BiDAF Model for Question Answering

Ramon Tuason **Daniel Grazian** **Genki Kondo**

Stanford University
Department of Computer Science
{rtuason, dgrazian, genki}@stanford.edu

Abstract

Over the years, many companies and research groups have invested many resources towards the development of question answering systems because of their latent potential in understanding unstructured text. A very active part of the field of artificial intelligence, these systems seek to capture the semantic and syntactic relationships between words as they are used in modern language. In our study of cutting-edge question answering models, our group has sought to recreate Bidirectional Attention Flow, one of the most successful models in recent years.

In this project, we implement a bidirectional attention flow neural network to answer questions in the SQuAD dataset. We encode the question and corresponding context paragraph using GloVe word embeddings, compute an attention matrix, and decode to find the answer to the question in the context paragraph. Our approach is largely modeled off that described in Bi-Directional Attention Flow for Machine Comprehension (Minjoon Seo et al). We believe we have implemented a good model, but our results at this time are less good than we would hope. We will describe these results, as well as our methodology, possible flaws, and improvements we hope to make in the future.

1 Introduction

We aim to answer questions in the SQuAd dataset, which consists of tens of thousands of paragraph-question pairs. The answer to a question is a contiguous segment of the corresponding paragraph. For example, given the paragraph, "The adults laughed, but the children were scared by the howling wolves" and the question, "Who were scared?", the answer would be, "the children." The answer key is human-curated.

Question answering is a major goal in natural language processing. While high-performance general question-answering remains beyond the reach of the machine learning field, computers are capable of strong performance in many domain-specific question-answering tasks, and they can perform quite well when substantial restrictions are placed on the problem domain. Our case is an example of the latter: the fact that the answer to the question must be a contiguous segment of the corresponding paragraph reduces the problem to finding the start and end words of the answer.

Answers are evaluated by two primary metrics, F1 score and EM score. F1 score is a measure of the overlap between the model's guesses and the true answers, while EM score is the proportion of the test questions that the model answers exactly correctly. State of the art models achieve F1 and EM scores of approximately 80% and 75% respectively, while humans score over 90% and 85% respectively on these metrics.

2 Background and Related Works

Various neural models have been used to attack this problem. We discuss two of them here. It is worth noting that most successful models have used long short term memory units (LSTMs) to capture interactions over long sequences of words (necessary for context paragraphs that may run for hundreds of words.) Successful models also use some sort of attention mechanism, to capture relevance of words in the question and context paragraph as an intermediate step to answering the question.

2.1 Dynamic Coattention Networks

Caiming Xiong et al. use dynamic coattention networks to capture the relevance of each word in a question for each word in a context paragraph. They encode the question and context paragraph with an LSTM whose inputs are first the word vector embeddings of the words in the question and then the word vector embeddings of the words in the paragraph. The same LSTM is used for encoding the question and the paragraph in order to condition one on the other.

The question and paragraph encodings are multiplied and the result is normalized to obtain attention 'weights' for each word in the question for each word in the paragraph, and for each word in the paragraph for each word in the question. In other words, given a word in the question, the attention mechanism encodes the relative importance of each word in the paragraph ('none' is a choice if the word in the question is deemed irrelevant.) Likewise, given a word in the paragraph, the attention mechanism encodes the relative importance of each word in the question ('none' is again a choice).

Decoding is done by alternating between predicting the start word and end word of the answer until convergence, the prediction being a function of the coattention matrix and the previous start and end predictions. This iterative process "allows the model to recover from initial local maxima corresponding to incorrect answer spans."

2.2 Dynamic Chunk Reader

Yang Yu et al. use a similar attention mechanism (they calculate attention values for each pair of words in the question and context paragraph) but they aggressively filter possible answers by selecting an answer from among candidate "chunks". The candidate chunks are determined by linguistic parsing of the roles (e.g. parts of speech) that the various words play in a sentence. Considering only these candidates substantially increases the training rate but leads to some correct answers not even being considered. But the authors report the correct answer was among the candidate chunks 92% of the time.

3 Implementation

3.1 Baseline, V1

Our baseline model begins by encoding the question with a bidirectional LSTM. Let q be the GloVe encoding of the question and p be the GloVe encoding of the context paragraph. We let:

$$h_q = \text{LSTM-output}(q) \tag{1}$$

The inputs to the LSTM are the GloVe embeddings of the words of the question, and h_q is specifically the final output state of the LSTM. The forward and backward outputs are concatenated.

We then encode the paragraph using a bidirectional LSTM, where the inputs are the words of the paragraph concatenated with the final hidden state of the question LSTM:

$$h_p = \text{LSTM-states}(p; h_q) \tag{2}$$

This allows the representation of the paragraph to be conditioned on the question.

Let H be h_q, concatenated to each column of h_p. We then decode using a linear model with a softmax:

$$O = \text{softmax}(WH + b) \tag{3}$$

to get a 'probability' distribution over the words of the context paragraph. The peak of the distribution is taken to be the start token of the answer. An analogous linear model is run to determine the end token of the answer.

3.2 Basic Attention, Dropout, and Gradient Clipping

Next we added some improvements, including a basic attention model, dropout, and gradient clipping. Our basic attention model calculates an attention value for each word in the paragraph, and our final representation of the paragraph is the hidden states of the paragraph LSTM weighted by the attention values. We also add dropout (drop probability = .15) to our model, and we add gradient clipping (capping gradient at 20) to prevent exploding gradients. These improvements made a modest impact on our model's performance.

3.3 Bidirectional Attention Flow (BiDAF)

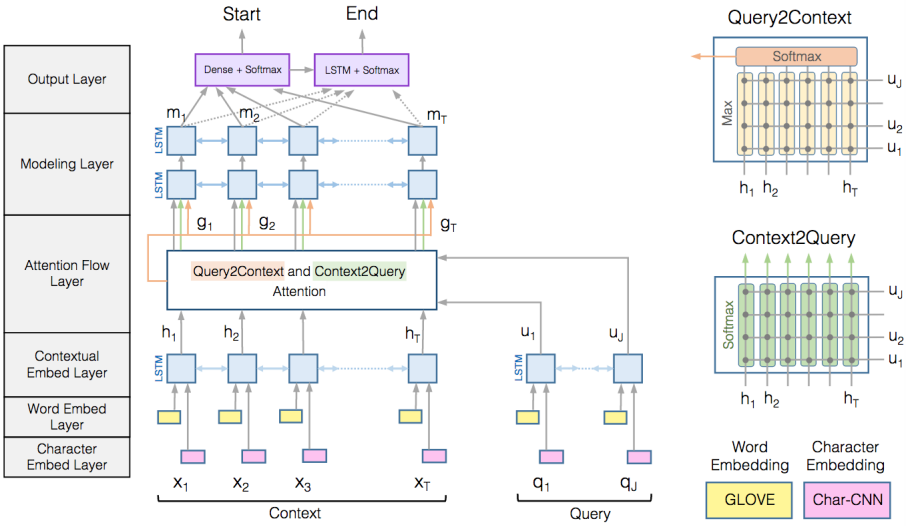


Diagram of the BiDAF model, taken from the paper by Minjoon Seo et al. Note that our implementation omits the character embedding layer, due to memory constraints.

Our main model was a Bidirectional Attention Flow (BiDAF) model, based closely on the paper by Minjoon Seo et al. mentioned in the abstract. We omitted the character embedding layer, because including it was consistently causing out-of-memory errors when running against Azure GPU's (see below for more discussion on this and other obstacles we encountered.)

As in the baseline, we first use a word embedding layer that maps each word to an embedding vector (using pre-trained GloVe embeddings to represent words), for both the question and context paragraph. We then encode the question and context paragraph embeddings by using a BiLSTM on each and concatenating the outputs of the forward and backward LSTMs. However, instead of just taking the final hidden state output of the BiLSTM as we did in the baseline, this time we took h_q to be the collection of *all* the hidden states of the BiLSTM. This enables the subsequent attention mechanism, by giving us a separate hidden state for each word in the question. Next, we added an attention flow layer, in which we calculate a similarity matrix capturing the similarity between each word in the paragraph and each word in the question. From this similarity matrix, we derive context-to-query and query-to-context matrices, which respectively represent the relevance of each word in the question to each word in the paragraph and the relevance of each word in the paragraph to each word in the question. We combine these two 'attention' matrices with the original paragraph embedding to yield a matrix, G, "where each column vector can be considered as the query-aware representation of each context word."

The similarity matrix between h_q and h_p is determined by:

$$S_{ab} = w_{(s)}^T [h_{p,u}; h_{q,v}; h_{p,u} \circ h_{q,v}] \tag{4}$$

where $h_{p,u}$ is the hidden state of the u-th context paragraph word, and $h_{q,v}$ is the hidden state of the v-th question word. Context-to-query attention represents the question words that are the most similar to each word in the context paragraph. The attention weights on the question words by the u-th context word is computed by:

$$a_u = \text{softmax}(S_u) \tag{5}$$

and the attended question representation is computed by:

$$\widetilde{h}_{q,u} = \sum_i a_{t,i} h_{q,i} \tag{6}$$

We then compute query-to-context attention, which represents the context paragraph words that are the most similar to each word in the question. The attention weights are computed by:

$$b = \text{softmax}(\text{max}_{\text{col}}(S)) \tag{7}$$

and the attended context paragraph representation is computed by:

$$\widetilde{h}_{p,u} = \sum_u b_u h_{p,u} \tag{8}$$

Then we tile $\widetilde{h}_{p,u}$ U times across the column such that its shape matches that of $\widetilde{h}_q$, giving us $\widetilde{h}_p$. Putting all the contextual embeddings and attention vectors together, we end up with G. Each column vector in G represents the question-aware representation of the context paragraph word. G is defined as:

$$G_t = [h_p; \widetilde{h}_q; h_p \circ \widetilde{h}_q; h_p \circ \widetilde{h}_p] \tag{9}$$

G is passed to a two-layer BiLSTM, with the outputs of each direction concatenated, yielding M. Finally, we run G and M through a linear layer and softmax to obtain probability distributions on the start word of the answer:

$$\text{start}_{predicted} = \text{softmax}(w_{\text{start}}^T [G; M]) \tag{10}$$

We run another BiLSTM on M to get N, which is used to calculate the probability distribution of the end word of the answer:

$$\text{end}_{predicted} = \text{softmax}(w_{\text{end}}^T [G; N]) \tag{11}$$

See Figure 1 for a diagram of the model.

3.4 Baseline, V2

On the last day of the project, we reimplemented our baseline model and achieved much better results. Our second baseline model again uses a bidirectional LSTM to encode the question. We concatenate the final output states for the forward and backward passes, which we call q. We encode the context paragraph with a bidirectional LSTM, concatenating each of the output states for the forward and backward passes, which we call h_p. During the decoding step, in order to determine the answer start index prediction, we dot q with each column vector in h_p and run softmax across the dot products. We then use a bidirectional LSTM on h_p and concatenate each of the output states for the forward and backward passes, which we call h_a. Then in order to determine the answer end index prediction, we dot q with each column vector in h_a and run softmax across the dot products. In future work, this model would definitely be our starting point.

3.5 Additional Improvements

3.5.1 Gradient Clipping

We also make use of gradient clipping, which is a common technique in recurrent neural networks because, as gradients are being propagated back in time, they they can "explode" or become exponentially large when they they are continuously multiplied by numbers greater than one. Clipping these gradients between two manually-set values accounts for this problem.

3.5.2 Adam Optimizer

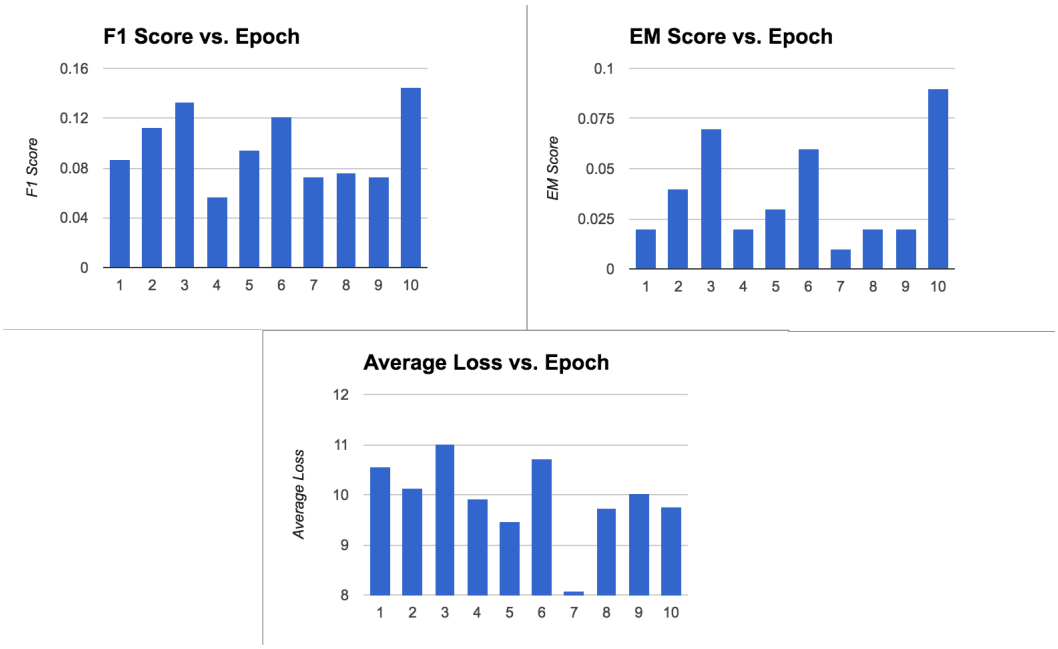
For our gradient descent optimization algorithm, we use Adam, a method for efficient stochastic optimization that only requires first-order gradients. With little memory requirement, the method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients. The method was designed to combine the advantages of two other popular methods: AdaGrad, which works well with sparse gradients, and RMSProp, which works well in on-line and non-stationary settings. Some of the advantages of the Adam method are that the magnitudes of parameter updates are invariant to rescaling of the gradient, its step sizes are approximately bounded by the step size hyperparameter, it does not require a stationary objective, it works with sparse gradients, and it naturally performs a form of step size annealing.

4 Results and Discussion

Here we discuss the results of our main models.

4.1 F1 and Exact Match Scores

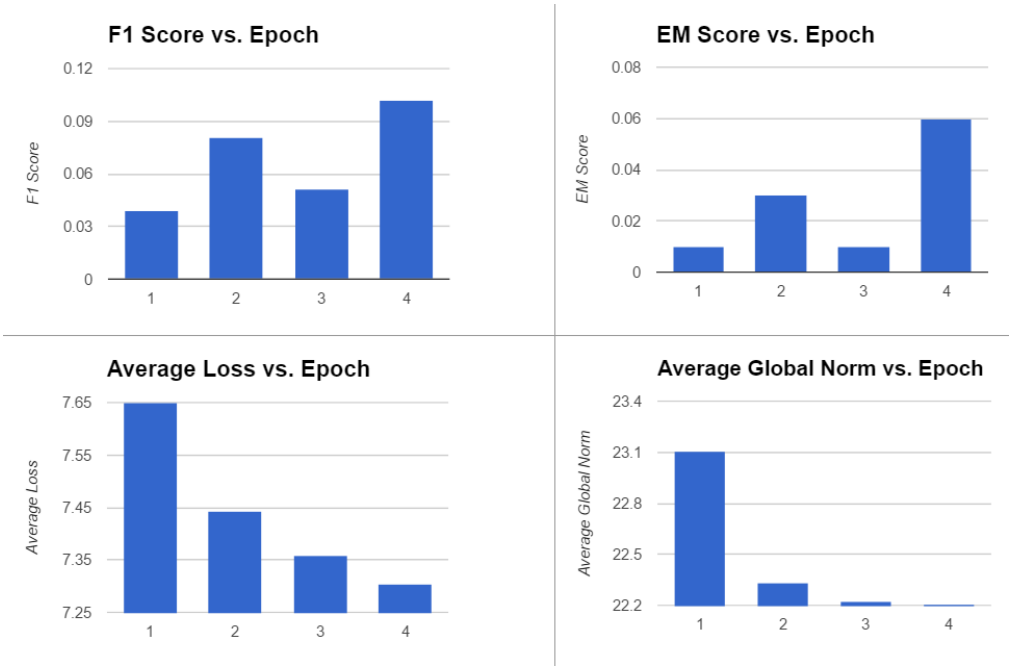
4.1.1 Baseline, V1



Our baseline model was very simple and as expected performed poorly. The plots above show the F1 and EM score by epoch against the *training* set. The final F1 and EM scores on the *validation* set were .101 and .040 respectively. The average training loss trends downward, but slowly and inconsistently.

We originally believed that this model was too simple to learn anything. But given the comparative success of our even-simpler baseline V2 model (see below) we believe that a bug in our code prevented this model from learning successfully. This first model contains enough complexity that it should be able to over-fit enough to get much higher scores on the training set.

4.1.2 BiDAF



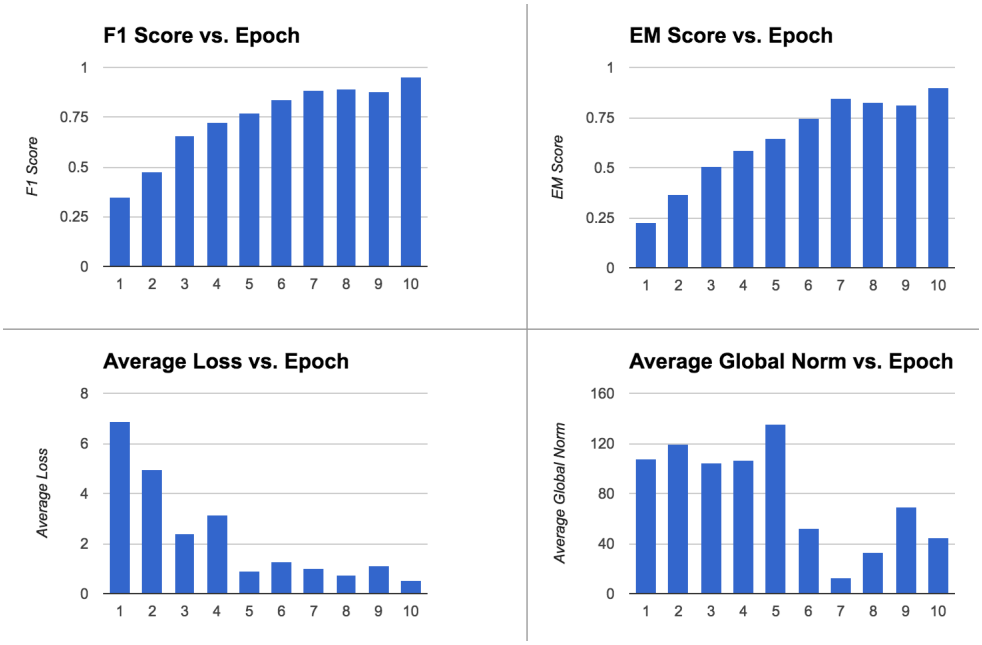
Unfortunately, our BiDAF model did not perform very well. Its final F1 and EM scores for the *training* set were respectively about 10% and 6% (shown in the plots above.) The final *validation* F1 and EM scores were approximately 7.6% and 3.7%. These results are even worse than for our original baseline model. The plots above show statistics gathered over four training epochs. Although very slow, we see our model's F1 and EM scores increase, while its average loss and global norm decrease.

The slow progress of our F1 and EM scores may be partially attributable to the low learning rate in these initial epochs. In our model, we used a constant learning rate of 0.0001. While a low learning rate is optimal for later epochs, we should have used a higher learning rate early on, especially for the first epoch, since the model doesn't have to be concerned about fine-grained learning in the earliest stages. Ideally, we would have used an exponentially decaying learning rate, though we would have had to spend a good amount of time experimenting with the initial value and the decay rate.

Our model's slow progress was also partially caused by the sheer number of parameters that it had to learn. Because we wanted to use word vectors that effectively capture semantic and syntactic information, we used trainable 300d GloVe vectors. While they sounded promising, they also contributed greatly to the 41m parameters that our model had to learn.

We only trained for four epochs due to time constraints; each epoch took nearly 11 hours. Aside from our F1/EM scores, our decreasing loss and global norm indicate that our model was learning in the right direction. We would expect these metrics to be very small once the optimal set of parameters for the model is found at the end of its training. However, given the concave F1 and EM curves, we suspect that the scores will not go much higher. This suggests a fundamental flaw in our model or a bug in our code (as we explain later, we suspect the latter.) A reasonable model with tens of millions of parameters should be able to over-fit on the training data and get a much higher training accuracy.

4.1.3 Baseline, V2



Our second baseline model, finally implemented at the very end of the project, performed much better. F1 score vs epoch shows the progression of the F1 score on the *training* set. It goes up to over 90% after 10 epochs, demonstrating that the model is successfully learning (albeit overfitting). The same situation holds for our EM score.

On the *validation* set (the more legitimate test of the model), our final F1 score after 10 epochs was 31.5%. The EM score was 21.9%. While these results are not impressive in an absolute sense, they are a dramatic improvement over our earlier performance. We also see average loss going to zero (again demonstrating that the model is learning) and the global norm trending downward (suggesting that the model is stabilizing.)

4.2 Time Performance

Time was a major obstacle for us in training, improving, and testing our models. Our BiDAF model took over about 10.5 hours per epoch to train on the GPU, which made it extremely difficult to test incremental improvements or even tune hyperparameters.

With that said, the greatest limiting factor of our experimentation was out-of-memory errors. Using a batch size greater than four on our over-forty-million-parameter model led to out-of-memory errors that could appear hours into training. In hindsight, we should have used smaller GloVe vectors or limited the context paragraph size to only a few hundred, to be able to run much larger batches.

We tried setting "swap_memory=true" in our bidirectional dynamic RNNs, but this did not have any noticeable impact on memory efficiency. We also carefully went through our code looking for explicit for-loops that might have been using too much memory or calculating tensors inefficiently, but we didn't find any red flags.

The attention flow layer at the heart of the BiDAF model also added a significant amount of time to training. Given that it used the 200-dimensional hidden states of each word in the question and context paragraph to create the attention layer's similarity matrix, this layer was very computation intensive.

5 Future Work

The first step for future work on our model is to identify the bugs in our code that are causing the model to run so slowly and to be unable to accommodate a batch-size greater than four. The next step is to identify the bugs that are causing our model to obtain such low scores. We believe that our model design is reasonable and are confident that there is some issue in our code, likely our use of the TensorFlow API, that is causing poor results, rather than our conceptual understanding.

Our code is structured into a core QAModel class, and all of our models inherit from this class. The day before this paper was submitted, we believed that the issue was in our QAModel superclass or its support code. This was because all the models we had tried were failing. But now that we finally have a reasonably working baseline (which also subclasses QAModel) we believe it is more likely that the problem lies in our specific model implementations.

We now describe some specific additions and modifications that we would like to make to our model.

5.1 Character Embeddings

Like word embeddings, character embeddings map words to a high-dimensional vector space as they seek to capture the linguistic subtleties behind how those words are used. An advantage to using character embeddings is that unexpected character combinations, such as misspellings, in a dataset may be better learned. We would like to add a character-embedding layer to our BiDAF model, using convolutional neural networks. Notably, the original paper off of which we based our model included a character embedding layer; we initially included one but dropped it due to memory constraints.

5.2 Ensemble Training

At a very high level, our task is to search through a hypothesis space to find a suitable hypothesis that will make good predictions given our training dataset. However, even if the hypothesis space contains hypotheses that are very well-suited for a particular problem, finding a good one is a very complex task. The use of ensembles allows us to combine multiple hypotheses with the hope of forming a better one. While a trained ensemble represents a single hypothesis, it is not necessarily contained within the hypothesis space of the models from which it was built. One advantage of using ensembles thus lies with its flexibility in the functions that they can represent. In addition, ensembles tend to give better results when there is a significant diversity among the models, so many ensemble methods seek to promote diversity among the models that they combine. We would need to implement some additional models to create an ensemble, and we are very interested in seeing whether the ensemble performs better than its individual component models.

5.3 Hyperparameter Tuning

Hyperparameter optimization or model selection seeks to optimize a measure of an algorithm's performance on any given dataset by choosing a set of hyperparameters for a learning algorithm. This concept is distinct from actual learning problems, which instead optimize a loss function on a training set only.

One hyperparameter optimization we would like to implement is grid search — also known as parameter sweeping — which simply searches through a manually-specified set of values within the hyperparameter space of a learning algorithm. Because a grid search algorithm must be guided by some performance metric, we would use internal cross validation on the training set. More specifically, given a lot of computing power and time, we would look into optimizing values of parameters such as learning rate, LSTM hidden state size, context paragraph cutoff limit, dropout rate, etc. We would also like to investigate whether Rectified Linear Unit (ReLU) activation functions perform better than the tanh functions that we use in our recurrent neural networks.

It should be noted that we intended to use grid search to optimize our BiDAF model, but this was impractical given the slow speed of the model. Once we tackle the speed problem, we will implement rigorous hyperparameter optimization.

6 Conclusion

We implemented a Bidirectional Attention Flow model. While our implementation of the model was not ultimately very successful, we gained some valuable experience and lessons from this project. We learned that neural models are extremely difficult to debug. The model parameters tell a human observer very little about the state of the learning process, and it can take a long time to determine whether the initial performance of a model is accurate. Subtle errors can result in out-of-memory crashes hours into the program, and even in an ideal situation a good model can take hours or days to train.

At the very tail-end of the project we implemented a simple baseline model that obtained much better results than our original baseline or our BiDAF model. This makes as much more hopeful about our potential for future progress on this project.

A lesson that follows from this last-minute turnaround is that even a very simple model in this problem domain should be able to do much better than random chance against a validation set and should be able to do extremely well against a training set (by overfitting.) When our original baseline gave little-better-than-random results on the training set, we assumed that the problem was the simplicity of our model. We attempted to build more sophisticated models to address the problem, but this did not fix the issue. In retrospect, we should have realized that our original baseline was complex enough to overfit to the training data, and we should have tenaciously debugged it until it was able to do so.

With a fully-functional baseline model and a much more sophisticated BiDAF model that just needs a little debugging, we are very optimistic about our future prospects with this project.

7 Contributions

All group members collaborated extensively on all parts of the project. We spent dozens of hours in a room together working on this project, and we all contributed extensively to the design, coding, and debugging processes.

With that said, Genki definitely gets a majority of the coding credit. He wrote most of the model code, and the much-better-performing baseline version two was his work. Daniel and Ramon did most of the work on the paper. Ramon designed the poster.

References

Seo, et al. "Bidirectional Attention Flow for Machine Comprehension." *University of Washington.* 2017. https://arxiv.org/pdf/1611.01603.pdf

Yu, et al. "End-to-End Answer Chunk Extraction and Ranking for Reading Comprehension." *IBM Watson.* 2016. https://arxiv.org/pdf/1610.09996.pdf

Pennington, et al. "Glove: Global Vectors for Word Representation." *Stanford University.* 2014. https://nlp.stanford.edu/pubs/glove.pdf

Srivastava, et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *University of Toronto.* 2014. https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

Duchi, et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." *University of California, Berkeley.* 2011. https://stanford.edu/~jduchi/projects/DuchiHaSi10_colt.pdf

Tieleman, et al. "Lecture 6.5-RMSProp, COURSERA: Neural networks for machine learning." *University of Toronto.* 2012.

Kim, Yoon. "Convolutional Neural Networks for Sentence Classification." *New York University.* 2014. https://arxiv.org/pdf/1408.5882.pdf

Srivastava, et al. "Highway Networks." *The Swiss AI Lab IDSIA.* 2015. https://arxiv.org/pdf/1505.00387.pdf