Machine Learning in Natural Language Processing: Assignment 3

Shifei Chen

1 Word embeddings

The results from embedding calculations were not always understandable either grammatically or semantically. In fact the only calculation that made sense to me was the famous "king - man + woman", where "queen" was the second in the candidate results. All other formulas I tried were not understandable to me, such as "-john" would give me mystery abberations like "afp02", "js03" or "mo96". They could also give me results that were not semantically correct. For example "europe - britain + france", the first two results were "france" and "europe" even though they were already parts of the formula. Idealy both of them should not appear in the results since they were the minuend and the subrahend.

Also, increasing the vocalbulary size or the number of dimensions made a very small change in the results, usually just the order of the kth closest results. All of the candidates remained the same.

2 POS Tagging

The comments in the `forward()` function laid a very clear structure for futher implementing the RNN forward pass. Two things are worth to mention here. The first is the call to the RNN layer. It was confusing to me at first as there was no reference nor documents about the `RNNBase` class in `PyTorch`, but Google did take me to the source code[1] and from there I realized that `RNNBase` was acutally the base class of `nn.RNN`. Hence I can get updates of both the input and the hidden layer from the return tuple of that. Another small thing was collapsing the first two dimensions of the input matrix. Just leave -1 if you have no idea about the new size and `PyTorch` will figure it out from other dimensions itself.[2]

Bidirectionality creates one more layer than the regular RNN as its name suggest. "For each 'left-to-right' layer there is one 'right-to-left' layer." therefore it was natural to double the number of layers by multiplying `rnn_layer_number` with 2. Also the existing code of the hidden layer in the regular RNN cell type inspired me of how to initialize the hidden layer for a LSTM cell. We have one more layer tier of cell states so just bundle it with the original hidden state into a tuple to make the new `hidden_layer`, as the `nn.LSTM` expects in its document. Since our `rnn_layer` was actually the base class of `nn.LSTM`, they share the same requirements for input parameters and everything worked just fine.

Here a typical RNN tagger has around 17000 trainable parameters. If it was equipped with LSTM cells the number of trainable parameters would grow to more than 62000, less than four times more. Also as expected, turning on bidirectional will double the number of trainable parameters.

3 Hyperparameter Tunning

I have run 20 different sets of hyperparameters. Some of the hyperparameters had more or less effect on the final accuracy and loss score, but generally most of them made subtle changes. All of the data is show in 1. My learing rate was 0.01, optimizer was `optim.Adam`, loss function was `nn.CrossEntropyLoss` and the activation function was `nn.LogSoftmax`. Most of the time I ran 12 epochs but there were times that I need to run a few epochs more to get a convey result as noted in the table. And last but not least, I tried to maintain a consistent number of trainable parameters, which was around 120000.

First of all `LSTM`s performed slightly better than regular `RNN_TANH`s. We can see in the table that both `RNN_TANH` and `LSTM` can achieve 0.905 of accuracy, but the best results of a `LSTM` tagger could go higher to 0.914 where `RNN_TANH` could not. Dropout rate improved the results by a little bit, but I have to run a little bit longer to see that. Bidirectionality had a great effect on the learing curve. In fact in all experiments where I have turned off bidirectionality, the learning speed would also been slowed down a lot. I had to made up that by

[1] `https://github.com/pytorch/pytorch/blob/master/torch/nn/modules/rnn.py`

[2] `https://pytorch.org/docs/stable/tensors.html?highlight=view#torch.Tensor.view`

Table 1: Results from Validation Runs

Size	Depth	Bidirectionality	Cell Type	Dropout	Loss	Acc	Note
100	1	True	LSTM	0	0.001250	0.9041	
30	5	True	LSTM	0	0.001302	0.9056	
43	3	True	LSTM	0.1	0.001203	0.9114	
100	1	True	LSTM	0.1	0.001220	0.9075	
30	5	True	LSTM	0.1	0.001302	0.9040	
50	8	True	RNN_TANH	0	0.001412	0.8935	
50	8	True	RNN_TANH	0.1	0.001438	0.8914	
120	2	True	RNN_TANH	0.1	0.001252	0.9042	
120	2	True	RNN_TANH	0	0.001275	0.9043	
14	100	True	RNN_TANH	0	0.009320	0.2538	
66	5	True	RNN_TANH	0	0.001259	0.9052	
44	8	False	LSTM	0.1	0.002560	0.8085	Extra epoch to 24
25	8	True	LSTM	0.1	0.001618	0.8852	Extra epoch to 24
40	10	False	LSTM	0	0.004014	0.6891	Extra epoch to 20
90	3	True	RNN_TANH	0.1	0.001234	0.9060	
66	5	True	RNN_TANH	0.1	0.001239	0.9083	
88	8	False	RNN_TANH	0.1	0.005086	0.5804	Extra epoch to 30
115	5	False	RNN_TANH	0.1	0.001588	0.8751	Extra epoch to 24
150	3	False	RNN_TANH	0.1	0.001532	0.8793	Extra epoch to 24
43	3	True	LSTM	0.3	0.001161	0.9143	Extra epoch to 24
43	3	True	LSTM	0.2	0.001188	0.9128	Extra epoch to 24

doubling the number of iterations but even though they sometimes still lagged behind their bidirectional siblings in accuracy. Finally, RNN size and RNN depth were the two worth to mention among these hyperparameters. A large size or a deep layer of RNN didn't prove to be the best efficient one in all case. You can see from the chart that the best RNN of two cell types were the ones with a mediocore size and an adequate depth (within a fixed number of trainable parameters). Also I have noticed that the deeper the RNN was the longer it took to train the network. For this particular POS tagging thing I think it doesn't make a lot of sense using a large size of RNN as the POS tags usually have connections to just a few words before. But for something like speech recognition I believe we should train the recognizer with a RNN whose size is larger than 100.

The program gave a confusion matrix after each run. For most of the times the tagger performed well in all tags and the precision and recall scores were close to each other, meaning that it didn't produce more false negatives nor false positives. However for class `INTJ`, `PROPN SYM` and `X` the presicion and recall dropped a bit than other classes. I can understand the poor performance on `PROPN` and `X` as `PROPN`s are difficult to tackle anyway and `X`s are usually not possible to analyze. For `INTJ`s and `SYM`s I suspect that the RNN tagger could not recognize them very well because they are usually standalone classes with no connections to the context. A punctuation or a word like "oh", "psst" can appear anywhere in the text and usually doesn't render the text impossbile to read if being removed. It can be a good idea to pre-process the text before feeding it into the RNN tagger, especially we could use regex to detect and remove punctuations effectively as they have a very clear pattern.

4 Peephole

The implementation was similar to the basic RNN. There was some hints inside the example code and everything is already well documented (except the comment suggesting adding a peephole connection to the proposed cell state gate. In fact nothing needs to be changed there).

On the other hand the performances was not good at all. I got 0.4-0.5 of accuracy with different combinations of hyperparameters and that was way worse than RNNs without peepholes. There were two things I need to mention here, one is that it seems like that the number of trainable parameters with peephole cells was no longer related to the depth of RNN layers any more. It was only affected by the size of the RNN. The other

thing was I could not make the code work under bidirectional mode. Anyway even comparing with a single directional `LSTM` or `RNN_TANH` tagger, the peephole tagger still underperformed. (Breuel, 2015) ran a bunch of benchmarks and said that "Peephole connections never resulted in any improved performance" in his conclusion.[1] Also, (Gers, Schraudolph & Schmidhuber, 2002) mentioned themselves that peephole connections is "a promising approach for tasks that require the accurate measurement or generation of time intervals."[2] For our RNN tagger task, peephole connection was not a better choice.

References

[1] Thomas M Breuel. "Benchmarking of LSTM networks". In: *arXiv preprint arXiv:1508.02774* (2015).

[2] Felix A Gers, Nicol N Schraudolph, and Jürgen Schmidhuber. "Learning precise timing with LSTM recurrent networks". In: *Journal of machine learning research* 3.Aug (2002), pp. 115–143.