Seq2seq-Attention Question Answering Model

Wenqi Hou (wenqihou), Yun Nie (yunn)

- **Abstract**:

A sequence-to-sequence attention reading comprehension model was implemented to fulfill Question Answering task defined in Stanford Question Answering Dataset (SQuAD). The basic structure was bidirectional LSTM (BiLSTM) encodings with attention mechanism as well as BiLSTM decoding. Several adjustments such as dropout, learning rate decay, and gradients clipping were used. Finally, the model achieved 57.8% F1 score and 47.5% Exact Match (EM) ratio on validation set; and 49.1% F1 and 35.9% EM on private test set. Future work concerns improvement on preventing overfitting while adding hidden layers.

- **Introduction**

Question Answering (QA) machines are expecting strong increases in daily use now and in near future. One particular task concerns reading comprehension: generate answer to a question by locating a span in some given context paragraph (Fig.1). In past researches of reading comprehension, available datasets were manually labelled and restricted in sizes. With the launch of Stanford Question Answering Dataset (SQuAD), models can be much better validated and tested. In this project, we utilized SQuAD to build a sequence-to-sequence attention based network for question answering. The intuition behind such attention mechanism is that the model could be trained to recognize the difference between two context encodings, one with question attention and another without; the different part is likely to be the answer, where the question pays more attention to.

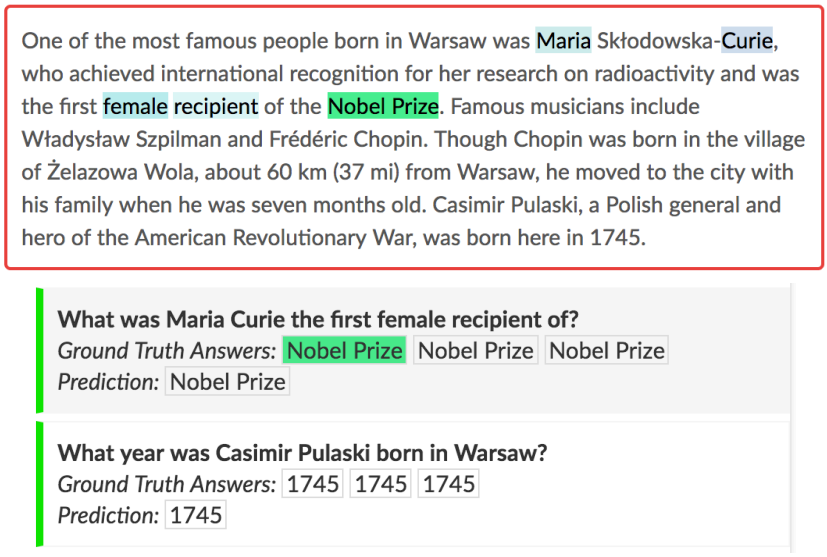

Fig.1 SQuAD question-answer example

- **Approach**:

The basic structure of the model is a network of two encoders and a decoder, all implemented in bidirectional LSTM's (BiLSTM) with minor variations (Fig. 2).

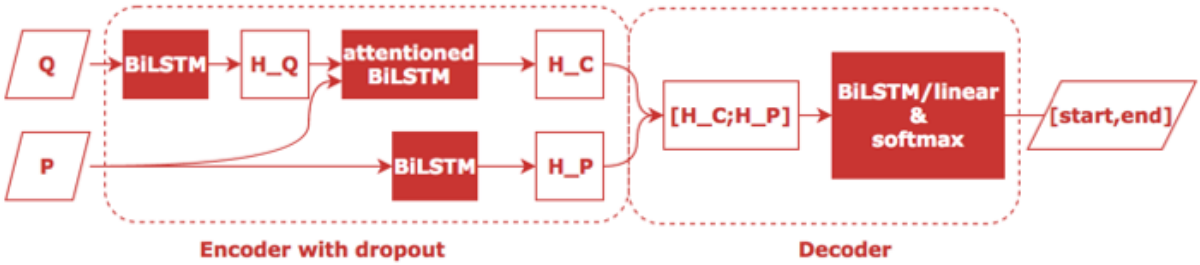

Fig.2 model architecture

First, question Q and context paragraph P are encoded in two independent BiLSTM's which produce corresponding hidden states at each word position as H_P and H_Q. Then the encoded question and paragraph matrices H_Q and H_P are put in to another encoder with sequence to sequence attention (Fig. 3). For each hidden state vector h_Q in H_Q, calculate its attention score over all hidden states in H_P, which we used simple dot product here. Then the score matrix of each question position over the whole paragraph is multiplied by H_P. The product is sent into another layer of LSTM to generate the weighted context H_C under this specific question. H_C is concatenated with H_P into a larger state matrix that contain information about which parts are strongly focused and which are not.

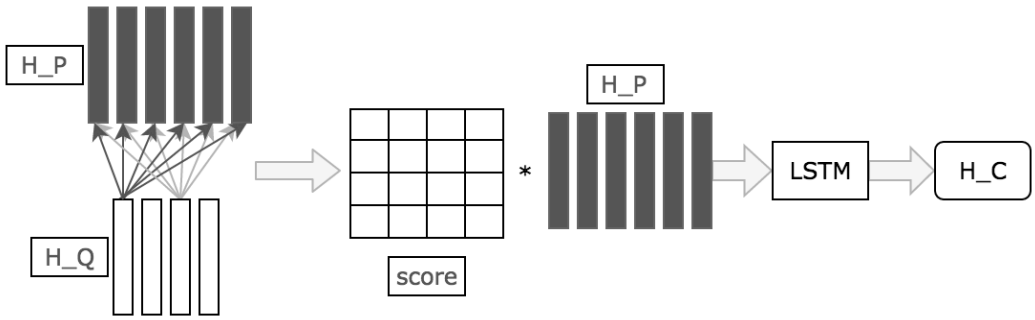

Fig.3 attention encoding

The next step is to feed the matrix [H_C;H_P] into encoder, which consists of two separate bidirectional LSTM networks, one for start index and the other for end index. These gives two output vectors a_s and a_e, where largest element of each is the predicted index. Finally, use softmax activation and cross-entropy loss to arrive at the terminal. Train this model for long enough iterations with several adjustments and regularization, which are discussed in the next part.

- **Implementation**:

1) dataset and task definition

The public dataset of SQuAD is split into 95% training and 5% validation (development) set on local, with 81,381 and 4,284 examples separately. The test set on leaderboard is kept private by SQuAD with unknown number of examples.

We transform original words into word vectors using GloVe with embedding size 100. Since later we discovered that some words as ground truths actually didn't exist in vocabulary list and thus resulted in <unk> "unknown" tags, we decide to restore prediction answer from original context paragraph rather than looking up vocabulary list.

When reading context paragraphs, we cut down contents which are beyond output size of 300, the length that most of our examples fall in (Fig. 4).

Evaluation metrics are i) F1, which is harmonious mean of prediction precision and recall, based on overall coverage of the answer sequence over ground truth, ii) EM (exact match), which is stricter than F1 in that it only accepts predictions that are 100% same as true answers, and iii) validation loss, which is the sum of two cross-entropy losses of start and end predictions. As an expectation, EM should be lower than F1, and validation results are likely to be inferior to training results, both in accuracy and loss.

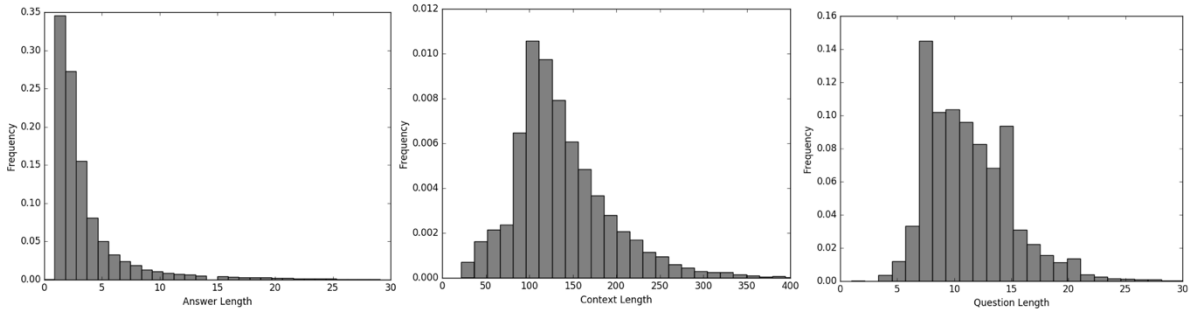

Fig.4 sequence length distribution

2) model with linear decoder

At first, we only used two simple linear functions as our decoder. Although it was learning gradually, it did not catch much information and never broke through 20% training F1 and 10% validation F1 lines (Fig. 5). Yet it still over-fitted the training set, in contrast with validation performance.

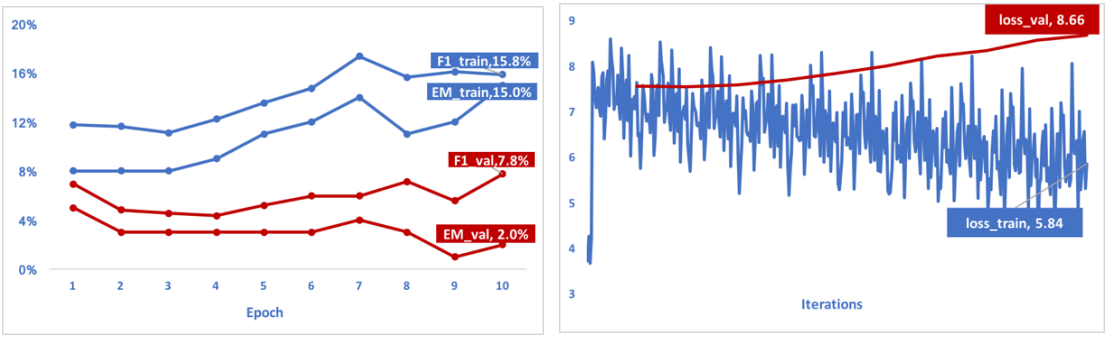

Fig.5 linear decoder model

3) model with LSTM decoder

Replacing linear decoder with a new one under recurrent neural network settings proved to be a big improvement lines (Fig.6), and adding dropout regularization in all encoding functions made such improvement even greater. Without dropout, our model suffered a lot from overfitting: validation loss only decreased a little in the very beginning and kept on increasing until becoming more than doubled of training loss; similarly, validation set only achieved less than half of training F1 and EM ratios, with almost 30% gaps. After dropout was used, the gaps dropped to below 20% and we could see clear decreasing trend in validation loss, which later on did not increase sharply and kept a relatively moderate distance from training loss. Clearly, overfitting was relieved to some extent, but it still existed. The final version of this model achieved 57.8% F1 score on validation dataset and 49.1% on leaderboard test set.

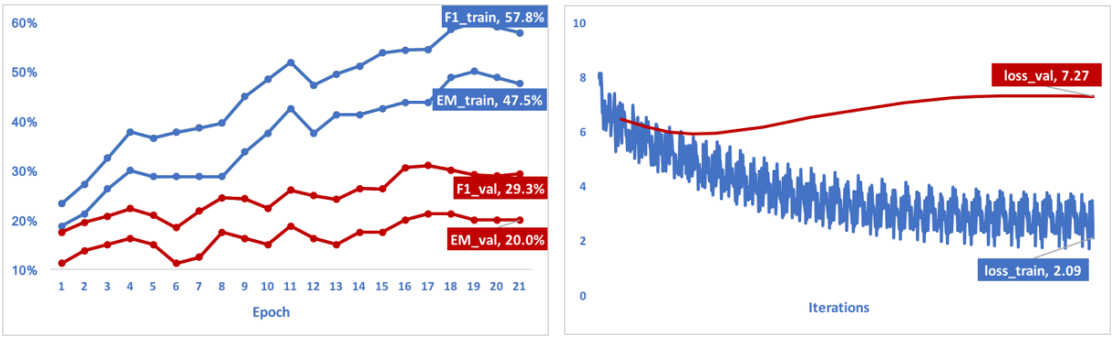

Fig.6 RNN decoder model

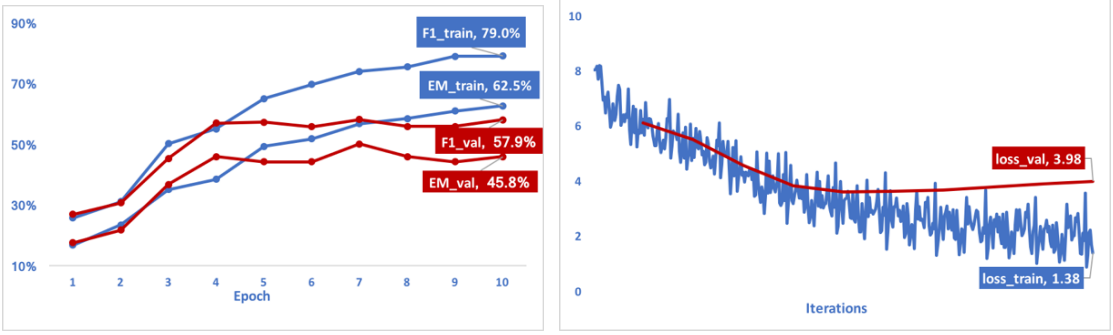

Fig.7 RNN decoder model with dropout

4) parameter tuning

Learning rate: For all implementations, we started from a learning rate of 0.01; however, it rarely survived due to gradients explosion, and even gradients clipping didn't fully help. A promising alternative had been 0.001, with exponential decay of rate 0.8 (Fig.7). This ensured that training loss could decrease in a smoother manner, and would not easily bounced large around local minimum or even diverge away.

State size: This refers to the number of hidden states in LSTM networks. We used the same state size for all LSTM's, and found size of 200 gave much better performance than with size of 100, on the sacrifice of slower learning and running time.

Output size: 300, as discussed before.

Batch size: Default size is 10 and we increased it to 40 to fully utilize GPU. This proved to save much of running time per epoch.

Model performances after 10 epochs are summarized in Table 1:

Table 1 model performances

		training			validation		
decoder	dropout rate	loss	F1 (%)	EM (%)	loss	F1 (%)	EM (%)
linear	-	5.84	15.8	15	8.66	7.8	2
LSTM	-	2.09	59.8	59	7.27	29.1	20
	0.15	3.22	67.2	50.8	4.92	45.5	34.2
	0.2	1.38	79	62.5	3.98	60.9	45.8
	0.3	3.39	62.9	45.8	5.52	34.9	24.2

5) running time analysis

The number of parameters of the model has a major influence on the training time of each epoch. When the size of hidden states increases from 100 to 200, the total number of parameters for the same model increases from 900,000 to 3,000,000, which makes the training time for one epoch twice as before.

Bigger batch size leads to faster training and answering time. The size of batches indicates the degree of parallelism of model. We used larger batch size to fully utilize GPU. But using too large batches will cause out of memory errors.

Dropout and learning rate also affect running time, but not as significantly as batch size and number of parameters. Dropout is used as a means of preventing overfitting, so it causes the model to learn slower but also makes the knowledge learned more universal applicable. Learning rate decides how fast we change the parameters according to the gradients, so it affects converging time.

6) performance analysis

Our model can answer reading comprehension questions with reasonable accuracy (57.8% F1 on validation dataset), but the model still has some overfitting. We can address this with other regularization methods, such as adding norm terms to the gradient descent process).

After examining the answer produced by the model, we discover that our model is still not subtle enough. For instance, when asked about a date, our model can realize that the answer should be in the format of a date. But it sometimes just picks an arbitrary date in the context but not the correct date. One way to address this problem is to feed the encoded representation to decoder multiple time, so that the decoder can output more sensible results.

Also, dealing with a complex task as this, we may need more layers of LSTMs and attention to fully understand the context and to produce more accurate results.

- **Conclusion**:

The most valuable thing we learned during the past month is how we approach a state-of-the-art research problem. To solve a research tasks, we need to first get familiar with the setting and environment, quickly build a working baseline model, and improve the performance with more complex modifications. Also, tuning the hyper-parameters can make a big difference on the performance of the model. We should design the logging formats as early as possible and methodologically analyze how performance changes with different hyper-parameters.

In the future, we can implement more regularization methods to prevent overfitting, and add more layers of LSTMs to build more complex models.