TV Script Generation

In this project, you'll generate your own Seinfeld TV scripts using RNNs. You'll be using part of the Seinfeld dataset 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]:

```
from workspace_utils import keep_awake, active_session
```

In [2]:

```
"""
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 [3]:

```
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 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 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
find us, they dont know where we are. (on an imaginary phone) did you ring?, i cant find him. wher
```

```
e 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, right? you take the shower, you get all ready, get the cash, get
your friends, the car, the spot, the reservation...then youre standing around, what do you do? you
go we gotta be getting back. once youre out, you wanna get back! you wanna go to sleep, you wanna
get up, you wanna go out again tomorrow, right? where ever you are in life, its my feeling, youve
gotta go.

jerry: (pointing at georges shirt) see, to me, that button is in the worst possible spot. the seco
nd button literally makes or breaks the shirt, look at it. its too high! its in no-mans-land. you
look like you live with your mother.

george: are you through?

jerry: you do of course try on, when you buy?

george: yes, it was purple, i liked it, i dont actually recall considering the buttons.
```

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]:

```python
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_counts = Counter(text)
    # sorting the words from most to least frequent in text occurrence
    sorted_vocab = sorted(word_counts, key=word_counts.get, reverse=True)
    # create int_to_vocab dictionaries
    int_to_vocab = {ii: word for ii, word in enumerate(sorted_vocab)}
    vocab_to_int = {word: ii for ii, word in int_to_vocab.items()}
    # 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
```

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 [5]:

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

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

    return token_dict

"""
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 lok 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 [6]:

```
"""
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 [7]:

```
"""
```

```
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 [8]:

```
"""
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 to provide a known format to our dataset; in combination with 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 [9]:

```
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

    # Convert text list to array
    words = np.array(words)

    ## TODO: Get the number of batches we can make
    total_batch_size = batch_size * sequence_length
    n_batches = len(words)//total_batch_size

    ## TODO: Keep only enough characters to make full batches
    words = words[:total_batch_size*n_batches]

    target_len = len(words) - sequence_length

    features, targets = [], []
    for idx in range(0, target_len):
        idx_end = sequence_length + idx
        feature_value = words[idx:idx + sequence_length]
        features.append(feature_value)
        target_value =  words[idx + sequence_length]
        targets.append(target_value)

    data = TensorDataset(torch.from_numpy(np.asarray(features)), torch.from_numpy(np.asarray(target
s)))
    data_loader = torch.utils.data.DataLoader(data,
                                              batch_size=batch_size)
    # 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 own
batch_data(int_text, 200, 20)
```

Out[9]:

```
<torch.utils.data.dataloader.DataLoader at 0x7f01ed1c45f8>
```

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 [10]:

```
# 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. 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 [11]:

```
import torch.nn as nn

class RNN(nn.Module):

    def __init__(self, vocab_size, output_size, embedding_dim, hidden_dim, 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 v
ocabulary)
        :param output_size: The number of output dimensions of the neural 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__()
        # TODO: Implement function

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

        # define model layers

        # define embedding layer
        self.embedding = nn.Embedding(vocab_size, embedding_dim)

        # Define the LSTM
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers,
                            dropout=dropout, batch_first=True)

        # Define a dropout layer
        self.dropout = nn.Dropout(dropout)

        # Define a final fully connected layer
        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 hidden state
        """
        # TODO: Implement function
        batch_size = nn_input.size(0)

        # embeddings and lstm_out
        nn_input = nn_input.long()
        embeds = self.embedding(nn_input)
        lstm_out, hidden = self.lstm(embeds, hidden)

        # stack up lstm outputs
        lstm_out = lstm_out.contiguous().view(-1, self.n_hidden)

        # dropout and fully-connected layer
        out = self.dropout(lstm_out)
        out = self.fc(out)

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

        # return one batch of output word scores and the hidden state
        return out, 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)
```

```
        :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_().cuda(),
                  weight.new(self.n_layers, batch_size, self.n_hidden).zero_().cuda())
        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, 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 [12]:

```
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

    # zero accumulated gradients
    rnn.zero_grad()

    # Creating new variables for the hidden state, otherwise
    # we'd backprop through the entire training history
    h = tuple([each.data for each in hidden])

    # get the output from the model
    output, h = rnn(inp, h)

    # calculate the loss and perform backprop
    loss = criterion(output.squeeze(), target.long())
    loss.backward()

    # `clip_grad_norm` helps prevent the exploding gradient problem in RNNs / LSTMs.
    clip=5 # gradient clipping
    nn.utils.clip_grad_norm_(rnn.parameters(), clip)
```

```
    optimizer.step()

    # return the loss over a batch and the hidden state produced by our model
    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 [13]:

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

def train_rnn(rnn, batch_size, optimizer, criterion, n_epochs, show_every_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
            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 uniqe 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 [14]:

```
# Data params
# Sequence Length
sequence_length = 10   # 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 [15]:

```
# 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 = 250
# Number of RNN Layers
n_layers = 2

# Show stats for every n number of batches
show_every_n_batches = 2000
```

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 [16]:

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

In [17]:

```
# Define a train_rnn function that take in new train_loader
def train_rnn_loader(rnn, batch_size, optimizer, criterion, n_epochs, train_loader, show_every_n_ba
tches=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
            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
```

In [ ]:

```
# Benchmarking on sequence length
for seq_len in keep_awake([4, 8, 16, 32, 64]):

    print('Running sequence_length of', seq_len)

    # 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()

    # create model to avoid transmitting old gradient and move to gpu if available
    train_loader = batch_data(int_text, seq_len, batch_size)

    # training model
    trained_rnn = train_rnn_loader(rnn, batch_size, optimizer, criterion, num_epochs, train_loader,
show_every_n_batches)
```

```
Running sequence_length of 4
Training for 10 epoch(s)...
Epoch:    1/10    Loss: 5.030831566095352

Epoch:    1/10    Loss: 4.68754501080513

Epoch:    1/10    Loss: 4.542055367231369
```

```
Epoch:    2/10    Loss: 4.363889659740806

Epoch:    2/10    Loss: 4.288830908894539

Epoch:    2/10    Loss: 4.260769580006599

Epoch:    3/10    Loss: 4.194367352663406

Epoch:    3/10    Loss: 4.164209968328476

Epoch:    3/10    Loss: 4.1478642616271975

Epoch:    4/10    Loss: 4.108358802480412

Epoch:    4/10    Loss: 4.0972938747406005

Epoch:    4/10    Loss: 4.085141358613968

Epoch:    5/10    Loss: 4.0522904365521555

Epoch:    5/10    Loss: 4.048977956652641

Epoch:    5/10    Loss: 4.04097435426712

Epoch:    6/10    Loss: 4.014866250912108

Epoch:    6/10    Loss: 4.012648279666901

Epoch:    6/10    Loss: 4.004262885212898

Epoch:    7/10    Loss: 3.98364559005799

Epoch:    7/10    Loss: 3.9839844017028807

Epoch:    7/10    Loss: 3.9791169451475144

Epoch:    8/10    Loss: 3.956533719603767

Epoch:    8/10    Loss: 3.9600338629484177

Epoch:    8/10    Loss: 3.9524362182617185

Epoch:    9/10    Loss: 3.935152441152708

Epoch:    9/10    Loss: 3.9382435834407805

Epoch:    9/10    Loss: 3.9317296994924544

Epoch:   10/10    Loss: 3.9185115853101666

Epoch:   10/10    Loss: 3.9203788800239563

Epoch:   10/10    Loss: 3.9164852900505065

Running sequence_length of 8
Training for 10 epoch(s)...
Epoch:    1/10    Loss: 5.049606608748436

Epoch:    1/10    Loss: 4.695447860836983

Epoch:    1/10    Loss: 4.551178238272667

Epoch:    2/10    Loss: 4.358571087265401

Epoch:    2/10    Loss: 4.278267015933991

Epoch:    2/10    Loss: 4.252616345286369

Epoch:    3/10    Loss: 4.17419450388318

Epoch:    3/10    Loss: 4.14831031870842

Epoch:    3/10    Loss: 4.130429332256317

Epoch:    4/10    Loss: 4.082922454550486

Epoch:    4/10    Loss: 4.069450147986412
```

```
Epoch:    4/10    Loss: 4.069450147980412

Epoch:    4/10    Loss: 4.058553851962089

Epoch:    5/10    Loss: 4.02795268768488

Epoch:    5/10    Loss: 4.021292750835419

Epoch:    5/10    Loss: 4.009592067718506

Epoch:    6/10    Loss: 3.9845810009648113

Epoch:    6/10    Loss: 3.9794440591335296

Epoch:    6/10    Loss: 3.9760824987888337

Epoch:    7/10    Loss: 3.9538910314214726

Epoch:    7/10    Loss: 3.946184950828552

Epoch:    7/10    Loss: 3.9426036378145217

Epoch:    8/10    Loss: 3.9235997800277387

Epoch:    8/10    Loss: 3.923160719871521

Epoch:    8/10    Loss: 3.9159933120012282

Epoch:    9/10    Loss: 3.8995088640591655

Epoch:    9/10    Loss: 3.9010807167291643

Epoch:    9/10    Loss: 3.8976487703323364

Epoch:   10/10    Loss: 3.8803668045459228

Epoch:   10/10    Loss: 3.8789091302156447

Epoch:   10/10    Loss: 3.8748503638505936

Running sequence_length of 16
Training for 10 epoch(s)...
Epoch:    1/10    Loss: 5.091920166730881

Epoch:    1/10    Loss: 4.693814112305641

Epoch:    1/10    Loss: 4.537886178374291

Epoch:    2/10    Loss: 4.347175470031811

Epoch:    2/10    Loss: 4.2684807709455495

Epoch:    2/10    Loss: 4.242186939120293

Epoch:    3/10    Loss: 4.162233731520261

Epoch:    3/10    Loss: 4.129295161724091

Epoch:    3/10    Loss: 4.118340971946716

Epoch:    4/10    Loss: 4.0693494984329295

Epoch:    4/10    Loss: 4.054982654690742

Epoch:    4/10    Loss: 4.0440035054683685

Epoch:    5/10    Loss: 4.009927307846337

Epoch:    5/10    Loss: 4.003524034380913

Epoch:    5/10    Loss: 3.993278011202812

Epoch:    6/10    Loss: 3.9660814145240644

Epoch:    6/10    Loss: 3.9605536019802092

Epoch:    6/10    Loss: 3.9544475165605544
```

```
Epoch:     7/10    Loss: 3.9325608340982408

Epoch:     7/10    Loss: 3.9289200222492218

Epoch:     7/10    Loss: 3.921965164065361

Epoch:     8/10    Loss: 3.903936480702319

Epoch:     8/10    Loss: 3.9015135813951494

Epoch:     8/10    Loss: 3.8917106899023057

Epoch:     9/10    Loss: 3.8805110340949933

Epoch:     9/10    Loss: 3.874718000173569

Epoch:     9/10    Loss: 3.8698871104717254

Epoch:    10/10    Loss: 3.859483825552097

Epoch:    10/10    Loss: 3.85573427605629

Epoch:    10/10    Loss: 3.8509625667333602

Running sequence_length of 32
Training for 10 epoch(s)...
Epoch:     1/10    Loss: 5.1012545731067656

Epoch:     1/10    Loss: 4.706112285375595

Epoch:     1/10    Loss: 4.547626896500588

Epoch:     2/10    Loss: 4.351454342784746

Epoch:     2/10    Loss: 4.271341796755791

Epoch:     2/10    Loss: 4.239795055866241

Epoch:     3/10    Loss: 4.161820475357631

Epoch:     3/10    Loss: 4.130325224518776

Epoch:     3/10    Loss: 4.1170843663215635

Epoch:     4/10    Loss: 4.066736377423773

Epoch:     4/10    Loss: 4.053010011434555

Epoch:     4/10    Loss: 4.045500982284546

Epoch:     5/10    Loss: 4.001764361565915

Epoch:     5/10    Loss: 3.9972387033700945

Epoch:     5/10    Loss: 3.9929261968135834

Epoch:     6/10    Loss: 3.960802937704327

Epoch:     6/10    Loss: 3.953010491371155

Epoch:     6/10    Loss: 3.954421490430832

Epoch:     7/10    Loss: 3.9241208891620696

Epoch:     7/10    Loss: 3.924793528676033

Epoch:     7/10    Loss: 3.9220856209993364

Epoch:     8/10    Loss: 3.8965663734115874

Epoch:     8/10    Loss: 3.8959193547964097

Epoch:     8/10    Loss: 3.900864527821541

Epoch:     9/10    Loss: 3.874624603830792

Epoch:     9/10    Loss: 3.8756879998445513
```

```
Epoch:     9/10     Loss: 3.8756879998445513

Epoch:     9/10     Loss: 3.877947264313698

Epoch:    10/10     Loss: 3.854515409615426

Epoch:    10/10     Loss: 3.8563224095106126

Epoch:    10/10     Loss: 3.8551054822206496

Running sequence_length of 64
Training for 10 epoch(s)...
Epoch:     1/10     Loss: 5.07340676176548

Epoch:     1/10     Loss: 4.679546286225319

Epoch:     1/10     Loss: 4.523088696479797

Epoch:     2/10     Loss: 4.326057492970028

Epoch:     2/10     Loss: 4.259556070327759

Epoch:     2/10     Loss: 4.230631127595902

Epoch:     3/10     Loss: 4.145313642721986

Epoch:     3/10     Loss: 4.125341555476188

Epoch:     3/10     Loss: 4.109908592581749

Epoch:     4/10     Loss: 4.054811417298889
```

In [18]:

```
# Benchmarking on sequence length - further zooming into range of 10-14 (It seems best settings fa
ll between 8-16)
for seq_len in keep_awake([10, 12, 14]):
    print('Running sequence_length of', seq_len)

    # 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()

    # create model to avoid transmitting old gradient and move to gpu if available
    train_loader = batch_data(int_text, seq_len, batch_size)

    # training model
    trained_rnn = train_rnn_loader(rnn, batch_size, optimizer, criterion, num_epochs, train_loader,
show_every_n_batches)
```

```
Running sequence_length of 10
Training for 10 epoch(s)...
Epoch:     1/10     Loss: 5.0668578966856

Epoch:     1/10     Loss: 4.690174400568009

Epoch:     1/10     Loss: 4.538383078098297

Epoch:     2/10     Loss: 4.351285161179197

Epoch:     2/10     Loss: 4.272015293240547

Epoch:     2/10     Loss: 4.244721225500107

Epoch:     3/10     Loss: 4.1705061053456225

Epoch:     3/10     Loss: 4.139677664637565

Epoch:     3/10     Loss: 4.128689180016518
```

```
Epoch:    4/10    Loss: 4.076292237626334

Epoch:    4/10    Loss: 4.064048186302185

Epoch:    4/10    Loss: 4.057484398245811

Epoch:    5/10    Loss: 4.017579913824223

Epoch:    5/10    Loss: 4.0088310234546665

Epoch:    5/10    Loss: 4.009409873485565

Epoch:    6/10    Loss: 3.9728626814916677

Epoch:    6/10    Loss: 3.968218847155571

Epoch:    6/10    Loss: 3.969567076444626

Epoch:    7/10    Loss: 3.9406504100865307

Epoch:    7/10    Loss: 3.9359531824588774

Epoch:    7/10    Loss: 3.93731982421875

Epoch:    8/10    Loss: 3.910109988174232

Epoch:    8/10    Loss: 3.911645032405853

Epoch:    8/10    Loss: 3.9113847172260283

Epoch:    9/10    Loss: 3.889679214123658

Epoch:    9/10    Loss: 3.8902684569358827

Epoch:    9/10    Loss: 3.8875526061058046

Epoch:   10/10    Loss: 3.868305193487556

Epoch:   10/10    Loss: 3.872440301179886

Epoch:   10/10    Loss: 3.8675780574083327

Running sequence_length of 12
Training for 10 epoch(s)...
Epoch:    1/10    Loss: 5.058011878848076

Epoch:    1/10    Loss: 4.676783779859543

Epoch:    1/10    Loss: 4.522633912444115

Epoch:    2/10    Loss: 4.339230401002704

Epoch:    2/10    Loss: 4.264359011888504

Epoch:    2/10    Loss: 4.235177852988243

Epoch:    3/10    Loss: 4.163479671078624

Epoch:    3/10    Loss: 4.12966040790081

Epoch:    3/10    Loss: 4.115615260243416

Epoch:    4/10    Loss: 4.072329422556576

Epoch:    4/10    Loss: 4.049632461071014

Epoch:    4/10    Loss: 4.043666745901108

Epoch:    5/10    Loss: 4.013038456057245

Epoch:    5/10    Loss: 3.99980165207386

Epoch:    5/10    Loss: 3.996955881118774

Epoch:    6/10    Loss: 3.9687890773610754

Epoch:    6/10    Loss: 3.960733065366745
```

```
Epoch:    6/10    Loss: 3.959843415737152

Epoch:    7/10    Loss: 3.9343799629417697

Epoch:    7/10    Loss: 3.9313077394962312

Epoch:    7/10    Loss: 3.9321419138908387

Epoch:    8/10    Loss: 3.9090073192470096

Epoch:    8/10    Loss: 3.9063141895532607

Epoch:    8/10    Loss: 3.902712106823921

Epoch:    9/10    Loss: 3.8842882917474437

Epoch:    9/10    Loss: 3.881952800631523

Epoch:    9/10    Loss: 3.882080427646637

Epoch:   10/10    Loss: 3.8646610721860153

Epoch:   10/10    Loss: 3.8620883461236954

Epoch:   10/10    Loss: 3.864724126696587

Running sequence_length of 14
Training for 10 epoch(s)...
Epoch:    1/10    Loss: 5.073086328744888

Epoch:    1/10    Loss: 4.674500318169594

Epoch:    1/10    Loss: 4.524098350405693

Epoch:    2/10    Loss: 4.34246194245729

Epoch:    2/10    Loss: 4.263204861998558

Epoch:    2/10    Loss: 4.2403300926685334

Epoch:    3/10    Loss: 4.162836945939798

Epoch:    3/10    Loss: 4.131267718315124

Epoch:    3/10    Loss: 4.123518665194512

Epoch:    4/10    Loss: 4.071368298338682

Epoch:    4/10    Loss: 4.05610164141655

Epoch:    4/10    Loss: 4.052835352301598

Epoch:    5/10    Loss: 4.014455322966126

Epoch:    5/10    Loss: 4.004852997660637

Epoch:    5/10    Loss: 4.003231623649597

Epoch:    6/10    Loss: 3.970667913873574

Epoch:    6/10    Loss: 3.966126459956169

Epoch:    6/10    Loss: 3.960816391348839

Epoch:    7/10    Loss: 3.934492225134175

Epoch:    7/10    Loss: 3.929933456301689

Epoch:    7/10    Loss: 3.9303101115226746

Epoch:    8/10    Loss: 3.907328752557047

Epoch:    8/10    Loss: 3.9062226849794386

Epoch:    8/10    Loss: 3.906948076963425
```

```
Epoch:    9/10    Loss: 3.8821900580273634

Epoch:    9/10    Loss: 3.881837451338768

Epoch:    9/10    Loss: 3.880782582163811

Epoch:   10/10    Loss: 3.8643681292683842

Epoch:   10/10    Loss: 3.85839375936985

Epoch:   10/10    Loss: 3.860123037099838
```

In [19]:

```
# Setting optimum Sequence Length from benchmarking above - best sequence_length = 12
sequence_length = 12  # of words in a sequence

# Benchmarking on hidden_dim
for hd in keep_awake([64, 128, 256]):
    print('Running hidden_dim of', hd)

    # create model and move to gpu if available
    rnn = RNN(vocab_size, output_size, embedding_dim, hd, 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()

    # create model to avoid transmitting old gradient and move to gpu if available
    train_loader = batch_data(int_text, sequence_length, batch_size)

    # training model
    trained_rnn = train_rnn_loader(rnn, batch_size, optimizer, criterion, num_epochs, train_loader,
show_every_n_batches)
```

```
Running hidden_dim of 64
Training for 10 epoch(s)...
Epoch:    1/10    Loss: 5.450915069818497

Epoch:    1/10    Loss: 5.05469793009758

Epoch:    1/10    Loss: 4.8962332417964936

Epoch:    2/10    Loss: 4.708435899142117

Epoch:    2/10    Loss: 4.619435594558716

Epoch:    2/10    Loss: 4.582438886642456

Epoch:    3/10    Loss: 4.514892336886329

Epoch:    3/10    Loss: 4.4792671085596085

Epoch:    3/10    Loss: 4.455360592722893

Epoch:    4/10    Loss: 4.422851854500315

Epoch:    4/10    Loss: 4.407734110236168

Epoch:    4/10    Loss: 4.390565774440765

Epoch:    5/10    Loss: 4.365063293353252

Epoch:    5/10    Loss: 4.357782131314278

Epoch:    5/10    Loss: 4.344465579032898

Epoch:    6/10    Loss: 4.325176413053595

Epoch:    6/10    Loss: 4.321956451654434

Epoch:    6/10    Loss: 4.3080144689083095
```

```
Epoch:    7/10    Loss: 4.2945277937444

Epoch:    7/10    Loss: 4.293518676042557

Epoch:    7/10    Loss: 4.2817509278059

Epoch:    8/10    Loss: 4.267463888901236

Epoch:    8/10    Loss: 4.2722502447366715

Epoch:    8/10    Loss: 4.259157072067261

Epoch:    9/10    Loss: 4.249334938987843

Epoch:    9/10    Loss: 4.249558662414551

Epoch:    9/10    Loss: 4.24083656001091

Epoch:   10/10    Loss: 4.231893526536865

Epoch:   10/10    Loss: 4.237226926922798

Epoch:   10/10    Loss: 4.225140819311142

Running hidden_dim of 128
Training for 10 epoch(s)...
Epoch:    1/10    Loss: 5.2200179708003995

Epoch:    1/10    Loss: 4.83263643348217

Epoch:    1/10    Loss: 4.674833074212074

Epoch:    2/10    Loss: 4.488777851325187

Epoch:    2/10    Loss: 4.404556245207787

Epoch:    2/10    Loss: 4.375503723740578

Epoch:    3/10    Loss: 4.3106753594893865

Epoch:    3/10    Loss: 4.277101368069649

Epoch:    3/10    Loss: 4.260208657979965

Epoch:    4/10    Loss: 4.220753558466023

Epoch:    4/10    Loss: 4.207875957965851

Epoch:    4/10    Loss: 4.197829115509987

Epoch:    5/10    Loss: 4.167665954456413

Epoch:    5/10    Loss: 4.1603099970817565

Epoch:    5/10    Loss: 4.150794974327088

Epoch:    6/10    Loss: 4.127928361734775

Epoch:    6/10    Loss: 4.1245164560079575

Epoch:    6/10    Loss: 4.113314685821533

Epoch:    7/10    Loss: 4.098669770763871

Epoch:    7/10    Loss: 4.092253068685531

Epoch:    7/10    Loss: 4.090677603363991

Epoch:    8/10    Loss: 4.071467801822767

Epoch:    8/10    Loss: 4.070384208917618

Epoch:    8/10    Loss: 4.068081882357597

Epoch:    9/10    Loss: 4.051879157646156
```

```
Epoch:    9/10    Loss: 4.050679687738419

Epoch:    9/10    Loss: 4.047003970861435

Epoch:   10/10    Loss: 4.034491581302835

Epoch:   10/10    Loss: 4.035498074769974

Epoch:   10/10    Loss: 4.028376005411148

Running hidden_dim of 256
Training for 10 epoch(s)...
Epoch:    1/10    Loss: 5.0636933841705325

Epoch:    1/10    Loss: 4.675021026968956

Epoch:    1/10    Loss: 4.526718683719635

Epoch:    2/10    Loss: 4.339026029064027

Epoch:    2/10    Loss: 4.258739171862603

Epoch:    2/10    Loss: 4.2331530041694645

Epoch:    3/10    Loss: 4.156289314732192

Epoch:    3/10    Loss: 4.126799794793129

Epoch:    3/10    Loss: 4.113949194550514

Epoch:    4/10    Loss: 4.066601488871444

Epoch:    4/10    Loss: 4.047891209483146

Epoch:    4/10    Loss: 4.040877478003502

Epoch:    5/10    Loss: 4.008396401361824

Epoch:    5/10    Loss: 3.995934213757515

Epoch:    5/10    Loss: 3.9896563725471497

Epoch:    6/10    Loss: 3.960448355963199

Epoch:    6/10    Loss: 3.954650399684906

Epoch:    6/10    Loss: 3.9515557795763017

Epoch:    7/10    Loss: 3.9250092325922847

Epoch:    7/10    Loss: 3.923574970126152

Epoch:    7/10    Loss: 3.921816256642342

Epoch:    8/10    Loss: 3.897331862564061

Epoch:    8/10    Loss: 3.899619082570076

Epoch:    8/10    Loss: 3.892997024536133

Epoch:    9/10    Loss: 3.874541268069908

Epoch:    9/10    Loss: 3.8742080159187315

Epoch:    9/10    Loss: 3.874335947871208

Epoch:   10/10    Loss: 3.8531923547069864

Epoch:   10/10    Loss: 3.856536533474922

Epoch:   10/10    Loss: 3.8493074253797532
```

In [20]:

```
# Setting optimum Sequence Length from benchmarking above - best sequence length = 12
```

```
sequence_length = 12  # of words in a sequence

# Setting optimum hidden_dim from benchmarking above - best hidden_dim = 256
hidden_dim = 256

# Benchmarking on n_layers
for n_lay in keep_awake([1, 2, 3, 4]):
    print('Running n_layers of', n_lay)

    # create model and move to gpu if available
    rnn = RNN(vocab_size, output_size, embedding_dim, hidden_dim, n_lay, 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()

    # create model to avoid transmitting old gradient and move to gpu if available
    train_loader = batch_data(int_text, sequence_length, batch_size)

    # training model
    trained_rnn = train_rnn_loader(rnn, batch_size, optimizer, criterion, num_epochs, train_loader,
show_every_n_batches)
```

```
Running n_layers of 1
```

```
/opt/conda/lib/python3.6/site-packages/torch/nn/modules/rnn.py:38: UserWarning: dropout option add
s dropout after all but last recurrent layer, so non-zero dropout expects num_layers greater than
1, but got dropout=0.5 and num_layers=1
  "num_layers={}".format(dropout, num_layers))
```

```
Training for 10 epoch(s)...
Epoch:    1/10    Loss: 4.827993685960769

Epoch:    1/10    Loss: 4.4864686797857285

Epoch:    1/10    Loss: 4.395602733135223

Epoch:    2/10    Loss: 4.21828943984498

Epoch:    2/10    Loss: 4.143759577155113

Epoch:    2/10    Loss: 4.125962798953056

Epoch:    3/10    Loss: 4.041298398942551

Epoch:    3/10    Loss: 4.004116086125374

Epoch:    3/10    Loss: 3.9939377108812333

Epoch:    4/10    Loss: 3.9426750325399222

Epoch:    4/10    Loss: 3.9163957740068436

Epoch:    4/10    Loss: 3.9111913163661955

Epoch:    5/10    Loss: 3.872926975593812

Epoch:    5/10    Loss: 3.857958855509758

Epoch:    5/10    Loss: 3.852050626039505

Epoch:    6/10    Loss: 3.8185493116162847

Epoch:    6/10    Loss: 3.804795140624046

Epoch:    6/10    Loss: 3.7958893895149233

Epoch:    7/10    Loss: 3.7746958854271777

Epoch:    7/10    Loss: 3.763240446329117

Epoch:    7/10    Loss: 3.757200853705406
```

```
Epoch:    8/10    Loss: 3.7374799145198665

Epoch:    8/10    Loss: 3.7252181128263473

Epoch:    8/10    Loss: 3.7197800149917604

Epoch:    9/10    Loss: 3.704984308135151

Epoch:    9/10    Loss: 3.694954185128212

Epoch:    9/10    Loss: 3.688948273420334

Epoch:   10/10    Loss: 3.674578393621596

Epoch:   10/10    Loss: 3.6641110084056856

Epoch:   10/10    Loss: 3.6565981377363204

Running n_layers of 2
Training for 10 epoch(s)...
Epoch:    1/10    Loss: 5.081298463821411

Epoch:    1/10    Loss: 4.694989518404007

Epoch:    1/10    Loss: 4.536349990963936

Epoch:    2/10    Loss: 4.344096155898238

Epoch:    2/10    Loss: 4.264375121712685

Epoch:    2/10    Loss: 4.240461059451103

Epoch:    3/10    Loss: 4.164154752922768

Epoch:    3/10    Loss: 4.133409495472908

Epoch:    3/10    Loss: 4.121259369015694

Epoch:    4/10    Loss: 4.070820680781368

Epoch:    4/10    Loss: 4.056176903605461

Epoch:    4/10    Loss: 4.0450246813297275

Epoch:    5/10    Loss: 4.011454758349203

Epoch:    5/10    Loss: 3.998626291394234

Epoch:    5/10    Loss: 3.993170464515686

Epoch:    6/10    Loss: 3.968254062282592

Epoch:    6/10    Loss: 3.9581048451662064

Epoch:    6/10    Loss: 3.951480159878731

Epoch:    7/10    Loss: 3.9314492004229518

Epoch:    7/10    Loss: 3.9262450256347656

Epoch:    7/10    Loss: 3.922018151640892

Epoch:    8/10    Loss: 3.9042625537632527

Epoch:    8/10    Loss: 3.8993201638460158

Epoch:    8/10    Loss: 3.893429371237755

Epoch:    9/10    Loss: 3.880107798017171

Epoch:    9/10    Loss: 3.8803913021087646

Epoch:    9/10    Loss: 3.8723696173429487

Epoch:   10/10    Loss: 3.8601538254305012

Epoch:   10/10    Loss: 3.8565680408477783
```

```
Epoch:   10/10    Loss: 3.8484560183286667

Running n_layers of 3
Training for 10 epoch(s)...
Epoch:    1/10    Loss: 5.242059007048607

Epoch:    1/10    Loss: 4.807429643034935

Epoch:    1/10    Loss: 4.632182552576065

Epoch:    2/10    Loss: 4.4305260861471885

Epoch:    2/10    Loss: 4.343110102653504

Epoch:    2/10    Loss: 4.307744821667671

Epoch:    3/10    Loss: 4.232041984202291

Epoch:    3/10    Loss: 4.198928831219673

Epoch:    3/10    Loss: 4.178389496564865

Epoch:    4/10    Loss: 4.132098517010203

Epoch:    4/10    Loss: 4.115476361513138

Epoch:    4/10    Loss: 4.106568762660027

Epoch:    5/10    Loss: 4.070355168787681

Epoch:    5/10    Loss: 4.05994928741455

Epoch:    5/10    Loss: 4.050463135838509

Epoch:    6/10    Loss: 4.0241145611453915

Epoch:    6/10    Loss: 4.018373151898384

Epoch:    6/10    Loss: 4.009080424547196

Epoch:    7/10    Loss: 3.9867977244340715

Epoch:    7/10    Loss: 3.9823520669937134

Epoch:    7/10    Loss: 3.9791020900011063

Epoch:    8/10    Loss: 3.9578976847908716

Epoch:    8/10    Loss: 3.9540018254518507

Epoch:    8/10    Loss: 3.948608466744423

Epoch:    9/10    Loss: 3.9313439666197567

Epoch:    9/10    Loss: 3.9288924371004104

Epoch:    9/10    Loss: 3.9252723363637925

Epoch:   10/10    Loss: 3.9097179286824

Epoch:   10/10    Loss: 3.909459484219551

Epoch:   10/10    Loss: 3.905462726354599

Running n_layers of 4
Training for 10 epoch(s)...
Epoch:    1/10    Loss: 5.458377731442451

Epoch:    1/10    Loss: 4.919169429659844

Epoch:    1/10    Loss: 4.729561921954155

Epoch:    2/10    Loss: 4.523982328674043

Epoch:    2/10    Loss: 4.432837570309639
```

```
Epoch:    2/10    Loss: 4.386602559924126

Epoch:    3/10    Loss: 4.3017772992505705

Epoch:    3/10    Loss: 4.271074437618256

Epoch:    3/10    Loss: 4.244830382823944

Epoch:    4/10    Loss: 4.190975593126638

Epoch:    4/10    Loss: 4.178444544196129

Epoch:    4/10    Loss: 4.160592354416847

Epoch:    5/10    Loss: 4.1225354640054395

Epoch:    5/10    Loss: 4.121228747606278

Epoch:    5/10    Loss: 4.105500072836876

Epoch:    6/10    Loss: 4.075368629351143

Epoch:    6/10    Loss: 4.078255465745926

Epoch:    6/10    Loss: 4.069279160261154

Epoch:    7/10    Loss: 4.040851798884246

Epoch:    7/10    Loss: 4.04360165476799

Epoch:    7/10    Loss: 4.029263930797577

Epoch:    8/10    Loss: 4.008396769504605

Epoch:    8/10    Loss: 4.013133779764176

Epoch:    8/10    Loss: 3.9979268600940703

Epoch:    9/10    Loss: 3.9806116575645247

Epoch:    9/10    Loss: 3.987957759022713

Epoch:    9/10    Loss: 3.9766521159410475

Epoch:   10/10    Loss: 3.9578776147648385

Epoch:   10/10    Loss: 3.9658851791620253

Epoch:   10/10    Loss: 3.95566320168972
```

In [19]:

```
with active_session():
    # Setting optimum Sequence Length from benchmarking above - best sequence_length = 12
    sequence_length = 12  # of words in a sequence

    # Setting optimum hidden_dim from benchmarking above - best hidden_dim = 256
    hidden_dim = 256

    # Setting optimum n_layers from benchmarking above - best n_layers = 2
    n_layers = 2

    # Setting 20 training epochs
    num_epochs = 20

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

    # 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)
```

```
    optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)
    criterion = nn.CrossEntropyLoss()

    # training the model
    trained_rnn = train_rnn_loader(rnn, batch_size, optimizer, criterion, num_epochs, train_loader,
show_every_n_batches)

    # 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: 5.066192290782928

Epoch:    1/20    Loss: 4.680339401006699

Epoch:    1/20    Loss: 4.5332785918712615

Epoch:    2/20    Loss: 4.342226116670955

Epoch:    2/20    Loss: 4.2649547395706175

Epoch:    2/20    Loss: 4.23646379172802

Epoch:    3/20    Loss: 4.161640526193668

Epoch:    3/20    Loss: 4.131129694342613

Epoch:    3/20    Loss: 4.117991368174553

Epoch:    4/20    Loss: 4.071402985488051

Epoch:    4/20    Loss: 4.0530394848585125

Epoch:    4/20    Loss: 4.046633889198303

Epoch:    5/20    Loss: 4.008541993895025

Epoch:    5/20    Loss: 3.998980216741562

Epoch:    5/20    Loss: 3.995488040924072

Epoch:    6/20    Loss: 3.964751453486678

Epoch:    6/20    Loss: 3.96140415930748

Epoch:    6/20    Loss: 3.957322168469429

Epoch:    7/20    Loss: 3.927036804861215

Epoch:    7/20    Loss: 3.9290575021505356

Epoch:    7/20    Loss: 3.924426206231117

Epoch:    8/20    Loss: 3.8991132912179753

Epoch:    8/20    Loss: 3.903884122133255

Epoch:    8/20    Loss: 3.8987489035129546

Epoch:    9/20    Loss: 3.8765044707871645

Epoch:    9/20    Loss: 3.8776234427690506

Epoch:    9/20    Loss: 3.8774470781087875

Epoch:   10/20    Loss: 3.8554635728763222

Epoch:   10/20    Loss: 3.8618676701784134

Epoch:   10/20    Loss: 3.8596161839962004

Epoch:   11/20    Loss: 3.8383012904875904

Epoch:   11/20    Loss: 3.8444667772054673

Epoch:   11/20    Loss: 3.840735204577446
```

```
Epoch:   12/20    Loss: 3.824592731094876

Epoch:   12/20    Loss: 3.827120130300522

Epoch:   12/20    Loss: 3.8281154276132585

Epoch:   13/20    Loss: 3.8124651178067985

Epoch:   13/20    Loss: 3.8159934694767

Epoch:   13/20    Loss: 3.8086030468940737

Epoch:   14/20    Loss: 3.7990884315971587

Epoch:   14/20    Loss: 3.800112972974777

Epoch:   14/20    Loss: 3.7995765624046327

Epoch:   15/20    Loss: 3.786486430359918

Epoch:   15/20    Loss: 3.7902070560455323

Epoch:   15/20    Loss: 3.7870881778001784

Epoch:   16/20    Loss: 3.774319999884334

Epoch:   16/20    Loss: 3.7773320100307464

Epoch:   16/20    Loss: 3.7769214222431184

Epoch:   17/20    Loss: 3.7641964636367895

Epoch:   17/20    Loss: 3.7680842016935348

Epoch:   17/20    Loss: 3.7640427399873735

Epoch:   18/20    Loss: 3.755513914858582

Epoch:   18/20    Loss: 3.75815685069561

Epoch:   18/20    Loss: 3.7532143633365633

Epoch:   19/20    Loss: 3.7432524142695587

Epoch:   19/20    Loss: 3.750968714475632

Epoch:   19/20    Loss: 3.7471437066793443

Epoch:   20/20    Loss: 3.737097036681412

Epoch:   20/20    Loss: 3.7425445289611816

Epoch:   20/20    Loss: 3.736092767238617
```

```
/opt/conda/lib/python3.6/site-packages/torch/serialization.py:193: UserWarning: Couldn't retrieve source code for container of type RNN. It won't be checked for correctness upon loading.
  "type " + obj.__name__ + ". It won't be checked "
```

```
Model Trained and Saved
```

In [23]:

```python
# Re-defining RNN with nodropout at fc

class RNN_nodropout_at_fc(nn.Module):

    def __init__(self, vocab_size, output_size, embedding_dim, hidden_dim, 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 neural network
        :param embedding_dim: The size of embeddings, should you choose to use them
```

```
        :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_nodropout_at_fc, self).__init__()
        # TODO: Implement function

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

        # define model layers

        # define embedding layer
        self.embedding = nn.Embedding(vocab_size, embedding_dim)

        # Define the LSTM
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers,
                            dropout=dropout, batch_first=True)

        # Define a dropout layer
        #self.dropout = nn.Dropout(dropout)  # REMOVED dropout layer < ================= ATTENTION

        # Define a final fully connected layer
        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 hidden state
        """
        # TODO: Implement function
        batch_size = nn_input.size(0)

        # embeddings and lstm_out
        nn_input = nn_input.long()
        embeds = self.embedding(nn_input)
        lstm_out, hidden = self.lstm(embeds, hidden)

        # stack up lstm outputs
        lstm_out = lstm_out.contiguous().view(-1, self.n_hidden)

        # dropout and fully-connected layer
        #out = self.dropout(lstm_out) # REMOVED dropout before fc layer  < ===============
ATTENTION
        out = self.fc(lstm_out)

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

        # return one batch of output word scores and the hidden state
        return out, 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_().cuda(),
                  weight.new(self.n_layers, batch_size, self.n_hidden).zero_().cuda())
        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
```

```
        return hidden

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

```
Tests Passed
```

In [24]:

```
# retraining with RNN without dropout before fc layer

with active_session():
    # Setting optimum Sequence Length from benchmarking above - best sequence_length = 12
    sequence_length = 12  # of words in a sequence

    # Setting optimum hidden_dim from benchmarking above - best hidden_dim = 256
    hidden_dim = 256

    # Setting optimum n_layers from benchmarking above - best n_layers = 2
    n_layers = 2

    # Setting 20 training epochs
    num_epochs = 20

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

    # create model and move to gpu if available
    rnn = RNN_nodropout_at_fc(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_loader(rnn, batch_size, optimizer, criterion, num_epochs, train_loader,
show_every_n_batches)

    # 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: 4.928821806311607

Epoch:    1/20    Loss: 4.498210277557373

Epoch:    1/20    Loss: 4.3541893773078915

Epoch:    2/20    Loss: 4.108565516953695

Epoch:    2/20    Loss: 3.9462108371257782

Epoch:    2/20    Loss: 3.890008877277374

Epoch:    3/20    Loss: 3.8057101320761126

Epoch:    3/20    Loss: 3.732829907655716

Epoch:    3/20    Loss: 3.689158411979675

Epoch:    4/20    Loss: 3.640435306797885

Epoch:    4/20    Loss: 3.585429753303528

Epoch:    4/20    Loss: 3.5462514429092407

Epoch:    5/20    Loss: 3.5182320583346085

Epoch:    5/20    Loss: 3.4776408649682997
```

```
Epoch:    5/20    Loss: 3.448477019071579
Epoch:    6/20    Loss: 3.434588319105488
Epoch:    6/20    Loss: 3.3959565209150315
Epoch:    6/20    Loss: 3.3728151720762254
Epoch:    7/20    Loss: 3.3625906134989263
Epoch:    7/20    Loss: 3.3296041229963302
Epoch:    7/20    Loss: 3.3187886251211167
Epoch:    8/20    Loss: 3.308924808660122
Epoch:    8/20    Loss: 3.277382355093956
Epoch:    8/20    Loss: 3.2637321413755416
Epoch:    9/20    Loss: 3.261229061363274
Epoch:    9/20    Loss: 3.2300160166025162
Epoch:    9/20    Loss: 3.217708145976067
Epoch:   10/20    Loss: 3.2183501822757496
Epoch:   10/20    Loss: 3.193575009584427
Epoch:   10/20    Loss: 3.181982138991356
Epoch:   11/20    Loss: 3.184906001669525
Epoch:   11/20    Loss: 3.159032764673233
Epoch:   11/20    Loss: 3.1493333135843278
Epoch:   12/20    Loss: 3.156046742480846
Epoch:   12/20    Loss: 3.12717631649971
Epoch:   12/20    Loss: 3.1202715188264847
Epoch:   13/20    Loss: 3.128052014763108
Epoch:   13/20    Loss: 3.102875945687294
Epoch:   13/20    Loss: 3.094082383990288
Epoch:   14/20    Loss: 3.1024397030276356
Epoch:   14/20    Loss: 3.0776156919002533
Epoch:   14/20    Loss: 3.0683654643297196
Epoch:   15/20    Loss: 3.081992022287769
Epoch:   15/20    Loss: 3.060397376060486
Epoch:   15/20    Loss: 3.048845343708992
Epoch:   16/20    Loss: 3.0616916170310393
Epoch:   16/20    Loss: 3.0374632929563523
Epoch:   16/20    Loss: 3.042585025906563
Epoch:   17/20    Loss: 3.041078211770504
Epoch:   17/20    Loss: 3.017947218775749
Epoch:   17/20    Loss: 3.016059026837349
Epoch:   18/20    Loss: 3.026310077749744
```

```
Epoch:   18/20    Loss: 3.0038467433452607

Epoch:   18/20    Loss: 2.9935972268581392

Epoch:   19/20    Loss: 3.007317683712056

Epoch:   19/20    Loss: 2.993667126774788

Epoch:   19/20    Loss: 2.979897417783737

Epoch:   20/20    Loss: 2.99481245028324

Epoch:   20/20    Loss: 2.980095269203186

Epoch:   20/20    Loss: 2.964919336795807

Model Trained and Saved
```

```
/opt/conda/lib/python3.6/site-packages/torch/serialization.py:193: UserWarning: Couldn't retrieve
source code for container of type RNN_nodropout_at_fc. It won't be checked for correctness upon lo
ading.
  "type " + obj.__name__ + ". It won't be checked "
```

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:

According to Reimers and Gurevych (2017), the LSTM-Networks for sequence labeling tasks are highly sensitive to hyperparameters such as:

1. embeddings model & dims (recommended Komninos and Manandhar (2016))
2. optimizer (rec'ed NADAM),
3. gradient clipping/normalization (rec'ed gradient normalization with defined threshold of 1),
4. Dropout (rec'ed to include)

and medium sensitive to:

1. Tagging scheme (rec'ed BIO)
2. LSTM layer (rec'ed 2)
3. Batch size (rec'ed 1-32)

All the parameters below were set as the default for target parameter benchmarking: -

- sequence_length = 10
- hidden_dim = 250
- n_layers = 2
- batch_size = 128
- embedding_dim = 200
- learning_rate = 0.001
- gradient_clipping = 5
- num_epochs = 10

Only the main questions (i.e. sequence_length, hidden_dim, and n_layers) were benchmarked in order to save the GPU hours usage allowed in this DL course. The following serial settings were set for benchmarking of each parameter (quite brute-force though):-

Sequence Length = 4 (loss = 3.92, Converging = 7 epochs), 8 (3.87, 9), 16 (3.85, >10), 32 (3.86, 6), 64 (not completed),
then i zoomed into range of 8-16 for further optimization,
Sequence Length = 10 (3.87, 5), **12 (loss = 3.86, Converging = 6 epochs)** , 14 (3.86, 7)

- The shorter the sequence length, the faster the convergence (i.e. Seq length of 4, 8, and 16 converge at 7, 9, and >10 epochs, respectively). However, I feel that the longer the sequence length, the slower learning, especially at sequence_length of 64, which i terminated half way to save computing resource (although i did not timeit, couldn't repeat the experiment due to limit of gpu hours). Sequence_length of 12 was chosen to proceed to next optimization with its superior loss at epoch 10 but faster convergence compared to that of the rest.

Hidden Dimension = 64 (loss = 4.23), 128 (4.03), **256 (loss = 3.85)**, with sequence_length of 12

- It is advisable to RNN hidden dimension within 256. The hidden_dim of 256 produced the lowest loss at 10 epoch.

Number of RNN Layers = 1 (loss = 3.66), **2 (loss = 3.85)**, 3 (3.91), 4(3.96), with sequence_length of 12 and hidden_dim of 256

- Although n_layer of 1 produced the lowest loss, however, I adopted dropout option in this lstm training model - which i feel it is important to create new story instead of producing similar story as the training datasets, which an overfitted model tends to do. Here, the pytorch nn module only adds dropout after all but last recurrent layer, so non-zero dropout expects n_layers greater than 1. As recommended by most studies (including one that cited here - Reimers and Gurevych (2017)), RNN layers of 2 produced the best performance in this dataset, using rnn model with embedding and hypterparameter settings above.

Finally, to further decrease the loss, i applied **20 epochs** for the final training before subsequent Checkpoint Section. However, I still cannot reach loss below 3.5 - only getting loss of 3.74 at 20 epoch. To achieve that, I have revised the RNN network by removing the dropout layer before the fully-connected layer (but still retaining the dropout of 0.5 in the lstm layers), achieving **final loss of 2.96 at 20 epoch** eventually.

References:

1. [Reimers and Gurevych, 2017]
2. https://github.com/wojzaremba/lstm/blob/76870253cfca069477f06b7056af87f98490b6eb/main.lua#L44
3. https://machinelearningmastery.com/tune-lstm-hyperparameters-keras-time-series-forecasting/

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 [25]:

```
"""
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 [26]:

```
"""
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.numpy().squeeze()
        
        # select the likely next word index with some element of randomness
        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)
        
        # the generated word becomes the next "current sequence" and the cycle can continue
        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 [27]:

```
# run the cell multiple times to get different results!
gen_length = 400 # modify the length to your preference
prime_word = 'elaine' # name for starting the script
```

```python
"""
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)
```

```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:50: UserWarning: RNN module weights a
re not part of single contiguous chunk of memory. This means they need to be compacted at every
call, possibly greatly increasing memory usage. To compact weights again call
flatten_parameters().
```

```
elaine: circus.

hoyt: what are you talking about?

jerry: i know, but i can't tell her what the reaction is.

george: oh, well, i can't. i was watchin' quincy and heads for the courtroom!

hoyt: oh, yeah. i got a mustache, and i got a little mishap.

elaine: i can't believe it's a successful laser.

george: i can't...

george: i can't. i mean, the only thing, i can't help you.

elaine: oh, no, that's it! i don't know how to unwind! i'm sorry, it's a little phone.

george: well, i was thinking about the manual of the oldest world of acquisitions.

george: oh, no! no! it's crisp ray's, it's crispy crisp.

george: what is your connection? you know, it's a lovely day, i have to go to paris.

jerry: i can't get that water.

jerry: oh, well, i'm going to unwind.

jerry: i can't believe that. it's original ray's, but it's the emergency.

jerry: what is this noise, sir?

jerry: well, i was joking for a tractor.

george: oh, yeah.

jerry: hey, hey.

cindy: hi, koko.

stu: you think that the maid is a crime of $85.

kramer: what?

jerry: i don't know how that this is the girls.

george: oh, no. no. it's just a weapon of static.

george: what?

george: no! i can't get this plane.

george: well, i was thinking of instituting golf.

george: i can't tell you what happened to him.

kramer: yeah, well, i'm not getting married. i'm gonna get a g.

kramer: yeah!

george: what is it?

elder nazi: the contest of the robbery of windsor
```

```
older nazi: the contest of the robbery] of windsor

george: i
```

Save your favorite scripts

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

In [28]:

```
# 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.