CS224N Default Final Project Write-Up

Mark Holmstrom
Department of Computer Science
Stanford University
Stanford, CA 94305
markholm@stanford.edu

Abstract

This projection is an exploration of the Stanford Question Answering Data set (SQuAD) and the process of solving question answers problems. The goal of the project was to observe how different pieces of information and structural changes could boost the performance of a baseline classifier provided by the instructional staff. This project focused especially on modularity: being able to piece together different parts together to make a classifier combining different improvements.

1 Algorithm Modifications

One of the goals of the project was to build a more complicated a stronger question answering neural network out of the given baseline. This section describes the baseline and the improvements made.

1.1 Baseline

The baseline question answering algorithm functions as follows. It uses pre-trained GloVe word embeddings to get an embedding vector for each word in the context paragraph and the question. From there, a 1-layer bi-directional GRU with shared parameters is used to get a forward and backward hidden vector for each word in the context and each word in the question. Note that the GRU is used twice, once for the context and once for the question. We concatenate these hidden vectors to make one hidden vector for each word. From there, we use a dot product attention layer with the context combined hidden states attending to the question combined hidden states and concatenate the results to the context hidden states. These blended representations are then sent through an output ReLU layer to get a raw output vector for each hidden state. These raw outputs are put through two linear transformations to turn each into a single unweighted probability approximation for the probability of the question answer start being at that location and the question answer end being at that location. We then put the question answer start and question answer end approximations through a softmax layer to get the final approximation for the start and end probability distributions. We calculate our loss to be the sum of the negative logs of the approximate probability of the start and end being at their true respective locations. For making predictions on inputs, we independently predict the start and end positions by choosing the argmax of these approximated probability distributions.

1.2 Character-Level CNN

One common recent way to improving algorithm performance has been using a character-level CNN to to enrich our embedding. My implementation works as follows: split the input into its individual, and for each character in each word, pick an index for that character (in this case its ASCII encoding). From there, the character is given a trainable embedding based on its ASCII character. Word-by-word, we then pass the character-level embeddings into a single-layer 1D convolution neural network. As recommended, I used a max word length of 20, an embedding size of 20, window size of 5 and an output size of 100. For each word, the outputs from the CNN are combined using

element-wise max-pooling to end with a resultant vector that will used to represent the word. The vector representing the word is then added onto the input for the main RNN layer.

1.3 Bidirection Attention

Dot-product attention is fairly simple and the baseline can be improved upon using a more complex form of attention. One such option is bidirectional attention. The idea is that the context does not only attend to the question but the question also attends to the context. The way this works in math is that we start by computing a similarity score between each pair of hidden states taking one from the question and another from the context. We compute the similarity by taking 3 different weight vectors, taking a dot product of the first with the context state, a dot product of the second with the question state, and taking a dot product of the third with the element-wise product of the two states. We then add up all of these dot products to get the similarity score. We then separately calculate the context to question and question to context attentions. For context to question, we take the vector of similarity scores for each context word compared to all of the question words and preform softmax to normalize the values. We use the result of this softmax to make an attention which is the expected hidden question state if we pick it using the results of the softmax as a probability distribution over the possible question states. We do something a bit different for the question to context layer. For the question to context layer, we choose the maximal similarity between a specific word in the context and any word in the question and take the softmax of these maximums. We then use this softmax as a probability distribution and find the expected hidden context state from this distribution. From there, we combine context to question and question to context layers together by making a large vector concatenating everything together. For each context hidden state, we take the original hidden state, the context to question attention, the element-wise product between the two, and the element-wise product between the context hidden state and the question to context attention, and concatenate all 4 of these vectors together to get the attention output.

1.4 Additional Input Features

One way of improving algorithm performance is increasing the number of input features. The baseline only has the word embedding but much more can be used. One option is the character-level embedding mentioned above. There are a number of other smaller additions that can provide useful information. Two were added to this model. This first is an indicator of whether or not a given word appears in the other half of the query. If we are forming a context embedding, we see if the given word is in the question. If the word is in the question we have a 1 and if it is not we have a 0. We do the the same thing for the question embedding: if the word is the context, we give it a 1 and if it is not we give it a 0. The second piece of additional information is another attention layer. The idea here is that by adding attention to the word embeddings we may get some information about their relation to each other. We make an attention layer with the same structure as in the baseline and have the context embeddings attend to the question embeddings. Additionally, we have an attention layer that goes in the reverse direction: having the question embeddings attend to the context embeddings. We append these new attention layers and matching features to the word embeddings before sending them into the GRU from the baseline.

1.5 More Nuanced Prediction

Our baseline model predicts the start and end location of the answer as independent of each other. This is clearly unrealistic since end location must be after the start location and we also usually expect the answer to be short, so the position are likely close together. Given this information, a new prediction model was implemented that still considers the start and end probability distributions to be independent, but also requires that the end position be between the start position and 15 words afterward.

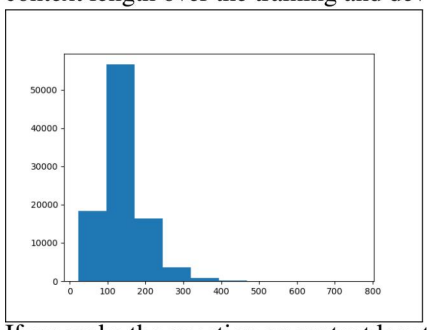
2 Experiments and Analysis

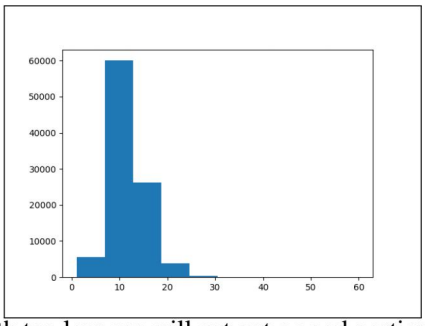
A number of experiments were performed to analyze the effectiveness of the different upgrades to the baseline as well as take a look at how changing the parameters effects the results. Many of these experiments were performed around an elevated baseline model. This model performs the

baseline with the added functionality of having the additional input features mentioned above. This was the quickest easiest way I found of improving the performance of the baseline so much of my experimentation was built around it.

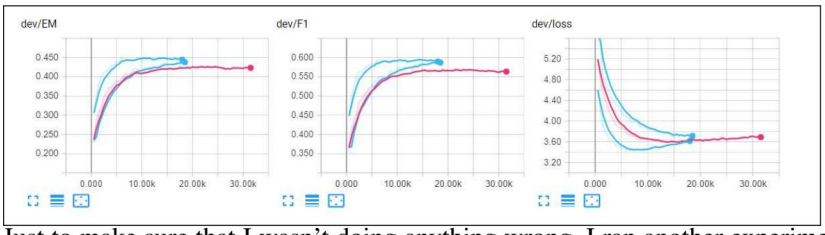
2.1 Managing The Weakness of Bidirectional Attention

One of the big issues with bidirectional attention (and why the baseline with extra features ended up working out much better) is that the the amount of memory needed to compute this attention is quite large. Specifically, we need to do operations on multiple 4-dimensional tensors of size (batch size-by-max context length-by max question length-by-2$*$hidden length) in order to properly compute the similarity. Due to this large tax on memory, the bidirectional attention model was forced to being smaller than the original since the GPU would literally run out of memory if I tried running it with similar parameters. In order to reduce the size of this step in particular, we clearly needed to reduce the size of these 4-dimensional tensors. That means that we needed sacrifice some of these values in order to get the model to run on this machine. This ended up making it weaker than the baseline when both utilize most of the other improvements. In particular, I found that it is risky to sacrifice the question length and context length too much since we end up losing important information when going in for predictions. The two figures below are histograms of the question and context length over the training and dev sets.

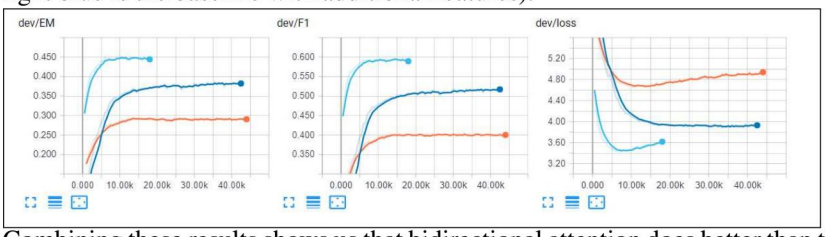




If we make the question or context length too low, we will cut out a good portion of important information from a sizable fraction of the examples and this will greatly reduce our performance on such examples. In order to remedy this, I initially decided on a minimum question and context length that I was comfortable with, which are the default question length of 30 and a lower than default context length of 250. This does cut off the ends of some contexts and questions, but the percentage which do get cut off is quite small compared to having reduced performance due to having small hidden size. I ended up deciding later that 250 might be too low and increased the minimum to 400. Even with these decreased max lengths, I still had to cut the batch size and hidden size quite a bit. I ran two experiments with the bidirectional attention model. One was aimed at seeing how well it could do compared to our improved baseline. For this one, I decided that I could cut the batch size down and try to keep the hidden size as large as possible. The reason for this is that in theory, decreasing the batch size should not hinder the models overall performance. It does make the training rockier: there are more wild fluctuations between individual iterations and it takes more iterations to converge, but after training for long enough, it should give us the same performance as running with a larger batch size. I ended up with having the model run-able on the GPU when the context length is 400, the hidden size is 180 and the batch size is 20 and when the contexg length is 250, the batch size is 50, and the hidden size is 150. I also had the additional bonus of running with the additional features added in. With running this way, as you can see, it runs just about as well as the the baseline but not any better with context length 400, and slightly worse with context length 250. (Here is the plot, maroon is bidirection attention with 250 context length, the upper light blue plot line is the baseline, and the light blue line between the two is bidirectional attention with 400 context length).



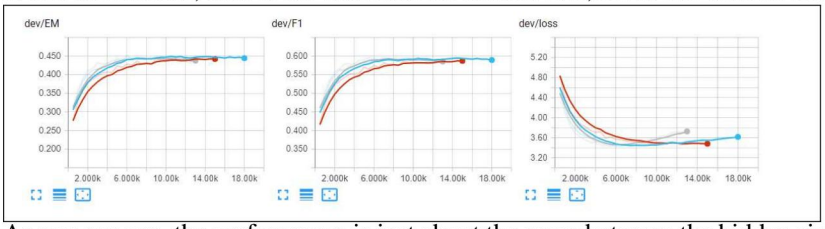
Just to make sure that I wasn't doing anything wrong, I ran another experiment trying out the bidirectional model with no other improvements and compared it to the baseline. As you can below, it did quite better than the baseline, but not better than the baseline with the additional features. (Here is the plot, orange is the baseline, dark blue is the bidirectional attention without additional features, light blue is the baseline with additional features).



Combining these results shows us that bidirectional attention does better than the baseline, but somehow adding in the additional features makes the baseline perform just as well. If we combine this with the fact that the bidirectional attention model forces us to have limited context and batch sizes, I found it better to use the baseline with additional featues as a standard moving forward.s

2.2 Experiments on Hidden Size

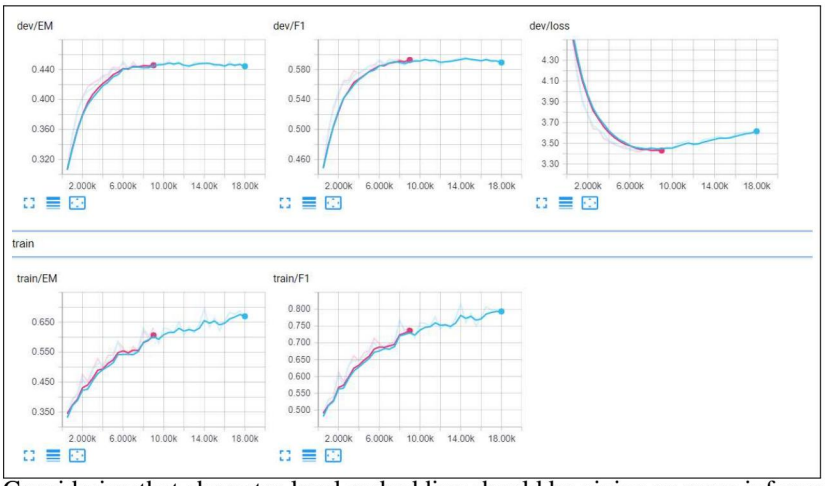
I also did experiments on the hidden size to see if increasing or decreasing it would change how much we get out of the model. This hidden size experiments were done by changing the hidden size with the baseline model and the baseline model with added additional parameters. Here are the results for the baseline model with additional features using the length 100 word embeddings: (grey is 300 hidden size, blue is the standard 200 hidden size, red is 100 hidden size).



As you can see, the performance is just about the same between the hidden sizes of 200 and 300, but somewhat less for a hidden size of 100. This structure implies that there is a sort of saturation point where the hidden size is large enough. At that point, having a larger hidden size doesn't necessarily help anything since there isn't any more information to be gained from the inputs using this GRU format. However, having a hidden size less than this saturating amount will cause it perform worse, but you need a substantially smaller hidden size to notice a large difference in the accuracy of the model between the two hidden sizes. As the graph shows, having half of the hidden size does negatively effect the model performance, but not by that much.

2.3 Character-Level CNN

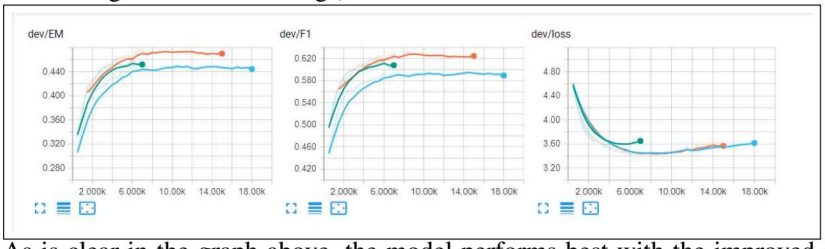
There definitely is a problem with my character-level CNN since it is not performing better than the baseline. I ran the model with the same parameters as our standard baseline with extended features model, and also gave it access to the extended featues and here is the resulting performance: (blue is the improve baseline model, and maroon is the same model with the Character-level CNN features added)



Considering that character-level embedding should be giving us more information, adding this layer in should be increasing the effectiveness of our model, but instead it has almost the exact same performance as before on dev set and only very slightly increased performance on the training set. There are a couple of possible explanations as to why this could be happening. Most likely, this has something to do with the fact that the character-level CNN I implemented was a single layer one. Papers about using such a character-level CNN usually emphasize the need for the model to be deep to work effectively (in the one I read they used 9 layers!). It is possible that my character-level CNN is not nearly deep enough to have a substantial effect on the results. Additionally, one big problem could also be the padding. Padding of 0's was added in after the character indices for each word of length less than the maximum. It is possible that this padding could mess with the output of the CNN in a way that isn't taken care of later by the masking done after the RNN layer.

2.4 Pushing Toward a Better Fl/EM Score

With a decent standard $F1/EM$ maximum score of about 0.59 and 0.44 for the baseline with extended features (not including the character-level CNN), I tried a few different ways to improve this score using a few extra tricks. First, I tried adding in the more strict requirements on choosing the predicted answer start and end positions. This increased the effectiveness of the model quite substantially, increasing its performance by 4% or so, increasing the score in both the F1 and EM categories. It seems that adding this enhanced choosing strategy will unsurprisingly not change the loss, but will always positively influence the F1/EM score. Additionally, I tried using a larger embedding vector size, which normally is 100, but I increased to 200. Here are the results: (blue is without the improved estimation choice, orange is with the improved estimation choice, and green is with larger word embeddings).



As is clear in the graph above, the model performs best with the improved estimation choice and the smaller embedding vector size of 100. It is possible that choosing an embedding vector size of 200 causes the model to overfit on the training data due to the increase in parameter size in the main GRU layer and actually perform slightly worse.

3 Further Work

There are many ways in which the algorithm can further be improved. Clearly what was done here is just a small number of the viable option for improving on the given baseline algorithm. Based on the work that was already done, there are a number of different ways in which the algorithm could

be further improved.
There is more work that could be done in terms of the character-level CNN. It is a tried a true method for NLP and the results it offered me did leave much to be desired. The success of the character-level CNN is often tied to having a CNN with many layers, a large data set, and a large amount of time to train the CNN. Possibly working with a deeper CNN and a better way of masking out placeholder 0's will improve the overall effectiveness of this model.
Additionally, there were no changes made to main the internal structure of the model, the bidirectional GRU. Bidirectional GRUs work well enough, but it is likely that other structures could work better for this task. It would have been nice to try a LSTM or a model with more layers to attempt to improve accuracy.
There is more than can be down with the prediction method. Outside of the strict positioning requirements, the improved prediction method still assume that the start and end positions are independent and this is certainly not the case. Having a more complex prediction method that takes their dependence into consideration would likely boost performance even more.
Lastly, there are a number of prediction methods out there, some of which are mentioned in the final project handout which work from a completely different structure than the baseline. It would have been a challenge to try to fit the modules I created into a complete new method and see how the completely new method fares.

References

[1]CS224N Staff (2018) Default Final Project Handout.

[2] Xiang Zhang, Junbo Zhao, Yann LeCun (2015) *Learning character-level Convolutional Networks for Text Classification* NIPS 2015.

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

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