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]:
```
# For better debugging
import os
os.environ['CUDA_LAUNCH_BLOCKING'] = "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 = (11, 50)

"""
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 fo
r word in text.split()})))

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_line)))

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 11 to 50:

george: (on an imaginary microphone) uh, no, not at this time.

jerry: well, senator, id just like to know, what you knew and when you knew it.

claire: mr. seinfeld. mr. costanza.

george: are, are you sure this is decaf? wheres the orange indicator?

claire: its missing, i have to do it in my head decaf left, regular rig ht, decaf left, regular right...its very challenging work.

jerry: can you relax, its a cup of coffee. claire is a professional wai tress.

claire: trust me george. no one has any interest in seeing you on caffe ine.

george: how come youre not doing the second show tomorrow?

jerry: well, theres this uh, woman might be coming in.

george: wait a second, wait a second, what coming in, what woman is com ing in?

jerry: i told you about laura, the girl i met in michigan?

george: no, you didnt!

jerry: i thought i told you about it, yes, she teaches political scienc e? i met her the night i did the show in lansing...

george: ha.

jerry: (looks in the creamer) theres no milk in here, what...

george: wait wait wait, what is she... (takes the milk can from jerry a nd puts it on the table) what is she like?

jerry: oh, shes really great. i mean, shes got like a real warmth about her and shes really bright and really pretty and uh... the conversation though, i mean, it was... talking with her is like talking with you, bu t, you know, obviously much better.

george: (smiling) so, you know, what, what happened?

jerry: oh, nothing happened, you know, but is was great.

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 [3]:
```
import problem_unittests as tests

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 = {}
    for idx, word in enumerate(set(text)):
        if word not in vocab_to_int:
            vocab_to_int[word] = int(idx)


    int_to_vocab = {v: k for k, v in vocab_to_int.items()}

    print(len(vocab_to_int))
    # 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)
```

```
71
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 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 [4]:
```
def token_lookup():
    """
    Generate a dict to turn punctuation into a token.
    :return: Tokenized dictionary where the key is the punctuation and t
he value is the token
    """
    # TODO: Implement Function

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

"""
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 [5]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL
"""
# pre-process training data
helper.preprocess_and_save_data(data_dir, token_lookup, create_lookup_ta
bles)
```

```
21388
```

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 [6]:
```
"""
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_preproces
s()
```

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 [7]:
```
"""
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)](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)](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 [8]:

```
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 i
n a batch
    :return: DataLoader with batched data
    """
    # TODO: Implement function
    feature_tensors = []
    target_tensors = []
    n_words = len(words)
    for i in range(n_words):
        if i + sequence_length < n_words:
            feature_tensors.append(words[i:i + sequence_length])
            target_tensors.append(words[i + sequence_length])
        else:
            break

    data = TensorDataset(torch.LongTensor(feature_tensors), torch.LongTe
nsor(target_tensors))

    return DataLoader(data, batch_size=batch_size, shuffle=True)

# there is no test for this function, but you are encouraged to create
# print statements and tests of your own
print(batch_data([1,2,3,4,5,6,7,8,9,0], 3, 7))
```

```
<torch.utils.data.dataloader.DataLoader object at 0x7f6c1a1b3978>
```

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 [9]:

```python
# 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([[ 8,  9, 10, 11, 12],
        [34, 35, 36, 37, 38],
        [ 5,  6,  7,  8,  9],
        [38, 39, 40, 41, 42],
        [26, 27, 28, 29, 30],
        [29, 30, 31, 32, 33],
        [11, 12, 13, 14, 15],
        [25, 26, 27, 28, 29],
        [21, 22, 23, 24, 25],
        [30, 31, 32, 33, 34]])

torch.Size([10])
tensor([13, 39, 10, 43, 31, 34, 16, 30, 26, 35])
```

Build the Neural Network

Implement an RNN using PyTorch's [Module class (http://pytorch.org/docs/master/nn.html#torch.nn.Module)](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 [10]:
```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, vocab_size, output_size, embedding_dim, hidden_di
m, n_layers, dropout=0.5):
        """
        Initialize the PyTorch RNN Module
        :param vocab_size: The number of input dimensions of the neural
 network (the size of the vocabulary)
        :param output_size: The number of output dimensions of the neura
l network
        :param embedding_dim: The size of embeddings, should you choose
 to use them
        :param hidden_dim: The size of the hidden layer outputs
        :param dropout: dropout to add in between LSTM/GRU layers
        """
        super(RNN, self).__init__()

        # set class variables
        self.n_layers = n_layers
        self.output_size = output_size
        self.hidden_dim = hidden_dim

        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)

        # define model layers
        self.lstm_layer = nn.LSTM(
            embedding_dim,
            hidden_dim,
            num_layers=n_layers,
            dropout=dropout,
            batch_first=True
        )
        self.output_layer = 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 l
atest hidden state
        """
        embeds = self.word_embeddings(nn_input)
        lstm_out, hidden = self.lstm_layer(embeds, hidden)
        lstm_out = lstm_out.contiguous().view(-1, self.hidden_dim)

        batch_size = nn_input.size(0)
        output = self.output_layer(lstm_out)
        output = output.view(batch_size, -1, self.output_size)
        last_batch = output[:, -1]

        return last_batch, 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)
        '''
        # 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.hidden_
dim).zero_().cuda(),
                  weight.new(self.n_layers, batch_size, self.hidden_dim)
.zero_().cuda())
        else:
            hidden = (weight.new(self.n_layers, batch_size, self.hidden_
dim).zero_(),
                      weight.new(self.n_layers, batch_size, self.hidden_
dim).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, target)
```

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 [11]: 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 netwo
rk
    :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
    """
    if train_on_gpu:
        inp, target = inp.cuda(), target.cuda()

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

    return loss.item(), h

# Note that these tests aren't completely extensive.
# they are here to act as general checks on the expected outputs of your
functions
"""
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 [12]:

```
def repackage_hidden(h):
    """Wraps hidden states in new Tensors, to detach them from their his
tory."""
    if isinstance(h, torch.Tensor):
        return h.detach()
    else:
        return tuple(repackage_hidden(v) for v in h)

"""
DON'T MODIFY ANYTHING IN THIS CELL
"""

def train_rnn(rnn, batch_size, optimizer, criterion, n_epochs, show_ever
y_n_batches=100):
    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
            hidden = repackage_hidden(hidden)
            loss, hidden = forward_back_prop(rnn, optimizer, criterion,
inputs, labels, hidden)
            # 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 [19]:

```
# Data params
# Sequence Length
sequence_length = 8 # of words in a sequence
# Batch Size
batch_size = 128

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

In [20]:

```
# Training parameters
# Number of Epochs
num_epochs = 10
# Learning Rate
learning_rate = 0.001

# Model parameters
# Vocab size
vocab_size = len(vocab_to_int)
# Output size
output_size = vocab_size
# Embedding Dimension
embedding_dim = 200
# Hidden Dimension
hidden_dim = 256
# Number of RNN Layers
n_layers = 2

# Show stats for every n number of batches
show_every_n_batches = 500
```

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 [21]:
```
"""
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=0.5)
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_epoch
s, show_every_n_batches)

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

```
Training for 10 epoch(s)...
Epoch:    1/10    Loss: 5.515188800811767

Epoch:    1/10    Loss: 4.894205183506012

Epoch:    1/10    Loss: 4.680005712985992

Epoch:    1/10    Loss: 4.556370007038116

Epoch:    1/10    Loss: 4.45140567445755

Epoch:    1/10    Loss: 4.404972780227661

Epoch:    1/10    Loss: 4.32517939043045

Epoch:    1/10    Loss: 4.31841893196106

Epoch:    1/10    Loss: 4.265971234798432

Epoch:    1/10    Loss: 4.246224465847015

Epoch:    1/10    Loss: 4.212311903953553

Epoch:    1/10    Loss: 4.201020967483521

Epoch:    1/10    Loss: 4.170561553001404

Epoch:    2/10    Loss: 4.055712251348269

Epoch:    2/10    Loss: 3.9973594183921812

Epoch:    2/10    Loss: 3.986263379096985

Epoch:    2/10    Loss: 3.9635803208351135

Epoch:    2/10    Loss: 3.9599059619903563

Epoch:    2/10    Loss: 3.94514505815506

Epoch:    2/10    Loss: 3.935329864501953

Epoch:    2/10    Loss: 3.938438714504242

Epoch:    2/10    Loss: 3.942914291381836

Epoch:    2/10    Loss: 3.952489317417145

Epoch:    2/10    Loss: 3.910309956073761

Epoch:    2/10    Loss: 3.913070430278778

Epoch:    2/10    Loss: 3.931608271598816

Epoch:    3/10    Loss: 3.8426282580545936

Epoch:    3/10    Loss: 3.7449702105522156
```

```
Epoch:    3/10    Loss: 3.7627976350784302

Epoch:    3/10    Loss: 3.7916578917503356

Epoch:    3/10    Loss: 3.7599433608055115

Epoch:    3/10    Loss: 3.754775415420532

Epoch:    3/10    Loss: 3.7655256996154787

Epoch:    3/10    Loss: 3.7686352763175965

Epoch:    3/10    Loss: 3.766996611595154

Epoch:    3/10    Loss: 3.767058864116669

Epoch:    3/10    Loss: 3.780586408138275

Epoch:    3/10    Loss: 3.773424741744995

Epoch:    3/10    Loss: 3.7779603548049927

Epoch:    4/10    Loss: 3.705124765480758

Epoch:    4/10    Loss: 3.6273978176116946

Epoch:    4/10    Loss: 3.616773958206177

Epoch:    4/10    Loss: 3.6674270362854005

Epoch:    4/10    Loss: 3.6360745553970335

Epoch:    4/10    Loss: 3.6589384570121766

Epoch:    4/10    Loss: 3.6663544912338257

Epoch:    4/10    Loss: 3.6454354720115663

Epoch:    4/10    Loss: 3.6662793803215026

Epoch:    4/10    Loss: 3.665962187767029

Epoch:    4/10    Loss: 3.6781682834625244

Epoch:    4/10    Loss: 3.676956202983856

Epoch:    4/10    Loss: 3.6632711391448973

Epoch:    5/10    Loss: 3.60518525849924

Epoch:    5/10    Loss: 3.5266586937904356

Epoch:    5/10    Loss: 3.5491776027679443

Epoch:    5/10    Loss: 3.546335247039795

Epoch:    5/10    Loss: 3.5509201469421385
```

```
Epoch:    5/10    Loss: 3.558977521896362
Epoch:    5/10    Loss: 3.5540020489692687
Epoch:    5/10    Loss: 3.589643236160278
Epoch:    5/10    Loss: 3.5868148427009583
Epoch:    5/10    Loss: 3.586656662464142
Epoch:    5/10    Loss: 3.584798780441284
Epoch:    5/10    Loss: 3.6107240200042723
Epoch:    5/10    Loss: 3.6055909996032716
Epoch:    6/10    Loss: 3.5303016454442737
Epoch:    6/10    Loss: 3.464726659297943
Epoch:    6/10    Loss: 3.4647241320610047
Epoch:    6/10    Loss: 3.4753291630744934
Epoch:    6/10    Loss: 3.470268227100372
Epoch:    6/10    Loss: 3.496699033737183
Epoch:    6/10    Loss: 3.4944398765563967
Epoch:    6/10    Loss: 3.519511894226074
Epoch:    6/10    Loss: 3.505395049571991
Epoch:    6/10    Loss: 3.520570902347565
Epoch:    6/10    Loss: 3.5238463711738586
Epoch:    6/10    Loss: 3.541940945625305
Epoch:    6/10    Loss: 3.5522071013450622
Epoch:    7/10    Loss: 3.470562788232069
Epoch:    7/10    Loss: 3.403359667301178
Epoch:    7/10    Loss: 3.4054346995353697
Epoch:    7/10    Loss: 3.414729241371155
Epoch:    7/10    Loss: 3.425702956676483
Epoch:    7/10    Loss: 3.4432071204185486
Epoch:    7/10    Loss: 3.446086753845215
```

```
Epoch:    7/10    Loss: 3.4525076165199278

Epoch:    7/10    Loss: 3.459822273731232

Epoch:    7/10    Loss: 3.4869738287925722

Epoch:    7/10    Loss: 3.4988680334091184

Epoch:    7/10    Loss: 3.496957610607147

Epoch:    7/10    Loss: 3.4960529704093934

Epoch:    8/10    Loss: 3.42372932498054

Epoch:    8/10    Loss: 3.3471236605644226

Epoch:    8/10    Loss: 3.3584524660110473

Epoch:    8/10    Loss: 3.3678841986656187

Epoch:    8/10    Loss: 3.391036041736603

Epoch:    8/10    Loss: 3.3893140263557435

Epoch:    8/10    Loss: 3.411256212234497

Epoch:    8/10    Loss: 3.4137536835670472

Epoch:    8/10    Loss: 3.409756651878357

Epoch:    8/10    Loss: 3.4282371163368226

Epoch:    8/10    Loss: 3.4518254375457764

Epoch:    8/10    Loss: 3.4454548215866088

Epoch:    8/10    Loss: 3.460259379863739

Epoch:    9/10    Loss: 3.3755362506252324

Epoch:    9/10    Loss: 3.306753198623657

Epoch:    9/10    Loss: 3.3166620497703554

Epoch:    9/10    Loss: 3.327660037994385

Epoch:    9/10    Loss: 3.3482672100067137

Epoch:    9/10    Loss: 3.3388496203422546

Epoch:    9/10    Loss: 3.381403299331665

Epoch:    9/10    Loss: 3.3726201076507567

Epoch:    9/10    Loss: 3.36599197101593

Epoch:    9/10    Loss: 3.4029546031951905
```

```
Epoch:    9/10    Loss: 3.4057063155174254

Epoch:    9/10    Loss: 3.4091373586654665

Epoch:    9/10    Loss: 3.4280067586898806

Epoch:   10/10    Loss: 3.3318529320944203

Epoch:   10/10    Loss: 3.2924032731056214

Epoch:   10/10    Loss: 3.275689582824707

Epoch:   10/10    Loss: 3.3046683926582334

Epoch:   10/10    Loss: 3.3120004024505616

Epoch:   10/10    Loss: 3.3203927888870237

Epoch:   10/10    Loss: 3.330588330745697

Epoch:   10/10    Loss: 3.3430827460289003

Epoch:   10/10    Loss: 3.3565797243118287

Epoch:   10/10    Loss: 3.3760718536376952

Epoch:   10/10    Loss: 3.368762094974518

Epoch:   10/10    Loss: 3.3841377515792845

Epoch:   10/10    Loss: 3.3717826838493345

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?

Answer:

We started with hyperparameters that were too huge, thus the learning process was really slow and did not converge after 4 epochs. Through several attempts, we decreased/increased the following hyperparameters as followed:

- epochs: from 4 to 10
- sequence_length: from 10 to 8
- batch size: from 256 to 128
- embedding dim: from 300 to 200
- hidden dim: from 512 to 256

With these final hyperparameters, we finally obtained convergence with a loss under 3.5.

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 [22]:
```
"""
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 [23]:
```
"""
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 net
work
    :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 valu
es
    :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.numpy().squeeze()

        # select the likely next word index with some element of randomn
ess
        p = p.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)
```

```
        if(train_on_gpu):
            current_seq = current_seq.cpu() # move to cpu
        # the generated word becomes the next "current sequence" and the
cycle can continue
        if train_on_gpu:
            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 [24]:

```
# run the cell multiple times to get different results!
gen_length = 400 # modify the length to your preference
prime_word = 'jerry' # 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_vocab, token_dict, vocab_to_int[pad_word], gen_length)
print(generated_script)
```

jerry:

kramer: well, it's not a good thing.

george:(to kramer) i think you're going to be a good guy.

jerry: i don't know, i don't know.

george:(to himself) you got the car!

jerry: i know. i mean, i have no idea that i was doing the same exercis es, i can't believe it.

george:(to kramer) you know, it's a little problem.

elaine: what?

kramer: no, no. no. i don't have to do it, but i have a lot of money.

elaine: what is the point of the exterminator?

jerry: well, i got the job.

elaine: oh, yeah.

jerry: well, i got a call.

kramer: well, what do you think? i mean, i have a little nervous.

jerry:(looking at jerry) you can't believe it. you know what you're doi ng here?

elaine:(from a very accent) i mean, i just want to go to the bathroom.

jerry: oh, you know, i'm sorry, i don't think so. i don't know what hap pened. you can do this.(jerry looks at the woman in the air and walks a way) i don't want to hear this!

elaine:(looking at his watch) oh, no!

kramer: well, you know, i think i should be going to do that.

elaine: what do you mean, that i'm a good idea for you.

jerry:(to george) i don't want you chuckle.

kramer: well, i'm going to the hospital, and you didn't get it.

jerry: you know what? i just got a little depressed.

george:(looking around, and starts dancing back)

kramer: oh, no. i just want the tape.

jerry: you can't do it?

george:(from the phone) oh my

Save your favorite scripts

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

In [25]:
```
# 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.