TV Script Generation

In this project, you'll generate your own Seinfeld (https://en.wikipedia.org/wiki/Seinfeld) TV scripts using RNNs. You'll be using part of the Seinfeld dataset (https://www.kaggle.com/thec03u5/seinfeld-chronicles#scripts.csv) of scripts from 9 seasons. The Neural Network you'll build will generate a new ,"fake" TV script, based on patterns it recognizes in this training data.

Get the Data

The data is already provided for you in `./data/Seinfeld_Scripts.txt` and you're encouraged to open that file and look at the text.

> - As a first step, we'll load in this data and look at some samples.
> - Then, you'll be tasked with defining and training an RNN to generate a new script!

In [1]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
# load in data
import helper
data_dir = './data/Seinfeld_Scripts.txt'
text = helper.load_data(data_dir)
```

Explore the Data

Play around with `view_line_range` to view different parts of the data. This will give you a sense of the data you'll be working with. You can see, for example, that it is all lowercase text, and each new line of dialogue is separated by a newline character `\n`.

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

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
import numpy as np

print('Dataset Stats')
print('Roughly the number of unique words: {}'.format(len({word: None for word in
lines = text.split('\n')
print('Number of lines: {}'.format(len(lines)))
word_count_line = [len(line.split()) for line in lines]
print('Average number of words in each line: {}'.format(np.average(word_count_lin

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

```
Dataset Stats
Roughly the number of unique words: 46367
Number of lines: 109233
Average number of words in each line: 5.544240293684143

The lines 0 to 10:
jerry: do you know what this is all about? do you know, why were here? to be
out, this is out...and out is one of the single most enjoyable experiences of
life. people...did you ever hear people talking about we should go out? this
is what theyre talking about...this whole thing, were all out now, no one is
home. not one person here is home, were all out! there are people trying to f
ind us, they dont know where we are. (on an imaginary phone) did you ring?, i
cant find him. where did he go? he didnt tell me where he was going. he must
have gone out. you wanna go out you get ready, you pick out the clothes, righ
t? you take the shower, you get all ready, get the cash, get your friends, th
e car, the spot, the reservation...then youre standing around, what do you d
o? you go we gotta be getting back. once youre out, you wanna get back! you w
anna go to sleep, you wanna get up, you wanna go out again tomorrow, right? w
here ever you are in life, its my feeling, youve gotta go.
```

In [3]:
```
vocab = []
for line in lines:
    for word in line.split():
        vocab.append(word)
vocab = set(vocab)
vocab_len = len(vocab)
```

Implement Pre-processing Functions

The first thing to do to any dataset is pre-processing. Implement the following pre-processing 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 [4]:
```
import problem_unittests as tests
from collections import Counter

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

    word_list = list(set(text))
    counts = Counter(word_list)
    vocab = sorted(counts, key=counts.get, reverse=True)
    vocab_to_int = {word:ii for ii, word in enumerate(vocab)}
    int_to_vocab = {ii:word for ii, word in enumerate(vocab)}

    # return tuple
    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
```

In [5]:
```
len(list(set(text)))
```

Out[5]:
```
75
```

Tokenize Punctuation

We'll be splitting the script into a word array using spaces as delimiters. However, punctuations like periods and exclamation marks can create multiple ids for the same word. For example, "bye" and "bye!" would generate two different word ids.

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 tokenize the symbols and add the delimiter (space) around it. This separates each symbols as its own word, making it easier for the neural network to predict the next word. Make sure you don't use a value that could be confused as a word; for example, instead of using the value "dash", try using something like "||dash||".

In [6]:
```
def token_lookup():
    """
    Generate a dict to turn punctuation into a token.
    :return: Tokenized dictionary where the key is the punctuation and the value
    """
    # TODO: Implement Function

    punc_dic = {'.':'||Period||',
                ',':'||Comma||',
                '"':'||Quotation_Mark||',
                ';':'||Semicolon||',
                '!':'||Exclamation_Mark||',
                '?':'||Question_Mark||',
                '(':'||Left_Parentheses||',
                ')':'||Right_Parentheses||',
                '-':'||Dash||',
                '\n':'||Return||'}

    return punc_dic

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

```
Tests Passed
```

Pre-process all the data and save it

Running the code cell below will pre-process all the data and save it to file. You're encouraged to look at the code for `preprocess_and_save_data` in the `helpers.py` file to see what it's doing in detail, but you do not need to change this code.

In [7]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
# pre-process training 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 [8]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
import helper
import problem_unittests as tests

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

Build the Neural Network

In this section, you'll build the components necessary to build an RNN by implementing the RNN Module and forward and backpropagation functions.

Check Access to GPU

In [9]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
import torch

# Check for a GPU
train_on_gpu = torch.cuda.is_available()
if not train_on_gpu:
    print('No GPU found. Please use a GPU to train your neural network.')
```

Input

Let's start with the preprocessed input data. We'll use TensorDataset (http://pytorch.org/docs/master/data.html#torch.utils.data.TensorDataset) to provide a known format to our dataset; in combination with DataLoader (http://pytorch.org/docs/master/data.html#torch.utils.data.DataLoader), it will handle batching, shuffling, and other dataset iteration functions.

You can create data with TensorDataset by passing in feature and target tensors. Then create a DataLoader as usual.

```
data = TensorDataset(feature_tensors, target_tensors)
data_loader = torch.utils.data.DataLoader(data,
                                          batch_size=batch_size)
```

Batching

Implement the `batch_data` function to batch `words` data into chunks of size `batch_size` using the `TensorDataset` and `DataLoader` classes.

> You can batch words using the DataLoader, but it will be up to you to create `feature_tensors` and `target_tensors` of the correct size and content for a given `sequence_length`.

For example, say we have these as input:

```
words = [1, 2, 3, 4, 5, 6, 7]
sequence_length = 4
```

Your first `feature_tensor` should contain the values:

```
[1, 2, 3, 4]
```

And the corresponding `target_tensor` should just be the next "word"/tokenized word value:

```
5
```

This should continue with the second `feature_tensor`, `target_tensor` being:

```
[2, 3, 4, 5]  # features
6             # target
```

In [10]:
```
from torch.utils.data import TensorDataset, DataLoader


def batch_data(words, sequence_length, batch_size):
    """
    Batch the neural network data using DataLoader
    :param words: The word ids of the TV scripts
    :param sequence_length: The sequence length of each batch
    :param batch_size: The size of each batch; the number of sequences in a batch
    :return: DataLoader with batched data
    """
    # TODO: Implement function

    feature_tensors, target_tensors = [], []
    data_count = len(words)
    for ii in range(data_count):
        last = ii + sequence_length
        if last < data_count:
            feature_tensors.append(words[ii:last])
            target_tensors.append(words[last])
        else:
            break

    feature_tensors = torch.tensor(feature_tensors)
    target_tensors = torch.tensor(target_tensors)

    if train_on_gpu:
        feature_tensors.cuda()
        target_tensors.cuda()

    data = TensorDataset(feature_tensors, target_tensors)
    data_loader = torch.utils.data.DataLoader(data,
                                              batch_size=batch_size,
                                              shuffle=False)


    # return a dataloader
    return data_loader

# there is no test for this function, but you are encouraged to create
# print statements and tests of your own0
```

Test your dataloader

You'll have to modify this code to test a batching function, but it should look fairly similar.

Below, we're generating some test text data and defining a dataloader using the function you defined, above. Then, we are getting some sample batch of inputs `sample_x` and targets `sample_y` from our dataloader.

Your code should return something like the following (likely in a different order, if you shuffled your data):

```
torch.Size([10, 5])
tensor([[ 28,  29,  30,  31,  32],
        [ 21,  22,  23,  24,  25],
        [ 17,  18,  19,  20,  21],
        [ 34,  35,  36,  37,  38],
        [ 11,  12,  13,  14,  15],
        [ 23,  24,  25,  26,  27],
        [  6,   7,   8,   9,  10],
        [ 38,  39,  40,  41,  42],
        [ 25,  26,  27,  28,  29],
        [  7,   8,   9,  10,  11]])

torch.Size([10])
tensor([ 33,  26,  22,  39,  16,  28,  11,  43,  30,  12])
```

Sizes

Your sample_x should be of size `(batch_size, sequence_length)` or (10, 5) in this case and sample_y should just have one dimension: batch_size (10).

Values

You should also notice that the targets, sample_y, are the *next* value in the ordered test_text data. So, for an input sequence `[ 28,  29,  30,  31,  32]` that ends with the value `32`, the corresponding output should be `33`.

In [11]:
```
# test dataloader

test_text = range(50)
t_loader = batch_data(test_text, sequence_length=5, batch_size=10)

data_iter = iter(t_loader)
sample_x, sample_y = data_iter.next()

print(sample_x.shape)
print(sample_x)
print()
print(sample_y.shape)
print(sample_y)
```

```
torch.Size([10, 5])
tensor([[ 0,  1,  2,  3,  4],
        [ 1,  2,  3,  4,  5],
        [ 2,  3,  4,  5,  6],
        [ 3,  4,  5,  6,  7],
        [ 4,  5,  6,  7,  8],
        [ 5,  6,  7,  8,  9],
        [ 6,  7,  8,  9, 10],
        [ 7,  8,  9, 10, 11],
        [ 8,  9, 10, 11, 12],
        [ 9, 10, 11, 12, 13]])

torch.Size([10])
tensor([ 5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Build the Neural Network

Implement an RNN using PyTorch's Module class (http://pytorch.org/docs/master/nn.html#torch.nn.Module). You may choose to use a GRU or an LSTM. To complete the RNN, you'll have to implement the following functions for the class:

- `__init__` - The initialize function.
- `init_hidden` - The initialization function for an LSTM/GRU hidden state
- `forward` - Forward propagation function.

The initialize function should create the layers of the neural network and save them to the class. The forward propagation function will use these layers to run forward propagation and generate an output and a hidden state.

The output of this model should be the *last* batch of word scores after a complete sequence has been processed. That is, for each input sequence of words, we only want to output the word scores for a single, most likely, next word.

Hints

1. Make sure to stack the outputs of the lstm to pass to your fully-connected layer, you can do this with `lstm_output = lstm_output.contiguous().view(-1, self.hidden_dim)`

2. You can get the last batch of word scores by shaping the output of the final, fully-connected layer like so:

```
# reshape into (batch_size, seq_length, output_size)
output = output.view(batch_size, -1, self.output_size)
# get last batch
out = output[:, -1]
```

In [50]:
```
import torch.nn as nn

class RNN(nn.Module):

    def __init__(self, vocab_size, output_size, embedding_dim, hidden_dim, n_laye
        """
        Initialize the PyTorch RNN Module
        :param vocab_size: The number of input dimensions of the neural network (
        :param output_size: The number of output dimensions of the neural network
        :param embedding_dim: The size of embeddings, should you choose to use th
        :param hidden_dim: The size of the hidden layer outputs
        :param dropout: dropout to add in between LSTM/GRU layers
        """
        super(RNN, self).__init__()
        # TODO: Implement function

        # set class variables

        self.embed_dim = embedding_dim
        self.n_hidden = hidden_dim
        self.drop_prob = dropout
        self.n_layers = n_layers
        self.output_size = output_size

        # define model layers

        self.embed = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers, batch_first=True
        self.drop = nn.Dropout(dropout)
        self.fc = nn.Linear(hidden_dim, output_size)

    def forward(self, nn_input, hidden):
        """
        Forward propagation of the neural network
        :param nn_input: The input to the neural network
        :param hidden: The hidden state
        :return: Two Tensors, the output of the neural network and the latest hid
        """
        # TODO: Implement function
        nn_input = nn_input.type(torch.cuda.LongTensor)
        x = self.embed(nn_input)
        lstm_out, new_hidden = self.lstm(x, hidden)
        lstm_out = self.drop(lstm_out)
        lstm_output = lstm_out.contiguous().view(-1, self.n_hidden)
        output = self.fc(lstm_output)
        output = output.view(nn_input.shape[0], -1, self.output_size)
        out = output[:, -1]

        # return one batch of output word scores and the hidden state

        return out, new_hidden


    def init_hidden(self, batch_size):
        '''
        Initialize the hidden state of an LSTM/GRU
```

```
        :param batch_size: The batch_size of the hidden state
        :return: hidden state of dims (n_layers, batch_size, hidden_dim)
        '''
        # Implement function

        # initialize hidden state with zero weights, and move to GPU if available

        weight = next(self.parameters()).data

        if (train_on_gpu):
            hidden = (weight.new(self.n_layers, batch_size, self.n_hidden).zero_(
                      weight.new(self.n_layers, batch_size, self.n_hidden).zero_(
        else:
            hidden = (weight.new(self.n_layers, batch_size, self.n_hidden).zero_(
                      weight.new(self.n_layers, batch_size, self.n_hidden).zero_(

        return hidden

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_rnn(RNN, train_on_gpu)
```

```
Tests Passed
```

Define forward and backpropagation

Use the RNN class you implemented to apply forward and back propagation. This function will be called, iteratively, in the training loop as follows:

```
loss = forward_back_prop(decoder, decoder_optimizer, criterion, inp, tar
get)
```

And it should return the average loss over a batch and the hidden state returned by a call to `RNN(inp, hidden)`. Recall that you can get this loss by computing it, as usual, and calling `loss.item()`.

If a GPU is available, you should move your data to that GPU device, here.

In [47]:
```
def forward_back_prop(rnn, optimizer, criterion, inp, target, hidden):
    """
    Forward and backward propagation on the neural network
    :param decoder: The PyTorch Module that holds the neural network
    :param decoder_optimizer: The PyTorch optimizer for the neural network
    :param criterion: The PyTorch loss function
    :param inp: A batch of input to the neural network
    :param target: The target output for the batch of input
    :return: The loss and the latest hidden state Tensor
    """

    # TODO: Implement Function

    # move data to GPU, if available

    if (train_on_gpu):
        inp, target = inp.cuda(), target.cuda()

    # perform backpropagation and optimization

    optimizer.zero_grad()
    hidden = tuple([h.data for h in hidden])
    output, new_hidden = rnn(inp, hidden)
    loss = criterion(output, target)
    loss.backward(retain_graph=True)
    nn.utils.clip_grad_norm_(rnn.parameters(), 5)
    optimizer.step()

    # return the loss over a batch and the hidden state produced by our model
    return loss.item(), new_hidden

# Note that these tests aren't completely extensive.
# they are here to act as general checks on the expected outputs of your functior
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
tests.test_forward_back_prop(RNN, forward_back_prop, train_on_gpu)
```

```
Tests Passed
```

Neural Network Training

With the structure of the network complete and data ready to be fed in the neural network, it's time to train it.

Train Loop

The training loop is implemented for you in the `train_decoder` function. This function will train the network over all the batches for the number of epochs given. The model progress will be shown every number of batches. This number is set with the `show_every_n_batches` parameter. You'll set this parameter along with other parameters in the next section.

In [14]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""

def train_rnn(rnn, batch_size, optimizer, criterion, n_epochs, show_every_n_batch
    batch_losses = []

    rnn.train()

    print("Training for %d epoch(s)..." % n_epochs)
    for epoch_i in range(1, n_epochs + 1):

        # initialize hidden state
        hidden = rnn.init_hidden(batch_size)

        for batch_i, (inputs, labels) in enumerate(train_loader, 1):

            # make sure you iterate over completely full batches, only
            n_batches = len(train_loader.dataset)//batch_size
            if(batch_i > n_batches):
                break

            # forward, back prop
            loss, hidden = forward_back_prop(rnn, optimizer, criterion, inputs, l
            # record loss
            batch_losses.append(loss)

            # printing loss stats
            if batch_i % show_every_n_batches == 0:
                print('Epoch: {:>4}/{:<4}  Loss: {}\n'.format(
                    epoch_i, n_epochs, np.average(batch_losses)))
                batch_losses = []

    # returns a trained rnn
    return rnn
```

Hyperparameters

Set and train the neural network with the following parameters:

- Set `sequence_length` to the length of a sequence.
- Set `batch_size` to the batch size.
- Set `num_epochs` to the number of epochs to train for.
- Set `learning_rate` to the learning rate for an Adam optimizer.
- Set `vocab_size` to the number of unique tokens in our vocabulary.
- Set `output_size` to the desired size of the output.
- Set `embedding_dim` to the embedding dimension; smaller than the vocab_size.
- Set `hidden_dim` to the hidden dimension of your RNN.
- Set `n_layers` to the number of layers/cells in your RNN.
- Set `show_every_n_batches` to the number of batches at which the neural network should print progress.

If the network isn't getting the desired results, tweak these parameters and/or the layers in the `RNN` class.

In [15]:
```
# Data params
# Sequence Length
sequence_length = int(np.average(word_count_line))  # of words in a sequence
# Batch Size
batch_size = 256

# data loader - do not change
train_loader = batch_data(int_text, sequence_length, batch_size)
```

In [16]:
```
# Training parameters
# Number of Epochs
num_epochs = 20
# Learning Rate
learning_rate = 0.0003

# Model parameters
# Vocab size
vocab_size = vocab_len
# Output size
output_size = vocab_size
# Embedding Dimension
embedding_dim = 128
# Hidden Dimension
hidden_dim = 512
# Number of RNN Layers
n_layers = 2

# Show stats for every n number of batches
show_every_n_batches = 300
```

Train

In the next cell, you'll train the neural network on the pre-processed data. If you have a hard time getting a good loss, you may consider changing your hyperparameters. In general, you may get better results with larger hidden and n_layer dimensions, but larger models take a longer time to train.

> **You should aim for a loss less than 3.5.**

You should also experiment with different sequence lengths, which determine the size of the long range dependencies that a model can learn.

In [17]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""

# create model and move to gpu if available
rnn = RNN(vocab_size, output_size, embedding_dim, hidden_dim, n_layers, dropout=0

#rnn = helper.load_model('./save/trained_rnn')

if train_on_gpu:
    rnn.cuda()

# defining loss and optimization functions for training
optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)
criterion = nn.CrossEntropyLoss()

# training the model
trained_rnn = train_rnn(rnn, batch_size, optimizer, criterion, num_epochs, show_e

# saving the trained model
helper.save_model('./save/trained_rnn', trained_rnn)
print('Model Trained and Saved')
```

```
Training for 20 epoch(s)...
Epoch:    1/20    Loss: 6.166906641324361

Epoch:    1/20    Loss: 5.284500611623129

Epoch:    1/20    Loss: 5.026172916094462

Epoch:    1/20    Loss: 5.055347129503886

Epoch:    1/20    Loss: 5.026087701320648

Epoch:    1/20    Loss: 4.92860019604365

Epoch:    1/20    Loss: 4.773310003280639

Epoch:    1/20    Loss: 4.72852267185847

Epoch:    1/20    Loss: 4.720375887552897

Epoch:    1/20    Loss: 4.79680813630422

Epoch:    1/20    Loss: 4.756834065119426

Epoch:    2/20    Loss: 4.5715331937655925

Epoch:    2/20    Loss: 4.461386924584707

Epoch:    2/20    Loss: 4.3143901475270585

Epoch:    2/20    Loss: 4.4360723662376405

Epoch:    2/20    Loss: 4.4417576702435815
```

```
Epoch:    2/20    Loss: 4.398240751425425
Epoch:    2/20    Loss: 4.2976508728663125
Epoch:    2/20    Loss: 4.300085591475169
Epoch:    2/20    Loss: 4.32673012971878
Epoch:    2/20    Loss: 4.431310733159383
Epoch:    2/20    Loss: 4.408769843578338
Epoch:    3/20    Loss: 4.291216683781837
Epoch:    3/20    Loss: 4.222835377057393
Epoch:    3/20    Loss: 4.078492635091146
Epoch:    3/20    Loss: 4.209536627928416
Epoch:    3/20    Loss: 4.227209824721019
Epoch:    3/20    Loss: 4.19306415160497
Epoch:    3/20    Loss: 4.103869194189707
Epoch:    3/20    Loss: 4.103632225195567
Epoch:    3/20    Loss: 4.138512329260508
Epoch:    3/20    Loss: 4.249287795225779
Epoch:    3/20    Loss: 4.2191723116238915
Epoch:    4/20    Loss: 4.12280001916176
Epoch:    4/20    Loss: 4.079385488828024
Epoch:    4/20    Loss: 3.937423752943675
Epoch:    4/20    Loss: 4.06597101688385
Epoch:    4/20    Loss: 4.081264910697937
Epoch:    4/20    Loss: 4.051883801619212
Epoch:    4/20    Loss: 3.9696614066759746
Epoch:    4/20    Loss: 3.965343670845032
Epoch:    4/20    Loss: 4.014019966920217
Epoch:    4/20    Loss: 4.117652072111766
Epoch:    4/20    Loss: 4.090604993502299
Epoch:    5/20    Loss: 4.005126928494981
```

```
Epoch:    5/20    Loss: 3.9660691523551943
Epoch:    5/20    Loss: 3.8293297123908996
Epoch:    5/20    Loss: 3.954466772079468
Epoch:    5/20    Loss: 3.9793971745173136
Epoch:    5/20    Loss: 3.940878473917643
Epoch:    5/20    Loss: 3.869077290693919
Epoch:    5/20    Loss: 3.8651479069391885
Epoch:    5/20    Loss: 3.9124817363421123
Epoch:    5/20    Loss: 4.0182735975583395
Epoch:    5/20    Loss: 3.988338828086853
Epoch:    6/20    Loss: 3.9044575666593127
Epoch:    6/20    Loss: 3.872883973916372
Epoch:    6/20    Loss: 3.7380058415730795
Epoch:    6/20    Loss: 3.857523137728373
Epoch:    6/20    Loss: 3.8927526330947875
Epoch:    6/20    Loss: 3.8566914860407513
Epoch:    6/20    Loss: 3.7862859789530434
Epoch:    6/20    Loss: 3.775526269276937
Epoch:    6/20    Loss: 3.827167099316915
Epoch:    6/20    Loss: 3.930541168053945
Epoch:    6/20    Loss: 3.9053866052627564
Epoch:    7/20    Loss: 3.8246246068930825
Epoch:    7/20    Loss: 3.7887618176142372
Epoch:    7/20    Loss: 3.6558095304171245
Epoch:    7/20    Loss: 3.7776917187372843
Epoch:    7/20    Loss: 3.8121523825327555
Epoch:    7/20    Loss: 3.781237179438273
Epoch:    7/20    Loss: 3.710743755499522
```

```
Epoch:    7/20    Loss: 3.7002864464124046

Epoch:    7/20    Loss: 3.7593068488438925

Epoch:    7/20    Loss: 3.8588015619913736

Epoch:    7/20    Loss: 3.828167091210683

Epoch:    8/20    Loss: 3.7484033358983755

Epoch:    8/20    Loss: 3.7187203629811605

Epoch:    8/20    Loss: 3.592126901944478

Epoch:    8/20    Loss: 3.7026097639401754

Epoch:    8/20    Loss: 3.7493184351921083

Epoch:    8/20    Loss: 3.713150468667348

Epoch:    8/20    Loss: 3.64392144203186

Epoch:    8/20    Loss: 3.634095646540324

Epoch:    8/20    Loss: 3.6924928538004558

Epoch:    8/20    Loss: 3.7849569741884865

Epoch:    8/20    Loss: 3.7608545740445454

Epoch:    9/20    Loss: 3.680623375679836

Epoch:    9/20    Loss: 3.6538587323824565

Epoch:    9/20    Loss: 3.5345018831888835

Epoch:    9/20    Loss: 3.635981786251068

Epoch:    9/20    Loss: 3.6877889053026833

Epoch:    9/20    Loss: 3.6507236615816754

Epoch:    9/20    Loss: 3.5814657767613727

Epoch:    9/20    Loss: 3.5735534771283466

Epoch:    9/20    Loss: 3.6358516176541644

Epoch:    9/20    Loss: 3.726883103052775

Epoch:    9/20    Loss: 3.7000667389233906

Epoch:   10/20    Loss: 3.6120176448309715

Epoch:   10/20    Loss: 3.5921640904744465

Epoch:   10/20    Loss: 3.476391252676646
```

```
Epoch:   10/20    Loss: 3.575380591551463
Epoch:   10/20    Loss: 3.6296756283442178
Epoch:   10/20    Loss: 3.587345542112986
Epoch:   10/20    Loss: 3.5261797229448955
Epoch:   10/20    Loss: 3.515717113018036
Epoch:   10/20    Loss: 3.5749727447827655
Epoch:   10/20    Loss: 3.6684603905677795
Epoch:   10/20    Loss: 3.6409364636739094
Epoch:   11/20    Loss: 3.554925933849713
Epoch:   11/20    Loss: 3.530753457546234
Epoch:   11/20    Loss: 3.4221147378285726
Epoch:   11/20    Loss: 3.5119271365801494
Epoch:   11/20    Loss: 3.5650330169995628
Epoch:   11/20    Loss: 3.526580033302307
Epoch:   11/20    Loss: 3.4643022831281027
Epoch:   11/20    Loss: 3.457857462565104
Epoch:   11/20    Loss: 3.5211846113204954
Epoch:   11/20    Loss: 3.6039963134129844
Epoch:   11/20    Loss: 3.5844985445340476
Epoch:   12/20    Loss: 3.4937391207237876
Epoch:   12/20    Loss: 3.477234500249227
Epoch:   12/20    Loss: 3.365831747055054
Epoch:   12/20    Loss: 3.4525098530451457
Epoch:   12/20    Loss: 3.5116126958529152
Epoch:   12/20    Loss: 3.468659019470215
Epoch:   12/20    Loss: 3.4117174776395163
Epoch:   12/20    Loss: 3.4036520075798036
Epoch:   12/20    Loss: 3.468314181168874
```

```
Epoch:   12/20    Loss: 3.549084057013194
Epoch:   12/20    Loss: 3.5261399523417154
Epoch:   13/20    Loss: 3.437016766918592
Epoch:   13/20    Loss: 3.4140725350379943
Epoch:   13/20    Loss: 3.316547814210256
Epoch:   13/20    Loss: 3.39327343861262
Epoch:   13/20    Loss: 3.4541206487019855
Epoch:   13/20    Loss: 3.4152610961596173
Epoch:   13/20    Loss: 3.361397310892741
Epoch:   13/20    Loss: 3.352918404738108
Epoch:   13/20    Loss: 3.4132275565465293
Epoch:   13/20    Loss: 3.486266792615255
Epoch:   13/20    Loss: 3.4737390462557474
Epoch:   14/20    Loss: 3.383209237382432
Epoch:   14/20    Loss: 3.367389775911967
Epoch:   14/20    Loss: 3.2671454254786174
Epoch:   14/20    Loss: 3.337113059361776
Epoch:   14/20    Loss: 3.397752025127411
Epoch:   14/20    Loss: 3.3556182193756103
Epoch:   14/20    Loss: 3.3065628218650818
Epoch:   14/20    Loss: 3.300511984825134
Epoch:   14/20    Loss: 3.3611881287892658
Epoch:   14/20    Loss: 3.4303612438837687
Epoch:   14/20    Loss: 3.422954454421997
Epoch:   15/20    Loss: 3.3262302826258763
Epoch:   15/20    Loss: 3.3021217211087546
Epoch:   15/20    Loss: 3.211452845732371
Epoch:   15/20    Loss: 3.2792199182510378
Epoch:   15/20    Loss: 3.3446946581204733
```

```
Epoch:   15/20    Loss: 3.3095811216036477

Epoch:   15/20    Loss: 3.2598287876447043

Epoch:   15/20    Loss: 3.2423482664426166

Epoch:   15/20    Loss: 3.3109946433703104

Epoch:   15/20    Loss: 3.3790525658925374

Epoch:   15/20    Loss: 3.36636088291804

Epoch:   16/20    Loss: 3.267802481808938

Epoch:   16/20    Loss: 3.2556560627619424

Epoch:   16/20    Loss: 3.16375337600708

Epoch:   16/20    Loss: 3.227025045553843

Epoch:   16/20    Loss: 3.2859163149197896

Epoch:   16/20    Loss: 3.2621730542182923

Epoch:   16/20    Loss: 3.207493882973989

Epoch:   16/20    Loss: 3.199490951697032

Epoch:   16/20    Loss: 3.2603433688481647

Epoch:   16/20    Loss: 3.3245895465215045

Epoch:   16/20    Loss: 3.307146739959717

Epoch:   17/20    Loss: 3.2169762312873336

Epoch:   17/20    Loss: 3.199352033138275

Epoch:   17/20    Loss: 3.122141695022583

Epoch:   17/20    Loss: 3.1728943951924644

Epoch:   17/20    Loss: 3.2372981961568197

Epoch:   17/20    Loss: 3.206293516953786

Epoch:   17/20    Loss: 3.1620909857749937

Epoch:   17/20    Loss: 3.1500430981318157

Epoch:   17/20    Loss: 3.210204145113627

Epoch:   17/20    Loss: 3.269510458310445

Epoch:   17/20    Loss: 3.2687523412704467
```

```
Epoch:   18/20    Loss: 3.165317495992361

Epoch:   18/20    Loss: 3.1481497581799824

Epoch:   18/20    Loss: 3.074621008237203

Epoch:   18/20    Loss: 3.119136564731598

Epoch:   18/20    Loss: 3.187447233994802

Epoch:   18/20    Loss: 3.15887544075648

Epoch:   18/20    Loss: 3.109820037682851

Epoch:   18/20    Loss: 3.1026255170504253

Epoch:   18/20    Loss: 3.166762925783793

Epoch:   18/20    Loss: 3.2217097155253094

Epoch:   18/20    Loss: 3.2160353620847064

Epoch:   19/20    Loss: 3.1124527873086536

Epoch:   19/20    Loss: 3.098372828960419

Epoch:   19/20    Loss: 3.022192854086558

Epoch:   19/20    Loss: 3.0760944310824074

Epoch:   19/20    Loss: 3.1348234550158183

Epoch:   19/20    Loss: 3.1109078923861184

Epoch:   19/20    Loss: 3.0608591794967652

Epoch:   19/20    Loss: 3.0550426189104716

Epoch:   19/20    Loss: 3.1238975977897643

Epoch:   19/20    Loss: 3.1667944971720376

Epoch:   19/20    Loss: 3.164069645404816

Epoch:   20/20    Loss: 3.058291680064083

Epoch:   20/20    Loss: 3.046052910486857

Epoch:   20/20    Loss: 2.980941465695699

Epoch:   20/20    Loss: 3.024784715970357

Epoch:   20/20    Loss: 3.085383850733439

Epoch:   20/20    Loss: 3.05956285238266
```

```
Epoch:   20/20    Loss: 3.0190571681658427

Epoch:   20/20    Loss: 3.007908809185028

Epoch:   20/20    Loss: 3.0757644271850584

Epoch:   20/20    Loss: 3.1104089331626894

Epoch:   20/20    Loss: 3.1168065293629965

C:\ProgramData\Anaconda3\lib\site-packages\torch\serialization.py:256: UserWarn
ing: Couldn't retrieve source code for container of type RNN. It won't be check
ed for correctness upon loading.
  "type " + obj.__name__ + ". It won't be checked "

Model Trained and Saved
```

Question: How did you decide on your model hyperparameters?

For example, did you try different sequence_lengths and find that one size made the model converge faster? What about your hidden_dim and n_layers; how did you decide on those?

In [18]:
```
helper.save_model('./save/trained_rnn', trained_rnn)
```

Answer:

I know that the training code was not supposed to be modified but I needed make the code run on my machine, which had all the latest libraries so I edited it for compatibility. Finally, with 20 epochs I was able to get the loss below 3.5. The sequence_length I just set to the average of one line, because I thought a script would be best broken down by lines. I ended up using a batch_size of 256 because I think that made things train better with respect to past weights. If I understand correctly, the detach that I had to add only comes into play after a batch completes. I wasted a lot of my GPU hours trying to get this to work on the Udacity workspace, but moving it to my machine gave me more power and freedom to figure it out.

My original learning rate of 0.0001 was working, but I increased it hoping to shave some time off the training. I experimented with a number of rates, but .0003 felt like a good middle-ground between the standard guesses for this criterion. It took a full two days for me to realize that the output size needs to be about the same size as the vocab length. But when I thought about it, it made sense that you would want to have as many choices for output as there are unique words in the original text.

After several attempts stalled out at a loss of about 4.2, I increased the hidden dimensions to 512 from an original 256. This seemed to help enough to get across the finish line. The two-layer LSTM was just a guess based off of previous guided assignments. The embedding dim is much smaller now at sixteen than what I started with, but I am still not sure that it's the right number.

Having an embedding layer that was smaller than the hidden layers was counter-intuitive, but enough googling pointed me in the right direction. I was able to up the hidden dimension considerably on my machine and still have it train at a reasonable time. This helped increase the

robustness of the model. I experimented with more than two layers, but it never seemed to produce consistent benefits. I stuck with two because that was used in a previous exercise.

Finally, it took a lot of googling to get the code to work on the latest libraries, but it was a good learning experience.

Checkpoint

After running the above training cell, your model will be saved by name, `trained_rnn`, and if you save your notebook progress, **you can pause here and come back to this code at another time**. You can resume your progress by running the next cell, which will load in our word:id dictionaries *and* load in your saved model by name!

In [86]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
import torch
import helper
import problem_unittests as tests

_, vocab_to_int, int_to_vocab, token_dict = helper.load_preprocess()
trained_rnn = helper.load_model('./save/trained_rnn')
```

Generate TV Script

With the network trained and saved, you'll use it to generate a new, "fake" Seinfeld TV script in this section.

Generate Text

To generate the text, the network needs to start with a single word and repeat its predictions until it reaches a set length. You'll be using the `generate` function to do this. It takes a word id to start with, `prime_id`, and generates a set length of text, `predict_len`. Also note that it uses topk sampling to introduce some randomness in choosing the most likely next word, given an output set of word scores!

In [113]:

```
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
import torch.nn.functional as F

def generate(rnn, prime_id, int_to_vocab, token_dict, pad_value, predict_len=100)
    """
    Generate text using the neural network
    :param decoder: The PyTorch Module that holds the trained neural network
    :param prime_id: The word id to start the first prediction
    :param int_to_vocab: Dict of word id keys to word values
    :param token_dict: Dict of puncuation tokens keys to puncuation values
    :param pad_value: The value used to pad a sequence
    :param predict_len: The length of text to generate
    :return: The generated text
    """
    rnn.eval()

    # create a sequence (batch_size=1) with the prime_id
    current_seq = np.full((1, sequence_length), pad_value)
    current_seq[-1][-1] = prime_id
    predicted = [int_to_vocab[prime_id]]

    for _ in range(predict_len):
        if train_on_gpu:
            current_seq = torch.LongTensor(current_seq).cuda()
        else:
            current_seq = torch.LongTensor(current_seq)

        # initialize the hidden state
        hidden = rnn.init_hidden(current_seq.size(0))

        # get the output of the rnn
        output, _ = rnn(current_seq, hidden)

        # get the next word probabilities
        p = F.softmax(output, dim=1).data
        #if(train_on_gpu):
            #p = p.cpu() # move to cpu

        # use top_k sampling to get the index of the next word
        top_k = 5
        p, top_i = p.topk(top_k)
        top_i = top_i.cpu().numpy().squeeze()

        # select the likely next word index with some element of randomness
        p = p.cpu().numpy().squeeze()
        word_i = np.random.choice(top_i, p=p/p.sum())

        # retrieve that word from the dictionary
        word = int_to_vocab[word_i]
        predicted.append(word)

        # the generated word becomes the next "current sequence" and the cycle co
        current_seq = current_seq.cpu()
        current_seq = np.roll(current_seq, -1, 1)
```

```
        current_seq[-1][-1] = word_i

    gen_sentences = ' '.join(predicted)

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

    # return all the sentences
    return gen_sentences
```

Generate a New Script

It's time to generate the text. Set `gen_length` to the length of TV script you want to generate and set `prime_word` to one of the following to start the prediction:

- "jerry"
- "elaine"
- "george"
- "kramer"

You can set the prime word to *any word* in our dictionary, but it's best to start with a name for generating a TV script. (You can also start with any other names you find in the original text file!)

In [114]:
```
# run the cell multiple times to get different results!
gen_length = 400 # modify the length to your preference
prime_word = 'kramer' # name for starting the script

"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
pad_word = helper.SPECIAL_WORDS['PADDING']
generated_script = generate(trained_rnn, vocab_to_int[prime_word + ':'], int_to_v
print(generated_script)
```

```
kramer: the the only one-- you------- you-- one?

kramer:(to himself, elaine) look here. this, don't do........(to jerry, then je
rry, you know how you're gonna do this.....

jerry:(to jerry) you know, i'm not gonna be able..

jerry:(trying to leave) what? you got a good one, i'm, what about the dutch?

george:(trying to get out of the question. you know, the only one who should be
here.

kramer:(pointing at george) i have to do it.

jerry: what about the defendants?

george: well, i'm gonna go.

george:(pointing) i'm sorry... i think we can get the the, uh, uh, what?

elaine: oh, no, it's the last thing, i can't do this all all the,, the new.

jerry: oh, no, no. i don't want to get the check, you get out, you have to go!
the one is the big one--- you are all the way, here...

puddy: hey, hey! hey.

george: oh. i got a little more, you know.

kramer:(to jerry) oh, you, uh, what do you want. you come to the movies, you do
n't understand. i just don't want to get it to the table) you have to have sex,
we have a little. i know you didn't have the last, uh.. i have a big piece of o
f, uh...(looks at jerry) i don't want to see the show, i think i'm not gonna ha
ve the money, the fact, what happened to your new. the saab.

kramer:(pointing at the door, the guy, uh. i'm sorry, uh. i
```

Save your favorite scripts

Once you have a script that you like (or find interesting), save it to a text file!

In [115]:
```
# save script to a text file
f =  open("generated_script_1.txt","w")
f.write(generated_script)
f.close()
```

The TV Script is Not Perfect

It's ok if the TV script doesn't make perfect sense. It should look like alternating lines of dialogue, here is one such example of a few generated lines.

Example generated script

> jerry: what about me?
>
> jerry: i don't have to wait.
>
> kramer:(to the sales table)
>
> elaine:(to jerry) hey, look at this, i'm a good doctor.
>
> newman:(to elaine) you think i have no idea of this...
>
> elaine: oh, you better take the phone, and he was a little nervous.
>
> kramer:(to the phone) hey, hey, jerry, i don't want to be a little bit.(to kramer and jerry) you can't.
>
> jerry: oh, yeah. i don't even know, i know.
>
> jerry:(to the phone) oh, i know.
>
> kramer:(laughing) you know...(to jerry) you don't know.

You can see that there are multiple characters that say (somewhat) complete sentences, but it doesn't have to be perfect! It takes quite a while to get good results, and often, you'll have to use a smaller vocabulary (and discard uncommon words), or get more data. The Seinfeld dataset is about 3.4 MB, which is big enough for our purposes; for script generation you'll want more than 1 MB of text, generally.

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 another copy as an HTML file by clicking "File" -> "Download as.."->"html". Include the "helper.py" and "problem_unittests.py" files in your submission. Once you download these files, compress them into one zip file for submission.

In [ ]:
```
```