Codalab Username: aaronlwy

Implementation of Multi-Perspective Context Matching for Machine Comprehension

Aaron Loh
Department of Computer Science
Stanford University
Stanford, CA 94305
aaronlwy@stanford.edu

Abstract

This is an implementation of the Multi-Perspective Context Matching for Machine Comprehension Model as established by Zhiguo Wang, Haitao Mi, Wael Hamza and Radu Florian of IBM, with a few changes. This model consists of 6 layers, the Word Representation Layer, for which I use Glove Vectors, the filter layer, context representation layer, multi-perspective context matching layer, aggregation layer, and prediction layer. This model aims to predict the span of words in a context paragraph that can answer a particular question. To train and evaluate my implementation of the model, I used the SQuAD dataset.

1 Introduction

1.1 Task Definition

The problem definition is as follows. Given a passage P with N words ($p_1 \ldots p_N$), and a question Q with M words ($q_1 \ldots q_M$), predict a span in the passage P [$p_{start} \ldots p_{end}$] that will answer the question Q. In this model, instead of predicting a span, we will instead predict the boundary of the span, that is, the starting and ending indices. We will denote the index of p_{start} as a_s and the index of p_{end} as a_e.

We can model this as 2 separate classification problems with N classes. The first classification will be for the start word, and the second classification will be the end word. That is, we will have 2 probability distributions over N classes, with the first probability distribution telling us which class (or context paragraph position) the start word will be in, and the second telling us which class the end word will be in.

In mathematical terms, we choose

$$a_s = argmax\ P(a_s|q,p)$$

And

$$a_e = argmax\ P(a_e|q,p)$$

We further enforce the constraint that $a_s < a_e$. One way of doing this is to simply choose a_s first, and then only consider the indexes after a_s when selecting a_e. Another way is to maximize over all pairs of a_s and a_e where the condition holds. Assuming independence in the predictions of the start and the end probabilities, we can write this as

$$[a_s, a_e] = argmax_{a_s,a_e}\ P(a_s|q,p)P(a_e|q,p)$$

where $a_s < a_e$.

2 Background/Related Work

Other than this model implemented by Wang et al., multiple teams have attempted this problem. A few of the other papers I looked at include the Dynamic Coattention Networks model proposed by Xiong et al. of Salesforce Research, which iterated over potential answer spans, and attaining a F1 score of 80.4% in an ensemble [2]. Their model performed well as it was able to recover from initial local maxima.

Another relevant paper was submitted by Shuohang Wang and Jing Jiang of Singapore Management University, where they attempted the task using Match-LSTM and answer pointer [3]. Match LSTM would use an attention mechanism to create a weighted representation of the question, which they then combined with the vector representation of the paragraph representation at a particular time step, to make the prediction. Their model was able to get a F1 Score of 71.2%.

3 Approach

3.1 Preprocessing

I implemented the model from scratch, first loading and pruning the training and validation data sets for consumption by the model. I first gathered summary statistics about the training and validation data sets, and found that very few of them had answers past the 200th token. In order to increase the predictive power of my model, I thus removed the few examples that had answer spans past that point, and only predicted answers within the first 200 positions. This allowed the model to have greater predictive power over a shorter span of paragraph, at the risk of overfitting to the training data.

I further implemented padding and masking so that the length of inputs for the questions and the contexts would be the same across the board.

3.2 Building a Baseline

I first implemented a simple baseline in order to ensure the architecture was sound. The baseline encoded the question and passage word embeddings using a bidirectional LSTM, and then used a feed forward mechanism to make predictions. The equations for the predictions were as follows:

$$a_s = h_{context} W_{startContext} + h_{Question} W_{startQuestion}$$

$$a_e = h_{context} W_{endContext} + h_{Question} W_{endQuestion}$$

Here, $h_{context}$ refers to the last hidden state of the context after passing it through a BiLSTM, while $h_{Question}$ refers to that of the question. This simple model was able to get me up to 5% F1 on the validation set, after just training on a small sample of training examples.

3.3 Building the Multi Perspective Context Matching Model

I implemented all parts of the original model, with slight changes to the Multi Perspective Context Matching Layer. In total, there are 6 layers, the word representation layer, given by Glove Word Embeddings, the filter layer, the context representation layer, the multi perspective context matching layer, the aggregation layer and the prediction layer. Here is the schematic from the original paper:

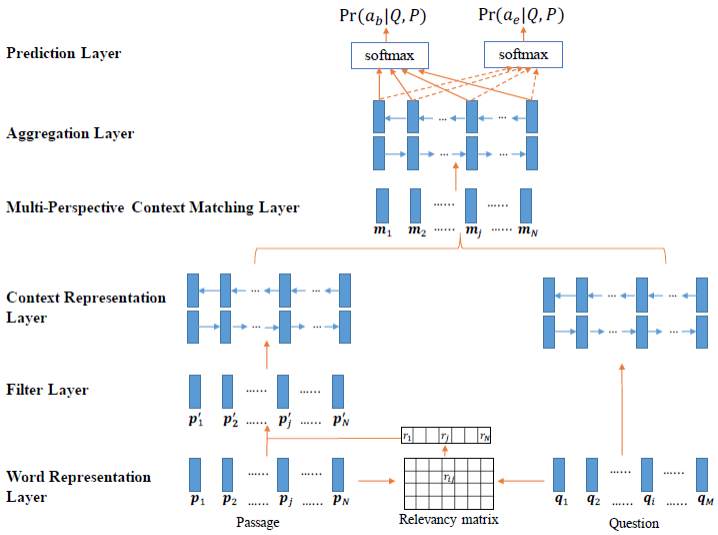


Figure 1. Architecture for Multi Perspective Context Matching Model. [1]

3.3.1 Word Representation Layer

The word representation I used for the passage and question embeddings came from the 100 dimensional GloVe word vectors provided.

3.3.2 Filter Layer

Intuitively, this layer serves to enhance the parts of the passage that are more relevant to the passage, with relevancy calculated by the cosine distance between vectors in the passage and vectors in the question. The relevancy matrix for a single example can be calculated using the following equations:

$$R = QP^T$$

In this case, Q and P represent the normalized word embeddings. Q is of dimension Max Question Length x Embedding Size, while P is transposed so that it is of Embedding Size, Max Context Length. Multiplying these two matrices will get is relevancy matrix for that example. To get the relevancy factor, we simply take the max along the Question Length dimension, which will then let us scale the context vectors appropriately through a simple element wise multiplication.

3.3.3 Context Representation Layer

This layer takes the filtered passage representation, referred to as P' from the previous layer as well as the question representation Q, and runs them through a Bidirectional LSTM. We use the final hidden states of both the forward and backward direction as the representation for the passage and the question.

For each position i in the question, we calculate the forward direction hidden state:s

$$\overrightarrow{h_i} = \overrightarrow{LSTM}(h_{i-1}^p, q_i)$$

As well as the backward direction hidden state:

$$\overleftarrow{h_i} = \overleftarrow{LSTM}(h_{i+1}^p, q_i)$$

We use the concatenated hidden states $[\overrightarrow{h_i}; \overleftarrow{h_i}]$ as the representation for the question.

Similarly, for each position j in the context, we calculate the forward direction hidden states

$$\overrightarrow{h_j} = \overrightarrow{LSTM}(h_{j-1}^p, p_j)$$

As well as the backward direction hidden state:

$$\overleftarrow{h_j} = \overleftarrow{LSTM}(h_{j+1}^p, p_j)$$

We use the concatenated hidden states $[\overrightarrow{h_j}; \overleftarrow{h_j}]$ as the representation for the context.

3.3.4 Multi-Perspective Context Matching Layer

In the original paper, they used this layer to compare each representation of the context from the previous layer to each representation of the question.

We first define the function

$$\boldsymbol{m} = f_m(v_1, v_2, W)$$

This function produces a l dimensional vector, where l corresponds to the number of perspectives. In our case, v_1 and v_2 are vectors of size $2H$, where H is the size of our hidden state. v_1 corresponds to a passage vector at a particular timestep, while v_2 corresponds to a question vector at a particular timestep. Our matrix is given by $W \in \mathbb{R}^{l \times 2H}$. With these parameters, we can calculate each index k of the $\boldsymbol{m}$ using the following formula:

$$m_k = cosine(W_k \circ \boldsymbol{v_1}, W_{\boldsymbol{k}} \circ v_2)$$

Where $cosine$ is calculates the cosine distance between the two vectors. W_k marks the kth layer of the matrix W.

I implemented the three matching schemes in the paper, full matching, max pooling matching, and mean pooling matching.

3.3.4.1 Full Matching

In full matching, we compare the representation of the context at each time step from the context representation layer with the final hidden state of the question. In the original model, they broke this up into two portions. For each index j of the context representation, from 1…N, they calculated

$$\overrightarrow{m_j}^{full} = f_m(\overrightarrow{h_j^p}, \overrightarrow{h_M^q}; W^1)$$

$$\overleftarrow{m_j}^{full} = f_m(\overleftarrow{h_j^p}, \overleftarrow{h_1^q}; W^2)$$

Instead of splitting this up into two parts, I decided to use the concatenated hidden vectors from both the backwards and forwards direction together, to capture more of the interactions between the two directions. That is, I calculated

$$m_j^{full} = f_m\left([\overrightarrow{h_j^p}; \overleftarrow{h_j^p}], [\overrightarrow{h_M^q}; \overleftarrow{h_1^q}]; W^3\right)$$

3.3.4.1 Max Pooling Matching

This follows a similar idea as the previous matching, but instead of just comparing the context representation at each index j from 1 to N with the last hidden state of the question representation, we now compare it with the question representation at each timestep, from 1 to M. For a particular context vector at position j, we then set the value of the lth dimension to be the maximum value taken across the dimension for the question representation. Once again, in the original model, this was represented as

$$\overrightarrow{m_j}^{max} = \max_{i \in (1 \ldots M)} f_m(\overrightarrow{h_j^p}, \overrightarrow{h_i^q}; W^4)$$

$$\overleftarrow{m_j}^{max} = \max_{i \in (1 \ldots M)} f_m(\overleftarrow{h_j^p}, \overleftarrow{h_i^q}; W^5)$$

For my model, I used this to calculate the max pooling vector:

$$m_j^{max} = \max_{i \in (1 \ldots M)} f_m([\vec{h}_j^p; \overleftarrow{h}_i^p], [\vec{h}_M^q; \overleftarrow{h}_i^q]; W^6)$$

This may cause the model to lose some degrees of freedom, as now the same index i has to be selected so that it maximizes across the backwards and forwards representation of the question.

3.3.4.1 Mean Pooling Matching

This is almost the same as the max pooling layer, but instead of taking the maximum across the question dimension, we take the mean. For my model, I used this to calculate the mean pooling vector:

$$m_j^{mean} = \frac{1}{M} \sum_{i=1}^{M} f_m([\vec{h}_j^p; \overleftarrow{h}_i^p], [\vec{h}_M^q; \overleftarrow{h}_i^q]; W^9)$$

My final mixed representation of the context and the question, that is, the output of this layer, is simply the concatenated vectors

$$[m_j^{full}; m_j^{max}; m_j^{mean}]$$

3.3.5 Aggregation Layer

This layer passes the vectors from the previous layer through another BiLSTM.

3.3.5 Prediction Layer

I use a simple feed forward network to generate the predictions. That is, for the aggregated vectors V, each of size Max Length Context by $2H$, I would do the following:

$$A_{start} = VW_{startPredictions}$$

$$A_{end} = VW_{endPredictions}$$

I would then pass it through a softmax to calculate the probabilities for each word position being the starting and the ending, and then apply the cross-entropy loss to train the model.

4 Experiments

4.1 Data

I used data from the SQuAD dataset. Unfortunately, due to time limitations and low GPU utilization, I was only able to train the model for 1 epoch. Furthermore, I was unable to train it on all the examples at once due to the GPU running out of memory. I thus applied random selection over the examples so as to prevent overfitting to a small dataset, saving the weights at each iteration. Given more time, and better efficiency I would run the model on the full dataset across all the epochs. The validation dataset was similarly extracted from the SQuAD dataset.

4.2 Hyperparameters

Here are the hyperparameters that I used for training this model:

Table 1: Hyperparameters

Hyperparameter	
H (State Size)	50
Learning Rate	0.1
GloVe Word	100

Embeddings	
Maximum Gradient Norm	10
Optimizer	Gradient Descent Optimizer
L (number of perspectives)	50

Unfortunately, these were mostly chosen due to hardware constraints, notably, memory size of the GPU, rather than actual performance reasons. A hyperparameter search would be useful for increasing the performance of the model, given more time.

4.3 Evaluation

For evaluation, at each epoch, I calculated the loss, F1, EM scores for both the training and validation set. The training set was used as a sanity check to see that the loss was indeed going down and the F1 and EM scores were increasing. The validation set was used as a metric to see if the model could generalize to unseen data. Lastly, I tested the model on the dev set, once again, using the F1 and EM scores as a metric.

4.4 Results

These are the F1 and EM Scores after 1.5 Epochs, evaluated on the Validation, Development, and Test Sets. Due to time constraints and low utilization of the GPU, I was only able to train for 1.5 Epochs. At the point of writing, the train F1 was 21.1% and increasing, evidence that the model was still learning.

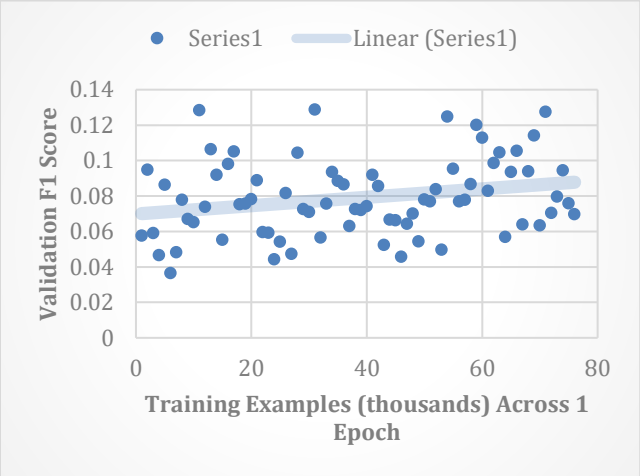
Table 2: F1/EM Results after 1.5 Epochs

	F1 Score	EM Score
Validation	19.2%	9%
Development	13.42%	5.421%
Test	14.003%	5.518%

4.4 Analysis

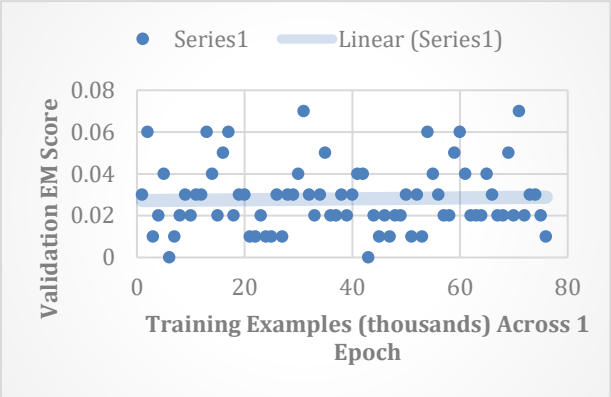
The first thing to note is that while the model did not converge in 1.5 Epochs, as was to be expected, there was a general upward trend in the performance across the epoch, as I trained on more and more examples. At every 1000 examples or so, I evaluated on the training set as well as a random sample of 100 examples from the validations set.

Fig 2: F1 Across 1 Epoch



Codalab Username: aaronlwy

Fig 3: EM Across 1 Epoch



We see that the model does improve with more examples, with the improvement in F1 being more marked as compared to the improvement in the EM score. This was accompanied by a decrease in the training and validation cost as well. In order to speed up the learning, in future iterations, I would experiment with larger learning rates, especially in the first epoch, and perhaps use an adaptive gradient descent algorithm with annealing across the epochs to help the model converge.

I also found that the model was greatly affected by the length of the context, as well as the answer. In general, there was a downward trend for both F1 and EM as the context length increased. Interestingly enough, there was a spike in performance for context lengths around 270. Possible reasons may include having seen more examples in that range, although it could also be random. It is also interesting that the model does not perform as well in the opposite extreme, when the context length is very short.

Fig 4: F1/EM for different context lengths in validation set

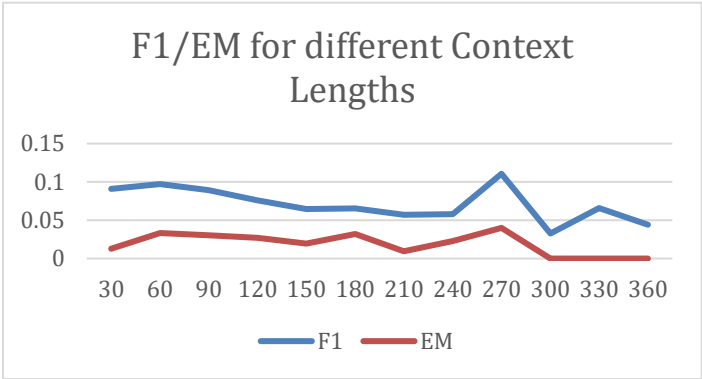
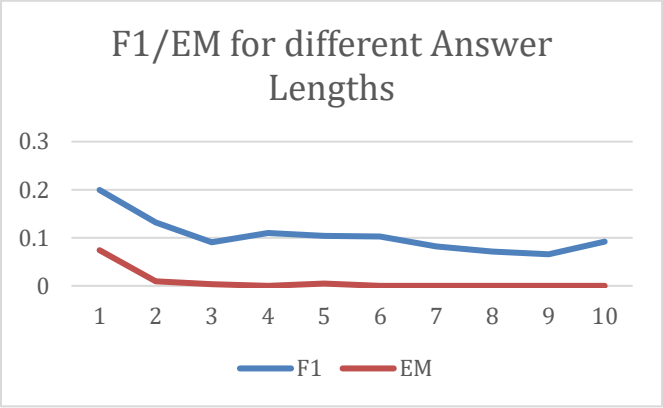
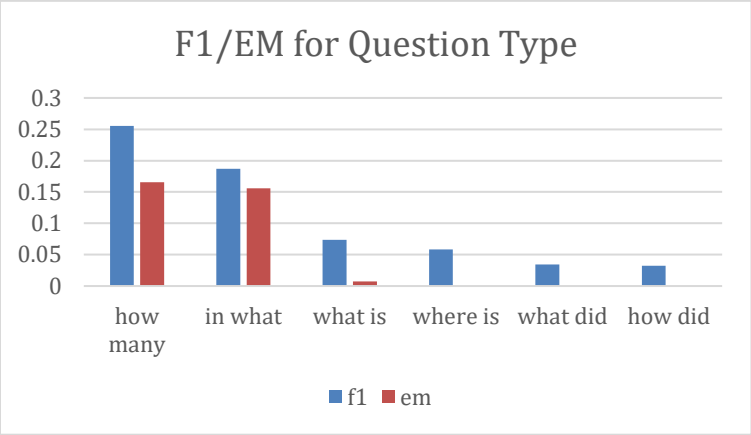


Fig 5: F1/EM for different answer lengths in validation set



The model also performs better for short answer spans, with the EM score decreasing sharply when the span is more than 3.

Fig 6: F1/EM for different question types



For the reasons above, the model performs much better for questions that have a simple concise answer, such as questions starting with "how many", as opposed to open ended questions like "how did". This is consistent with the findings of the original paper, where the "how did" questions received the lowest F1 and EM scores.

In appendix A, we see an example of an exact match where the answer is to predict when a certain event happened, the correct answer being "1913". We see that in this case the model is able to identify the right answer, with probabilities near the answer being close to 0, while there is a spike at the correct position. The model is able to handle such cases with an answer span of 1 well.

5 Conclusion

As far as implementation goes, I have learned about how to set up a neural network model from scratch, from loading and parsing the data, to setting up the TensorFlow graph and its operations, setting up the encoder and decoder, to training the model and evaluating it. I have also learned how important it is to use the hardware fully. Due to the long training time, training on the CPU was an intractable task, and not being able to fully utilize the GPU was a big roadblock for me.

The first step to improving the performance of my model is definitely to try to get the GPU to have a higher utilization so that I can train the model on more data. Running the model on the full dataset over 10 epochs would likely give it a performance boost. I would also increase some of the parameters such as the hidden state size as well as the number of perspectives, as the increase parameter space would allow the model to adapt to more situations. I would also like to use tools like dropout, learning rate annealing, as well as adaptive optimizers, instead of just a gradient descent optimizer.

Beyond that, I would look into different models other than the boundary model, to see if that would increase performance. For instance, one possible method of prediction would be to classify each passage position as either part of the answer or not part of the answer. Then, an identification of the answer span could be done by finding a subsequence that maximizes the number of predicted answer words while minimizing the number of non-answer words.

Acknowledgments

I would like to thank Professor Christopher Manning and Professor Richard Socher, as well as the TAs of CS 224N for their support throughout this project.

Codalab Username: aaronlwy

Appendix A

Fig 7: Start Probability Example For Exact Match

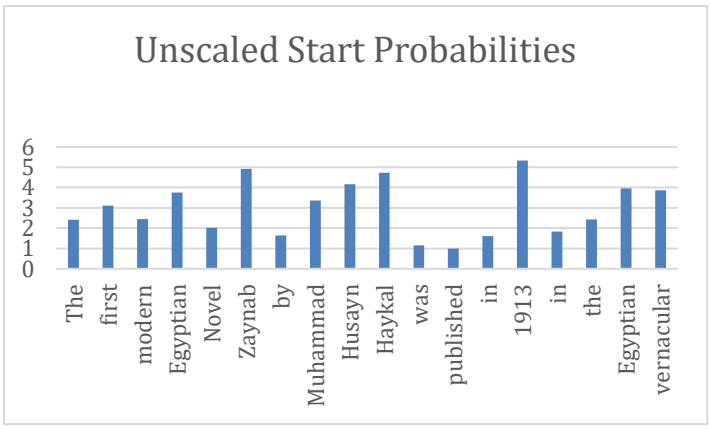
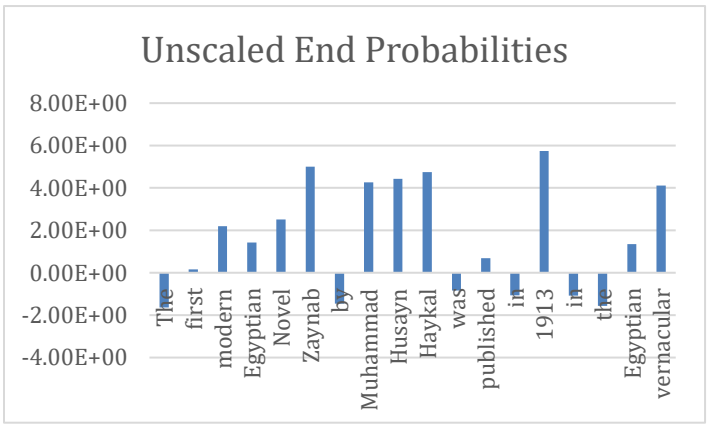


Fig 8: End Probability Example For Exact Match



References

[1] Zhiguo Wang, Haitao Mi, Wael Hamza, Radu Florian (2016) Multi-Perspective Context Matching for Machine Comprehension. arXiv:1612.04211

[2] Caiming Xiong, Victor Zhong, Richard Socher (2017) Dynamic Coattention Networks for Question Answering. arXiv:1611.01604

[3] Shuohang Wang, Jing Jiang (2016) Machine Comprehension using Match-LSTM and Answer Pointer arXiv:1608.07905