Atanas Delevski

ECE 559 Homework #4 Report

11/02/2020

Question 1:

Architecture:

* Topology:
  + For my topology, I chose right away to have 2 hidden layers. I did not change this throughout my design process. I assumed it was going to be possible to achieve >90% on the testing set with two hidden layers, so I never changed that. In terms of the number of neurons per layer, I started out small (around 10 per hidden layer), and then went up from there. I eventually decided to go with 100 neurons per hidden layer.
* Desired Output format:
  + For my desired output format, I did the same thing as the HW #2 way. I had 10 neurons in the output layer, and used the softmax function. Then the neuron closest to 1 will be the predicted integer, i.e. “winner”
* Activation functions, eta, dynamic updating of eta:
  + For my activation functions, I went with Sigmoid and Softmax. Sigmoid for the 1st and 2nd hidden layers, and then softmax for the output layer. For my backpropagation algorithm, I had to use the derivatives of these functions.
  + For my learning rate, I started out with 0.1, but that was only getting me about ~70% accuracy, so then I started changing it.
  + I also decided to have a dynamit learning rate, and so I multiplied it by 0.9 everytime the accuracy decreased instead of increasing.
* Distance function:
  + For my cost functions, I used the euclidean distance to measure the loss.
* Other Tricks
  + I used batch learning so that I could have many quick epochs instead of a few slow epochs.

Design Process for Architecture:

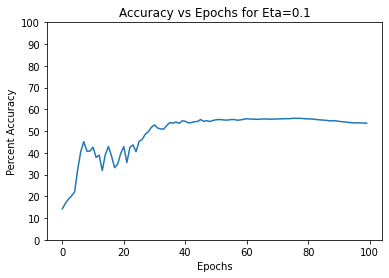
I changed many different things over the course of my testing. I first began with my weights being initialized only between [-.1,.1], but then tried the interval [-1,1] and also [-.3,.3]. It seemed to work best with the last one. I also changed the number of neurons for my hidden layers a lot as well. I first started out with 48, and 24. Then I moved on to 100, and 100. Then I tried 50 and 50, 256 and 128, 392 and 186, 100 and 50.

I also changed my learning rate a lot as well. I started with 0.1, then 0.01, then 0.05, then 0.001, 0.025, etc.

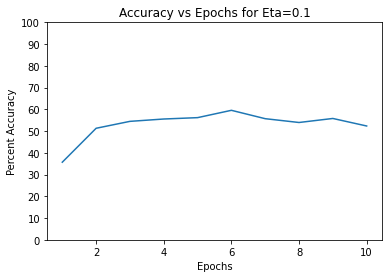
The batch size was another thing I messed around with but I do not think it had such a big impact.

I kept trying to change all of these things and I kept trying to come up with different combinations of changes that would result in higher accuracies. It was honestly pretty difficult to figure out the magical combination… It probably took me like almost 50 tries.

Plots:

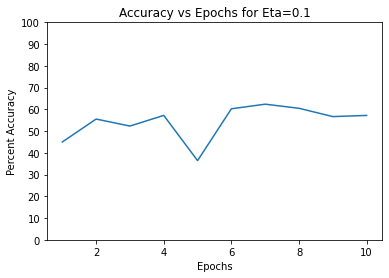
1st:

2nd: (different weight initialization)

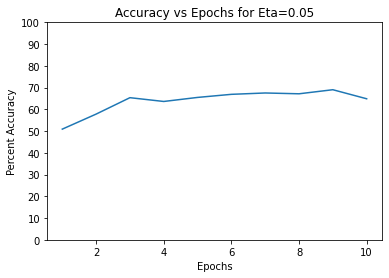


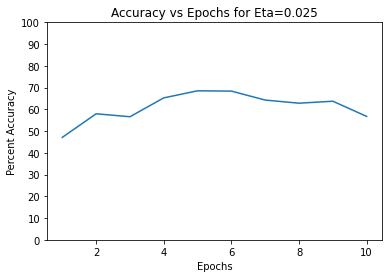
3rd:

(Here I increased the number of neurons to 196, and 98 for hidden layer 1 and 2, respectively)



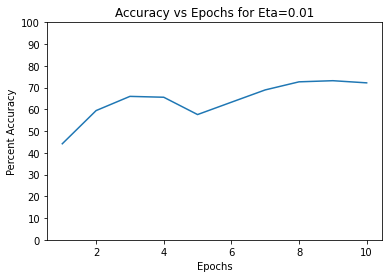
4) This one is for eta=0.05



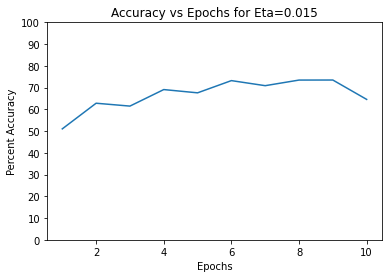
5) (Eta = 0.025)

6)

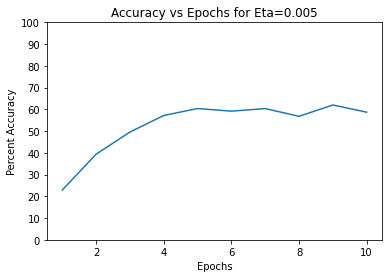
(This one I changed the hidden layers to have 400 and 200 neurons respectively, and I also changed the learning rate back to 0.01)



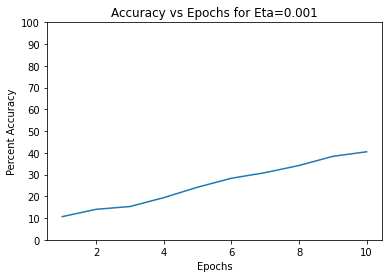
7) Same as 6 but I changed the LR to 0.015



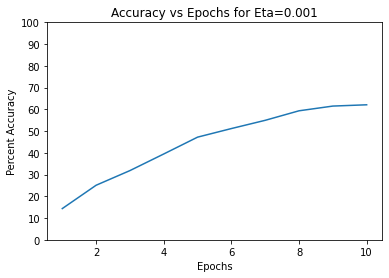
8) (Eta = 0.005)

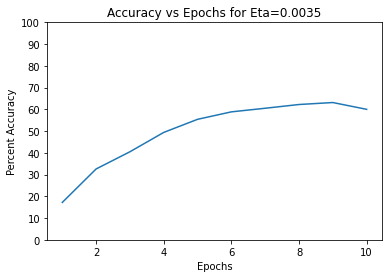


9) (Eta = 0.001)



10) (Eta = 0.0025)



11) (Eta = 0.0035)

Code:

# !pip install python-mnist

from matplotlib import pyplot as plt

from mnist import MNIST

import numpy as np

import time

np.random.seed(1)

mnist = MNIST('MNIST')

rs, rl = mnist.load\_training() #60000 samples

ts, tl = mnist.load\_testing() #10000 samples

train\_set = np.asarray(rs)

train\_labels = np.asarray(rl)

test\_set = np.asarray(ts)

test\_labels = np.asarray(tl)

sizes = [784, 100, 100, 10] # The number of neurons per layer (input, hidden\_layer\_1, hidden\_layer\_2, output\_layer)

epochs = 10 # The number of epochs (epochs \* batch MUST == 60,000 !!!!!)

eta = 0.01 # The learning rate (eta)

batch = 6000 # Using batch learning for faster epochs (epochs \* batch MUST == 60,000 !!!!!!)

def sigmoid(x):

return 1/(1 + np.exp(-x))

def sigmoid\_der(x):

f = sigmoid(x)

df = f \* (1 - f)

return df

def softmax(x):

e\_x = np.exp(x)

return e\_x / e\_x.sum()

def softmax\_der(x):

f = softmax(x)

df = f \* (1 - f)

return df

def shuffle(a, b):

rng\_state = np.random.get\_state() # function for shuffling training and testing sets (called before each epoch)

np.random.shuffle(a)

np.random.set\_state(rng\_state)

np.random.shuffle(b)

class DNN():

def \_\_init\_\_(self, eta):

self.eta = eta

self.weights = self.weight\_initialization()

self.accuracy\_list = []

self.error\_list = []

def weight\_initialization(self):

w1 = np.random.uniform(-.3,.3,(sizes[1], sizes[0]+1)) # weights on interval [-0.3, 0.3]

w2 = np.random.uniform(-.3,.3,(sizes[2], sizes[1]+1))

w3 = np.random.uniform(-.3,.3,(sizes[3], sizes[2]+1))

weights = [w1, w2, w3]

return weights

def forward(self, tsample):

temp1 = np.ones(len(tsample)+1)

temp1[1:] = tsample

v1 = self.weights[0]@temp1

y1 = sigmoid(v1) # forward pass algorithm

# returns v1, v2, v3, yL and some temp values for convenience

temp2 = np.ones(len(y1)+1)

temp2[1:] = y1

v2 = self.weights[1]@temp2

y2 = sigmoid(v2)

temp3 = np.ones(len(y2)+1)

temp3[1:] = y2

v3 = self.weights[2]@temp3

y3 = softmax(v3)

return temp1, temp2, temp3, v1, v2, v3, y3

def backward(self, lsample, t1, t2, t3, v1, v2, v3, yL):

d = np.zeros(10) # backwards pass algorithm

d[lsample] = 1 # desired vector

error = np.square(sum(d-yL))

delta3 = (d-yL)\*softmax\_der(v3)

cost3 = delta3.reshape(sizes[3], 1)@t3.reshape(1, sizes[2]+1)

self.weights[2] = self.weights[2] + self.eta\*(cost3)

delta2 = (self.weights[2][:,1:].reshape(sizes[2], sizes[3])@delta3)\*sigmoid\_der(v2)

cost2 = delta2.reshape(sizes[2], 1)@t2.reshape(1, sizes[1]+1)

self.weights[1] = self.weights[1] + self.eta\*(cost2)

delta1 = (self.weights[1][:,1:].reshape(sizes[1], sizes[2])@delta2)\*sigmoid\_der(v1)

cost1 = delta1.reshape(sizes[1], 1)@t1.reshape(1, sizes[0]+1)

self.weights[0] = self.weights[0] + self.eta\*(cost1)

return error

def compute\_accuracy(self, test\_set, test\_labels):

correct = 0

incorrect = 0

for i in range(len(test\_set)):

temp1, temp2, temp3, v1, v2, v3, y3 = self.forward(test\_set[i]) # computing accuracy by testing on all 10k Test samples

pred = np.argmax(y3)

if pred == test\_labels[i]:

correct += 1

else:

incorrect += 1

return correct, incorrect

def train(self, train\_set, train\_labels, test\_set, test\_labels):

start\_time = time.time()

min = 0 # train function

max = 0

for epoch in range(1, epochs+1):

# shuffle(train\_set, train\_labels) #shuffling of training and testing sets before each epoch

# shuffle(test\_set, test\_labels)

max = epoch\*batch

temp = 0

for i in range(min, max):

temp1, temp2, temp3, v1, v2, v3, y3 = self.forward(train\_set[i])

error = self.backward(train\_labels[i], temp1, temp2, temp3, v1, v2, v3, y3)

self.error\_list.append(error)

correct, incorrect = self.compute\_accuracy(test\_set, test\_labels)

accuracy = (correct/(correct+incorrect))\*100

min = epoch\*batch

if accuracy < temp:

self.eta = 0.9\*self.eta # dynamic learning rate

temp = accuracy

self.accuracy\_list.append(accuracy)

print(f'Epoch: {epoch}, Time Spent: {time.time() - start\_time}s')

print(f'Correct: {correct}, Incorrect: {incorrect}, Accuracy: {accuracy}%')

dnn = DNN(eta)

dnn.train(train\_set, train\_labels, test\_set, test\_labels) # USAGE

eps = np.arange(1, epochs+1)

plt.plot(eps, dnn.accuracy\_list)

plt.xlabel('Epochs')

plt.ylabel('Percent Accuracy') #PLOTTING

plt.title('Accuracy vs Epochs for Eta=0.005')

plt.yticks(np.arange(0, 110, 10))

ers = np.arange(0, len(train\_set))

plt.plot(ers, dnn.error\_list)

plt.xlabel('Epochs')

plt.ylabel('Energy (Euclidean Distance)')

plt.title('Energy vs Epochs')