CS224N HW4: SQuAD Reading Comprehension Learning

Thomas Ayoul
tayoul@stanford.edu

Sebastien Levy
seblevy@stanford.edu
codalab username: seblevy

Abstract

The Stanford Question Answering Dataset (SQuAD) consists on more than 100000 triplets paragraph, question, answer from Wikipedia, with various answer types and length. By training a coattention-based encoder followed by a mostly linear spanning decoder and a inversion handling post processing, we are able to achieve a F1 of 56% and an exact match of 43% with a particularly good accuracy on Named entities and numerical answers.

1 Introduction

1.1 Problem Description

When learning a new language, the first skill to master is understanding a text. This can easily be tested by answering simple questions about it. In a similar way, a good Natural Language Processing model must at least be able to capture the information contained in a text, before focusing to harder or more specific tasks. We thus want them to be able to answer some reading comprehension questions. The goal of this project is to develop a tool that can master reading comprehension using neural networks. To do so, we want our model to understand the text and how each of its word is related to the question asked. This part is called the encoder, and would correspond to a machine language. Once that information is captured, we now want our model to be able to locate the answer to the given question in the paragraph. We call this part the decoder.

1.2 Metrics and Related Work

We use two metrics to test the performance of the predicted answers. Exact match (EM) give the proportion of prediction that exactly matches one of the ground truth answers. The second metric is the F1 score (geometrical mean between precision and recall) between the predicted words from the answer and the ground truth. In case of multiple answers, we just take the maximum of the scores.

Model	Baseline	r-net (single)	r-net (ensemble)	Oracle
F1 Score	51.0	80.8	84.0	91.2
Exact Match	40.0	72.4	76.9	82.3

Table 1: Performance of the baseline, the state of the art methods and the oracle

The original baseline model for this task is a simple logistic regression trained on feature extracted of every possible candidate for the answer. The feature set includes unigram/bigram frequencies, number of words to left/right, Part of Speech and Dependency features and features matching frequencies of words in the question and in the answer. The oracle has been made by comparing human performances on this task. Finally, the state of the art method is r-net developed by Microsoft Research Asia. The performance for both metrics are summarized in table 1.

2 Data description

We will use the data from Stanford Question Answering Dataset (SQuAD [3]), which consists in around 100,000 questions-paragraphs-answers triplets. The texts were extracted from Wikipedia and the answers were selected by humans. The answers are defined as a spanning subset of the initial paragraphs. The distribution of questions and paragraph lengths can be found in figure 1.

The answers to SQUAD questions are quite varied with numerical answers (date and oth-

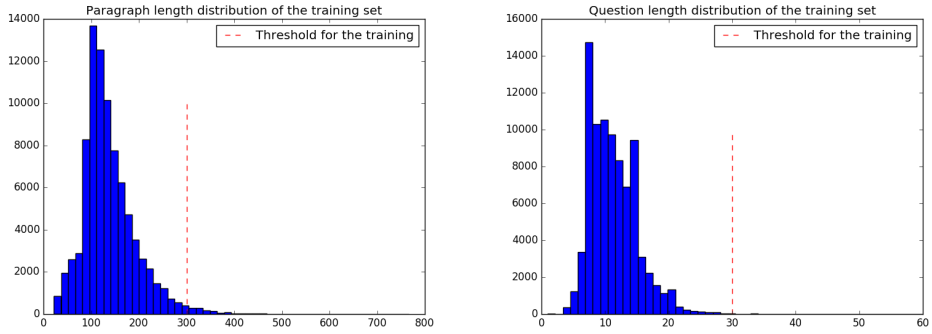

Figure 1: Distribution of paragraph and question length in the training set

ers), named entities (person, location ...), clauses and noun/verb/adjective phrases. The distribution of these different answer types is detailed in figure 2.

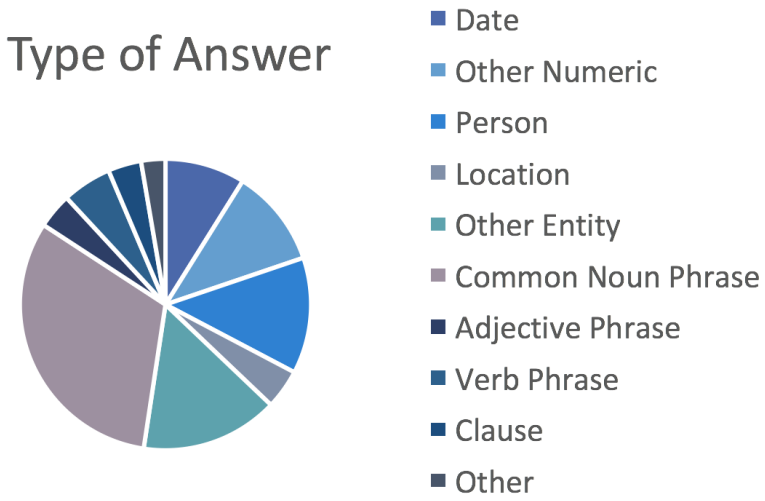

Figure 2: Proportion of different answer types in the dataset

3 Models

3.1 Pre-processing

Before actually training/testing our model, we need to perform a few preprocessing step:

- We tokenize the paragraph and the question by simply splitting on spaces and separating punctuation from text
- Because we do the optimization by batches, we need to extend paragraphs and questions to have the same length. At training time, we chose a paragraph size of 300 and a question size of 30 and at test time the longest paragraph size and question size in the dev/test set. We then fill the vectors with 0 and create a boolean mask stating if the values are added or not.
- At training time, we discard paragraph longer than 300 words and questions longer than 30 words. This simplifies the architecture used for the training and can still be used for longer size when testing because the variables never depend on the number of words (recurrent approach). We chose those limits to keep more than 95% of the examples at training time.
- We finally convert words to its corresponding indices in the embedding table (see section 6) for an easier lookup and a decrease in variable storage.

3.2 Encoder

The first step for all encoders is to convert in embeddings the word indices of the paragraphs and questions in the batch. We used GloVe embeddings trained on Wikipedia 2014 and Gigaword5. As explained in section 6 we chose to not train the embeddings (constant). In the rest of the section, we will describe the two most performing models we used, the attention encoder and the coattention encoder.

3.2.1 Attention Encoder

The attention encoder consists in three step: encoding the question ant the paragraph, computing the attention to the question for each word of the paragraph and mixing the attention and the paragraph representation to get the final encoded state. The whole process is summarized in figure 3.

Encoding the embeddings: We used bidirectional Recurrent Neural Network (RNN) with LSTM cells to encode separately the question and the paragraph. Using a bidirectional RNN helps capturing the relationship between words in the two directions and has proven to improve results on unidirectional ones. The LSTM cell slightly improves the overall performance.

Computing the attention: To compute the attention, we first do a weighted dot product between each element of the question with the elements of the paragraph and we then multiply it with the paragraph. This step can be summarized as:

$$A = \text{softmax}\left(PWQ^T\right)Q$$

Although learning the weights gives more power to the model, it implies larger overfitting and yields worse performance. The best performing attention encoder used $W = I$.

Mixing attention and paragraph representation: To obtain the final encoding, we finally concatenate our attention to the paragraph representation and apply a linear layer to each element of the batch.

3.2.2 Coattention Encoder

The coattention encoder [5, 2] tries to extend the previous model by computing the attention of the each word of the question to the paragraph and then get the attention of each represented word of this attentive question to the paragraph. The three same phases appear:

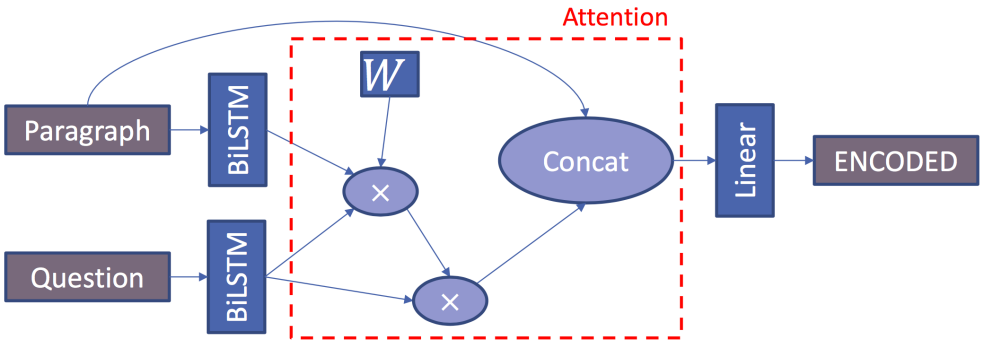

Figure 3: Attention based encoder

Encoding the embeddings: There are two major modifications from the attention encoder. First we use unidirectional RNN to learn the representations. This decreases the parameter space and the two directions will be handled by the two successive attention followed by the mixing step. The other difference is that we use a connected layer before the LSTM. This helps preventing words that do not carry that much meaning but that are still in both texts to have less importance (”the”, ”a” ...).

Computing the coattention: Computing the coattention is pretty similar to the previous process applied twice. We do not use attention weights this time as the model is already more complex and it has proven useless in the simpler setting.

Mixing coattention and paragraph representation: To mix coattention with the paragraph, we will use a bidirectional LSTM. This captures interactions between ”attentive” representations of the words in the paragraphs.

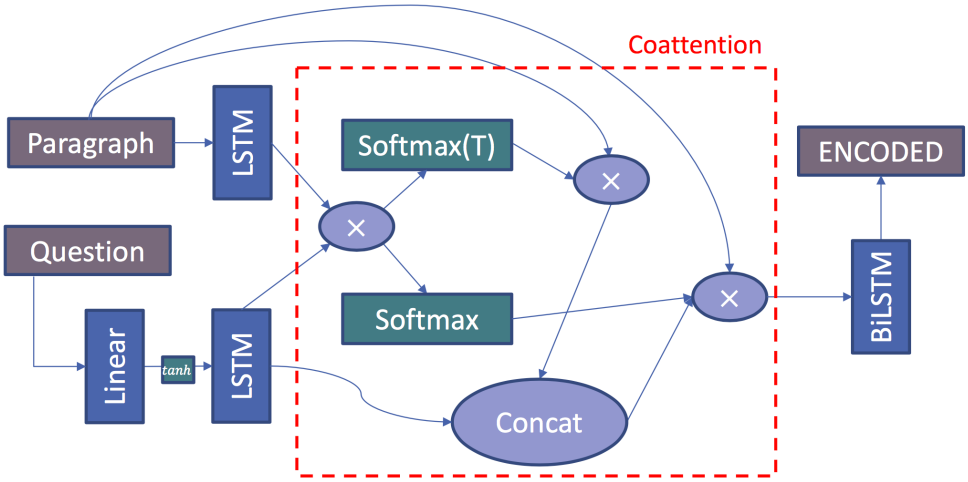

Figure 4: Coattention based encoder

3.3 Decoder

3.3.1 Simple Linear Decoder

The decoder is the one proposed for the baseline. We first apply a linear layer to the encoded vector to get the start. We then feed this prediction to a LSTM to get the prediction for the end, as shown in the figure 5. This will use the probabilities of each word being the start to learn the end position.

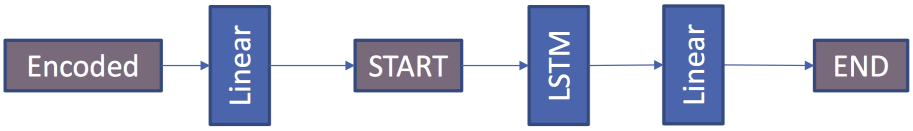

Figure 5: Simple linear decoder

Linear Mixed decoder A drawback of the simple linear decoder is that it only uses the starting probabilities. To improve the decoding, we modified the previous decoder by adding adding a LSTM at the beginning, and a feedback loop and give the predicted starting probabilities and the encoded vector as an input to the LSTM that predicts the end. We decided to add as well the information of the first LSTM by mixing it linearly with the output of the second LSTM. This second steps uses different information as it can memorize information used to indirectly compute the start but not outputted. The architecture for the decoder is summarized in figure 6:

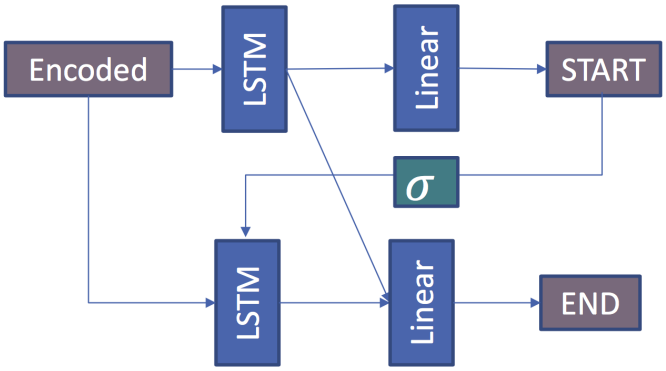

Figure 6: Linear mixed decoder

3.4 Post-Processing

Our biggest fear for this project was that the decoder predicts a start after the end. We thus printed at each epoch what we called the number of inversions, i.e. the number of times the end is predicted before the start. Even for our best performing models, we converge to a 6-7% inversion rate. These predictions are genuinely wrong so we decided to post process them. In those cases, we choose set the most probable between the start and the end as the predicted position and look for the other as the argmax probability in the admissible range. In the extreme cases, we could only pick one word for the answer.

4 Training process

4.1 Regularization techniques

In order not to overfit, several techniques were used. For big neural networks, dropout was proved to be a very good way to reduce overfitting. Indeed, it enables a better diversification of the neurons by randomly switching some of them off during each epoch of the training.

We also tried L^2 regularization on the linear layers of our model. However, it did not improve our model : It yielded worse performance. We observed that early stopping and dropout were enough for us to reduce overfitting, with difference between training and validation F1 of less than 5-7% before convergence.

The last regularization technique tried was batch normalization[1] of the connected layer

output. Although it speeded up the decrease in loss, it did not increase our metrics, so we decided to discard it in our final model.

4.2 Loss and Descent algorithm

To train our model, we minimized the sum of the cross entropy between the predicted start/end indices and the ground truth.

Optimizer : Vanilla gradient descent was not very appropriated to our highly non convex problem, so we opted for ADAM optimizer. By using momentum to escape local minima and considering the norm of the gradient in the learning rate in order to do small steps when we meet some steep gradients, it generally performs better on complicated neural network architectures.

Learning rate: Using learning rate around 0.01 lead to poor results. We understood that by seeing some oscillations of the loss and high normed gradients. By reducing the learning rate after a few epochs, we gained a lot of precision in our algorithm.

In order to cope with this trade-off, we decided to us a learning rate with exponential decay. Indeed, we want the learning rate to be high enough to converge quickly to minimum but we want it to be small to find the actual minimum. The learning rate changes with time and is defined as:

$$\alpha_t = \max(\alpha_\infty, \alpha_0 e^{-rt}) = \max(10^{-6}, 10^{-3} \times (0.84)^{t/7000})$$

Batch size: For precision and computational efficiency, we want to use batches of samples.

Regarding performance, the batch size matters a lot. Indeed, the bigger it is, the less updates we make. We want to do frequent updates to converge faster but we do not want to do "bad" updates. Considering samples by batch enables us to average their gradients and thus remove the noise created by "bad" samples.

The performance issue comes from the fact that we are using GPUs. For instance, when you are programming matrix operations in CUDA, you need to be careful about batch size. Indeed, the size of your submatrix has to be consistent with the architecture of the hardware. You then usually choose batch size that are dividers of the numbers of cells of the GPUs, which are both powers of 2. We assumed, that this fact was still true for more complex operations (as RNN in our case) and used batch size of 64 samples, as the GPU provided by Azure is a TESLA M60 GPU. It has 4096 NVIDIA CUDA cores (2048 per GPU). We hope this fact will bring computational efficiency.

Gradient clipping: In order to avoid exploding gradients we performed clipping to a global norm of 8. It enables us to reduce the "jumps" when exploring our space to find the minimum. We noticed that near the optimum, the gradient was often getting large and clipping gradient helped a lot there.

5 Results

5.1 Training loss

For the co-attention model with our linear mixed decoder, we get the training curves in figure 7. We can see in it that we stop our training process when the loss starts to oscillate which occurs after 8-9 epochs in this case. It coincides when the EM and F1 scores start to flatten and the gradient norms start to increase. This is how we performed early stopping in order not to overfit the train set.

Number of inversions: Even with the most performing models, we get between 6 and 7 % of inversion. The simple post-processing is thus useful there as it enables us to gain around 1% of F1, which is quite significant.

5.2 Dev Set

Table 2 shows some of our best performing models and their scores. Our best model on the dev set was the one using co-attention based encoding and our homemade linear mixed de-

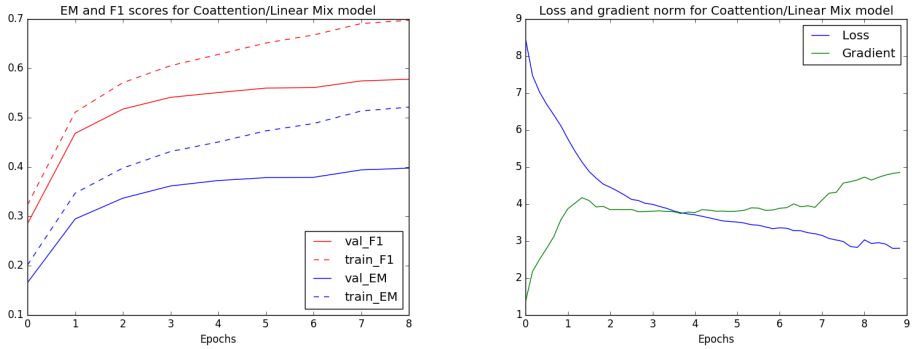

Figure 7: Loss and gradients

Encoder \ Decoder	Linear Sequence	Linear Mixed
Attention LSTM	36.03 (24.47)	/
Coattention LSTM	55.88 (**43.30**)	**57.96** (39.76)
Coattention GRU	54.32 (34.81)	53.78 (35.01)

Table 2: Performance of different models on the dev set

coder. We got a F1 of 58% and a EM of 40%. The EM is slightly better for the simple linear decoder.

This table shows as well that models that use LSTM cells, were generally better than the ones with GRU cells. Note that we tried several values for the size of the hidden state of both the encoder and the decoder, as well as different embedding sizes. The trade-off is clear: we need enough parameters to have a predictive power, but not too many to limit overfitting and to keep the computation time quite low.

5.3 Test Set

Although performing a little worse on the validation set, our best performing model on the test set was the coattention with simple linear decoder. We got a F1 of **56.086** and a EM of **43.858**. The parameters used are listed in 3. Because of the simplicity of the decoder, a higher state size was necessary.

We currently are 52nd in the leaderboard.

RNN cell	Embedding size	Encoder size	Decoder size	Dropout	Batch Normalization
LSTM	100	50	300	0.85	No

Table 3: Parameters for the best performing model

5.4 Diagnostic Experiments

Our best model works great with named entities and numerical answers. For instance, let us take the question ”What performer lead the Super Bowl XLVIII halftime show?”. The answer is Coldplay. It is easier to find as it is a ”simple” answer : only one name which is heavily related to music, and thus highly correlated to term like ”performer” or ”show” which are not related to the football field.

However, it had poor results when the questions are long, syntactically complicated and/or featuring unclear words (or with more than one meaning) like things or stuff ...). For instance, for the question : ”What was the name of the event at the The Embarcadero that was held prior to Super Bowl 50 to help show off some of the things that San Francisco has to offer?” The true answer is:

```
"Super Bowl City"
```

whereas we predicted

```
"Super Bowl City" opened on January 30 at Justin Herman Plaza
on The Embarcadero, featuring games and activities that will
highlight the Bay Area 's technology, culinary creations,
and cultural diversity. More than 1 million people are expected
to attend the festivities in San Francisco during Super Bowl Week"
```

We can clearly see that it gets confused by the "things" in the question and the multiple subordinate clauses as well as the two appearances of Super Bowl at the two extremes of the answer. Trying to use a model that estimates the probability of each word to be in the answer could help for those kinds of question. Using a mixture of these two types of models could definitely improve the performance when the current model outputs long answers.

6 Next Steps

There are other directions we did not have the time to explore that could have improved our results.

Embedding: variable or constant? GloVe embedding were used as constants in our model. One could consider setting the embedding as variables and specify their training to that task. However, we did not think it was such a good idea as it would have increased a lot the number of parameters and then tend to overfit the training set, and maybe "destroy" the word vector representation in the early steps. Moreover, when selecting an architecture it can be better to compare different modelisation given constant GloVe embeddings.

PTE: Learning an embedding for the task: Another way of getting embedding more adapted to this task would have been to follow the approach of Predictive Text Embedding [4]. By designing several graphs to represent the data (connecting words to their question id, to their paragraph id, to word appearing in the same window, connecting questions to answers ...) we could directly get better information on the relationships between questions, paragraphs and answers in the embeddings themselves and then apply any of the architectures stated above.

GPU vs CPU allocation We did not really try to think about memory allocation here. We just decided to put embedding computations and storage on the CPU in order not to fill the GPU with the word embeddings lookup. A better distribution of resources between them can improve the speed of our training.

Convolutional neural network As the answer is a span of the initial paragraph, we thought of a convolutional based approach to better capture group of words and in particular patterns within the sentence.

7 Conclusion

Although for us it seems to be an easy task, understanding a text and answering questions about it, is very difficult. It takes much time for children to learn how to read and understand paragraphs and questions. Moreover, the learning process for humans is driven by necessity and oriented teaching. In comparison, the learning metric for a computer (cross entropy) and its performance metric (F1) seems quite far. Moreover, capturing complex patterns used by author for style can be hard to put close to easier formulation of the same idea by algorithm not trained to learn to only read.

In few training hours, we still manage to achieve promising results with F1 scores close to 60%, which is not that far from human performance. The encoders/decoders used can easily be plugged in more complex task as one learns to "read" and the other to "answer".

References

[1] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[2] Jiasen Lu, Jianwei Yang, Dhruv Batra, and Devi Parikh. Hierarchical question-image co-attention for visual question answering. In *Advances In Neural Information Processing Systems*, pages 289–297, 2016.

[3] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[4] Jian Tang, Meng Qu, and Qiaozhu Mei. Pte: Predictive text embedding through large-scale heterogeneous text networks. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1165–1174. ACM, 2015.

[5] Caiming Xiong, Victor Zhong, and Richard Socher. Dynamic coattention networks for question answering. *arXiv preprint arXiv:1611.01604*, 2016.