1$ score of 93.6 on the labeled in-domain DM data, their parser only scores 89.4. This gives them much more room for improvement, allowing them to utilize features which perhaps are not effective given a more powerful model. It is possible that the 0.5 – 1.4 improvement they observed using syntactic features overlaps with the 4.2 gap between the semantic parsers.

5.2. Method

Differences to the Dozat and Manning Model

As highlighted in Section 3.4, the replicated models deviate slightly in some aspects compared to the model described by Dozat and Manning (2018). Almost all of them were made on the basis of gaining a significant boost in training speed while not losing much, if anything, in performance.

GloVe Embeddings

By keeping the 100-dimensional GloVe vectors frozen during training instead of transforming them to 125-dimensional vectors, both memory and computational requirements were reduced. In addition, since the embeddings are frozen there is no need to perform embedding dropout on them.

Embedding Dropout

Embedding types were dropped over the entire batch rather than on a per sentence basis. This may have reduced the regularization effect from embedding dropout slightly since the same types of embeddings were missing over multiple sentences. However, given the mini-batch size of 50, and training data size of 33964, there were still 680 mini-batches per epoch where different embedding types were randomly dropped.

DropConnect Instead of Dropout

PyTorch, the deep learning framework used during the thesis, does not natively support recurrent dropout. This can be circumvented by creating a new LSTM implementation or using a third party implementation, but this causes a substantial slowdown compared to the native implementation since the GPU-optimizations provided by cuDNN (Chetlur et al.,
cannot be utilized. However, the weights in the native implementation can be accessed and overwritten, which enables the use of DropConnect (Wan et al., 2013; Merity et al., 2017). The DropConnect approach was used during this thesis on the hidden-to-hidden weights in the BiLSTMs. Although DropConnect has similar qualities to Dropout, it does cause the same hidden to hidden dropout mask to be the same over the batch. As mentioned in Section 5.2, this may reduce the regularization capability of DropConnect slightly.

Training

Dozat and Manning (2018) used mini-batch sizes of 3000 tokens and trained for 75000 steps with early stopping. The fact that they were batching over a fixed number of tokens rather than a number of sentences indicates that they used a different approach to dealing with the tokens and feeding them to the network.

In this thesis the network was trained for 210 epochs, after which point the best performing epoch according to the development set was used to predict the test data. To compare this with the 75000 steps used in Dozat and Manning (2018) the total amount of tokens in the training data, 765025, can be used. Given the mini-batch size of 3000, 255 steps are required to finish one epoch. When they are training for 75000 steps they are thus finishing roughly 294 epochs. Since they are training for more epochs, they could theoretically gain additional performance compared to the systems replicated in this thesis. However, given the plateaus seen in the performance graphs on the development set this difference is likely very marginal.

Replicability

A large part of this thesis was dedicated to replicate the model described by Dozat and Manning (2018). Since this was successful it indicates that both their models and the models replicated in this thesis are replicable to a great extent. As for the additional tests and improvements made in the thesis, they should be adequately described in the method chapter and also be highly replicable.

Reliability

There is randomness involved in parts of the model which among other things affects the weight initialization, batching and dropout. To ensure a replicable result, the random seed powering the random generators was fixed during the different experiments. However, this does not mean that the seed chosen is good representative of the population of random seeds. Researchers has in the past made the mistake of concluding that their model was stronger than another model based on a single run, when in reality it was not (Reimers and Gurevych, 2017). The appropriate way of dealing with this uncertainty is to randomly pick multiple seeds, and then repeat each experiment on each of the seeds, forming confidence intervals which can be used to draw conclusions based on statistical significance. Due to the limited computational resources available during this thesis, multiple runs were not possible for all models. For the strongest setting, two models were trained with different random seeds. The difference in the performances of the models was largely negligible, indicating that choice of random seed does not severely affect the model. When Dozat and Manning (2018) evaluated their results on twenty different models, the difference between the best and worst performing models was in the range \[0.3, 0.6\] \(F_1\), depending on the variation evaluated. Since the model used here largely replicates their model, it is reasonable to assume that the difference between seeds would be similar here.

Another potential problem concerns how well the test set represents the population it was drawn from. If the test set does not represent it well, invalid conclusions may be made when evaluating the performance of the models. In this thesis this problem was addressed by performing bootstrap based significance tests. Pairs of models with similar performance
scores were compared to ensure that the difference between them were statistically significant with respect to the test set. Model pairs with a p-value lower than 0.05 were deemed to have a significant difference.

Validity

Evaluation Metric

The metric used to evaluate the performance of the model is the $F_1$-score, which by far is the most common metric in the area of dependency parsing. When a model achieves a good $F_1$-score it implies that the model is performing well in regards to both recall and precision.

Evaluation Data

The dataset used to train and evaluate the model comes from the Wall Street Journal section of the Penn Treebank. The dataset is commonly used to evaluate dependency parsers and by using this dataset the model can easily be compared to other models. However, it is not necessarily a fair dataset in the aspect of evaluating semantic dependency parsing on text in general. Since the data is based on published news articles written roughly 30 years ago, the quality of the text is very high. In contrast, much of the text which is available today is written by ordinary people in an informal setting. For example, Twitter messages are restricted to a few hundred characters and are often written in a hurry which have led to frequent use of innovative abbreviations as well as spelling mistakes. Given these differences, it is unlikely that a model trained on the Wall Street Journal would perform well in the social media context. However, if training data based on social media was to emerge from future work, the model should be able to learn these new dependencies as well given the flexible nature of neural networks.

Limitations

The results have not been validated over different seeds due to the the lack of computational resources. This makes the results acquired more like indicators than the definite truth.

The experiments were only conducted on the DM dataset from the SemEval 2015 Task 18 due to the fact that this is the only dataset which is publicly available. It is however the most interesting dataset of the three since it is the most commonly used one throughout the literature.

The syntactic trees used in this thesis comes from DT gold trees and SB trees predicted by the Bohnet-Nivre parser. While it can be concluded that the DT gold trees are useful, it is not possible to conclude why the predicted SB trees were not. It may be that SB trees are less informative, or that the uncertainty introduced in the predictions neutralizes any potential benefit of using them.

Source Criticism

Most of the references used are papers published in well established journals from Association for Computational Linguistics (ACL). When sources from arXiv are used, they have been deemed credible based on a combination of factors such as who the authors were, the number of citations and when it was released. As for secondary resources, review papers by Goldberg (2016) and LeCun et al. (2015) have been used along with the book Deep Learning by Goodfellow et al. (2016).

---

1https://twitter.com/
5.3 The Work in a Wider Context

Improving the computer’s ability to interpret text has a few ethical and societal implications. The role of Semantic Dependency Parsing is not entirely established yet, thus the following text is based on the assumption that it will prove to be useful.

The Average Computer User

The average computer user may notice a better user experience when using some services based on text analysis. When asking Siri, Cortana or Google Assistant a question, the system will not only better understand the question but also be able to find a better answer and present the answer in a more human-like way. When the user buys things on the internet or uses a service the user will probably receive better product recommendations due to the product descriptions being more accurately compared and matched by the product or service provider.

Companies

Companies will be able to better understand their customers, and this is especially true for the big social media companies whose customers are providing large amounts of text. By analyzing the text more accurately they can better determine what types of advertisements are the most profitable to display to the user. Companies could also use external social media platforms such as Twitter to conduct opinion mining more accurately. Instead of sending out surveys or calling the customer, they could analyze the Twitter messages mentioning the product and automatically summarize the liked and disliked properties. The public opinion has also been found useful in other tasks such as predicting election results (Tumasjan et al., 2010) and the stock market (Bollen et al., 2011).

State Institutions

Improved text interpretation improves the means of which state actors are using to monitor their citizens. This could be used to automatically detect behavior which is deemed illegal, for example hate speech, blasphemy, insults, threats or drug distribution.
This thesis has successfully replicated the current state of the art model in semantic dependency parsing and validated its results. It has also tried to incorporate two types of syntactic trees into the prediction, showing promising results for one of them. In addition, the impact of some of the architecture choices made in the original model have been compared to alternative approaches. The results indicate that the choice of embedding dropout method, replacement token or zero-replacement, does not greatly affect the performance of the parser. It is also shown that the use of predicted part-of-speech tags are useful in less feature-rich models while having questionable impact when combined with lemma and character based embeddings.

6.1 Research Questions

1. How much can the $F_1$ score of Dozat and Manning’s architecture be improved by the integration of syntactic information?

Using the syntactic gold trees from the DT scheme greatly increased the ability of the parser to predict the semantic dependencies in the DM dataset. Concatenating words with their syntactic head at the input level increased the labeled in-domain $F_1$ score from 93.6 to 96.7. When also concatenating the head of the head and the sum of dependents the score further increased to 97.1.

In contrast to the promising results of the DT gold trees, the use of predicted trees from the SB scheme performed worse or equal to using no trees at all. The strongest setting as determined by the DT gold trees reduced the labeled in-domain $F_1$ score from the 93.6 baseline to 92.8.

2. How much do specific design decisions in Dozat and Manning’s work such as the choice of the dropout strategy and the integration of part-of-speech tags contribute to the $F_1$ score of the model?

The drop-token embedding did perform better in the experiment which was made. However, the difference was only marginally significant in the case of the labeled in-domain $F_1$ metric (+0.2) and not significant in any of the other metrics. The use of gold PoS tags always improved the performance. In the full and basic models the in-domain
6.2 Consequences

It is clear that the presence of syntactic trees can be very useful in semantic dependency parsing. However, it can not be established under what circumstances they are useful, given the results in this thesis. It is possible that they are only useful when the syntactic parser is substantially more accurate in predicting its dependencies than the semantic parser. It may also be the case that only certain types of syntactic dependency schemes are useful.

The use of predicted part-of-speech tags was consistently outperformed by the gold tags. While this is to be expected, it also implies that the current state of the art performance in semantic dependency parsing is lower than what is claimed in the literature since it relies on gold tags, which are never going to be available outside of the SemEval Task 18 competition.

6.3 Future Work

Continuing With Syntactic Features

Given more computational resources and access to the complete SemEval Task 18 dataset, it would be interesting to see if the improvements in $F_1$ scores using the syntactic gold DT trees also applies the remaining annotations of the dataset, PAS and PSD. Further, investigating how predicted DT dependencies and gold SB dependencies performs would give a more complete picture regarding what is usable and what is not.

New Architecture

The KG-model by Kiperwasser and Goldberg (2016) where words are encoded using BiLSTMs and then scored the dependencies using attention-inspired scoring functions is the foundation of most recent dependency parser models, including the one replicated in this thesis.

An interesting alternative to using BiLSTMs to encode the word representations is to use a Transformer (Vaswani et al., 2017) instead. The Transformer uses multiple layers self-attention to encode each word representation as a function of the remaining words in the sentence. In addition it incorporates layer normalization (Ba et al., 2016) and residual connections (He et al., 2016), which both are useful techniques when dealing with deep neural networks. One of the biggest benefits of using the Transformer instead of the LSTM networks is that the distance between all words becomes constant, while in the LSTM network the distance depends on where in the sentence the words are located. Thus to encode the last word in the sentence given all previous words, the LSTM does this by passing hidden states from one LSTM cell to the next, up to $n$ times where $n$ is the number of words in the sentence. This could make it difficult to capture long range dependencies since as only so much information will make it through the series of gates. Using self-attention, the transformer is able to compare each word with each other word, and based on a scoring function incorporate the appropriate amount of information from each word into the new word representation. Empirically, Vaswani et al. (2017) demonstrated new state of the arts results in machine translation tasks using the Transformer model.
New Feature

Pretrained word embeddings such as word2vec and GloVe have been used extensively in various NLP tasks. These embeddings are two examples of context free embeddings, that is the embedding of a word remains the same regardless of the context it is in. This is not optimal as a word may have a completely different meaning in different contexts. For example, the word *bank* in *I will withdraw money from my bank account.* has a different meaning than it does in *Sitting on the sand bank.*. Recently attempts have been made to create context tailored embeddings. ELMO (Peters et al., 2018) uses multiple layers of BiLSTMs to create these embeddings while BERT (Devlin et al., 2018) uses the Transformer model. Since both methods have shown to be useful in a wide range of NLP tasks, future work could try to incorporate them also in semantic dependency parsing.


Bibliography


