Dynamic Coattention Networks with Encoder Maxout

Thaminda Edirisooriya
`tediris@stanford.edu`

Morgan Tenney
`mtenney@stanford.edu`

Hansohl Kim
`hansohl@stanford.edu (codalab:hansohl)`

Abstract

We implement a system based on the Dynamic Coattention Network outlined in Xiong et al. to address the Stanford Question Answering Dataset (SQuAD), learning to find continuous answer spans for prompted questions contained in a context document. We further experiment with variations on the original architecture, including the use of BiLSTMs and the addition of a maxout layer in the coattention encoder. We find that our attempts to simultaneously incorporate BiLSTMs and avoid a significant increase in the number of parameters result in reduced performance. We also find that the encoder maxout layer provides a small but consistent performance boost. Our final model incorporates the encoder maxout layer and obtains a test set F1 score of 69.22% and EM of 58.764%.

1 Introduction

The goal of our project was to achieve performance on the Stanford Question Answering Dataset (SQuAD) (Rajpurkar et al.). Specifically, we seek to accurately predict an answer to a question that refers to a specific context paragraph, under the constraint that the answer is fully contained within the context paragraph, and is continuous. Performance is evaluated with the standard tokenwise F1 score and EM (exact match) percentage over the predicted answers.

The dataset consists of 100,000+ question answer pairs derived from 500+ Wikipedia articles. The articles are sampled randomly from the 10,000 articles ordered by page rank, and question answer pairs were created using the Amazon mechanical turk service. The dataset is ideal for the reading comprehension task both due to its quality and its size - unlike datasets like bAbI, the data is human-generated, and the number of examples is very large for the quality of the data.

2 Related Work

The SQuAD dataset problem has been approached using a variety of techniques. For instance, the Match-LSTM model in conjunction with Pointer Net was shown to be performant on the dataset (Wang et al.), as well as Bi-directional Attention Flow models (Seo et al.). One of the common approaches among all models is to first encode a question-document importance vector for each example, and then make use of the encoding in some intelligent way to output a prediction.

We based our implementation on the Dynamic Coattention Network (DCN), which has several unique innovations but shares the same overall pattern as above: it encodes a relationship between the question and context document and proceeds to predict start and end tokens for the answer span.

Finally, the DCN above, as well our main addition to the model, draw heavily upon Maxout Networks, a network architecture that allows a model to learn multiple variants of the weights it will use to try to encode a relationship between input and desired output.

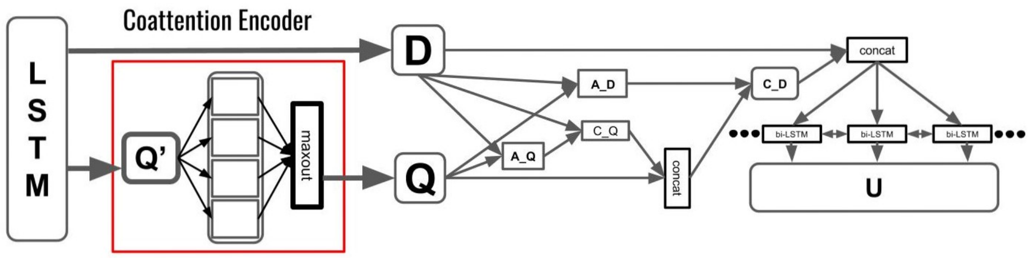

Figure 1: Architecture of Coattention Encoder - the highlighted red portion is our contribution and is not a feature of the baseline DCN

3 Approach

3.1 Dynamic Coattention Networks

We implemented a DCN based on the architecture presented in ”Dynamic Coattention Networks for Question Answering” and experimented with variations on the original architecture.

3.1.1 DCN Overview

The DCN is comprised of two components - the Coattention Encoder and the Dynamic Pointer Decoder - that we will briefly describe before expanding on the changes we made. The DCN first encodes the question and document by feeding the input into a single LSTM. To allow for variation between the question and document encoding space, the LSTM output for the question is processed by an additional non-linear projection layer. We denote the question and document encodings as Q and D, respectively. These encodings are multiplied to acquire the affinity matrix L between the question and the document.

We then softmax normalize L row-wise and column-wise to produce attention weights A^Q and A^D respectively. We use the attention weights to acquire the attention contexts DA^Q, denoted C^Q, and QA^D, which we combine into a later operation for increased performance. We define the coattention context as $C^D = [Q; C^Q]A^D$. The notation $[a; b]$ signifies a concatenation of a and b. The coattention context is given as input to a final BiLSTM, the output of which forms our coattention encoding U.

The task of the Dynamic Decoder is to determine the start and end indices of the answer in the input document, given the coattention encoding U as input. It takes an iterative approach, alternating between acquiring an updated guess for the start and end indices using two separate Highway Maxout Networks with identical architecture described below.

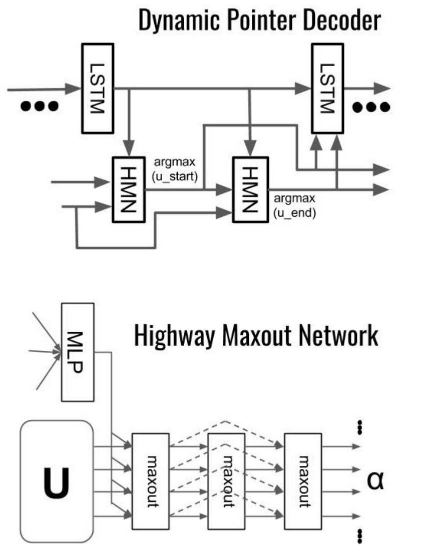

Figure 2: Dynamic Pointer Decoder and Highway Maxout Network Architecture

$$HMN(u_t, h_i, u_{s_{i-1}}, u_{e_{i-1}}) = max(W^{(3)}[m_t^{(1)}; m_t^{(2)}] + b^{(3)})$$

$$r = tanh(W^{(D)}[h_i; u_{s_{i-1}}; u_{e_{i-1}}])$$

$$m_t^{(1)} = max(W^{(1)}[u_t; r] + b^{(1)})$$

$$m_t^{(2)} = max(W^{(2)}m_t^{(1)} + b^{(2)})$$

At each stage, an overlying LSTM gives its current hidden state and the two previous guesses $u_{s_{i-1}}$ and $u_{e_{i-1}}$ for the start and end index as input to one of the HMNs. After several maxout layers, we run $argmax$ on the output of the HMN to find the new most likely candidate for the start or end index, and use that as input for the next HMN. Once we reach the iteration limit, we consider the start and end indices at that time to be our result.

One of the more complicated portions of this network in terms of implementation was the pair of Highway Maxout Networks. While Tensorflow has a built-in maxout network, we built ours from scratch in order to handle batching. The weights for each layer of the maxout network when unbatched were 3-dimensional in the math that the algorithm was based off of, but in practice this made multiplication difficult. Rather than further complicate things by imposing a fourth dimension for batching we expanded in two dimensions to accommodate the additional information. This enabled our matrix multiplications, but required reshaping at every layer of the network.

3.1.2 Dropout

The original DCN paper mentions the use of dropout throughout the network. However, it does give details as to the implementation of the dropout, or which layers have dropout applied. We experimented with adding dropout to many parts of the model, but found the best performance by putting dropout on all inputs to LSTM units (Zaremba et al.), and inputs to the Maxout layers specifically. In Maxout Networks (Goodfellow et al., 2013), the authors discuss the power of applying dropout and maxout together to allow a network to learn complex approximations of the functions we desire to learn, and we found that applying dropout to our model helped greatly. It enabled us to avoid overfitting and we retained a low validation loss throughout our optimization.

3.1.3 Encoder Maxout

Our main change to the architecture is the application of a maxout layer to the encoding of the question matrix Q. In the original architecture, Q was calculated as follows:

$$Q = \tanh(W^{(Q)}Q' + b^{(Q)})$$

Here, Q' is the encoding of the question tokens retrieved after running the word vectors for each token through a Recurrent Neural Network with LSTM units. In our case, we add a dimension p to our weight matrix $W^{(Q)}$ and bias $b^{(Q)}$, and perform a max along this dimension after computing the multiplication and addition result. This can be expressed as:

$$Q = \max_{PoolDim}(\tanh(W_P^{(Q)}Q' + b_P^{(Q)}))$$

Where $W_P^{(Q)}$ has the same shape as $W^{(Q)}$ but with an added (leftmost) dimension of size p and the same is true of $b_P^{(Q)}$. We still use the $\tanh$ calculation, but by performing a max over this added dimension, we ideally provide our model the capacity to learn multiple projections and take the most relevant information from each. The motivation for this is to have greater flexibility to address the different types of questions in the SQuAD data differently to obtain a more expressive coattention matrix and knowledge representation.

3.2 Loss and Optimizer

We implemented our loss as the sum of the two sublosses for the predicted start position and end position of the answer span. We note that the structure of the SQuAD dataset, where an answer is a single span within the context document, allows us to indirectly predict answers in this manner. Our final prediction for each position is given as the final output of each HMN, a vector of weights over each word in the context document corresponding to the likelihood of each word being the actual position of the span. A predication is made by an argmax over these vectors. As in the original paper, we use a softmax cross entropy loss on our output weights for each subloss, where the labels are a one-hot encoding of the actual positions for the start and end of an answer span. The loss is averaged over each batch.

We note that this loss, while relatively simple and useful, does not directly take into account how far off the actual predicted span limits are. It makes no difference whether the predicted span limit is near or far from the actual position, as the cross entropy loss simply considers the weight placed on the actual span limit's position.

For our optimizer we used the existing Tensorflow implementation of Adam, an optimizer with adaptable learning rates and momentum (Kingma et al.). The Adam optimizer generally requires very little hyperparameter tuning, as typically the only adjusted hyperparameter is the initial learning rate.

3.3 GLoVe vectors

We developed our system using pretrained GLoVe word vectors (Pennington et al.) as the initial encoding for question and document tokens. We built and tested our model using 100-dimensional vectors from a 6 billion token crawl, and switched to the 300-dimensional vectors from a 840 billion token crawl once we had our system running smoothly. We saw a noticeable performance increase after doing so, since the system had more information with which to understand the tokens it was receiving as input.

3.4 Implementation Details and Hyperparameters

As in the original paper, the size of all of our RNN hidden layers was 200, and our maxout pool size was 16. We used a learning rate of 0.001, and experimented with dropout probabilities of 0.15 and 0.25. We processed data in batches of 64 examples, and our dynamic pointer decoder ran for 4 iterations.

Our initialization schemes for weights and biases in our network was consistent throughout. Weights used Xavier initialization while biases were initialized with zeros, with the exception of LSTM forget gates. These were initialized to 1 to encourage LSTM cells to retain state initially, as recommended by Jozefowicz et al. Hidden states for LSTMs were initialized to zeros and the encoding sentinel vectors were initialized randomly, as per the original paper. The start and end index predictions in the decoder were initialized as the beginning and end of the context documents respectively.

In our training we used batches that contained questions and documents of varying lengths. To allow Tensorflow to easily handle batch operations, we padded the questions and context documents to the maximum sequence length in the batch. To allow for this, we implemented masking throughout our model with the actual sequence lengths.

We note that as a special case, the masking value applied to out of bounds tokens was $-1 \times e^{30}$ before feeding into a softmax layer (exponential masking). This is to prevent the softmax from artificially inflating the probabilities of tokens we know cannot be part of the answer.

The training was done on an Nvidia GTX 1070 for up to 10 epochs and each epoch with batch size 64 took roughly 25 minutes. Validation F1 and EM scores were computed on random samples of size 100 every 100 iterations. The validation loss over the full validation set was computed every 400 iterations. Each epoch was 1271 iterations.

4 Experiments and Results

4.1 Learning Rate

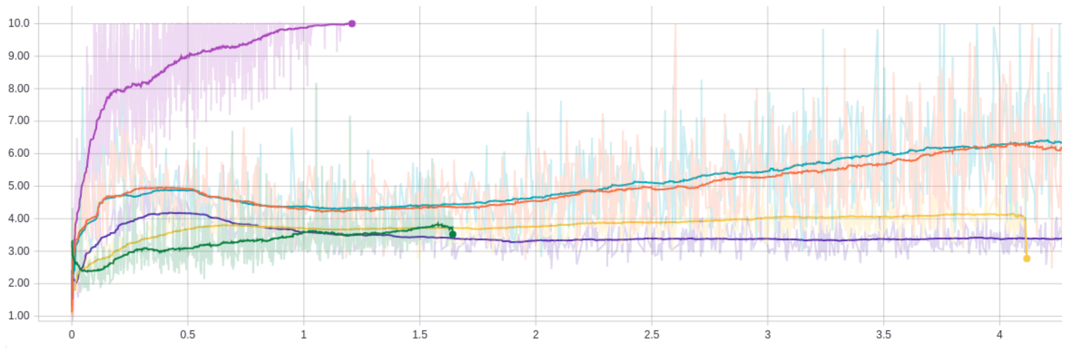

Figure 3: Gradient norm over training hours. Light purple had an initial learning rate more than two orders of magnitudes from 0.001. Teal and orange had no dropout and were overfitting. Other models were remarkably consistent across a wide range of changes.

Adam performed well with little hyperparameter tuning. The one optimizer hyperparameter we experimented with was the initial learning rate, and even this proved to be stable within roughly an order of magnitude. Through experimentation we found initial learning rates around 0.001 provided reliable training. Gradient norms were very consistent as well, remaining mostly within the 2-7 range, even across different variations on our model. From this extremely consistent gradient range we decided on 10 as the maximum for our gradient clipping window.

4.2 Overfitting and Dropout

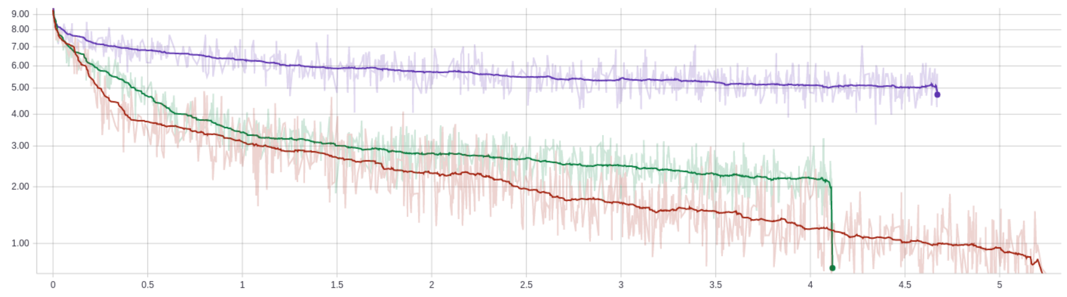

Figure 4: Smoothed training loss over hours. Brown has no dropout and significant overfitting (min val loss 3.6). Purple has extreme dropout and exhibits underfitting. Green is our final dropout setup (min val loss 2.7).

The original paper mentions the inclusion of usage of dropout and we found it to be vital to the performance of our model. Initial implementations without dropout exhibited strong overfitting on the training set, approaching a training loss of 0.5 over 10 epochs while the validation loss rebound from a low of 3.6 back to 6.1. As mentioned previously, we experimented with several methods of applying dropout. In one case we selectively applied dropout to only a few parts of the encoder or decoder, which was not enough to resolve the overfitting. On the other end of the spectrum we experimented with extreme dropout including on gates within the LSTM. This drastically cut the expressive power of our model and it instead exhibited underfitting. Our experimentation and the existing literature on LSTMs and maxout networks (Zaremba et al.), (Goodfellow et al.) led us to our final setup where we apply dropout on the inputs of the LSTM cells and maxout layers, with the exception of previous hidden state inputs (i.e. no horizontal dropout).

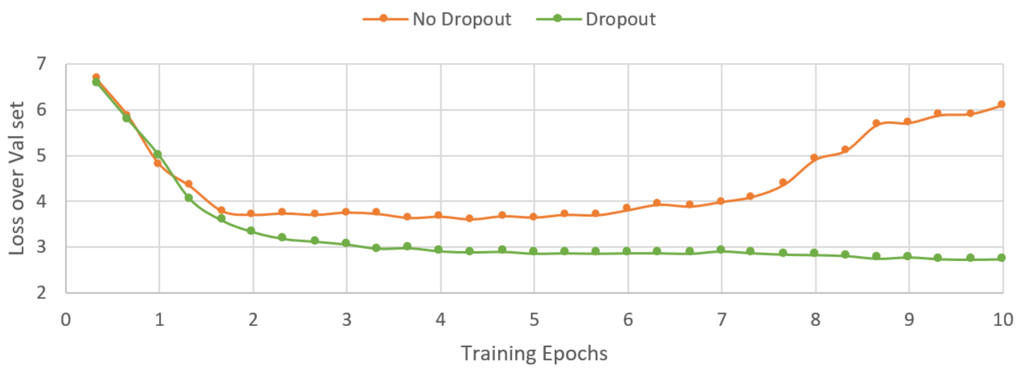

Figure 5: Val loss over epochs, corresponds to Figure 4. Val loss was evaluated over the entire val set every 400 iterations (3 times per epoch). Here the overfitting is very obvious after two epochs.

4.3 Decoder HMN Links

The Dynamic Pointer Decoder uses two HMNs to propose a predicted start and end for the answer span at each iteration. In the original paper, the start HMN passes its new prediction immediately to the end HMN in the same iteration. Our initial implementation, however, experimented with taking the start and end predictions at each iteration and passing them simultaneously to the HMNs in the next iteration. This created a disconnect in communication between the start HMN and the end HMN. We observed that switching back to the original implementation reduced our end loss and we obtained qualitatively better end predictions.

4.4 Bidirectional LSTMs

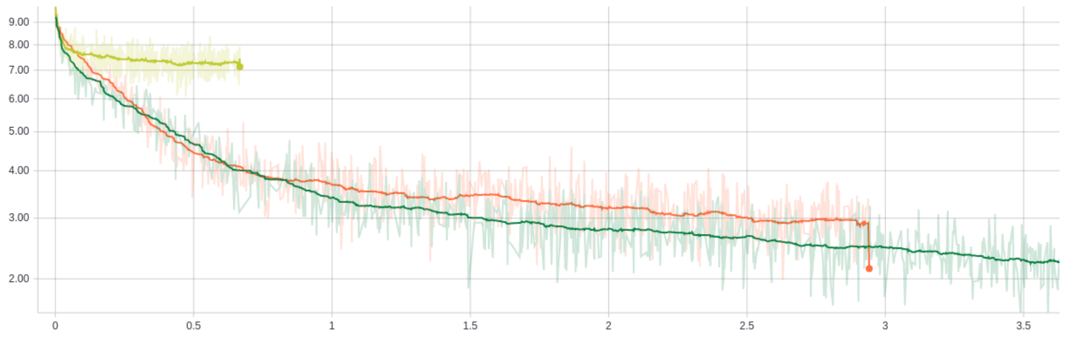

Figure 6: Smoothed training loss over hours. Green is our final model for reference. Orange uses a BiLSTM with size 100 word vectors and performs slightly worse in validation and dev. Yellow uses a BiLSTM with size 300 word vectors and performs very poorly.

We experimented with using bidirectional LSTMs in place of the single-direction LSTMs in the model. Due to the setup of the DCN architecture, the switch to BiLSTMs introduced hidden dimension issues. BiLSTMs return twice the output with the same hidden size which must be combined somehow. Concatenating them vastly increases the parameters in the model, since this doubling is carried throughout the entire model, which we aimed to avoid. Naive combinations like addition or averaging risk loss or degradation of information. We therefore tried halving the hidden size of these BiLSTMs and concatenating.This functioned reasonably while we were still working with size 100 word vectors (since the hidden size was still 100 even after being halved). Qualitatively, this seemed to reduce our predictions of overly long answers. Many of our incorrect answers have the start token correct but choose a very late end token, which may be an artifact of the single direction LSTM reading the paragraph. Furthermore, this was far more common than the inverse problem.

An example of this phenomenon can be seen with a question from the sanity-check dev subset: Consider the question "Which player had the most interceptions for the season?" with a context document about football.

The correct response is given as "Kurt Coleman", which the BiLSTM identified. The original LSTM-based implementation, however, responded with "Kurt Coleman, who led the team with a career high seven interceptions, while also racking up 88 tackles and Pro Bowl cornerback Josh Norman." Similar situations were common, where an answer had the correct start but the following words were still relevant to the entity of interest and the end token was placed far later than it should have been, typically upon seeing something that clearly indicated a new entity (in this case a different subject).

With the BiLSTM there did seem to be a more even balance. However, we did not see a significant overall performance increase at the time, and the same method performed poorly with size 300 word vectors, likely due to the significant loss of information when moving from 300 to 100 dimensions.

Due to the size constraints we could not concatenate original size BiLSTM hidden states and the naive alternatives we explored were suboptimal so we ultimately did not use this feature in our final model.

4.5 Encoder Question Maxout

Through our experimentation we developed our final models described in section 3. Our result was two sets of final models corresponding to whether we included our additional encoder maxout layer or not. To test the effect of this added layer, we ensured each model in one set had a counterpart in the other with all other details held consistent. We then submitted these models to the dev leaderboard and observed the dev F1 and EM scores.

We found that adding encoder maxout resulted in a small but consistent improvement to our F1 score, with a smaller change to the exact match score of our model. Typically the improvement was 1-3% for F1 and up to 1% for EM. We note that an examination of the qualitative evidence from our (limited) dev subset suggested that the encoder maxout models performed better on many examples that we were previously completely failing (guessing massive spans of the paragraph, for example, or a span that did not overlap the correct one), while examples that were closer to correct experienced much less improvement. This is also supported by the relative changes in our F1 compared to our EM. EM only improved slightly while F1 received a larger boost, possibly from improvement in our worst performers that would not affect EM.

4.6 Final Models and Leaderboard Results

Model	Val Loss	Dev (F1)	Dev (EM)
100D W-Vec, No Q Maxout, 15% Dropout	3.14	64.54%	51.34%
300D W-Vec, No Q Maxout, 15% Dropout	2.91	67.35%	57.27%
300D W-Vec, No Q Maxout, 25% Dropout	2.89	67.93%	57.66%
300D W-Vec, Q Maxout, 15% Dropout	2.73	69.08%	58.25%
300D W-Vec, Q Maxout, 25% Dropout	2.79	69.18%	57.25%

Based on our validation losses and dev leaderboard results, we chose to submit our size 300 word vector, 15% dropout model with encoder maxout for our final test submission. The results on the test set were an F1 of 69.22% and an EM of 58.764%, consistent with both our val and dev results.

Qualitatively, our final model predictions on the dev subset exhibit behavior similar to that noted in the original paper. We observe that many questions are either reasonable and very close to correct or completely off. Examples of the former are "Miami's Sun Life Stadium" instead of the correct "Sun Life Stadium", "Coldplay with special guest performers Beyoncé and Bruno Mars" instead of the correct "Coldplay", and "Santa Clara, California" instead of the correct "Santa Clara". Even when an answer is incorrect it can often be the right type of entity, such as asking for a person and predicting a different but relevant person.

However, when an answer is more than slightly incorrect, it is usually extremely incorrect. Examples of this include situations like predicting the answer:

> Sanders was his top receiver with six receptions for 83 yards. Anderson was the game's leading rusher with 90 yards and a touchdown, along with four receptions for 10 yards. Miller had six total tackles (five solo), $2\frac{1}{2}$ sacks, and two forced fumbles. Ware had five total tackles and two sacks. Ward had seven total tackles, a fumble recovery, and an interception. McManus made all four of his field goals, making him perfect on all 11 attempts during the post-season. Newton completed 18 of 41 passes for 265 yards, with one interception. He was also the team's leading rusher with 45 yards on six carries. Brown caught four passes for 80 yards, while Ginn had four receptions for 74. Ealy was the top defensive performer for Carolina with four total tackles, three sacks, a forced fumble, a fumble recovery, and an interception. Defensive End Charles Johnson

when the correct answer was simply "Sanders." Even here, the start token at least is correct.

5 Conclusion

Dynamic Coattention Networks are a very complicated but powerful architecture for QA tasks. We were able to successfully reimplement one and obtain reasonable performance with behavior similar to the original model. Furthermore, we observed throughout our experimentation that the DCN architecture framework consists of individual pieces that are modular to the point that they could be combined with different systems in future experiments. Although we ultimately were unable to incorporate bidirectional LSTMs, there may be a better way to successfully integrate them without massively increasing the size of the model. Our BiLSTM experiments with size 100 word vectors hinted that this may help with asymmetry in start-end prediction performance and address many of the observed cases of a correct start token with absurd end token predictions. The maxout encoder, meanwhile, consistently provided a slight boost to performance, and our final model benefited from its inclusion. Moving forward, the use of multiple different coattention encodings and a way to take maxout or ensemble feedback to produce the final encoded knowledge representation may be worth exploring. For the decoder, many key properties come from the iterative interaction between the HMN and LSTM. The properties of this dynamic pointer decoder when using other existing decoders as the index proposer in place of the Highway Maxout Network represents a possible topic for future exploration.

References

[1] Xiong, C. & Zhong, V. & Socher, R. (2017) Dynamic Coattention Networks for Question Answering

[2] Rajpurkar, P. & Zhang, J. & Lopyrev, K. & Liang, P. (2016) SQuAD: 100,000+ Questions for Machine Comprehension of Text

[3] Pennington, J. & Socher, R. & Manning, C.D. (2014) GloVe: Global Vectors for Word Representation. *Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532-1543.

[4] Wang, S. & Jiang, J. (2016) Machine Comprehension Using Match-LSTM and Answer Pointer

[5] Seo, M. & Kembhavi, A. & Farhadi, A. & Hajishirzi, H. (2017) Bi-Directional Attention Flow for Machine Comprehension

[6] Goodfellow, I. & Warde-Farley, D. & Mirza, M. & Courville, A. & Bengio, Y (2013) Maxout Networks. *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, pp. 1319-1327.

[7] Zaremba, W. & Sutskever, I. & Vinyals, O. (2015) Recurrent Neural Network Regularization

[8] Jozefowicz, R. & Zaremba, W. & Sutskever, I. (2015) An Empirical Exploration of Recurrent Network Architectures

[9] Kingma, D. & Ba, J. (2015) Adam: A Method for Stochastic Optimization

Contributions

Thaminda Edirisooriya: Custom LSTM and decoder implementation

Morgan Tenney: Custom HMN and decoder implementation

Hansohl Kim: Encoder and encoder variants implementation

Other contributions were shared