Project Report

Author: Tianlei Zhu
Datum: 21.07.2017

1.1 Abstract

In this report I will mainly talk about my implementation of this project by using LSTM and some crucial points, that why it performs better than other implementations. Also I will briefly introduce the process of implementation, main problems and some confusions.

2.1 Technology introduction

In this task I built up my neuron network by using one input layer, one hidden layer (LSTM), and one output layer. The structure is simple but performs good.

2.1.1 Input layer

For constructing the input layer I use a 4 X N matrix as my input value. The matrix here means, we use 4 prefixed tokens (sequentially) and each of them with a length N, which is the length of the one-hot vector. If the query with a prefix length which is smaller than 4, I used padding (talked later) to pad it to 4.

2.1.2 LSTM layer

When I began coding, the first idea come upon my mind is using RNN (Recurrent Neuron Network), which is typically designed for this situation. But during the implementation, I found that the drawbacks of RNN, which is called *Vanishing Gradient Problem*. This is a big issue that reduce the accuracy. Because during the learning process of RNN, it is actually forgetting at the same time, so when I use RNN to train the model, the earlier information which we presented to it may lose. So I changed to LSTM (Long Short Term Memory) as my hidden layer, LSTM has a better 'memory' than RNN so it can remember a sequence of inputs during the whole training step. However, it risks overfitting problem, so here what I actually did is combined with a dropout layer (Figure 1), which is used to drop some inputs and outputs randomly, I choose 0.8 as the keeprate by exhaustive tries and found it's the optimal choice (actually input = 1 and output = 0.5 is the best choice, but that conflicts the purposes of using a such layer and may cause overfitting). The training result (Figure 2 and Figure 3) below shows the performance difference between RNN and LSTM in the same architecture, the total loss is significantly reduced by using LSTM. But LSTM is time consuming because of its complexity, anyway we choose LSTM.

2.1.3 Output layer

I didn't change the output layer so it remains the same as what is give, it uses 'softmax', which is a generalization of logistic function that "squashes"(maps) a K-dimensional vector z of arbitrary real values to a K-dimensional vector σ(z) of real values in the range (0, 1) that add up to 1. It uses 'adam' for regression.

2.1.4 Configurations of training

I use 3 epochs for training, because after the second epoch, the 'total loss' is not significantly reduced during the training, and the size of batches is 256.

```
self.net = tflearn.lstm(self.net, 128,dropout=(0.8,0.8))
```

Figure 1

```
Training Step: 20072  | total loss: 1.40611 | time: 88.325s
| Adam | epoch: 003 | loss: 1.40611 - acc: 0.5632 -- iter: 1712640/1712862
Training Step: 20073  | total loss: 1.39268 | time: 88.340s
| Adam | epoch: 003 | loss: 1.39268 - acc: 0.5670 -- iter: 1712862/1712862
```

Figure 2(with RNN)

```
Training Step: 20072  | total loss: 1.33311 | time: 173.922s
| Adam | epoch: 003 | loss: 1.33311 - acc: 0.5825 -- iter: 1712640/1712862
Training Step: 20073  | total loss: 1.33013 | time: 173.948s
| Adam | epoch: 003 | loss: 1.33013 - acc: 0.5817 -- iter: 1712862/1712862
```

Figure 3(with LSTM)

2.2 Pre-processing, Predict and Result truncation

2.2.1 Pre-processing (padding)

As I mentioned before, I use last 4 tokens of prefix to predict the following tokens, when the prefix is shorter than 4, I just simply add some N*0 vectors before it. For example, the incoming prefix contains 2 tokens, so I add 2 zero vectors with length N before these 2 tokens, pay greatly attention to the sequence, and then wrap it as a list and feed it to the model.

Why last 4 tokens? It's obvious, that more tokens are used to predict, the higher accuracy will be achieved (see figure 4 and 5, which is implemented by using 4 tokens and 2 tokens, the performance decreases significantly). However, some data sources even don't have so many tokens before hole creating (I thought the shortest one only contains 6 tokens), so here I choose 4 as my input size.

2.2.2 Predict

Because the size of removed tokens is unknown, so I predict a sequence of tokens which is restricted to some stop criteria (introducing soon), In the predict step, we pick the token with the highest probability in the output predict list. We append this token not only to the return tokens but only to the prefix so that in the next iteration this token can be used to predict its next token by combining with the previous tokens

2.2.3 When to stop?

I use 2 stop-conditions (concluded form observing the probability of each token) to let the prediction stop at a proper time. The first condition is when the last token's probability (initialed with 0) bigger than 0.8 and the current token's condition bigger than 0.5 (showed in figure 6), which shows a very likely combination so I suggest to continue predicting. The second condition is when the current token's probability minus last token's probability and the result is greater than 0.1 (showed in figure 7), that means the current token is very likely to be append to the last token, so the prediction will be continued.

2.2.4 Result truncation

We finished the prediction and get a list of tokens which should be processed by the suffix, we search through the list and find the first token which is equals to the first token of suffix, Attention! Sometimes the first token in the list is coincidentally the first token in suffix, so we should add a condition that index != 0. If no hits, we just return what we predicted.

```
Accuracy: 53 correct vs. 147 incorrect = 0.265
```

Figure 4(4 Tokens)

```
Accuracy: 29 correct vs. 171 incorrect = 0.145
```

Figure 5(2 Tokens)

```
Last token has a probability:0
This token has a probability:0.9994938373565674
Last token has a probability:0.9994938373565674
This token has a probability:0.5467467308044434
Last token has a probability:0.5467467308044434
This token has a probability:0.34858912229537964
completion words:[{'type': 'Identifier', 'value': 'ID'}, {'type': 'Punctuator', 'value': '='}]:
expected words:[{'type': 'Identifier', 'value': 'ID'}, {'type': 'Punctuator', 'value': '='}]:
```

Figure 6

```
Last token has a probability:0
This token has a probability:0.21660085022449493
Last token has a probability:0.21660085022449493
This token has a probability:0.6677716970443726
Last token has a probability:0.6677716970443726
This token has a probability:0.46707749366760254
completion words:[{'type': 'Punctuator', 'value': ')'}]:
expected words:[{'type': 'Punctuator', 'value': ')'}]:
```

Figure 7

3.1 Summary

After several tests, this program can reach an accuracy with one token ~0.58, two tokens ~0.38, three tokens ~0.26, four tokens ~0.21 . And there are about 800 json files are used to train and 200 json files are used to test.

In my implementation there are still some problems and I am trying to figure it out. The first problem is this prediction strategy seems a little bit ‘conservative’, that means it prefer to predict a short tokenlist than a long one, but actually in real life it is so, the more you predict, the more errors you may make. The second problem is by using the first word in suffix to truncate the result, it may match the item in the return list a little bit early, for example what we predict is [‘a’, ‘<’, ‘b’, ’?’, ‘a’, ’:’, ’b’] and the suffix is [‘b’, …..], so this algorithm will match the first ‘b’ with index 2, but what we want is the last ‘b’ with index ‘6’.

Sorry I didn’t finish the optional tasks. If I have time in the next coming days I will try it.