BiDAF with Dependency Parse Tree for Question Answering in SQuAD 2

Stanford CS224N Default Project

Mohammad Mahbubuzzaman
Department of Computer Science
Stanford University
tachyon@stanford.edu

Abstract

Machine comprehension (MC) is one of the key research areas in the field of Natural Language Processing (NLP). Question-Answering (QA) is a task that can measure progress in MC with some objective consistency. We aim to improve upon the existing BiDAF [1] model provided in our course that addresses the task of QA. We used the same dataset SQuAD v2 [2] as in the original paper and implemented a novel idea on top the the given model to improve EM and F1 scores. Our primary contribution is based on the hypothesize that many downstream language processing models can be made more efficient by augmenting the dataset with syntactic structures learned via other existing models. There are many possible ways to utilize such syntactic information. In this paper we specifically explore the effectiveness of dependency parse tree for the task of QA. We explore a variety of approaches to encode dependency trees to understand which ones work as well as which ones are practical in terms of memory and processing. Our final model is able to significantly outperform the baseline model. We achieved a F1 score 67.7 and 65.77 in dev and test dataset whereas the baseline model gets 60.8 and 58 in the respective cases. Further, we also show that our model trains significantly faster than the baseline model despite the additional data preprocessing required for our model.

1 Key Information

- Mentor: Lauren Zhu.
- No collaborator or sharing.

2 Introduction

Machine comprehension is a central objectives in NLP. Since language understanding is a complex and very loosely defined objective, typically there are more specific goals for models addressing language understanding. Question answering is one such objective and is our focus in this paper. Specifically we use a subset of SQuAD2 [2] dataset for training and evaluating our model. SQuAD 2 is based on SQuAD 1 which includes a collection of (paragraph, question, answers) triplets as train dataset. The paragraphs are taken from Wikipedia and answers are written by humans. The objective of the model is to predict the start and end index of the correct answer, which assumes that the answer, if present, is always an exact subsequence of the context paragraph. The additional challenge with SQuAD 2 is that there are examples where the correct answer does not exist in the context and the model has to predict "No Answer" for such cases.

Latest research in this field includes models that use some variant of attention mechanism like transformers [3]. Another widely popular model is BERT [4], which is designed to pre-train deep

bidirectional representations by jointly conditioning on both left and right context in each transformer layers.

In this paper, we start our work from the given code base which implements a simpler version of the BiDAF [1]. Our goal was to improve it by exploring novel ideas without changing the model architecture fundamentally. As such, the performance comparison with state of the art models was not our primary focus. Our main idea utilizes dependency parse trees for creating a better embedding of the context and question words. The second part of our original contribution is in the attention layer of the existing model. In addition to this we implemented character level encoding to further improve our model. But this is strictly an implementation effort of existing ideas. Overall, we got some very interesting results which shows clear advantage of using dependency tree which is further enhanced by character level embedding. Our model not only scored higher for AvNA, F1, EM metrics but trained significantly faster on the same hardware compared to the baseline model.

3 Related Work

We used the provided code base that as the starting place for for our model. The base model implements a simpler version of the Bi-Directional Attention Flow (BiDAF) [1] model.

Significant breakthroughs have been made in the field of natural language processing in the last few years since BiDAF was published. Some of the key advancement includes the idea of Coattention [5] which adds a second level attention over representations that are themselves attention outputs; Conditioning end prediction on start prediction [6]; Span representation [7], where the model directly computes the probability of each span (as opposed to separate star and end predictions); Combining Local convolution with global self-Attention [8] and other variants [9] [10] of transformer [3] based approaches.

Since our work is based on BiDAF, we briefly describe here the research works that was relevant in that timeline. A relatively newer idea at that moment was neural attention mechanism which allows the model to focus on specific spans within the input paragraph [11]. The main idea for attention involves computing weights to extract the most likely candidate range from the context which is usually done by first creating a fixed sized summary vector for the context. Then some sort of uni-directional attention from query to the context is learned.

The BiDAF model introduces three main improvements to the existing attention mechanisms. First, unlike some of the other approaches BiDAF, preserves hidden states of all time steps to be utilized by the next layer. Second, the attention is memoryless; meaning it only looks at the current query and context states and not of the previous steps. Their rationale is that this lets the attention deal with the relation between query and context and leave the inter-context relationships up to the LSTM based modeling layer to discover. Third and possibly the most defining one is that attention is computed in both directions, from query to context and from context to query. We found this to be the most interesting of the three points, which also seems very intuitive and relatively simple.

Although Deep Learning models usually need less feature engineering, using the right input features can still boost performance significantly. The DrQA [12] model is interesting to us, because it also utilized language features such as Part of Speech (POS) and Named Entity (NE) tags. While these features were also in our plan to explore, due to time constraints we ended up focusing exclusively on exploiting dependency parse tree which is also a form of feature engineering. Feature engineering also has its limitation including the extra work required to create features. We reflect upon the advantages and drawbacks more in the results and analysis section.

4 Approach

Our model is based on the hypothesis that a key part of understanding language is about understanding the syntax or grammar of it. The meaning of a sentence can be argued to depend on at least two factors - what words are in it and how they are organized relative to each other. The first part can be addressed by learning embeddings for words in a context-less or global context - which is what we usually have from pretrained word vectors. But the second factor is critical as it alters the meaning of the same word in different sentences. Our hypothesis is that to account for this second factor, learning syntactic composition of sentences can be a critical element. This is especially

applicable for a relatively smaller dataset like ours (130, 000 examples, and even fewer count of unique paragraphs). Because with limited dataset it is very difficult or unlikely to be able to learn any factual or causal interpretation of the text. Questions that require even a rudimentary level of reasoning can be practically impossible to learn with smaller dataset. Based on this observation and exploring model output for various examples, we think most of what is learned with small datasets has to do with learning how the sentences are structured grammatically. Specifically, the model learns a function between the grammar of a question with the grammar of the context or the answer-sentence. For example, for a question that begins with "Where", the model could learn a high probability of selecting a phrase in the context followed by words such as "at", "to", "in" etc. Since dependency parse trees provides a very close version of that same kind of information with only better accuracy, it can boost the final objective of the model significantly. This is in fact what we observe with our model.

In this paper we explore a novel approach to improve the accuracy of the base model by utilizing dependency parse tree. Specifically we use dependency parse tree generated by Spacy and process them further so that we can efficiently incorporate it in the embeddings of context and question tokens. Since most of the successful models with attention layers also learns contextual relationship between words, we think that adding dependency information between words loosely amount to adding a type of *grammar aware attention*.

Dependency Tree A dependency tree is a special kind of tree graph that describes grammatical relationship between words and phrases within a sentence. It has a specific node as the root $ROOT$. Every non leaf node has one or more children and each node has exactly one parent. Root's parent is itself. An edge between two nodes is defined as the dependency tag that connects a parent node to a child node. For a list of dependency tags, refer to appendix A.

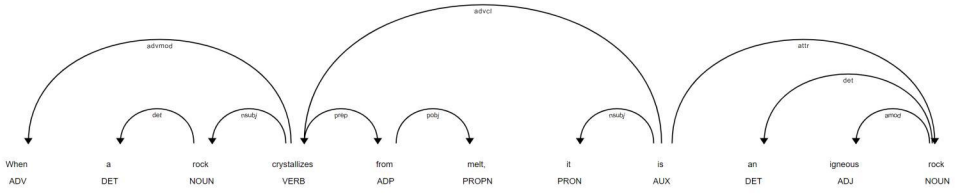


Figure 1: Dependency parse tree for "When a rock crystallizes from melt, it is an igneous rock"

Our original contribution in the model is primarly in the embedding layer and in the attention layer. However most of the work done to create additional input features are done outside of the model and in the setup or preprocessing layers. Following are the additions to the existing model.

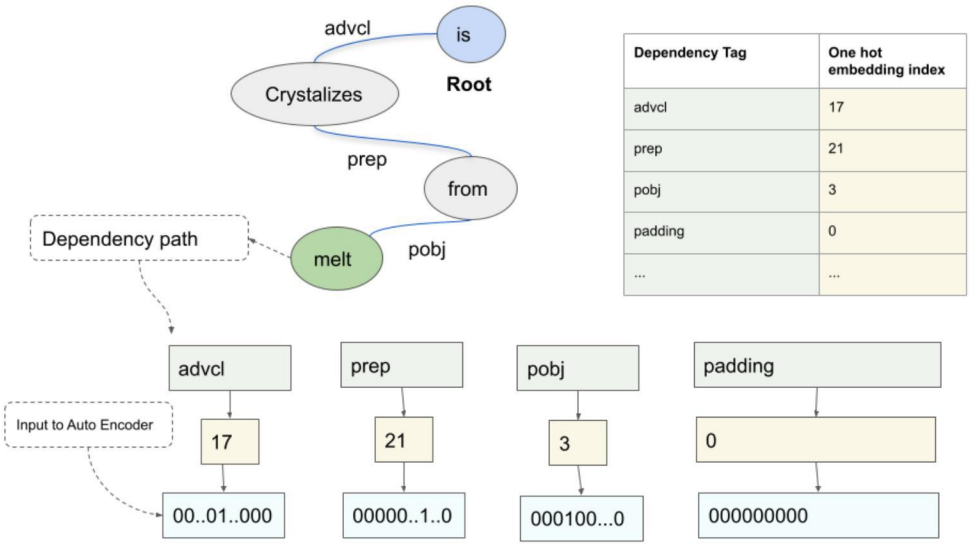
4.1 Encoding Dependency Tree Efficiently for Deep Learning

Our goal is to incorporate this tree somehow to the model so that it can use this information along with existing attention layers to learn a relationship between the syntax of a question with that of the answer sentence. Since we can't just feed the tree directly which is unsuitable for deep learning based algorithms, we come up with the following strategy. To be compatible with the existing layers of the model, we need to embed this tree in a token by token basis. That is we should store additional features for each of the context token that encodes the structure of the tree. To that end, we store for each node the path from itself to the root. The way we assess whether this encoding scheme faithfully represents the entire tree is by observing that - the original tree could be completely recovered from the new representation. We explored other methods which are more computationally expensive and infeasible in our context. But we provide an analysis of that in the experiment section.

4.2 Path Summarizing using Auto-encoder

As described above, our model needs to store paths from each node (a.k.a token, context word or simply word) to the root. A dependency path, or path is defined for two nodes as:

$$path(u, v) = [edge(u, k_1), ..., edge(k_n, v)]$$



Dependency Tag	One hot embedding index
advcl	17
prep	21
pobj	3
padding	0
...	...

Figure 2: Dependency Path Generator

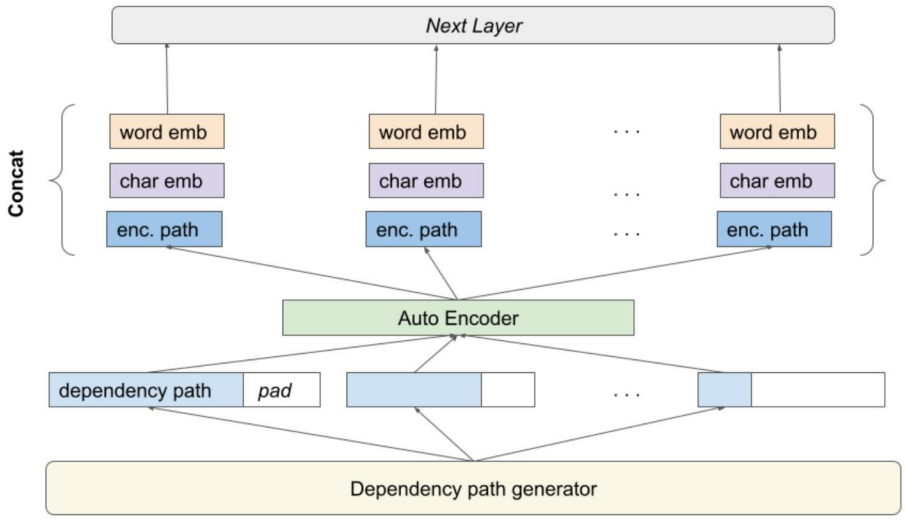


Figure 3: Embedding Layer

$$= [dep\_tag(u, k_1), ..., dep\_tag(k_n, v)]$$
$$dep\_tag(u, v) = one\_hot(tag_{u \to v})$$

where k_1 through k_n are intermediate nodes between the path. $dep\_tag(u, v)$ returns a string representing the dependency relationship between u and v.

There are 44 different dependency tags (Full listing: Appendix A) provided by the library we used for parsing. To create a vector representation of these tags, we defined a one hot embedding for these set of tags. The rational is that, without prior knowledge they are just mutually independent classes. We tried two variations with the embedding where it is either allowed to be trained or kept frozen.

But these paths are of variable lengths and to be efficient we need them to be of fixed lengths. This posed a unique challenge for our model as simply padding them would be very undesirable as it will imply each index in that sequence has a special meaning which they do not. These paths are sequences of edges and they should be learned in the same manner sequences are normally learned,

which can be a Recurrent Neural Networks (RNN). We in fact started with an RNN based solution which didn't work as good and is explained in the experiments section.

The solution that worked great is auto-encoder. It is the perfect tool for compressing vectors when we do not have an explicit learning objective. Regardless of what the downstream task is, auto-encoders at least creates a representation that has the input features encoded in some fashion with arbitrarily small loss, because the decoder has to reconstruct it from the compressed encoding.

$$encoded\_path(c, root) = auto\_encode(dep(c, parent(c)), ..., dep(k, ROOT))$$
$$reconstructed\_path = auto\_decode(encoded\_path)$$

The main advantage over RNNs is that we can give training feedback to the auto encoder directly on its ability to reconstruct input and do not have to learn it from the question answering task which provides learning feedback from far down the layers.

$$Loss = Loss_{original} + MSE(path, reconstructed\_path)$$

Concretely, we added a new loss term for the model. It computes Mean Squared Error (MSE) loss for the input path vectors and reconstructed path vectors that come out of the embedding layer. To account for the variable length paths, we simply pad to make them fixed size. Since auto encoder faithfully reconstructs the original input, padding is not problematic like it is with RNN.

Finally we concatenate this encoded dependency paths for each token to their respective word embedding and project it to the matching dimension for the next layer.

$$c^{new}_{embed} = [c_{embed}; encoded\_path(c, root)]$$

4.3 Modified Attention

We modified the attention layer to shrink the context-to-query and query-to-context vectors and expanded the query aware context vectors. This improved the overall EM score by about 1.5% in the dev dataset. The motivation for doing it comes from a very unexpected finding that we discuss in depth in the analysis section.

This concludes our original contribution. The character level embedding is a known idea to improve performance and the only reason we implemented it is to see if any of the original contribution interferes with the gain expected by character level embeddings.

4.4 Character level Embedding with CNN

We implemented a character level embedding and concatenated it to the existing embedding to incorporate morphology inspired embeddings. This is not our original contribution. We implemented two layer 1 dimensional convolution which boosted the model performance further. Specifically the idea of concatenating two convolution outputs instead of just 1, was taken from here: `https://github.com/tomassykora/bidaf-question-answering/blob/master/paper.pdf`

5 Experiments

5.1 Data

In addition to the SQuAD dataset provided for the course we utilized additional preprocessing scripts that uses Spacy to create additional features. This includes information necessary to build the dependency tree for each context and questions.

5.2 Evaluation method

Performance is measured via two metrics: Exact Match (EM) score and F1 score.

Exact Match is a binary measure (i.e. true/false) of whether the system output matches the ground truth answer exactly.

F1 is a less strict metric – it is the harmonic mean of precision and recall.

For training and validating models locally, we used the train and dev dataset as provided by the course.

5.3 Experimental Details

We focused more on exploring different models and novel approaches and less on hyperparamter tuning. Although we did some exploration on parameter tuning whenever we deemed it can have any significant impact on the overall performance.

Following is a summary of the different models we explored, grouped by their shared architecture.

5.3.1 Only using Parent Embedding

Our very first approach simply adds the embedding of parent token for each position. This resulted in a small decrease in the performance. This is due to the fact that summing vectors can destroy features.

$$c^{new}_{embed} = c_{embed} + parent(c)_{embed}$$

We then tried concatenation instead of sum. This resulted in a very small gain. Since the new information did not destroy anything existing it was better than the first approach. But it was still not nearly good enough - most likely because just parent embeddings do not tell enough about the overall tree.

$$c^{new}_{embed} = [c_{embed}; parent(c)_{embed}]$$

5.3.2 All-pair shortest paths for Encoding Dependency Tree

The first approach we implemented for encoding dependency paths includes much more detailed information about the tree. We hypothesized that, the relationship between a pair of words can be defined by the dependency path between them. These paths do not all necessarily go through the roots. Since each node is connected to every other nodes through exactly one path, we compute paths to every other node from a given node and store that in N x N matrix where cell i, j refers to the dependency path between $token_i$ and $token_j$. This is a much more explicit way to represent the tree. The matrix is then used for computing similarity or attention to modify the corresponding token embeddings.

To compute paths between all nodes, we used Floyd Warshall's all pair shortest path algorithm with significant optimization exploiting the fact that our graph is always a tree.

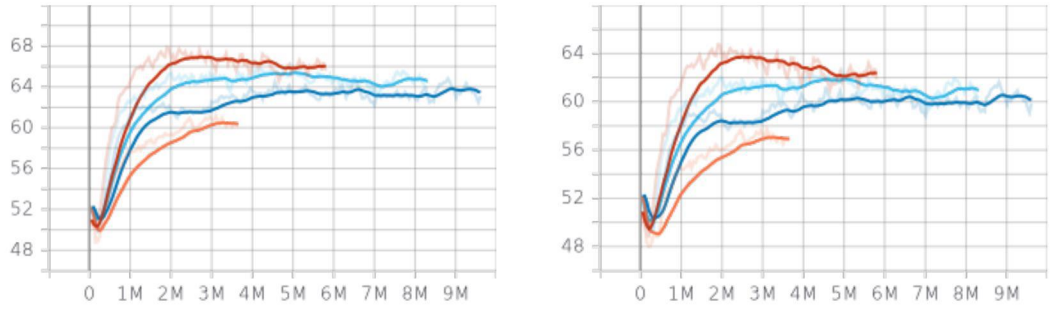
Although it seemed theoretically sound and arguably provides a better representation of the tree, it turned out to be extremely challenging to keep the time and space required to handle this within an acceptable limit, even after a series of optimizations. We were unable to do any significant amount of training as the setup time itself was completely prohibiting. However, some of these issues are recoverable given enough time and we plan to revisit this in future work.

5.3.3 Path Summarizing using RNN

Our first attempt of summarizing dependency paths involved treating them as sequences and employed a many-to-one sequence learner using single-layer, single-directional LSTM based RNN. The input to this RNN are path sequences of variable lengths and the outputs are fixed size vectors, one for each path.

$$encoded\_path(c, root) = LSTM(dep\_emb(c, parent(c)), ..., dep(k, ROOT))[-1]$$
$$edge(u, parent(u)) = one\_hot(dependency\_tag(u, v))$$

The subscript [-1] refers to the fact that we only extract output from the last time step of the LSTM. We use only the final output as this is a many-to-one problem where a sequence needs to be encoded in one fixed size vector.



(a) Dev F1 Score (b) Dev EM Score

Figure 4: Red: Final Model, Light blue: Single layer Char CNN, Dark blue: Without Char CNN, Orange: Basline model.

We were not completely satisfied with the performance with this approach and we argue that it was too hard for these RNNs to learn effectively as the feedback coming from the final layers through backpropagation were not enough. We needed a more direct way to train these RNNs. This led to the idea of the auto encoders which proved to be very effective and much faster.

5.3.4 Other tuning

We tried with some variations of hidden sizes that did not produce any notable variations.

5.4 Results

5.4.1 Validation Dataset

Model	EM	F1
⋆BiDAF baseline	57.7	60.8
Embedding x 2	58.9	62
Concat parent emb	58	60
RNN Path Encoding	58	61
Auto encoded Path	61.6	64.8
Auto encode + 1 layer Char CNN	62.2	65.5
Final Model (Auto enc + 2 layer Char CNN)	**64.75**	**67.7**

5.4.2 Test Dataset

Model	EM	F1
BiDAF baseline	55	58
Our Model without Character embedding	59.8	63.15
Our Model with Character embedding	**62.34**	**65.77**

We expected better results with the RNN based encoding. But as mentioned before, it could be suffering from too little feedback due to being too far away from the loss layer.

However the most surprising result is the second row of the validation dataset table. In our initial implementation of adding parent embedding there was a bug causing the embedding of the current node to be added to itself instead of the embbedding of the parent. And this actually improved the scores significantly enough to be ruled outside of statistical error margin. We explain our understanding on this in the analysis section.

The improvement with character level embedding was also impressive. Specially the idea of concatenating outputs from two CNNs instead of just one. This is not our original idea. But we implemented it to see if benefits are still applicable to our modified model.

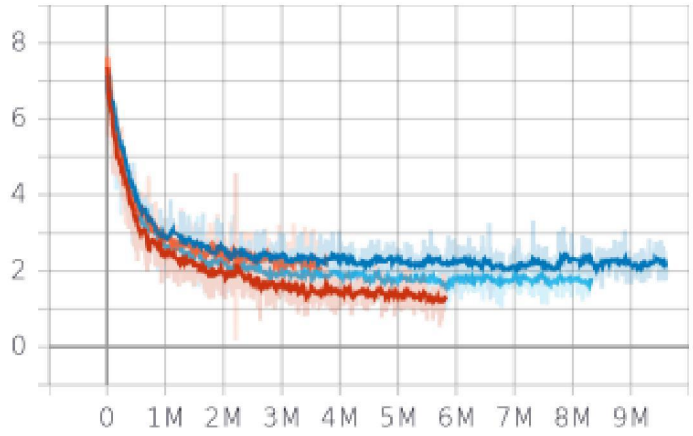


Figure 5: Train error on Dev

In addition to score improvements, we also noticed a drastic speedup of the training convergence. As seen in the figures, our model improves scores at least twice as fast. Even for the variation of the model that doesn't improve scores much, they still train a lot faster than the baseline on the same hardware.

The setup time for our model doubles from the baseline setup due to compute intensive graph processing. Our setup takes about 30 minutes whereas the initial code took only 15 minutes. However since the setup is a one time cost, the overall speedup is still significant.

We believe this speedup in training validates our hypothesis that models like BiDAF learns mostly syntactic mapping when trained with relatively smaller dataset. Since our preprocessing gives a major heads up in that regard, the model finds it very convenient to utilize on this feature during training.

6 Analysis

Apart from the success of our final model, the most striking observation we made resulted from a bug in our initial implementation. While attempting to add the embedding of the parent nodes, we actually added the embedding of the same word; basically multiplying the embeddings by a factor of two. This actually resulted in a noticeable improvement. The EM went up approximately 1% to 59. We initially could not explain it. However, after exploring many ideas and seeing what works, we have a possible explanation for this. For SQuAD 2.0, the No Answer questions are introduced. A lot of the times the model predicts an answer based on significant number of matching common words between question and context words. However, sometimes these matches are misleading. For example, if there 4 words that are common between a context line and question, but there is a word in question, whose antonym is present in the context causing the the meaning of the sentence to flip, the model is still very likely to choose that sentence as the answer. So there would be many false positive (answer given for no answer scenario). With amplifying the context embedding the individual non-attended portion of the embedding gets more weight in the total concatenated embedding. This probably dilutes the attention values to a degree that the sentence is no longer assigned a high enough probability as a candidate answer. We need to run more experiments and explore the theory further to validate it.

This is the motivation for us to modify the attention layer such that the context embeddings (which include only information from contexts and questions separately till that point) occupy more dimensions than the attention outputs (which include the matching between question and context). And we indeed see improvement with this approach which is reflected in the best version of our model.

7 Conclusion

We explored a new approach to improve BiDAF model as provided in this course. The base model is neither the most up-to-date version of the model, nor is this architecture currently the best one. So a direct comparison with any of the current best model on SQuAD is not going to be convincing.

Nevertheless we think the core idea of this paper has potential to open avenues of research that could potentially boost the performance of other more recent models as well. We showed that not only is our model able to outperform the baseline model, it can also learn almost twice as fast. It validates our hypothesis that for smaller dataset, learning syntactic structure is a big part of what a model could possibly do. We have a number of things that we would like explore more on this model. First, we have not yet added any of the other language features such as POS, NER that are known to improve scores. We also would like to experiment with the attention layer more by adding more explicit mechanism to utilize dependency. Because, even though we inject dependency paths in the embedding, the attention layer is still not fully utilizing it. The fact that we were able to boost performance by reducing the attention outputs suggests that there are further room for improvements in that layer.

References

[1] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603*, 2016.

[2] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

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

[5] Caiming Xiong, Victor Zhong, and Richard Socher. Dynamic coattention networks for question answering. *arXiv preprint arXiv:1611.01604*, 2016.

[6] Shuohang Wang and Jing Jiang. Machine comprehension using match-lstm and answer pointer. *arXiv preprint arXiv:1608.07905*, 2016.

[7] Yang Yu, Wei Zhang, Kazi Hasan, Mo Yu, Bing Xiang, and Bowen Zhou. End-to-end answer chunk extraction and ranking for reading comprehension. *arXiv preprint arXiv:1610.09996*, 2016.

[8] Adams Wei Yu, David Dohan, Minh-Thang Luong, Rui Zhao, Kai Chen, Mohammad Norouzi, and Quoc V Le. Qanet: Combining local convolution with global self-attention for reading comprehension. *arXiv preprint arXiv:1804.09541*, 2018.

[9] Zihang Dai, Zhilin Yang, Yiming Yang, William W Cohen, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Language modeling with longer-term dependency. 2018.

[10] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

[11] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

[12] Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. Reading wikipedia to answer open-domain questions. *arXiv preprint arXiv:1704.00051*, 2017.

A Appendix (optional)

A.1 List of Dependency Tags

ROOT, acl, acomp, advcl, advmod, agent, amod, appos, attr, aux, auxpass, case, cc, ccomp, compound, conj, csubj, csubjpass, dative, dep, det, dobj, expl, intj, mark, meta, neg, nmod, npadvmod, nsubj, nsubjpass, nummod, oprd, parataxis, pcomp, pobj, poss, preconj, predet, prep, prt, punct, quantmod, relcl, xcomp