BME 495 Homework 4 PDF
By: Siddharth Shah shah255@purdue.edu

Codes

NeuralNetwork.py

```
import torch
import math
import numpy as np
import copy
import random

random.seed(0)

class NeuralNetwork:

    def __init__(self, inputVars):
        self.output_dim = inputVars[len(inputVars) - 1]
        self.theta = {}
        for i in range(len(inputVars) - 1):
            mat = torch.FloatTensor(inputVars[i] + 1, inputVars[i + 1])
            sigma = 1 / math.sqrt(inputVars[i])
            mat.normal_(mean=0, std=sigma)
            self.theta[i] = mat
        self.localGrads = {}
        self.dE_dTheta = {}

    def forward(self, input2d):
        preds = torch.FloatTensor(input2d.size()[0], self.output_dim).zero_()
        # preds = torch.FloatTensor(1, 2).zero_()
        i = 0
        for inp in input2d:
            prediction = self.forward1d(inp)
            preds[i] = prediction
            i += 1
        return preds

    def forward1d(self, inputTensor):
        inp = copy.deepcopy(inputTensor)
        self.localGrads[0] = inp
        for i in range(len(self.theta)):
            layer = self.theta[i]
            inputHat = torch.cat((inp, torch.FloatTensor([1])))
            output = torch.FloatTensor(np.dot(inputHat, layer))
            output = torch.pow((1 + torch.exp(-output)), -1)
            inp = copy.deepcopy(output)
            # print("sig", inp)
            self.localGrads[i + 1] = inp
        return torch.FloatTensor(inp)

    def getLayer(self, layerNum):
        return self.theta[layerNum]
```

```
    def backward1d(self, target):
        error = 0
        errors = []
        lDers = []
        lDiffs = []
        e = 0
        for i in reversed(range(len(self.theta))):
            layer = self.theta[i]
            lGrads = self.localGrads[i + 1]
            if i == len(self.theta) - 1:
                error = lGrads - target
                lDiffs.append(error)
                deriv = lGrads * (1 - lGrads)
                error = error * deriv
                error = error.resize_(1, len(error))
                errors.append(error)
                error = error.t() * self.localGrads[i]
                # error = torch.cat((error, torch.zeros(len(error), len(error[0]))))
                # print("fin_error", error)
            else:
                error_sums = []
                for error in errors:
                    error = error * self.theta[i + 1]
                    for row in error:
                        error_sums.append(sum(row))
                error = torch.FloatTensor(error_sums[:-1])
                gradDerivs = lGrads * (1 - lGrads)
                error = error * gradDerivs
                error = error.resize_(1, len(error))
                error = error.t() * self.localGrads[i]
                # print("Lerror", error)
                # error = torch.cat((error, torch.zeros(len(error), len(error[0]))))
            # print("error", error.t())
            # input("")
            errN = error.t()
            errN = torch.cat((errN, torch.zeros(1, len(errN[0]))))
            self.dE_dTheta[i] = errN

    def backward(self, target):
        for tgt in target:
            self.backward1d(tgt)

    def updateParams(self, eta):
        for i in range(len(self.theta)):
            # print("theta_i", self.theta[i], "dedtheta_i", self.dE_dTheta[i])
            # input("")
            self.theta[i] = self.theta[i] - eta * self.dE_dTheta[i]
            # print("updated_theta", self.theta[i])
```

2) my_img2num.py

```
from torchvision import datasets, transforms
from NeuralNetwork import NeuralNetwork
```

```
import torch
from torch.autograd import Variable
import torch.nn.functional as F
import numpy as np
from matplotlib import pyplot as plt
import pickle

class MyImg2Num:

    def __init__(self, fromFile = False):
        self.train_loader = torch.utils.data.DataLoader(datasets.MNIST('data', train=True,
download=True, transform=transforms.ToTensor()),batch_size=600, shuffle=True)
        self.test_loader = torch.utils.data.DataLoader(datasets.MNIST('data', train=False,
download=False, transform=transforms.ToTensor()), batch_size=600, shuffle=True)
        if not fromFile:
            self.model = NeuralNetwork((784, 32, 10))
            # self.model = NeuralNetwork((2, 2, 2))
        else:
            pickle_in = open("myimagenn.pkl", "rb")
            self.model = pickle.load(pickle_in)
            pickle_in.close()
        self.max_epochs = 10

    def __oneHot(self, target):
        return np.eye(10, dtype='uint8')[target]
        # return np.eye(2, dtype='uint8')[target]

    def train(self):
        all_mse = []
        for epoch in range(self.max_epochs):
            print("epoch", epoch)
            epoch_mse = []
            for data, target in self.train_loader:
                oneDX = data.resize_(600, 784)
                oneDY = torch.FloatTensor(self.__oneHot(target)).resize_(600, 10)
                y_pred = self.model.forward(oneDX)
                mse = F.mse_loss(Variable(y_pred), Variable(oneDY))
                epoch_mse.append(mse)
                self.model.backward(oneDY)
                self.model.updateParams(0.5)
            all_mse.append(sum(epoch_mse) / len(epoch_mse))
        print("train_err", all_mse)
        modelFile = open("myimagenn.pkl", "wb")
        pickle.dump(self.model, modelFile)
        modelFile.close()

    def __evaluate(self):
        all_mse = []
        for epoch in range(10):
            epoch_mse = []
            for data, target in self.test_loader:
                oneDX = data.resize_(600, 784)
                oneDY = torch.FloatTensor(self.__oneHot(target)).resize_(600, 10)
                y_pred = self.model.forward(oneDX)
```

```
            mse = F.mse_loss(Variable(y_pred), Variable(oneDY))
            tyt = oneDY.numpy()
            tyy = y_pred.numpy()
            eqn = 0
            for j in range(len(tyy)):
                if list(tyy[j]).index(max(tyy[j])) == list(tyt[j]).index(max(tyt[j])):
                    eqn+=1
            # print(eqn / 600, mse.numpy()[0])
            # print(oneDX[0], oneDY[0], y_pred[0])
            # input("")
            epoch_mse.append(mse)
        all_mse.append(sum(epoch_mse) / (len(epoch_mse) * len(all_mse)))
    print("eval_err", all_mse)

  def forward(self, img):
    imgSize = img.size()
    nImg = img.resize_(1, imgSize[2] * imgSize[3])
    y_pred = self.model.forward(img)
    print(list(y_pred[0]).index(max(list(y_pred[0]))))

if __name__ == '__main__':
  # nn = MyImg2Num(fromFile = False)
  nn = MyImg2Num(fromFile = True)
  nn.train()
  # nn.evaluate()
  test_loader = torch.utils.data.DataLoader(datasets.MNIST('data', train=False,
download=False, transform=transforms.ToTensor()), batch_size=1, shuffle=True)
  for data, target in test_loader:
    nn.forward(data)
    print(target)
    break
```

3) nn_img2num.py

```
from torchvision import datasets, transforms
from NeuralNetwork import NeuralNetwork
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
from matplotlib import pyplot as plt
from torch.autograd import Variable
import pickle

class NnImg2Num:

  def __init__(self, fromFile = False):
    self.train_loader = torch.utils.data.DataLoader(datasets.MNIST('data', train=True,
download=True, transform=transforms.ToTensor()),batch_size=600, shuffle=False)
    self.max_epochs = 5
    if not fromFile:
      self.model = torch.nn.Sequential(torch.nn.Linear(784, 32), torch.nn.Sigmoid(),
torch.nn.Linear(32, 10))
```

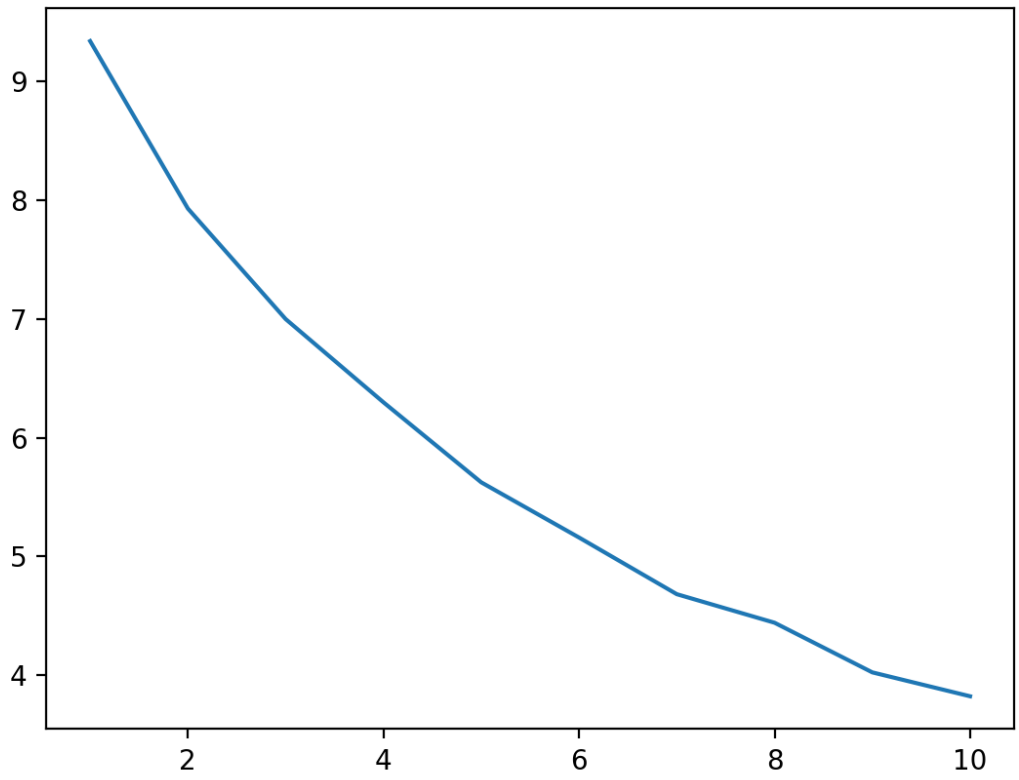
```
        else:
            pickle_in = open("pyimagenn.pkl", "rb")
            self.model = pickle.load(pickle_in)
            pickle_in.close()
        self.test_loader = torch.utils.data.DataLoader(datasets.MNIST('data', train=False,
download=True, transform=transforms.ToTensor()), batch_size=600, shuffle=True)

    def __oneHot(self, target):
        return np.eye(10, dtype='uint8')[target]

    def train(self):
        criterion = nn.MSELoss(size_average=False)
        optimizer = optim.SGD(self.model.parameters(), lr=1e-4)
        all_mse = []
        for epoch in range(self.max_epochs):
            epoch_mse = []
            for data, target in self.train_loader:
                oneDX = data.resize_(600, 784)
                oneDY = torch.FloatTensor(self.__oneHot(target)).resize_(600, 10)
                y_pred = self.model(Variable(oneDX))
                loss = criterion(y_pred, Variable(oneDY))
                # print("loss", loss.data[0])
                epoch_mse.append(loss)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()
            all_mse.append(sum(epoch_mse) / len(epoch_mse))
        print("train_err", all_mse)
        modelFile = open("pyimagenn.pkl", "wb")
        pickle.dump(self.model, modelFile)
        modelFile.close()

    def forward(self, img):
        imgSize = img.size()
        nImg = img.resize_(1, imgSize[2] * imgSize[3])
        y_pred = self.model(Variable(img))
        print(list(y_pred[0]).index(max(list(y_pred[0]))))

    def __evaluate(self):
        criterion = nn.MSELoss(size_average=False)
        optimizer = optim.SGD(self.model.parameters(), lr=1e-4)
        all_mse = []
        for epoch in range(self.max_epochs):
            epoch_mse = []
            for data, target in self.test_loader:
                oneDX = data.resize_(600, 784)
                oneDY = torch.FloatTensor(self.__oneHot(target)).resize_(600, 10)
                y_pred = self.model(Variable(oneDX))
                loss = criterion(y_pred, Variable(oneDY))
                # print("eval_loss", loss.data[0])
                epoch_mse.append(loss)
            all_mse.append(sum(epoch_mse) / len(epoch_mse))
        print("eval_err", all_mse)
```

```
if __name__ == '__main__':
    snn = NnImg2Num(fromFile = True)
    snn.train()
    #snn.evaluate()
    test_loader = torch.utils.data.DataLoader(datasets.MNIST('data', train=False,
download=False, transform=transforms.ToTensor()), batch_size=1, shuffle=True)
    for data, target in test_loader:
        snn.forward(data)
        print(target)
        break
```

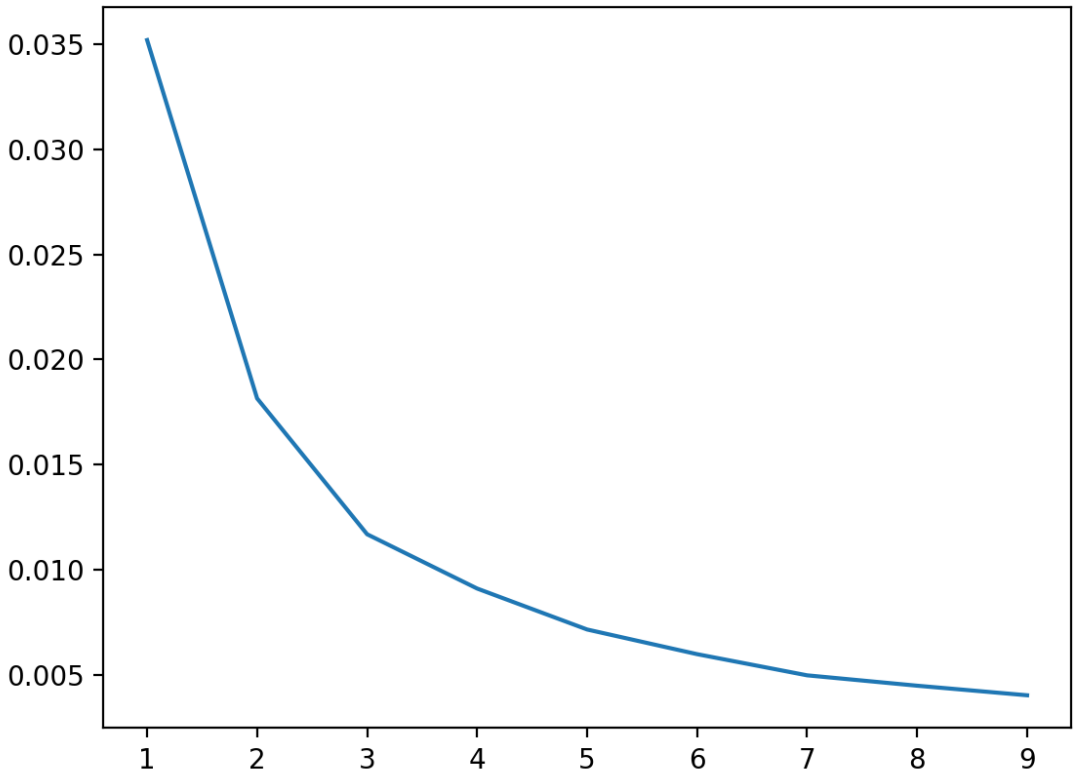
The evaluate functions are for plotting test error plots.

Plots

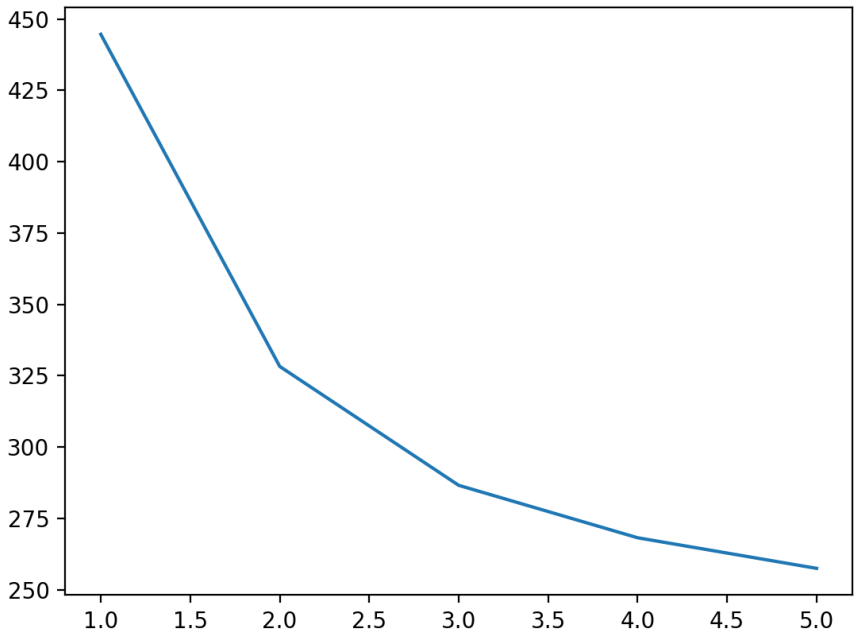
1) my_img2num Training



my_img2num Testing



nn_img2num Training



nn_img2num Testing