Project 2 Seq-to-Seq Question Answering Report

Chenkuan Liu
Natural Language Processing
UT Austin
Apr 2021

1 Introduction

This project constructs a sequence to sequence model for question answering on Geoquery data. The inputs are geographically related questions, while the outputs are query languages which will be fed into a knowledge base to retrieve the answers. The way to build this model is to use LSTM for both encoder and decoder. However, using LSTM only will not lead to decent performance. To improve the model, we will employ attention mechanism for the decoder so that it will learn to look at specific parts of the input when generating the output query. During training, we will apply teacher forcing to let the model better learn the data. Changing the teacher forcing rate will also lead to different performance. When decoding on a given input, we can apply beam search so that the model will give more reasonable output instead of the greediest one. All these techniques on model construction, learning and decoding, as well as their performance, will be demonstrated in the following sections.

2 LSTM only

When we only use LSTM, the encoder will be similar to the one in sentiment classification. The difference is that, at here, since the input tokens are specified for geographically-related questions, we will not use GloVe but initialize our own embedding and make updates on it during training. The last hidden state and cell state of the encoder LSTM are fed into the decoder. The decoder will start with an empty token, and generate one output token at each step until the END token is generated or the max length limit is reached. Teacher forcing is applied for decoder during training so that the correct token is fed into the network at each stage.

The training set output contains approximately 150 tokens. I applied unidirectional LSTM here and initialized both the embedding dimension and the hidden dimension as 200. The word embedding size is $(\text{number of tokens} + 1, \text{embedding dimension})$, in which the "+1" is used for all unknown tokens that might be encountered during dev stage. The embedding for unknown tokens is a zero column vector. Inside the decoder, each stage will generate a 200 dimension vector. It will go through two FC layers. The size of the first FC layer is still 200, while the size of the second FC layer equals to the total number of output tokens. Then the logsoftmax function is used and we take the token with the highest log probability as the generated token.

The loss for each example is set to be the sum of negative log-likelihood over the length of the gold output divided by the length itself. In this way, the loss for longer outputs can be balanced. The optimizer is Adam and learning rate is 0.001. During training, I am experimenting with both full teacher forcing and 0.8 teacher forcing rate (in which each decoder stage has 0.8 probability to employ teacher forcing and 0.2 probability to feed the previously generated token). After running for 5 epochs, the performance on dev set is demonstrated below.

Table 1: Dev Performance, LSTM only

Teacher forcing rate	Exact match (%)	Token accuracy (%)	Denotation match (%)
1	5.8	64.8	7.5
0.8	1.7	41.1	1.7

We can see that, with full teacher forcing, the token-level accuracy is about 65% and the denotation accuracy is below 10%. The model is severely underfitting, and so changing teacher forcing rate does not bring any improvements to the performance. To improve the model, we will incorporate attention mechanism so that it will learn to generate tokens with respective attentions to certain words inside the input.

3 LSTM + Attention

To add attention inside, I am using the dot product function so that $e_{ij} = f(\overline{h}_i, h_j) = \overline{h}_i \cdot h_j$, $\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{j'} exp(e_{ij'})}$, and $c_i = \sum_j \alpha_{ij} h_j$. The attention is computed after RNN cell so that c_i and $\overline{h}_i$ are concatenated together as the output of the ith stage. In this way, we will have a 400 dimensional vector as our output for each stage. Similar to the previous part, we will feed it into two FC layers. The first FC layer has size 200, and the second FC layer has size equal to the toal number of output tokens. All other settings are the same as the previous section. After running for 5 epochs, the performance on dev set is demonstrated below.

Table 2: Dev Performance, LSTM + Attention

Teacher forcing rate	Exact match (%)	Token accuracy (%)	Denotation match (%)
1	38.3	72.6	42.5
0.8	38.3	77.0	**45.0**

We see that attention mechanism brings significant improvements to the model. Since the model becomes better trained, changing teacher forcing rate to 0.8 brings more flexibility during learning and actually improves the performance by a few percentage. The model with 0.8 teacher forcing rate and 45.0 dev set denotation match is used for testing and the generated file is written into `geo_test_output.tsv`.

4 Extension: Beam Search

After the model is sufficiently trained, we can employ beam search for output decoding. There are two benefits of beam search: it will output a more reasonable sequence instead of the greediest choice, and it will choose other candidates inside if the sequence with highest probability does not compile. In particular, the sequence to sequence model decoding has some special characteristics: since the log probability is always negative, a sentence with lower probability at the beginning will not increase its chance later on. Therefore, when an output with END token is generated, it will be kept inside the beam. The beam search will stop if either all choices inside the beam have generated END token or the output length limit is reached.

At here, we only test the beam search decoding on LSTM + Attention model. To better compare the results, I modified the `evaluate` function inside `lf_evaluator.py` so that the beam search decoding results are generated from the same trained model used for Table 2. The beam width are set to be 2, 4 and 8. The teacher forcing rates are set to be 1.0 and 0.8. After running for 5 epochs, the performance is demonstrated below. To better visualize the change of performance, I also list the results from Table 2 as beam width 1.

Table 3: 1.0 teacher forcing rate, LSTM + Attention + Beam Search

Beam width	Exact match (%)	Token accuracy (%)	Denotation match (%)
1	38.3	72.6	42.5 (51/120)
2	41.7	73.5	45.8 (55/120)
4	45.8	75.0	50.0 (60/120)
8	45.0	72.9	48.3 (58/120)

Table 4: 0.8 teacher forcing rate, LSTM + Attention + Beam Search

Beam width	Exact match (%)	Token accuracy (%)	Denotation match (%)
1	38.3	77.0	45.0 (54/120)
2	40.0	75.3	47.5 (57/120)
4	41.7	76.5	49.2 (59/120)
8	44.2	74.8	51.7 (62/120)

We see that, for both cases, the performance is improved when we increase beam width from 1 (the original greedy version) to 2 and from 2 to 4. When we increase beam width from 4 to 8, there are decreases in certain metrics: all values are decreased a little bit in Table 3 with full teacher forcing; the token level accuracy is decreased a little bit while exact and denotation matches increase in Table 4 with 0.8 teacher forcing rate.

The performance will slightly vary depending on random initialization inside PyTorch. However, in both cases we can see more than 5% increase in exact match and denotation match, and the denotation match can get close or surpass 50% with certain beam width. As a result, we can infer that a good choice of beam width in this case would be between 2 and 8. Beam width larger than this range will cost more computation time and might also decrease the performance (though it is generally still higher than greedy decoding).

5 Summary

From the methods and modifications implemented above, we can observe that using LSTM only does not bring decent results. Attention mechanism is necessary and it significantly improves our sequence to sequence model performance. Changing teacher forcing rate during training and applying beam search during decoding can also bring certain improvements.

To further improve the performance, there are some additional approaches that can be experimented. One way is architecture-wise: we can change to bidirectional LSTM, adding dropout and possibly more FC layers, or applying different attention function to $f(\overline{h}_i, h_j)$. However, as the model becomes more complex, it also requires more epochs and time to train in order to sufficiently update all the parameters. To deal with this, we can also apply batching here to speed up training time. Another way is training-wise: we can further tune the learning rate, increase epoch number, or randomly shuffle the training data inside each epoch. We can also experiment with different teacher forcing rates so that the model starts with full teacher forcing and slowly decays the rate later on. With proper adjustments and experimentation, it is possible to reach 60% denotation match accuracy or even higher on development set.