Project Report
Analyzing Software using Deep Learning

Haroon Khaleeq
Fachbereich Informatik
July 24, 2017

1. Introduction

In this project, deep neural network has been used to train our model with around 800 javascript programs in the form of sequence of tokens. What it does is basically train the model with these sequence of tokens. After training, this model can predict the next expected token given the previous tokens as input. Based on this prediction we will verify how effective is the network accuracy to predict the expected tokens. This project report will discuss all the important aspects which are closely related to the project. Implemented approach in section 1 describes briefly about implemented RNN(Recurrant Neural Network), LSTM(Long Short Term Memory) and why it has been chosen over other networks. Section 4 will discuss in detail the practical evaluation of the approach including different cases with correct predictions and where are the limitations.

2. Framework Setup

As per the project description, which had been provided, Python version 3.5.2 has been installed. Tensorflow 1.1 (with CPU support only) and TFLearn version 0.3.1 had been used to implement the project which is also same as per project description.
For the final submission, the model is trained with 800 javascript programs from the given dataset. An attempt to train the model with 1000 program files was made but due to the limitation of the system resources the application stopped with 'MemoryError'. During testing, 800 javascript token files were used for training and 200 files for quering the network. Empirical results are discussed in detail later in section 4.

3. Implemented Approach

LSTM (Long Short Term Memory) which is a special kind of Recurrant Neural Network has been used. Let's now discuss at first about why this approach has been used. Recurrant Neural Networks are really helpful in specially those scenarios where input or output is a sequence. Every unit in a sequence has more relevance to its neighbors than to the nodes which are farther away. In conventional Neural Network information from these previous units is discarded away and do not contribute in the network for the next input units. Recurrant Neural Network has this ability that it does not discard this information. Rather, the result from the previous step acts as an input towards the next step in the network along with the current input.

This is very useful in keeping the results from previous steps in the network and using them further in the network. This part is useful in training a network for sequences of input. The dependency distance between different inputs can be very long for which purpose, LSTM (Long short term memory) which is one of RNN approaches are really good in keeping the old information in the network. This is because of the innate nature of LSTM. Since, this is just a project reports so we would not go further in to the detailed working of LSTM.

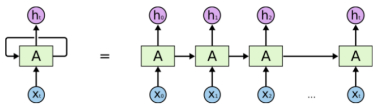

Figure 1. This picture also shows the basic idea behind RNN.(Olah 2015)

3.1 Network Hidden Layers

In the implemented design, 4 hidden layers have been used which are described as following. In the input layer, 4 hot vectors have been given which correspond to the 4 consecutive token sequence in a javascript program. After the input layer, first hidden layer is of LSTM (Long Short Term Memory) which has 128 units for this layer and return sequence has been set to true which means that the output would be 3D Tensor. It is followed by another LSTM layer which also has 128 units and output in 2D Tensor. A fully connected layer has been added after 2 LSTM layers. Fully connected layer takes the input and maps to units which has size equal to 86 which is the unique tokens in the dataset. Softmax is been used as the activation function. Bias and Trainable parameters for this layer is also set to true. In the final hidden layer, the results have been summarized in the regression layer. Adam optimization algorithm has been used. Categorical crossentropy has been used for loss.

3.2 Input Output Representation

For every 4 consecutive token sequence, the next expected token is been set as the output. This is the representation of the input output data, based on which we are training our model. I also tried using more than 4 input token sequence but as the hot vector is very costly in terms of space, my python application was crashed even though my machine has pretty good RAM size which is 8GB. Due to this limitation, I stick to using 4 input token sequence to feed our model. Following code snippet 1 from `prepare_data` in `code_completion_final.py` shows how the input and output pairs haven been generated.

```
1 token_string = self.token_to_string(token)
2 previous_4_token_string = self.one_hot(self.
      token_to_string(token_list[idx - 4]))
3 previous_3_token_string = self.one_hot(self.
      token_to_string(token_list[idx - 3]))
4 previous_2_token_string = self.one_hot(self.
      token_to_string(token_list[idx - 2]))
```

```
previous_1_token_string = self.one_hot(self.
    token_to_string(token_list[idx - 1]))

xs.append([previous_4_token_string,
    previous_3_token_string,
    previous_2_token_string,
    previous_1_token_string])
ys.append(self.one_hot(token_string))
```

Listing 1. Preparing Input Output pairs for the model

4. Empirical Reuslts

In the initial phase, the model was trained with only two consecutive token sequence e.g. a and b being two consecutive token sequence and c as the next expected token sequence i.e. `[a,b] -> [c]`. Network Accuracy at this stage was not that good and measured at around 54%. Later, after analyzing the poor network accuracy, I changed the representation from two input token sequence to four input token sequence i.e. `[a,b,c,d] -> [e]`. This really helped in improving the network accuracy significantly to around 65%. Please note here that one token here is nothing but a json object which has two properties "type" and "value" e.g. `{ "type": "Punctuator", "value": "("}`. Our network was tested with different hole sizes, results of which are like this: for hole size equal to 1, prediction accuracy is about 55%. For hole size equal to 2 and 3, the accuracy measured was about 40% and 29% respectively. Maximum hole size for which the network was tested with, was hole size equal to 8. In this case, the accuracy measured was about 10%.

As discussed previously in the section 3.1, 4 hidden layers have been used in the network. Adding further layers to the network would not only utilize all my machine resources RAM(8GB) and SWAP(8GB), but also crashed the python application. This was one limitation which was been exploited at this point. The model has been trained with 1 epoch. Adding further epochs does not affect the accuracy of the network. 'Softmax' has been used as an activation function. In the same way, 'adam' works best as an optimizer and 'categorical crossentropy' for loss.

Query Method

The designed approach for predicting the next expected token sequence is quite simpe and straigt forward. The `query` method keeps on predicting the best next token until it predicts such a token whose type and value matches with the suffix first token. This means that we keep on storing best found tokens predicted by our model and return this array in the end. There are many cases where our model predicts false positives. In this case, it is very unlikely that query method predicts correctly. To avoid these false positives and infinite loops a maximum boundary is set to 10. We will take a look at few examples where our model predicts correctly and also incorrect predictions where it has limitations.

Example Cases

Following example case 1 shows that our model predicts first token correctly but second prediction was wrong and third predicted token matches the suffix first token so it stopped and returned previous predicted tokens. Second wrong prediction makes this overall incorrect prediction.

```
Suffix
{'type': 'Punctuator', 'value': '.'}
Expected Token
[{'type': 'Punctuator', 'value': '('}, {'type':
    'Keyword', 'value': 'this'}]
Predicted Token
[{'type': 'Punctuator', 'value': '('}, {'type':
    'Identifier', 'value': 'ID'}]
```

Listing 2. Case 1

Here at this example case 2, our model correctly predicts all next 3 tokens. Further, it also predicts the 4th token which matches the suffix so it stopped and returned the previous 3 tokens.

```
Suffix
{'type': 'Keyword', 'value': 'var'}
Expected Token
[{'type': 'Identifier', 'value': 'ID'}, {'type'
    : 'Punctuator', 'value': ')'}, {'type': '
    Punctuator', 'value': '{'}]
Predicted Token
[{'type': 'Identifier', 'value': 'ID'}, {'type'
    : 'Punctuator', 'value': ')'}, {'type': '
    Punctuator', 'value': '{'}]
```

Listing 3. Case 2

In the next example 3, is the limitation of our approach. This is one of the false positives. At the very first step it predicted first token correctly. But it stopped when the next predicted token matches the suffix ','. Ideally the next predicted token should not match the ',' rather a string 'STR'. This happened because of two reasons. Our trained model did not see sufficient examples where there comes a 'STR' after a '(' punctuator. Second reason might be that our model was not managed to keep this information in the network and lost it in between the hidden layers.

```
Suffix
{'type': 'Punctuator', 'value': ','}
Expected Token
[{'type': 'Punctuator', 'value': '('}, {'type':
    'String', 'value': '"STR"'}]
Predicted Token
[{'type': 'Punctuator', 'value': '('}]
```

Listing 4. Case 3

This example 4 shows the correct result. It successfully predict all the expected 4 tokens.

```
Suffix
{'type': 'Punctuator', 'value': '{'}
Expected Token
[{'type': 'Identifier', 'value': 'ID'}, {'type'
    : 'Punctuator', 'value': ','}, {'type': '
    Identifier', 'value': 'ID'}, {'type': '
    Punctuator', 'value': ')'}]
Predicted Token
[{'type': 'Identifier', 'value': 'ID'}, {'type'
    : 'Punctuator', 'value': ','}, {'type': '
    Identifier', 'value': 'ID'}, {'type': '
    Punctuator', 'value': ')'}]
```

Listing 5. Case 4

5. Conclusion

The network's ability to predict missing token sequence is measured at around 65%. However, the accuracy for query method varies from 55% to 10% with hole size equal to 1 and 8 respectively. Training the model with more input output information would definitely increase the accuracy of our model which would definitely require more system resources than on which it is currently been performed.

References

C. Olah. Understanding lstm networks. *http://colah.github.io/posts/2015-08-Understanding-LSTMs/*, Aug. 2015.