CS 559: Neural Networks

Homework 7

Author:
Gaetano Coppoletta
Email:
gcoppo2@uic.edu

November 11, 2022

1 Introduction

We want to implement a character-level text generation LSTM. Specifically, the goal will be to generate person names. We will use a text file containing 2000 lineas of names as training set.

2

We train the neural network using the names contained in names.txt. The neural network is trained for 1000 epochs. The loss values versus epochs is illustrated below.

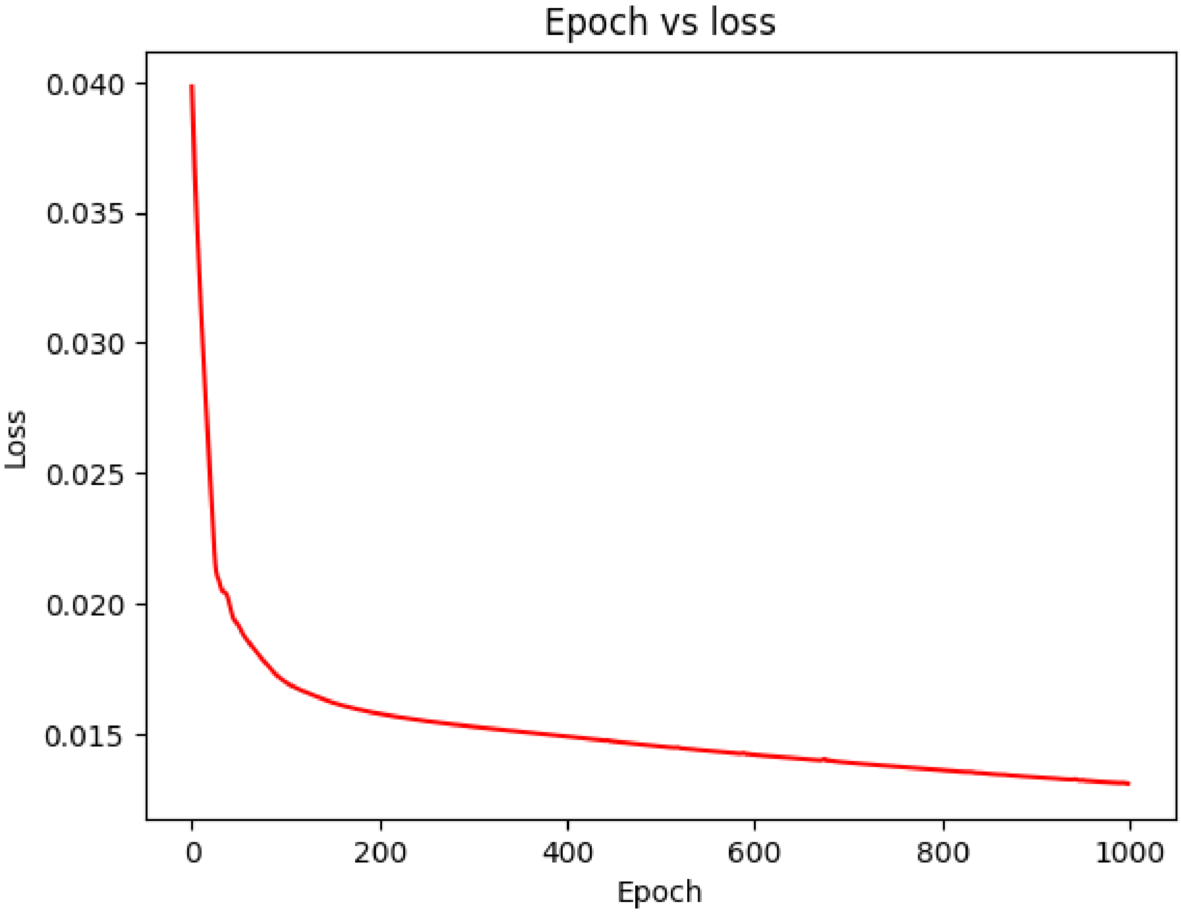

Figure 1: Loss vs epochs

We have used the mean-squared error as loss function because after some tries is the one that lead to better results. Using the cross-entropy, instead, leads to worst performance of the neural network. With the mean-squared error we obtain a loss of 0.0133.

3

In order to produce different names, we don't take always the letter with the highest probability but another algorithm is used. The neural network returns an array containing the probability of each letter. The algorithm works as follow: first we generate a random number between 0 and 1. If this number is greater than 0.7 we set the highest probability of the array returned by the neural network to 0 and we generate another random number, we repeat those steps until the random number is less than 0.7. When the random number is less than 0.7 we take the highest probability from the array returned by the neural network and use its index to choose the next letter. With this algorithm we are able to generate different names.
The names produced with the letter "a" are:

- alelieh
- aley
- alana
- alani
- alianna
- aliana
- anna
- andelyn
- aria
- alisoa
- alena
- alianna
- alelah
- aliannah
- alena
- ana
- annolla
- ariyahely
- alia

- alelia

The names produced with the letter "x" are:

- xannon
- xillie
- xanden
- xilliella
- xani
- xamie
- xannera
- xailee
- xanner
- xillie
- xaniel
- xanie
- xistonl
- xiston
- xani
- xaileyn
- xanniel
- xaniei
- xaiel
- xina

4 Code 1

```python
import argparse
from torch.autograd import Variable
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import matplotlib.pyplot as plt
from torchvision import datasets, transforms
from torch.optim.lr_scheduler import StepLR
import os
import shutil
import numpy as np
import random

def read_file():
    f = open("names.txt","r")
    lines = f.readlines()
    lower_case=[]
    for line in lines:
        lower_case.append(line.lower()[:-1])
    return lower_case

def transform_letter(names):
    matrix_of_letters = np.zeros((2000,11,27))
    num_names=0
    for name in names:
        for i in range(11):
            if i < len(name):
                ascii_number = ord(name[i])-96
            else:
                ascii_number = 0 #end of word encoded
            matrix_of_letters[num_names, i, ascii_number] = 1
        num_names+=1
    return matrix_of_letters

def transform_letter_for_name(name):
    matrix_of_letters = np.zeros((11,27))
    num_names=0
    for i in range(11):
        if i < len(name):
            ascii_number = ord(name[i])-96
        else:
            ascii_number = 0 #end of word encoded
        matrix_of_letters[i, ascii_number] = 1
    num_names+=1
    return matrix_of_letters

def desired_output(names):
    matrix_of_letters = np.zeros((2000,11,27))
    num_names=0
    for name in names:
        for i in range(1,12):
            if i < len(name):
                ascii_number = ord(name[i])-96
            else:
                ascii_number=0
            matrix_of_letters[num_names, i-1, ascii_number] = 1
        num_names+=1
    return matrix_of_letters


class LSTM1(nn.Module):
    def __init__(self, num_classes, input_size, hidden_size, num_layers, seq_length):
        super(LSTM1, self).__init__()
        self.num_classes = num_classes
        self.num_layers = num_layers
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.seq_length = seq_length

        self.lstm = nn.LSTM(input_size=input_size, hidden_size=hidden_size,
                            num_layers=num_layers, batch_first=True) #lstm
        self.fc_1 =  nn.Linear(hidden_size, 128)
        self.fc = nn.Linear(128, num_classes)

        self.relu = nn.ReLU()

    def forward(self,x):
        h_0 = Variable(torch.zeros(self.num_layers, x.size(0), self.hidden_size))
        c_0 = Variable(torch.zeros(self.num_layers, x.size(0), self.hidden_size))

        output, (hn, cn) = self.lstm(x, (h_0, c_0))
        hn = hn.view(-1, self.hidden_size)
        out = self.relu(output)
        out = self.fc_1(out)
        out = self.relu(out)
        out = self.fc(out)
        return out

names = read_file()
inputs = transform_letter(names)

desired_o = desired_output(names)


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


num_epochs = 1000
learning_rate = 0.001
```

```

input_size = 27 #number of features
hidden_size = 64 #number of features in hidden state
num_layers = 1 #number of stacked lstm layers

num_classes = 27 #number of output classes

X_train_tensors_final = torch.tensor(inputs, dtype=torch.float).view(2000,11,27)
y_train_tensors = torch.tensor(desired_o, dtype=torch.float).view(2000,11,27)

lstm1 = LSTM1(num_classes, input_size, hidden_size, num_layers, X_train_tensors_final.shape[1]) #our lstm class

criterion = torch.nn.MSELoss()    # mean-squared error
optimizer = torch.optim.Adam(lstm1.parameters(), lr=learning_rate)

loss_vs_epoch=np.zeros((2,num_epochs))
#0 is the loss, 1 is the epoch
for epoch in range(num_epochs):
    outputs = lstm1.forward(X_train_tensors_final) #forward pass

    optimizer.zero_grad() #caluclate the gradient, manually setting to 0

    loss = criterion(outputs, y_train_tensors)

    loss.backward()
    optimizer.step()
    if epoch % 100 == 0:
        print("Epoch: %d, loss: %1.5f" % (epoch, loss.item()))
    loss_vs_epoch[0,epoch]=epoch
    loss_vs_epoch[1,epoch]=loss.item()
print(loss.item())
plt.title("Epoch vs loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.plot(loss_vs_epoch[0,:epoch],loss_vs_epoch[1,:epoch],’r’)
plt.show()

#save the model
torch.save(lstm1.state_dict(), "0702-657811153-Coppoletta.ZZZ")
```

5 Code 2

```
import torch
import torch.nn as nn
from torch.autograd import Variable
import numpy as np
import random

def transform_letter_for_name(name):
    matrix_of_letters = np.zeros((11,27))
```

```python
    if len(name)>=11:
        new_name = name[-10:]
       # print("NAME:",new_name)
    else:
        new_name=name
    for i in range(11):
        if i < len(new_name):
            ascii_number = ord(new_name[i])-96
        else:
            ascii_number = 0 #end of word encoded
        matrix_of_letters[i, ascii_number] = 1
    return matrix_of_letters


class LSTM1(nn.Module):
    def __init__(self, num_classes, input_size, hidden_size, num_layers, seq_length):
        super(LSTM1, self).__init__()
        self.num_classes = num_classes
        self.num_layers = num_layers
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.seq_length = seq_length

        self.lstm = nn.LSTM(input_size=input_size, hidden_size=hidden_size,
                          num_layers=num_layers, batch_first=True) #lstm
        self.fc_1 =  nn.Linear(hidden_size, 128)
        self.fc = nn.Linear(128, num_classes)

        self.relu = nn.ReLU()

    def forward(self,x):
        h_0 = Variable(torch.zeros(self.num_layers, x.size(0), self.hidden_size)) #hidden state
        c_0 = Variable(torch.zeros(self.num_layers, x.size(0), self.hidden_size)) #internal state

        output, (hn, cn) = self.lstm(x, (h_0, c_0))
        hn = hn.view(-1, self.hidden_size)
        out = self.relu(output)
        out = self.fc_1(out)
        out = self.relu(out)
        out = self.fc(out)
        return out

input_size = 27 #number of features
hidden_size = 64 #number of features in hidden state
num_layers = 1 #number of stacked lstm layers

num_classes = 27 #number of output classes

lstm1 = LSTM1(num_classes, input_size, hidden_size, num_layers, 11)
lstm1.eval()
```

```

lstm1.load_state_dict(torch.load(’0702-657811153-Coppoletta.ZZZ’))
#use the model
l=input("Insert a letter:\n")
letter=l
letter_encoded=transform_letter_for_name(letter)
letter_tensor =torch.tensor(letter_encoded, dtype=torch.float).view(1,11,27)
name=letter
num_letters=1
position=1
num_names=0
while num_names<20:
    while True:

        output = lstm1.forward(letter_tensor)
        random_number = random.uniform(0,1)
        while random_number>0.7:
            output[:,num_letters-1, np.argmax(output[:,position-1,:].cpu().detach().numpy())]=0
            random_number = random.uniform(0,1)

        letter = np.argmax(output[:,position-1,:].cpu().detach().numpy())
        letter+=96
        letter= chr(letter)
        if(letter==’‘’):
            break
        position+=1
        if position>=11:
            position=10
        name+=letter
        letter_encoded=transform_letter_for_name(name)
        letter_tensor= torch.tensor(letter_encoded, dtype=torch.float).view(1,11,27)
        num_letters+=1

    print(name[:],num_letters)
    letter=l
    position=1
    name = letter
    num_letters=1
    num_names+=1
    letter_encoded=transform_letter_for_name(letter)
    letter_tensor =torch.tensor(letter_encoded, dtype=torch.float).view(1,11,27)
```