2.1. (25 points) Modify the implementation of the network to leverage the RNN subclass of module torch.nn, which readily incorporates support for batch training. Note that the “nn.RNN” class is modified to operate in a batch mode.

Submitted by:

Aradhya Mathur

Lakshmi Nikhil Goduguluri

Set the hidden state size to 128 and train the network through five epochs with a batch size equal to the total number of samples. Note that, since the data samples are of different lengths, you will need to pad the length of the samples to a unique sequence length (e.g., at least the length of the longest sequence) in order to be able to feed the batch to the network. This is because RNN expects the input to be a tensor of shape (batch, seq_len, input_size). It is best to manually pad with 0s, or you can use built-in functions such as torch.nn.utils.rnn.pad_sequence to perform the padding.

Report the accuracy yielded by this approach on the full training set after training for 5 epochs.

The code below indicates the modified section of nn.RNN class. This section is included in the example script.

class RNN(nn.Module): def init (self): super(RNN, self). init ()

self.rnn = nn.RNN(input_size = INPUT_SIZE, hidden_size = HIDDEN_SIZE, # number of hidden units num_layers = N-LAYERS, # number of layers batch_first = True, # If your input data is of shape (seq_len, batch_size, features) then you don’t need batch_first=True and your RNN will output a tensor with shape (seq_len, batch.

If your input data is of shape (batch_size, seq_len, features) then you need batch_first=True and your RNN will output a tensor with shape (batch_size, seq_len, hidden_size).

) self.out = nn.Linear(HIDDEN_SIZE, OUTPUT_SIZE)

def forward(self, x):

r_out, (h_n, h_c) = self.rnn(x, None) # None represents zero initial hidden state

r_out, h = self.rnn(x, None) # None represents zero initial hidden state

choose last time step of output out = self.out(r_out[:, -1, :]) return out

Recall that input_size refers to the size of the features (in this case one-hot encoded representation of each letter). Hidden size is a hyper parameter you can adjust (we will keep it fixed at 128 in this question). The comments in the code above explain how to format your batch data, depending on the value of batch_first. Lastly, OUTPUT_SIZE denotes the number of classes.

In [1]:
```
from __future__ import unicode_literals, print_function, division
from io import open
import glob
import os
import numpy as np
import pandas as pd
import unicodedata
import string
import torch
import torch.nn as nn
import random
import matplotlib.pyplot as plt
import matplotlib.ticker as ticke


def findFiles(path):
    return glob.glob(path)

all_letters = string.ascii_letters + " .,;'"
n_letters = len(all_letters)

def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
        and c in all_letters
    )



names = {}
languages = []


def readLines(filename):
    lines = open(filename, encoding='utf-8').read().strip().split('\n')
    return [unicodeToAscii(line) for line in lines]

# (TO DO:) CHANGE FILE PATH AS NECESSARY
for filename in findFiles(r"C:\Users\aradh\Desktop\Fall 22\TSA\Project 3.1\data\da
    category = os.path.splitext(os.path.basename(filename))[0]
    languages.append(category)
    lines = readLines(filename)
    names[category] = lines

n_categories = len(languages)
```

```
def letterToIndex(letter):
    return all_letters.find(letter)


def nameToTensor(name):
    tensor = torch.zeros(len(name), 1, n_letters)
    for li, letter in enumerate(name):
        tensor[li][0][letterToIndex(letter)] = 1
    return tensor

class RNN(nn.Module):
    def __init__(self, INPUT_SIZE, HIDDEN_SIZE, N_LAYERS,OUTPUT_SIZE):
        super(RNN, self).__init__()
        self.rnn = nn.RNN(
            input_size = INPUT_SIZE,
            hidden_size = HIDDEN_SIZE, # number of hidden units
            num_layers = N_LAYERS, # number of layers
            batch_first = True)
        self.out = nn.Linear(HIDDEN_SIZE, OUTPUT_SIZE)

    def forward(self, x):
        r_out, h = self.rnn(x, None) # None represents zero initial hidden state
        out = self.out(r_out[:, -1, :])
        return out

n_hidden = 128

allnames = [] # Create list of all names and corresponding output language
for language in list(names.keys()):
    for name in names[language]:
        allnames.append([name, language])

## (TO DO:) Determine Padding length (this is the length of the longest string)
# maxlen = ..... # Add code here to compute the maximum length of string

maxlen = max(len(x[0]) for x in allnames)
padded_length = maxlen
print(padded_length)

n_letters = len(all_letters)
n_categories = len(languages)

def categoryFromOutput(output):
    top_n, top_i = output.topk(1)
    category_i = top_i.item()
    return languages[category_i], category_i
```

```
19
```

In [2]:

```
learning_rate = 0.005
rnn = RNN(n_letters, 128, 1, n_categories)
optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)   # optimize all 
loss_func = nn.CrossEntropyLoss()
for epoch in range(5):
    batch_size = len(allnames)
    random.shuffle(allnames)
     # if "b_in" and "b_out" are the variable names for input and output tensors, y
    b_in = torch.zeros(batch_size, padded_length, n_letters)  # (TO DO:) Initializ
    b_out = torch.zeros(batch_size, n_categories, dtype=torch.long)  # (TO DO:) In
    def get(charachter):
        return [x for x in charachter]
```

```python
    # (TO DO:) Populate "b_in" and "b_out" tensor. Can be done in a single loop
    for i in allnames:
        j=allnames.index(i)
        k=get(i[0])
        for l in range(len(i[0])):
            b_in[j][l][letterToIndex(k[l])]=1
        m=i[1]
        l=languages.index(m)
        b_out[j][l]=1
    max_b_out=torch.max(b_out,1)[1]
    output = rnn(b_in)                              # rnn output
    #(TO DO:)
    loss = loss_func(output, max_b_out)   # (TO DO:) Fill "...." to calculate the c
    optimizer.zero_grad()                          # clear gradients for this trai
    loss.backward()                                # backpropagation, compute grad
    optimizer.step()                               # apply gradients
         # Print accuracy
    test_output = rnn(b_in)                 #
    pred_y = torch.max(test_output, 1)[1].data.numpy().squeeze()
    test_y = torch.max(b_out, 1)[1].data.numpy().squeeze()
    accuracy = sum(pred_y == test_y)/batch_size
    print("Epoch: ", epoch, "| train loss: %.4f" % loss.item(), '| accuracy: %.2f'
```

```
Epoch:  0 | train loss: 2.8916 | accuracy: 0.47
Epoch:  1 | train loss: 2.6784 | accuracy: 0.47
Epoch:  2 | train loss: 2.1190 | accuracy: 0.47
Epoch:  3 | train loss: 1.9796 | accuracy: 0.47
Epoch:  4 | train loss: 1.9510 | accuracy: 0.47
```

In [3]:
```python
b_in.shape
```

Out[3]:
```
torch.Size([20074, 19, 57])
```

In [ ]:
```python

```