TV Script Generation

In this project, you'll generate your own Simpsons (https://en.wikipedia.org/wiki/The_Simpsons) TV scripts using RNNs. You'll be using part of the Simpsons dataset (https://www.kaggle.com/wcukierski/the-simpsons-by-the-data) of scripts from 27 seasons. The Neural Network you'll build will generate a new TV script for a scene at Moe's Tavern (https://simpsonswiki.com/wiki/Moe's_Tavern).

Get the Data

The data is already provided for you. You'll be using a subset of the original dataset. It consists of only the scenes in Moe's Tavern. This doesn't include other versions of the tavern, like "Moe's Cavern", "Flaming Moe's", "Uncle Moe's Family Feed-Bag", etc..

In [1]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
import helper

data_dir = './data/simpsons/moes_tavern_lines.txt'
text = helper.load_data(data_dir)
# Ignore notice, since we don't use it for analysing the data
text = text[81:]
```

Explore the Data

Play around with `view_sentence_range` to view different parts of the data.

In [2]:
```
view_sentence_range = (0, 10)

"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
import numpy as np

print('Dataset Stats')
print('Roughly the number of unique words: {}'\
      .format(len({word: None for word in text.split()})))
scenes = text.split('\n\n')
print('Number of scenes: {}'.format(len(scenes)))
sentence_count_scene = [scene.count('\n') for scene in scenes]
print('Average number of sentences in each scene: {}'\
      .format(np.average(sentence_count_scene)))

sentences = [sentence for scene in scenes \
             for sentence in scene.split('\n')]
```

```
print('Number of lines: {}'.format(len(sentences)))
word_count_sentence = [len(sentence.split()) \
                       for sentence in sentences]
print('Average number of words in each line: {}'\
      .format(np.average(word_count_sentence)))

print()
print('The sentences {} to {}:'.format(*view_sentence_range))
print('\n'.join(text.split('\n')\
                [view_sentence_range[0]:view_sentence_range[1]]))
```

```
Dataset Stats
Roughly the number of unique words: 11492
Number of scenes: 262
Average number of sentences in each scene: 15.248091603053435
Number of lines: 4257
Average number of words in each line: 11.50434578341555

The sentences 0 to 10:
Moe_Szyslak: (INTO PHONE) Moe's Tavern. Where the elite meet to drin
k.
Bart_Simpson: Eh, yeah, hello, is Mike there? Last name, Rotch.
Moe_Szyslak: (INTO PHONE) Hold on, I'll check. (TO BARFLIES) Mike Ro
tch. Mike Rotch. Hey, has anybody seen Mike Rotch, lately?
Moe_Szyslak: (INTO PHONE) Listen you little puke. One of these days
I'm gonna catch you, and I'm gonna carve my name on your back with a
n ice pick.
Moe_Szyslak: What's the matter Homer? You're not your normal efferve
scent self.
Homer_Simpson: I got my problems, Moe. Give me another one.
Moe_Szyslak: Homer, hey, you should not drink to forget your problem
s.
Barney_Gumble: Yeah, you should only drink to enhance your social sk
ills.
```

Implement Preprocessing Functions

The first thing to do to any dataset is preprocessing. Implement the following preprocessing functions below:

- Lookup Table
- Tokenize Punctuation

Lookup Table

To create a word embedding, you first need to transform the words to ids. In this function, 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`

Return these dictionaries in the following tuple `(vocab_to_int, int_to_vocab)`

In [3]:
```
import numpy as np
import problem_unittests as tests
```

In [4]:
```
def create_lookup_tables(text):
    """
    Create lookup tables for vocabulary
    :param text: The text of tv scripts split into words
    :return: A tuple of dicts (vocab_to_int, int_to_vocab)
    """
    # TODO: Implement Function
    vocab_to_int = {word: value \
                    for value, word in enumerate(set(text),0)}
    # use set of text, duplicates can mess up training, i.e. leads to
    # need of reducing batch size to avoid memory issues
    int_to_vocab = {vocab_to_int[word]: word for word in vocab_to_int}
    return (vocab_to_int, int_to_vocab)

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_create_lookup_tables(create_lookup_tables)
```

```
Tests Passed
```

Tokenize Punctuation

We'll be splitting the script into a word array using spaces as delimiters. However, punctuations like periods and exclamation marks make it hard for the neural network to distinguish between the word "bye" and "bye!".

Implement the function `token_lookup` to return a dict that will be used to tokenize symbols like "!" into "||Exclamation_Mark||". Create a dictionary for the following symbols where the symbol is the key and value is the token:

- Period (.)
- Comma (,)
- Quotation Mark (")
- Semicolon (;)
- Exclamation mark (!)
- Question mark (?)
- Left Parentheses (( )
- Right Parentheses ( ))
- Dash (--)
- Return (\n)

This dictionary will be used to token the symbols and add the delimiter (space) around it. This separates the symbols as it's own word, making it easier for the neural network to predict on the next word. Make sure you don't use a token that could be confused as a word. Instead of using the token "dash", try using something like "||dash||".

In [5]:
```
def token_lookup():
    """
    Generate a dict to turn punctuation into a token.
    :return: Tokenize dictionary where
             the key is the punctuation
             and the value is the token
    """
    # TODO: Implement Function
    punctuation_dict =             \
    {'.' : '||Period||',           \
     ',' : '||Comma||',            \
     '"' : '||QuotationMark||',    \
     ';' : '||Semicolon||',        \
     '!' : '||ExclamationMark||',  \
     '?' : '||QuestionMark||',     \
     '(' : '||LeftParentheses||',  \
     ')' : '||RightParentheses||', \
     '--': '||Dash||',             \
     '\n': '||Return||'}

    return punctuation_dict

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_tokenize(token_lookup)
```

```
Tests Passed
```

Preprocess all the data and save it

Running the code cell below will preprocess all the data and save it to file.

In [6]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
# Preprocess Training, Validation, and Testing Data
helper.preprocess_and_save_data(data_dir, token_lookup, \
                                create_lookup_tables)
```

Check Point

This is your first checkpoint. If you ever decide to come back to this notebook or have to restart the notebook, you can start from here. The preprocessed data has been saved to disk.

In [7]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
import helper
import numpy as np
import problem_unittests as tests

int_text, vocab_to_int, int_to_vocab, token_dict = \
helper.load_preprocess()
```

Build the Neural Network

You'll build the components necessary to build a RNN by implementing the following functions below:

- get_inputs
- get_init_cell
- get_embed
- build_rnn
- build_nn
- get_batches

Check the Version of TensorFlow and Access to GPU

In [8]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
from distutils.version import LooseVersion
import warnings
import tensorflow as tf

# Check TensorFlow Version
assert LooseVersion(tf.__version__) >= LooseVersion('1.3'), \
'Please use TensorFlow version 1.3 or newer'
print('TensorFlow Version: {}'.format(tf.__version__))

# Check for a GPU
if not tf.test.gpu_device_name():
    warnings.warn('No GPU found. Please use a GPU to train \
    your neural network.')
else:
    print('Default GPU Device: {}'\
          .format(tf.test.gpu_device_name()))
```

```
TensorFlow Version: 1.4.1
Default GPU Device: /device:GPU:0
```

Input

Implement the `get_inputs()` function to create TF Placeholders for the Neural Network. It should create the following placeholders:

- Input text placeholder named "input" using the TF Placeholder (https://www.tensorflow.org/api_docs/python/tf/placeholder) `name` parameter.
- Targets placeholder
- Learning Rate placeholder

Return the placeholders in the following tuple `(Input, Targets, LearningRate)`

In [9]:
```
def get_inputs(): #use correct datatypes: learning_rate has to be floa
    """
    Create TF Placeholders for input, targets, and learning rate.
    :return: Tuple (input, targets, learning rate)
    """
    # TODO: Implement Function
    input_  = tf.placeholder(tf.int32, shape = (None, None), \
                             name='input')
    target_ = tf.placeholder(tf.int32, shape = (None, None), \
                             name='target')
    learning_rate = tf.placeholder(tf.float32)
    return input_, target_, learning_rate

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_get_inputs(get_inputs)
```

```
Tests Passed
```

Build RNN Cell and Initialize

Stack one or more `BasicLSTMCells` (https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/BasicLSTMCell) in a `MultiRNNCell` (https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/MultiRNNCell).

- The Rnn size should be set using `rnn_size`
- Initalize Cell State using the MultiRNNCell's `zero_state()` (https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/MultiRNNCell#zero_state) function
  - Apply the name "initial_state" to the initial state using `tf.identity()` (https://www.tensorflow.org/api_docs/python/tf/identity)

Return the cell and initial state in the following tuple `(Cell, InitialState)`

In [10]:
```
num_layers = 1
def get_init_cell(batch_size, rnn_size):
    """
    Create an RNN Cell and initialize it.
    :param batch_size: Size of batches
    :param rnn_size: Size of RNNs
    :return: Tuple (cell, initialize state)
    """
    # TODO: Implement Function
    cells = []
    for _ in range(num_layers):
        lstm_cell = tf.nn.rnn_cell.LSTMCell(rnn_size)
        #lstm_cell = tf.nn.rnn_cell.DropoutWrapper\
        #(cell = lstm_cell, output_keep_prob = 0.2)
        cells.append(lstm_cell)
    cell = tf.contrib.rnn.MultiRNNCell(cells)
    initial_state = tf.identity(cell.zero_state(batch_size, \
                                                tf.float32), \
                                name='initial_state')

    return (cell, initial_state)

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_get_init_cell(get_init_cell)
```

```
Tests Passed
```

Word Embedding

Apply embedding to `input_data` using TensorFlow. Return the embedded sequence.

In [11]:
```
def get_embed(input_data, vocab_size, embed_dim):
    """
    Create embedding for <input_data>.
    :param input_data: TF placeholder for text input.
    :param vocab_size: Number of words in vocabulary.
    :param embed_dim: Number of embedding dimensions
    :return: Embedded input.
    """
    # TODO: Implement Function
    embedding_matrix = tf.Variable\
    (tf.random_uniform(shape = (vocab_size, embed_dim), \
                       minval = -1, maxval = 1, dtype = tf.float32))
    embed  = tf.nn.embedding_lookup(embedding_matrix, input_data)
    return embed

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_get_embed(get_embed)
```

```
Tests Passed
```

Build RNN

You created a RNN Cell in the `get_init_cell()` function. Time to use the cell to create a RNN.

- Build the RNN using the `tf.nn.dynamic_rnn()` (https://www.tensorflow.org/api_docs/python/tf/nn/dynamic_rnn)
  - Apply the name "final_state" to the final state using `tf.identity()` (https://www.tensorflow.org/api_docs/python/tf/identity)

Return the outputs and final_state state in the following tuple `(Outputs, FinalState)`

In [12]:
```
def build_rnn(cell, inputs):
    """
    Create a RNN using a RNN Cell
    :param cell: RNN Cell
    :param inputs: Input text data
    :return: Tuple (Outputs, Final State)
    """
    # TODO: Implement Function
    outputs, final_state = tf.nn.dynamic_rnn(cell, inputs, \
                                             dtype=tf.float32)
    final_state          = tf.identity(final_state, \
                                       name = "final_state")
    return (outputs, final_state)

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_build_rnn(build_rnn)
```

```
Tests Passed
```

Build the Neural Network

Apply the functions you implemented above to:

- Apply embedding to `input_data` using your `get_embed(input_data, vocab_size, embed_dim)` function.
- Build RNN using `cell` and your `build_rnn(cell, inputs)` 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)

In [13]:
```
def build_nn(cell, rnn_size, input_data, vocab_size, embed_dim):
    """
    Build part of the neural network
    :param cell        : RNN cell
    :param rnn_size    : Size of rnns
    :param input_data  : Input data
    :param vocab_size  : Vocabulary size
    :param embed_dim   : Number of embedding dimensions
    :return            : Tuple (Logits, FinalState)
    """
    # TODO: Implement Function
    embed                 = get_embed(input_data, vocab_size, \
                                      embed_dim)
    outputs, final_state = build_rnn(cell, embed)
    num_outputs          = vocab_size
    logits               = tf.contrib.layers.fully_connected\
    (outputs, num_outputs, activation_fn = None)
    return (logits, final_state)

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_build_nn(build_nn)
```

```
Tests Passed
```

Batches

Implement `get_batches` to create batches of input and targets using `int_text`. The batches should be a Numpy array with the shape `(number of batches, 2, batch size, sequence length)`. Each batch contains two elements:

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

If you can't fill the last batch with enough data, drop the last batch.

For example, `get_batches([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20], 3, 2)` would return a Numpy array of the following:

```
[
  # First Batch
  [
    # Batch of Input
    [[ 1  2], [ 7  8], [13 14]]
    # Batch of targets
    [[ 2  3], [ 8  9], [14 15]]
  ]

  # Second Batch
  [
    # Batch of Input
    [[ 3  4], [ 9 10], [15 16]]
    # Batch of targets
    [[ 4  5], [10 11], [16 17]]
  ]

  # Third Batch
  [
    # Batch of Input
    [[ 5  6], [11 12], [17 18]]
    # Batch of targets
    [[ 6  7], [12 13], [18  1]]
  ]
]
```

Notice that the last target value in the last batch is the first input value of the first batch. In this case, `1`. This is a common technique used when creating sequence batches, although it is rather unintuitive.

In [14]:
```
def get_batches(int_text, batch_size, seq_length):
    """
    Return batches of input and target
    :param int_text  : Text with the words replaced by their ids
    :param batch_size: The size of batch
    :param seq_length: The length of sequence
    :return          : Batches as a Numpy array
    """
    batchrows      = 2 #1 row for input, 1 row for target
    batch_entries  = batch_size * seq_length

    batch_rows     = len(int_text) // batch_entries
    batch_cols     = batch_size * seq_length
    batch_elems    = batch_rows * batch_cols

    int_text       = int_text[:batch_elems]

    vertical_tiles = batch_rows
    A              = np.reshape(np.array(int_text),\
                                (batch_size, \
                                 seq_length * vertical_tiles))
    batch_arr      = np.zeros((batchrows*vertical_tiles, \
                               batch_entries))
    j_range        = A.shape[0]//(batch_size)

    for i in range(vertical_tiles):
        for j in range(j_range):
            row_idx  = slice(j*batch_size,j*batch_size+batch_size)
            col_idx  = slice(i*seq_length,(i+1)*seq_length)
            inp_ij   = A[row_idx, col_idx]
            tar_ij   = (inp_ij+1) % (np.max(A)+1)

            inp_idx  = batchrows*(j_range*i+j)
            tar_idx  = inp_idx + 1
            batch_arr[inp_idx,:] = np.reshape(inp_ij, \
                                              (1, batch_entries))
            batch_arr[tar_idx,:] = np.reshape(tar_ij, \
                                              (1, batch_entries))

    batches        = np.reshape(batch_arr, \
                                (-1, batchrows, \
                                 batch_size, seq_length))
    return batches

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_get_batches(get_batches)
```

```
Tests Passed
```

Neural Network Training

Hyperparameters

Tune the following parameters:

- Set `num_epochs` to the number of epochs.
- Set `batch_size` to the batch size.
- Set `rnn_size` to the size of the RNNs.
- Set `embed_dim` to the size of the embedding.
- Set `seq_length` to the length of sequence.
- Set `learning_rate` to the learning rate.
- Set `show_every_n_batches` to the number of batches the neural network should print progress.

In [15]:
```
# Number of Epochs
num_epochs = 16
# Batch Size
batch_size = 64
# RNN Size
rnn_size = 1024
# Embedding Dimension Size
embed_dim = 300
# Sequence Length
seq_length = 30
# Learning Rate
learning_rate = 0.02
# Show stats for every n number of batches
show_every_n_batches = 30

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
save_dir = './save'
```

Build the Graph

Build the graph using the neural network you implemented.

In [16]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
from tensorflow.contrib import seq2seq

train_graph = tf.Graph()
with train_graph.as_default():
    vocab_size              = len(int_to_vocab)
    input_text, targets, lr = get_inputs()
    input_data_shape        = tf.shape(input_text)
    cell, initial_state     = get_init_cell(input_data_shape[0], \
                                            rnn_size)
    logits, final_state     = build_nn(cell, rnn_size, input_text, \
                                       vocab_size, embed_dim)

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

    # Loss function
    cost = seq2seq.sequence_loss(
        logits,
        targets,
        tf.ones([input_data_shape[0], input_data_shape[1]]))

    # Optimizer
    optimizer        = tf.train.AdamOptimizer(lr)

    # Gradient Clipping
    gradients        = optimizer.compute_gradients(cost)
    capped_gradients =[(tf.clip_by_value(grad, -1., 1.), var) \
                       for grad, var in gradients if grad is not None]
    train_op         = optimizer.apply_gradients(capped_gradients)
```

Train

Train the neural network on the preprocessed data. If you have a hard time getting a good loss, check the forums (https://discussions.udacity.com/) to see if anyone is having the same problem.

In [17]:
```
%%time
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
batches = get_batches(int_text, batch_size, seq_length)

with tf.Session(graph=train_graph) as sess:
    sess.run(tf.global_variables_initializer())

    for epoch_i in range(num_epochs):
        state = sess.run(initial_state, {input_text: batches[0][0]})
```

```
        for batch_i, (x, y) in enumerate(batches):
            feed = {
                input_text: x,
                targets: y,
                initial_state: state,
                lr: learning_rate}
            train_loss, state, _ = sess.run([cost, final_state, \
                                             train_op], feed)

            # Show every <show_every_n_batches> batches
            if (epoch_i * len(batches) \
                + batch_i) % show_every_n_batches == 0:
                print('Epoch {:>3} Batch {:>4}/{} \
                train_loss = {:.3f}'.format(
                    epoch_i, batch_i, len(batches), train_loss))

    # Save Model
    saver = tf.train.Saver()
    saver.save(sess, save_dir)
    print('Model Trained and Saved')
```

```
Epoch   0 Batch    0/35                 train_loss = 8.827
Epoch   0 Batch   30/35                 train_loss = 1.600
Epoch   1 Batch   25/35                 train_loss = 0.456
Epoch   2 Batch   20/35                 train_loss = 0.077
Epoch   3 Batch   15/35                 train_loss = 0.021
Epoch   4 Batch   10/35                 train_loss = 0.001
Epoch   5 Batch    5/35                 train_loss = 0.000
Epoch   6 Batch    0/35                 train_loss = 0.000
Epoch   6 Batch   30/35                 train_loss = 0.000
Epoch   7 Batch   25/35                 train_loss = 0.000
Epoch   8 Batch   20/35                 train_loss = 0.000
Epoch   9 Batch   15/35                 train_loss = 0.000
Epoch  10 Batch   10/35                 train_loss = 0.000
Epoch  11 Batch    5/35                 train_loss = 0.000
Epoch  12 Batch    0/35                 train_loss = 0.000
Epoch  12 Batch   30/35                 train_loss = 0.000
Epoch  13 Batch   25/35                 train_loss = 0.000
Epoch  14 Batch   20/35                 train_loss = 0.000
Epoch  15 Batch   15/35                 train_loss = 0.000
Model Trained and Saved
CPU times: user 58.1 s, sys: 19 s, total: 1min 17s
Wall time: 1min 11s
```

Save Parameters

Save `seq_length` and `save_dir` for generating a new TV script.

In [18]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
# Save parameters for checkpoint
helper.save_params((seq_length, save_dir))
```

Checkpoint

In [19]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
import tensorflow as tf
import numpy as np
import helper
import problem_unittests as tests

_, vocab_to_int, int_to_vocab, token_dict = helper.load_preprocess()
seq_length, load_dir = helper.load_params()
```

Implement Generate Functions

Get Tensors

Get tensors from `loaded_graph` using the function `get_tensor_by_name()` (https://www.tensorflow.org/api_docs/python/tf/Graph#get_tensor_by_name). Get the tensors using the following names:

- "input:0"
- "initial_state:0"
- "final_state:0"
- "probs:0"

Return the tensors in the following tuple `(InputTensor, InitialStateTensor, FinalStateTensor, ProbsTensor)`

In [20]:
```
def get_tensors(loaded_graph):
    """
    Get input, initial state, final state,
    and probabilities tensor from <loaded_graph>
    :param loaded_graph: TensorFlow graph loaded from file
    :return: Tuple (InputTensor, InitialStateTensor, \
                    FinalStateTensor, ProbsTensor)
    """
    # TODO: Implement Function
    return loaded_graph.get_tensor_by_name("input:0"), \
           loaded_graph.get_tensor_by_name("initial_state:0"), \
           loaded_graph.get_tensor_by_name("final_state:0"), \
           loaded_graph.get_tensor_by_name("probs:0")

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_get_tensors(get_tensors)
```

```
Tests Passed
```

Choose Word

Implement the `pick_word()` function to select the next word using `probabilities`.

In [21]:
```
def pick_word(probabilities, int_to_vocab):
    """
    Pick the next word in the generated text
    :param probabilities: Probabilities of the next word
    :param int_to_vocab:  Dictionary of
                          word ids as keys and
                          words as values
    :return: String of the predicted word
    """
    # TODO: Implement Function
    return int_to_vocab[np.argmax(probabilities)]


"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_pick_word(pick_word)
```

```
Tests Passed
```

Generate TV Script

This will generate the TV script for you. Set `gen_length` to the length of TV script you want to generate.

In [22]:
```
%%time
gen_length = 200
# homer_simpson, moe_szyslak, or Barney_Gumble
prime_word = 'moe_szyslak'

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
loaded_graph = tf.Graph()
with tf.Session(graph=loaded_graph) as sess:
    # Load saved model
    loader = tf.train.import_meta_graph(load_dir + '.meta')
    loader.restore(sess, load_dir)

    # Get Tensors from loaded model
    input_text, initial_state, final_state, probs = \
    get_tensors(loaded_graph)

    # Sentences generation setup
    gen_sentences = [prime_word + ':']
    prev_state = sess.run(initial_state, \
                          {input_text: np.array([[1]])})

    # Generate sentences
    for n in range(gen_length):
        # Dynamic Input
        dyn_input = [[vocab_to_int[word] \
                      for word in gen_sentences[-seq_length:]]]
        dyn_seq_length = len(dyn_input[0])

        # Get Prediction
        probabilities, prev_state = sess.run(
            [probs, final_state],
            {input_text: dyn_input, initial_state: prev_state})

        pred_word = pick_word(probabilities[0][dyn_seq_length-1], \
                              int_to_vocab)

        gen_sentences.append(pred_word)

    # Remove tokens
    tv_script = ' '.join(gen_sentences)
    for key, token in token_dict.items():
        ending = ' ' if key in ['\n', '(', '"'] else ''
        tv_script = tv_script.replace(' ' + token.lower(), key)
    tv_script = tv_script.replace('\n ', '\n')
    tv_script = tv_script.replace('( ', '(')

    print(tv_script)
```

```
INFO:tensorflow:Restoring parameters from ./save
moe_szyslak: reconsidering closing shoot lotsa miles haircuts slit n
udge busiest trucks sagely vincent heard presents africa tonight's t
```

```
renchant confession indignant life's bleak countryman calling sing-s
ong eggshell parenting cyrano blow elocution renovations selection s
hreda americans! nudge busiest trucks sagely vincent heard presents
africa tonight's trenchant confession indignant life's bleak country
man calling sing-song eggshell parenting cyrano blow elocution renov
ations selection shreda americans! nudge busiest trucks sagely vince
nt heard presents africa tonight's trenchant confession indignant li
fe's bleak countryman calling sing-song eggshell parenting cyrano bl
ow elocution renovations selection shreda americans! nudge busiest t
rucks sagely vincent heard presents africa tonight's trenchant confe
ssion indignant life's bleak countryman calling sing-song eggshell p
arenting cyrano blow elocution renovations selection shreda american
s! nudge busiest trucks sagely vincent heard presents africa tonight
's trenchant confession indignant life's bleak countryman calling si
ng-song eggshell parenting cyrano blow elocution renovations selecti
on shreda americans! nudge busiest trucks sagely vincent heard prese
nts africa tonight's trenchant confession indignant life's bleak cou
ntryman calling sing-song eggshell parenting cyrano blow elocution r
enovations selection shreda americans! nudge busiest trucks sagely v
incent heard presents africa tonight's trenchant confession indignan
t life's bleak countryman calling sing-song eggshell parenting cyran
o blow elocution renovations selection shreda americans! nudge busie
st trucks sagely
CPU times: user 2.04 s, sys: 236 ms, total: 2.28 s
Wall time: 2.33 s
```

The TV Script is Nonsensical

It's ok if the TV script doesn't make any sense. We trained on less than a megabyte of text. In order to get good results, you'll have to use a smaller vocabulary or get more data. Luckily there's more data! As we mentioned in the beggining of this project, this is a subset of another dataset (https://www.kaggle.com/wcukierski/the-simpsons-by-the-data). We didn't have you train on all the data, because that would take too long. However, you are free to train your neural network on all the data. After you complete the project, of course.

Submitting This Project

When submitting this project, make sure to run all the cells before saving the notebook. Save the notebook file as "dlnd_tv_script_generation.ipynb" and save it as a HTML file under "File" -> "Download as". Include the "helper.py" and "problem_unittests.py" files in your submission.