Sentence Unscrambler: Exploring Deep Learning Models for Word Linearization

Kent Vainio
kentv@cs.stanford.edu

Jason Zheng
jzzheng@cs.stanford.edu

Sonja Johnson-Yu
sonja@cs.stanford.edu

Abstract

Linearization is the task of finding a grammatical order for a given bag of words. This task has versatile applications in the fields of NLP and linguistics, especially in the realms of machine and human translation. In this paper we investigate methods for improving upon the syntax-free linearization model designed by Schmaltz, Rush, and Sheiber (2016). We improve the model of Schmaltz et al. by elaborating on the model's simple LSTM, adding several encoder mechanisms and a self-attention system. While our baseline model yielded a BLEU score of 20.4 on our NLTK Dataset, we were able to achieve a BLEU score of 25.1 using a 3-convolutional-layer CNN encoder.

1 Introduction

Linearization is the task of ordering a set of words into a grammatical sentence. Improving linearization systems is important, as this task has versatile applications in the fields of NLP and linguistics, ranging from providing inputs to translation systems, to helping language-learning students improve their word ordering.

Traditionally, techniques for the task of linearization have been statistical in nature, utilizing discrete models with syntactic features (e.g. transition-based parsing)[6]. Because re-ordering a bag of words into a grammatical sentence seems to be naturally dependent on creating a correct grammatical structure, dependency trees were adopted as the default method of linearization. With the advent of the LSTM, however, linearization changed dramatically. In 2016, Schmaltz, Rush, and Sheiber designed a syntax-free linearization model that outperformed previous syntactic systems by up to 11.5 BLEU points [1]. This model took an unordered set of words as input, ran the word embeddings through an LSTM, and output an ordered sentence. The model's performance was surprising, as it only used information from the word embeddings themselves and did not utilize auxiliary information relating to grammar or syntax. These promising results led us to believe that creating a more extensive surface-level linearizer could reach the new state of the art in the linearization task.

As a result, in this paper we build upon Schmaltz et al.'s approach by adding an encoder as well as attention to a baseline LSTM language model. In our experiments, we compare the performance of an LSTM encoder with a CNN encoder and observe marked improvement over the baseline LSTM both in terms of re-ordering longer sentences correctly and in terms of dealing with punctuation and numbers.

2 Related Work

The current state-of-the-art in linearization is closely related to the model of Schmaltz et al. It is a syntactic linearization model designed by Song, Zhang, and Gildea (2018)[4], which generates output sentences along with their syntactic trees. Following the structure of dependency parsing in the vein of Chen and Manning (2014)[3], transition-based syntactic linearization utilizes a stack of partially-built syntactic trees and a set of incoming unordered words to predict the next best action.

This allows for the construction of the correct syntax tree, and produces a prediction of the correct ordering.

Although the Song et al. model performs slightly better than the Schmaltz et al. model, we augment the Schmaltz et al. model for several reasons. While the syntactic linearization model performs better than a simple LSTM using a beam search with a beam length of 1, its performance is barely an improvement over the simple LSTM when using a beam search with a large beam length, such as 512. Furthermore, the current state-of-the-art surface-level implementation is a simple 2-layer LSTM with no additional attention model or encoding (Schmaltz et. al)[1]. For this reason, we choose to explore the surface-level model, which does not generate syntactic trees but rather uses surface information, i.e. the words in the sentence. Our hope is that this approach may become the state-of-the-art, as it does not require dependency parsing and therefore avoids error introduced by automatic parsing. Additionally, a surface-level model is more lightweight and can easily be trained with large amounts of data. It is able to train more quickly, and on more diverse data, since it does not require the use or generation of syntactic trees. For these reasons, we use the approach of Schmaltz et al. as our baseline.

3 Approach

3.1 Baseline Training

Our baseline is the LSTM model used by Schmaltz et al.[1], which is a simple decoder. While the original model was implemented in Lua, we re-implement our own version of it in PyTorch and Python. We only use the higher level ideas from their paper to inform our baseline and otherwise code everything ourselves. Our code modifies makes use of several of the functions in the run.py and vocab.py files from assignment 4 as a basic framework to train our model, as our system is very similar to the NMT decoder from that assignment. We also add our own new vocabulary, and additionally add new modules to define our embeddings and baseline LSTM, CNN, LSTM-encoder and attention models.

Our LSTM Baseline architecture uses a PyTorch LSTMCell instead of an LSTM module as we perform customized calculations (e.g. we integrate an attention network which requires a calculation at each LSTM step) on the output projections of each hidden state.

During training, the model takes in a tensor of word embeddings from the target (grammatically ordered) sentence. This tensor is then split into many smaller tensors, one for every input word. The tensor representing the t^{th} target word is $\mathbf{Y}_t$. We input $\mathbf{Y}_t$ and the previous state of the LSTMCell, $\mathbf{s}_{prev}$ into the Cell to produce a new state, $\mathbf{s}_{curr}$:

$$\mathbf{s}_{curr} = LSTMCell(\mathbf{Y}_t, \mathbf{s}_{prev})$$

We then project $\mathbf{s}_{curr}$ into a vector $\mathbf{p}_{word}$ of the size of the model's vocabulary using a linear layer:

$$\mathbf{p}_{word} = \mathbf{W}_{proj}\mathbf{s}_{curr}$$

Next, we apply the softmax function to calculate the probability of each potential word. This gives us a probability vector $\mathbf{p}_{words}$. This probability vector is then appended to a list of previous probability vectors. Once all of the target words have been passed in to the LSTM Cell, we have a tensor, $\mathbf{P}_{allwords}$ of $\mathbf{p}_{words}$ vectors from each step. We then use this tensor to calculate the loss and update the gradients. We use a cross-entropy loss function between $\mathbf{P}_{allwords}$ and the one-hot vectors representing each target word.

3.2 Baseline Decoding

When decoding, we make use of two different decoding algorithms. The first is a greedy decoder, which takes the maximum probability word at each step of the LSTM and then feeds that as input into the next step. This algorithm terminates when the predicted sentence reaches the length of the unordered source sentence, or "bag of words", and it returns the predicted sentence. The second decoding algorithm is a beam-search decoder which maintains a list of k best hypotheses, or partially complete sentences, as it progresses. Once the beam search is complete, we choose the most probable output sentence from our current beams. We evaluate our predicted sentences against the reference

sentences using the NLTK `corpus_bleu` function. We spent a long time implementing various forms of beam search until we finally settled on our best approach. The first approach we tried was a priority queue that first sorted by beam length and then by log probability, as that would guarantee all the beams of shorter length would be processed first. We realized, however that this had a runtime of $O(k^n)$, where k is the number of hyptoheses we choose at each layer, and n is the number of layers of the search (which in this case would be the length of the source sentence). As a result, we then switched to a simple beam search that processed a queue of k beams every layer, then obtained their successors and chose the k most likely of them to continue the search. This version of the algorithm only has run-time $O(nk^2)$, which is much faster.

During decoding, we mask out all words in the vocabulary that are not in the unordered source sentence. While decoding, we continue to mask out words that we have already predicted, in order to avoid repetition. We only apply this mask while decoding because during training, we want the model to penalize the use of words not in the sentence, and calculate a differentiable loss. Additionally, we want to avoid overfitting during training.

We will now describe the masking procedure in further detail. At the start, we generate a mask $\mathbf{M}$, which is set to 1 at the vocabulary indices of the words in the input "bag of words," i.e. the unordered source sentence. The mask is set to $-\infty$ at the indices of words that are not in the bag. We do this in order to make the softmax probability of these indices equal to zero, as we do not want to predict them. Once the a word is predicted, we then set its index to $-\infty$ so that we do not predict it again. However, if this word is repeated in the sentence, then we do not set its index to $-\infty$, as we will need to predict it again in a subsequent step. We did this by creating a list of indices of length equal to the vocabulary, where each index stores the number of times its corresponding word appears in the "bag". Every time a word is predicted, we decrease its corresponding count until the count reaches zero, which is when we set the mask to $-\infty$. This approach also addresses the problem of multiple $<$ `unk` $>$ tokens in the bag of words.

3.3 LSTM Encoder

The baseline model is a single-layer LSTM and does not include a hidden layer. We implement an LSTM and with a multi-layer bidirectional encoder as a first elaboration on the Schmaltz model. The addition of an encoder improves the model by replacing the randomly initialized state in the LSTM with a "smart" initial state yielded by the encoder, which is comprised of the projection of the concatenated hidden states associated with the top two layers of the multi-layer encoder.

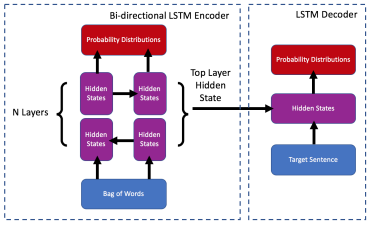

Figure 1: Bi-directional LSTM encoder with N layers and LSTM decoder

3.4 CNN Encoder

The CNN encoder takes in the bag of words in tensor format and then run it through multiple convolutional layers in order to produce a hidden state vector that can be used as the initial state of the baseline LSTM model. As with the LSTM encoder, adding a "smarter" initial hidden state that has learned something about the bag through the CNN enables our model to better linearize it. We experimented with multiple different CNN models, which are described in the experiments section below. The best model was a 3-layer convolutional neural network with the following structure:

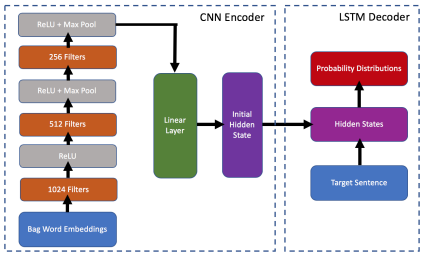

Figure 2: 3-Layer CNN encoder and LSTM decoder

Input→ **1024**→ relu→ **512**→ relu→ maxpool→ **256**→ relu→ maxpool→ linear→ out

The above numbers indicate the number of output filters in a convolutional layer. All of the convolutional layers were based off of a simple CNN classifier obtained from a Deep Learning Overview Lecture by Chris Manning and Russ Salakhutdinov [5].

3.5 Self-attention

Inspired by the multiple uses of attention in the Attention is All You Need paper on transformer models by Vaswani et al. [2], we decided to implement self-attention on our decoder LSTM. This means that each hidden state incorporates attention information from the word embeddings of all of the previous words that the model has predicted. We utilize simple multiplicative attention, and concatenate our attention output with each hidden state. We then perform dropout on the projection of our concatenated hidden state in each step of decoding. During training the model calculates attention on all words in the target sentence, in the past and future. During decoding, however, we mask out the future to only look at the past. This is comparable to how we train the decoder LSTM on a non-masked vocabulary, as described above.

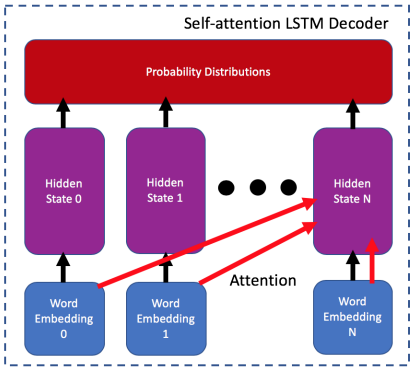

Figure 3: LSTM decoder with self-attention

3.6 UNK Replacement

One of the main problems we encountered when training our models was the presence of $< unk >$ tokens in the output. Not only do these severely impact the BLEU scores as they break up larger n-grams, but they also cause our output to differ from the intended task, which is to re-order the words in the input bag of words. If there are $< unk >$s present, then we are not exactly outputting what we specified we would, and as a result we decided to implement an $< unk >$ replacement scheme so that the output of our model would contain all of the words from the input bag. Due to time constraints, we were not able to implement any advanced models such as a Char-LSTM or a pointer network, and so settled on random $< unk >$ replacement in the decoding stage. This replacement scheme generated a list of words that mapped to $< unk >$ for each source sentence and then randomly inserted them into the $< unk >$ slots produced in the output sentence. We set a fixed random seed (0) so we could compare this process between our different models, and it lead to a relatively consistent increase in BLEU score for all tests.

4 Experiments

4.1 Data

We used a combination of three corpora provided by NLTK, Python's natural language toolkit. These are the Gutenberg, Brown, and Reuters corpora. The Gutenberg corpus contains a small selection of texts from the Project Gutenberg electronic text archive, which contains over 25,000 electronic books. The full archive can be found at `http://www.gutenberg.org/`. The Brown corpus was created in 1961 at Brown University and contains text from 500 sources and multiple genres. Finally, the Reuters corpus contains news over 10,000 new documents, totaling over 1.3 million words. Using these three corpora in combination will give our model exposure to a wide range of English language usage and hopefully allow it to learn sentence re-ordering as well as possible.

When creating our list of sentences from the Gutenberg, Brown, and Reuters corpora, we omit sentences that have a length greater than 20 tokens. A sentence's set of tokens is comprised of its words and punctuation markings, so the sentence length is equal to the number of words and punctuation markings. We divide the full list of 96,805 sentences into a 97/0.5/2.5 train/dev/test split. For each of these sentences, which serve as our target sentences, we create an unordered source

sentence by shuffling its tokens. We use these source sentences as input to the model, and we use the target sentences to calculate the loss.

4.2 Evaluation Method & Experimental Details

We evaluate our parsed sentences using a BLEU score metric comparing the predicted sentence with its target sentence. For all of our training, we use a learning rate of 0.001, and we train our model on an Azure Standard NV6 virtual machine for 30 epochs on our train set. We conducted several initial experiments to determine what model variants we should use for our final comparison.

4.3 Results

Baseline: We trained our baseline on the full dataset. Our baseline yielded a BLEU score of 20.4 on 2420 test examples. Our baseline performance serves as the comparison point for our other experiments. While Schmaltz et al. were able to get a BLEU score of 40 on their LSTM model using a beam width of 64, we are training on a more heterogeneous corpus drawing from multiple genres and time periods (instead of just the Wall Street Journal). Our more heterogeneous corpus increases the number of $<$ `unk` $>$s that appear in our testing set, thereby lowering the BLEU score.

LSTM Encoder: We trained and tested our bidirectional LSTM encoder on the medium-sized corpus (containing 970 training examples) using n stacked LSTMs, and we compared the BLEU scores. Figure 4 indicates that the two-layer bidirectional LSTM yielded the highest performance, with a BLEU score of 9.59.

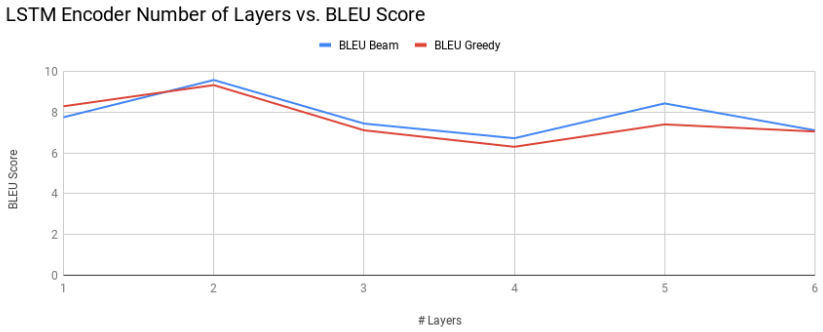

Figure 4: Effect of search algorithm and number of layers on BLEU score with LSTM encoder

Greedy vs. Beam Search: Additionally, Figure 4 shows that a beam search with beam size of 2 yields slightly better BLEU scores than greedy search does. On average, the beam search BLEU score was higher by 0.26. We additionally compared the performance of the baseline, CNN-3, LSTM encoder, and CNN + Attention in Table 1 using both greedy search and beam search, finding that beam search typically yielded higher BLEU scores. We only used a beam size of 2 for our testing due to time constraints. However in the future, we would like to test with larger beam sizes, up to the 256 and 512 that Schmaltz et al. used in their model [1].

CNN Encoder: We compared the performance of several CNN encoders using n convolutional layers using the medium-sized corpus (see Figure 5). We found that the CNN encoders that had 3 and 6 convolutional layers yielded the best performance on this dataset. The 6-layer CNN had a BLEU score of 13.26 when using UNK replacement, and the 3-layer CNN had a BLEU score of 12.89. We followed up with a second experiment comparing the performance of the 3-layer and 6-layer CNN on a 10,000 sentence dataset. In this experiment, the 3-layer CNN yielded a BLEU score of 14.18 in 715.15 seconds of training, while the 6-layer CNN yielded a BLEU score of 12.85 in 786.61 seconds of training. For this reason, we conducted later experiments using a 3-layer CNN. See Appendix 1 (A.1) for a table of CNN architectures and corresponding BLEU scores.

UNK Replacement: On average, UNK replacement yielded an average BLEU score improvement of 1.64 over no UNK replacement in the experiments shown in Figure 5. This is expected, as it increases

Size	BL G	BL B	LSTM G	LSTM B	CNN G	CNN B	CNN+A G	CNN+A B
Small	**9.65**	8.46	9.34	9.59	8.49	8.89	6.68	9.59
Med	13.11	12.42	11.50	10.95	14.66	**15.38**	3.63	6.57
Large	18.79	20.4	18.98	20.19	24.17	**25.06**	3.07	4.29

Table 1: Comparison of beam (B) and greedy (G) search on baseline (BL), LSTM-2 encoder (LSTM), CNN-3 encoder (CNN), and CNN-3 encoder with attention(CNN+A), using BLEU score. Small dataset contains 970 training sentences, Med dataset contains 9700, and Large contains 93901.

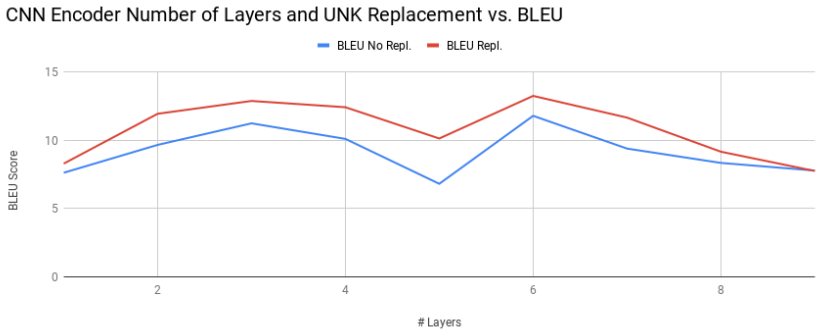

Figure 5: Effect of $< unk >$ replacement and number of layers of CNN encoder on BLEU score

the number of correct unigrams in a sentence originally containing UNKs and possibly increases the bigram, trigram, and 4-gram counts if the randomly selected word is correct.

Highway Layer: Using the small dataset that contains 970 training sentences, we conducted one experiment using a highway layer with our 3-layer CNN encoder. Averaging over five different trials each, our BLEU score for the CNN that used the Highway layer was 6.57, and the BLEU score for the CNN that did not use the Highway layer was 7.51. For this reason, we chose to omit the Highway layer in further experiments.

Attention: The introduction of attention was postulated to improve the performance of our models by allowing for previously decoded words to influence decoding steps in each bag of words. We test attention on our different encoding architectures and found that it yielded a good score on our small dataset, but lower BLEU scores on medium and large datasets, potentially due to over-fitting on the training dataset.

Model Comparison: As shown in Figure 6, CNN-3 without attention yields the highest BLEU score on the full dataset. The LSTM encoder performs similarly to the baseline, and the CNN model with attention yields a lower BLEU score.

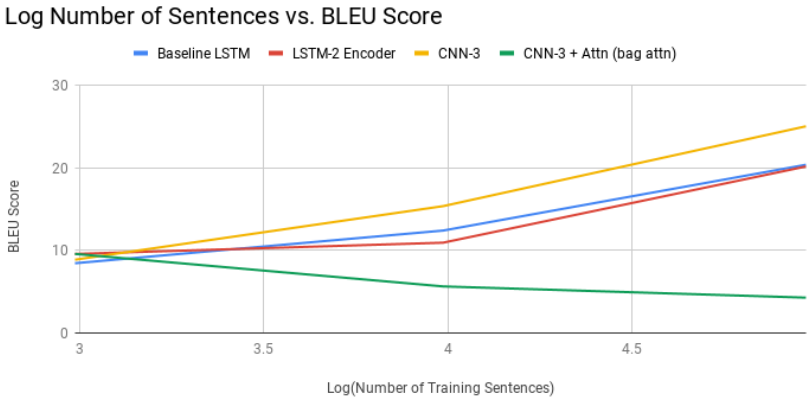

Figure 6: Comparison of different models on datasets of varying sizes. CNN-3 without attention performs best.

5 Analysis

The following table contains various outputs from our best CNN model that illustrate where it succeeds, and where it goes wrong.

CNN-LSTM, Baseline LSTM & Target Sentence Comparison				
No.	**Baseline LSTM Output**	**CNN-LSTM Output**	**Target Sentence**	**Evaluation**
1	" How can she be so happy !"	" How can she be so happy !"	" How can she be so happy !"	Perfect. Short sentence. Same as baseline.
2	(interrupting).	(interrupting).	(interrupting).	Perfect. Learned punctuation. Same as baseline.
3	He was that . its ceiling denied production Opec exceeding agreed	He denied that Opec was exceeding its agreed production ceiling .	He denied that Opec was exceeding its agreed production ceiling .	Perfect. Medium length sentence. Significantly better than baseline.
4	The two of three , and children . 2 : 4 hundred seventy Shephatiah	2 : 4 The children of Shephatiah , three hundred seventy and two .	2 : 4 The children of Shephatiah , three hundred seventy and two .	Perfect. Medium length and numbers. Significantly better than baseline.
5	The bank said the will be the early July of August . end by central or has dropped controls	The central bank has said . controls the will be dropped by the end of August or July early	The central bank has said the controls will be dropped by the end of July or early August .	Good. Long sentence. Significantly better than baseline.
6	" It is the exchange of ," said that rates . rates system under floating Sumita fluctuate inevitable	" It is inevitable that exchange rates fluctuate under the system of floating rates ," Sumita said .	" It is inevitable that exchange rates fluctuate under the system of floating rates ," Sumita said .	Perfect. Very long (18 words). Significantly better than baseline.
7	It said the new process , xylene and xylene , include isomerization hydrodealkylation . units fractionation extraction thermal aromatic BTX	It said the new units and include hydrodealkylation , isomerization , xylene xylene . extraction process thermal aromatic fractionation BTX	It said the new BTX process units include aromatic extraction , xylene fractionation , xylene isomerization and thermal hydrodealkylation .	Bad. $< unk >$ problem. As bad as baseline.
8	the buoyed of a round of this new . interest rate cuts have market further possible stock week Thai Reports	buoyed of the new market have a further this week of possible cuts round interest rate . stock Thai Reports	Reports of a possible new round of interest rate cuts have further buoyed the Thai stock market this week .	Bad. Long sentence backwards. Marginally better than baseline.

As can be seen from the table above, our CNN-LSTM model was able to produce some impressive results on all ranges of sentence length. We include the output of the baseline LSTM for comparison. We found that the baseline LSTM tended to perform poorly on long sentences, so this comparison demonstrates the improvement achieved by the CNN encoder.

As illustrated in examples 1 and 2, the model is able to perfectly re-order most shorter sentences, including those with punctuation. Examples 3 and 4 show the model working perfectly on medium length sentences as well, and even one involving numbers. This model shows marked improvement over our baseline, which struggled to reproduce even one medium-length sentence perfectly. Examples 5 and 6 show the model's good performance on the longest sentences in our test set. Example 5 is better-than-average, as far as long sentences go, and example 6 shows that the model is capable, in some instances, of perfectly re-ordering a sentence as long as 18 words (which is just 2 under our maximum sentence length of 20). Examples 1 to 6 show us that the model has effectively learned

how to re-order sentences, even when they contain multiple items of punctuation, numbers, and rare words.

However, as examples 7 and 8 show, the model does break down in certain cases. More specifically, it does not work well in two cases, which are the presence of multiple $< unk >$ tokens in the output sentence, and very long sentences. In the first case, when our model predicts multiple $< unk >$ tokens it almost always produces a nonsensical output because it uses random $< unk >$ replacement. In the second case, despite certain cherry-picked examples of long sentences being correctly re-ordered, there are also a lot of sentences that do not get re-ordered correctly. This is a challenge of LSTM models, as they progressively break down with increasing input length due to the vanishing gradient problem. We hoped that adding in self-attention would give the model the capability to reach back arbitrarily far into the set of previously predicted words at each time step. However, our current implementation did not result in such improvements. Both of these failure cases can be remedied by extending and improving the model, as suggested in the conclusion section below.

6 Conclusion

Our work has demonstrated that surface-level linearization models are capable of achieving high scores without syntactic information, and that extensions to the LSTM Baseline model developed by Schmaltz. et al significantly improve performance on this task. In particular, a convolutional encoder works better than an LSTM encoder for obtaining a hidden state representation of a bag of words due its non-sequential structure. We were limited by time constraints that stopped us from further improving our self-attention mechanism and from adding in more sophisticated unk replacement models. Moreover, if we had more time we could run our model on the Penn Treebank dataset that Schmaltz et al. used to train and test their model. This would allow us to directly compare the performance of our model with theirs and the previous syntax-based models, and thus determine whether we have achieved state of the art. Despite the difference in datasets, however, we believe that our extended LSTM model is a signficant step-up from the LSTM baseline, and thus has the potential to produce the best scores out of all the surface linearization models developed to date if it were run on a different data set.

Avenues for further work include improving self-attention, implementing a multi-layer LSTM for our decoder, implementing a transformer model for linearization, and creating a pointer-generator networks and Char-LSTM decoder that will help fix the $< unk >$ problem. There are many more potential avenues for continued research not mentioned here, and we believe we have just begun to tap into the potential of surface-level linearization models.

7 Additional Information

Mentor: Michael Hahn

References

[1] Alexander M. Rush Allen Schmaltz and Stuart Shieber. Word ordering without syntax. *In Conference on Empirical Methods in Natural Language Processing(EMNLP-16). Austin, Texas*, pages 2319–2324, 2016.

[2] Niki Parmar Ashish Vaswani, Noam Shazeer and Jacob Uszkoreit. Attention is all you need. *NIPS 2017. Long Beach, California*, pages 6000–6010, 2017.

[3] Danqi Chen and Christopher Manning. A fast and accurate dependency parser using neural networks. *Conference on Empirical Methods in Natural Language Processing. Doha, Qatar*, pages 740–750, 2014.

[4] Yue Zhang Linfeng Song and Daniel Gildea. Neural transition-based syntactic linearization. *INLG 2018 (International Natural Language Generation Conference). Tilburg, Netherlands.*, 2018.

[5] Christopher Manning and Russ Salakhutdinov. The deep learning revolution. *JSM 2018. Vancouver, British Columbia*, page Slide 10, 2018.

[6] Wanxiang Che Yijia Liu, Yue Zhang and Bing Qin. Transition-based syntactic linearization. *In Conference on Empirical Methods in Natural Language Processing(EMNLP-15). Denver, Colorado*, page 113–122, 2015.

8 Appendix

A.1: CNN Configurations and BLEU Score Table

CNN Configurations & BLEU Scores	
LSTM Output	**Target Sentence**
Input -> 1024 -> relu -> 512 -> relu -> max pool -> 256 -> relu -> 128 -> relu -> max pool -> 64 -> relu -> 32 -> relu -> max pool -> 16 -> relu -> 8 -> relu -> max pool -> 4 -> relu -> max pool -> linear -> hidden size output	7.76
Input -> 1024 -> relu -> 512 -> relu -> max pool -> 256 -> relu -> 128 -> relu -> max pool -> 64 -> relu -> 32 -> relu -> max pool -> 16 -> relu -> 8 -> relu -> max pool -> linear -> hidden_size output	9.17
Input -> 1024 -> relu -> 512 -> relu -> max pool -> 256 -> relu -> 128 -> relu -> max pool -> 64 -> relu -> 32 -> relu -> max pool -> 16 -> relu -> max pool -> linear -> hidden_size output	11.67
Input -> 1024 -> relu -> 512 -> relu -> max pool -> 256 -> relu -> 128 -> relu -> max pool -> 64 -> relu -> 32 -> relu -> max pool -> linear -> hidden_size output	13.26
Input -> 1024 -> relu -> 512 -> relu -> max pool -> 256 -> relu -> 128 -> relu -> max pool -> 64 -> relu -> max pool -> linear -> hidden_size output	10.14
Input -> 1024 -> relu -> 512 -> relu -> max pool -> 256 -> relu -> 128 -> relu -> max pool -> linear -> hidden_size output	12.43
Input -> 1024 -> relu -> 512 -> relu -> max pool -> 256 -> relu -> max pool -> linear -> hidden_size output	12.89
Input -> 1024 -> relu -> 512 -> relu -> max pool -> linear -> hidden_size output	11.95
Input -> 1024 -> relu -> max pool -> linear -> hidden_size output	8.3