Exploring How Differences in Embeddings Change BiDAF Performance

Stanford CS224N Default Project, IID Track

Theo Culhane
Department of Computer Science
Stanford University
tculhane@stanford.edu

Abstract

The SQuAD dataset is very well studied, and at this point the models that are scoring the best are consistently transformer-based models that finetune massive pretrained models like BERT. However, the default final project starter code revolves around starting with an LSTM based model, BiDAF [1], in order to solve the SQuAD problem, and so the results that can be achieved for the SQuAD track of the final project likely cannot come even close to state of the art, since the scope of a final project is much too small to compete with a pretrained model. Therefore, I am focusing on different ways to manipulate the embeddings generated by the input data, which should give model- independent results about how to better solve the problem. I focused on augmenting the existing GloVe embeddings given to us with different ways of using character and morpheme level embeddings. I found that while adding both types of embeddings generated improvements over the baseline model that used neither, adding in subword embeddings tended to allow the model to overfit on the training set, limiting the improvement realized while using such embeddings. Therefore, in order to maximize performance, it seems that augmenting the data with only character level embeddings is the most effective strategy.

1 Key Information to include

- Mentor: Vincent Li
- External Collaborators (if you have any): None
- Sharing project: No

2 Introduction

The SQuAD 2.0 dataset [2] is a question answering dataset comprised of pairs of a paragraph (sometimes called context) and a question (sometimes called a query) pertaining to that paragraph (essentially a reading comprehension question, similar to what one might see on the SAT) that came out in 2018, and is an extension of the previous SQuAD dataset, adding in some paragraph-question pairs where the question cannot be answered in the paragraph, which forces the model to say when a question cannot be answered instead of just guessing. As mentioned in the Abstract, SQuAD is an extremely well studied dataset because it is very robust, and so the set of models that perform well on the set is very well known. The models that do the best are consistently transformer based models that are backed by massive pretrained models, like T5, that are too complex and require too many resources to properly train to be in scope for a final project for this class. In addition, the starter code for this project used an LSTM based model called BiDAF, making the leap to using a more state-of-the-art model even less attainable. I therefore decided to focus on exploring how augmentations to the data the model takes as input impacts the performance, which should give more

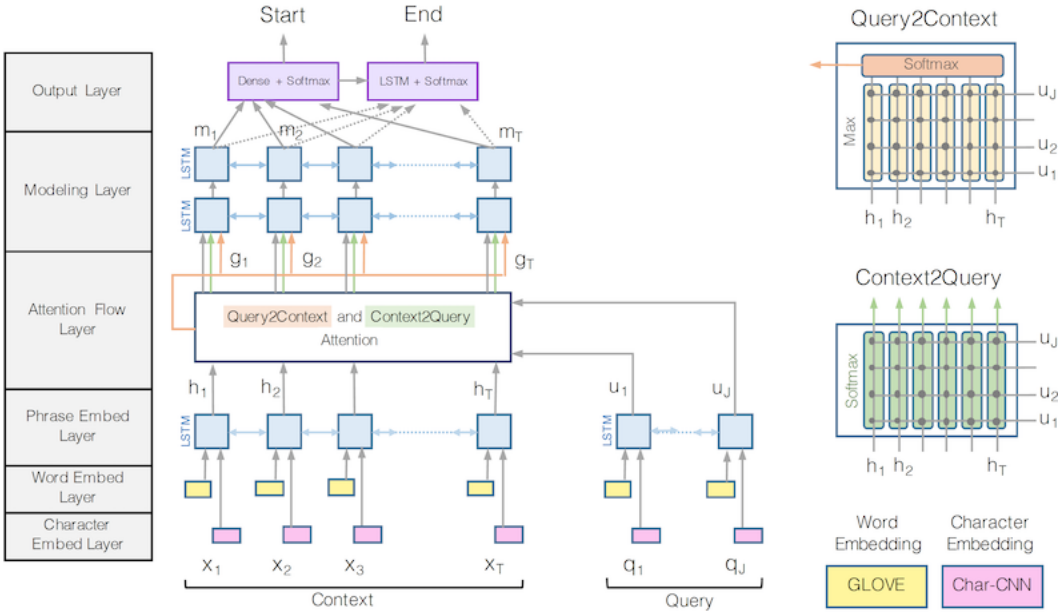
model-independent insights. The most intuitive augmentation to me was appending on morpheme level embeddings to the word embeddings we are given, since that helps with reducing the number of out-of-vocabulary words the model comes across, since morpheme level embeddings should help it guess more accurately about the meaning of those words, and because the primary paper I was basing my model on had only dealt with character level embeddings and so I was curious how morpheme level embeddings would compare. I ended up finding that using character embeddings alone, similar to the paper I was basing my work on used, performed the best, as morpheme level embeddings proved to be flexible enough to allow overfitting, decreasing performance on the dev and test sets.

3 Related Work

As stated earlier, the majority of recent work on SQuAD has come in the form of work on transformer based models, which is worth briefly acknowledging. However, since those models are so distinct from the method I used, it is worth only the brief acknowledgment, and then directing the reader to run a search for themselves if they want more information. The particular work that I most heavily based my own on was the original BiDAF paper [1], and a paper that they reference by Kim [3], which explains much of how and why CNNs might be useful in the realm of manipulating embeddings. Much of the way that BiDAF works is covered in the project assignment sheet located at https://web.stanford.edu/class/cs224n/project/default-final-project-handout-squad-track.pdf, and so in the interest of space I will refer the reader there or to the original paper for more in depth information about how the system works and what modifications to the system the default starter code makes. One thing worth noting about the original BiDAF paper, though, is that it makes no mention of how they decided on the particulars character embedding model that they chose, and more specifically doesn't reference if any other embedding augmentations, such as subword level embeddings, were considered. Similarly, the Kim paper made no reference to whether they considered subword level embeddings, which seemed strange to me, because it seems like the work that the CNN model they used is trying to do is combine characters into subwords, and then combine those subwords into words, in such a way that the meaning of the word can be inferred by only looking at the characters.

4 Approach

The model used in the original paper is diagrammed below:



(Image taken from the original paper [1] and then resized to fit in this report)
This model was recreated in the starter code we were given, with the only modification being the absence of character level embeddings in the starter code, and with the exception of the word embed layer and the character embed layer, I used the model implemented in the starter code without modification. Since I didn't make significant changes to the other parts of the model, I will only

briefly discuss each of those parts at a high level, and will point the reader to the project handout or the original paper should you want more details. The phrase embed layer is a bidirectional LSTM encoder that operates on the output of the embedding layers (which I will discuss in greater detail later) in order to incorporate information about the meaning of the sentence and the ways the words of the sentence interact with one another.

The attention layer is the core improvement that BiDAF made over previous models, and uses attention both from the query to the context, and from the context to the query, hence the name "bidirectional attention flow". The context to question attention is slightly simpler, in which we take the softmax of the rows of a matrix of the similarities between context and question hidden states (the output of the phrase embed layer), and then these softmaxs are multiplied with the question hidden states to generate attended versions of the question hidden states. In the question to context attention, we first calculate the softmaxs along the columns of the matrix, and then multiply those softmaxes against the softmaxes of the rows that we obtained earlier. We then multiply the context hidden states against the resulting matrix, giving us the attended versions of our context hidden states. Finally, we concatenate our original context hidden states, our attended question hidden states, element wise multiplications of our context hidden states with the attended question hidden states, and element wise multiplications of our context hidden states with the attended context hidden states.

Next, our modeling layer takes the output of the previous layer, and runs it through two layers of bidirectional LSTMs. This allows the model as a whole to incorporate temporal information (word relationships and the meaning of the sentence) about the attended versions of the context and question representations.

Finally, we have our output layer. The output layer generates two vectors: one for the probability that the answer starts at a particular index in the context, and one for the probability that the answer ends at a particular index in the context. If the model wants to predict that the question doesn't have an answer in the context, it simply predicts a start and end index of 0 (this is a slight oversimplification, since it is is predicting probabilities and not indexes directly, but it nevertheless works adequately for this explanation). For the start indexes, the model simply uses a densely connected layer from the output of the modeling layer. For the end indexes, the model uses a two layer LSTM, followed by a densely connected layer, which helps the model take what the starting index is likely to be into account when trying to predict the end index.

The embedding layers, which are essentially the input layers, are where most of my changes and experimentation were, so I will now discuss those. The baseline model accepted as input a vector of GloVe word embeddings. My first set of experiments revolved around adding in character embeddings, for which I tried two different models (both of which started with randomly generated vectors of length 64 for the character embedding, and both of which allowed the embeddings to be trained as the model learned). In both cases, the character summary embedding generated was then concatenated onto the word embedding, similar to the BiDAF model in the original paper. For the first model, I simply took each of the character embeddings for a particular word, then concatenated them, and then used a 1 dimensional CNN to generate a 100-dimensional vector summary of the sequence. For a more concrete example, consider a word with 5 characters. In this case, we would generate the 64 digit embedding for each character, and then concatenate all of them into a 320 dimensional vector. We then run a CNN over this vector, generating a 100-dimensional summary vector. For the second model, I instead stacked the vectors, used a CNN to generate multiple 100 dimensional summaries along the first axis of the stacked vectors, and then use max pooling to turn this in a single 100 dimensional representation.

After the character level embeddings, I also tried out several models to generate subword level summary embeddings, which I then also concatenated onto the word embeddings. In all versions, I used a list I found on the internet of English morphemes [4], and I started out with randomly generated vectors of length 64 for each subword embedding, and allowed the embeddings to be trained as the model learned. In the first two models I tried, I simply concatenated the subword embeddings generated for each word onto the word embeddings and character level embeddings, and so the first two models are only different in how they handle character level embeddings (there is one corresponding to each of the character level models tried). First the rest of the models, I only used the second character embedding model. In the third subword model, I used a CNN architecture identical to the second character embedding architecture, in which I stacked the subword embeddings generated for a word, and then used a CNN and max pooling layer to turn the stacked vectors into a single 100 dimensional summary. I tried this model both with and without the character level summary also appended. In my final model, after generating the 100 dimensional summaries for both the character level and subword level embeddings, I used a 2 layer multi-layer perceptron to turn the

concatenated 200 dimensional summary into a single 100 dimensional summary. In this, the hidden layer was 150 dimensional, and the activation function used was ReLU.
As the final part of this layer I experimented with, I ran a few experiments in which a trainable version of the word embeddings were concatenated onto the frozen word embeddings, such that the word embeddings were 600 dimensional instead of 300 dimensional. The initialization values for the trainable word embeddings were the word embeddings given to us, but I hoped that the model might gain some domain knowledge by having trainable versions of the embeddings.
For the baseline model, I simply used the baseline given to us in the starter code, so should the reader have any deeper questions about the baseline I am comparing against, I would refer you to the project handout at https://web.stanford.edu/class/cs224n/project/default-final-project-handout-squad-track.pdf.

5 Experiments

5.1 Data

As described in the project handout, this project only uses the SQuAD 2.0 dataset, as adding in additional training data was not allowed. Within the SQuAD task, the model is given a question and a paragraph of context about the question, and must "highlight", or output the start and end indexes of, which part of the paragraph is the answer to the question, with an extra Out-Of-Vocab token appended at index 0 to facilitate the answering of unanswerable questions. For example, should the context be "The first African American President of the United States was Barack Obama, who was in office from 2008 to 2016", and the question was "Who was the President of the United States in 2008?", the model should output a start index of 11 (since we start with an extra token at index 0), and an end index of 12, referencing that the answer in the context is "Barack Obama" (with the model, as mentioned earlier, seeing the context as "<OOV> The first African American President of the United States was Barack Obama, who was in office from 2008 to 2016". Note that within the dataset, each training example is tagged by three separate humans, and the model is considered correct if it agrees with any of the three. So, if for example two taggers had said the answer is "Barack Obama", but one had said that the answer is instead simply "Obama", the model would be considered correct whether it predicted a start index of 11 and end index of 12 or a start index of 12 and an end index of 12. Around half of the questions within the dataset do not have answers in their corresponding context, in which case the model is expected to predict index 0 for both the start and end. For example, if the context is "The first African American President of the United States was Barack Obama, who was in office from 2008 to 2016", and the question was "Who was the President of the United States in 2020?", the model should predict a start and end index of 0, indicating that the model thinks the answer is "<OOV>", which is again a sentinel for the question not being answerable.

5.2 Evaluation method

The model is evaluated using a combination of two metrics, EM and F1. EM stands for exact match, and the EM score for a particular example is either 0 or 1, with the score being a 0 if the model did not output an exact match for the actual answer and the score being 1 if the model did output an exact match. For example, remember our Obama example from above. If the expected answer was "Barack Obama", and the model output "Barack Obama", the model would score a 1 on EM for that example. If the model instead predicted the answer as "Barack Obama, who was in office from 2008 to 2016", the model would receive a 0 for EM for that example. The EM score reported is percentage of examples in the given dataset for which the model scored a 1 for EM. The F1 score is the harmonic mean between precision and recall that the model scores on a particular example. Reusing our example from above, say the expected answer is "Barack Obama", but the model said the correct response was "Obama". This is 100% precision, and 50% recall, and so the model would score an F1 score for this example of 66.7%. If the model had said "Barack Obama", this would be an F1 score of 100% (perfect precision and recall), and if the model had said the answer was "Barack Obama, who", the precision is 66.7%, and the recall is 100%, giving an F1 score of $\approx 79.5\%$. For both EM and F1, for each example, the model scores the best on each metric that it scores out of the three correct answers. So, for example, had one tagger put "Obama" as the correct answer and two had put "Barack Obama", if the model answered either "Obama" or "Barack Obama", the model would score a 100% for both EM and F1 for this example.

5.3 Experimental details

For many of the tunable hyperparameters, I just used the defaults that the baseline uses, and so in the interest of space, I will not repeat those here, and will instead point the reader to the project handout. For subword level embeddings, I truncated the embedding produced by the model to 5 subwords maximum per word. While this may have caused the model to lose some information on particularly long words, it seemed like a large enough number to capture most of the words it would encounter. The list of subwords I used contained 5000 morphemes, some of which are very common English subwords, like the prefix "un", and some of which are slightly less common (for example, the list included the full word "american" as a subword). I ran the following experiments, the results of which are reported on below: Baseline model is exactly the baseline from the starter code. Character Embeddings Version 1 is the first character model described earlier, with no subword embeddings used at all. Character Embeddings Version 2 is the second character model described earlier, with no subword embeddings used. Subword Embeddings With Char V1 is the raw subword embeddings concatenated with the word embeddings and character embeddings, with the character embeddings using the Character Embeddings Version 1 model. Subword Embedding with Char V2 is the same, but with the Character Embeddings Version 2 model. Subword Embeddings Combined With CNN uses the subword embedding CNN summary vector model described earlier, concatenated with the Character Embeddings Version 2 model embeddings and the word emmbeddings. MLP Combination of Char and Subword uses the multilayer perceptron model described earlier. Subword without Character Embeddings uses the same model as Subword Embeddings Combined with CNN, but does not include the character embeddings, so just concatenates the CNN output on subword embeddings with the word embeddings. Unfrozen Word Embeddings With Subword is the same as Subword Embeddings Combined With CNN, but also concatenates onto the embedding the learned word embeddings, described above. Unfrozen Word Embeddings Without Subword is the same concept, except instead of being an extension of Subword Embeddings Combined With CNN, it is an extension of the Character Embeddings Version 2 model. Finally, Ensemble of Subword Models uses an ensemble of Subword Embeddings With Char V1, Subword Embedding With Char V2, Subword Embeddings Combined with CNN, MLP Combination of Char and Subword, and Subword without Character Embeddings, taking the average of the output of all of the models to make its prediction.

5.4 Results

On the dev set I found the following results:

Model	Dev NLL	Dev F1	Dev EM	Dev AvNA
Baseline	03.10	60.60	57.33	67.4
Character Embeddings Version 1	02.84	64.43	60.85	70.9
Character Embeddings Version 2	02.96	64.05	60.48	70.5
Subword Embeddings With Char V1	03.56	61.80	58.09	68.8
Subword Embedding With Char V2	03.23	63.49	59.62	70.0
Subword Embeddings Combined with CNN	03.18	62.99	59.50	69.4
MLP Combination of Char and Subword	02.91	62.71	59.44	68.9
Subword without Character Embeddings	03.25	62.36	58.75	69.0
Unfrozen Word Embeddings With Subword	03.32	60.96	57.62	68.2
Unfrozen Word Embeddings Without Subword	03.25	62.33	58.81	69.6
Ensemble of Subword Models	05.27	58.35	54.60	65.2

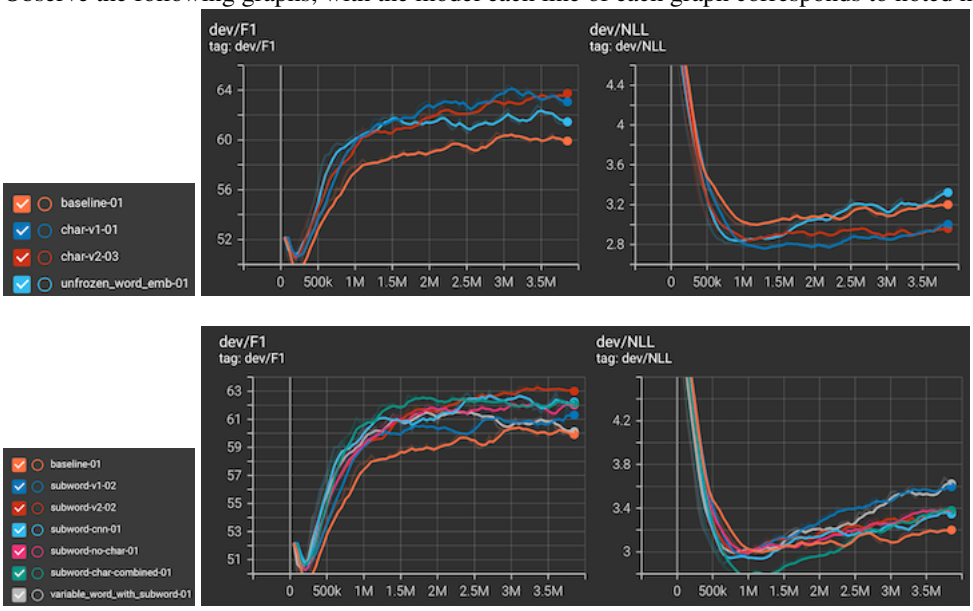
On the test set, I got the following:

Model	Test F1	Test EM
Character Embeddings Version 1	63.357	59.932
Character Embeddings Version 2	63.802	60.186
Subword Embedding With Char V2	63.853	60.000

It appears that all of the subword models ended up overfitting somewhat, hence their slightly reduced performance in comparison to only using character level embeddings. I was expecting subword level embeddings to work pretty well, because they made more intuitive sense to me than the CNN character embeddings, and so I was disappointed with these results. However, upon reflection it makes sense that adding in as many extra trainable parameters as I did with the subword embeddings, especially with so many subwords and therefore so few examples per subword, allowed the model too much flexibility and allowed it to overfit. For further experiments, I would play with some ways to reduce the flexibility of the model, like using a shorter list of subwords or using smaller embeddings, like 8 dimensional vectors instead of 64 dimensional, or perhaps adding in dropout earlier on so that the model had to distribute its learning about the embeddings a bit more. However, all models tested other than the ensemble did outperform the baseline, so that means the models were at least adding some value over the baseline, somewhat validating this as a useful line of study.
I was also disappointed to see how poorly the ensemble model performed. Instead of the usual pattern in which ensemble is able to take the best of each model, in this case ensembling seemed to compound the errors the model ended up making significantly increasing NLL and decreasing performance to below the performance of the baseline. I'm not sure if this is a result of the ensemble technique I used being malformed, or if the models were all making a similar set of mistakes, and so ensembling just made the model make those mistakes more confidently.

6 Analysis

Observe the following graphs, with the model each line of each graph corresponds to noted next to it.



The first pair of graphs correspond to models that don't include any subword embeddings, and the second pair of graphs correspond to models that do include subword embeddings (with the exception of baseline, which is provided for reference). Note how in the models that do not include subword embeddings, the NLL tends to stabilize at the minimum it reaches, but in the subword models, the NLL tends to reach a minimum very quickly and then increase, sometimes very dramatically. In my opinion, this is indicative of the models overfitting, leading to NLL on the dev set to increase as the model overfits more. Using subword embeddings only, with no character level embeddings, seems to reduce this overfitting only slightly, if at all, and using variable word embeddings seems to dramatically increase the amount of overfitting the model does. This second observation holds true even without subword embeddings, as can be seen on the teal line in the first set of graphs. Also, note that attempting to combine the character and subword embeddings with an MLP before they are fed into the model, which was designed to provide some earlier summarization, also did not really ameliorate the overfitting problem. This indicates that the overfitting is more a result of the subword embeddings themselves being the source of the overfitting, and not the way that those embeddings are processed before being fed into the next layer.

7 Conclusion

Though the results weren't quite as immediately positive as I was hoping for, I think this work illustrates that subword embeddings could be a promising avenue of future study in helping understand question answering, so long as one is able to combat the overfitting problem. Because all but one of the subword models did manage to outperform the baseline, it seems that subwords do provide a value add. I think one particularly promising future experiment would be to use pretrained subword embeddings (something that wasn't feasible with the scope of this project), since that would eliminate the need for subword embeddings to be trained during the regular training cycles, which would prevent overfitting on the training data. Another possible area would be to use more complex models to process the embeddings earlier to force greater summarization earlier on and force some bottlenecking with the ability of the model to overfit, which might allow the model to learn more robust, less training-data-specific embeddings.

References

[1] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension, 2018.

[2] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for SQuAD. In *Association for Computational Linguistics (ACL)*, 2018.

[3] Yoon Kim. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882, 2014.

[4] Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, and Armand Joulin. Advances in pre-training distributed word representations. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.