

CSCE 501-002

Assignment 1

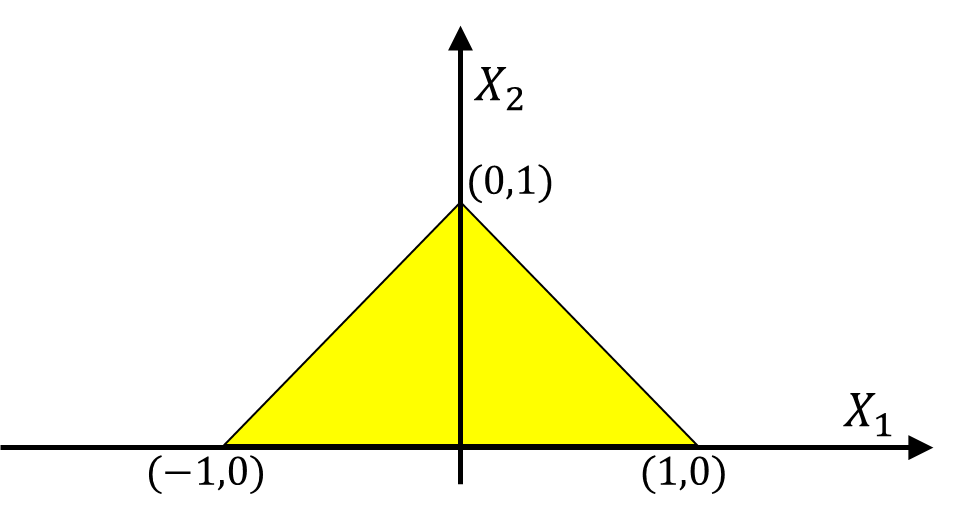
Imam Al Razi

ID-010850660

E-mail: [ialrazi@email.uark.edu](mailto:ialrazi@email.uark.edu)

**Problem 1**

We want to build an MLP that composes the decision boundary shown in the figure below. The output of the MLP must 1 in the yellow regions and 0 elsewhere



**Solution:**

As there are three boundary lines, for the first layer, we need three perceptrons. Then, to have the yellow shaded area as output = 1, another perceptron is required with an AND operation. The network is shown here:



**Problem 2**

We are given the relationship

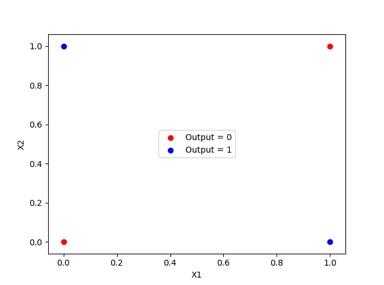
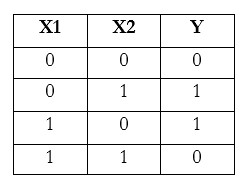
𝑦 = 𝑓(𝐱, 𝑔(𝐱, 𝑐))𝑔(𝐱, 𝑓(𝐱, 𝑑))

Where **x** is a vector. 𝑓(.) and 𝑔(.) are both scalar functions which take a vector and a scalar as inputs. Compute .

**Solution:**

**Problem 3**

Manually design MLP network to perform the XOR Gate with the truth table and its plot on 2D as follows:



Start with uniform random initialization for parameters wij . Perform forward and backward pass in the following case:

1. Activation function is (a) Sigmod (b) ReLu, (c)Tanh
2. Divergence is defined as (a) L2\_norm, (b) cross entropy
3. Train the network for 2 iterations (3 forward pass and 2 backward pass)

Please report the parameters and actual output from the MLP in each iteration

**Solution:**

**Code:**

import numpy as np

np.random.seed(1)

class Neural\_Network(object):

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

#Define Hyperparameters

self.inputLayerSize = 2

self.outputLayerSize = 1

self.hiddenLayerSize = 2

#Weights (parameters)

self.init\_weights()

self.init\_bias()

def init\_weights(self):

self.Wh = np.random.rand(self.inputLayerSize, self.hiddenLayerSize)

self.Wo = np.random.rand(self.hiddenLayerSize, self.outputLayerSize)

def init\_bias(self):

self.Bh = np.full((1, self.hiddenLayerSize), np.random.rand())

self.Bo = np.full((1, self.outputLayerSize), np.random.rand())

def sigmoid(self,Z):

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

def sigmoid\_prime(self,Z):

return self.sigmoid(Z)\*(1-self.sigmoid(Z))

def ReLu(self,Z):

return Z\*(Z>=0)

def ReLu\_prime(self,Z):

return 1\*(Z>=0)

def Tanh(self,Z):

return np.tanh(Z)

def Tanh\_prime(self,Z):

return 1.0 - np.tanh(Z)\*\*2

def activation\_function(self,opt,Z):

if opt==1:

return self.sigmoid(Z)

elif opt==2:

return self.ReLu(Z)

elif opt==3:

return self.Tanh(Z)

def activation\_function\_prime(self,opt,Z):

if opt==1:

return self.sigmoid\_prime(Z)

elif opt==2:

return self.ReLu\_prime(Z)

elif opt==3:

return self.Tanh\_prime(Z)

def feed\_forward(self,X,opt):

'''

X - input matrix

opt - option for activation function (1:sigmoid, 2:ReLu, 3:Tanh)

Zh - hidden layer weighted input

Zo - output layer weighted input

H - hidden layer activation

y - output layer

yHat - output layer predictions

'''

# Hidden layer

self.Zh = np.dot(X, self.Wh) + self.Bh

self.H = self.activation\_function(opt,self.Zh)

# Output layer

self.Zo = np.dot(self.H, self.Wo) + self.Bo

self.yHat = self.activation\_function(opt,self.Zo)

def costFunctionPrime(self, X, y,opt,choice):

#Compute derivative with respect to W and W2 for a given X and y:

self.feed\_forward(X,opt)

if choice==1:

div\_loss=(y - self.yHat)

elif choice==2:

div\_loss=np.zeros(self.yHat.shape)

div\_loss[y==1]=-1/self.yHat[y==1]

div\_loss[y==0]=1/(1-self.yHat[y==0])

dJdWo = np.dot(self.H.T,(div\_loss\* self.activation\_function\_prime(opt,self.Zo)))

dJdWh = np.dot(X.T, (np.dot(div\_loss \* self.activation\_function\_prime(opt,self.Zo), self.Wo.T) \* self.activation\_function\_prime(opt,self.Zh)))

dJdbh = np.sum((np.dot((div\_loss\*self.activation\_function\_prime(opt,self.Zo)),self.Wo.T))\* self.activation\_function\_prime(opt,self.Zh))

dJdbo = np.sum(div\_loss\*self.activation\_function\_prime(opt,self.Zo))

return dJdWh, dJdWo,dJdbh,dJdbo

def backprop(self,X, y,lr,choice,opt):

dWh, dWo, dBh, dBo = self.costFunctionPrime(X, y,opt,choice)

self.Wh = self.Wh + dWh \* lr

self.Wo = self.Wo + dWo \* lr

self.Bh = self.Bh + dBh \* lr

self.Bo = self.Bo + dBo \* lr

#if \_\_name\_\_== "main":

X=np.array([[0,0],[0,1],[1,0],[1,1]])

y=np.array([[0],[1],[1],[0]])

lr=0.1

iters=2

opts=[1,2,3]

choices=[1,2]

for j in range(len(opts)):

NN=Neural\_Network()

opt=opts[j]

for k in range(len(choices)):

choice=choices[k]

print "OPT:",opt

print "Choice:",choice

for i in range(iters):

NN.feed\_forward(X,opt)

NN.backprop(X,y,lr,choice,opt)

print "After Iteration",i+1

print "Hidden\_Layer\_Weights",NN.Wh

print "Hidden\_Layer\_bias",NN.Bh

print "Output\_Layer\_Weights",NN.Wo

print "Output\_Layer\_bias",NN.Bo

print "Output", NN.yHat

NN.feed\_forward(X,opt)

print "Hidden\_Layer\_Weights",NN.Wh

print "Hidden\_Layer\_bias",NN.Bh

print "Output\_Layer\_Weights",NN.Wo

print "Output\_Layer\_bias",NN.Bo

print "Output", NN.yHat

**Results:**

1. Activation function = “Sigmoid”:

(a) Loss function = “L2\_norm”:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Iteration # | Hidden\_Layer\_Weights | Hidden\_Layer\_bias | Output\_Layer\_Weights | Output\_Layer\_bias | Output |
| 1 | [[4.17e-01 7.20e-01]  [2.88e-04 3.02e-01]] | [[0.186 0.186]] | [[0.1533]  [0.0994 ]] | [[0.35657451]] | [[0.61685237]  [0.61845272]  [0.62391468]  [0.62516723]] |
| 2 | [[4.17e-01 7.20e-01]  [2.95e-04 3.02e-01]] | [[0.186 0.186]] | [[0.1535]  [0.0997]] | [[0.35699465]] | [[0.621227 ]  [0.62294524]  [0.62868852]  [0.63003223]] |
| 3rd Forward Pass | [[4.17e-01 7.20e-01]  [2.95e-04 3.02e-01]] | [[0.186 0.186]] | [[0.15359096]  [0.09975817]] | [[0.35699465]] | [[0.62139453]  [0.62311739]  [  0.62887147]  [0.63021874]] |

(b) Loss function = “cross entropy”:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Iteration # | Hidden\_Layer\_Weights | Hidden\_Layer\_bias | Output\_Layer\_Weights | Output\_Layer\_bias | Output |
| 1 | [[4.17e-01 7.20e-01]  [2.94e-04 3.02e-01]] | [[0.186 0.186]] | [[0.1535]  [0.0997]] | [[0.35692588]] | [[0.62139453]  [0.62311739]  [0.62887147] [0.63021874]] |
| 2 | [[4.17e-01 7.20e-01]  [2.94e-04 3.02e-01]] | [[0.186 0.186]] | [[0.1535]  [0.0997]] | [[0.35693714]] | [[0.6213671 ]  [0.62308921]  [0.62884152]  [0.63018821]] |
| 3rd Forward Pass | [[4.17e-01 7.20e-01]  [2.94e-04 3.02e-01]] | [[0.186 0.186]] | [[0.15355652]  [0.09971985]] | [[0.35693714]] | [[0.62137159]  [0.62309382]  [0.62884642]  [0.6301932 ]] |

2. Activation function = “ReLu”:

(a) Loss function = “L2\_norm”:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Iteration # | Hidden\_Layer\_Weights | Hidden\_Layer\_bias | Output\_Layer\_Weights | Output\_Layer\_bias | Output |
| 1 | [[ 0.41493 0.71895] [-0.00351 0.29996]] | [[0.17284 0.17284]] | [[0.13699]  [0.07115]] | [[0.3015187]] | [[0.40426918] [0.43447352] [0.54017733] [0.57038167]] |
| 2 | [[ 0.41221 0.71754] [-0.00598 0.29867]] | [[0.16573 0.16573]] | [[0.12291]  [0.045 ]] | [[0.26735965]] | [[0.34208635]  [0.43274728]  [0.35289814]  [0.64103395]] |
| 3rd Forward Pass | [[0.41221 0.71754]  [-0.00598 0.29867]] | [[0.16573 0.16573]] | [[0.12291263] [0.0455603 ]] | [[0.26735965]] | [[0.36520978]  [0.46835011]  [0.39459023]  [0.66166997]] |

(b) Loss function = “cross entropy”:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Iteration # | Hidden\_Layer\_Weights | Hidden\_Layer\_bias | Output\_Layer\_Weights | Output\_Layer\_bias | Output |
| 1 | [[0.41956 0.72026] [0.00076 0.30118]] | [[0.18319851 0.18319851]] | [[0.16443078] [0.12209497]] | [[0.37103171]] | [[0.36520978]  [0.46835011]  [0.39459023]  [0.66166997]] |
| 2 | [[ 0.39022 0.69847] [-0.02502 0.2820]] | [[0.09944918 0.09944918]] | [[ 0.03588006]  [-0.10725715]] | [[0.07873921]] | [[0.35136858]  [0.44136289]  [0.35136858]  [0.71886144]] |
| 3rd Forward Pass | [[0.3902 0.69847]  [-0.02502 0.28202]] | [[0.09944918 0.09944918]] | [[ 0.03588006]  [-0.10725715]] | [[0.07873921]] | [[0.35006347]  [0.65385821]  [0.64978023]  [1.23263521]] |

3. Activation function = “Tanh”:

(a) Loss function = “L2\_norm”:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Iteration # | Hidden\_Layer\_Weights | Hidden\_Layer\_bias | Output\_Layer\_Weights | Output\_Layer\_bias | Output |
| 1 | [[ 0.38676 0.70611] [-0.02833 0.28963]] | [[0.10855063 0.10855063]] | [[-0.02232229]  [-0.21101983]] | [[-0.11444101]] | [[0.84237693]  [0.90311116]  [0.95375232]  [0.96533756]] |
| 2 | [[ 0.38775 0.71299] [-0.02732 0.29700]] | [[0.12585137 0.12585137]] | [[-0.05040367]  [-0.26152037]] | [[-0.2115684]] | [[0.8100415 ]  [0.88303578]  [0.94516725]  [0.95935902]] |
| 3rd Forward Pass | [[ 0.38775 0.71299] [-0.02732 0.29700]] | [[0.12585137 0.12585137]] | [[-0.05040367]  [-0.26152037]] | [[-0.2115684]] | [[0.76673228]  [0.85577097]  [0.93355129]  [0.95138511]] |

(b) Loss function = “cross entropy”:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Iteration # | Hidden\_Layer\_Weights | Hidden\_Layer\_bias | Output\_Layer\_Weights | Output\_Layer\_bias | Output |
| 1 | [[ 0.38938 0.71923] [-0.02523 0.30558]] | [[0.15097123 0.15097123]] | [[-0.07548259]  [-0.30574885]] | [[-0.31034454]] | [[0.76673228]  [0.85577097]  [0.93355129]  [0.95138511]] |
| 2 | [[ 0.3907 0.72331]  [-0.0234 0.31137]] | [[0.1689692 0.1689692]] | [[-0.09136964]  [-0.33263613]] | [[-0.36845206]] | [[0.96937507]  [0.98251721]  [0.98994769]  [0.99208102]] |
| 3rd Forward Pass | [[ 0.3907 0.72331]  [-0.0234 0.31137]] | [[0.1689692 0.1689692]] | [[-0.09136964]  [-0.33263613]] | [[-0.36845206]] | [[0.99731293]  [0.99835904]  [0.99886269]  [0.99903997]] |

**Problem 4**

Building a MLP with one hidden layer to perform classification task with the following description:

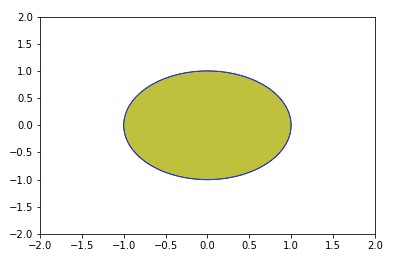
+ Training data (X, Y):

Training data contains N1 = 10,000 points in 2-dimentional space and are followed by the uniform radius between 0 and 2 and its label is 1 is it is inside the yellow circle, otherwise it is 0 + Validation data (X, Y):

Validation data contains N2 = 2,000 points in 2-dimentional space and are followed by the uniform radius between 0 and 2 and its label is 1 is it is inside the yellow circle, otherwise it is 0

+ Testing data (X, Y):

Testing data contains N2 = 2,000 points in 2-dimentional space and are followed by the uniform radius between 0 and 2 and its label is 1 is it is inside the yellow circle, otherwise it is 0



Assume that we use CrossEntropyLoss (nn. CrossEntropyLoss) and

GradientDescent(torch.optim.SGD) with lr = 0.01

Report the training loss, validation loss, testing accuracy (in number and visualized by figure) in the following case

+ Train the MLP with 10 iterations

+ Train the MLP with 100 iterations

+ Train the MLP with 1000 iterations

Note: use import **matplotlib.pyplot** to plot figures

**Solution:**

***After training with 10 iterations:***

|  |  |  |
| --- | --- | --- |
| **Parameters** | **Value** | **Plot** |
| Training loss | 0.69662243 |  |
| Validation loss | 0.6973139 |
| Testing accuracy | 0.4485 |

***After training with 100 iterations:***

|  |  |  |
| --- | --- | --- |
| **Parameters** | **Value** | **Plot** |
| Training loss | 0.702289 |  |
| Validation loss | 0.7016716 |
| Testing accuracy | 0.4525 |

***After training with 1000 iterations:***

|  |  |  |
| --- | --- | --- |
| **Parameters** | **Value** | **Plot** |
| Training loss | 0.6067765 |  |
| Validation loss | .60660905 |
| Testing accuracy | 0.8995 |

**Code:**

import matplotlib.pyplot as plt

import matplotlib

import numpy as np

import torch.utils.data as data\_utils

import torch

from torch import nn

def sample\_points(n):

# returns (X,Y), where X of shape (n,2) is the numpy array of points and Y is the array of classes

radius = np.random.uniform(low=0,high=2,size=n).reshape(-1,1) # uniform radius between 0 and 2

angle = np.random.uniform(low=0,high=2\*np.pi,size=n).reshape(-1,1) # uniform angle

x1 = radius\*np.cos(angle)

x2=radius\*np.sin(angle)

y = (radius<1).astype(int).reshape(-1)

x = np.concatenate([x1,x2],axis=1)

return x,y

def testing\_routine(net,dataset):

# Now for the validation set

test\_data,test\_labels=dataset

test\_output = net(test\_data)

# compute the accuracy of the prediction

test\_prediction = test\_output.cpu().detach().argmax(dim=1)

test\_accuracy = (test\_prediction.numpy()==test\_labels.numpy()).mean()

print("Testing accuracy :",test\_accuracy)

plot\_points([test\_data,test\_prediction])

# Define training process

def training\_routine(net,dataset,n\_iters,gpu):

# organize the data

train\_data,train\_labels,val\_data,val\_labels = dataset

#train,valiadation=dataset

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.SGD(net.parameters(),lr=0.01)

# use the flag

if gpu:

train\_data,train\_labels = train\_data.cuda(),train\_labels.cuda()

val\_data,val\_labels = val\_data.cuda(),val\_labels.cuda()

net = net.cuda() # the network parameters also need to be on the gpu !

print("Using GPU")

else:

print("Using CPU")

for i in range(n\_iters):

# forward pass

train\_output = net(train\_data)

train\_loss = criterion(train\_output,train\_labels)

# backward pass and optimization

train\_loss.backward()

optimizer.step()

optimizer.zero\_grad()

# Once every 100 iterations, print statistics

if i%100==0:

print("At iteration",i)

# compute the accuracy of the prediction

train\_prediction = train\_output.cpu().detach().argmax(dim=1)

train\_accuracy = (train\_prediction.numpy()==train\_labels.numpy()).mean()

# Now for the validation set

val\_output = net(val\_data)

val\_loss = criterion(val\_output,val\_labels)

# compute the accuracy of the prediction

val\_prediction = val\_output.cpu().detach().argmax(dim=1)

val\_accuracy = (val\_prediction.numpy()==val\_labels.numpy()).mean()

print("Training loss :",train\_loss.cpu().detach().numpy())

print("Training accuracy :",train\_accuracy)

print("Validation loss :",val\_loss.cpu().detach().numpy())

print("Validation accuracy :",val\_accuracy)

def generate\_dataset(training\_points\_num,validation\_points\_num,testing\_points\_num):

# generating dataset

train\_data,train\_labels= sample\_points(training\_points\_num)

train\_data=torch.from\_numpy(train\_data).float()

train\_labels=torch.from\_numpy(train\_labels)

val\_data,val\_labels= sample\_points(validation\_points\_num)

val\_data=torch.from\_numpy(val\_data).float()

val\_labels=torch.from\_numpy(val\_labels)

testing\_data,testing\_labels=sample\_points(testing\_points\_num)

testing\_data=torch.from\_numpy(testing\_data).float()

testing\_labels=torch.from\_numpy(testing\_labels)

#generating testing dataset

dataset=[train\_data,train\_labels,val\_data,val\_labels]

testing\_dataset=[testing\_data,testing\_labels]

return dataset,testing\_dataset

class Net(torch.nn.Module):

def \_\_init\_\_(self, input\_size, hidden\_size):

super(Net, self).\_\_init\_\_()

self.input\_size = input\_size

self.hidden\_size = hidden\_size

self.fc1 = torch.nn.Linear(self.input\_size, self.hidden\_size)

self.relu = torch.nn.ReLU()

self.fc2 = torch.nn.Linear(self.hidden\_size, 2)

self.sigmoid = torch.nn.Sigmoid()

def plot\_points(points):

fig, ax = plt.subplots()

ax.set\_xlim((-2, 2))

ax.set\_ylim((-2, 2))

colors=['red','green']

X,Y=points[0],points[1]

circle1 = plt.Circle((0, 0), 1, color='yellow',zorder=1)

ax.add\_artist(circle1)

for i in range(len(X)):

x,y=X[i]

x=float(x)

y=float(y)

color=int(Y[i])

circle=matplotlib.patches.Circle(xy=(x,y),radius=0.02,color=colors[color],zorder=2)

ax.add\_patch(circle)

plt.show()

dataset,testing\_dataset=generate\_dataset(10000,2000,2000)

net=Net(2,128)

n\_iters=1000

gpu=False

# call training routine

training\_routine(net,dataset,n\_iters,gpu)

testing\_routine(net,testing\_dataset)