DSC 475: Time Series Analysis and Forecasting (Fall 2020)

Project 3.2 – Sequence Classification with Recurrent Neural Networks (Contd.)

Jingwen Zhong

Overview

In Part 2 of this project, you will continue to work with recurrent neural networks on real-world data. This phase of the project will highlight the aspect of batch and then mini-batch data processing.

Like Part 3.1, the goal of this project continues to be to implement a vanilla RNN from scratch and train it on a set of data containing over 20 thousand last names and their respective country of origin. The number of possible countries, or classes, is 18.

Batch training of data (25 + 15 points)

The training process in Project 3.1 is still suboptimal since the gradient estimates are computed on a sample by sample basis (i.e., a batch size of 1). In this question, you are exposed to batch processing.

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 ticker
```

In [2]:

```
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]

for filename in findFiles(r"C:/Users/Jingwen/Desktop/475 Time Series Analysis/
作业/names/*.txt"):
    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
```

1.1

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.

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 (25 points)

In [3]:
```
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, # the size of the features
            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) # OUTPUT_SIZE denotes t
he number of classes

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

In [4]:
```
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 strin
g)

# maxlen = ..... # Add code here to compute the maximum length of string

maxlen = 0
for name in allnames:
    name_tensor = nameToTensor(name[0])
    if len(name_tensor) >= maxlen:
        maxlen = len(name_tensor)
padded_length = maxlen

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
```

In [5]:
```
learning_rate = 0.005
rnn = RNN(n_letters, 128, 1, n_categories)
optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)   # optimize
 all rnn parameters
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 tensor
s, you need to create those

    b_in = torch.zeros(batch_size, padded_length, n_letters)  # (TO DO:) Initi
alize "b_in" to a tensor with size of input (batch size, padded_length, n_lett
ers)
    b_out = b_out = torch.zeros(batch_size, n_categories, dtype=torch.long) #
 (TO DO:) Initialize "b_out" to tensor with size (batch_size, n_categories, dt
ype=torch.long)

    # a loop:

    # (TO DO:) Populate "b_in" tensor
    for i, name in enumerate(allnames):
        for li, letter in enumerate(name[0]):
            b_in[i][li][letterToIndex(letter)]=1

    # (TO DO:) Populate "b_out" tensor
    for i, name in enumerate(allnames):
        b_out[i][languages.index(name[1])]=1


    output = rnn(b_in)                                    # rnn output
    #(TO DO:)
    loss = loss_func(output, torch.max(b_out, 1)[1])   # (TO DO:) Fill "...."
 to calculate the cross entropy loss
    optimizer.zero_grad()                              # clear gradients for this
training step
    loss.backward()                                    # backpropagation, compute
gradients
    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:
%.6f' % accuracy)
```

```
Epoch:  0 | train loss: 2.9007 | accuracy: 0.468666
Epoch:  1 | train loss: 2.6757 | accuracy: 0.468666
Epoch:  2 | train loss: 2.1667 | accuracy: 0.468666
Epoch:  3 | train loss: 1.9165 | accuracy: 0.468666
Epoch:  4 | train loss: 1.9486 | accuracy: 0.468666
```

1.2.

Modify the implementation from 1.1 to support arbitrary mini-batch sizes. In this case, instead of padding to a unique sequence length, adaptively pad the length of the mini batch to the length of the longest sample in the mini batch itself. Report the accuracy number (on the full training set) yielded by this approach on mini batch sizes of 1000, 2000, 3000 after five epochs of training. (15 points).

batch size = 1000

In [6]:
```
learning_rate = 0.005
rnn = RNN(n_letters, 128, 1, n_categories)
optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)   # optimize
 all rnn parameters
loss_func = nn.CrossEntropyLoss()

for epoch in range(5):
    batch_size = 1000
    random.shuffle(allnames)

    maxlen = 0
    for name in allnames[:1000]:
        name_tensor = nameToTensor(name[0])
        if len(name_tensor) >= maxlen:
            maxlen = len(name_tensor)


    b_in = torch.zeros(batch_size, maxlen, n_letters)
    b_out = torch.zeros(batch_size, n_categories, dtype=torch.long)


    for i, name in enumerate(allnames):
        for li, letter in enumerate(name[0]):
            if i < 1000:
                b_in[i][li][letterToIndex(letter)]=1


    for i, name in enumerate(allnames):
        if i < 1000:
            b_out[i][languages.index(name[1])]=1


    output = rnn(b_in)

    loss = loss_func(output, torch.max(b_out, 1)[1])

    optimizer.zero_grad()                              # clear gradients for this
training step
    loss.backward()                                    # backpropagation, compute
gradients
    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:
%.6f' % accuracy)
```

```
Epoch:  0 | train loss: 2.9352 | accuracy: 0.464000
Epoch:  1 | train loss: 2.7494 | accuracy: 0.485000
Epoch:  2 | train loss: 2.3103 | accuracy: 0.478000
Epoch:  3 | train loss: 1.9845 | accuracy: 0.456000
Epoch:  4 | train loss: 1.9465 | accuracy: 0.478000
```

In [7]:
```
# Accuracy number on the full training set:

b_in = torch.zeros(len(allnames), padded_length, n_letters)
b_out = torch.zeros(len(allnames), n_categories, dtype=torch.long)


for i, name in enumerate(allnames):
    for li, letter in enumerate(name[0]):
        b_in[i][li][letterToIndex(letter)]=1


for i, name in enumerate(allnames):
    b_out[i][languages.index(name[1])]=1


    # 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)/len(allnames)
print('Accuracy: %.6f' % accuracy)
```

```
Accuracy: 0.468666
```

batch size = 2000

In [8]:
```
learning_rate = 0.005
rnn = RNN(n_letters, 128, 1, n_categories)
optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)   # optimize
 all rnn parameters
loss_func = nn.CrossEntropyLoss()

for epoch in range(5):
    batch_size = 2000
    random.shuffle(allnames)

    maxlen = 0
    for name in allnames[:2000]:
        name_tensor = nameToTensor(name[0])
        if len(name_tensor) >= maxlen:
            maxlen = len(name_tensor)


    b_in = torch.zeros(batch_size, maxlen, n_letters)
    b_out = torch.zeros(batch_size, n_categories, dtype=torch.long)


    for i, name in enumerate(allnames):
        for li, letter in enumerate(name[0]):
            if i < 2000:
                b_in[i][li][letterToIndex(letter)]=1


    for i, name in enumerate(allnames):
        if i < 2000:
            b_out[i][languages.index(name[1])]=1


    output = rnn(b_in)

    loss = loss_func(output, torch.max(b_out, 1)[1])

    optimizer.zero_grad()                           # clear gradients for this
training step
    loss.backward()                                 # backpropagation, compute
gradients
    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)/len(allnames)
    print("Epoch: ", epoch, "| train loss: %.4f" % loss.item(), '| accuracy:
%.6f' % accuracy)
```

```
Epoch:  0 | train loss: 2.8760 | accuracy: 0.046229
Epoch:  1 | train loss: 2.6397 | accuracy: 0.046179
Epoch:  2 | train loss: 2.1250 | accuracy: 0.046129
Epoch:  3 | train loss: 1.8950 | accuracy: 0.047026
Epoch:  4 | train loss: 1.9048 | accuracy: 0.046976
```

In [9]:
```
# Accuracy number on the full training set:

b_in = torch.zeros(len(allnames), padded_length, n_letters)
b_out = torch.zeros(len(allnames), n_categories, dtype=torch.long)


for i, name in enumerate(allnames):
    for li, letter in enumerate(name[0]):
        b_in[i][li][letterToIndex(letter)]=1


for i, name in enumerate(allnames):
    b_out[i][languages.index(name[1])]=1


    # 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('Accuracy: %.6f' % accuracy)
```

```
Accuracy: 4.704000
```

batch size = 3000

In [10]:
```
learning_rate = 0.005
rnn = RNN(n_letters, 128, 1, n_categories)
optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)   # optimize
 all rnn parameters
loss_func = nn.CrossEntropyLoss()

for epoch in range(5):
    batch_size = 3000
    random.shuffle(allnames)

    maxlen = 0
    for name in allnames[:3000]:
        name_tensor = nameToTensor(name[0])
        if len(name_tensor) >= maxlen:
            maxlen = len(name_tensor)


    b_in = torch.zeros(batch_size, maxlen, n_letters)
    b_out = torch.zeros(batch_size, n_categories, dtype=torch.long)


    for i, name in enumerate(allnames):
        for li, letter in enumerate(name[0]):
            if i < 3000:
                b_in[i][li][letterToIndex(letter)]=1


    for i, name in enumerate(allnames):
        if i < 3000:
            b_out[i][languages.index(name[1])]=1


    output = rnn(b_in)

    loss = loss_func(output, torch.max(b_out, 1)[1])

    optimizer.zero_grad()                               # clear gradients for this
training step
    loss.backward()                                     # backpropagation, compute
gradients
    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:
%.6f' % accuracy)
```

```
Epoch:  0 | train loss: 2.9343 | accuracy: 0.468667
Epoch:  1 | train loss: 2.6978 | accuracy: 0.464000
Epoch:  2 | train loss: 2.1261 | accuracy: 0.462667
Epoch:  3 | train loss: 1.9138 | accuracy: 0.479667
Epoch:  4 | train loss: 1.9694 | accuracy: 0.463000
```

In [11]:
```
# Accuracy number on the full training set:

b_in = torch.zeros(len(allnames), padded_length, n_letters)
b_out = torch.zeros(len(allnames), n_categories, dtype=torch.long)


for i, name in enumerate(allnames):
    for li, letter in enumerate(name[0]):
        b_in[i][li][letterToIndex(letter)]=1


for i, name in enumerate(allnames):
    b_out[i][languages.index(name[1])]=1


    # 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)/len(allnames)
print('Accuracy: %.6f' % accuracy)
```

```
Accuracy: 0.468666
```