Meets Specifications

Great first submission! 

Regarding your comments:

*git\_init\_cell(batch\_size, rnn\_size)  
-not clear about how/when to use keep\_prob and the layers when initializing the the RNN cell.*

For this project it's better not to use dropout and therefore a keep probability, check out my comment below for more information.

*build\_nn(cell, rnn\_size, input\_data, vocab\_size)  
-not clear on how/when to use the truncated normal initializer:  
Logits = tf.contrib.layers.fully\_connected(Outputs, vocab\_size, weights\_initializer = tf.truncated\_normal\_initializer(stddev = 0.01), activation\_fn=None)*

*In general it's better to always initialize the weights to speed up the training.  
Why did a very slow learning rate of 0.001 work better?*

Choosing a too big learning rate can result in overshooting the minimum of the cost function, therefore it can be better to choose a low learning rate. The obvious disadvantage of a low learning rate is that it will take longer for your network to train.

**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.**

All tests 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.**

**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)**

You have implemented all the placeholders with the right shapes, well done!

**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!

We are handling the LTSM cells as a black box in our project, if you want to dive into the details I would recommend to read the following resource:  
<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

Note that it is not really necessary to have a dropout layer in here, as the goal of the project is to generate text and there is not performance metric so overfitting is not a real concern. Also when calculating the loss and generating the text you would have to set the keep probability to 1, as you want to use the full network there.

**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 putting the previously defined functions together here!

Note that you are using the rnn\_size parameter for the embedding dimension, this is actually a different hyperparameter, but it doesn't matter much for the results of the project.

**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]**

**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.**

Good choices for the hyperparameters:

1. The number of epochs is chosen such that the loss is low enough (<1.0) and has stabilized.
2. Batch size is large enough and learning rate is small enough to train quickly.
3. The sequence length (15) is reasonable for the length of the sentences we want to generate - in the start of the exercise you calculated the average line length to be 11.5.

**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**

Note that you are able to easily recover the tensors after saving and loading the model, because you previously defined their names.

**The pick\_word function predicts the next word correctly.**

Smart to add randomness here to make the network generate a new script for every call 

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

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