Neural Dependency Parsing

Vighnesh Subramaniam
vsub851@mit.edu

1 Introduction

Dependency parsers annotate sentences in order to analyze the syntactic structure of a sentence. These parsers find latent syntactic features in a sentence and have been found to be extremely useful in many NLP tasks such as machine translation (Duan et al. (2020)), natural language understanding (Zhang et al. (2020)), and text summarization (Xu et al. (2020)). However, inaccurate parses can hinder the performance of these downstream tasks and in order to improve the performance of these tasks, it is essential to have an accurate parser.

We build two models for dependency parsing using the corresponding dependency tree for the parse of a sentence. The first model, **Arc Prediction**, directly builds the dependency parse tree for a sentence by predicting the edges in the parse tree which represents the head-dependent relationship between words, as well as the dependency label on each edge. Specifically, for each word w it predicts the head of w in the sentence and the dependency label. The second model, **Arc Pair**, builds the dependency parse tree for a sentence by considering a every pair of words in the sentence. For each pair (w_i, w_j) of words, the model predicts if w_i is the head of w_j i.e. the pair form a head-dependent relationship and if so, what is the dependency label associated with this relationship. In both models, we create sentence encodings for each sentence using both an LSTM and BERT (Devlin et al. (2018)). We then pass these encodings through several linear layers to predict the head and dependency label for each word in the sentence encoding. Both models take inspiration from Dozat and Manning (2017).

Additionally, we train these models using different architectures and approaches for building sentence encodings as well as different hyperparameters, and compare the results with the state-of-the-art dependency parser performance measured using UAS/LAS. We find that while the Arc Prediction Model does not achieve state-of-the-art results, the Arc Pair Model achieves results comparable to Dozat and Manning (2017) on both unlabeled attachment score (UAS) and labeled attachment score (LAS) using particular layers of BERT sentence encodings despite not using the same attention mechanism. We see that both models have interesting differences in performance as well as interesting difference in performance of approahces we took in each model. These differences can give interesting insight into the importance of sentence encodings and word representations, and how different approaches to building these encodings can give vastly different results. Based on our results, our future work is to study what linguistic features are actually being embedded into the sentence encodings obtained using the traditional methods.

2 Background and Related Work

Dependency Grammar is a syntactic formalism based on dependency relations, where *dependency* is the notion that linguistic units (e.g. words) are connected to each other by direct links, and these links are described by a label called a *dependency label* (De Marneffe et al., 2006). Dependency grammars are determined by the relation between a word (a head) and its dependents. So, when we parse a sentence using the Dependency Grammar formalism, we identify all head-dependent pairs and the dependency label for each pair, creating a *Dependency Parse*. We show an example of a dependency parse in Figure 1.

In most dependency grammar parses, the finite verb is assumed to be the structural center of the parse. All other words are dependents of the finite verb, or of another word, or a chain of words that are dependent on the finite verb. The finite verb does not have an official head, as seen with the word "prefer" in Figure 1 but we do have a dependency label called *root*. When we parse in our models, we create a $ROOT$ token not originally in

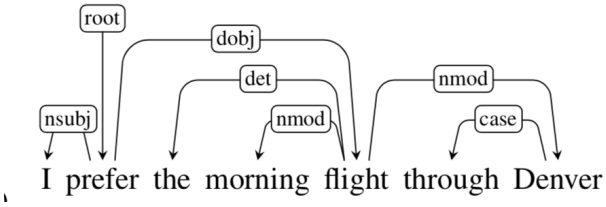


Figure 1: An example of a dependency parse. The words are connected to each other by the direct links, which are arrows in this case. Each arrow has its dependency label in a box. For example, see "I" and "prefer" connected by an arrow from "prefer" to "I" with the label *nsubj*. Finally, note the word "prefer" which lacks a head. We say "prefer" has a root dependency and is the structural center of the parse. When we process, we create a head-dependent relation between an artificial *ROOT* token and the finite verb ("prefer" in this case).

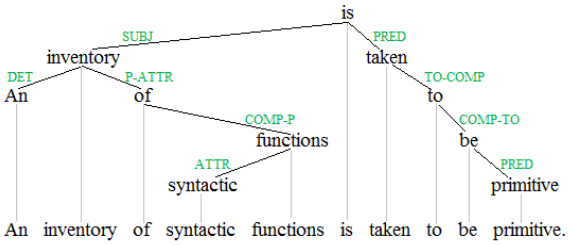


Figure 2: An example of a dependency tree. The nodes of the tree represent the linguistic units that we are forming dependencies between, in this case the words in the sentence. The dependency tree edges represent the head-dependent relation and are labeled with the dependency labels. The Root dependency label is not shown explicitly here but would be from the finite verb "is". We can notice that "is" is the root of the tree.

the sentence to capture this dependency label by creating an artificial head-dependent relation.

When we consider each dependency label for a head-dependent pair, the dependency label describes the nature of the dependencies. Some examples of dependency labels include:

- *nsubj*: relation between sentence subject (dependent) and finite verb (head).
- *det*: relation between determiner (dependent) and noun (head).
- *advmod*: relation between adverb (dependent) and verb is modifies (head).

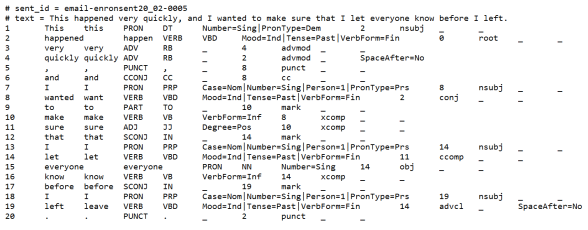
We can take the dependency parse in Figure 1 and convert it to a tree structure called a *Dependency Tree*. We do so in Figure 2 on a different sentence. We can use the dependency tree to visualize the dependency structure by representing each word as a node in the tree and head-dependent relations as edges. We label each edge with its corresponding dependency label. It was also argued in Rambow and Joshi (1997) that the depth of a dependency tree (length of longest path from root to child) captures the difficulty of comprehending a sentence. Both our models use the idea of a dependency tree as a way to build a dependency parse.

The traditional, non-neural way of performing dependency parsing is called the shift-reduce method. This is a dynamic program which builds the dependency tree for a sentence one edge at a time.

One of our models, Arc Prediction Approach, is based on the methods used by Dozat and Manning 2017. This method takes in a sentence and for each word in the sentence predicts its head and dependency label that describes its relation with its head. The classifier built in that paper uses a multilayer perceptron and BiLSTM which is similar to the architectures described in Section 4. However, our Arc Prediction Approach uses BERT as the attention mechanism whereas the model described in Dozat and Manning 2017 uses *biaffine attention*. The biaffine attention mechanism is a linear transformation of encoded word representations obtained from the BiLSTM and the linear transformation matrix becomes a parameter in the model. BERT instead uses *Multi-headed Attention*.

Recent papers have incorporated global greedy approaches to parsing that use smarter forms of encoding and attention in order to achieve similar accuracy as that in Dozat and Manning 2017 but obtain parses much faster (Li et al., 2020). These models introduce newer forms of decoding to obtain parse trees in linear time instead of polynomial time. There have also been more emphasis on parsing in other languages outside of English in recent years as well as incorporating different transformer pipelines (Mohammadshahi and Henderson, 2021).

The current best dependency parser is given by Mrini et al. (2020) which obtains an accuracy of 97% according to one of our evaluation metrics described in Section **??**. This parser uses a *label attention layer* to create better word representations/encodings and incorporate syntactic information into word encodings to achieve better performance.



```
# sent_id = email-enronsent20_02-0005
# text = This happened very quickly, and I wanted to make sure that I let everyone know before I left.
1       This    this    PRON    DT      Number=Sing|PronType=Dem        2       nsubj   _       _
2       happened        happen  VERB    VBD     Mood=Ind|Tense=Past|VerbForm=Fin        0       root    _       _
3       very    very    ADV     RB      _       4       advmod  _       _
4       quickly quickly ADV     RB      _       2       advmod  _       SpaceAfter=No
5       ,       ,       PUNCT   ,       _       8       punct   _       _
6       and     and     CCONJ   CC      _       8       cc      _       _
7       I       I       PRON    PRP     Case=Nom|Number=Sing|Person=1|PronType=Prs      8       nsubj   _       _
8       wanted  want    VERB    VBD     Mood=Ind|Tense=Past|VerbForm=Fin        2       conj    _       _
9       to      to      PART    TO      _       10      mark    _       _
10      make    make    VERB    VB      VerbForm=Inf    8       xcomp   _       _
11      sure    sure    ADJ     JJ      Degree=Pos      10      xcomp   _       _
12      that    that    SCONJ   IN      _       14      mark    _       _
13      I       I       PRON    PRP     Case=Nom|Number=Sing|Person=1|PronType=Prs      14      nsubj   _       _
14      let     let     VERB    VBD     Mood=Ind|Tense=Past|VerbForm=Fin        11      ccomp   _       _
15      everyone        everyone        PRON    NN      Number=Sing     14      obj     _       _
16      know    know    VERB    VB      VerbForm=Inf    14      xcomp   _       _
17      before  before  SCONJ   IN      _       19      mark    _       _
18      I       I       PRON    PRP     Case=Nom|Number=Sing|Person=1|PronType=Prs      19      nsubj   _       _
19      left    leave   VERB    VBD     Mood=Ind|Tense=Past|VerbForm=Fin        14      advcl   _       SpaceAfter=No
20      .       .       PUNCT   .       _       2       punct   _       _
```

Figure 3: Example CoNLL-U file. It contains linguistic information for each word in a tab-delimited CSV format. Each row has a tab that describes a word, each tab having its index (1-indexed) in the sentence, the word itself, lemma, UPOS, XPOS, morphological features, head index, dependency label, head-dependency label pair, and miscallaneous information, in that order

3 Dataset

We used the Universal Dependencies dataset Version 2.8 as our main dataset for dependency parsing. The dataset contains CoNLL-U files, where each CoNLL-U file is a formatted tab-delimited CSV file that contains several sentences and linguistic information about the words in the sentences. The linguistic information for a word is contained in a row which has several fields such as index (the numeric position) in the sentence, the word itself, the lemma, POS tag, dependency head index (numeric position of dependency head), dependency label, and miscellaneous information. We see an example of a CoNLL-U file in Figure 3.

Our data consists of 3 English CoNLL-U files – one file for training, one for validation, and one for testing. We had 12543 sentences in the training dataset which was 13.5 MB, 2002 sentences in the validation dataset which was 1.7 MB, and 2077 sentences in the testing dataset which was 1.7 MB.

We processed these files by extracting the words, lemmas, dependency heads, and dependency labels.

4 Arc Prediction Model

In this model, we take in a sentence and initially process it through an LSTM or BERT language model to obtain a sentence encoding. We then process this encoding to output a matrix containing the probability distribution for each word w_i in the sentence over the index (location) of the head of w_i in the sentence when we want to predict heads. Or we output a matrix containing the probability distribution over all possible dependency labels that w_i could have when we want to predict dependency labels. We call this the arc prediction model since we are predicting the dependency arc (where is the head for word w_i in the sentence) and label for that arc for each word in the sentence we pass in to the model.

In order to boost accuracy, we tried incorporating both an LSTM and BERT tokenization and language modeling to create sentence encodings of each input sentence. This ensures that all long-distance dependencies are being accounted for as much as possible which is important given that words can have dependencies across the sentence and a sentence could be very long.

4.1 Data Processing

In this model, we processed each CoNLL-U file by extracting the word, lemma, head, and dependency label as described in Section 3. We converted lemmas and dependency labels to integer IDs to be processed in the model. We collect the lemmas and labels into respective dictionaries mapping each lemma or label to its IDs.

We created a dictionary for each sentence that contained the words in the sentence, the lemma integer ID for each word in the sentence, the head for each word in the sentence, and the dependency label ID for each word in the sentence.

Next, if we used BERT in our architecture, we processed the words using the BERT Tokenizer from `bert-based-uncased` in the Hugging Face Toolkit (Wolf et al. (2019)) which we used to output both input IDs that represented the sentence as well as an attention mask that we used for our language model in our architecture. We added these to the dictionary for each sentence as described above.

4.2 Architecture

As described above, we built the arc prediction model using two separate encoding architectures which we describe below. The two architectures took in slightly different inputs and use different encodings, but outside of these two differences the architectures are the same. We describe the inputs, layers, and outputs for the architectures below.

4.2.1 LSTM Encodings

In the LSTM Encodings architecture, we take in as input the lemma IDs for a sentence as described in Section 4.1. We pass the lemma IDs through an embedding layer and then through a bidirectional three layer LSTM. We use a bidirectional LSTM to accurately capture both rightward dependency arcs

as well as leftward dependency arcs. Heads can be to the left or right of a word and this allows the full context to be captured. We use the output hidden state as the sentence encoding which will be input to the rest of our architecture.

Following this, we use our sentence encoding as input to 9 fully connected linear layers. After that, we obtained a matrix where each word w_i in our sentence was described by a row. The row for w_i contained a probability distribution either over the location of the head for w_i if we were predicting heads or over which dependency label w_i had if we were predicting dependency label. This matrix was our output from the model.

4.2.2 BERT Encodings

In the BERT Encodings architecture, we took in as input the integer IDs and attention mask returned from the BERT tokenizer described in Section 4.1 in order to get the correct input for the BERT language model. We used the pretrained language model from `bert-base-uncased` to process the input IDs and attention mask and obtain hidden states from all layers of BERT which we refer to as BERT encodings. We accessed the hidden states from a particular layer which becomes our encoded sentence.

Following this, we passed our sentence encoding as input to 9 fully connected linear layers as in Section 4.2.1. We obtain a similar output matrix where each row in the matrix describes each word w_i in the sentence. Each row for w_i contained a probability distribution either over the location of the head for w_i if we were predicting heads or over which dependency label w_i had if were predicting the dependency label. This matrix was once again the output from the model.

4.3 Training

We trained both architectures using the English training data from the Universal Dependencies dataset described in Section 3. We processed this data using the same steps as described in Section 4.1.

We trained for 3, 10, and 20 epochs with a batch size of 1 so we only processed architectures one sentence at a time. Since we had made this a classification task for predicting the head of a word and its dependency label, we used Cross Entropy Loss as our loss function and used the Adam Optimizer for optimizing the architectures.

We also trained using different layers of BERT

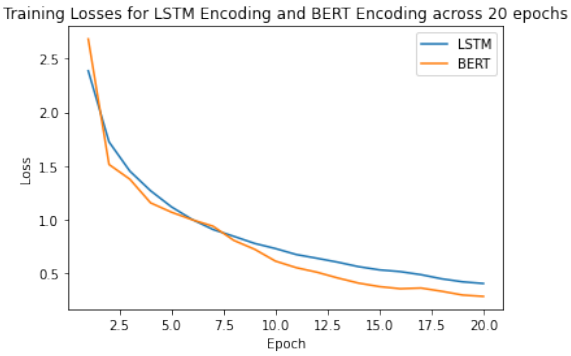


Figure 4: Training losses when training the LSTM encodings and BERT encodings from the 8^{th} layer over 20 epochs.

encodings. BERT has encodings from all of its 13 transformer layers and we trained our model on encodings from layer 4, layer 8, and the last layer.

We had the following hyperparameters for training:

1. We used a learning rate of 0.005 on the LSTM Embeddings Architecture and a learning rate of 0.0005 on the BERT Embeddings Architecture to allow for proper convergence.
2. A dropout percentage of 0.33.
3. An embedding layer dimension of 200
4. 3 layers in the LSTM unit

We plotted our training losses for 20 epochs for the LSTM Encodings Architecture and BERT Encodings Architecture with encodings from the 8^{th} layer of BERT in Figure 5. We noticed generally in both architectures, the training losses decrease across epochs which means that the model is improving as the epochs increase and thus the learning rate and optimizer is helping to converge to the optimal architecture. Our minimum training loss was 0.2877 which occurred in the BERT model after running for 20 epochs using the encodings from the 8^{th} layer.

4.4 Decoding Algorithm

After training was completed as shown in Section 5.3, we had to decode the model output to obtain a final prediction for the head and dependency label for each word in the input sentence.

As described before, when we ran either the LSTM Encoding Architecture or BERT Encoding Architecture, we obtained as output a matrix M where each row in M describes each word w_i in

the input sentence. Each row for w_i contained a probability distribution either over the index of the head for w_i if we were predicting heads or over which dependency label w_i had if were predicting the dependency label.

We now decoded M as follows. We know each row in M gives the probability distribution D over heads or dependency labels for a word w_i. We cover predicting heads first. If we are predicting heads, then each j in D corresponds to an index in the sentence. The value for each position j in D gives the probability that the sentence index j is the head of w_i. We find the j with the maximum probability by using $argmax$ and call the position j^*.

If we are trying to predict dependency labels, then each j in D corresponds to an index in the dependency label dictionary that maps dependency labels to integer ID as described in Section 4.1. Thus, j corresponds to an individual dependency label as encoded by our dependency label dictionary. The value in each position j in D gives the probability that w_i has the dependency label corresponding to index j in our dependency label dictionary. We find the j with the maximum probability by using $argmax$ which gives the position j^* with maximum probability.

Thus, we have either found the predicted head or dependency label through j^*. This represents the model's prediction for what the head and dependency label is for each word w_i. We used the prediction for evaluation as described in the next section.

5 Arc Pair Model

In this model, similar to the Arc Prediction Model in Section 4, we take in a sentence and initially process it through an LSTM or BERT language model to obtain a sentence encoding. We then take each word embedding in the sentence encoding since the sentence encoding is a concatenation of word embedding and form words pairs. For pair (w_i, w_j), the model outputs the probability that w_i is the head of w_j when we want to predict heads or the model outputs a probability distribution over the dependency label between w_i and w_j which we label as $NONE$ when w_i and w_j do not have a head-dependent relationship. We call this the arc pair model since we have created a binary classification task when predicting heads with this model on each pair of words. For each pair, (w_i, w_j), we want to predict if it forms a head-dependent relationship (1) or does not (0).

Similar to the Arc Prediction Model, we incorporated both an LSTM and BERT tokenization and language modeling to create sentence encodings and in this case extract word embeddings to form our words pairs. This once again ensured long-distance dependencies are being accounted for as much as possible.

5.1 Data Processing

In this model, similar to the data processing described in Section 4.1, we processed each CoNLL-U file by extracting the word, lemma, head, and dependency label as described in Section 3. We converted lemmas and dependency labels to integer IDs to be processed in the model. We collect the lemmas and labels into respective dictionaries mapping each lemma or label to its IDs.

We created a dictionary for each sentence that contained the words in the sentence, the lemma integer ID for each word in the sentence, the head for each word in the sentence, and the dependency label ID for each word in the sentence.

Next, if we used BERT in our architecture, we processed the words using the BERT Tokenizer from `bert-based-uncased` in the Hugging Face Toolkit (Wolf et al. (2019)) which we used to output both input IDs that represented the sentence as well as an attention mask that we used for our language model in our architecture. We added these to the dictionary for each sentence as described above.

In this model, we also needed to form word pairs. We did so by taking each sentence which had length n and forming all n^2 pairs between words. If two words had a head-dependent relationship, we added the label 1 for them. If not, we added 0. In order to cover the root dependency that most finite verbs have in the sentence, we created a token called $ROOT$ which is the head of the root dependency. This adds $2n + 1$ pairs since we must add pairs for every word in the sentence with $ROOT$ so a length n sentence forms $(n + 1)^2$ pairs. For each pair (w_i, w_j), we associate a $NONE$ dependency label ID if they do not form a head-dependent pair. If w_i is the head of w_j, then we associate the dependency label ID for w_j.

5.2 Architecture

As described above, we built the arc pair model using two separate encoding architectures which

we describe below. The two architectures took in slightly different inputs and use different encodings, but outside of these two differences the architectures are the same. We describe the inputs, layers, and outputs for the architectures below.

5.2.1 LSTM Encodings

In the LSTM Encodings architecture, we take in as input the lemma IDs for a sentence as described in Section 4.1. We pass the lemma IDs through an embedding layer and then through a bidirectional three layer LSTM. We use a bidirectional LSTM to accurately capture both rightward dependency arcs as well as leftward dependency arcs. Heads can be to the left or right of a word and this allows the full context to be captured. We use the output hidden state as the sentence encoding which will be input to the rest of our architecture.

Afterwards, we collected the word embedding pairs from the sentence encoding from the LSTM. We used the dictionary built in Section 5.1 to see the corresponding binary label with each pair indicating if it formed a head dependent relationship or not. Following this, we use our sentence encoding as input to 9 fully connected linear layers. After that, we obtained a matrix where each word w_i in our sentence was described by a row. The row for w_i contained a probability distribution either over the location of the head for w_i if we were predicting heads or over which dependency label w_i had if we were predicting dependency label. This matrix was our output from the model.

5.2.2 BERT Encodings

In the BERT Encodings architecture, we took in as input the integer IDs and attention mask returned from the BERT tokenizer described in Section 4.1 in order to get the correct input for the BERT language model. We used the pretrained language model from `bert-base-uncased` to process the input IDs and attention mask and obtain hidden states from all layers of BERT which we refer to as BERT encodings. We accessed the hidden states from a particular layer which becomes our encoded sentence.

Following this, we collected the word embedding pairs similar to the LSTM encoding architecture. By using `bert-based-uncased`, we ensured that BERT was not splitting a sentence into subwords. We then passed our sentence encoding as input to 9 fully connected linear layers as in Section 4.2.1. We obtain a similar output matrix where each row in the matrix describes each word w_i in the sentence. Each row for w_i contained a probability distribution either over the location of the head for w_i if we were predicting heads or over which dependency label w_i had if were predicting the dependency label. This matrix was once again the output from the model.

5.3 Training

We trained both architectures using the English training data from the Universal Dependencies dataset described in Section 3. We processed this data using the same steps as described in Section 4.1.

We trained for 3, 10, and 20 epochs with a batch size of 1 so we only processed architectures one sentence at a time. Since we had made this a classification task for predicting the head of a word and its dependency label, we used Cross Entropy Loss as our loss function and used the Adam Optimizer for optimizing the architectures.

We also trained using different layers of BERT encodings. BERT has encodings from all of its 13 transformer layers and we trained our model on encodings from layer 4, layer 8, and the last layer.

We had the following hyperparameters for training:

1. We used a learning rate of 0.001 on the LSTM Embeddings Architecture and a learning rate of 0.0005 on the BERT Embeddings Architecture to allow for proper convergence.

2. A dropout percentage of 0.33.

3. An embedding layer dimension of 200

4. 3 layers in the LSTM unit

We plotted our training losses for 20 epochs for the LSTM Encodings Architecture and BERT Encodings Architecture with encodings from the 8^{th} layer of BERT in Figure 5. We noticed generally in both architectures, the training losses decrease across epochs which means that the model is improving as the epochs increase and thus the learning rate and optimizer is helping to converge to the optimal architecture. Our minimum training loss was 0.2877 which occurred in the BERT model after running for 20 epochs using the encodings from the 4^{th} layer.

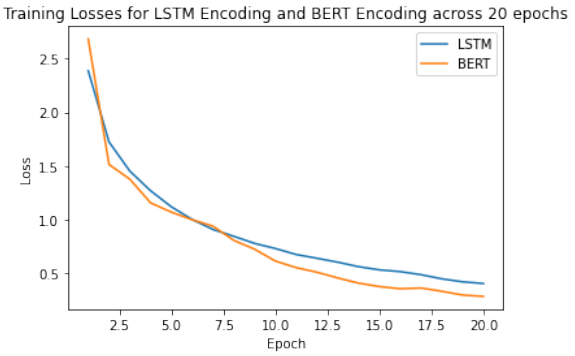


Figure 5: Training losses when training the LSTM encodings and BERT encodings from the 8^{th} layer over 20 epochs.

References

Marie-Catherine De Marneffe, Bill MacCartney, Christopher D Manning, et al. 2006. Generating typed dependency parses from phrase structure parses. In *Lrec*, volume 6, pages 449–454.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Timothy Dozat and Christopher D. Manning. 2017. Deep biaffine attention for neural dependency parsing.

Sufeng Duan, Hai Zhao, Dongdong Zhang, and Rui Wang. 2020. Syntax-aware data augmentation for neural machine translation. *arXiv preprint arXiv:2004.14200*.

Zuchao Li, Hai Zhao, and Kevin Parnow. 2020. Global greedy dependency parsing. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(05):8319–8326.

Alireza Mohammadshahi and James Henderson. 2021. Recursive non-autoregressive graph-to-graph transformer for dependency parsing with iterative refinement. *Transactions of the Association for Computational Linguistics*, 9:120–138.

Khalil Mrini, Franck Dernoncourt, Quan Hung Tran, Trung Bui, Walter Chang, and Ndapa Nakashole. 2020. Rethinking self-attention: Towards interpretability in neural parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 731–742, Online. Association for Computational Linguistics.

Owen Rambow and Aravind Joshi. 1997. A formal look at dependency grammars and phrase-structure grammars, with special consideration of word-order phenomena. *Recent trends in meaning-text theory*, 39:167–190.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

Haiyang Xu, Yun Wang, Kun Han, Baochang Ma, Junwen Chen, and Xiangang Li. 2020. Selective attention encoders by syntactic graph convolutional networks for document summarization. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 8219–8223. IEEE.

Zhuosheng Zhang, Yuwei Wu, Junru Zhou, Sufeng Duan, Hai Zhao, and Rui Wang. 2020. Sg-net: Syntax-guided machine reading comprehension. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9636–9643.