**Introduction to Machine Learning (Spring 2019)**

**Homework #4 (50 Pts, May 22)**

**Student ID 2014310279**

**Name 김 승현**

**Instruction:** We provide all codes and datasets in Python. Please write your code to complete Perceptron & MLP. Compress ‘Answer.py’ & your report ONLY and submit with the filename ‘HW2\_STUDENT\_ID.zip’.

1. **[30 pts]** Implement Perceptron & MLP in ‘Answer.py’.
2. **[Perceptron, 10 pts]** Implement sign function and perceptron in ‘Answer.py’ (‘sign’, ‘Perceptron’).

**Answer: Fill your code here. You also have to submit your code to i-campus.**

*# ============================ Perceptron ==============================*

def sign(z):

sign\_z = None

*# =============== EDIT HERE ===============*

sign\_z = np.where(z > 0 , 1, -1)

*# =========================================*

*return* sign\_z

class Perceptron:

def \_\_init\_\_(self, num\_features):

*# NOTE : In this assignment, weight and bias are separated. Be careful.*

*self*.W = np.random.rand(num\_features, 1)

*self*.b = np.random.rand(1)

def forward(self, x):

out = None

*if* len(x.shape) < 2:

x = np.expand\_dims(x, 0)

*# =============== EDIT HERE ===============*

out = sign(np.dot(x, *self*.W) + *self*.b)

*# =========================================*

*return* out

def stochastic\_train(self, x, y, learning\_rate):

num\_data = x.shape[0]

*while* True:

*# Repeat until quit condition is satisfied.*

quit = True

*for* i in range(num\_data):

*# =============== EDIT HERE ===============*

predicted\_y = int(*self*.forward(x[i]))

*if*(predicted\_y != y[i]):

quit = False

*for* j in range(x.shape[1]):

*self*.W[j] += learning\_rate\*y[i]\*(x[i][j])

*self*.b += learning\_rate\*y[i]

*# =========================================*

*if* quit:

*break*

def batch\_train(self, x, y, learning\_rate):

num\_data = x.shape[0]

*while* True:

*# gradients of W & b*

dW = np.zeros\_like(*self*.W)

db = np.zeros\_like(*self*.b)

*# Repeat until quit condition is satisfied.*

quit = True

*for* i in range(num\_data):

*# =============== EDIT HERE ===============*

predicted\_y = int(*self*.forward(x[i]))

*if*(predicted\_y != y[i]):

quit = False

*for* j in range(x.shape[1]):

dW[j] += y[i]\*x[i][j]

db += y[i]

*self*.W += learning\_rate\* dW

*self*.b += learning\_rate\* db

*# =========================================*

1. **[MLP, 20 pts]** Implement activation functions and MLP layers in ‘Answer.py’ (‘Sigmoid’, ‘ReLU’, ‘Input/Hidden/(Sigmoid, Softmax) Output Layers’).

**Answer: Fill your code here. You also have to submit your code to i-campus.**

*# ====================== MultiLayer Perceptron =========================*

class ReLU:

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

*# 1 (True) if ReLU input < 0*

*self*.zero\_mask = None

def forward(self, z):

out = None

*# =============== EDIT HERE ===============*

*self*.zero\_mask = (z < 0)

out = z.copy()

out[*self*.zero\_mask] = 0

*# =========================================*

*return* out

def backward(self, d\_prev):

dz = None

*# =============== EDIT HERE ===============*

d\_prev[*self*.zero\_mask] = 0

dz = d\_prev

*# =========================================*

*return* dz

class Sigmoid:

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

*self*.out = None

def forward(self, z):

*self*.out = None

*# =============== EDIT HERE ===============*

*self*.out = 1 / (1 + np.exp(-z))

*# =========================================*

*return* *self*.out

def backward(self, d\_prev):

dz = None

*# =============== EDIT HERE ===============*

dz = d\_prev \* ( 1.0 - *self*.out) \* *self*.out

*# =========================================*

*return* dz

class InputLayer:

def \_\_init\_\_(self, num\_features, num\_hidden\_1, activation):

*# Weights and bias*

*self*.W = np.random.rand(num\_features, num\_hidden\_1)

*self*.b = np.zeros(num\_hidden\_1)

*# Gradient of Weights and bias*

*self*.dW = None

*self*.db = None

*# Forward input*

*self*.x = None

*# Activation function (Sigmoid or ReLU)*

*self*.act = activation()

def forward(self, x):

*self*.x = None

*self*.out = None

*# =============== EDIT HERE ===============*

*self*.x = x

out = np.dot(*self*.x, *self*.W) + *self*.b

*self*.out = *self*.act.forward(out)

*# =========================================*

*return* *self*.out

def backward(self, d\_prev):

*self*.dW = None

*self*.db = None

*# =============== EDIT HERE ===============*

dz = *self*.act.backward(d\_prev)

*self*.dW = np.dot(np.transpose(*self*.x), dz)

*self*.db = np.sum(dz, axis=0)

*# =========================================*

class SigmoidOutputLayer:

def \_\_init\_\_(self, num\_hidden\_2, num\_outputs):

*# Weights and bias*

*self*.W = np.random.rand(num\_hidden\_2, num\_outputs)

*self*.b = np.zeros(num\_outputs)

*# Gradient of Weights and bias*

*self*.dW = None

*self*.db = None

*# Input (x), label(y), prediction(y\_hat)*

*self*.x = None

*self*.y = None

*self*.y\_hat = None

*# Loss*

*self*.loss = None

*# Sigmoid function*

*self*.sigmoid = Sigmoid()

def forward(self, x, y):

*self*.y\_hat = *self*.predict(x)

*self*.y = y

*self*.x = x

*self*.loss = *self*.binary\_ce\_loss(*self*.y\_hat, *self*.y)

*return* *self*.loss

def binary\_ce\_loss(self, y\_hat, y):

eps = 1e-10

bce\_loss = None

*# =============== EDIT HERE ===============*

batch\_size = y.shape[0]

cost = -np.sum(y\*np.log(y\_hat+eps) + (1-y)\*np.log(1+eps-y\_hat))

bce\_loss = cost / batch\_size

*# =========================================*

*return* bce\_loss

def predict(self, x):

y\_hat = None

*# =============== EDIT HERE ===============*

z = np.dot(x, *self*.W) + *self*.b

y\_hat = *self*.sigmoid.forward(z)

*# =========================================*

*return* y\_hat

def backward(self, d\_prev=1):

batch\_size = *self*.y.shape[0]

dx = None

*# =============== EDIT HERE ===============*

dy\_hat = *self*.y\_hat - *self*.y

dz = dy\_hat \* d\_prev

dx = np.dot(dz,(*self*.W).T) / batch\_size

*self*.dW = np.dot(np.transpose(*self*.x), dz) / batch\_size

*self*.db = np.sum(dz, axis=0) / batch\_size

*# =========================================*

*return* dx

class HiddenLayer:

def \_\_init\_\_(self, num\_hidden\_1, num\_hidden\_2):

*# Weights and bias*

*self*.W = np.random.rand(num\_hidden\_1, num\_hidden\_2)

*self*.b = np.zeros(num\_hidden\_2)

*# Gradient of Weights and bias*

*self*.dW = None

*self*.db = None

*# ReLU function*

*self*.act = ReLU()

def forward(self, x):

*self*.x = None

*self*.out = None

*# =============== EDIT HERE ===============*

*self*.x = x

*self*.out = np.dot(x, *self*.W) + *self*.b

*self*.out = *self*.act.forward(*self*.out)

*# =========================================*

*return* *self*.out

def backward(self, d\_prev):

dx = None

*self*.dW = None

*self*.db = None

*# =============== EDIT HERE ===============*

dz = *self*.act.backward(d\_prev)

*self*.dW = np.dot((*self*.x).T, dz)

*self*.db = np.sum(dz, axis=0)

dx = np.dot(dz, (*self*.W).T)

*# =========================================*

*return* dx

class SoftmaxOutputLayer:

def \_\_init\_\_(self, num\_hidden\_2, num\_outputs):

*# Weights and bias*

*self*.W = np.random.rand(num\_hidden\_2, num\_outputs)

*self*.b = np.zeros(num\_outputs)

*# Gradient of Weights and bias*

*self*.dW = None

*self*.db = None

*# Input (x), label(y), prediction(y\_hat)*

*self*.x = None

*self*.y = None

*self*.y\_hat = None

*# Loss*

*self*.loss = None

def forward(self, x, y):

*self*.y\_hat = *self*.predict(x)

*self*.y = y

*self*.x = x

*self*.loss = *self*.ce\_loss(*self*.y\_hat, *self*.y)

*return* *self*.loss

def ce\_loss(self, y\_hat, y):

eps = 1e-10

ce\_loss = None

*# =============== EDIT HERE ===============*

batch\_size = y.shape[0]

log\_probs = -y \* np.log(y\_hat + eps)

ce\_loss = np.sum(log\_probs) / batch\_size

*# =========================================*

*return* ce\_loss

def predict(self, x):

y\_hat = None

*# =============== EDIT HERE ===============*

z = np.dot(x, *self*.W) + *self*.b

y\_hat = softmax(z)

*# =========================================*

*return* y\_hat

def backward(self, d\_prev=1):

batch\_size = *self*.y.shape[0]

dx = None

*# =============== EDIT HERE ===============*

dProb = (*self*.y\_hat).copy()

dProb[np.arange(batch\_size), np.argmax(*self*.y,axis=1)] -= 1

dProb /= batch\_size

dz = dProb \* d\_prev

*self*.dW = np.dot((*self*.x).T, dz)

*self*.db = np.sum(dz, axis=0)

dx = np.dot(dz, (*self*.W).T)

*# =========================================*

*return* dx

*# ======================================================================*

NOTE: You should write your codes in ‘EDIT HERE’ signs. It is not recommended to edit other parts. Once you complete your implementation, run the check codes (‘PLA\_Checker.py’, ‘‘MLP\_Checker.py’’) to check if it is done correctly.

1. **[20 Pts]** Experiment results
2. **[MLP-1]** Adjust ‘num\_epochs’ and ‘learning\_rate’ and run ‘MLP\_1.py’ to solve XOR problem. Report training accuracy with given code and explain how the MLP solve XOR problem by analyzing values of hidden nodes.

**Answer: Fill your code here. You also have to submit your code to i-campus.**

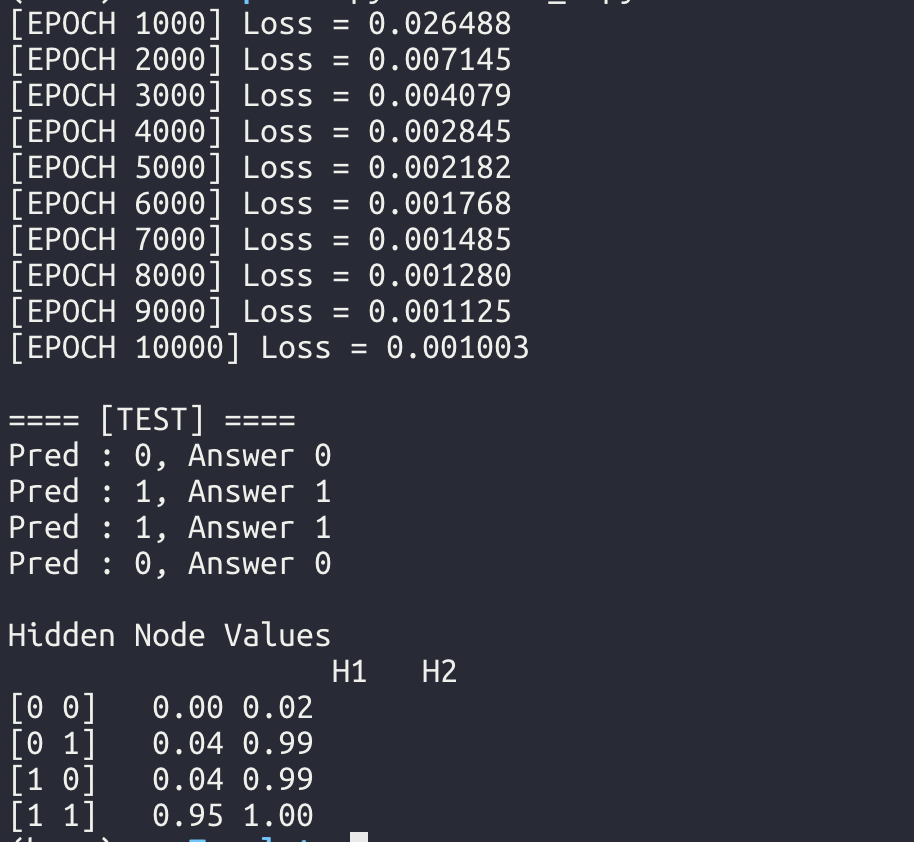
*# =============== EDIT HERE ===============*

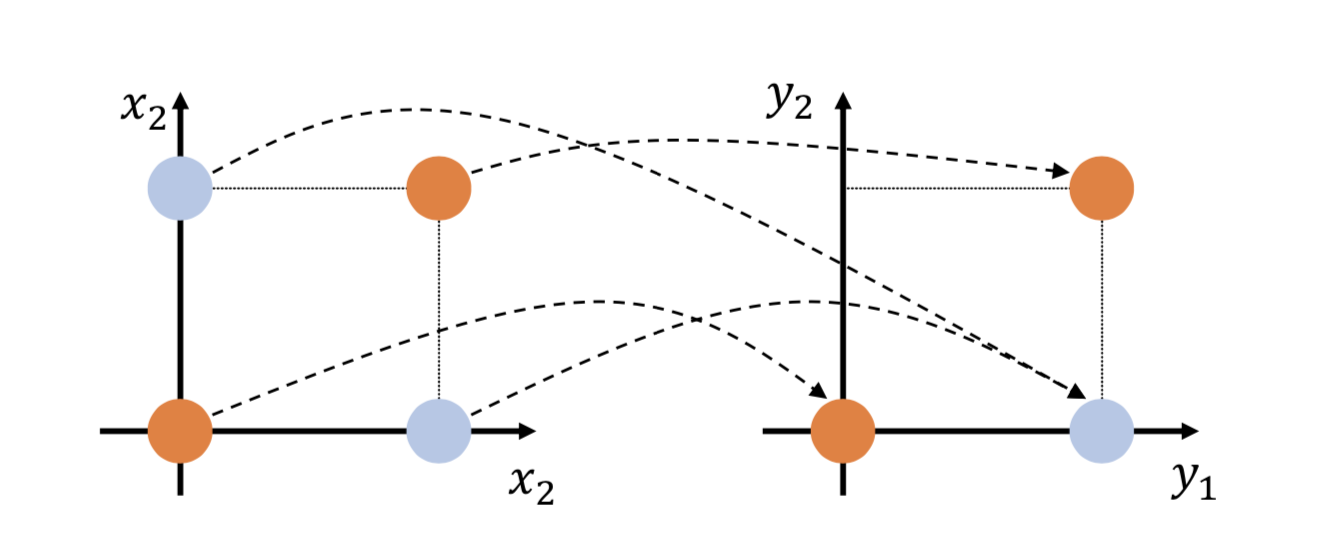
num\_epochs = 10000

learning\_rate = 1

print\_every = 1000

*# =========================================*





위와 같은 원리로 hidden node에서 input [0, 0] 은 [0, 0] input [0, 1], [1,0] 은 [0,1] input [1, 1] 은 [1, 1] 로 변환하여 초기 input을 linearly seperable 하게 변환한다.

1. **[MLP-2]** Adjust hyperparameters and run ‘MLP\_2.py’ on fashion MNIST to get the best results. Report your best results with the hyperparameters. Show the plot of training and test accuracy according to the number of training epochs with the given code and briefly explain the plot. (batch size = 100)

**Answer: Fill the blank in the table. Show the plot of training & test accuracy with a brief explanation.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Hidden 1** | **Hidden 2** | **# of epochs** | **Learning rate** | **Acc.** |
| **10** | 10 | 10000 | 0.0004 | 0.836 |

