Disjoint predictions and complete classification accuracy in automated dependency parsing

Abstract

A common approach for solving the dependency parsing problem is to apply the SHIFT-REDUCE algorithm in combination with neural networks outputting the desired transition and/or label at each iteration. This study compares the performances of different models for labeled dependency parsing.

First, an unlabeled dependency parsing was implemented which consists of a bi-LSTM and an MLP on the top of it outputting the selected transition. This model was then extended by adding a two hidden layer MLP which takes the representations of the head and the tail of the transition and outputs one of 49 labels. This MLP was then altered to additionally accept as an input the parent of the current head. It was also built an other version that accepts the corresponding GloVe word embeddings instead of LSTM output vectors. Finally it was created an architecture with a bi-LSTM followed by only one MLP, that predicts one of 99 possible labeled transitions out.

The purpose of this work is to evaluate such different architectures.

1 Introduction

A dependency grammar is a grammar that maps dependency relations. A dependency is a labeled link that connects a pair of words to each other. A dependency consists of a head, a dependent and a label. The head is the word that is modified, the dependent is the modifier word and the label is an identification of the type of the relation. For example, a verb is expected to have an agent - the acting entity, and a patient - the object acted upon. The model assumption is that in a sentence all the words always depend on others, except one, which is the root of a sentence. Multiple dependencies in a sentence form a dependency structure, namely an acyclic directed graph. Since in this formulation a word can only be modified by another one, every word has only one direct parent. This means that the final sentence structure is represented as a tree, called *dependency tree*.

Since natural language is in some extent ambiguous, it is possible for a sentence to have multiple valid dependency trees. Establishing the most probable one for a phrase can lead us to an understanding of the sentence. More precisely, the tree gives us information about more meaningful ways to combine words than adjacency. A dependency parser takes a sentence and possibly other additional data (e.g. POS tags) as an input and outputs a representation of the highest scoring dependency tree.

For the reasons stated above, it is useful to study those word-to-word connections and find ways of automatically extracting them in a suitable representation. An example of a parse tree is provided in figure 1.

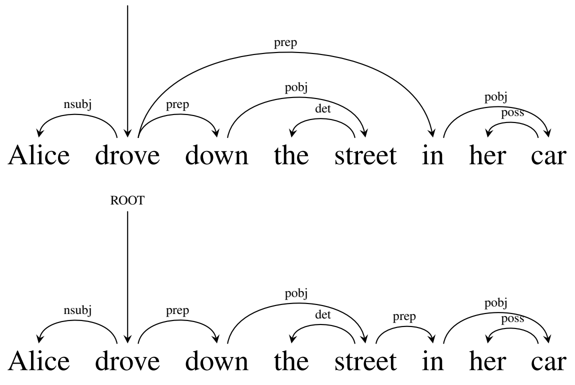
ROOT
prep
nsubj
prep
pobj
det
pobj
poss
Alice drove down the street in her car
ROOT
nsubj
prep
pobj
det
prep
pobj
poss
Alice drove down the street in her car

Figure 1: An example of possible parse trees of a sentence.

In the recent years have been introduced new methods that exploit deep learning techniques, mainly LSTMs, to improve classical parsing methods, such as the SHIFT-REDUCE algorithm. These models exploit the learning capabilities of deep neural networks to automatically extract features that will be used to determine the parser actions. In fact, using this approach, features are learned trough an end-to-end training instead of being handcrafted by domain experts.

2 Problem

In our work we chose to implement one of the most widely used architecture for neural dependency parsing. We used a bidirectional LSTM to process the sentences, in order to automatically learn their structures. The output representations of the recurrent network are then employed to predict the SHIFT-REDUCE action, that the standard algorithm has to perform. The details of this implementation will be extensively addressed in section 3.

The simplest approach is to create a complete classification model. It is a model able to successfully predict dependencies together with their labels. This is the human behavior, we understand that a word depend on another given their logical relationship.

A more machine-refined approach makes disjoint predictions. This means it tries to predict first the dependencies and only afterwards the labels. In the SHIFT-REDUCE paradigm in each step there are three possibilities to choose from - the transitions SHIFT, REDUCE-LEFT, REDUCE-RIGHT.

In our experiments our main focus was comparing different models and their performance in complete classification (labeled parsing) measuring the labeled accuracy score.

We first trained a basic model that outputs only unlabeled actions. This was a baseline for the subsequent experiments (see sections 3.4.1 and 3.3 for more details about the model).

Then, we modified the basic architecture to output the labels of the transitions. Because of the larger number of classes for label prediction (49), it is clear that this problem is more complex than the unlabeled prediction.

Even if the dependency labeling is an important feature, our main concern was not to undermine the first task of transition classification. Since our current implementation is not able to backtrack from mistakes, a wrong action prediction during the parsing process will affect negatively all the subsequent transitions. For this reason the models which predict the labels were trained separately without training any other end-to-end model. The label classifiers only take the words vectors outputted by the bi-LSTM and they were trained without any further training of it.

The basic label classifier model takes the bi-LSTM vectors for the head and for the tail of the dependency and outputs the label. An intuition behind this is that, even if the bi-LSTM output vectors were trained only for the transition prediction task, they can still be useful for label prediction, since they encode the *context-aware* meaning of the words in the sentence (see section 3.4.2).

In attempt to prove this allegation, we trained the same classifier using vectors, which represent the *context-unaware* meaning of the words (see section 3.4.4).

For determining a label of a connection, the head and the dependent of this connection are used. However, we expected the label of the current connection to be dependent also on the preceding connection. More precisely, we expected to get an higher accuracy of label prediction providing the parent of the head word of the currently unlabeled connection in the dependency tree. In an attempt to prove it, we trained a model that has this additional information as input (see section 3.4.3).

Finally, to evaluate whether a disjoint transition/label model performs better than a complete one which classify between all 99 cases, we also implemented the latter (see section 3.4.5). There are 99 possible cases since the possible labels are 49 and they can be applied in both directions. This will likely cause additional error while performing predictions because of the larger amount of classes.

3 Approach

To automate the process of extracting dependencies we have to represent in a suitable way the function of words inside a context and the way they relate to each other. The difficulty resides on the fact that dependencies are not necessarily between words in direct sequence, but may span all around the sentence. We cannot feed couples of words, linearly or randomly, and let the machine decide the dependency, since it would not consider the context, which is hidden in the whole sentence.

We need an instrument to synthesize the role of the word in the sentence and add this information to the meaning of the word itself.

In this section we will explain how the SHIFT-REDUCE algorithm works and how we use it (section 3.1). Moreover we will show how we represent words (section 3.2) and the details of the bi-LSTM (section 3.3) and the MLP we implemented (from section 3.4.1).

3.1 SHIFT-REDUCE algorithm

The SHIFT-REDUCE algorithm is used to output a dependency tree for a given sentence. It is an iterative algorithm that takes a sentence as input and, on every iteration, makes a transition to a new state. There are multiple transitions possible. This approach consists of processing words one by one left-to-right, while maintaining two data structures: a queue (called also buffer) of unprocessed words and a stack of partially processed words. At each point the SHIFT-REDUCE algorithm choose one of three possible actions:

- SHIFT: move one word from the queue to the stack;
- REDUCE-LEFT: add a dependency from the top word on the stack to the second word of the stack, then remove the second word from the stack;
- REDUCE-RIGHT: add a dependency from the second word on the stack to the top word of the stack, then remove the top word from the stack.

The main task is to learn how to choose each action with a classifier. Features should generally cover at least the last stack entries and first queue entry. In our case we use the top two words on the stack and the first word in the buffer. A word is identified as an embedding vector described in section 3.2.

3.2 Word embedding

Our model has to be able to handle words as an input. The standard method is to use one hot vectors. If the vocabulary size is $|V|$ the one hot vector representation for the i-th word in the vocabulary is a sparse vector of all zeros with a one in the i-th entry. Since the the input is highly dimensional and sparse, the first layer of the network consists of a linear projection in a lower dimensional dense embedding space. This projection matrix can be learned during the training process. However, a common choice is to use a pre-trained word embedding. One advantage of this choice is that the word embeddings, which were trained on a much larger text corpus, already encode a semantic meaning. That means that words with similar meaning are mapped to close points in the embedding space. Moreover it has also been shown that they can also encode relationships between concepts, such as the *state:capital* relation (see figure 3.2).

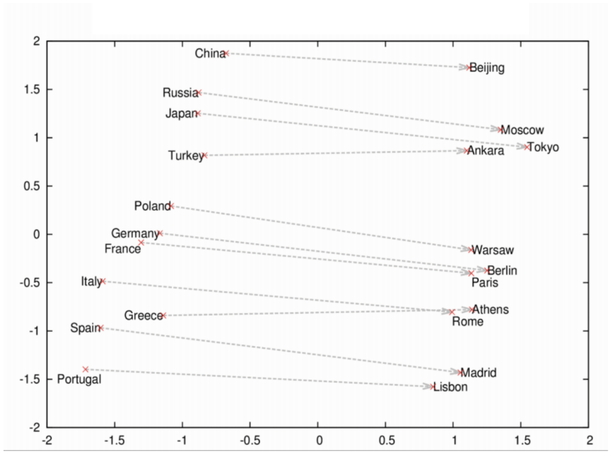

Figure 2: A two dimensional representation of words embedding. Here it is shown how such representations can encode relationship between concepts

In our model, we decided to use a fixed embedding matrix during training. One possible extension for our work could be to use pre-trained embeddings as an initialization and then to fine-tune them during the training process. This would adapt the weights to the specific characteristics of the training set. However this choice could also lead us to over-fit on the training set.

In our case, we used the GloVe embedding(Pennington et al.,). The original vectors were trained on a corpus of 840 billion tokens, resulting in a vocabulary of 2.2 million words. [1] The biggest advantage of using GloVe is that it contains a huge amount of signs, including symbols. Words are represented as a 300 dimensional vector which are created factorizing a word-word co-occurrence matrix, which tabulates how frequently words co-occur with one another in a given corpus.

[1]http://nlp.stanford.edu/projects/glove/

3.3 Long Short Term Memory (LSTM)

Once a sentence is converted in a list of vectors, we feed these word embedding into a bidirectional Long Short Term Memory (Hochreiter and Schmidhuber, 1997), which is a special kind of Recurrent Neural Network (RNN). An RNN is an attempt to simulate memory in a Neural Network (NN) adding to the input layer the previous output so that the NN can use informations from the former state. An LSTM is a more advanced type of RNN capable of learning long-term dependencies. Special gate nodes are added to provide the NN the possibility of remember, forget and use the current or previous inputs (see figure 3).

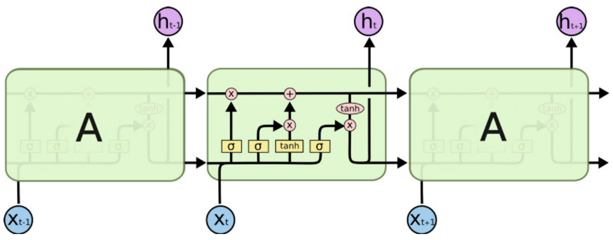

Figure 3: Simple overview of the architecture of a LSTM cell [3].

These improvements permit the recall of data inputted many steps before, allowing it to analyze long term dependencies. Moreover at the same time it prevents back-propagated errors from vanishing or rising uncontrollably. Because of their recurrent nature LSTMs are particular suited for handling ordered sequential inputs of varying length, such as time series. In particular, a sentence is an ordered sequence of words, so LSTMs are good tools for the task we want to perform.

Since both forward and backward words order influence the dependencies, we take into consideration both. We do that by concatenating the output of two LSTM, one which take the word in one direction and the second in the other. Then we take the sequence of hidden states of both the recurrent networks and we join them together as a new representation of the sentence. This will be the input for the next layers of the network. This architecture is known as bidirectional LSTM (Kiperwasser and Goldberg, 2016) (Cross and Huang, 2016). This type of networks is used to grasp meaning by data which logical order is non-univocal, as in the case of words in a sentence (see figure 4).

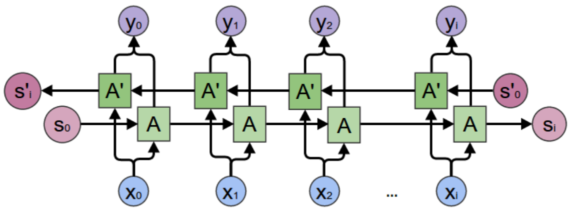

Figure 4: Bidirectional LSTM with concatenation of the hidden unit outputs.

3.4 Multi-Layer Perceptrons

The next step is to feed these representations of words to another kind of neural network. In our case, we simply used multi-layer perceptrons (MLPs) (see figure 5). Since all MLPs we used have to perform a multi-class classification, on the output of all our MLPs we take a softmax to compute probabilities. Our models use categorical cross-entropy (known also as log loss) as loss function:

$$L(t, y) = \sum_{i}^{C} t_i \log y_i$$

where t is the one-hot target vector and y is the prediction.

3.4.1 MLP for transition prediction

The first MLPs we used in our experiments predicts only the transitions of the SHIFT-REDUCE model, henceforth MLP_T. The input of MLP_T are 3 word representations from the bidirectional LSTM. They represent the top 2 words of the stack and the first word in the buffer. Using this information MLP_T predicts one of the 3 possible transitions. Namely: SHIFT, LEFT-REDUCE and RIGHT-REDUCE. MLP_T has only one hidden layer so the output is expressed as:

$$MLP_T(\theta) = W_2 \cdot \tanh(W_1 \cdot x + b_1) + b_2$$

where $\theta = \{W_1, W_2, b_1, b_2\}$ are the model parameters.

3.4.2 MLP for label prediction with bi-LSTM

Another MLP we used is a label classifier, henceforth MLP_L. The aim of MLP_L is to predict the right label for an arc in the dependency graph. The input of MLP_L are 2 word representations from the bidirectional LSTM which represent the head and the tail of a dependency. Using these embeddings MLP_L predict one of the 49 possible labels in the CONLL format. MLP_L has 2 hidden layers, one linear and one non-linear. We experimentally saw that this configuration performs better than others with multiple non-linear layers. So the output is expressed as:

$$MLP_L(\theta) = W_3 \cdot \tanh(W_2 \cdot (W_1 \cdot x + b_1) + b_2) + b_3$$

where $\theta = \{W_1, W_2, W_3, b_1, b_2, b_3\}$ are the model parameters.

3.4.3 MLP for label prediction with parent

Since it is known that natural language is not well explained by context free grammar, an attempt for better predictions of labels is to add information about the head parents to the classifier. In order to do so we build an improved MLP for label prediction which also uses the embedding of the direct parent of the head. In such a way this new MLP_P should be able to identify labels better. The architecture of this classifier is the same as MLP_L described in section 3.4.2.

3.4.4 MLP for label prediction with GloVe

Another slightly different MLP we used is a label classifier that, instead of using the output from the bidirectional LSTM, uses directly the embeddings from the pre-trained GloVe words embedding. The aim and the output of this MLP_G is the same of MLP_L so to predict the right label for an arc in the dependency graph. The architecture of this classifier is the same as MLP_L described in section 3.4.2.

3.4.5 MLP for complete prediction

For our finial experiment we used only one MLP to predict both the transition and the label if is applicable. The input of this MLP_C are, as in MLP_T the embeddings of the top 2 words in the stack and the first word in the buffer. MLP_C then should predict one of 99 $(49 \cdot 2 + 1)$ possible actions, which are the combinations of LEFT-REDUCE and RIGHT-REDUCE with all the labels and the SHIFT action from that particular state. The architecture of this classifier is the same as MLP_L described in section 3.4.2.

4 Training and testing

4.1 Dataset

The data that we used for training is Stanfords Dependency Dataset. It contains sentences and their

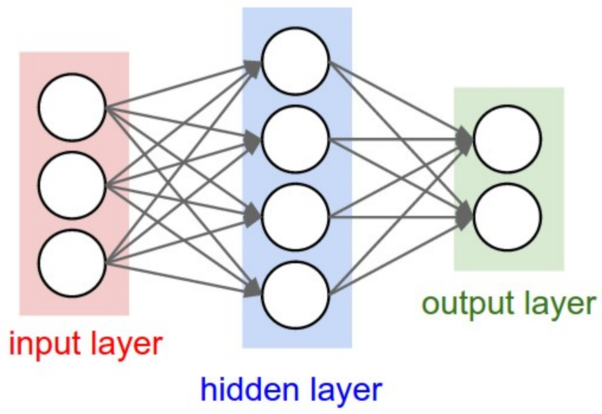

Figure 5: A representation of a MultiLayer Perceptron with one hidden layer

dependency trees. It is divided into a training and a test set of 39’833 and 2’416 dependency trees respectively. It is organized into a CoNLL format [4]. For our purposes we use only the sentence and the data for dependency connections.We don’t employ other information, like POS tags.

This dataset doesn’t not contain transitions. Thus one of the pre-processing steps was to convert all the dependency trees into a ordered list of transitions.

4.2 Pre-processing

Before the training, we ran some tests in order to know how much of our corpus vocabulary was represented in the GLoVe dictionary. We discovered that the training data contains approximately a set of 40’000 unique words, and unfortunately 10’000 of them do not have an embedding in the database. We decided to investigate the nature of these new words and discovered that most of them were numbers, other were personal names and most of them were actually composition of words (e.g *18-years-old*). Instead of simply embedding all of them as a same *UNKNOWN* vector, we tried different approaches for each case in order to gain precision in our classifications.

Numbers For numbers we decided to compute the average of all word embedding vectors inside GLoVe, and assign this value to a new entry in the dictionary, the generic *NUM*. Whenever we encountered a number in the corpus we transformed it into *NUM* so that it would get the calculated embedding. This idea arises from the concepts of vector representation. In fact we observed that all

[4]http://universaldependencies.org/format.html

number embeddings lied near each other in a cluster. I fact their use in a sentence is the same. Finding an average would be like finding the true value of the concept of number, around which all realizations revolve.

Composition For composite words we proceeded by splitting them around the non-alphabetical characters. Then the embedding for the composite word becomes the normalized average of each word vector in the composition. The intuition behind taking the average instead of having a placeholder is that this way leads to train the classifier with much more accuracy. We thought about alternative ways of averaging such compositions: one hypothesis was to perform a weighted average by position because in English the last word usually carry more meaning. The problem was that there are cases which this rule is not true. Another idea was to weight the average based on the frequency of the word: usually a less frequent word is more informative. This approach were not applicable since we do not have a way to measure how frequent a word is, since our corpus is too limited and GLoVe did not offer this information.

Unknown Furthermore, for words that were not numbers or compositions, we used a special vector, initialized as the average of the whole dictionary. The reason behind this choice, instead of a zero vector for example, is that it would have no bias. This means it would contain the same information about any cluster. An unknown word would then be just an uninformative placeholder in the bi-LSTM that would not swing the interpretation of the context in any direction.

Special tokens Finally, our implementation needs two more vectors, one for the *ROOT* and one as a placeholder for the empty buffer (*EMPTY*). *EMPTY* has the function to identify the end of the sentence. This token will always remain in the buffer. *ROOT* would be the first input in the LSTM in the left-to-right direction, while *EMPTY* will be the the last, and obviously the opposite for the right-to-left LSTM. We decided to assign the embedding of the full stop to the *ROOT* and a vector of zeros for the empty. It is possible to have an improvement on this assignments, but we found no information in the literature, being this a relatively young field.

4.3 Post-processing

A general algorithm would use a greedy approach to make the best possible choice between transitions and find the best dependency tree. However this approach adds a lot of computational cost. We decided to make all the choices based on the local best choice. This is based on the scores (probabilities) the MLP gives to each action, the highest is considered the right one. However, we add some heuristics to make sure our parser to do not make a choice leading to no solutions or make trivial mistakes for a human to spot.

Valid transitions Regarding the dependencies, our parser could make some choices that make no sense or could lead to inconsistencies. To avoid them, whenever the MLP predicts an action, we will check whether such operation is valid with the respect of the current state. If it is not we will apply the second choice and so on. Specifically we apply this set of rules:

- we do not apply a $SHIFT$ if the buffer is empty;
- we do not apply a *REDUCE-LEFT* on *ROOT*;
- we do not apply a *REDUCE-RIGHT* on *ROOT* except when the buffer is empty and

To apply this we use algorithm 1.

Algorithm 1 Rules to avoid inconsistencies during parsing.

```
for all $t_i$ in sorted(transitions) do
    if $t_i$ = RIGHT-R $\wedge$ |STACK| $> 2$ then
        return right-reduce()
    else if $t_i$ = RIGHT-R $\wedge$ BUFFER $= \emptyset$ then
        return right-reduce()
    else if $t_i$ = LEFT-R $\wedge$ |STACK| $> 2$ then
        return left-reduce()
    else if $t_i$ = SHIFT $\wedge$ BUFFER $\neq \emptyset$ then
        return shift()
    end if
end for
```

Root label constraints About labeling, we limited our intervention to what happens with root. Since we label in the same order we reduce, only the last action will reduce under *ROOT*. Taken this in consideration we act with an heuristic similar to the previous one. If the label root is associated to a reduction that is not the last one we instead apply the second most likely. On the other end, independently from the score given by the MLP we assign the root label the last reduction.

Number labeling We noticed that both in the training and the test set there was a curious repetition of labels, precisely there was a *num* label and a *number* label for reductions. We checked in the official encoding and discovered that only *number* is the right one and decided to change the other form to it.

Other tricks We know that a sentence has to be at least one sign long, so we always start with two automatic SHIFTs that bring *ROOT* and the first word inside the stack, before running the parsing procedure. This also avoids the problem of feeding just two *EMPTY* vectors and the *ROOT* to the models.

4.4 Transition classifier training

In MLP_T we used 200 hidden units for the hidden layer (as explained in section 3.4.1). We trained our model for multiple epochs through the training dataset. After the first epoch, the transition classifier did not perform so well in the test set (accuracy score of 51.17) and seemed that training for others would have not affected the performance enough. Surprisingly after 5 epochs the transition classifier achieved an accuracy score of 90.26 on the test set, which is very high. We tried to improve the accuracy training for more epochs but seems that the model maintains its performance and then starts to lose precision on the test set (see table 4.4). The most likely explanation is that the model starts to over-fit on the training set.

Epoch	1	2	**5**	8
Accuracy	51.17	54.90	**90.26**	90.05

Table 1: Convergence and over-fitting of the transition classifier training.

4.5 Label classifier training

In MLP_L we used 400 hidden units for the first hidden (linear) layer and 100 hidden units for the second hidden (non-linear) layer (as explained in section 3.4.2). This model is prone to over-fit as well as MLP_T so we trained this one for only 5 epochs. Since MLP_L relies on the already trained bi-LSTM, the accuracy depend also on that. The bi-LSTM was not trained on label classification but on the transition classification instead. Besides that, MLP_L achieved an accuracy score of 88.02 which leads to an overall label accuracy score of 83.58 on the test set (see table 2 for the complete comparison with the other methods). Since the product of label and dependency accuracies is less than that results we know that there is a positive correlation between mistakes (χ^2 test for independence, $p < 0.05$), which means that the wrong label are not independent but instead more frequent on wrong branches.

4.6 Label classifier with parent training

For a fair comparison with all the other MLPs, in MLP_P we used as many hidden units as in MLP_L and trained the model for the same number of epochs. MLP_P achieved an accuracy score of 79.29 which leads to an overall label accuracy score of 74.99 on the test set (see table 2). As the previous result here we do not have independence between mistakes in labeling and those in parsing, but the same positive correlation (χ^2 test for independence, $p < 0.05$)

4.7 Label classifier with GloVe embedding

Again for a fair comparison with all the other MLPs, in MLP_G we used as many hidden units as in MLP_L and MLP_P, and trained the model for the same number of epochs. MLP_G achieved an accuracy score of 72.54 which leads to an overall label accuracy score of 66.17 on the test set (see table 2). In this case we cannot exclude independence in the mistakes, (χ^2 test for independence, $p = 0.089$), this is probably due to the higher error rate, but we would like to think the LSTM could have helped in this matter, doing part of the analysis for the classifier.

4.8 Transition-label classifier

Finally in MLP_C we used 200 hidden units for the LSTM hidden layer (as explained in section 3.4.5) and we used as many hidden units as in MLP_L and MLP_P, so 400 and 100 respectively. Even if this model would require more training, since has to predict more classes, we trained this model for 5 epochs as the other models for a fair comparison. MLP_C achieved an unlabeled accuracy score of 84.54 and a labeled accuracy score of 72.47 (see table 2 for the complete comparison with the other methods).

5 Conclusion

5.1 Results

	labels only	UAS	LAS
MLP_L	88.02	90.26	**83.58**
MLP_P	79.29	90.26	74.99
MLP_G	72.54	90.26	66.17
MLP_C	-	84.54	72.47

Table 2: Comparison between different MLPs we tested.

Disjoint versus complete classification From the results, we can say that the classifier for the dependencies followed by the label classification performs better than the complete one, as expected (t-test for coupled samples, $p < 0.05$). This was to be expected since classifying between so many choices is very difficult even for a neural network powered by a bi-LSTM. We would say that MLP_C might eventually converge to comparable results but only after much more epochs.

Labeling with parent information The experiment to improve the quality of labeling adding parent information brought to unsatisfying results. The information brought by the head's parent revealed to be an element of confusion for the classifier that lost accuracy in substantial amount. Theoretically this should not happen, as more information could be ignored by the neural network by weighting with a zero and giving the same results as the more compact version. In practice, though, we ran it for 5 epochs as its simpler counterpart, and the optimal state might be ahead. We can only say that its efficiency is worst than the base network (t-test for coupled samples $p < 0.05$), but no claims or conclusions should be done on its efficacy.

Labeling without LSTM information To test how much the LSTM improved the label prediction, we trained the same MLP on the values directly taken from the GLoVe dictionary. The results were significantly worse (t-test for coupled samples $p < 0.05$). This is interesting and remarkable as we trained the LSTM for the parsing, yet it was still able to capture the relationship between words that helped their labeling.

5.2 Future work

Is is easier to think about several improvements that could be done to improve our models and achieve better performance.

End-to-end model for labeling It is likely that training a new full end-to-end model just for the labeling would bring even better results than the ones we achieved. This is beyond the aims of the original study, as the results already confirmed our hypothesis. This will be slower to train but it will increase the accuracy.

Self-embedding We used a pre-trained word embedding dataset to lighten the computational burden, be it time and weight on the CPU. We think that creating a word embedding specifically for dependency parsing would improve the performance, but this would require much more time to converge and a way bigger corpus since GLoVe is trained on a 840 billions word corpus to reach 2.2 millions unique signs. We do not need to cover so many possible inflection of language, but the numbers are still staggering.

Full training of MLP with parents To reach a state of the art comparable accuracy with this method, we would need much more training time so training for more epochs on the corpus. Originally, even the simpler method gave similar results so it is reasonable that a more complex model would need additional training.

Introducing backtracking One big problem of the current model is that there is no chance for backtracking. The main effect of this is that after the classifier makes an error there is no possibility to recover. Moreover, after the first mistake, it generates a cascade of errors. Therefore a huge improvement would be to allow the possibility of backtracking. However this is highly non-trivial, since there is no dataset of errors to train the model for backtracking. One possible idea is to use the mistakes made by the parser on the train set and train itself on how to recover from them.

References

James Cross and Liang Huang. 2016. Incremental parsing with minimal features using bi-directional LSTM. *CoRR*, abs/1606.06406.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional LSTM feature representations. *CoRR*, abs/1603.04351.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation.