UDACITY

PROJECT

Generate TV Scripts

A part of the Deep Learning Nanodegree Foundation Program

PROJECT REVIEW

CODE REVIEW

NOTES

SHARE YOUR ACCOMPLISHMENT!

Meets Specifications

I really enjoyed reviewing your project. It is truly a remarkable sight that your network is able to generate the script as shown. It is amazing indeed.

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.

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.

All the 10 symbols are taken as key and the tokens of those symbols are taken as values into the dictionary.

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)

All the three placeholders have been defined correctly.

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)

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

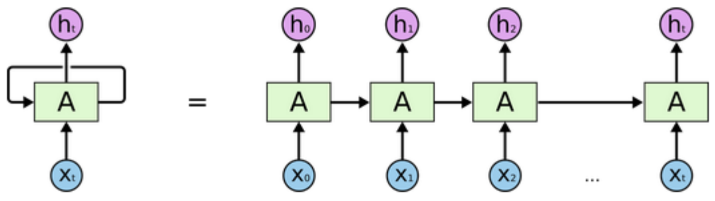
Word Embedding is a technique for learning dense representation of words in a low dimensional vector space. Each word can be seen as a point in this space, represented by a fixed length vector. Semantic relations between words are captured by this technique.
Reference;
https://indico.io/blog/sequence-modeling-neuralnets-part1

Good job in initializing the embedding weight matrix with values between -1 and 1. Weight values should be small Check out this lesson video to gain the intuition behind weight initializations.
https://www.youtube.com/watch?v=JIQl0jMpdsI

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)

http://colah.github.io/posts/2015-08-Understanding-LSTMs/
In reality the LSTM network is implemented as given below at the left side of the diagram. Right-side of the diagram is just for the people to understand all the math and computational theory (it's like we are "imagining" how the network will when it is unrolled).



An unrolled recurrent neural network.

In reality (implementation) Recurrent Neural Networks have loops (`tf.While` is used in the source code) as shown at the left-side of the above diagram allowing information (sequential input) to persist dynamically (`tf.nn.dynamic_rnn` can process unfixed sequence of inputs) .

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)

Logits are created from the output layer. Logits means linear combination of weights multiplied by the units (from previous layer) and then bias being added. This is the term used for the output of neural net's output layer before adding any non-linear activation function.

`logits = tf.contrib.layers.fully_connected(output, vocab_size, activation_fn=None)`

These logits are used by the non-linear activation function like softmax or sigmoid activation function which will generate the predicted values as below taken from Build the Graph section.

```
# Probabilities for generating words
probs = tf.nn.softmax(logits, name='probs')
```

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]

By running `get_batches([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20], 3, 2)` I can get the expected output

```
array([[[[ 1,  2],
         [ 7,  8],
         [13, 14]],

        [[ 2,  3],
         [ 8,  9],
         [14, 15]]],


       [[[ 3,  4],
         [ 9, 10],
         [15, 16]],

        [[ 4,  5],
         [10, 11],
         [16, 17]]],


       [[[ 5,  6],
         [11, 12],
         [17, 18]],

        [[ 6,  7],
         [12, 13],
         [18,  1]]]])
```

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.

Few number of LSTMCell layers like one or two layer(s) is perfect for this project. Large number of layers would have increased the training loss convergence time.

Batch size is also appropriate. If you have too large batch size then that means there will be fewer weight updates in each epoch. Therefore you have to increase the number of epochs so that the model converges.
If you use higher batch size and do not increase the number of epochs, accuracy level might seem lower and that is because the model has not converged (model still has the capacity to learn more).

The project gets a loss less than 1.0

Your project meets the loss requirements!

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

You have retrieved all the tensors with respective tensor names over here.

The `pick_word` function predicts the next word correctly.

You should use `np.random.choice()` here https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.choice.html

Probabilities are already calculated by the network i.e. which word is next likely to occur.
To make this more concrete;

```
vocab_list = ["hello", "how", "is"]
probabilities = [0.3,0.5,0.2]
np.random.choice(vocab_list, p=probabilities)
```

Now you can see in the above that the word 'how' is likely to appear by 0.5 probability, 'hello' is likely to appear by 0.3 probability, 'is' is likely to appear by 0.2 probability.

You are encouraged to use slight randomness when choosing the next word. If you don't, the predictions can fall into a loop of the same words.

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

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