Extractive Question Answering Using Match-LSTM and Answer Pointer

Alexander Haigh, Cameron van de Graaf, Jake Rachleff
Department of Computer Science
Stanford University
Stanford, CA 94305
Codalab username: jakerachleff
{haighal, camvdg, jakerach}@stanford.edu

Abstract

One of the major goals of Natural Language Processing is understanding text, and, in particular, reading comprehension (RC): the ability to read a passage and answer a question about it. The recently released Stanford Question Answering Dataset (SQuAD) is one of the first robust RC datasets and introduces the complexity of variable length and open-ended rather than multiple choice answers. Here, we replicate the match-LSTM with Pointer Net model introduced by Wang and Jiang (2016) and apply their boundary model to directly predict the start and end indices of the answer. Our model shows proof of concept and achieves basic functionality but falls short of the original authors' results. We analyze our results and suggest future model modifications to help achieve or surpass theirs.

1 Introduction

The Stanford Question Answering Dataset is a collection of over one hundred thousand pairs of questions and answers from different articles. Each answer in the SQuAD dataset is a subsequence of the paragraph the original question asks about. With the recent rise of compute power and deep learning, this relatively new dataset has allowed computer scientists around the world to test the bounds of complex comprehension models. Each model created to predict answers is benchmarked against human performance, which performs at 91 percent F1 and 82 percent exact matches.

Successful models for machine comprehension, like many other NLP tasks, often use end to end neural network architectures as they capture a complete idea of the relationships between questions, contexts, and answers. These models have been a vast improvement over previous NLP based models which used techniques such as dependency syntax and frame semantics. The gold standard for such models, published ten days before this paper, performs at a level of 84 percent F1 and 77 percent EM, remarkably close to human performance.

In this paper, we aim to describe the process of implementing such an end to end model. Researchers often leave out key implementation details when discussing their work - we try to replicate their model. Specifically, we will walk through the steps of implementing Wang and Jiang's aforementioned Match-LSTM model. This model is an end to end system that includes preprocessing data with an LSTM, creating an attention vector with a bidirectional LSTM, then feeding the attention vector into a Pointer Net that predicts the start and end locations in the context of the answer. Wang and Jiang successfully recorded an F1 of 77 and an EM of 67 on their best model (granted, as an ensemble).

2 Related Work

The match LSTM model refers to several concepts brought into vogue by past researchers. The implementation of the algorithm relies heavily on the recurrent nature and temporal understanding of Long Term Short Memory, introduced by Hochreiter and Schmidhuber in 1997. In our implementation, this is extended into a bidirectional LSTM model for both preprocessing and matching, which was introduced in 1997 shortly after by Schuster and Paliwal to understand words both forwards and backwards in time. The attention vector created by the bidirectional LSTM furthers work on attention by Manning et. al in NLP in 2015 to understand certain parts of a context and question's relationship in higher focus. Finally, the output layer was based heavily on Pointer Net, introduced by Vinalys et. al in 2015, which allowed certain sequences of words to be chosen as outputs. Our own work is based heavily off of the previously described match LSTM by Want and Jiang. Our model includes dropout, a concept introduced by Srivastava et. al in 2014 to reduce overfitting in neural networks.

3 Approach

The formal task on the SQuAD dataset, as well as the structure of our model, are as follows. Sample question/paragraph/answer triples can be found in section 4.4.

- *Inputs:* A question $q = \{q_1, ..., q_Q\}$ of length Q and a context paragraph $p = \{p_1, ..., p_P\}$ of length P.
- *Output:* An answer span $\{a_s, a_e\}$ where a_s is the index of the first answer token in p, a_e is the index of the last answer token in p, $0 \leq a_s, a_e \leq m$, and $a_s \leq a_e$.

In our model, we represent words as d-dimensional embeddings from the GloVe dataset, which was trained on over 6 billion tokens from Wikipedia and the Gigaword 5 datasets. We use these to represent our paragraph as $\boldsymbol{P} \in \mathbb{R}^{P \times d}$ and our question as $\boldsymbol{Q} \in \mathbb{R}^{Q \times d}$. We then use a three-layer neural network model to predict a_s and a_e: (1) A forward-only LSTM that preprocesses $\boldsymbol{P}$ and $\boldsymbol{Q}$ passages to create contextual encodings of each ($\boldsymbol{H}^p$ and $\boldsymbol{H}^q$, respectively), (2) A bidirectional Match-LSTM that tries to align our question and paragraph encodings, and (3) a Pointer-Net that predicts the start and end tokens of the answer within the passage.

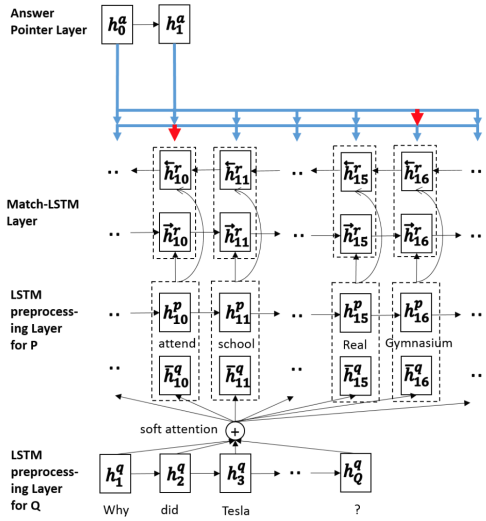

Figure 1: Model architecture. For each Match-LSTM timestep, we compute an attention vector over $\boldsymbol{H}^q$ and concatenate that to $\boldsymbol{H}^p$s value at that timestep. This is then fed into another LSTM (one in each direction) and used to compute β_1 and β_2 (originally appeared in Wang and Jiang 16).

3.1 LSTM Preprocessing Layer

Here, we run standard one-directional LSTMs (introduced by Hochreither and Schmidhuber, 1997) over $\boldsymbol{P}$ and $\boldsymbol{Q}$:

$$\boldsymbol{H}^p = \overrightarrow{LSTM}(\boldsymbol{P}) \text{ and } \boldsymbol{H}^q = \overrightarrow{LSTM}(\boldsymbol{Q})$$

Where $\boldsymbol{H}^p \in \mathbb{R}^{P \times h}$ and $\boldsymbol{H}^q \in \mathbb{R}^{Q \times h}$ (h is the hidden size, the same across all letters). Intuitively, this means that the i^{th} row of $\boldsymbol{H}^p$ is the output of the $LSTM$ over $\boldsymbol{P}$ at timestep i, and the same for $\boldsymbol{Q}$.

3.2 Match-LSTM Layer

As originally proposed by Wang and Jiang (2016), we sequentially apply match-LSTM to align the question and paragraph. At each timestep i in the context paragraph, we use a word-level attention mechanism to create an attention vector $\alpha_i \in \mathbb{R}^{1 \times Q}$. We then apply this attention vector to $\boldsymbol{H}^q$ and concatenate it with the corresponding timestep's vector in $\boldsymbol{H}^p$ ($\boldsymbol{h}_i^p$) to create an intermediate vector z_i. We then feed z_i into a standard LSTM and track its output, $\boldsymbol{h}_r^i$. Mathematically, this is:

$$\begin{aligned}
\boldsymbol{G}_i &= \tanh(\boldsymbol{H^q W}^q + (\boldsymbol{h}_i^p \boldsymbol{W}^p + \boldsymbol{h}_{i-1}^r \boldsymbol{W}^r + \boldsymbol{b}^p) \otimes \boldsymbol{e}_Q) \\
\alpha_i &= \text{softmax}(\boldsymbol{G}_i \boldsymbol{w} + b \otimes \boldsymbol{e}_Q) \\
\boldsymbol{z}_i &= \begin{bmatrix} \boldsymbol{h}_i^p & \alpha_i \boldsymbol{H}^q \end{bmatrix} \\
\boldsymbol{h}_i^r &= \overrightarrow{LSTM}(\boldsymbol{z}_i, \boldsymbol{h}_{i-1}^r)
\end{aligned}$$

Here, $\boldsymbol{h}_i^p$ is the i^{th} row of $\boldsymbol{H_p}$, $\boldsymbol{h}_{i-1}^r$ is the output of the LSTM described in the last equation (initialized to 0 for the first timestep), and $\boldsymbol{e}_Q$ means to repeat the vector on the left until we have a matrix or row vector of the same dimensionality of the other. This is implemented using the function `tf.tile()`.

The dimensionality of the model weights and intermediate layers are as follows:

- $\boldsymbol{W}^q, \boldsymbol{W}^p, \boldsymbol{W}^q \in \mathbb{R}^{h \times h}$
- $\boldsymbol{G}_i \in \mathbb{R}^{Q \times h}$
- $\boldsymbol{b}^p, \boldsymbol{h}_i^r \in \mathbb{R}^{1 \times h}$, $\boldsymbol{w} \in \mathbb{R}^{h \times 1}$
- $b \in \mathbb{R}$

We run this match-LSTM in both the forward and backward direction. $\overrightarrow{\boldsymbol{H}}^r$ are the forward LSTM outputs at each timestep, $\overleftarrow{\boldsymbol{H}}^r$ are the backward LSTM outputs at each timestep, and $\overrightarrow{\boldsymbol{H}}^r, \overleftarrow{\boldsymbol{H}}^r \in \mathbb{R}^{P \times h}$. That is:

$$\overrightarrow{\boldsymbol{H}}^r = \begin{bmatrix} \overrightarrow{\boldsymbol{h}}_1^r \\ \vdots \\ \overrightarrow{\boldsymbol{h}}_P^r \end{bmatrix} \text{ and } \overleftarrow{\boldsymbol{H}}^r = \begin{bmatrix} \overleftarrow{\boldsymbol{h}}_1^r \\ \vdots \\ \overleftarrow{\boldsymbol{h}}_P^r \end{bmatrix}$$

We concatenate these two matrices to create a joint encoding $\boldsymbol{H}^r \in \mathbb{R}^{P \times 2h}$:

$$\boldsymbol{H}^r = \begin{bmatrix} \overrightarrow{\boldsymbol{H}}^r & \overleftarrow{\boldsymbol{H}}^r \end{bmatrix}$$

3.3 Answer Pointer Layer

Rather than sequentially predicting or classifying the tokens, we directly model the probability of the answer having start index a_s and end index a_e:

$$p(a_s, a_e | \boldsymbol{H_r}) = p(a_s | \boldsymbol{H_r}) \times p(a_e | a_s, \boldsymbol{H_r})$$

We generate two probability distributions $\beta_1 = p(a_s|\boldsymbol{H_r})$ and $\beta_2 = p(a_e|\beta_1, \boldsymbol{H_r})$ as follows:

$$\boldsymbol{F}_k = \tanh(\boldsymbol{H^r V} + (\boldsymbol{h}^a_{k-1}\boldsymbol{W}^a + \boldsymbol{b}^a) \otimes \boldsymbol{e}_Q)$$
$$\beta_k = \text{softmax}(\boldsymbol{F}_k \boldsymbol{v} + c \otimes \boldsymbol{e}_Q)$$
$$\boldsymbol{h}^a_k = \overrightarrow{LSTM}(\beta_i \boldsymbol{H}_r, \boldsymbol{h}^a_{k-1})$$

Here, the weights and intermediate components have the following dimensionality:

- $\boldsymbol{V}, \boldsymbol{W}^a, \in \mathbb{R}^{2h \times h}$
- $\boldsymbol{F}_k \in \mathbb{R}^{P \times h}$
- $\boldsymbol{b}^a \in \mathbb{R}^{1 \times h}$, $\boldsymbol{v} \in \mathbb{R}^{h \times 1}$
- $\boldsymbol{h}^k \in \mathbb{R}^{1 \times 2h}$
- $c \in \mathbb{R}$

We apply the above recurrence once with an initial $\boldsymbol{h}^k = \overrightarrow{0}$ to generate β_1, then use β_1 as an attention vector over $\boldsymbol{H}_r$ and feed the product into an LSTM to generate $\boldsymbol{h}^a_0$. Using this conditional encoding, we apply the recurrence again to generate β_2.

Once we have the probability distributions, we predict a_s, a_e by searching for the highest joint probability:

$$(a_s, a_e) = \arg \max_{\substack{0 \leq i,j \leq P \\ i \leq j \leq i+20}} \beta_{1,i} \times \beta_{2,j}$$

We train our model by minimizing the sum of cross-entropy losses

$$\text{Loss} = CE(\beta_1, \overrightarrow{u}) + CE(\beta_2, \overrightarrow{v})$$

where $\overrightarrow{u}$ is a one-hot vector with a 1 at the actual start index and 0 elsewhere and $\overrightarrow{v}$ is a one-hot vector with a 1 at the actual end index and 0 at the others.

3.4 Model Settings

We initialize the model using GloVe word embeddings from Pennington, Socher, and Manning (`https://nlp.stanford.edu/projects/glove/`); our default model used 100-dimensional embeddings, but, as described in the next section, we also experimented with 300-dimensional embeddings. If a word embedding was not in the GloVe vocabulary, it was initialized as a zero vector. We did not update word embeddings during training.

Based on the dataset statistics described below, we capped question length at 20 and context paragraph length at 250. These numbers allowed us to use 98%+ of the dataset, while keeping padding to a minimum (we also used `tf.nn.dynamic_rnn` to help speed up rnn's). We ignored any samples with context paragraph length greater than 250 and used exponential masking to zero out the probability distributions (β_i's and α_i's).

We use different weights for both the forward and backward passes of the match-LSTM; we also use separate cells for the two preprocessing letters.

As in Wang and Jiang (2016), we use a hidden layer dimensionality $h = 150$. We used the TensorFlow AdamOptimizer with learning rate .001 and minibatches of size 32. Depending on word embedding size, our model had 1,800,000 - 2,100,000 parameters.

Our initial model did not use dropout or L2 regularization. In later models, we applied dropout to the non-recurrent connections in the model using a procedure similar to the one in Zaremba et. al (2015). We applied $p_{drop} = 0.3$ to model's inputs ($\boldsymbol{P}$ and $\boldsymbol{Q}$) before the preprocessing layer as well

	Count	Mean	Std	min	25%	50%	75%	85%	99%	max
Context	81386	137.525066	56.993778	22	101	126	164	190	324	766
Question	81386	11.31659	3.739408	1	9	11	13	15	23	60
Answer	81386	3.409653	3.830426	-1	1	2	4	5	21	46

Table 1: Descriptive statistics of the SQUAD dataset

as to $\boldsymbol{H}^r$ before the Answer Pointer Layer.

Our evaluation metrics were F1 and "Exact Match" (EM). To compute F1, we treat the predicted answer span for each question and the actual answer span as bags of tokens and compute their F1 (harmonic mean of precision and recall); we average this across all samples. EM is simply the percentage of triples on which the predicted span exactly matches one of the ground truth answers.

4 Experiments

We tested a number of different combinations of word embedding sizes/sources, hidden layer dimensions, search implementations, dropout configurations, etc. over the course of the project - however, due to time constraints we were unable to conduct these investigations with the rigor that we would have liked. That is to say that many of the variables were changed simultaneously in an effort to boost performance, making it difficult to assess the explanatory power of any single variable. Nonetheless we

4.1 Dataset

As mentioned above, we made use of the Stanford Question Answering Dataset (SQUAD) which contains 107,785 question, context paragraph, and answer span triplets sourced from Wikipedia. The methodology for collecting these data was to randomly sample 536 of the top English-language articles on the site, extract individual paragraphs, and then contract individuals on Amazon Mechanical Turk to devise question-answer pairs for the given paragraphs. Answers were further validated by having two other participants attempt to answer the questions - these answers were then turned into the three ground truth reference points. Once the dataset was finalized, it was split into train (80%), dev (10%), and test (10%) by the creators of the dataset [1]. For this project, the training set was further subdivided into a main training set and a validation set. In order to devise optimal ranges for question and context masking/padding we computed a range of descriptive statistics and histograms for the lengths of the question, context, and answers (Table 1 and figs. 1-3).

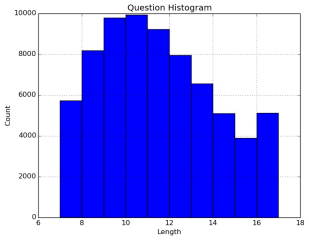

Figure 2: Question Lengths

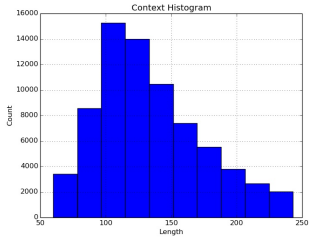

Figure 3: Context Lengths

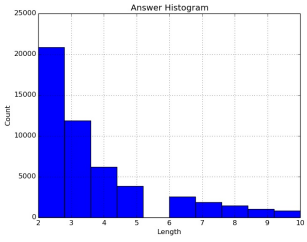

Figure 4: Answer Lengths

4.2 Results

Dropout and Embeddings
Our most rigorous experiment / model variation involved comparing two models with different dropout rates and word embedding size. More specifically, we observed that when we switched from

	Train	Train	Dev	Dev	Test	Test
	F1	EM	F1	EM	F1	EM
Human	-	-	-	-	90.5	80.3
Wang and Jiang Match-LSTM	-	-	77.2	67	66.9	77.1
(*) Our baseline Match-LSTM	77.6	64	40.5	27	-	-
Ours + dropout + grad. clip	51.9	39	41.7	28.4	-	-
(**) Ours + 300d vectors + dropout	60.3	46	43	34	45	35

Table 2: Summary of Model Results. Our experiment results compared with human-level performance and the best model from Wang and Jiang, who introduced the Match-LSTM. The 2nd and 3rd models used dropout on non-recurrent connections in the preprocessing and Answer Pointer layers, and the third used 300-dimensional GloVe embeddings instead of 100-dimensional ones.

word embeddings with length 100 (trained on the Wikipedia corpus) to embeddings with length 300 (trained on Gigaword) and simultaneously changed the dropout rate from 0 to 30%. After 10 epochs, the baseline achieved an F1 of 40.5 and an EM of 27 compared to 43 and 34 for the improved model respectively. Moreover, in later trials comparing dropout percentages of 15% and 30%, we observed that learning improvement (as measured by validation F1 and EM scores) continued into later epochs (9 and 10) for the latter model compared to the former model - which would often plateau at epoch 6 or 7.

Length
We examined whether the length of questions and contexts had any predictive effect on the F1 and EM scores received. Beginning with contexts, we observed an almost completely monotonically decreasing relationship between the context length and F1/EM scores. F1 decreased from 49.24 at a context length of 40 to 39.28 at a length of 160 and 25.5 at a length of 340. A similar trend could be observed between question length and performance, with F1 accuracy diminishing from 45.35 at a question length of 5-10 tokens vs. 35.5 at a length of 20+ tokens. These results make intuitive sense from a probabilistic and modeling perspective, since a shorter context length obviously has a smaller search space.

Reading Levels
Another interesting experiment we undertook was investigating the potential impact of context word complexity (understood in terms of human reading level) on the accuracy of our model. Specifically, we used the textstat Python library's "Reading Consensus" measure to measure an estimated reading comprehension grade level for each context paragraph. This measure is driven by a number of factors including syllable count, the Dale-Chall Readability Score, and the SMOG index among others (all described in detail at https://pypi.python.org/pypi/textstat). Fascinatingly, we did not observe any correlation between reading level and model performance. For levels 8 through 12, which encapsulated over 80% of the dataset, each level hovered around between an F1 score of 43-45.

4.3 Analysis of Selected Errors

In order to understand why our model performed poorly, we decided to look at the questions we missed and tried to derive reasons why.

Let us consider the dev context that contains a list of many Harvard grads as follows "...conductor Leonard Bernstein; cellist Yo Yo Ma; pianist and composer Charlie Albright;..." This is a simple structure, showing connection between occupation and name. Questions were of the form "What is the name of world renowned cellist is a former Harvard student?" Of the four questions asked of this form, all were answered with random names. From these random answers, we realize that though the model has learned to pick out the proper part of speech in the context, but cannot pick out the correct name based on the occupation in the question. The fact that our model cannot use a relationship between two words in the question to find that same relationship in the answer shows that our match layer is broken. When creating the attention layer, we do not properly encode these

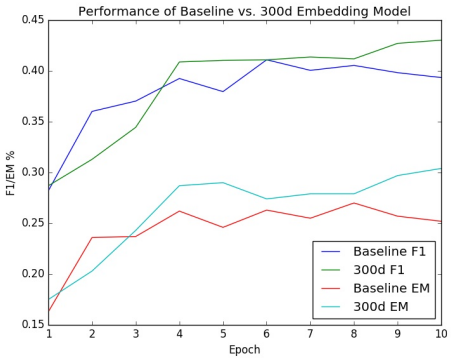

Figure 5: Comparison of Baseline and Improved Models

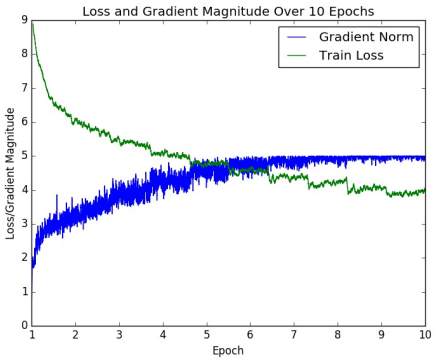

Figure 6: Train Loss and Gradient Magnitude over Time

relationships. This in part accounts for our decrease in performance in comparison with the original paper's implementation.

Upon further investigation, we found that our project often missed dependencies in high complexity sentences. Consider the following sentence: "The Super Bowl 50 halftime show was headlined by the British rock group Coldplay with special guest performers Beyonce and Bruno Mars, who headlined the Super Bowl XLVII and Super Bowl XLVIII halftime shows, respectively." When answering questions about guest performers, super bowl numbers, and other auxiliary information, the model spits out Coldplay as the answer. We believe that the lack of encoded dependencies makes it impossible for our model to understand sentences like this where there are many levels of word dependencies. Thus, the simple issue described in the above paragraph is intensified in questions like these. This could be made worse by the fact that we did not train word embeddings, and thus did not encode extra dependency information into our words and help counteract this effect. With more time on this project, we would debug exactly why the attention layer does not encode said information, and would test if training word embeddings would increase our performance.

5 Conclusion

In this paper, we develop a model for question answering on the SQuAD dataset based on match-LSTM and Answer-Pointer first proposed by Wang and Jiang (2016). Our experiments show proof of concept and a basic functional model, though not approaching state-of-the-art or Wang and Jiang's best single boundary model.

In the future, we plan to build on this promising start by doing a more exhaustive hyperparameter search to fully tune the model (we did not yet experience a "U-shape" in F1 and EM from overfitting) as well as ensembling this with a sequence based model. We also plan to explore parameter sharing between the forward and backward layers of our match-LSTM as well as different word embedding strategies: different embedding sizes, training all embeddings, and initializing out-of-vocabulary words to random vectors that we then train. Exploring these strategies could prove particularly useful since many OOV words are nouns and often the answer to the question. Finally, we will look into more advanced attention mechanisms to align the question and paragraph encodings (and more advanced encoding mechanisms, e.g. bi-LSTM preprocessing), since this is one of our model's weaknesses.

6 Acknowledgements

Thanks to Professors Manning and Socher and the entire CS224N teaching team for an engaging, difficult, and ultimately very fun quarter. Special thanks to everyone who helped us in office hours and/or answered our late night Piazza questions throughout the quarter!

7 Contributions

Alex Haigh: I worked out a lot of the math from the match-LSTM paper and converted the column vector notation to row vector notation. Along with Cameron and the rest of the team, I wrote the initial implementation of most of `qa_model.py`, and debugged it along with the other team members. I also wrote our data loading scripts in `util.py` Most of my time was spent debugging the model (including time alone), running downstream experiments, and making the poster. The writeup was split fairly evenly between all team members.

Cameron Van De Graaf: I wrote scripts to gather introductory statistics and plot histograms on the dataset. I worked out much of the math from the match-LSTM paper in greater depth in advance of implementation and coded/debugged a significant portion of `qa_model.py` in conjunction with Alex Haigh. I also conceived and ran several experiments, collecting the results and plotting all visualizations.

Jake Rachleff: With the rest of the team, I worked a lot on working out the model. Specifically, I worked heavily on the prediction layer. I was the most experienced software engineer on the project, so I worked on a lot of debugging tensorflow, creating utility methods like padding data, all of `qa_answer.py`, setting up technologies like codalab, etc. Like everyone else, I spent significant time working on `qa_model.py`.

8 References

[1] Rajpurkar et. al. SQuAD: 100,000+ Questions for Machine Comprehension of Text. *arXiv:1606.05250v3*, 2016.

[2] Wang and Jiang. Machine Comprehension Using Match-LSTM and Answer Pointer. *arXiv:1608.07905v2*, 2016.

[3] Zuremba, Sutskever, and Vinyals. Recurrent Neural Network Regularization. *arXiv:1409.2329v5*, 2015.

[4] Xiong, Zhong, and Socher. Dynamic Coattention Networks For Question Answering. *arXiv:1611.01604v3*, 2017.

[5] Wang, Bansal, Gimpel, and McAllister. Machine Comprehension with Syntax, Frames, and Semantics. *http://ttic.uchicago.edu/ dmcallester/ACL2015.pdf*, 2013.

[6] Hochreiter and Schmidhuber. Long Short-Term Memory. *http://ieeexplore.ieee.org/document/650093/*, 1997.

[7] Schuster and Paliwal. Bidirectional Recurrent Neural Networks. *http://dl.acm.org/citation.cfm?id=2205129*, 1997.

[8] Luong, Pham, and Manning. Effective Approaches to Attention-based Neural Machine Translation *arxiv:1508.04025*, 2015.

[9] Vinalys, Fortunato, Jaitly. Pointer Networks. *arxiv: 1506.03134*, 2015.

[10] Srivastava, Hinton, Krizhevsky, Sutskever, Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting *https://www.cs.toronto.edu/ hinton/absps/JMLRdropout.pdf*, 2014.