Exploring speed and memory trade-offs for achieving optimum performance on SQuAD dataset

Renat Aksitov
raksitov@stanford.edu

Abstract

In this project I am building deep learning system for reading comprehension in Stanford Question Answering Dataset. I am constructing the architecture of my system by exploring some high performing models for SQuaD, and carefully choosing which of their details I could adapt, which ones I might need to change and which ones to drop altogether. My choices are primarily dictated by the computing resources constraints. Overall, I was able to train single model that achieved **76.37** F1 and **66.00** EM on dev dataset and ensemble of 12 models that achieved **80.044** F1 and **71.971** EM on test dataset.

1 Introduction

The focus of this project is applying deep learning techniques to the question answering for reading comprehension. This is a challenging task for an algorithm, as it requires both understanding of natural language and knowledge about the world. Deep learning system might acquire required understanding and knowledge purely from the annotated data, but it will need high quality data and a lot of it.

Recently, after introduction of the SQuAD data set [1], which is arguably the first data set for question answering that satisfies these requirements, a lot of progress has been made in a very short time in applying deep learning to a reading comprehension problem. Several original high performing models were introduced in a fast succession by researchers around the world with a culmination in February this year when Microsoft Research submitted [6] first model that exceeded human performance on one of the SQuAD metrics.

2 Background

To make a very difficult problem more approachable, SQuAD dataset defines some constraints on questions and answers that are considered. The problem definition is as follows. Given the text (also called context or passage) and the question (also called query) about the text, we need to choose a span in the text, that will be answering the question. For evaluating correctness of the answer, we will be using human labels. Evaluation data has 3 labels from different people per example, and sometimes it is 3 different spans. Selecting either one of them will be considered correct, which makes the task a bit easier. SQuAD defines two evaluation metrics: the exact match (EM) and more forgiving partial match (for which F1-score is used).

One method, that had a lot of success on SQuAD data set, came from Neural Machine Translation and is called attention. The idea is to make different parts of a model to *attend* to each other. For example, we might want the model to be aware of a query, when encoding a context or vice versa. One high performing model that adds both context2query and query2context attention is called BiDAF [2] and it has achieved state of the art results on SQuAD when it was first introduced.

3 Approach

I am starting with the provided baseline and then build upon it with various improvements, many of which I am borrowing from the BiDAF model [2]. In this section I will be describing improvements themselves, and I will outline their impact on the final model performance in the next section.

The baseline model has 4 main components: **word embeddings**, **context encoding**, **attention flow layer** and **output layer**. The only 2 components that are missing in the baseline from the BiDAF architecture are **character embeddings** and **modeling layer**.

Character embeddings are considered useful for handling unknown (OOV) words, but, as could be seen from the ablation table in [2], adding them boosts BiDAFs F1-score on SQuAD dev set by less than 2 points. After considering this I decided that instead of implementing **character embeddings** I might as well try to reduce the amount of OOV words directly. The baseline model uses GloVe **word embeddings** pre-trained on Wikipedia data with 400K vocabulary. Replacing these embeddings with the ones pre-trained on Common Crawl data with 1.9M vocabulary increases amount of the words known to the model almost fivefold.

Context encoding component in the baseline applies a 1-layer bidirectional GRU to the question and context embeddings. The resulting forward and backward hidden states produced by GRU are concatenated to obtain the context hidden states and the question hidden states. I have modified the encoder in the following ways:

- added support of multiple layers.
- provided option to choose different encoder for question and context (e.g., not to share weights, as baseline does).
- implemented capability to pass final encoding states from question encoder as an initial states into the context encoder (this could be done independently from previous choice of whether to share weights or not), with the idea that it could be useful for the context encoding to be aware of the question representation.
- made RNN cell type configurable (specifically, other than GRU, I have also tried LSTM and LayerNormBasicLSTMCell; the authors of the latter promise performance improvements on NLP tasks [4] over standard LSTM, but, unfortunately, I have found that Tensorflow implementation of it is very slow, so it seems prohibitively expensive to use at the moment).
- set up 3 additional ways to combine final states besides concatenation - adding, averaging and max pooling.

The next step is to apply **attention** to the hidden states produced by the context encoding layer. In the baseline we take basic dot-product attention, with the context hidden states attending to the question hidden states. The attention outputs are then concatenated to the context hidden states to obtain the blended representations. I have made 2 changes for this layer:

- in the basic attention module, I have added element-wise product between attention output and context hidden states, similarly to how it is done in BiDAF. Theoretically, deep network should be able to learn such *feature cross* by itself if needed, but, by providing it, we are simplifying the task for the network and the convergence speed might improve.
- I have also added a new module, with an implementation of Bidirectional Attention Flow from BiDAF paper. Unlike baseline, which has only Context2Query attention, here we also add Query2Context attention. Another difference from the baseline is that the dot product is replaced with trainable *similarity function*. New module could be used instead of basic attention module interchangeably in my implementation.

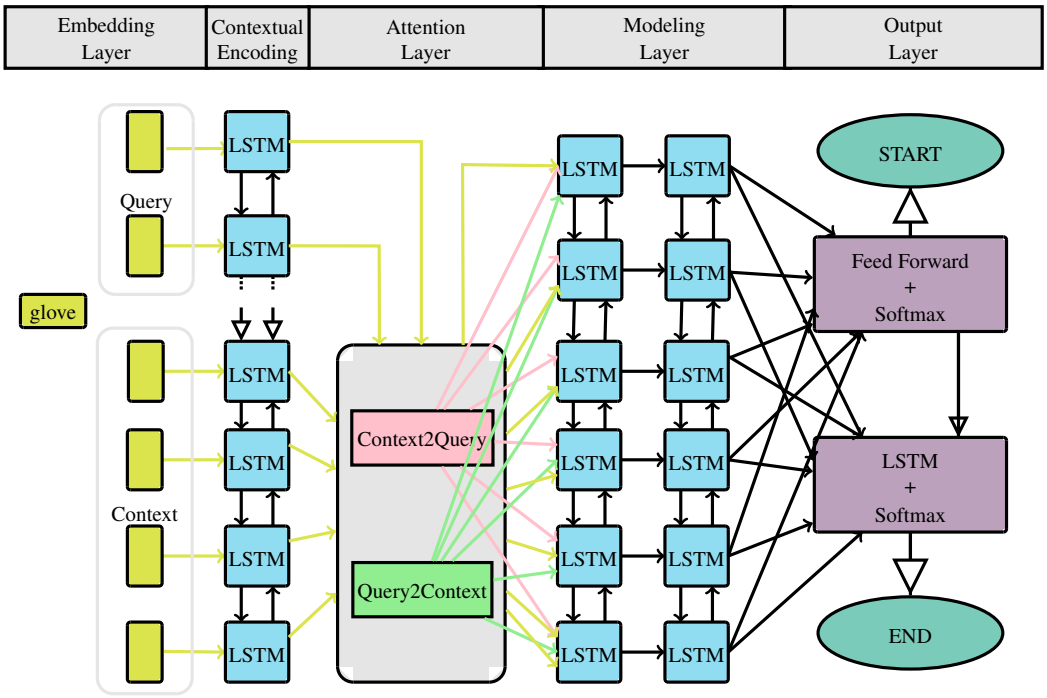

Figure 1: Model Architecture

Next is a **modeling layer**. This is the only new architectural level component that I am adding to the baseline structure. It takes an output of the attention flow layer and encodes it with an RNN encoder, similarly to what context encoding component does. In fact, I am using the same encoder module for both of these layers and all of the options mentioned in the context encoding layer description are applicable in the modeling layer as well.

The **output layer** in the baseline takes representations produced by the previous layer, passes them through the feed forward network and applies softmax to get probabilities of the start positions. The same is done independently for the end positions. BiDAF improves on this by making end predictions dependent on start predictions and by adding another bidirectional RNN before the end prediction module. The final layout of the model is presented in the **Figure 1**.

4 Experiments

While in the previous section I was describing my model architecture from the hierarchical point of view, layer by layer, in this section I will present what I did in the order of implementation and will provide justifications of this ordering. My main goal was to train the best model I could in the limited time I had, which defined my overall strategy for the project. The first priority was to lower training time and to keep it low (for example, for the baseline, getting to peak performance with the default hyper parameters required 15K iterations or 5 hours on GPU, which is way too long).

4.1 Reducing training time

Following the suggestion from the project handout, I have started with looking into the the data. As could be seen from the histogram at **Figure 2**, we could limit *context length* to 300 tokens and get 2x speed up as a result (in exchange for a very small, less than 2%, reduction in training data). On the other hand, I left *question length* limit without change, as questions are much shorter than contexts anyway and there is not much to gain from them in terms of overall speed improvement.

2.5 hours for the baseline training still felt like too much, so I decided to reduce my training time further. To achieve it, I sampled 10 times smaller data set from the training data and did most of my experiments and hyper parameter tuning on it. For example, training baseline to the peak on this data was reduced to 10-15 minutes. Later, when the model became more complex, the training time increased, but mostly stayed under 1 hour.

Another important point in optimizing training speed is to make sure that GPU is always saturated, as the processing time for batches of different sizes is the same, as long as GPU does not crash with OOM. As a consequence, I am always choosing the largest batch size that could fit into GPU memory.

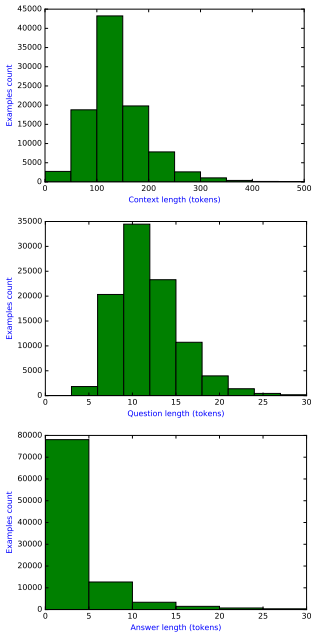

Figure 2: Data distribution.

4.2 Improving spans predictions

After making speed of hyper parameters search reasonable and choosing set of hparams on small data set, I have trained the baseline on the full training data. When looking into the output of this model, one problem was painfully obvious. The model was choosing start and end of an answer span based purely on the softmax output, and it was often producing invalid spans, e.g. spans with the end happening before the start. The simplest possible fix is first to choose a start, and then to choose an end from the interval $[start\ :]$. But we could do better than that! As we already know (see **Figure 2**) the answers are in general very short, so we can limit this interval further to something like $[start : start + 15)$. Finally, instead of choosing start and end separately based on maximum probability of each, we could, as done in [5], choose the span (i, j) such that $i <= j < i + 15$ and probability $p^{start}(i) * p^{end}(j)$ is maximized. This approach produces slightly better results and the resulting number could be interpreted as a *confidence score*, which I will use later when building ensemble.

Model architecture	F1 score (dev)	EM (dev)	F1 score (test)	EM (test)
Baseline	43.37	34.19	n/a	n/a
As above + spans improvements	50.19	39.26	n/a	n/a
As above + LSTM instead of GRU	52.37	41.15	n/a	n/a
As above + 1 modeling layer	65.58	54.32	n/a	n/a
As above + 2 modeling layers	74.7	64.05	n/a	n/a
As above + bidirectional attention	74.84	64.26	n/a	n/a
As above + Common Crawl	75.75	65.71	n/a	n/a
Final model (single)	**76.37**	**66.00**	n/a	n/a
Final model (ensemble)	79.01	70.31	**80.04**	**71.97**
original BiDAF (single model)	77.3	67.7	77.3	68.0
original BiDAF (ensemble)	80.7	72.6	**81.1**	**73.3**
Current top F1 ([6])	n/a	n/a	**89.28**	82.48
Current top EM ([6])	n/a	n/a	88.76	**82.84**
Human Performance ([6])	n/a	n/a	91.22	82.30

Table 1: Performance results and comparisons

4.3 Main modifications

After fixing basic span prediction errors, I replaced GRU with LSTM, as was suggested by my further hparam search on the small data set, and decided to add the modeling component. Adding it produced the biggest gains of all the changes and became very hard for me to improve upon. For example, replacing basic attention with bidirectional in this architecture was not producing any measurable gains for some time, until I also switched to Common Crawl embedding with much larger vocabulary

(see the corresponding F1/EM numbers in **Table 1**). Adding the modeling layer and 300 dimensional embedding led to large increase in model size, which immediately raised an issue of exceeding GPU RAM limit.

4.4 Reducing memory usage

Two things that helped the most with lowering my memory consumption were limiting context size and using maximum for combining hidden states in the encoders output instead of concatenation. Applied together, they have reduced memory footprint of the model by the factor of 4. At the same time, the performance drop from using these optimizations was, from my measurements, below 1%.

Another problem with memory was the size of the checkpoint on disk, which was sometimes large than 100Mb. I solved it purely by luck. When I switched to CommonCrawl embeddings, I have found out that Tensorflow does not support creating tensors larger than 2Gb and that the solution is to create placeholder and feed the embedding into it during run. After this change, checkpoint size dropped below 20Mb even for models with a lot of parameters. I think that without this optimization it would have been much harder for me, if not impossible, to do ensembling later.

4.5 Things that did not work

Some things that were used in the BiDAF paper or in some other high performing SQuAD models, like R-Net [7], for example, just did not work for me for various reasons. One, notably, was Adadelta optimizer. The problem with it that it converges much slower than Adam, so even if it will eventually find a bit better solution, it was just impractical to use under project's time constraints. Another was using additional RNN encoder when predicting ends. I have left this encoder as part of the architecture (turned off by default), but in my experiments I was not able to notice any improvement from using it.

4.6 Finalizing hyper parameters

I have listed all the hyper parameters of my best single model in the **Table 2**. To train this model I've switched back to concatenation and increased context length to 400 to improve final performance by additional 1%. Also, notice that I am not listing batch size in the table. This is because, as I mentioned before, I'm always choosing batch size so that GPU is saturated.

Hyper Parameter	Value	Search Space
Optimizer	Adam	{ Adam; Adadelta }
Learning rate	0.003	{ 0.1 .. 0.0001 }
Context length	400	{ 250 .. 600 }
Context encoder (# of layers)	1	{ 1; 2 }
Context encoder (hidden size)	100	{ 50 .. 200 }
Modeling layer (# of layers)	2	{ 1; 2; 3 }
Modeling layer (hidden size)	75	{ 25 .. 125 }
Dropout	0.19	{ 0.1 .. 0.3 }
RNN cell	LSTM	{ GRU; LSTM; LayerNormBasicLSTMCell }
Grad norm	None	{ None; 1.0 .. 10.0 }
Combiner	concat	{ concat; mean; max; average }
Answer length	23	{ 10 .. 30 }

Table 2: Final hyper parameters

4.7 Ensemble

To improve the numbers a bit further still I am using ensemble of several most recently trained models. I have tried two techniques for ensembling: choosing an answer of a model with the highest confidence score and "voting", when the answer rating is the sum of confidence scores of the models that have chosen this answer. A bit surprisingly for me, voting performs about two times better than "highest confidence". My expectation before trying was that all the models would likely be choosing a bit different span (because the space to choose from is large) and the situation when the same

answer is chosen by more than 1 model will occur very rarely. In reality it seems that the opposite is happening.

To decide whether the specific model should be included into the ensemble, I am using greedy approach. E.g. I am adding models one by one and keep only the ones that improve the results of evaluation. After deciding in this way on a specific set of models to include, I evaluate this ensemble with different limits for answer length set during inference. One surprising thing that I have found from this is that ensembling seems to improve model understanding of longer answers: if for single model optimum limit is around 15, for ensemble it grows with more models added and for 10 models optimum limit rises to 23.

5 Conclusions

At the end, I was able to train a high performing model for Reading Comprehension, which achieved competitive ranking on the class leader board. I had several ideas, that I did not have the time to implement, but I still believe that they could improve my model performance further and might be worse trying in the future. One idea was to use some form of a **dynamic padding** instead of limiting context length and, as a result, to avoid even minimal loss of training data. Another was adding self-attention layer from R-Net paper [7] after the bidirectional attention layer. Judging from the official leader board [6], this is a popular and high performing combination. Finally, I think that the span prediction could still be improved further, so it could be interesting to try pointer networks in the future, like done in [8].

The implementation will be made available at `github. com/ raksitov/ squad`.

References

[1] Rajpurkar, P., Zhang, J., Lopyrev, K. and Liang, P., 2016. **Squad: 100,000+ questions for machine comprehension of text.** *arXiv preprint arXiv:1606.05250.*

[2] Seo, M., Kembhavi, A., Farhadi, A. and Hajishirzi, H., 2016. **Bidirectional attention flow for machine comprehension.** *arXiv preprint arXiv:1611.01603.*

[3] Pennington, J., Socher, R. and Manning, C., 2014. **Glove: Global vectors for word representation.** *In Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP) (pp. 1532-1543).*

[4] Semeniuta, S., Severyn, A. and Barth, E., 2016. **Recurrent dropout without memory loss.** *arXiv preprint arXiv:1603.05118.*

[5] Chen, D., Fisch, A., Weston, J. and Bordes, A., 2017. **Reading wikipedia to answer open-domain questions.** *arXiv preprint arXiv:1704.00051.*

[6] `rajpurkar.github.io/SQuAD-explorer/`

[7] `www.microsoft.com/en-us/research/wp-content/uploads/2017/05/r-net.pdf`

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