Building a Neural Graph-Based Dependency Parser

Jack Harding Student Number: 11623608
Bram Kooiman Student Number: 11415665
Akash Raj K N Student Number: 11617586

Abstract

In this paper, we implement a neural graph-based dependency parser inspired by those of Kiperwasser and Goldberg (Kiperwasser and Goldberg, 2016) and Dozat and Manning (Dozat and Manning, 2017). We train and test our parser on the English and Hindi Treebanks from the Universal Dependencies Project, achieving a UAS of 84.80% and an LAS of 78.61% on the English corpus, and a UAS of 91.92% and an LAS of 83.94% on the Hindi corpus.

1 Introduction

Our project revolves around the construction of a (neural) graph-based dependency parser. A dependency parser takes a sentence as input and outputs the dependency tree of the sentence (and the associated arc labels). In this section, we give a brief outline of graph-based dependency parsing.

1.1 Dependency Parsing

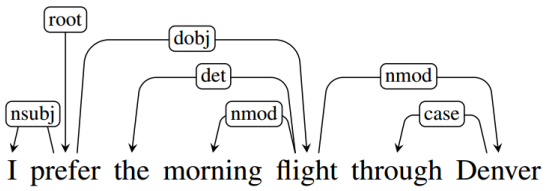
For a given sentence S, the dependency tree of S is a pair (W, A) where:

W is the union of the set of words in S and a dummy node 'ROOT', and;
A is a binary relation on W obeying certain constraints.

So each member of A (each 'arc') is a pair (word 1, word 2), where 'word 1' is referred to as the 'head' and 'word 2' is referred to as the 'dependent'. This terminology is convenient, since it conveys the intended interpretation of an arc between two words, namely that the latter grammatically depends on the former. To be a valid dependency parse, each word in S must have exactly one incoming arc (i.e. must play the role of dependent exactly once), the 'ROOT' node must have no incoming arcs, and there must be exactly one path from 'ROOT' to each word in S.

It is also common to assign to each arc in A a label, which fine-grains the grammatical relation signified by the arc.

To make this clearer, a pictorial representation of the dependency tree of the sentence 'I prefer the morning flight through Denver' is displayed below:



1.2 The Value of Dependency Parsing

Constituency parsers parse a sentence using some auxiliary linguistic structure, such as a context-free grammar. Relations between words in the sentence are then extracted from the parse tree. Since this auxiliary linguistic structure is often fine-grained, parsing can be brittle as we move across languages (or even within a single language with flexible grammatical rules). Dependency parsers, by contrast, assign grammatical relations to pairs of words in the sentence directly. Since such relations are typically more coarse-grained than the auxiliary linguistic structure used by constituency parsers, dependency parsers perform better as we move across languages (or, once more, within a single language with flexible grammatical rules).

1.3 Graph-Based Dependency Parsing

There are several approaches to dependency parsing. In this project, we focus on *graph-based* dependency parsing. The core insight of graph-based approaches to dependency parsing is to represent every possible dependency tree for a

sentence S in a very compact manner, using a connected graph (W, R, V), where:

W is defined as before;
R = W×W;
V: R → ℝ, a function which assigns a real-valued weight to each arc in R.

From the definitions above, it follows that every possible valid dependency tree will be contained within this connected graph. Assume the weight allocated to an arc between two words by the weighting function V corresponds in some way to the probability that the second word depends on the first (in this paper, we use the convention that a higher weight means that a given arc is more likely). We also follow Kiperwasser and Goldberg (2016) and Dozat and Manning (2017) in making an 'arc-factored' assumption about the weight assigned to the whole dependency tree; specifically, we assume the weight for a whole tree is the sum of the weights assigned to each arc in the tree. Then the dependency parsing problem has been transformed into a graph-theoretic problem. The best dependency parse (the most likely dependency tree) is simply the Maximum Spanning Tree (MST) of the connected graph. So:

$$parse(s) = \arg\max_{t \in T} \sum_{(h,d) \in t} weight(s, h, d)$$

where 's' is a sentence, 'h' is a candidate head for an arc to dependent 'd' and 'T' is the set of all of the sentence's possible dependency trees.

1.4 Advantages of Graph-Based Approaches

There are two key advantages of graph-based dependency parsers over many of their rivals.

Firstly, graph-based dependency parsers' outputs can be non-projective. Recall that a given arc between a head and dependent is projective if there is a path from the head to every word between the head and dependent. A given dependency tree is projective if all of its arcs are projective. A significant proportion of dependency trees of natural language sentences are non-projective (particularly in languages where word order is more flexible).[1] So a dependency parser which struggles to deliver non-projective output (like a transition-based dependency parser) will deliver the wrong output on these sentences. Since a benefit of dependency parsing over constituency parsing is that the former copes well with languages with more flexible grammars, this gives graph-based approaches an important advantage.

Secondly, graph-based dependency parsers are well placed to capture long-range dependencies in a sentence (that is, dependencies where the head and dependent have several words between them). A graph-based dependency parser makes choices at the level of whole trees; its output is the dependency tree with the highest overall weight. By contrast, a transition-based dependency parser makes choices at the level of arcs. As the number of words between a head and dependent increases, the probability that some intervening word is a plausible dependent for the head (or plausible head for the dependent), such that it will be selected by the transition-based parser as the dependent (head), increases. A graph-based dependency parser has no such problem, given a good weighting function.

1.5 Calculating Weights for Arcs

Clearly, the function which assigns a real-valued weight to each possible arc between two words plays a vital role in the success of a graph-based dependency parser. Unless the weight assigned to an arc corresponds to the probability of the arc, extracting the MST from the connected graph will be a futile task. But how do we arrive at this function? In other words, which features of an input sentence determine which dependency arcs are most likely?

Historically, these features were hand-crafted. Kiperwasser and Goldberg (2016) and Dozat and Manning (2017), though, treat the problem of extracting the relevant features from a sentence as a supervised learning task, using a neural network trained on a large annotated corpus.

2 Problem

In this section, we describe the experimental setup.

2.1 Corpora Choice

In this paper, we focus on corpora from the Universal Dependencies Project. Specifically, we train and test our model on the English and Hindi TreeBanks. Since English has a relatively rigid word order, a negligible proportion of its sentences have non-projective dependency trees. But

[1] See (Jurafsky and Martin, Forthcoming), Ch 14.2) for more discussion.

an advantage of both dependency parsers in general and graph-based dependency parsers in particular is that they outperform their competitors on languages with a more flexible grammar. For this reason, we choose to investigate the performance of our parser on the Hindi corpus, since Hindi is known to have a higher proportion of sentences with non-projective dependency trees.

We also trained and tested our model on a proper subset of each corpus. For English, we trained and tested our model on those sentences with fewer than 13 words. For Hindi, we trained and tested our model on those sentences with fewer than 18 words. The results for these 'short' data sets are included in our 'Results' section.

2.2 Pre-processing data

We convert the CoNLL-U files from the Universal Dependencies Project's English and Hindi TreeBanks into json files. Words that only occur once in the training data are replaced with '<unk>' (so small statistical irregularities do not affect our model). Words that occur in the test data but not the training data are also replaced with '<unk>', since our training is unlikely to accommodate such words. We add the dummy 'ROOT' to the beginning of each sentence in the training and each sentence in the test data, along with a dummy POS tag ('ROOT') and dummy arc label for arcs from 'ROOT' to itself ('-'). Our choice to add 'ROOT' at the start of the sentence was motivated by convenience; later, we will show that the network (specifically, the BiLSTM) takes in information about word position, so the choice of where to place 'ROOT' will not affect the final result.

2.3 The Data

For a given sentence S in the corpus, the training data included the POS tag for each word in S, the gold dependency tree and a label for each arc in the dependency tree.

3 Our Approach

In this section, we outline the core of our model, highlighting at each turn the choices we have made and reasons for these choices.

3.1 Word Embeddings

We begin by embedding each word in the training corpus. We represent each word as a 100-dimensional vector. Intuitively, a word's POS tag is relevant in determining its dependency relations (and we are given the POS tags in the testing data), so we also represent each POS tag as a 20-dimensional vector. A (word, POS tag) pair is then represented as the 120-dimensional concatenation of these two vectors. A given sentence of length n is represented as a sequence (of length n) of these concatenated vectors.

The (word, POS tag) embeddings are parameters in our model, meaning they change during training. We also choose to initialise the (word, POS tag) embeddings using Gensim's Word2Vec method. We do this for two reasons. Firstly, starting with an accurate word embedding reduces the training time. Secondly, although the Gensim word embeddings are obtained from our training corpus, they are trained independently of the task to which we apply them. So using these precomputed word embeddings helps mitigate against the risk of overfitting the word embeddings (of course, this positive effect decreases as we increase the number of training iterations).

3.2 Bi-directional LSTM

So, given a sentence S of length n, we have a sequence (of length n) of 120-dimensional vectors . Following Kiperwasser and Goldberg (2016), we feed this sequence through a Bi-directional Long Short Term Memory Network (BiLSTM).

An LSTM is a type of Recurrent Neural Network (RNN). One advantage of an LSTM over a more vanilla RNN is that an LSTM is able to capture long-range dependencies (for example, an LSTM is less susceptible to the problem of exploding/vanishing gradient). In the first section, we showed that an advantage of graph-based dependency parsers over (e.g.) transition-based parsers is that the former is better placed to capture long range dependencies. So we use an LSTM over a basic RNN to make sure this benefit is delivered.

Intuitively, a given dependent's history (the words that precede it) is not sufficient to determine its head. We also want to keep track of its future (the words that follow it). It is for this reason that we use a BiLSTM. A BiLSTM is built from a forward LSTM (LSTM_F) and a backward LSTM (LSTM_B); this means that we can take into account an arbitrarily large context 'window'[2] around each dependent.

[2] Kiperwasser and Goldberg 2016, p.316.

3.3 Head/Dependent Splitting

We have hitherto followed the approach of Kipperwasser and Goldberg (2016). At this point, though, we diverge from their model, choosing instead to follow Dozat and Manning (2017). We do this for two reasons.

Note that, for our input sentence S, the output of our BiLSTM is a sequence of length n, where each member of the sequence is a 120-dimensional vector corresponding to a contextualised (word, POS tag) pair. Kipperwasser and Goldberg assign a weight to a given (head, dependent) pair by feeding the concatenation of the head's vector with the dependent's vector through a scoring function (where the scoring function is a simple MLP). But, as Dozat and Manning note (p.3), each of these two 120-dimensional vectors contains a significant amount of redundant information. To see this, note that a lot of what each vector represents will be specific to the sentence of which it is a part. But all we want from this vector is a prediction of its head (and, if they exist, dependents), as well as the label for any arc of which it is a part. So using this redundant information won't help us; in fact, it may hinder us, by increasing training time and the risk of overfitting.

Ironically, given the observations in the previous paragraph, there seem also to be limitations on these 120-dimensional vectors' representational power. Recall that we are using only a single representation for each (word, POS tag) pair. This results in a subtle problem for Kipperwasser and Goldberg. It's true that their model scores an arc (call it 'arc 1') between 'word 1' and 'word 2' differently from an arc (call it 'arc 2') between 'word 2' and 'word 1', since concatenation is not a commutative relation. But in the training phase, the weights assigned to both 'arc 1' and 'arc 2' will — after backpropagation — affect the very same vectors, namely those corresponding to 'word 1' and 'word 2'. One might worry that this doesn't permit us enough flexibility; we might like to have a more explicit way of accounting for the two roles words can play (one for when they are heads and another for when they are dependents). This worry is exacerbated when we take into account the fact that the problem in the previous paragraph entails that the dimensions of these vectors ought to be reduced.

Dozat and Manning (2017) solve these problems in a single step. They feed each vector in the output of the BiLSTM through two different MLPs, resulting in two different output vectors for each vector in the output of the BiLSTM. Intuitively, one of these output vectors corresponds to the contextualised (word, POS tag) pair *qua* head, whilst the other corresponds to the contextualised (word, POS tag) pair *qua* dependent. In effect, the presence of these MLPs constitutes an explicit parameter (more accurately, set of parameters) for the different role each (word, POS tag) pair can play, solving our second problem. Furthermore, we can set the dimensions of the output layer of each MLP at will, solving our first problem. Following Dozat and Manning, we chose the output layer to have a dimension of 20. More formally:

$$\mathbf{h}_i^{(arc-dep)} = MLP^{(arc-dep)}(\mathbf{r}_i)$$

$$\mathbf{h}_j^{(arc-head)} = MLP^{(arc-head)}(\mathbf{r}_j)$$

where $\mathbf{r}_i$ and $\mathbf{r}_j$ are two elements of the output of the BiLSTM.

3.4 Scoring

We also use Dozat and Manning's scoring function (2017, p.3). The weight of an arc with 'j' as head and 'i' as dependent (written $score(j, i)$) is defined as follows:

$$score(j,i) = \mathbf{h}_j^{(arc-head)^T} U^{(1)} \mathbf{h}_i^{(arc-dep)} + \mathbf{h}_j^{(arc-head)^T} \mathbf{u}^{(2)}$$

where $\mathbf{h}_j^{(arc-head)}$ and $\mathbf{h}_i^{(arc-dep)}$ are defined as in the previous section, $U^{(1)}$ is a 20x20 matrix and $\mathbf{u}^{(2)}$ is a 20-dimensional column vector. Note that both $U^{(1)}$ and $\mathbf{u}^{(2)}$ are parameters of the model, meaning they will change during training.

As Dozat and Manning (p.3) observe, one advantage of this scoring function is that the (bias) term $\mathbf{h}_j^{(arc-head)^T} \mathbf{u}^{(2)}$ corresponds to the prior probability of the candidate head being a head in any arc. Intuitively, some words are simply more likely than others to be heads, and we would like the weights assigned to arcs with these words as heads to reflect this. The scoring function explicitly takes this into account.

3.5 Learning the Labels

Up to this point, we have discussed how our model predicts the weight assigned to each possible arc

between two words. As noted above, it is also useful for a dependency parser to be able to predict the label for a given arc, since the label gives us more information about the grammatical relation between the head and dependent. In this section, we discuss the part of our model devoted to predicting labels for each arc.

Before entering into the specifics of our label prediction mechanism, there is an important point to make. Unlike with our arc weighting mechanism, which assigns a weight to every possible arc in a sentence, our label weighting mechanism should train only on the gold arcs in the training corpus (and their associated labels). For it will be impossible to decide which label an arc which doesn't actually exist ought to receive; any attempt to assign a label to a non-existent arc suffers from a presupposition failure.

In the previous section, we showed that our model for arc prediction follows closely that of Dozat and Manning (2017). Dozat and Manning use another 'biaffine classifier' for predicting which labels should be assigned to arcs. Let (j, i) be an arc in the gold tree in the training data. Then the weight which is assigned to a label 'y' for an arc from 'j' to 'i' (written $label_y(j,i)$) is:

$$label_y(j,i) = \mathbf{h}_j^{(arc-head)^T} U^{(3)} \mathbf{h}_i^{(arc-dep)} + (\mathbf{h}_j^{(arc-head)} \bigoplus \mathbf{h}_i^{(arc-dep)})^T \mathbf{u}^{(4)} + \text{b}$$

where $\mathbf{h}_j^{(arc-head)}$ and $\mathbf{h}_i^{(arc-dep)}$ are defined as before, $U^{(3)}$ is a 20x20 matrix, $\mathbf{u}^{(4)}$ is a 40-dimensional column vector and 'b' is a bias scalar.

Despite this, Dozat and Manning (p.7) are disappointed with the performance of their labelling predictor relative to that of their arc predictor. With this in mind, we thought it might be interesting to try a (slightly) new architecture. One possibility is that the 'likelihood' term (or rather, its biaffine counterpart) $\mathbf{h}_j^{(arc-head)^T} U^{(3)} \mathbf{h}_i^{(arc-dep)}$ is redundant, since the term $(\mathbf{h}_j^{(arc-head)} \bigoplus \mathbf{h}_i^{(arc-dep)})^T \mathbf{u}^{(4)}$, which uses the concatenated vector $(\mathbf{h}_j^{(arc-head)} \bigoplus \mathbf{h}_i^{(arc-dep)})$, seems to contain the same information. If it were redundant, the inclusion of this term would lead to overfitting. We thought we'd make a slight modification to the architecture to test this intuition. As before, we follow Dozat and Manning in putting each candidate head through the 'head' MLP and each candidate dependent through the 'dependent' MLP, to obtain $\mathbf{h}_j^{(arc-head)}$ and $\mathbf{h}_i^{(arc-dep)}$ respectively. Rather than using Dozat and Manning's scoring function at this point, though, we put the concatenated 40-dimensional vector $(\mathbf{h}_j^{(arc-head)} \bigoplus \mathbf{h}_i^{(arc-dep)})$ through an MLP with a single hidden layer (of dimension 20). In effect, then, our label weighting function is a hybrid of that in Kipperwasser and Goldberg (2016) and Dozat and Manning (2017).

3.6 Training

For each sentence (of length n, including 'ROOT') in our training corpus, our network gives two outputs on the forward pass. Firstly, it gives an $n \times n$ adjacency matrix, where the entry at (j, i) represents the weight of an arc with head as the word indexed by 'j' and dependent as the word indexed by 'i'. Since each dependent has exactly one head, we can view the columns of this adjacency matrix as corresponding to a probability distribution over possible heads. Secondly, it outputs an $n \times l$ matrix, where 'l' is the number of labels in the training corpus. Each row of this matrix will correspond to a probability distribution over labels to gold arcs whose dependent is the word indexed by the row. So the entry at (j, i) with be the weight that the ith label attaches to the gold arc whose dependent is the jth word. Given that we have added the dummy node 'ROOT' to the beginning of the sentence, any entries in the first column of the $n \times n$ adjacency matrix and the first row of the $n \times l$ matrix will be redundant. Nevertheless, we include these entries for notational convenience.

From the gold tree for the sentence, we construct a gold $n \times n$ adjacency matrix, where the only entries with non-zero values correspond to the weight of the arcs in the gold tree. From the gold labels for each arc in the gold tree, we construct a gold nxl matrix, where the only entries with non-zero values correspond to the correct label for the arc.

To calculate the loss, we calculate the cross-entropy loss (with respect to the columns) between the predicted $n \times n$ adjacency matrix and the gold $n \times n$ adjacency matrix. We also calculate the cross-entropy loss (with respect to the rows) between the predicted $n \times l$ matrix and the gold $n \times l$ matrix. We then sum these two losses to obtain the total loss for a given forward pass of the model on this sentence, using this loss to backpropagate ac-

cordingly. Note that by combining the two losses in this way, we are essentially doing multi-task learning (we are using the very same network to predict arc and labels).

3.7 MST Algorithm

In the previous section, we described what a forward pass of the network would look like. So, when testing, we obtain (for a given sentence S of length n) an $n \times n$ adjacency matrix and an $n \times l$ labels matrix.

It is at this point that our MST algorithm comes into play. This $n \times n$ adjacency matrix is a representation of the connected graph introduced in the first section. So, from this adjacency matrix, we use an MST algorithm to extract a dependency tree. For each dependent in this dependency tree, we also label the dependent's incoming arc with the highest weighted label of the dependent's row in the $n \times l$ labels matrix.

As suggested in the project guidelines, we use the Chu-Liu-Edmonds MST algorithm to extract the dependency tree from the adjacency matrix. In this project, we implemented this algorithm from scratch, using the version of the algorithm presented in the NLP1 lecture on dependency parsing. A benefit of using this MST algorithm is that it can return both non-projective MSTs and projective MSTs. This is crucial to retaining the flexibility of graph-based dependency parsing when we parse languages with a less rigid word order (or grammar), since these languages will contain a higher proportion of sentences whose dependency trees are non-projective.

One problem arises with using an MST algorithm to find a dependency tree, namely that not all MSTs are dependency trees. Specifically, an MST can have two outward arcs from the 'ROOT' node, whilst a valid dependency tree must have exactly one outward arc from the root node.

One way to overcome this problem is simply to choose the arc from the 'ROOT' with the highest outbound score, then get the MST rooted at the dependent of this arc. This is the strategy the MST notebook we were provided with takes. The problem with this strategy is that it is not guaranteed to find the best dependency tree (for our greedy local choice might not yield the best global result).

For our MST algorithm, then, we choose instead to take a more brute force approach that is guaranteed to arrive at the optimal solution. Specifically, we check if the output of our MST algorithm has multiple arcs from the 'ROOT'. If it does, we calculate the sum of the score of the MSTs rooted at the dependent of each of these arcs and the arc from the 'ROOT' to this dependent. Our dependency tree is then the tree with the maximal overall score. Although this means that in the worst case we run the MST algorithm once for each word in the sentence being tested, this case occurs very rarely. Moreover, the lengths of the sentences are relatively small, so this additional time at testing is negligible.

Although the model MST algorithm provided in the Jupyter notebook and our MST algorithm diverge only occasionally (meaning that the model MST algorithm gives the wrong dependency tree only occasionally), this still has an impact on the scores achievable during testing time. For example, on a testing set of a thousand 20×20 matrices initialised with random weights between 0 and 1, the mean score of our MST algorithm was 18.48 (out of a maximal 19), whilst the average score of the MST algorithm in the notebook was 18.39.[3]

Parameter	Value
Word embedding size	100
POS tag embedding size	20
BiLSTM size	400
BiLSTM depth	3
Arc MLP size	500
Label MLP size	200
Weight decay	e^{-6}
β_1, β_2	0.9
Learning Rate	0.002
Embedding dropout	0.0
BiLSTM dropout	0.33
Arc MLP dropout	0.0
MLP depth	1
Number of Epochs, English Data	65
Number of Epochs, Hindi Data	30

Table 1: Model hyperparameters

4 Experiments

4.1 Hyperparameters

In the table above, we show some of our hyperparameters for the model. At most points, we have copied the choices made by Dozat and Manning.

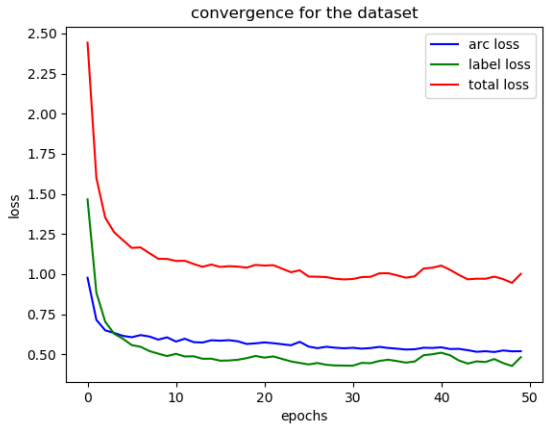
[3]The code for these comparisons can be found in MST_FINAL.py on the GitHub repository for this project. Our MST algorithm is the function 'get_edmonds'.

There are a few places where our hyperparameters differ from theirs, though.

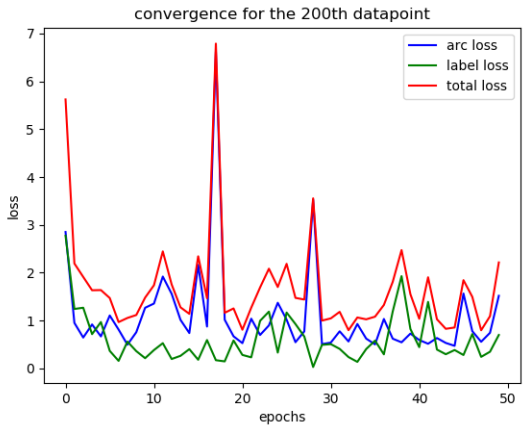
Firstly, Dozat and Manning use .33 dropout on the BiLSTM, the head/dependent MLPs and the word embeddings, whilst we only use dropout on the BiLSTM. In retrospect, this was a mistake on our part. Given the number of parameters in the Arc MLP and the word embeddings, by not using dropout our model is at danger of overfitting. This is doubtless compounded by the fact that our embeddings have 120 dimensions, whilst theirs have 100 dimensions.

4.2 Results

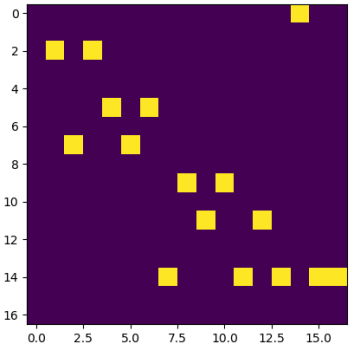
One way to visualise the effects of training is to look at the loss value. This graph shows the loss during training for the short English training data:



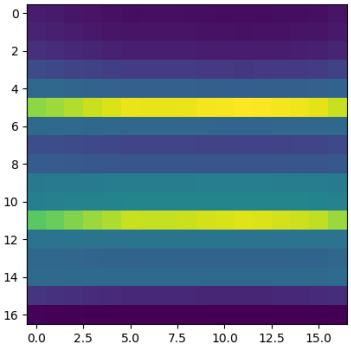
This graph shows the loss for an arbitrary sentence (number 200) in the short English data set:



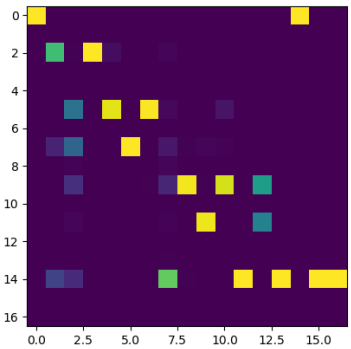
Another way to visualise the training phase is to look at how the predicted weights for the arcs in a given sentence change during training. We use a heatmap to represent these weights for a sentence in the Hindi training set (sentence 4). The heatmap for the gold tree is:



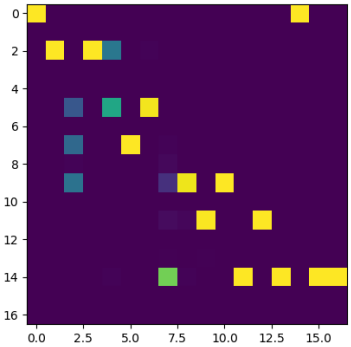
When the weights are initialised, their heatmap is as follows:



After 10 epochs, the heatmap is:



After 24 epochs, the heatmap is:



Rather than converting the results back into the

Testing Data Set	UAS score	LAS score
EN Long	84.80	78.61
EN Short (<13)	89.42	82.47
HI Long	91.92	83.94
HI Short (<18)	87.04	86.07

Table 2: UAS and LAS scores for testing data

Training Data Set	UAS score	LAS score
EN Long	83.83	78.18
EN Short (<13)	89.88	83.36
HI Long	91.85	83.81
HI Short (<18)	87.54	86.66

Table 3: UAS and LAS scores for training data

CoNLL-U format, we implemented code to test the Unlabelled Attachment Score (UAS) and Labelled Attachment Score (LAS) directly. The UAS is the proportion of arcs in the test set that were predicted correctly. The LAS is the proportion of (arc, label) pairs in the test set that were predicted correctly. We tested our model on four data sets: the full English data set, the full Hindi data set, a subset of the English data set (where each sentence has fewer than 15 words) and a subset of the Hindi data set (where each sentence has fewer than 18 words). The results are summarised in Table 2.

On the full English corpus, we achieved a UAS of 84.80% and an LAS of 78.61%. The baselines for the English corpus are 87.68% UAS and 85.82% LAS. On the Hindi corpus, we achieved a UAS of 91.92% and an LAS of 83.94%. The baselines for the Hindi corpus are 94.23% UAS and 91.07% LAS.[4]

4.3 Discussion

In order to investigate the extent to which our model overfitted the data, we also tested our model on the training data itself. The results are summarised in Table 3. Surprisingly, given the lack of dropout in the word embeddings and MLP, our scores for the training data were not significantly better than those for the corresponding test data, implying that our model is not overfitting the data.

[4]Note that our UAS score might be slightly inflated, since the arc from the root to itself always appears in both the prediction and training set. In the shorter data sets, this inflation is more pronounced (since there are fewer arcs in each sentence).

5 Conclusion

In this paper, we implemented a neural graph-based dependency parser. We discussed the choices we made

6 Team Responsibilities

Jack Harding implemented the MST algorithm and the function for testing the UAS/LAS scores. He also wrote the whole of this report.

Bram Kooiman implemented the trainable algorithm (with GPU drive) that formed the core of the project.

Akash Raj was responsible for all the data cleanup and manipulation. He also assisted Bram in implementing the trainable algorithm, and assisted Jack with the function for testing the UAS/LAS scores.

Debugging was mainly done in a team setting, although Bram and Akash did pretty much all of the debugging of the neural network (and Jack did the debugging of the MST algorithm).

Acknowledgments

We would like to thank Daan van Stigt and Joost Bastings for their constant assistance throughout this project.

References

Timothy Dozat and Christopher D. Manning. 2017. Deep biaffine attention for neural dependency parsing. *ICLR* pages 1–8.

Dan Jurafsky and James H. Martin. Forthcoming. *Speech and Language Processing*. 3rd edition.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional lstm feature representations. *TACL* 4:313–327.

A GitHub Repository

The link to the GitHub repository for this project can be found at: https://github.com/akashrajkn/dependency-parser