UDACITY

PROJECT

Generate TV Scripts

A part of the Deep Learning Nanodegree Foundation Program

PROJECT REVIEW

CODE REVIEW

NOTES

SHARE YOUR ACCOMPLISHMENT!

Meets Specifications

Congratulations you passed this project.
Now you should implement your own RNN from scratch for better understanding. Following links would help you(You can any time take help from mentors)

http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/
https://www.youtube.com/watch?v=cO0a0QYmFm8&index=10&list=PLlJy-eBtNFt6EuMxFYRiNRS07MCWN5UIA

Good Luck for next project.

Required Files and Tests

The project submission contains the project notebook, called "dlnd_tv_script_generation.ipynb".

All the unit tests in project have passed.

Great work, All unit test are working fine.
Unit testing is very good practice to ensure that your code is free from all bugs without getting confused and prevent you from wasting a lot of time while debugging minor things. Unit test also help in improving our code standards. For more details read this(https://cgoldberg.github.io/python-unittest-tutorial/) and I really hope that you will continue to use unit testing in every module that you write to keep it clean and speed up your development and for quality development. Unit testing is highly motivated in industries.

It is not always that if you passed unit test your code is okay, there can be some errors.

Preprocessing

The function `create_lookup_tables` create two dictionaries:

- **Dictionary to go from the words to an id, we'll call vocab_to_int**
- **Dictionary to go from the id to word, we'll call int_to_vocab**

The function `create_lookup_tables` return these dictionaries in the a tuple (vocab_to_int, int_to_vocab)

The function `token_lookup` returns a dict that can correctly tokenizes the provided symbols.

Build the Neural Network

Implemented the `get_inputs` function to create TF Placeholders for the Neural Network with the following placeholders:

- Input text placeholder named "input" using the TF Placeholder name parameter.
- Targets placeholder
- Learning Rate placeholder

The `get_inputs` function return the placeholders in the following the tuple (Input, Targets, LearingRate)

Fantastic! You've perfectly used "None"'s to adequately create a dynamic sized placeholder variables.

get_inputs function is correctly implemented. Seems that you have good grasp of TF Placeholder, I appreciate your efforts.
There is a very good course offered by Stanford University, CS 20SI: Tensorflow for Deep Learning Research. This would help you in further understanding of Tensorflow and its application.

The `get_init_cell` function does the following:

- Stacks one or more BasicLSTMCells in a MultiRNNCell using the RNN size `rnn_size`.
- Initializes Cell State using the MultiRNNCell's `zero_state` function
- The name "initial_state" is applied to the initial state.
- The `get_init_cell` function return the cell and initial state in the following tuple (Cell, InitialState)

Good Work! You've

- Stacked 1 layer, for a more complex behaviour please try to include more layers(2 layers would be sufficient for this project). (Please have a look at https://arxiv.org/abs/1506.02078)
- Cell state initiallized correctly.
- Used the identity function to name the state. (Good Work)

Awesome Job!! initiated cell correctly.
To know more about LSTM and RNN, please follow following links (explained very clearly)

http://karpathy.github.io/2015/05/21/rnn-effectiveness/
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

The function `get_embed` applies embedding to `input_data` and returns embedded sequence.

The function `build_rnn` does the following:

- Builds the RNN using the `tf.nn.dynamic_rnn`.
- Applies the name "final_state" to the final state.
- Returns the outputs and final_state state in the following tuple (Outputs, FinalState)

The `build_nn` function does the following in order:

- Apply embedding to `input_data` using `get_embed` function.
- Build RNN using cell using `build_rnn` function.
- Apply a fully connected layer with a linear activation and `vocab_size` as the number of outputs.
- Return the logits and final state in the following tuple (Logits, FinalState)

Good Work!! Taken care off activation function in fully connected layer function.

More on embedding layer:

```
It might be necessary to add an Embedding layer just before the RNN to embed the words into a smaller dimensional space. Sin
ce the LSTM operates linearly over its input X, then linearly embedding the 1-of-K words to some embedding dimension D first
 corresponds to the original case you tried to get working here, except the matrix is factored through a rank D-dimensional
 bottleneck. If that makes sense.

Also, it is common to keep track of frequency of all words and discard words that appear, say, less than 5 times in the data
set.
(Words from Andrej karpathy)
```

The `get_batches` function create batches of input and targets using `int_text`. The batches should be a Numpy array of tuples. Each tuple is (batch of input, batch of target).

- The first element in the tuple is a single batch of input with the shape [batch size, sequence length]
- The second element in the tuple is a single batch of targets with the shape [batch size, sequence length]

Good work!!

Code for good use of Numpy Library and more pythonic.

```
n_batches = int(len(int_text) / (batch_size * seq_length))

# Drop the last few characters to make only full batches
xdata = np.array(int_text[: n_batches * batch_size * seq_length])
ydata = np.array(int_text[1: n_batches * batch_size * seq_length + 1])
ydata[-1] = xdata[0]

x_batches = np.split(xdata.reshape(batch_size, -1), n_batches, 1)
y_batches = np.split(ydata.reshape(batch_size, -1), n_batches, 1)

#print(np.array(list(zip(x_batches, y_batches))))

return np.array(list(zip(x_batches, y_batches)))
```

If there is still some doubt, this video will definitely help you better understand how batching operates.

Neural Network Training

- Enough epochs to get near a minimum in the training loss, no real upper limit on this. Just need to make sure the training loss is low and not improving much with more training.
- Batch size is large enough to train efficiently, but small enough to fit the data in memory. No real “best” value here, depends on GPU memory usually.
- Size of the RNN cells (number of units in the hidden layers) is large enough to fit the data well. Again, no real “best” value.
- The sequence length (seq_length) here should be about the size of the length of sentences you want to generate. Should match the structure of the data.
  The learning rate shouldn’t be too large because the training algorithm won’t converge. But needs to be large enough that training doesn’t take forever.
  Set show_every_n_batches to the number of batches the neural network should print progress.

Some Tips

The values of hyper parameters should be power of 2. Tensorflow optimizes our computation if we do so.

The `seq_length` should be kept similar to average size of sentence or say according to the structure of our data, this would help in getting better results. Link: http://stats.stackexchange.com/questions/158834/what-is-a-feasible-sequence-length-for-an-rnn-to-model (for this project 12-15 is a desirable range)

Select a learning rate such the model converges well with minimum oscillations/spikes (where the training loss mostly decreases and doesn't "jitter around" in value).

Some links that would help in fine tune model.
http://neupy.com/2016/12/17/hyperparameter_optimization_for_neural_networks.html

The project gets a loss less than 1.0

Generate TV Script

"input:0", "initial_state:0", "final_state:0", and "probs:0" are all returned by `get_tensor_by_name`, in that order, and in a tuple

The `pick_word` function predicts the next word correctly.

Please try to use slight randomness when choosing the next word.
If you don’t use randomness, the predictions will fall into a loop of the same words.

Suggestion

- Your `pick_word` function always returns back the most probably word as you are taking word with maximum probability(`np.argmax`). so when you give a word to your model to generate a script it essentially generating more or less same words following the previous word. considering this problem we should use some randomness while choosing next word, for example, you could pick a word randomly based on it's probability distribution. Say I've a total vocab of 3 words, A, B and C. Now say the model predicts that the next word should be A with 0.5, B with 0.3 and C with 0.2. Then when I pick a word, I should pick A 50% of the time, B 30% of the time and C 20% of the time. Now when your model is trained really well with a lot of data, what this will imply is that it'll result in the model giving out slightly different scripts each time, making it look very human.

Helpful Link:
https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.choice.html

The generated script looks similar to the TV script in the dataset.

It doesn't have to be grammatically correct or make sense.

You have some mighty fine generated text here! If you are interested in doing another project about generating text, I encourage you to take a look at Project Gutenberg: http://www.gutenberg.org. They have thousands of free books that you can use to build a recurrent neural network with much more training data. You will also notice, as you use more training data, the quality of your generated text will improve!

To make script that make more sense. (More involved)

- We need a good amount of data and large network to learn it.
- You an follow text generation by GAN, this can also lead to good results. (https://www.reddit.com/r/MachineLearning/comments/40ldq6/generative_adversarial_networks_for_text/)

DOWNLOAD PROJECT

RETURN TO PATH

Rate this review