Bidirectional LSTM-RNN with Bi-attention for reading comprehension

(On leaderboard, we use the team name zy99)

Ziyi Yang
Department of Electrical Engineering
Stanford University
Stanford, CA 94305
zy99@stanford.edu

Guoxi Xu
Department of Materials Science and Engineering
Stanford University
Stanford, CA 94305
guoxixu@stanford.edu

Bera Shi
Institute for Computational & Mathematical Engineering
Stanford University
Stanford, CA 94305
bshi@stanford.edu

Abstract

In this work, we implemented bi-directional LSTM-RNN network to solve the reading comprehension problem. The problem is, given a question and a context (contains the answer to the question), find the answer in the context. Following the method in paper [11], we use bi-attention to make the link from question to context and from context to question, to make good use of the information of relationship between the two parts. By using inner product, we find the probabilities of the context word to be the first or last word of answer. Also, we used some improvement to the paper reducing the training time and improving the accuracy. After adjusting parameters, the best model has performance of F1=48% and EM=33% leaderboard.

1 Introduction

We implemented a neural network architecture for Reading Comprehension using the recently published Stanford Question Answering Dataset (SQuAD) [1].

SQuAD is comprised of around 100K question-answer pairs, along with a context paragraph. The context paragraphs were extracted from a set of articles from Wikipedia. Humans generated questions using that paragraph as a context, and selected a span from the same paragraph as the target answer. The following is an example of a triplet ⟨question, context, answer⟩.

> Question: Why was Tesla returned to Gospic?
>
> Context paragraph: On 24 March 1879, Tesla was returned to Gospic under police guard for **not having a residence permit**. On 17 April 1879, Milutin Tesla died at the age of 60 after contracting an unspecified illness (although some sources say that he died of a stroke). During that year, Tesla taught a large class of students in his old school, Higher Real Gymnasium, in Gospic.
>
> Answer: not having a residence permit

In the SQuAD task, answering a question is defined as predicting an answer span within a given context paragraph. In our project, we need to give the indexes of the start word and the end word. We explored different models to solve this problem.

2 Approach

2.1 Problem Setup

In the SQuAD task, the goal is to predict an answer span tuple $\{a_s, a_e\}$ given a question of length n, $\mathbf{q} = \{q_1, q_2, ..., q_n\}$, and a supporting context paragraph $\mathbf{p} = \{p_1, p_2, ..., p_m\}$ of length m. So, the input is ($\mathbf{q}$, $\mathbf{p}$) and we want to get the output $\{a_s,a_e\}$ indicating the start position and end position of the answer in paragraph p, respectively. Note that in this task $a_s \leqslant a_e$, and $0 \leqslant a_s, a_e \leqslant m$. In the training set, for each question the answer interval is unique. In the dev set, for each question there are 3 possible answers.

2.2 Architecture

We refer a paper[11] and do some revision to set up our very fuirst model. The method in paper consists of six layers and each layer is illustrated as follows(we modified some of them to improve the performance, so some of them will be different from the ones showed in paper, but we keep the definition of symbols to be the same with that in the original paper):

1. Word Embedding Layer. Word embedding layer maps each word to a high-dimensional vector space. We use pre-trained word vectors, GloVe, to obtain the fixed word embedding of each word.

2. Embedding Layer. We use a Long Short-Term Memory Network (LSTM) on top of the word embeddings provided by the previous layers to model the temporal interactions between words of query and context. As we mentioned "Bi-directional LSTM", we use one LSTM to deal with text from start to the end, and the other LSTM reversely deals with the text. Then we concatenate the output of two LSTMs as the final encoding of query and context. Hence we obtain $H \in R^{2d\times T}$ from the context word vectors X, and $U \in R^{2d\times J}$ from query word vectors Q. Note that each column vector of H and U is 2d-dimensional because of the concatenation of the outputs of the forward and backward LSTMs, each with d-dimensional output. The forward and backward LSTMs help us get more information about the inside relationship among words inside the context and the question.

3. Attention Flow Layer. Attention flow layer to find the "resonance" between contexts and query, namely find the most relevant query words for each context words and the most relevant context words for each context words. In our work, the attention vector will change in every step, base on the attention vector at last step and the current word. So the attention vector at each time step and the embeddings from previous layers flow during the reading process of the whole text. And this temporal-related property of LSTM successfully prevents information losing.

Two attentions, namely query-to-context attention and context-to-query attention, are computed from question encoding H and the query encoding U. The outputs of the layer are the query-aware vector representations of the context words, G, along with the contextual embeddings from the previous layer. We compute attentions in two directions: from context to query as well as from query to context. We create a matrix S to describe the similarities between the contextual embeddings of question(U) and the embeddings in context(H), $S \in R^{T \times J}$, where S_{tj} indicates the similarity between t-th context word and j-th query word and it Is computed via

$$S_{tj} = \alpha\,(H_{:t}, U_{:j}) \in R$$

where α is a function that computes the similarity between its two input vectors with trainable weights, $H_{:t}$ is t-th column vector of H, and $U_{:j}$ is j-th column vector of U. There multiple ways of choosing α and w did experiment to find out the best function. We will come back to this in later part of this report. Now we use S to obtain the attentions and the attended vectors in both directions.

In our work we use different kinds of inner product to describe the similarity of words in question and in context. For example, one way we used to calculate $\alpha(h, u)$ is $\alpha(h, u) = h^{\top} W_{bi} u$, and the matrix W_{bi} can be optimized by training. More detailed information can be seen in “Experiments” part.

3(1). Context-to-query Attention. Context-to-query (C2Q) attention signifies which query words are most relevant to each context word. In other words, it likes that we first read the context then find the match part in the question. $a_t \in R^J$ represents the attention weights on the query words by t-th context word, $\Sigma a_{tj}=1$ for all t. The attention weight is computed by $a_t = \mathrm{softmax}(S_{t:}) \in R^J$, and each attended query vector is $\tilde{U}_{:t} = \Sigma_j \, a_{tj} \, U_{:j}$. Hence $\tilde{U}$ is a $2d \times T$ matrix containing the attended query vectors for the entire context.

3(2). Query-to-context Attention. Query-to-context (Q2C) attention signifies which context words have the closest similarity to one of the query words and are hence critical for answering the query.

We get the attention weights on the context words by $b = \mathrm{softmax}(\max_{col}(S)) \in R^T$, where the maximum function ($\max_{col}$) is performed across the column. Then the attended context vector is $\tilde{h} = \Sigma_t \, b_t H_{:t} \in R^{2d}$. This vector indicates the weighted sum of the most important words in the context with respect to the query. $\tilde{h}$ is tiled T times across the column, thus giving $\tilde{H} \in R^{2d \times T}$.

After the two steps we get two sets of attentions $\tilde{U}$ and $\tilde{H}$. Finally, the contextual embeddings and the attention vectors are combined to yield G, where each column vector can be considered as the query-aware representation of each context word. We define G by

$$G_{:t} = \beta(H_{:t}, \tilde{U}_{:t}, \tilde{H}_{:t}) \in R^{dG}$$

where $G_{:t}$ is the t-th column vector (corresponding to t-th context word), β is a trainable vector function that fuses its (three) input vectors, and d_G is the output dimension of the β function. β can be different, such as in the paper: $\beta(h, \tilde{u}, \tilde{h}) = [h;\tilde{u};h \circ \tilde{u};h \circ \tilde{h}] \in R^{8d \times T}$ (i.e.,$d_G = 8d$). In our model, we use a modified version of $\beta(h, \tilde{u}, \tilde{h}) = \max(0, W_{mlp}[h;\tilde{u};h \circ \tilde{u};h \circ \tilde{h}] + b_{mlp}) \in R^{8d \times T}$, which is a linear transformation of the original β, and it’s like a ReLU. After this process, we combined the information in question and in context.

5. Modeling Layer. We use again, LSTM, to encode query-aware representation for final output. The output of the modeling layer represents the interaction among the context words conditioned on the query. We use two layers of bi-directional LSTM, with the output size of d for each direction. So we get a matrix $M \in R^{2d \times T}$, which is passed onto the output layer to predict the answer. Each column vector of M contains contextual information about the word with respect to the entire context paragraph and the query, which comes from the information in G.

6. Output Layer. As we need to get the start word and the end word in the answer, we try to transform the question to be that finding the probability of every word to be the start word. We obtain the probability distribution of the start index over the entire paragraph by

$$p^1 = \mathrm{softmax}(w^T_{(p1)}[G;M])$$

where $w_{(p1)} \in R^{10d}$ is a trainable weight vector. For the end index of the answer phrase, we pass M to another bidirectional LSTM layer and obtain $M^2 \in R^{2d \times T}$. Then we use M^2 to obtain the probability distribution of the end index in a similar manner:

$$p^2 = \mathrm{softmax}(w^T_{(p2)}[G;M^2])$$

7. Training. We define the training loss (to be minimized) as the sum of the negative log probabilities of the true start and end indices by the predicted distributions, averaged over all examples:

$$L(\theta) = -\frac{1}{N}\sum_{i}^{N} \log\left(P^1_{y^1_i}\right) + \log\left(P^2_{y^2_i}\right)$$

And we used Adam optimizer with exponential decay learning rate. Details on how to pick index form the vector will be elaborated in later part.

2.3 Evaluation

The original SQuAD paper introduces two metrics to evaluate the performance of a model: Exact-Match (EM) and F1 score. We use the same metrics to evaluate our model. Exact match is a metric that measures the percentage of predictions that match one of the ground truth answers exactly.

F1 score is a metric that loosely measures the average overlap between the prediction and ground truth answer. We treat the prediction and ground truth as bags of tokens, and compute their F1. We take the maximum F1 over all the ground truth answers for a given question, and then average over all the questions.

4 Experiments

In this section, we presented several strategies to improve model accuracy and training speed, which includes aspects of data loading, texts padding, different choices of fusing information and dropout to LSTM and hyper-parameter tuning (for instance, learning rate, hidden size, batch size and dropout).

4.1 Model-free improvement

Here, we implemented some simple, model-free and effective modifications on pipelines like, data loading, sentence padding, output prediction, etc. These modifications, although seemingly trivial, have surprisingly huge impacts on model training and performance. More details will be given below.

In previous homework, we just simply padded every text into the global max length. The method doesn't negatively affect the model's performance, since the max length is quite small. But in the question answering task, the context lengths are distinct for each other. The max context length is 766, but most context have length around 300, it not reasonable to padded it to the double length! In our model, the padding max length is passes by feed dictionary and is the max length in one batch. It turned out that tons of time is saved on LSTM RNN since we it doesn't need to go to blank states and computation cost drops also because the tensor size decreases dramatically by 50%. Another twin method is that when loading data, we can simply load example with similar lengths together, so that the max length in the batch is economical on overall datasets. It tuned that the running time drops per epoch drops from 4 hours to 30 minutes, making the training speed almost 8 times faster.

Another method is to change the way of predicting index. Naively, we can just pick the argmax index form the start-predicting vector and end-predicting vector. One problem is the start index predicted can be bigger than end index predicted. Of course, we empirically swap the index but after all this is not a scientific method. Here, we just pick the pair argmax(S_i*E_j), this can be achieved by dynamic programming in linear time and naturally avoids the problem mentioned above. The table below shows that this method improves the performance dramatically by simply changing the way to index form predicted vector.

Index Picking Method (on the very same prediction at one development set)	F1	EM
Naïve Argmax	22.7%	15.8%
Dynamic Programming	**32.9%**	**23.5%**

The starter code will simply samples word not included in vocabulary to <unk> token. If we just simply use the reverse dictionary mapping word ids back to word, lots of answers predicted will include <unk>, and this has negative impacts on F1 ad EM. Here we passed the original context into the model and slice the answer block directly from contexts. This model-free strategy boost the F1 and EM as indicated in table below.

Dealing with <unk>	F1	EM
Predicting with <unk>	32.2%	19.6%
Slicing directly from context	**45.8%**	**25.0%**

4.2 Hyper-parameter tuning

In our experiment, we first modified the batch to improve the efficiency. We sort the contexts in terms of length so that in one batch the context length will be similar. After that process, the LSTM can save the time of dealing with many unnecessary spaces after the end of context. Compared to randomly selecting contexts into the batch, this method saved much time and space. It only takes 40 minutes to run an epoch on the training dataset, however, without this modification the time for running one epoch will be 4 hours. And at first this change increased F1 by about 10 percent.

Aside from the model mentioned above, there are many parameters that can be adjusted. For example, learning rate, hidden size and dropout can be adjusted to get better performances.

First, we use different hidden size to train the model. We use 32,64,128 hidden sizes, and in an epoch, keep the other conditions to be the same and observe the loss versus step during the training process. We get the figure below.

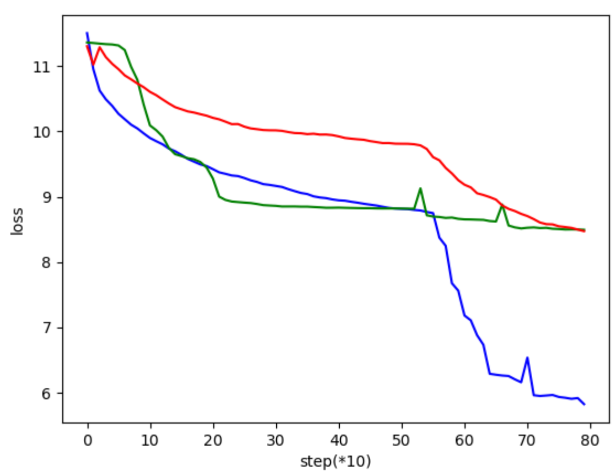

Figure 1: Loss-step with different hidden size (green:128 blue:64 red:32)

From Fig.1 we can see when the hidden size is too large, at last the model will be much complicated and the learning rate will decrease. But for the small hidden size, the learning rate is also slow because there isn’t enough space to learn the features. We can get a conclusion that when hidden size is about 64 the learning result is better. However, in the paper the hidden size is 100, I think in this range the model will perform well.

Also, learning rate will have an influence on the model. We use learning rate 0.01,0.02 and 0.04 and keep the other conditions to be the same, during one training epoch we get the figure below:

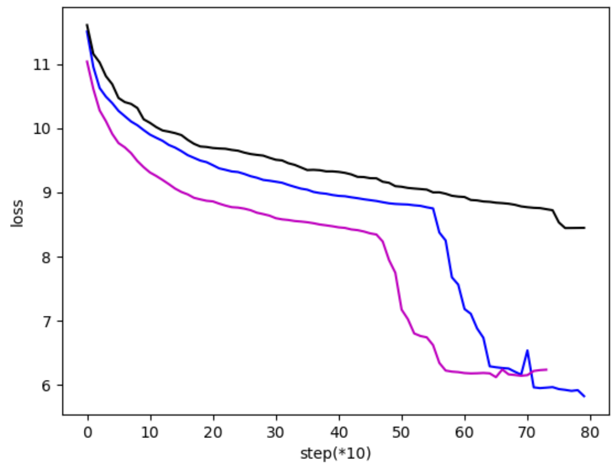

Figure 2: Loss-step with different learning rate (black:0.01 blue:0.02 purple:0.04)

From Fig.2 we can see the small learning rate will make the loss decrease slowly especially in the longer timescale. But with the very high learning rate, at first it can have a good effect but in the longer timescale there is some fluctuation around the optimal point. A medium learning rate will also be helpful to the model.

Also, the dropout will affect the model performance. Due to the complication of our model, we need to add some dropout in our network. At first, we used different values of dropout and the results can be seen below:

Dropout	F1	EM
0.6	26.2%	9.6%
0.7	33.6%	13.4%
0.8	**39.4%**	**15.1%**

After this trial, we choose 0.8 as the dropout rate in our model.

4.3 Final Performance

We went through four major versions of model. The first one is naïve model without any attention for model pipeline building. The second version is the baseline model v2 suggested on Piazza. The third one is BIDAF model evolving from baseline model. The fourth one is the model finally submitted with all model-free modifications and tricks mentioned above. The progress on best F1 and EM is shown in the table below.

Model	F1	EM
Naïve Model without attention	10.5%	1.9%
Baseline model v2 suggested on Piazza	22.5%	7.3.7%
Pure BIDAF	33%	12.9%
BIDAF with our improvements	48%	33%

5 Error Analysis

Although our model performs well in the whole dataset, there are some errors in the prediction. From observation, we can divide the major errors into 4 categories:

1.predict several more words: (the upper one is prediction and the lower one is real answer)
winner Taylor Swift
Taylor Swift

2. predict several less words:
Establishing the President 's Committee
Establishing the President 's Committee on Equality of Treatment and Opportunity

3.find the wrong place (the kind of word is right but the answer is wrong):
2002
2008

4.prediction is long text contains the target answer:
$ 737,000. Estimated revenue more than doubled from $ 404 million in season three to $ 870 million in season six. While that declined from season eight onwards, it still earned significantly more than its nearest competitor, with advertising revenue topping $ 800 million annually the next few seasons. However, the sharp drop in ratings in season eleven also resulted in a sharp drop in advertising rate for season twelve, and the show lost its leading position as the costliest show for advertisers. By 2014, ad revenue from had fallen to $ 427 million
800 million

Now almost all the errors are one of the 4 categories. As we noticed most of the answers are nouns or numbers, and most of answers are very short, we can add some penalize to the very long prediction. Also, in the future we can make improvement on finding the full set of one phrase, and decrease the answers which contains more than 1 kinds of words (like 1, because most of this kind of predictions are wrong). We are still trying to realize the mechanisms behind the prediction behavior.

6 Conclusion

We established a LSTM-based bidirectional recurrent neural networks with bi-attention mechanism for context understanding on SQuAD data set. We find that slight modification on data loading and texts padding gives rise to huge decrease on training time, from 4 hours to 30 mins per epoch. Also, we experimented on different hyper-parameters, such as learning rate, batch size and hidden size, etc to polish the model. We find that small learning rate with exponential decay (0.02), with big batch size (64) and small hidden size (64) works best for us. And the training finishes usually after 4 or 5 epochs the best F1 and EM score we get now is 48% and 33% after 10 epochs on around 80000 examples

7 Acknowledgement

We want to thank Prof. Chris Manning, Dr. Richard Socher and all the TAs for their efforts through this quarter. This is a big and amazing class. Especially for those helpful and patient TAs, thank you all for giving so valuable advice on homework! And thank Microsoft for generously providing computation resources.

References

[1] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. CoRR, abs/1606.05250, 2016.

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

[3] Zhiguo Wang, Wael Hamza, and Radu Florian. Bilateral multi-perspective matching for natural language sentences. arXiv preprint arXiv:1702.03814, 2017.

[4] Shuohang Wang and Jing Jiang. Machine comprehension using match-lstm and answer pointer. arXiv preprint arXiv:1608.07905, 2016.

[5] Danqi Chen, Jason Bolton, and Christopher D Manning. A thorough examination of the cnn/daily mail reading comprehension task. arXiv preprint arXiv:1606.02858, 2016.

[6] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. arXiv preprint arXiv:1611.01603, 2016.

[7] Kenton Lee, Tom Kwiatkowski, Ankur Parikh, and Dipanjan Das. Learning recurrent span representations for extractive question answering. arXiv preprint arXiv:1611.01436, 2016.

[8] Yelong Shen, Po-Sen Huang, Jianfeng Gao, and Weizhu Chen. Reasonet: Learning to stop reading in machine comprehension. arXiv preprint arXiv:1609.05284, 2016.

[9] Yang Yu, Wei Zhang, Kazi Hasan, Mo Yu, Bing Xiang, and Bowen Zhou. End-to-end answer chunk extraction and ranking for reading comprehension. arXiv preprint arXiv:1610.09996, 2016.

[10] Zhilin Yang, Bhuwan Dhingra, Ye Yuan, Junjie Hu, William W Cohen, and Ruslan Salakhutdinov. Words or characters? fine-grained gating for reading comprehension. arXiv preprint arXiv:1611.01724, 2016.

[11] MinJoon Seo, Aniruddha Kembhavi, Ali Farhadi, and, Hananneh Hajishirzi. Bi-Directional Attention Flow For Machine Comprehension. arXiv preprint arXiv:1611.01603v5, 2017