Implementing Multi-Perspective Context Matching for the SQuAD Task in TensorFlow

Christopher Pesto
Codalab: cpesto
cpesto@stanford.edu

Abstract

The Multi-Perspective Context Matching model introduced by Wang, et al. [1] in 2016 is known to be capable of producing strong results in the SQuAD question answering task. As of this writing, it is tied for 3rd place on the SQuAD leaderboard.[1] Implementing the model efficiently is difficult in practice, and the original introduction paper leaves out some implementation details. The goal of this paper is to give practical details regarding my own implementation of this model and its performance, to accompany a code submission that is known to produce reasonably good scores.

1 The SQuAD task

This is an introduction to SQuAD. Please see the original paper for more information on the dataset, task, and evaluation.

1.1 Dataset

The SQuAD question answering task was introduced by Rajpurkar, et al. [2] in 2016 to provide a challenging, large, high quality dataset for evaluating Reading Comprehension (RC). Those authors found existing datasets to be either too small or unreflective of genuine human reading comprehension, and intended for SQuAD to be both large and qualitatively representative of real human understanding.

The dataset consists of 100,000+ questions with one or more ground truth answers per question. Each ground truth correct answer is a span of text taken verbatim from the corresponding reading passage the question is associated with.

1.2 Task

The task is to take pairs of context reading passages and questions, and return the span of text from the context that answers the question.

1.3 Evaluation

Success on the task is evaluated in two ways.

F1: This is the average overlap between predictions and correct answers. The F1 is calculated for each prediction/correct answer pair for the answers under each question, the maximum is taken per question, then the average is computed over the per-question max values.

[1] SQuAD homepage with leaderboard: https://rajpurkar.github.io/SQuAD-explorer/

Exact Match (EM): This is the percentage of predictions that match any one of the correct answers for its question exactly.

2 Introduction of the model

2.1 Model layers

The Multi-Perspective Context Matching Model consists of 6 layers. This is a general description of each layer - for a more precise description of each (particularly the MPCM layer), please consult the original paper.

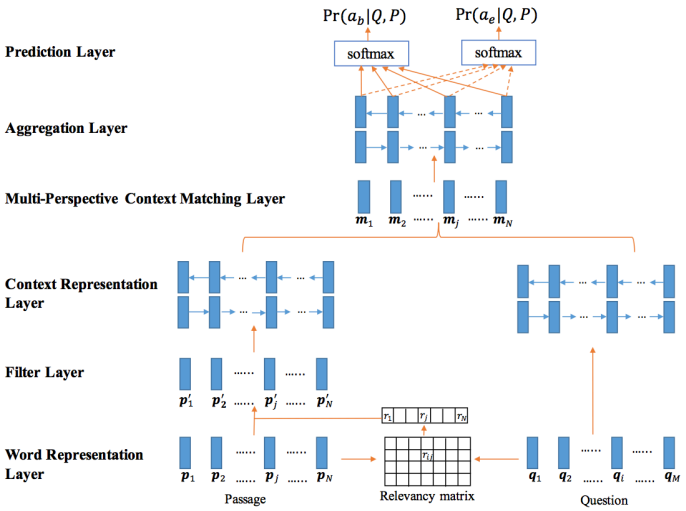

Figure 1: Figure 1 from the paper by Wang, et al.

Let C denote the length of the input sequence of context tokens and Q denote the length of the input sequence of question tokens.

Word Representation Layer: The context and question inputs are tokenized and the sequences of tokens are converted into sequences of word embedding vectors, lengths Q and C.

Filter Layer: For each context time step, the cosine similarity is computed between it and all Q question time steps. Each context time step vector is scaled by the maximum cosine similarity between it and all question time steps, producing a new filtered context sequence of length C.

Context Representation Layer: Pass both the filtered (scaled) context and question through the same BiLSTM. Output a sequence of hidden states for each, resulting in two new C- and Q-length sequences.

Multi-Perspective Context Matching (MPCM) Layer: Compare the sequences of hidden states for the context and question produced in the previous layer, to produce a single new C-length sequence. This is described in more detail below.

Aggregation Layer: Pass the output of the MPCM layer through a BiLSTM. Output another C-length sequence of hidden states.

Prediction Layer: Pass the output sequence from the previous layer through two different feedforward neural networks, to produce two C-length probability distributions for the start

and end positions of the answer span within the context.

2.2 MPCM layer detail

This layer accepts two sequences of hidden states, lengths C and Q, the results of applying a BiLSTM to the filtered context and question. Using three different matching strategies, each applied to both the forward and backward results from the BiLSTM, it outputs 6 vectors which are concatenated together at each time step.

This layer is complex. I found the original description very confusing, and I misunderstood how it worked at first. I describe it a little differently here than in the original paper, hopefully in a somewhat simpler way.

For each of the 6 matching strategy vectors $\mathbf{m}^i$, the kth element is defined as the cosine similarity between the kth rows of two vectors $\mathbf{v_1}$ and $\mathbf{v_2}$ multiplied element-wise down the rows of the ith matching matrix $\mathbf{W}^i$.

$$m_k^i = cosine(W_k^i \circ v_1, W_k^i \circ v_2)$$

Each matching matrix is lxd, where l is a tunable hyperparameter of the model that the authors call the "number of perspectives" and d is the number of hidden states in the BiLSTM providing the layer input.

The vectors $\mathbf{v_1}$ and $\mathbf{v_2}$ at each output time step depend on the matcher. Let $\mathbf{c}$ denote the context input sequence and $\mathbf{q}$ denote the question input sequence. Each matcher is applied to both the forward and backward inputs. Each is defined for the jth timestep of the context input, $\mathbf{c_j}$.

Full-Matching: This is just the above operation applied to $\mathbf{v_1} = \mathbf{c_j}$ and $\mathbf{v_2} = \mathbf{q_{final}}$.

Maxpool-Matching: The above operation is applied to $\mathbf{v_1} = \mathbf{c_j}$ and $\mathbf{v_2} = \mathbf{q_h}$, $h \in (1 \ldots Q)$. Each element m_k is the max for row k over the whole question.

Meanpooling-Matching: This is the same as Maxpooling-Matching, but using the mean instead of the max.

The result is a single C-length sequence, where each element is $6l$-dimensional.

3 Dataset analysis

For efficient training and prediction generation, it's necessary to process data in uniform-length batches. This means establishing maximum cutoff lengths for contexts and questions, which determine the largest possible batch size for a given GPU memory (larger batch sizes give faster execution) as well as execution speed for a given batch size. Cutting off more context and question data will obviously yield lower accuracy, though. Effectively making this tradeoff between execution speed and model accuracy requires understanding the data distribution.

The SQuAD dataset is divided into two disjoint files, train-v1.1.json and dev-v1.1.json. The preprocessing code we were provided splits train-v1.1.json into a 95% training set and a 5% validation set. The contents of dev-v1.1.json act as a publicly-available test set. Its contents are different in that each question can contain multiple correct answers, while those in train-v1.1.json contain only one correct answer per question.

All three datasets follow a similar long-tail distribution, where the bulk of their lengths is concentrated at the lower end of their ranges. The statistics and figures below are for my 95% training set.

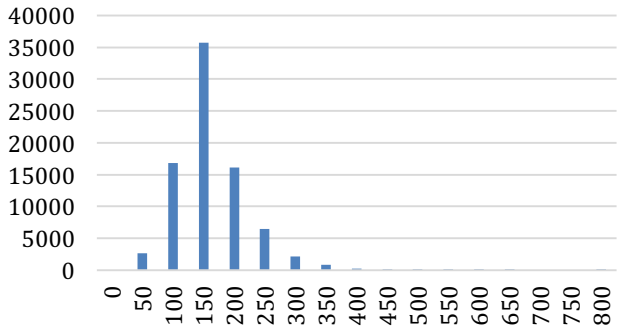

Figure 2: Histogram of tokenized training set context lengths

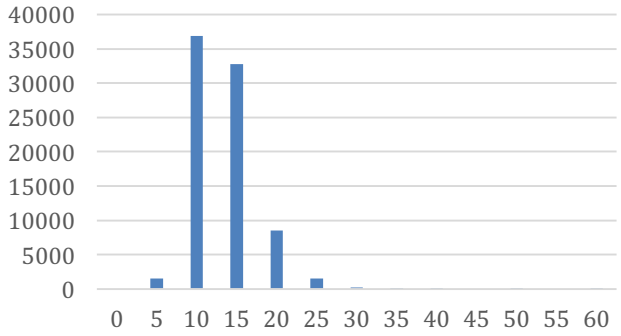

Figure 3: Histogram of tokenized training set question lengths

	95th percentile	99th percentile	Max
Context length	244	324	766
Question length	18	23	60
Correct answer span, max(start, end)	152	217	605

Table 1: Statistics on tokenized training set

4 My implementation

4.1 Simplifications

I made two simplifications to the MPCM model in my implementation.

No character embeddings: The original paper combines character embeddings (the result of feeding each character in a word through an LSTM) with the word embeddings. I cut this to get to a working implementation on time, and was unable to add it in later in the time I had. According to the MPCM authors' ablation studies, this only accounts for a 2.5 percentage point F1 loss relative to their full single model.

Less dropout: The original paper claims dropout is used in all layers from Figure 1. In my implementation I only apply dropout in the Context Representation Layer (outputs), MPCM layer (outputs), Aggregation Layer (outputs), and Prediction Layer (middle layer of two-layer neural networks). I omitted dropout on the Word Representation Layer and Filter Layers because I wanted all token embedding information to always reach the network.

4.2 Missing details

There are several details on which the original paper is ambiguous that need to be filled in, or which are missing but seem necessary.

Feedforward neural network: The paper describes the prediction layer as just two feedforward neural networks. I chose to implement a two-layer network, defined as

$$h = dropout(ReLU(xW_1 + b_1))$$
$$probability = h^T W_2 + b_2$$

$\mathbf{W_2}$ is actually a weight vector, rather than a weight matrix, because this layer must reduce the rank of the input tensor to produce the probabilities, a detail not described in the original paper.

End point greater than or equal to start point: The original paper states that they enforce that the end point is greater than or equal to the start point. Since the two prediction networks are independent at the last layer, they can produce the endpoints in either order. I enforce this constraint by choosing a start point with *argmax* over the start probabilities, then setting the end probabilities before that point to *–inf* before selecting an end point. I experimenting with altering an existing model to apply this symmetrically, reselecting the start or end point first depending on which had a higher probability when *end* < *start*, but the gain from doing that was negligible.

Sequence masking: The final step of my decoder is to set the probabilities of indices beyond the original context lengths within each batch to *–inf*. The original paper doesn't mention this, but it is necessary to get an accurate softmax cross entropy loss that does not include probability predictions on padding tokens.

5 Implementation pitfalls

Vectorization: This model defines operations (outside of the RNNs) that operate over the length of the context and question. Effectively vectorizing these is extremely difficult – in the MPCM layer, this means operating on pairs of rank 4 tensors. It is necessary though, as implementing these with even just a loop over context length makes TensorFlow unusably inefficient.

Embedding quality: This model is extremely sensitive to the quality of the word embeddings used. Moving from the 100-dimensional 6B token uncased GloVe embeddings to the 300-dimensional 840B token cased GloVe embeddings was essential for getting good results.

6 Results

6.1 Dev set tokens

We were provided with code that produced a trimmed vocabulary and corresponding set of embeddings. That is, it applied tokenization to the train and validation sets and created a vocabulary from the full set of resulting tokens, the PAD padding token, the UNK out of vocabulary token, and one other special token. It created an embedding array equal in length to this vocabulary, initialized randomly. For each token/vector pair in the full embedding, it then copied in the true embedding if the token was present in this vocabulary. (It originally matched against multiple case variants in the vocabulary, but I added an option to use only case sensitive matching in order to use the 840B GloVe embeddings, which are cased.)

I retained this trimming strategy, but in addition to the training and validation set tokens, I included the tokens from the tokenized dev set as well. My results below are based on models trained with this vocabulary and trimmed embedding.

I justified this decision under the assumption that this was equivalent to training and predicting with the full embedding set, which I only didn't do out of memory concerns. No matter what, there aren't embeddings available to the model that aren't in the full set, so adding those that were necessary for the dev set seemed more reflective of what the model is actually capable of.

As outlined below, though, this assumption was possibly flawed.

6.2 Submitted results

For all my training, I followed the original paper recommendations of $l = 50$ and 100-dimensional LSTM hidden states.

For my submitted results, I used dropout of 0.2 (also recommended in the paper), and cut off context lengths at 375 (99^{th} percentile of dev context lengths) and question lengths at 34 (max of dev question lengths). This yielded the following.

	F1	EM
Dev set	67.189	55.951
Test set leaderboard	58.496	46.176

Table 2: Results on dev set and test set leaderboard for model

6.3 Dev/test set discrepancy

There is a significant gap (8.693 F1, 9.775 EM) for this trained model.

It seems that the missing token embeddings that are trimmed from the full embedding set have a significant effect.

Additionally, I suspect that not including character embeddings hurts the model's ability to perform well on the test set as well, possibly more than the original paper's ablation studies would suggest. Even if all the missing OOV tokens from the test set are mapped to the opaque UNK vector in the word embeddings, character embeddings would give the model visibility into the token strings themselves, letting it effectively match them between question and context.

6.4 Sensitivity to cutoff length

Within the long tail, the model is actually not extremely sensitive to changes in cutoff length. In comparison to the above, the results for another model trained with a context cutoff of 248 (95^{th} percentile of dev context lengths) and question cutoff of 23 (99^{th} percentile of dev question lengths) are not substantially different.

	F1	EM
Q_{max} 34, C_{max} 375	67.189	55.951
Q_{max} 23, C_{max} 248	66.862	55.430

Table 3: Results on dev set for two different cutoff lengths

At these shorter cutoffs, the model could be trained on an Azure NV6 instance with an 8GB GPU with a batch size of 50, yielding about 45 minutes per epoch. The larger cutoffs require over an hour per epoch.

6.5 Sensitivity to dropout

I tried testing how the model trained with different dropout values, which the paper did not cover.

The "Train" results in this figure are from a random sample of 2000 rows from the training set at the end of the 4th epoch.

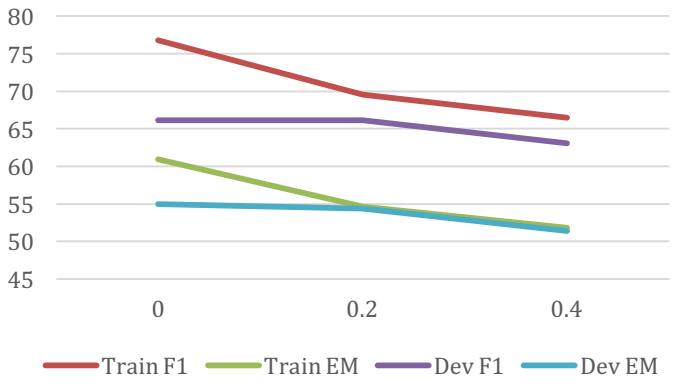

Figure 4: F1 and EM on train and dev sets after 4 epochs of training

Clearly the model overfits quickly with 0 dropout.

6.6 Errors

In general, as can be seen in the dev-predictions.json file for the Q_{max} 34, C_{max} 375 model included with the code submission, the model seems to perform well. Its errors make sense. For example:

Which NFL team represented the AFC at Super Bowl 50?
Prediction: Denver, Correct: Denver Broncos

What color was used to emphasize the 50th anniversary of the Super Bowl?
Prediction: golden, Correct: gold

The model struggles with these very similar answer forms. In the second case, given the context paragraph it was effectively impossible to know the actual correct answer was *gold* without knowing that is the name of the color.

6.7 Use of full embedding set

After noting the large discrepancy between the dev set and test leaderboard scores, I modified the vocabulary generation code to be able to produce an untrimmed vocabulary. In this case, I print exactly all the tokens in the GloVe 840B embeddings in order as the vocabulary, rather than developing it from any of the datasets.

This increases the vocabulary length from 123,292 tokens to 2,196,020, and produces an enormous embeddings file of 2.9GB in NumPy's *npz* format. TensorFlow unfortunately will not even load a tensor over 2GB normally. I was able to load it using a workaround technique I found in a GitHub thread with a Google Brain engineer, although unfortunately (in my implementation at

least) this requires my code to know the length of the embeddings file, which I pass in as a new command-line argument (this is in the version of the code I submitted). To my surprise, though, the model trained quickly on the Azure GPU – I was able to train with Q_{max} 23, C_{max} 248 and a batch size of 30, in not much more than an hour per epoch (up from ~45 minutes per epoch with the trimmed vocabulary for the same cutoffs).

Unfortunately, the results after 6 epochs of training were not what I expected. (The train and validation results are according to a final random sample of 2000 rows from each set at the end of the 6th epoch.)

	F1	EM
Training set	63.567	43.667
Validation set	50.643	33.000
Dev set	18.205	6.566

Table 4: Results using the entire GloVe 840B embeddings as vocabulary

The aberrant dev set results seem like they could be a bug or other error on my part, as there should be no significant difference between the validation and dev sets.

That said, these results illustrate that there is potentially a meaningful difference between including the dev set tokens in the trimmed vocabulary and simply using the entire embedding set as the vocabulary.

In the first case, when a token from the sets included in the trim operation does not appear in the full embeddings, it still receives a unique token id and vector (even though that vector is just randomly-generated). This means that at prediction time, all the tokens received are at least distinguishable.

In the second case, any token that does not appear in the full embeddings will just be collapsed into the single UNK token id/embedding at training and prediction time, making them indistinguishable from one another.

Again, character embeddings would probably address some of this difficulty, as the model would be able to use them to still distinguish between tokens in this case. Additionally, if this is really the main issue, it would probably be possible to use a trick such as introducing multiple UNK tokens to alleviate it somewhat.

7 Future exploration

If using the full embedding set is necessary for achieving generalizable high-quality results, the model is unpractical for use outside of a high-powered GPU-based server. It would be interesting to produce an effective model able to run on less powerful hardware.

Acknowledgments

Special thanks to both instructors and all the teaching assistants for leading a very challenging but also extremely rewarding and fun course.

References

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

[2] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ Questions for Machine Comprehension of Text. *arXiv preprint arXiv:1606.05250.*