Name: Advait Pai
UIN: 677368201
Email: apai21@uic.edu

Homework 7

1. LSTM Model:

```python
# LSTM Model
class LSTMText(nn.Module):
    def __init__(self):
        super(LSTMText,self).__init__()
        self.in_size = 27
        self.hidden = 40
        self.layers = 2
        self.out_size = 27

        self.lstm = nn.LSTM(input_size = self.in_size, hidden_size=self.hidden, num_layers=self.layers,dropout=0.1)
        self.linear = nn.Linear(in_features = self.hidden, out_features=self.out_size)

    def forward(self,x):
        x, (h,c) = self.lstm(x)
        x = self.linear(x)
        return x,(h,c)
```

My LSTM model is made of up one LSTM part and one Linear Layer. The LSTM part has the following parameters:

- input_size = 27
- output_size = 27
- layers = 2
- hidden_size = 40 (I have experimented with different numbers like 10,20,30,40,128 and I did not observe a change in the loss obtained in the end, hence I took 40, as it gave good performance in terms of epochs per second as well as I did not want to have simplistic assumptions in my model).
- dropout = 0.1

The linear layer has:

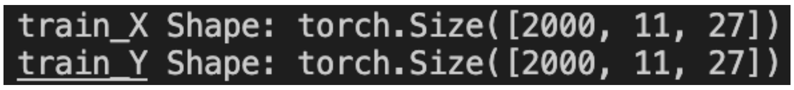
- in_features = 40 (same as hidden size)
- out_features = 27

And I only apply the pass the input through both these layers, and return x. I do not use the (h,c) from the LSTM to feedback into the LSTM.

Training Hyperparameters:

- Batch_size = 100 (I experimented with different batch sizes, like 20,100,2000 and then got the least loss for batch_size = 100, hence I chose this).
- Epochs = 500
- optimizer = Adam
- learning_rate = 0.05
- scheduler = StepLR
- gamma = 0.9

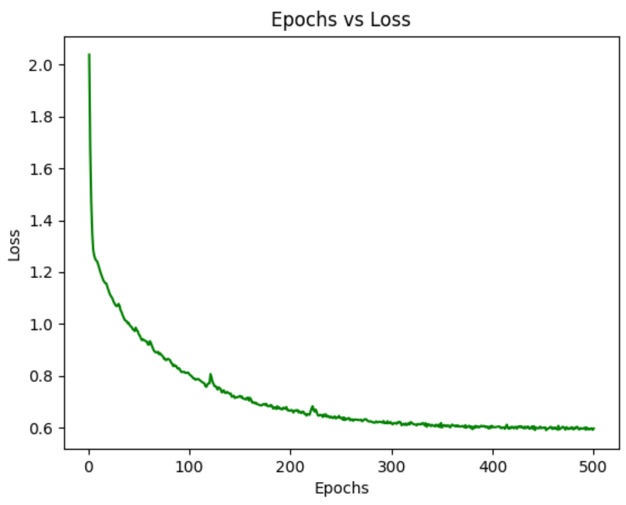
Input Shapes:



```
train_X Shape: torch.Size([2000, 11, 27])
train_Y Shape: torch.Size([2000, 11, 27])
```

2. **<u>Loss value vs Epoch Graph:</u>**

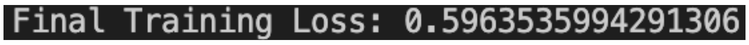
Graph:



While I tried to apply StepLR to reduce the slight spikes in my line, I was able to get this as the best possible line.

Final Training Loss: 0.5964

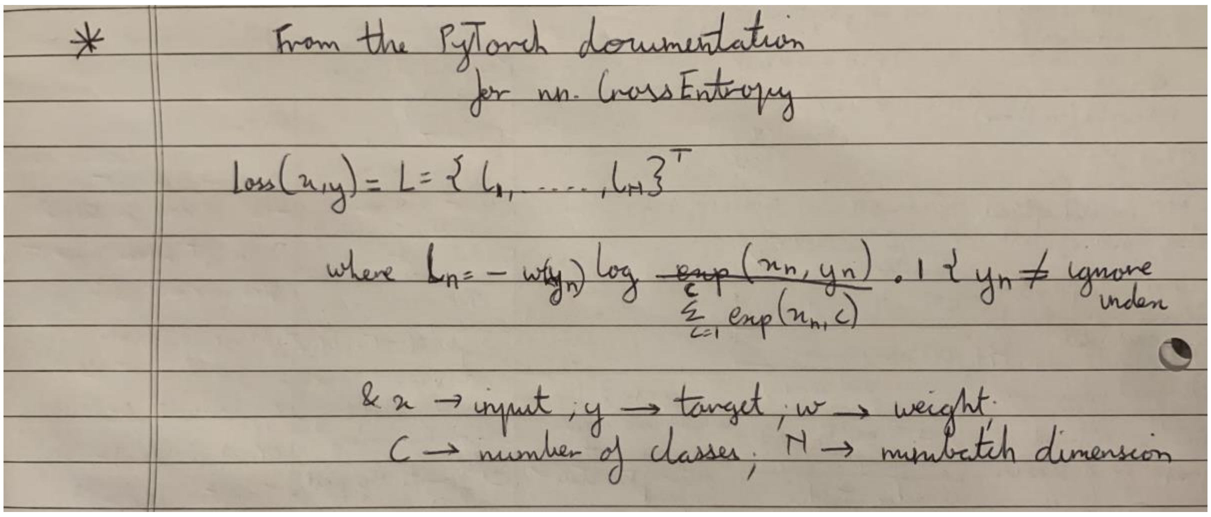
From terminal:



```
Final Training Loss: 0.5963535994291306
```

Loss Function Used: nn.CrossEntropyLoss() i.e Cross Entropy Loss

Formula:



* From the PyTorch documentation for nn. CrossEntropy

$$\text{Loss}(x, y) = L = \{l_1, \ldots, l_N\}^T$$

where $l_n = -w_{y_n} \log \frac{\exp(x_n, y_n)}{\sum_{c=1}^{C} \exp(x_{n}, c)} \cdot 1\{y_n \neq \text{ignore index}\}$

& $x \rightarrow$ input ; $y \rightarrow$ target ; $w \rightarrow$ weight.
$C \rightarrow$ number of classes ; $N \rightarrow$ minibatch dimension

Explanation:

I tried changing the various parameters of the model during testing of my model design, but the best loss I achieved for batch_size = 100 was 0.5963 after running for 500 epochs. I tried changing the batch_size, the number of hidden layers and the dropout as well, but I was not able to achieve a smaller loss.

I attribute this to the assumptions I have made during training the model. I have used a simple LSTM with a Linear Layer to achieve a name generator, and since the input size is of 2000, there are exponential combinations which my simplistic model may not be able to learn. Also the model only trains on a small batch_size of 100, thus learning only unique patters for 100 names at once, whereas the unique patters exist along 2000 names.

As mentioned, I tried for a batch_size of 2000 but the loss was 1.06 approximately, and since the compute resources are limited, I could not expand my LSTM much without reducing performance. Even if I tried to increase my hidden layers of the LSTM to 128, I wasn’t getting accurate performance, I tried increasing dropout to 0.25 as well, but it did not make a difference. I have not tried this, but I believe bidirectional LSTM’s should also tried because knowing the previous alphabet may help as well. Given more time for experimenting and training, I am certain I will be able to reduce the loss.

3. **<u>Name Generation:</u>**

Output:

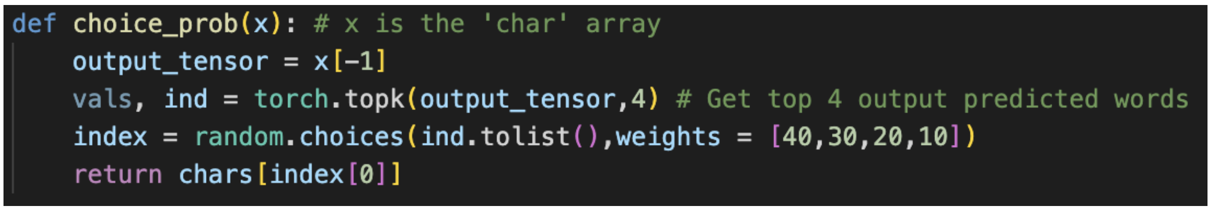
```
20 names generated for a:
['aylethncerougud', 'asiereya', 'ayremslerdiaele', 'aniolilorlillie', 'ayleryaylittagc',
'aniyltthelonaha', 'aykobylltoranan', 'aykebariselinan', 'anidsengedrondi', 'anykmeidonelay',
'ayllourophelylv', 'aykewnisniacila', 'anyn', 'anykaiaralirist', 'ayrreerlaremayl', 'anyayn',
'anieelowneweyai', 'aseeninaciommoo', 'asokeykelivavys', 'ayretenisnddore']

20 names generated for x:
['xylilarolonnciy', 'xredrulalariaer', 'xranditelilille', 'xruylergayacary', 'xrudyleelardyad',
'xeinaynn', 'xruliylagox', 'xaiidendynsauly', 'xaseyaraditarab', 'xyveiaisenzerin',
'xanolivekyntonn', 'xrulalinneveric', 'xaiox', 'xasereneleleysy', 'xyniessch', 'xaseyltpringeri',
'xaianeriaroaril', 'xranitonariande', 'xynserine', 'xeinndelinenath']
```

Parameters:

Max length of each name = 15
Min length of each name = 3
Random Probability Assigned =
1st Best Match = 0.4
2nd Best Match = 0.3
3rd Best Match = 0.2
4th Best Match = 0.1

Code for character choice:



```python
def choice_prob(x): # x is the 'char' array
    output_tensor = x[-1]
    vals, ind = torch.topk(output_tensor,4) # Get top 4 output predicted words
    index = random.choices(ind.tolist(),weights = [40,30,20,10])
    return chars[index[0]]
```

I have a written a custom function, which find the top 4 predicted characters given an input sequence. This uses the torch.topk like this {vals, ind = torch.topk(output_tensor,4)}. Once I have the top 4 predicted words, I use the indices returned from torch.topk function. Once I have the top 4 predicted words, I use the Python in-built function from class random called random.choices. It allows me to assign a weight to every index obtained from torch.topk. using the random.choice, I get a random choice of index of the top 4 words, from which I extract and return the corresponding character.

Code for generating the names:

```python
def generate_names(char, l, model):
   model.eval()
   input = torch.tensor(convert_char_array(char),dtype=torch.float32)
   input = input.to(device)
   output,(h,c) = model(input)
   output_char = choice_prob(output)
   if(output_char == 'EON' or len(char)==l):
      if(len(char)<3):
         return generate_names(char,l,model)
      else:
         return char
   else:
      char = char+output_char
      return generate_names(char,l,model)
```

The code calls generate_names recursively. First supposing 'a' is passed and 'b' is the character returned from the function {choice_prob}, I call generate_names for 'ab', and this continue till either 'EON' is received or max_length is achieved, which is 15 in my case. If the length of the name is less than 3, I ignore an 'EON' and rerun the generate_names function. I also check for duplicate names in the final list.

FINAL CODE – 0701-677368201-Pai.py

```python
# Import file
import os
from string import ascii_lowercase

# Imports for LSTM
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torch.utils.data as data_utils
from torch.optim.lr_scheduler import StepLR

# General imports
from tqdm import tqdm

# LSTM Model
class LSTMText(nn.Module):
    def __init__(self):
        super(LSTMText,self).__init__()
        self.in_size = 27
        self.hidden = 40
        self.layers = 2
        self.out_size = 27

        self.lstm = nn.LSTM(input_size = self.in_size, hidden_size=self.hidden, num_layers=self.layers,dropout=0.1)
        self.linear = nn.Linear(in_features = self.hidden, out_features=self.out_size)

    def forward(self,x):
        x, (h,c) = self.lstm(x)
        x = self.linear(x)
        return x,(h,c)


# Torch init settings
device_type = "cpu"
device = torch.device(device_type)
torch.manual_seed(2702) # Fixed Seed Value
```

```python
# Extracting names from the file
names = []
with open('names.txt',mode='r') as f:
    names = f.readlines()
    names = [x.lower().replace('\n',"") for x in names]
print("Length of Name:",len(names))
chars = ['EON']
chars.extend([a for a in ascii_lowercase])

# One hot encoding for each character
char_encode = dict()
for i in range(len(chars)):
    temp = np.zeros(27)
    temp[i] = 1.0
    char_encode[chars[i]] = temp

# Function to create a name array of shape (11,27)
def convert_char_array(name):
    convert_arr = []
    for i in range(11):
        if i<len(name):
            convert_arr.append(char_encode[name[i]])
        else:
            convert_arr.append(char_encode['EON'])
    return np.array(convert_arr)

# Creating (xi,yi) pairs
train_x = [] # Temporary x list with numpy arrays
train_y = [] # Temporary y list with numpy arrays
for name in names:
    train_x.append(convert_char_array(name))
    train_y.append(convert_char_array(name[1:]))

train_X = torch.tensor(np.array(train_x),dtype=torch.float32)
train_Y = torch.tensor(np.array(train_y),dtype=torch.float32)

print("train_X Shape:",train_X.shape)
print("train_Y Shape:",train_Y.shape)
```

```
dataset = data_utils.TensorDataset(train_X, train_Y) # Creating (Xi,Yi) pairs
train_loader = data_utils.DataLoader(dataset,batch_size=100)

def train(model,device,train_loader,optimizer,loss_function):
    model.train()
    total_loss = 0

    for input, label in train_loader:
        input,label = input.to(device),label.to(device)
        optimizer.zero_grad()
        output,(h,c) = model(input)
        loss = loss_function(output.permute(0,2,1),torch.argmax(label,dim=2))
        loss.backward()
        optimizer.step()
        total_loss+=loss.item()
    # print("Training Loss:",total_loss/(100))
    return total_loss/(20)

# Model Initialization
model = LSTMText().to(device)
learning_rate = 0.05
gamma = 0.9
epochs = 500
optimizer = optim.Adam(model.parameters(),lr=learning_rate)
scheduler = StepLR(optimizer,step_size=10,gamma=gamma)

loss_function = nn.CrossEntropyLoss() # Used with Logits output (no activation)
training_loss = []
print("Training for Epochs:",)
for i in tqdm(range(1,epochs+1)):
    loss = train(model,device,train_loader,optimizer,loss_function)
    training_loss.append(loss)
    scheduler.step()
print("Final Training Loss:",training_loss[-1])

model_path = "0702-677368201-Pai.pt"
torch.save(model.state_dict(), model_path)

import matplotlib.pyplot as plt
```

```
plt.title("Epochs vs Loss")

plt.xlabel("Epochs")

plt.ylabel("Loss")

plt.plot([i for i in range(1,epochs+1)], training_loss, color ="green")

plt.show()
```

FINAL CODE: 0703-677368201-Pai.py

```python
# General Imports
from string import ascii_lowercase
import random

# Imports for LSTM
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torch.utils.data as data_utils
from torch.optim.lr_scheduler import StepLR

# Fixed Seed Value
torch.manual_seed(2702)

# LSTM Model
class LSTMText(nn.Module):
    def __init__(self):
        super(LSTMText,self).__init__()
        self.in_size = 27
        self.hidden = 40
        self.layers = 2
        self.out_size = 27

        self.lstm = nn.LSTM(input_size = self.in_size, hidden_size=self.hidden, num_layers=self.layers,dropout=0.1)
        self.linear = nn.Linear(in_features = self.hidden, out_features=self.out_size)

    def forward(self,x):
        x, (h,c) = self.lstm(x)
        x = self.linear(x)
        return x,(h,c)

# Character List
chars = ['EON']
chars.extend([a for a in ascii_lowercase])

# One hot encoding for each character
```

```
char_encode = dict()
for i in range(len(chars)):
    temp = np.zeros(27)
    temp[i] = 1.0
    char_encode[chars[i]] = temp


# Function to create a name array of shape (11,27)
def convert_char_array(name):
    convert_arr = []
    for i in range(len(name)):
        if i<len(name):
            convert_arr.append(char_encode[name[i]])
        else:
            convert_arr.append(char_encode['EON'])
    return np.array(convert_arr)

def choice_prob(x): # x is the 'char' array
    output_tensor = x[-1]
    vals, ind = torch.topk(output_tensor,4) # Get top 4 output predicted words
    index = random.choices(ind.tolist(),weights = [40,30,20,10])
    return chars[index[0]]


def generate_names(char, l, model):
    model.eval()
    input = torch.tensor(convert_char_array(char),dtype=torch.float32)
    input = input.to(device)
    output,(h,c) = model(input)
    output_char = choice_prob(output)
    if(output_char == 'EON' or len(char)==l):
        if(len(char)<3):
            return generate_names(char,l,model)
        else:
            return char
    else:
        char = char+output_char
        return generate_names(char,l,model)

# Reload the model
```

```python
device = torch.device("cpu")
model = LSTMText().to(device)
model_path = "0702-677368201-Pai.pt"
checkpoint = torch.load(model_path,map_location=device)
model.load_state_dict(checkpoint)


gen_names = {'a':[],'x':[]}
while len(gen_names['a'])<20:
    name = generate_names('a',15,model)
    if(name not in gen_names['a']):
        gen_names['a'].append(name)
print(str(len(gen_names['a'])),"names generated for a: ")
print(gen_names['a'])

while len(gen_names['x'])<20:
    name = generate_names('x',15,model)
    if(name not in gen_names['x']):
        gen_names['x'].append(name)
print(str(len(gen_names['x'])),"names generated for x: ")
print(gen_names['x'])
```