```
from google.colab import drive
drive.mount('/content/gdrive')
import os
os.chdir("gdrive/My Drive/assignment4")
```

```
    Drive already mounted at /content/gdrive; to attempt to forcibly remount, call
```

Generating Text with an RNN

```
! pip install unidecode
import unidecode
import string
import random
import re
import time
import tqdm
import torch
import torch.nn as nn

%matplotlib inline

%load_ext autoreload
%autoreload 2
```

```
    Requirement already satisfied: unidecode in /usr/local/lib/python3.6/dist-pack
```

```
# from rnn.model import RNN
from rnn.helpers import time_since
from rnn.generate import generate
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

Data Processing

The file we are using is a plain text file. We turn any potential unicode characters into plain ASCII by using the `unidecode` package (which you can install via `pip` or `conda`).

```
all_characters = string.printable
n_characters = len(all_characters)

file_path = './shakespeare.txt'
file = unidecode.unidecode(open(file_path).read())
file_len = len(file)
print('file_len =', file_len)

# we will leave the last 1/10th of text as test
split = int(0.9*file_len)
```

```
train_text = file[:split]
test_text = file[split:]

print('train len: ', len(train_text))
print('test len: ', len(test_text))
```

```
file_len = 4573338
train len:  4116004
test len:  457334
```

```
chunk_len = 200

def random_chunk(text):
    start_index = random.randint(0, len(text) - chunk_len)
    end_index = start_index + chunk_len + 1
    return text[start_index:end_index]

print(random_chunk(train_text))
```

```
because more dangerous.

Third Watchman:
Ay, but give me worship and quietness;
I like it better than a dangerous honour.
If Warwick knew in what estate he stands,
'Tis to be doubted he would waken him
```

Input and Target data

To make training samples out of the large string of text data, we will be splitting the text into chunks.

Each chunk will be turned into a tensor, specifically a `LongTensor` (used for integer values), by looping through the characters of the string and looking up the index of each character in `all_characters`.

```
# Turn string into list of longs
def char_tensor(string):
    tensor = torch.zeros(len(string), requires_grad=True).long()
    for c in range(len(string)):
        tensor[c] = all_characters.index(string[c])
    return tensor
```

The following function loads a batch of input and target tensors for training. Each sample comes from a random chunk of text. A sample input will consist of all characters *except the last*, while the target wil contain all characters *following the first*. For example: if random_chunk='abc', then input='ab' and target='bc'

```
def load_random_batch(text, chunk_len, batch_size):
    input_data = torch.zeros(batch_size, chunk_len).long().to(device)
    target = torch.zeros(batch_size, chunk_len).long().to(device)
    for i in range(batch_size):
        start_index = random.randint(0, len(text) - chunk_len - 1)
        end_index = start_index + chunk_len + 1
        chunk = text[start_index:end_index]
        input_data[i] = char_tensor(chunk[:-1])
        target[i] = char_tensor(chunk[1:])
    return input_data, target
```

Implement model

Your RNN model will take as input the character for step t_{-1} and output a prediction for the next character t. The model should consiste of three layers - a linear layer that encodes the input character into an embedded state, an RNN layer (which may itself have multiple layers) that operates on that embedded state and a hidden state, and a decoder layer that outputs the predicted character scores distribution.

You must implement your model in the `rnn/model.py` file. You should use a `nn.Embedding` object for the encoding layer, a RNN model like `nn.RNN` or `nn.LSTM`, and a `nn.Linear` layer for the final a predicted character score decoding layer.

TODO: Implement the model in RNN `rnn/model.py`

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, model_type="rnn", n_la
        super(RNN, self).__init__()
        """
        Initialize the RNN model.

        You should create:
        - An Embedding object which will learn a mapping from tensors
        of dimension input_size to embedding of dimension hidden_size.
        - Your RNN network which takes the embedding as input (use models
        in torch.nn). This network should have input size hidden_size and
        output size hidden_size.
        - A linear layer of dimension hidden_size x output_size which
        will predict output scores.

        Inputs:
        - input_size: Dimension of individual element in input sequence to model
        - hidden_size: Hidden layer dimension of RNN model
        - output_size: Dimension of individual element in output sequence from mode
        - model_type: RNN network type can be "rnn" (for basic rnn), "gru", or "lst
        - n_layers: number of layers in your RNN network
        """

        self.model_type = model_type
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers

        #####################################
        #          YOUR CODE HERE           #
        #####################################

        self.input2hidden  = nn.Embedding(input_size, hidden_size)
        self.hidden2output = nn.Linear(hidden_size, output_size)

        if model_type == "rnn":
            self.rnn = nn.RNN(hidden_size, hidden_size, n_layers)
        elif model_type == "lstm":
            self.rnn = nn.LSTM(hidden_size, hidden_size, n_layers)
        elif model_type == "gru":
            self.rnn = nn.GRU(hidden_size, hidden_size, n_layers)

        ##########       END      ##########



    def forward(self, input, hidden):
        """
        Forward pass through RNN model. Use your Embedding object to create
        an embedded input to your RNN network. You should then use the
        linear layer to get an output of self.output_size.

        Inputs:
        - input: the input data tensor to your model of dimension (batch_size)
        - hidden: the hidden state tensor of dimension (n_layers x batch_size x hid

        Returns:
        - output: the output of your linear layer
        - hidden: the output of the RNN network before your linear layer (hidden st
        """

        output = None

        #####################################
        #          YOUR CODE HERE           #
        #####################################
        batch_size = input.shape[0]
        embedding = self.input2hidden(input).view(1, batch_size, -1)
        # required input size for nn.RNN: seq_len, batch, input_size
        output, hidden = self.rnn(embedding, hidden)
        output = self.hidden2output(output.view(1, batch_size, -1))

        ##########       END      ##########
        return output, hidden

    def init_hidden(self, batch_size, device=None):
        """
        Initialize hidden states to all 0s during training.

        Hidden states should be initilized to dimension (n_layers x batch_size x hi

        Inputs:
```

```
        Inputs:
        - batch_size: batch size

        Returns:
        - hidden: initialized hidden values for input to forward function
        """

        hidden = None

        ####################################
        #          YOUR CODE HERE          #
        ####################################

        if self.model_type == "lstm":
            hidden = (torch.zeros(self.n_layers, batch_size, self.hidden_size).to(d
                      torch.zeros(self.n_layers, batch_size, self.hidden_size).to(dev
        else:
            hidden = torch.zeros(self.n_layers, batch_size, self.hidden_size).to(de

        ##########       END      ##########

        return hidden
```

Evaluating

To evaluate the network we will feed one character at a time, use the outputs of the network as a probability distribution for the next character, and repeat. To start generation we pass a priming string to start building up the hidden state, from which we then generate one character at a time.

Note that in the `evaluate` function, every time a prediction is made the outputs are divided by the "temperature" argument. Higher temperature values make actions more equally likely giving more "random" outputs. Lower temperature values (less than 1) high likelihood options contribute more. A temperature near 0 outputs only the most likely outputs.

You may check different temperature values yourself, but we have provided a default which should work well.

```
def evaluate(rnn, prime_str='A', predict_len=100, temperature=0.8):
    hidden = rnn.init_hidden(1, device=device)
    prime_input = char_tensor(prime_str)
    predicted = prime_str

    # Use priming string to "build up" hidden state
    for p in range(len(prime_str) - 1):
        _, hidden = rnn(prime_input[p].unsqueeze(0).to(device), hidden)
    inp = prime_input[-1]

    for p in range(predict_len):
        output, hidden = rnn(inp.unsqueeze(0).to(device), hidden)

        # Sample from the network as a multinomial distribution
        output_dist = output.data.view(-1).div(temperature).exp()
        output_dist = output.data.view(-1).div(temperature).exp()
        top_i = torch.multinomial(output_dist, 1)[0]

        # Add predicted character to string and use as next input
        predicted_char = all_characters[top_i]
        predicted += predicted_char
        inp = char_tensor(predicted_char)

    return predicted
```

Train RNN

```
batch_size = 100
n_epochs = 2500
hidden_size = 150
n_layers = 2
learning_rate = 0.01
model_type = 'lstm'
print_every = 50
plot_every = 50
```

```
def eval_test(rnn, inp, target):
    with torch.no_grad():
        hidden = rnn.init_hidden(batch_size, device=device)
        loss = 0
        for c in range(chunk_len):
            output, hidden = rnn(inp[:,c], hidden)
            loss += criterion(output.view(batch_size, -1), target[:,c])

    return loss.data.item() / chunk_len
```

Train function

TODO: Fill in the train function. You should initialize a hidden layer representation using your RNN's `init_hidden` function, set the model gradients to zero, and loop over each time step (character) in the input tensor. For each time step compute the output of the of the RNN and compute the loss over the output and the corresponding ground truth time step in `target`. The loss should be averaged over all time steps. Lastly, call backward on the averaged loss and take an optimizer step.

```
def train(rnn, input, target, optimizer, criterion):
    """
    Inputs:
    - rnn: model
    - input: input character data tensor of shape (batch_size, chunk_len)
    - target: target character data tensor of shape (batch_size, chunk_len)
    - optimizer: rnn model optimizer
    - criterion: loss function
```

```
      criterion: loss function

    Returns:
    - loss: computed loss value as python float
    """
    loss = 0

    ####################################
    #          YOUR CODE HERE          #
    ####################################
    batch_size = input.shape[0]
    optimizer.zero_grad()
    hidden = rnn.init_hidden(batch_size, device = device)

    for i in range(chunk_len):
        pred, hidden = rnn(input[:, i], hidden)
        loss += criterion(pred.view(batch_size, -1), target[:, i])
    loss /= chunk_len
    loss.backward()
    optimizer.step()

    ##########       END      ##########

    return loss



rnn = RNN(n_characters, hidden_size, n_characters, model_type=model_type, n_layers=
rnn_optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)
criterion = nn.CrossEntropyLoss()

start = time.time()
all_losses = []
test_losses = []
loss_avg = 0
test_loss_avg = 0



print("Training for %d epochs..." % n_epochs)
for epoch in range(1, n_epochs + 1):
    loss = train(rnn, *load_random_batch(train_text, chunk_len, batch_size), rnn_op
    loss_avg += loss

    test_loss = eval_test(rnn, *load_random_batch(test_text, chunk_len, batch_size)
    test_loss_avg += test_loss

    if epoch % print_every == 0:
        print('[%s (%d %d%%) train loss: %.4f, test_loss: %.4f]' % (time_since(star
        print(generate(rnn, 'Wh', 100, device=device), '\n')

    if epoch % plot_every == 0:
        all_losses.append(loss_avg / plot_every)
        test_losses.append(test_loss_avg / plot_every)
        loss_avg = 0
        test_loss_avg = 0

     Which will glong of the what one
    which will glong of the what one.

    ALONSO:
    Sir, I will thou shalt shall not so double humble,
    For love

    [31m 38s (2150 86%) train loss: 1.2993, test_loss: 1.4641]
    Which I peradvers in that are them, we are a silk;
    Will weep with the ringleness of a thing-day: any w

    [32m 22s (2200 88%) train loss: 1.3125, test_loss: 1.4607]
    Who is more than the more griefs.

    SPEED:
    Because the moon Edward that defend,
    His song, the eyes.

    DO

    [33m 6s (2250 90%) train loss: 1.3296, test_loss: 1.4448]
    What very near our lord, all understand saucy
    In the bush is gone.

    CLIFFORD:
    It shall the faults are

    [33m 51s (2300 92%) train loss: 1.3312, test_loss: 1.4417]
    What, is this pay him not?

    HOTSPUR:
    God break the govern'd.

    HELENA:
    I know thee enough my soul!

    FAL

    [34m 35s (2350 94%) train loss: 1.3171, test_loss: 1.4653]
    Why, look, we would have respect to the cold accompeans,
    Have some men, and yet more both: the gallows

    [35m 19s (2400 96%) train loss: 1.2946, test_loss: 1.5014]
    While, and his weapon at her suppers, and sweet one
    oftenfing it.

    BORACHIO:
    I have to keep a fesh Rat

    [36m 3s (2450 98%) train loss: 1.3184, test_loss: 1.4783]
    Why, 'twas left up.


    PERICLES:
    He's the good of valiant.

    MARCUS ANDRONICUS:
    Who can would it he be su

    [36m 47s (2500 100%) train loss: 1.2990, test_loss: 1.4454]
    Who by this fancise and fear of the follow's
```

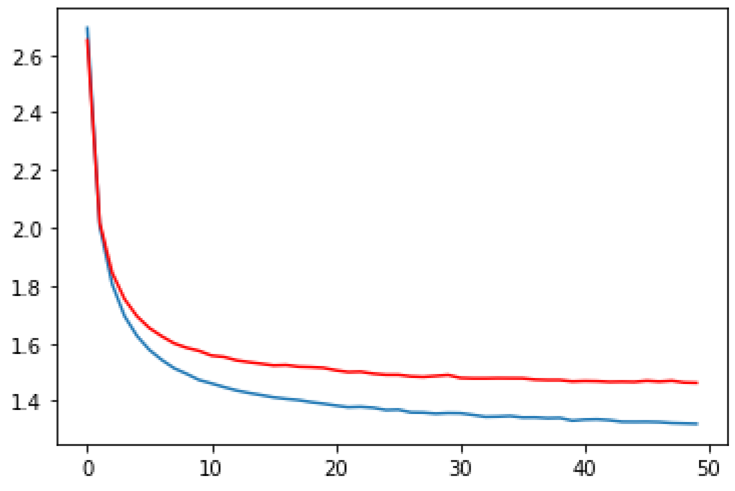
```
# save network
# torch.save(classifier.state_dict(), './rnn_generator.pth')
```

Plot the Training and Test Losses

```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

plt.figure()
plt.plot(all_losses)
plt.plot(test_losses, color='r')
```

```
[<matplotlib.lines.Line2D at 0x7f03410bdba8>]
```



Evaluate text generation

Check what the outputted text looks like

```
print(evaluate(rnn, prime_str='Th', predict_len=1000))
```

```
They are as a bumble shrainton's to thee at my heart
Which let it enter and given a soldier, her man of your thousand chanis
A crown and honour and brack. My honour mock'd the
holy-a
sounded brings of men's searched broken would rememberness
From Actor's throats of the virtues before
And more than you were talk that seem to me,
Who took's men doth new lodges false and meant
That I am possible friented of a secreal,
That hath between your vow off, to hear it:
My lord, he be long and princesses,
The deed be your wish.

FRIAR LAURENCE:
What's bounted in the well-time and we'll eat, comparise,
'Tis any you without bear and my bonnder say.

BEATRICE:
I will have perceive thee of the orator encounter.
```

```
I warrant her soul we not such present to-morrow
In a case of my mistress of a world it fill one.

HASTINGS:
He cannot neither so step for it by
As they say thou be very ten love, that am you
Concerns of consumbers and place and modest
tunkest: on the soldier parts to Silvia,
What it is the king,
```

+ 代码 + 文本

Hyperparameter Tuning

Some things you should try to improve your network performance are:

- Different RNN types. Switch the basic RNN network in your model to a GRU and LSTM to compare all three.
- Try adding 1 or two more layers
- Increase the hidden layer size
- Changing the learning rate

TODO: Try changing the RNN type and hyperparameters. Record your results.