Machine Reading Comprehension on the SQuAD Dataset Using R-NET

Jason Luo
Department of Political Science
Stanford University
jasonluo@stanford.edu

William Zeng
Department of Computer Science
Stanford University
wizeng@stanford.edu

Abstract

Machine comprehension is a complicated natural language processing task with several recent breakthroughs thanks to the advent of deep neural networks. In our project, we re-implemented the neural network layers of the R-Net model, including self-attention and an output pointer network, and analyzed their individual contributions to the final model. We achieved F1 score 82 and EM score 70 on train set, F1 69 EM 59 on dev leaderboard, and F1 71.1 and EM 61.5 on test leaderboard. We experimented with different dropout values and regularization methods.

1 Introduction

Natural language processing (NLP) has always been a prominent field in computer science. While the nuances of different languages used to prove difficult to crack with traditional or statistical methods, the development of deep neural networks led to an explosion of exciting new applications for NLP tasks. However, one of the most formidable tasks still is machine reading comprehension, where a computer tries to predict the answer to a question given a passage.

In 2016, the introduction of the Stanford Question Answer Database (SQuAD) [1], which contains over 100,000 question-answer pairs, sparked the development of several models that attempted to tackle the problem of machine comprehension. Such models have continued to improve over time, with January 2018 marking the monumental point where a model was able to beat human performance in the exact match score.

In this paper, we created a model inspired by the R-NET paper [2], using the gated attention-based recurrent neural network (GABR), self-attention, and pointer network layers. We will analyze the improvements in performance each of these layers have over the baseline and visualize how the final model performs on examples in the SQuAD dataset.

2 Related Work

Several successful models have been trained using the SQuAD dataset in the past years. While they have several differing combinations of features, one included in virtually all the submissions is some form of attention, such as bidirectional attention flow [3] and self-attention. Attention is the primary way of encoding information about the question into the passage, and even encoding information in the passage into itself.

Another prominent feature is pointer networks [4], which use something akin to a decoder to select indices of the input as the output. When applied to the SQuAD challenge, there are only two outputs: the probability distribution of the start index, which is based on attention-pooling of the question representation, and the probability distribution of the end index, which is conditioned on the start

probabilities. One of the first papers to utilize pointer networks is Wang et al. [5], which in addition uses a Match-LSTM attention layer.

The paper we tried to re-create, R-NET, contains four layers: word/character embeddings, gated attention-based RNNs, self-attention, and a pointer network. The R-NET model received a EM score of 82.136 and a F1 score of 88.126. A later iteration, R-NET+, earned 82.650 and 88.493, respectively. While the implementation is proprietary, there exists an open-source implementation on GitHub which the TAs told us to reference for practical implementation of the layers [6].

3 Problem Definition and Dataset

The SQuAD dataset consists of multiple passages, with at least one question per passage. Given a passage/context of length P and a question of length Q, the model needs to predict

$$0 \leq i_{start}, i_{end} < P$$

as indices of the context that correspond to the start and end indices of the answer. In addition to using SQuAD as our dataset, we used pre-trained GloVe word vectors for our embeddings.

4 Model

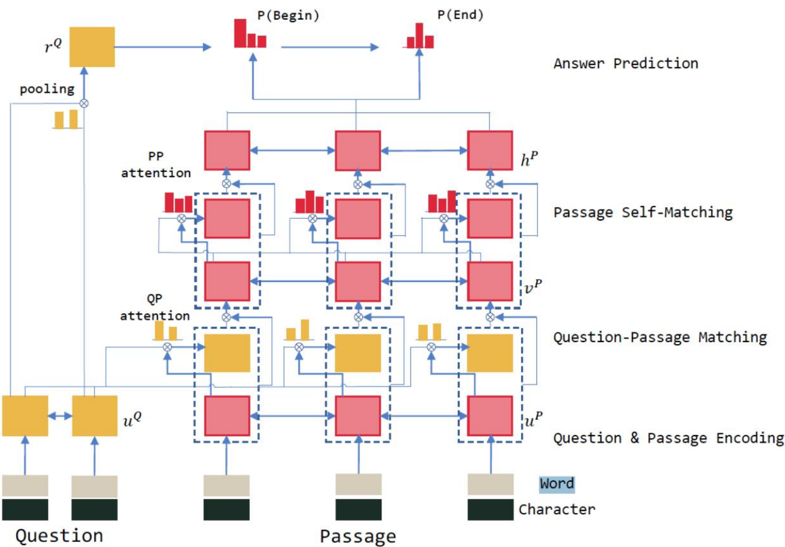

Figure 1: A diagram of the r-net model. Our model includes every layer except for the character embeddings.

4.1 Embeddings

We used the given word embedding layer of the default project. The question and context embeddings are each run through a bidirectional GRU, the outputs of which are concatenated to form the question and context hidden states.

4.2 Gated Attention-Based Recurrent Neural Network

This layer replaces the BasicAttn module of the default project. The context attends to the question in order to produce a question-aware context hidden state that encodes information of the whole question into every time step of the context.

At each time step of a dynamic RNN, the context hidden state is concatenated with the attention-pool of itself, the whole question, and the RNN's hidden state. This concatenated input is then run through an additional sigmoid gate to produce the final input to the RNN at that time step.

4.3 Self-attention

While the GABR layer makes the context aware of the question, the context has limited knowledge about information contained in other parts of the passage. To address this, this additional layer has the context attend to itself to create a self-aware context representation. At each time set of a bidirectional dynamic RNN, the context hidden state is concatenated with the attention-pool of itself and the entire context representation. This pool is again passed through the sigmoid gate, and the final input is fed to the RNN. The outputs are again concatenated to produce the self-aware context representation.

4.4 Answer Pointer

While the default project has the start and end indices predicted separately, the pointer network model conditions the latter on the former. The pointer network is actually a two-stage RNN, referred to as the answer RNN. The initial state of the answer RNN is an attention-pool over the question representation and a trainable parameter. At each time step of the self-aware context representation, the hidden state at that time step is pooled with the answer RNN hidden state to produce the probability distribution of the start index. The attention specified by this distribution is then used as the input to the answer RNN. The previous is then repeated with the new answer RNN hidden state instead of the initial hidden state to generate the probability distribution of the end index.

5 Experiment

We started out using default baseline model, which came with context length 600, hidden size 200, and batch size 100. It ran okay in the baseline, but once we inserted a self-attention layer after the original basic attention layer, training would always run into Out-of-Memory (OOM) issues both locally and on cloud. We tried several ways to combat this problem. First, we experimented and decreased hyperparameters, turning to using context length 300 (which we borrowed from the R-NET paper), hidden size 75, and batch size 30-50. This combination of hyperparameters allowed the model to train but very slowly (50 secs per iteration locally and 15 secs on Azure NV12 machine). We also realized the memory bottleneck came from the GPU, which for M60 is 8GB. We thus switched to NC12, whose K80 machine came with 12GB memory.

The first solution is not a real, sustainable one, but rather a way to evade that issue. If we add more layers in the model, OOM issues will always come up again. Alternatively, we tried to share weights as much as possible between layers and made the hidden layers shrink down to the hidden size even after concatenation at every layer. This solved our problem and greatly improved training efficiency. We were able to set the batch size back to 100 and every iteration only took about 3-5 seconds on Azure.

The following graph presents the training process of our final model. As shown, our model learns quickly in the beginning, achieved relatively good performance after a mere 3k iterations, but also stopped learning to better generalize to dev examples after 7k.

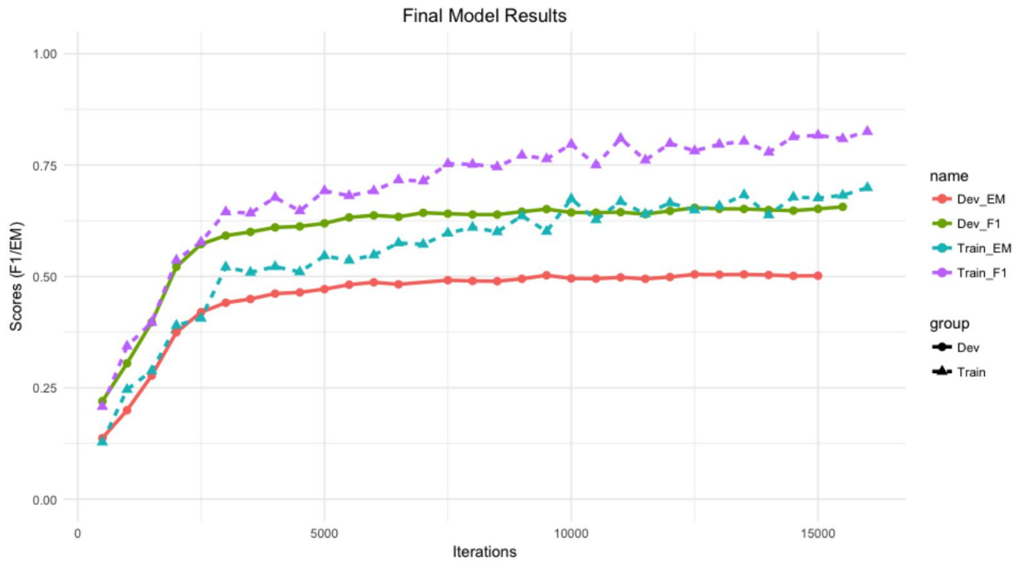

Figure 2: A Plot of Train/Dev Results for the Final Model

Our final model quickly achieved F1 score around 80 and EM around 70 on train set, and F1 around 65, EM around 50 on local dev set, after around 7k iterations. The dev loss shown right started bouncing back after around 8k steps. We also plotted the dev loss to inspect the possible overfitting issue.

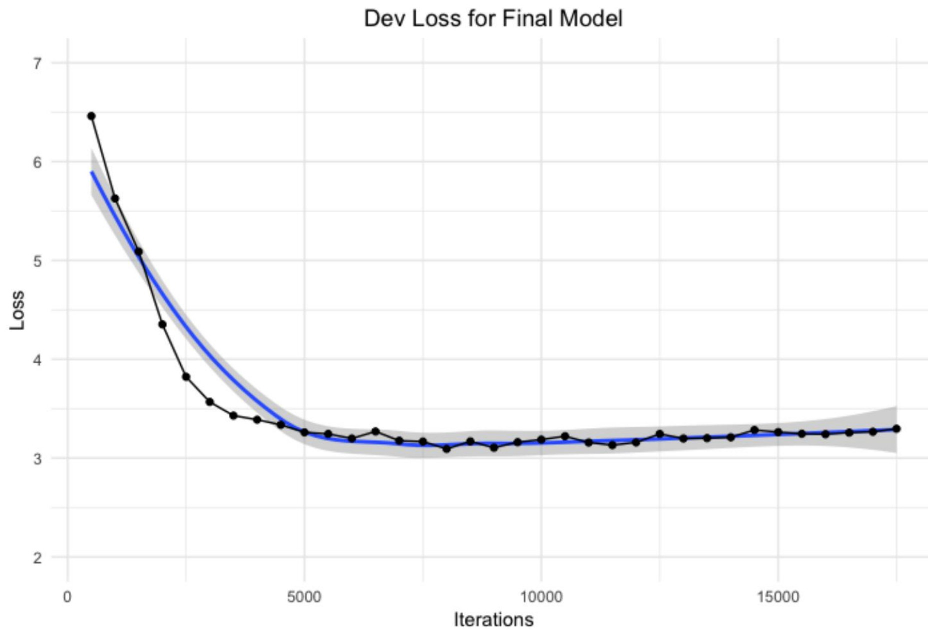

Figure 3: A Plot of Train/Dev Results for the Final Model

It is clear that after around 7-8k iterations, our dev loss stopping decreasing. But the train loss also stopped decreasing and the discrepancy between the two was not closing. This suggested that our model wasn't powerful or complex enough both regarding capturing entire information in the train set and generating to unseen examples.

To address this problem, we tried not sharing weights between layers. But because we had memory size limit and always needed to avoid OOM problem, we first tried not sharing weights for the self-attention layer. We let self-attention layer had its own weights including both the attention-matching part and encoding with the bi-directional GRU part. This model yielded almost the same F1/EM results as our original model did. So the bottleneck of model performance did not come from weights-sharing. It might just because of the model structure. In addition to exploring weights sharing, we also experimented with hyper-parameters.

We then experimented high dropout rate at 0.6, but that turned out to not work so well. After already 11k iterations, the model was still struggling to surpass train F1 30 and EM 20 (shown below). Both train F1 and EM also started trending down, suggesting high-dropout rate at 0.6 really hurt model performance.

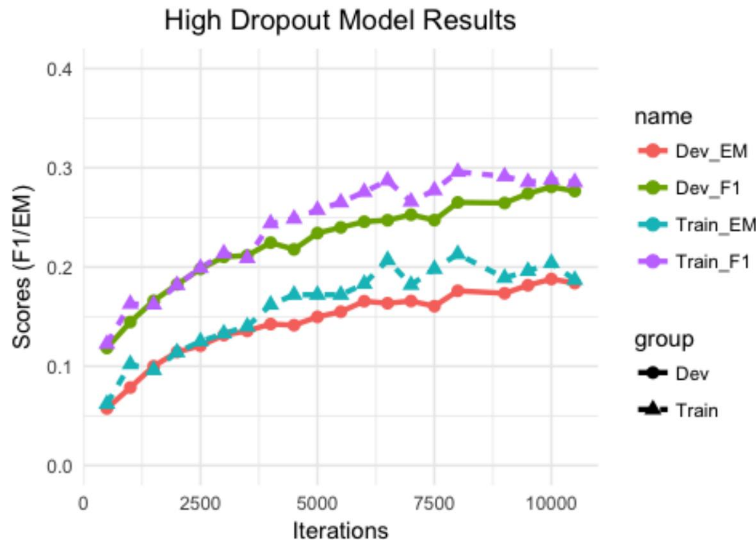

Figure 4: A Plot of Train/Dev Results for the Final Model with Dropout Rate=0.6

We then tried using a medium dropout rate at 0.4 with an L2 loss regularization. The model yielded F1 58 and EM 43 on dev and F1 65 and EM 52 on train, suggesting less discrepancy between train and dev results and thus less over-fitting. But the model now also failed to achieve the optimal results we had with dropout rate = 0.2, suggesting even medium level drop-out can affect our model undesirably.

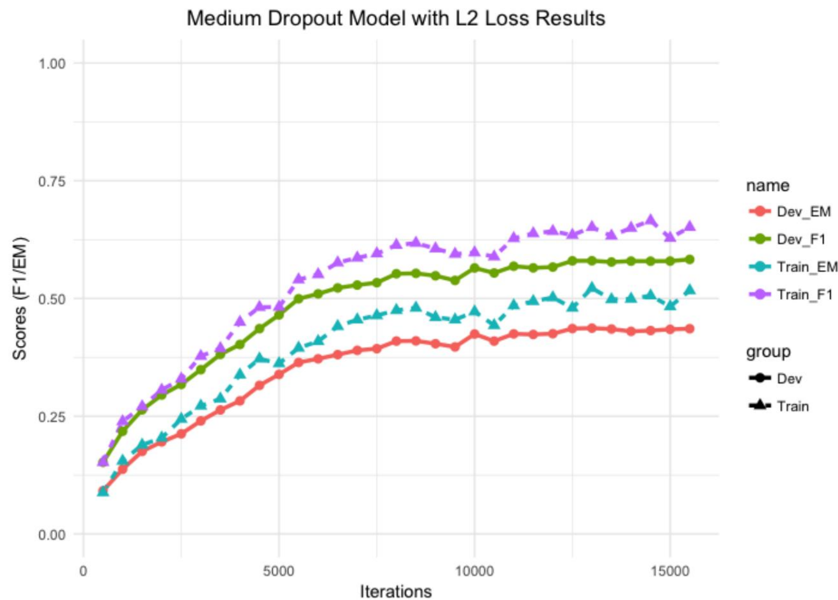

Figure 5: A Plot of Train/Dev Results for the Final Model with Dropout Rate=0.2 and L2 Regularization Loss

6 Conclusions and Future Work

The following table presents performance break-downs of the layers and features in our model. Our experiments suggested that the self-attention layer and GABR performed the best with a dropout rate of 0.2. L2 loss seemed unnecessary if drop-out is applied. The idea of self-attention that having context hidden states attend to themselves is powerful and can improve computational speed compared to RNN structure while maintaining similar levels of performance (as argued in the "Attention is All Your Need" paper[6].)

Model Variant	Train F1/EM	Dev F1/EM
Baseline (basic attention + BiGRU)	64/52	44/35
Self-attention only (GloVe 100d, hidden size 75, batch size 30)	48/35	45/31
Self-attention only (GloVe 300d, hidden size 200)	69/53	61/46
Self-attention + Pointer Network (GloVe 300d, hidden size 75)	68/53	61/45
Final model : Self-attention + GABR + Pointer Network (dropout=0.2)	82/70	65/50
Final model (with high dropout=0.6)	28/20	27/18
Final model (with medium dropout=0.4 and L2 loss)	65/52	58/43

Figure 6: Each Layer's Performance over Baseline Model

Since the F1 and EM scores of both the training and dev sets plateaued, we believe our final model was under-fitting the data. Even in the baseline model, the training loss would continue to decrease over time, and the dev loss would reach some minimum value then start to climb, which indicates over-fitting. Since our model doesn't reach that point, we believe it's not powerful enough to capture all the information available during training. Possible fixes would be to get rid of dropout, not share weights (for example, the vector used for attention calculation is shared among all three layers), and not shrink the hidden size at each layer. However, as mentioned above, that would have likely caused us to run into OOM issues.

In the future, we would like to add character embedding to our model to improve the encoding layer. We would also have liked to explore other methods of attention and how they could be combined with our current attention layers. We would also try to combat underfitting by making the changes above, and using a TensorFlow memory profiler to try to prevent OOM issues. Finally, we would experiment with more hyperparameters layer sizes, regularization methods, and CNN/RNN types.

Acknowledgments

We are extremely grateful to the CS224N course staff for providing the default implementation and copious amounts of assistance throughout our project. We would also like to thank Richard Socher for teaching the class and Microsoft for providing us Azure credits to train our models with.

References

[1] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. *CoRR, abs/1606.05250*, 2016.

[2] Wang W, Yang N, Wei F, et al. Gated self-matching networks for reading comprehension and question answering. *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, ACL 2017, *Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pp. 189198, 2017.

[3] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603*, 2016.

[4] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer Networks. *arXiv preprint arXiv:1506.03134*, 2015.

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

[6] A Tensorflow Implementation of R-net: Machine reading comprehension with self matching networks. *https://github.com/minsangkim142/R-net*

[7] Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, ukasz Kaiser, and Illia Polosukhin. "Attention is all you need." In *Advances in Neural Information Processing Systems*, pp. 6000-6010. 2017.