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 [ ]:

```
"""
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 [ ]:

```
view_line_range = (0, 10)

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

print('Dataset Stats')
print('Roughly the number of unique words: {}'.format(len({word: None for word in

lines = text.split('\n')
print('Number of lines: {}'.format(len(lines)))
word_count_line = [len(line.split()) for line in lines]
print('Average number of words in each line: {}'.format(np.average(word_count_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. peo
ple...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 whe
re we are. (on an imaginary phone) did you ring?, i cant find him. where did he g
o? he didnt tell me where he was going. he must have gone out. you wanna go out yo
u get ready, you pick out the clothes, right? you take the shower, you get all rea
dy, get the cash, get your friends, the car, the spot, the reservation...then your
e standing around, what do you do? you go we gotta be getting back. once youre ou
t, you wanna get back! you wanna go to sleep, you wanna get up, you wanna go out a
gain 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 possibl
e spot. the second 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 but
tons.
```

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 [ ]:

```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
    # print(type(text))
    # print(text)
    # print(len(text))

    # counter = Counter(text)
    # # print(counter)
    # t_list = list(counter.most_common())

    # int_to_vocab = {}
    # vocab_to_int = {}

    # for index,i in enumerate(t_list):
    #     # print(index,i)
    #     w = i[0]
    #     int_to_vocab[index] = w
    #     vocab_to_int[w] = index

    # # print('int_to_vocab:\n',str(int_to_vocab)[0:100])
    # # print('vocab_to_int:\n',str(vocab_to_int)[0:100])

    # # return tuple
    # # return (int_to_vocab, vocab_to_int)
    # return (vocab_to_int,int_to_vocab)

    vocab = set(text)
    vocab_to_int, int_to_vocab = {}, {}
    for i, w in enumerate(vocab):
        vocab_to_int[w] = i
        int_to_vocab[i] = w
    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 [ ]:

```
def token_lookup():
    """
    Generate a dict to turn punctuation into a token.
    :return: Tokenized dictionary where the key is the punctuation and the value i
    """
    # TODO: Implement Function
    punct_dict = {
        '.':'||period||',
        ',':'||comma||',
        '\"':'||quote||',
        ';':'||semi_comma||',
        '!':'||exclamation_mark||',
        '?':'||question_mark||',
        '(':'||left_parentheses||',
        ')':'||right_parentheses||',
        '-':'||dash||',
        '\n':'||new_line||',
    }

    return punct_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 [ ]:

```
"""
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 [ ]:

```
"""
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 [ ]:

```
"""
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 [ ]:

```
import torch
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
    # https://pytorch.org/docs/stable/data.html


    inps = []
    tgts = []
    for i in range(len(words)-sequence_length):
        # print(i,type(words[i:(i+sequence_length)]),type(words[i+sequence_length]
        # print(i,words[i:(i+sequence_length)],words[i+sequence_length])
        inps.append(words[i:(i+sequence_length)])
        tgts.append([words[i+sequence_length]])

    inps = torch.IntTensor(inps)
    tgts = torch.IntTensor(tgts)

    dataset = TensorDataset(inps, tgts)

    # return a dataloader
    return DataLoader(dataset, batch_size=batch_size,  pin_memory=True)

# there is no test for this function, but you are encouraged to create
# print statements and tests of your own
dataloader = batch_data(list(range(100)),5,10)
for batch_ndx, batch in enumerate(dataloader):
    print(batch_ndx,'\n',batch[0],'\n',batch[1])
```

```
0
 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]], dtype=torch.int32)
 tensor([[ 5],
        [ 6],
        [ 7],
        [ 8],
        [ 9],
        [10],
        [11],
        [12],
        [13],
        [14]], dtype=torch.int32)
1
 tensor([[10, 11, 12, 13, 14],
        [11, 12, 13, 14, 15],
        [12, 13, 14, 15, 16],
        [13, 14, 15, 16, 17],
        [14, 15, 16, 17, 18],
        [15, 16, 17, 18, 19],
        [16, 17, 18, 19, 20],
        [17, 18, 19, 20, 21],
        [18, 19, 20, 21, 22],
        [19, 20, 21, 22, 23]], dtype=torch.int32)
 tensor([[15],
        [16],
        [17],
        [18],
        [19],
        [20],
        [21],
        [22],
        [23],
        [24]], dtype=torch.int32)
2
 tensor([[20, 21, 22, 23, 24],
        [21, 22, 23, 24, 25],
        [22, 23, 24, 25, 26],
        [23, 24, 25, 26, 27],
        [24, 25, 26, 27, 28],
        [25, 26, 27, 28, 29],
        [26, 27, 28, 29, 30],
        [27, 28, 29, 30, 31],
        [28, 29, 30, 31, 32],
        [29, 30, 31, 32, 33]], dtype=torch.int32)
 tensor([[25],
        [26],
        [27],
        [28],
        [29],
        [30],
        [31],
        [32],
```

```
        [33],
        [34]], dtype=torch.int32)
3
 tensor([[30, 31, 32, 33, 34],
        [31, 32, 33, 34, 35],
        [32, 33, 34, 35, 36],
        [33, 34, 35, 36, 37],
        [34, 35, 36, 37, 38],
        [35, 36, 37, 38, 39],
        [36, 37, 38, 39, 40],
        [37, 38, 39, 40, 41],
        [38, 39, 40, 41, 42],
        [39, 40, 41, 42, 43]], dtype=torch.int32)
 tensor([[35],
        [36],
        [37],
        [38],
        [39],
        [40],
        [41],
        [42],
        [43],
        [44]], dtype=torch.int32)
4
 tensor([[40, 41, 42, 43, 44],
        [41, 42, 43, 44, 45],
        [42, 43, 44, 45, 46],
        [43, 44, 45, 46, 47],
        [44, 45, 46, 47, 48],
        [45, 46, 47, 48, 49],
        [46, 47, 48, 49, 50],
        [47, 48, 49, 50, 51],
        [48, 49, 50, 51, 52],
        [49, 50, 51, 52, 53]], dtype=torch.int32)
 tensor([[45],
        [46],
        [47],
        [48],
        [49],
        [50],
        [51],
        [52],
        [53],
        [54]], dtype=torch.int32)
5
 tensor([[50, 51, 52, 53, 54],
        [51, 52, 53, 54, 55],
        [52, 53, 54, 55, 56],
        [53, 54, 55, 56, 57],
        [54, 55, 56, 57, 58],
        [55, 56, 57, 58, 59],
        [56, 57, 58, 59, 60],
        [57, 58, 59, 60, 61],
        [58, 59, 60, 61, 62],
        [59, 60, 61, 62, 63]], dtype=torch.int32)
 tensor([[55],
        [56],
        [57],
        [58],
        [59],
        [60],
```

```
        [61],
        [62],
        [63],
        [64]], dtype=torch.int32)
6
 tensor([[60, 61, 62, 63, 64],
        [61, 62, 63, 64, 65],
        [62, 63, 64, 65, 66],
        [63, 64, 65, 66, 67],
        [64, 65, 66, 67, 68],
        [65, 66, 67, 68, 69],
        [66, 67, 68, 69, 70],
        [67, 68, 69, 70, 71],
        [68, 69, 70, 71, 72],
        [69, 70, 71, 72, 73]], dtype=torch.int32)
 tensor([[65],
        [66],
        [67],
        [68],
        [69],
        [70],
        [71],
        [72],
        [73],
        [74]], dtype=torch.int32)
7
 tensor([[70, 71, 72, 73, 74],
        [71, 72, 73, 74, 75],
        [72, 73, 74, 75, 76],
        [73, 74, 75, 76, 77],
        [74, 75, 76, 77, 78],
        [75, 76, 77, 78, 79],
        [76, 77, 78, 79, 80],
        [77, 78, 79, 80, 81],
        [78, 79, 80, 81, 82],
        [79, 80, 81, 82, 83]], dtype=torch.int32)
 tensor([[75],
        [76],
        [77],
        [78],
        [79],
        [80],
        [81],
        [82],
        [83],
        [84]], dtype=torch.int32)
8
 tensor([[80, 81, 82, 83, 84],
        [81, 82, 83, 84, 85],
        [82, 83, 84, 85, 86],
        [83, 84, 85, 86, 87],
        [84, 85, 86, 87, 88],
        [85, 86, 87, 88, 89],
        [86, 87, 88, 89, 90],
        [87, 88, 89, 90, 91],
        [88, 89, 90, 91, 92],
        [89, 90, 91, 92, 93]], dtype=torch.int32)
 tensor([[85],
        [86],
        [87],
        [88],
```

```
        [89],
        [90],
        [91],
        [92],
        [93],
        [94]], dtype=torch.int32)
9
 tensor([[90, 91, 92, 93, 94],
        [91, 92, 93, 94, 95],
        [92, 93, 94, 95, 96],
        [93, 94, 95, 96, 97],
        [94, 95, 96, 97, 98]], dtype=torch.int32)
 tensor([[95],
        [96],
        [97],
        [98],
        [99]], dtype=torch.int32)
```

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 [ ]:

```
# 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]], dtype=torch.int32)

torch.Size([10, 1])
tensor([[ 5],
        [ 6],
        [ 7],
        [ 8],
        [ 9],
        [10],
        [11],
        [12],
        [13],
        [14]], dtype=torch.int32)
```

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 [ ]:

```
# used example at
# https://github.com/atremblay/mat6115/blob/7b27e1276807a6d0f2ff975d5b891246a8e6f3
# https://pytorch.org/docs/stable/generated/torch.nn.RNN.html#torch.nn.RNN
import torch.nn as nn

class RNN(nn.Module):

    def __init__(self, vocab_size, output_size, embedding_dim, hidden_dim, n_layer
        """
        Initialize the PyTorch RNN Module
        :param vocab_size: The number of input dimensions of the neural network (t
        :param output_size: The number of output dimensions of the neural network
        :param embedding_dim: The size of embeddings, should you choose to use the
        :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

        # print('vocab_size: ',vocab_size)
        # print('output_size: ',output_size)
        # print('embedding_dim: ',embedding_dim)
        # print('hidden_dim: ',hidden_dim)
        # print('n_layers: ',n_layers)
        # print('dropout: ',dropout)

        # set class variables
        # self.vocab_size = vocab_size
        self.output_size = output_size
        # self.embedding_dim = embedding_dim
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers
        # self.dropout = dropout
        # self.batch_size = 50

        # self.hidden_state = self.init_hidden()

        # define model layers

        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        # self.rnn_type = rnn_type.lower()
```

```python
        self.rnn = nn.RNN(
            input_size = embedding_dim,
            hidden_size = hidden_dim,
            num_layers = n_layers,
            dropout = dropout,
            batch_first=True
            )

        # must use lstm since the output of the init_hidden() doesn't agree with r
        self.lstm = nn.LSTM(
            input_size = embedding_dim,
            hidden_size = hidden_dim,
            num_layers = n_layers,
            dropout = dropout,
            batch_first=True
            )

        # predictor in the example from github..but i might need more than 1
        # self.predictor = nn.Linear(hidden_dim, output_size)
        self.fc1 = nn.Linear(hidden_dim,output_size)

        self.dropout = nn.Dropout(p=dropout)

        # possible other layers top try

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

        # self.gru = nn.GRU(
        #     embedding_dim,
        #     hidden_dim,
        #     dropout,
        #     batch_first = True
        # )


    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 hidd
        """
        # TODO: Implement function

        #rename --- if it's the hidden state name it hidden_state
        hidden_state = hidden

        nn_input = torch.tensor(nn_input).to(torch.int64)
        batch_size = nn_input.size(0)

        embedded = self.embedding(nn_input)
        x, hidden_state = self.lstm(embedded,hidden_state)

        # x = self.dropout(x)

        x = self.fc1(x)
```

```python
        x = x.view(batch_size, -1, self.output_size)

        # get last batch
        x = x[:, -1]


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

        # print('n_layers',self.n_layers)
        # print('batch_size',batch_size)
        # print('hidden_dim',self.hidden_dim)

        # get help from the knowledge section of Udacity
        # seems like a lot of people had issues with this section

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

        # if train_on_gpu:
        #     hidden = (weight.new(self.n_layers, batch_size, self.hidden_dim).zer
        #             weight.new(self.n_layers, batch_size, self.hidden_dim).zero_
        # else:
        #     hidden = (weight.new(self.n_layers, batch_size, self.hidden_dim).zer
        #             weight.new(self.n_layers, batch_size, self.hidden_dim).zero_
        # return hidden

        if train_on_gpu:
            hidden = (torch.zeros(self.n_layers, batch_size, self.hidden_dim).cuda
                    torch.zeros(self.n_layers, batch_size, self.hidden_dim).cuda()
        else:
            hidden = (torch.zeros(self.n_layers, batch_size, self.hidden_dim),
                    torch.zeros(self.n_layers, batch_size, self.hidden_dim))
        return hidden

        return None

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

```
Tests Passed
```

```
C:\Users\JGarza\AppData\Roaming\Python\Python37\site-packages\ipykernel_launcher.p
y:95: UserWarning: To copy construct from a tensor, it is recommended to use sourc
eTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), ra
ther than torch.tensor(sourceTensor).
```

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 [ ]:

```
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()


    inp = torch.tensor(inp).to(torch.float)
    # target = torch.tensor(target).to(torch.int64)

    # perform backpropagation and optimization
    hidden = tuple([each.data for each in hidden])

    rnn.zero_grad()

    output, hidden = rnn(inp,hidden)
    # output = torch.tensor(output).to(torch.int64)

    target = torch.tensor(target).to(torch.int64)
    # output = torch.tensor(output).to(torch.int64)

    # print('output:')
    # print('\tsize',output.size)
    # print('\tshape',output.shape)

    # print('target:')
    # print('\tvalue',target)
    # print('\tshape',target.shape)
    # target = target[None,:]
    target = target.flatten()
    # print('\tshape',target.shape)
```

```
    # output = torch.tensor(output).to(torch.int64)

    # target = target.type(torch.LongTensor)
    # target = torch.tensor(target).to(torch.int64)

    # print(target.shape)
    # print(target)
    # target = target[None,:]
    # print(target.shape)

    # target = target.long()
    # target = target.float()

    # output = output[None,:]

    loss = criterion(output,target)
    # loss = criterion(output.squeeze(),target)
    loss.backward()

    nn.utils.clip_grad_norm_(rnn.parameters(),5)
    optimizer.step()

    # return the loss over a batch and the hidden state produced by our model
    # return None,None
    return loss.item(),hidden

# 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
C:\Users\JGarza\AppData\Roaming\Python\Python37\site-packages\ipykernel_launcher.p
y:19: UserWarning: To copy construct from a tensor, it is recommended to use sourc
eTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), ra
ther than torch.tensor(sourceTensor).
C:\Users\JGarza\AppData\Roaming\Python\Python37\site-packages\ipykernel_launcher.p
y:30: UserWarning: To copy construct from a tensor, it is recommended to use sourc
eTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), ra
ther than torch.tensor(sourceTensor).
```

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

def train_rnn(rnn, batch_size, optimizer, criterion, n_epochs, show_every_n_batche
    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, la
            # 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 [ ]:

```
# Data params
# Sequence Length
sequence_length = 10  # of words in a sequence
# Batch Size
batch_size = 256

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

In [ ]:

```

# Training parameters
# Number of Epochs
num_epochs = 16
# Learning Rate
learning_rate = 0.0001

# Model parameters
# Vocab size
print('len(vocab_to_int)',len(vocab_to_int))
vocab_size = len(vocab_to_int)
# Output size
output_size = vocab_size
# Embedding Dimension
embedding_dim = 512
# Hidden Dimension
hidden_dim = 512
# Number of RNN Layers
n_layers = 3

# Show stats for every n number of batches
show_every_n_batches = 1000
```

```
len(vocab_to_int) 21388
```

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 [ ]:

```
"""
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.
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_ev

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

```
Training for 16 epoch(s)...
C:\Users\JGarza\AppData\Roaming\Python\Python37\site-packages\ipykernel_launcher.p
y:19: UserWarning: To copy construct from a tensor, it is recommended to use sourc
eTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), ra
ther than torch.tensor(sourceTensor).
C:\Users\JGarza\AppData\Roaming\Python\Python37\site-packages\ipykernel_launcher.p
y:95: UserWarning: To copy construct from a tensor, it is recommended to use sourc
eTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), ra
ther than torch.tensor(sourceTensor).
C:\Users\JGarza\AppData\Roaming\Python\Python37\site-packages\ipykernel_launcher.p
y:30: UserWarning: To copy construct from a tensor, it is recommended to use sourc
eTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), ra
ther than torch.tensor(sourceTensor).
Epoch:    1/16    Loss: 5.639312689781189

Epoch:    1/16    Loss: 5.054358322620391

Epoch:    1/16    Loss: 4.780596557617187

Epoch:    2/16    Loss: 4.554707446991594

Epoch:    2/16    Loss: 4.419215393543244

Epoch:    2/16    Loss: 4.336504318475724

Epoch:    3/16    Loss: 4.248210528308169

Epoch:    3/16    Loss: 4.1881794097423555

Epoch:    3/16    Loss: 4.142945285081863

Epoch:    4/16    Loss: 4.0764936504338305

Epoch:    4/16    Loss: 4.041702488899231

Epoch:    4/16    Loss: 4.006689583301545

Epoch:    5/16    Loss: 3.9491799672980195

Epoch:    5/16    Loss: 3.928576568365097

Epoch:    5/16    Loss: 3.904162686109543

Epoch:    6/16    Loss: 3.848041197681684

Epoch:    6/16    Loss: 3.835622817993164

Epoch:    6/16    Loss: 3.816148514509201

Epoch:    7/16    Loss: 3.76292621607407
```

```
Epoch:    7/16    Loss: 3.7559081881046295

Epoch:    7/16    Loss: 3.7393806946277617

Epoch:    8/16    Loss: 3.6906151925778454

Epoch:    8/16    Loss: 3.6845326159000398

Epoch:    8/16    Loss: 3.6703808839321135

Epoch:    9/16    Loss: 3.6262911588676534

Epoch:    9/16    Loss: 3.618408263206482

Epoch:    9/16    Loss: 3.607994146823883

Epoch:   10/16    Loss: 3.565003758010196

Epoch:   10/16    Loss: 3.5588595378398895

Epoch:   10/16    Loss: 3.5499779198169708

Epoch:   11/16    Loss: 3.509847363532393

Epoch:   11/16    Loss: 3.5028757507801056

Epoch:   11/16    Loss: 3.494347870349884

Epoch:   12/16    Loss: 3.457857852515506

Epoch:   12/16    Loss: 3.4508331997394563

Epoch:   12/16    Loss: 3.443680145740509

Epoch:   13/16    Loss: 3.4097580906515814

Epoch:   13/16    Loss: 3.4023965871334076

Epoch:   13/16    Loss: 3.3947636795043947

Epoch:   14/16    Loss: 3.364686367968022

Epoch:   14/16    Loss: 3.355980377674103

Epoch:   14/16    Loss: 3.348944793701172

Epoch:   15/16    Loss: 3.3195263992422674

Epoch:   15/16    Loss: 3.310400018453598

Epoch:   15/16    Loss: 3.3031608300209045

Epoch:   16/16    Loss: 3.27723955626115

Epoch:   16/16    Loss: 3.266915283918381

Epoch:   16/16    Loss: 3.260083507299423

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: (Write answer, here)

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 [ ]:

```
"""
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 [ ]:

```
"""
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

        # https://knowledge.udacity.com/questions/43439
        # current_seq = np.roll(current_seq, -1, 1)
        current_seq = np.roll(current_seq.cpu(), -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 [ ]:

```
# 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_vo
print(generated_script)
```

```
C:\Users\JGarza\AppData\Roaming\Python\Python37\site-packages\ipykernel_launcher.p
y:95: UserWarning: To copy construct from a tensor, it is recommended to use sourc
eTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), ra
ther than torch.tensor(sourceTensor).
jerry: they have.

kramer: no. no.

george: so, uh, i don't know how to do anything.

jerry: i thought it was a long idea, huh?

hoyt: no, it's not my fault.

elaine: what are you going to do?

george: you know, you know, it's just something, i was just trying to get to the b
athroom.

jerry: oh, you know, i was going to tell you, i can't go to see him and we get a g
ood time with you.

jerry: oh, you can get it out of the time.

kramer: hey, what do we want to do?

elaine: you know, the only thing i was going to be a lot of kind of people.

jerry: oh, yeah, i think you can get out of this.

hoyt: you know what i mean, i was thinking of my life that i have to be a lot- fiv
e dollars.
```

```
george: i think i could be the best.

george: well, what do we want?

kramer: oh, yeah, yeah, yeah.

george: so, what are you gonna do?

kramer: i can't.

elaine: i don't have any time, i don't know how that is what i can do about you?

george: no.

jerry: oh, i can't do this, it's just a good thing.

jerry: i don't care.

george: so, i don't want to talk to you to tell you what you can do.

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

elaine: i can't tell you that. i mean, i'm gonna get a cab.

george: oh.

jerry: i know.

jerry: i can't believe it was a good time!

kramer: yeah!

kramer: yeah.

elaine: oh, hi.

jerry: what are you talking about?
```

Save your favorite scripts

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

In [ ]:

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

The TV Script is Not Perfect

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

Example generated script

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

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

Submitting This Project

When submitting this project, make sure to run all the cells before saving the notebook. Save the notebook file as "dlnd_tv_script_generation.ipynb" and save another copy as an HTML file by clicking "File" -> "Download as.."->"html". Include the "helper.py" and "problem_unittests.py" files in your submission. Once you download these files, compress them into one zip file for submission.

In [ ]:

```

```