An Exploration of State of the Art Techniques for Question Answering Systems

James Payette
Department of Computer Science
Stanford University
Stanford, CA 94305
jpayette@stanford.edu

Abstract

In this paper, I explore the impact that a number of different techniques for feature extraction, encoding, and attention have on model performance on the Stanford Question Answering Dataset (SQuAD). After exploring the marginal benefits that each technique delivers over that of a simple baseline model, I combine the techniques that showed promise in increasing performance or training efficiency to train models to perform well on SQuAD. I then go into error analysis and consider possible extensions that could further increase performance.

1 Introduction

The SQuAD challenge invites participants to create a question answering (QA) system using the publicly available SQuAD training development set that will generalize well to the private SQuAD test set. Models are evaluated using F1 and Exact Match (EM) statistics that are calculated over unseen test data. While the highest performing systems in the world come close to human performance (F1: 91 and EM: 82), the problem is generally considered a very difficult one, and high performing models are generally very complex and require a lot of computing resources.

The primary objective of this project was to create a model that achieved high performance on the SQuAD challenge. The secondary objective of this project was to explore the marginal utility of a number of different techniques proposed recently as optimizations for QA systems. For each technique, I replaced a single module of the baseline model with an optimized module, and measured the difference in performance. Finally, I used this data to combine the most compelling of these modules into two models in order to achieve high performance on the SQuAD challenge.

The baseline model that I used was provided as the baseline model by the CS224N teaching staff [1]. It consisted of a GloVe word embedding layer feeding into a single bidirectional RNN encoding layer, followed by a basic attention layer, and a simple output layer that takes the maximum joint probability over all possible spans in the context. I focused on optimizations in each of the four layers, and I used the highest performing optimization (keeping efficiency in mind) for each layer in the final model.

2 Background and Related Work

Since its release in 2016, the SQuAD challenge has drawn a lot of attention in the NLP community. Since the writers of the original paper introduced a model with F1 score of 51.0% [6], the public leaderboard at `https://rajpurkar.github.io/SQuAD-explorer/` shows that the ML community has been creeping closer and closer to human level performance on the dataset [10]. Many groups have released their novel research alongside their high performing models, many of

whom attribute high scores to complex forms of attention such as Bidirectional Attention Flow and Dynamic Coattention [8], [11]. Other improvements over simple encoder/decoder models have been proposed including adding attention layers and more complex RNN cells in the output layer to capture more complex dependencies [7], [8], [9]. While using word embedding layers like word2vec and GloVe are standard, groups have also suggested using embeddings at the character lever, along with character level CNNs [3], [4], [5], [8]. Another recent advancement in the field of deep learning comes with the introduction of ResNet, which makes use of shortcut connections to allow for incredibly deep models to learn efficiently [2].

3 Problem Definition

The SQuAD challenge can be formalized as follows. Each of the 100k+ datapoints in the dataset consists of a context, a question, and an answer. The context is a list of words $[c_1, ..., c_n]$ that gives enough background knowledge to answer the question (in fact, contexts are paragraphs taken from Wikipedia). The question is a list of words $[q_1, ..., q_m]$ that asks for some specific knowledge about the context. The answer is a list of words that $[c_k, ..., c_{k+j}]$ that is some slice of the context. As a result, answers can be represented as two indices, the start index (corresponding to the index of the word in the context where the answer begins) and the end index (corresponding to the index of the word in the context where the answer ends).

3.1 SQuAD Challenge Objective

The objective of the SQuAD challenge is to build a system that can successfully answer unseen questions given unseen contexts. Note that QA systems are not expected to generate text, but instead find the correct span for a question given a context. Systems are evaluated on two metrics: EM Score and F1 Score. One final note is that each datapoint contains 3 human-labeled true answers. The F1 and EM scores for a prediction are calculated for each 'true' answer, and the maximum value is used.

3.2 EM Score

EM score is defined on a datapoint as the binary measure of whether or not the system predicts the exact span given as the target label. The total EM score for a system is just the pointwise EM score averaged over the entire dataset.

$$\text{EM Score} = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}\Big\{\text{PredictedSpan}(i) == \text{TrueSpan}(i)\Big\}$$

Because of the somewhat arbitrary decision about the boundary of many answers ("Lincoln" and "Abraham Lincoln" may refer to the same person and would be considered by most humans a correct answer to "Who was the 16th president?", but would be different spans in a context and thus would be a miss if the label was one and the system predicted the other) EM is considered a strict evaluation metric [1].

3.3 F1 Score

F1 score is an attempt to relax some of the strictness that EM suffers from. F1 score is defined to be the harmonic mean of recall and precision. Recall (in this context) can be thought of as the percentage of the predicted answer that is a subset of the true answer. Precision (in this context) can be thought of as the percentage of the true answer that is a subset of the predicted answer. Using these definitions, we define F1 score

$$\text{F1 Score} = \frac{1}{n}\sum_{i=1}^{n} \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

4 Approach

In the following section, I'll be explaining each of the optimizations that I implemented in depth.

4.1 Character Level CNN

In order to augment the feature space of my system, I implemented a character level embeddings layer and convolutional neural net (CharCNN) to go alongside the pre-trained GloVe layer. The CharCNN operates on the word level. First, a word is translated into a zero-padded array of character tokens. These tokens are run through a trainable character embedding layer, so that each token is represented by a low-dimensional vector. We then run the word (which is now a two-dimensional matrix of size (max word length) x (embedding size)) through a convolutional layer, such that the output h_i of each convolution can be thought of as a function of a window of character embeddings $[\mathbf{e}_{i-k}...\mathbf{e}_i...\mathbf{e}_{i+k}]$. A visualization of a one-dimensional convolution can be seen in Figure 1. We then use a max pooling layer to get the maximum h_i over all of the over all of the computed hidden states for a given word to get the character level embedding for a given word. We append this character level word embeddings to the GloVe word embedding to get the final word embedding for every word in the input. This concatenation of embeddings is the output of the embedding layer.

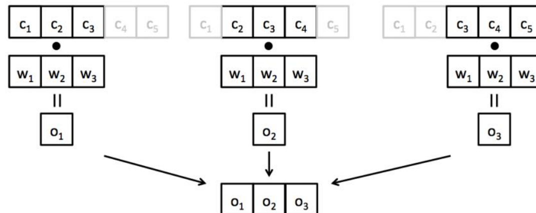

Figure 1: o is the output of a convolution over c with window size 3

4.2 Stacked RNNs

Rather than using a single bidirectional RNN as the encoding layer for my model, I decided to stack multiple RNNs to increase the complexity of my model. Each layer of the encoder is a separate bidirectional RNN. The output of each layer serves as the input of the next layer. While adding layers increases the number of parameters in the system (which will slow down learning and eventually lead to memory issues) stacking RNN layers allows the model to learn more complex dependencies between inputs. Since generating a correct answer to a question requires a relatively complex understanding of the question, a complex model is required.

4.3 Shortcut Connections

It's no secret that as neural networks get deeper they get harder to train. Gradient signals can have a much more difficult time flowing back through deep network, and as I stack more and more layers in my encoder, the network becomes more and more difficult to train. By adding shortcut connections in the encoder, I can facilitate gradient flow through much deeper networks, making it much easier to train more complex models. The basic idea behind a shortcut connection is that the input is added directly to the output of a neural network layer (in this example, an encoder layer). As a result, there is a direct link between input and output, which allows gradient to more freely flow backwards through the network. Shortcut connections make training deep networks more robust to unlucky initialization, which makes successfully training a model much easier. A visualization of the stacked RNNs with shortcut connections can be found in Figure 2.

4.4 BiDirectional Attention Flow - (BiDAF)

Bidirectional attention flow (BiDAF) is a more complex attention mechanism first proposed in [8]. In the simple attention layer provided in the baseline code, context hidden states attend to question hidden states. The basic idea behind BiDAF is that attention should flow in both directions. In other words, question hidden states should attend to context hidden states and context hidden states should attend to question hidden states. To implement this layer, a similarity matrix S is calculated, where $S_{i,j}$ is a calculated similarity between context state c_i and question hidden state q_j. This similarity matrix is used to calculate a question to context attention and a context to question attention.

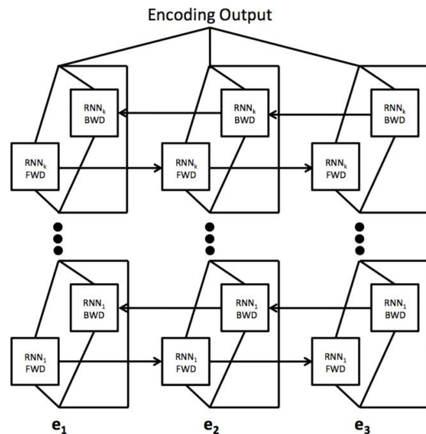

Figure 2: k stacked RNNs with shortcut connections

The exact equations for these calculations can be found in [1]. The final output of the layer is the concatenation of a collection of vectors calculated in this layer.

4.5 Coattention

Coattention is another complex attention layer first proposed in [11]. Like BiDAF, it also involves attention that flows from question to context and vice versa, but it also adds on an extra layer of attention. The coattention layer itself attends to the output of an attention layer. In this layer, the question hidden states q are put through a fully connected neural net layer to produce q'. Then an affinity matrix L is calculated, where $L_{i,j}$ is the inner product of c_i and q'_j. The rows of this matrix are used to calculate the context to question attention output, and the columns of this matrix are used to calculate the question to context attention output. The context to question attention distribution is then used as a weighting for the question to context attention outputs to get the second level distributions. The context to question attention outputs are concatenated with the second level attention outputs, and these are run through a bidirectional RNN to get the output of this layer. The equations for all of these calculations can be found in [1].

4.6 MultiAttention

MultiAttention is simply the concatenation of the output of a Coattention layer and a BiDAF layer. Since both attention optimizations provide a boost in performance over the baseline, I hypothesized that using the concatenation of the two as the attention layer might lead to even better performance.

4.7 ConditionOutput

Rather than a the simple downprojection and masked softmax layer used in the baseline, my ConditionOutput layer is an output layer motivated by the output layer described in [8] with a few modifications. The output layer takes the blended representation b returned by the attention layer and feeds it though a bidirectional RNN. It then concatenates this output o_{start} with b to get $[b; o_{start}]$ and applies a linear downprojection and a masked softmax to get the start probability distribution. Next, it feeds o through another bidirectional RNN. It concatenates this output o_{end} with b to get $[b; o_{end}]$ and applies a linear downprojection and a masked softmax to get the end probability distribution. This method allows the model to take the start probability distribution into account when calculating the end probability distribution. The start and end position are calculated as arg-maxes over these two distributions, just like in the baseline. Figure 3 shows the architecture of the ConditionOutput module.

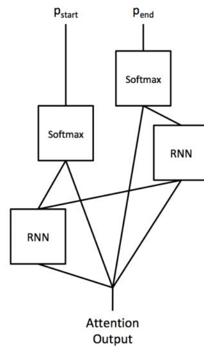

Figure 3: The architecture of the ConditionOutput module.

4.8 Final Model

Based on the outcome of the experiments pertaining to each of the above optimizations, I decided to run two separate experiments for the final model. The first, called FinaLite, is a relatively lightweight model that uses a basic GloVe word embedding layer, a single bidirectional RNN in the encoding layer, a coattention layer, and a ConditionOutput layer. This model incorporates the optimizations that provided the biggest performance gains and keeps the rest of the layers as basic as possible. The idea behind this model is that it will be more lightweight, making it easier to train and tune.

We'll also train a Final model that uses the best optimization for each layer. The model consists of an embedding layer that is combination of the GloVe embeddings and the CharCNN layer, an encoding layer contains three stacked bidirectional RNNs with shortcut connections, a coattention layer, and a ConditionOutput layer. This model is heavyweight, and requires a fair amount of tuning to even get to a point where it does not exhaust the memory of the GPU. However, this should (in theory) perform well as it is a step up in complexity from any of the other models.

5 Experiments

In this section, I'll first report and reflect on the intermediate experiments. Then, I'll use the outcomes of these intermediate experiments as motivation for final model selection, report the results of the final model, and analyze the model in terms of its successes and its failures.

5.1 Intermediate Experiments

For each of the intermediate experiments, hyperparameters were chosen to match that of the baseline. Some experiments required adding more hyperparameters, and these will be commented on in their respective sections. Each model uses an Adam Optimizer with a learning rate of 0.001, a dropout rate of 0.15, and uses bidirectional GRUs as the RNN cells.

5.1.1 Embedding Layer Experiments

For the embedding layer experiment, I compared the performance of a model with a character level CNN to that of the provided baseline. The charCNN layer's architecture is described in section 4. Table 1 shows the results of this experiment. For hyperparameters, I selected a character embedding size of 20, a window size of 5, and an output size of 100. Based on the results of this experiment, I decided to included a CharCNN in the embedding layer of my final model. Because it is relatively costly and only provides a slight performance boost, it is not included in the FinaLite model.

Table 1. Intermediate Experiment Results

Model	F1	EM
Baseline	43.36	34.56
CharCNN	45.22	36.13
ML-RNN	46.66	37.61
ML-RNN (SC)	46.76	37.94
BiDAF	50.03	40.12
Coattention	65.15	54.18
MultiAttention	65.74	55.00
ConditionOutput	68.42	57.76

5.1.2 Encoding Layer Experiments

For the encoding layer experiments, I compared the performance of a model with an encoder consisting of 3 stacked RNNs both with and without shortcut connections to that of the baseline model. Table 1 shows the results of this experiment. Stacking RNNs provides a marginal performance boost to the model at the cost of computation time. A depth of three was chosen because, although stacking more layers will likely cause a slight increase, the cost in terms of computation time and model size is not worth the tiny marginal increase in performance. The shortcut connections, unfortunately, have no impact on the performance of the model. This is likely because the encoder is relatively shallow. If I had the time and resources to train a model with a deeper encoder, it's likely that at a certain depth the shortcut connections would help the model learn more efficiently. Because the multilevel encoder provides only a marginal performance increase and is relatively costly in terms of compute, we use only a single bidirectional RNN in the FinaLite model. Three stacked bidirectional RNNs are used in the encoding layer of the Final model, as that number seems to be a good compromise between performance and computation cost. I've also decided to include shortcut connections, as they may help gradient flow back into the embedding layer and facilitate learning in the CharCNN.

5.1.3 Attention Layer Experiments

For the attention layer experiments, I compared the performance of models with BiDAF, Coattention, and MultiAttention layers to that of the baseline. Table 1 shows the results of this experiment. While each of the three models sees a large leap in performance over the baseline, the Coattention model far out performs the BiDAF model, and the MultiAttention model sees only a negligible increase over the Coattention model. Because the BiDAF layer is a strain on GPU memory and does not seem to add any marginal benefit over the Coattention layer, the Coattention layer is used as the attention layer in both of the final models.

5.1.4 Output Layer Experiments

For the output layer experiment, I measure the performance difference between a baseline model and a model with a ConditionOutput layer. The results of this experiment can be found in table 1. These results are stunning, as the ConditionOutput layer saw enormous performance gains over the baseline. In fact, the marginal gains from adding this layer are higher than that of any other optimization in any other layer. As a result, both final models use a ConditionOutput layer as their output layers.

5.2 Final Model

As described in the approaches section, I trained and evaluated two models to predict on the test set, a Final model and a FinaLite model. The results for these experiments on the dev set can be found in table 2. I attempted to run a hyperparameter search for both models, but because of memory constraints (GPUs are limited to 8 GB, which limits model size) and time constraints (models take days to train, with the FinalModel not reaching optimal performance within 5 days) the extent of the hyperparameter search was limited. Final++ and FinaLite++ models are actually identical to their non-incremented counterparts, except that the end prediction index is constrained to be after

the start prediction index. This optimization was added after a first pass of error analysis, which will be explained in the section below.

Table 2. Final Model Performance

Model	Dev F1	Dev EM	Test F1	Test EM
Final	40.46	31.70		
FinaLite	70.17	60.15		
Final++	44.20	34.99		
FinaLite++	**72.41**	**62.20**	**73.25**	**63.42**

5.3 Error Analysis

After training my first FinalLite model, I examined over 100 example datapoints to try to gain insight into where my model struggles. My hope was that patterns would emerge in the data (most importantly, the datapoints that my model was incorrect on) and that I could make improvements to my model based on these patterns. The first obvious issue with my model was that, for a nontrivial amount of examples, the system predicted the end location to be before the start location. The outcome of a prediction like this is essentially an empty answer. This seemed problematic to me, because not only does a data point with a prediction like this score 0 on EM and F1, it doesn't even stand a chance. While it's possible the system only outputs a prediction like this if it is completely baffled by the question (and thus enforcing the constraint that end cannot be before start might not be all that helpful), by enforcing this constraint we at least give the system a chance to make an educated guess.

I implemented this constraint in the prediction system, and re-evaluated my models. The FinaLite model running under this constraint is the FinaLite++ model, and it sees a non-trivial boost in performance. The interesting aspect of this optimization is that I was able to see a noticeable boost in performance without actually having to retrain my model.

I wasn't able to see any other obvious examples by just looking at examples, so I decided to break the dataset up over a few different axes and looked at how my model performed. I looked at model performance based on question type, question length, answer length, context length, and answer location within the context. Most of these explorations provided uninteresting results (i.e. the model performed similarly across different buckets, or the performance difference across buckets is small and easily explained. For instance, the system does slightly worse on very short questions, which makes sense as there is less information encoded in these questions). Tables for FinaLite++ performance on these models are included in the supplementary materials.

One analysis that did reveal some interesting results was the analysis of model performance based on question type. I broke the dataset up across the 7 most prevalent question types. The categories and results can be found in table 3. I noticed a few interesting patterns looking at this table. The model performs really well on 'when' questions, implying that it has a good understanding of the date/time regime. Poor performance on the 'why' questions is to be expected. Why questions are generally more abstract, and require a much more nuanced understanding of context. It shows that, despite the reasonable complexity of the model, there are still improvements to be made. The class of questions that I found most interesting, however, is the where questions. These questions should be (for the most part) relatively simple and straightforward. I would have expected the model to perform similarly on the 'where' questions as it does on 'who' questions, as each requires finding and identifying a specific named entity. However, the model performs noticeably worse on 'where' questions. This insight helps to identify a category of questions for which the model may be ripe for improvement. I'll propose a potential extension in the next section that might help the model with these questions and provide a relatively substantial performance boost.

Table 3. FinaLite++ Performance based on Question Type

Question Type	F1	EM	Question Type	F1	EM
who	73.12	65.00	what	62.88	47.58
when	81.07	70.27	where	60.99	44.55
why	54.99	21.33	how	66.29	49.09
which	66.15	52.99			

Finally, I looked at the learning curves to try to understand why the Final models weren't able to outperform the FinaLite models. While I initially assumed that the more complex Final models had quickly overfit the data and thus did not generalize well to the dev set, I quickly realized that they had a higher training loss than the FinaLite models. Further, although the loss and F1/EM performance seemed to plateau a number of times, manually lowering the learning rate allowed the Final models to continue to improve. As a result, I've concluded that problem is not with overfitting but rather with training. More complex models are more difficult to train and are more sensitive to initialization, and my experiments are no exception to that rule. I believe that, given a few more days (or weeks) and more intense babysitting of the training of the Final model, its performance would continue to improve. At some point, however, the cost of time and computation outweighs the marginal increase, and in the context of this problem, that is the case.

6 Conclusion

The experiments I conducted for this project were certainly eye opening. They reinforced the importance of a smart attention layer, and allowed me to conclude that complex output layers are also an important part of any question answering system. I also learned of the inherent tradeoff between model complexity and required resources (in both time and memory), and that lightweight models that are constructed intelligently can achieve high performance and are much more robust to initialization and hyperparamater tuning, making them much easier to train. Further, While the results from my experiment are promising, there are a number of improvements that could be made that would further increase optimization. First and foremost, it's alarming that the Final Model was not able to outperform the FinaLite model, even on the training set. As I stated above, the training time required for the Final Model really limited the hyperparameter search that I was able to complete. Complex models are notoriously difficult to train and are more sensitive to initialization, so a more thorough search likely would have led to much better performance, and it's an investment I'd definitely make if given more time and compute resources. Further, based on the error analysis I did, it seems like enriching feature vectors (with NER tags and POS tags, for instance) might help the system in some areas that it currently struggles, like in answering "who" questions. One main drawback to this system is that many of the important tokens in both the context, questions, and answers are proper nouns, which are less likely to have entries in the word embedding table. Character embeddings may have helped with this drawback to some extent, but enriching feature vectors with more information would likely help the model deal with these scenarios more effectively and achieve better performance.

References

[1] CS 224N Default Final Project: Question Answering (2018). Online. http://web.stanford.edu/class/cs224n/default_project/default_project_v2.pdf

[2] He, K., Zhang, X., Ren, S., Sun, J. (2015). Deep Residual Learning for Image Recognition. arXiv:1512.03385.

[3] Mikolov T., Chen, K., Greg Corrado, G., Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781 [cs.CL].

[4] Mikolov, T., Sutskever I., Chen K., Corrado G., Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. arXiv:1310.4546 [cs.CL].

[5] Pennington, J., Socher, R., Manning, C.D. (2014). GloVe: Global Vectors for Word Representation. Online. https://nlp.stanford.edu/pubs/glove.pdf

[6] Rajpurkar, P., Zhang, J., Lopyrev, K., Liang, P. (2016). SQuAD: 100,000+ Questions for Machine Comprehension of Text. arXiv:1606.05250.

[7] See, A. (2018). Machine Translation, Sequence-to-Sequence, and Attention. Stanford University. CS224N Lecture.

[8] Seo, M., Kembhavi, A., Farhadi, A., Hajishirzi, H. (2016). Bidirectional Attention Flow for Machine Comprehension. arXiv:1611.01603.

[9] Socher, R. (2018). Vanishing Gradients and Fancy RNNs (LSTMs and GRUs). Stanford University. CS224N Lecture.

[10] SQuAD: The Stanford Question Answering Dataset (2018). Online. https://rajpurkar.github.io/SQuAD-explorer/

[11] Xiong, C., Zhong, V., Socher, S. (2016). Dynamic Coattention Networks for Question Answering. arXiv:1611.01604.