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

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

**Student ID 2014313303**

**Name 홍태하**

**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 function for Perceptron

sign(z) = 1 if z > 0, -1 otherwise.

[Inputs]

z : input for sign function in any shape

[Outputs]

sign\_z : sign value for z

"""

sign\_z = None

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

sign\_z = np.zeros(z.shape)

sign\_z[np.where(z>=0)] = 1

sign\_z[np.where(z<0)] = -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):

"""

Forward path of Perceptron (single neuron).

x -- (weight, bias) --> z -- (sign function) --> sign(z)

[Inputs]

x : input for perceptron. Numpy array of (N, D)

[Outputs]

out : output of perceptron. Numpy array of (N, 1)

"""

out = None

if len(x.shape) < 2:

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

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

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

# =========================================

return out

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

num\_data = x.shape[0]

"""

Stochastic Training of perceptron

Perceptron Stochastic Training updates weights on every data (not batch)

Training ends when there is no misclassified data.

See Lecture Notes 'W09 Neural network basics'

Again, Do not implement funtionalities such as shuffling data or sth.

[Inputs]

x : input for perceptron. Numpy array of (N, D)

y : label of data x. Numpy array of (N, )

learning\_rate : learning rate.

[Outputs]

None

"""

while True:

# Repeat until quit condition is satisfied.

quit = True

for i in range(num\_data):

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

num\_features = x.shape[1]

y\_hat = self.forward(x)

if y\_hat[i] != y[i]:

quit = False

for j in range(num\_features):

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]

"""

Batch Training of perceptron

Perceptron Batch Training updates weights all at once for every data (not everytime)

Training ends when there is no misclassified data.

See Lecture Notes 'W09 Neural network basics'

Again, Do not implement funtionalities such as shuffling data or sth.

[Inputs]

x : input for perceptron. Numpy array of (N, D)

y : label of data x. Numpy array of (N, )

learning\_rate : learning rate.

[Outputs]

None

"""

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 ===============

num\_features = x.shape[1]

if i == 0:

S = []

y\_hat = self.forward(x)

if y\_hat[i] != y[i]:

quit = False

S.append(i)

if len(S) != 0:

for i, index in enumerate(S):

for j in range(num\_features):

dW[j] = y[index]\*x[index,j]

self.W[j] += learning\_rate\*dW[j]

db = y[index]

self.b = learning\_rate\*db

# =========================================

if quit:

break

# ======================================================================

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 =========================

def softmax(z):

# Numerically stable softmax. Already implemented.

z = z - np.max(z, axis=1, keepdims=True)

\_exp = np.exp(z)

\_sum = np.sum(\_exp, axis=1, keepdims=True)

sm = \_exp / \_sum

return sm

class ReLU:

"""

ReLU Function. ReLU(x) = max(0, x)

Implement forward & backward path of ReLU.

"""

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

# 1 (True) if ReLU input < 0

self.zero\_mask = None

def forward(self, z):

"""

ReLU Forward.

ReLU(x) = max(0, x)

z --> (ReLU) --> out

[Inputs]

z : ReLU input in any shape.

[Outputs]

out : ReLU(z).

"""

out = None

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

out = np.zeros(z.shape)

self.zero\_mask = np.zeros(z.shape)

index = np.where(z>=0)

out[index] = z[index]

self.zero\_mask[np.where(z<0)] = 1

# =========================================

return out

def backward(self, d\_prev):

"""

ReLU Backward.

z --> (ReLU) --> out

dz <-- (dReLU) <-- d\_prev(dL/dout)

[Inputs]

d\_prev : Gradients until now.

d\_prev = dL/dk, where k = ReLU(z).

[Outputs]

dz : Gradients w.r.t. ReLU input z.

"""

dz = None

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

dz = d\_prev\*(1-self.zero\_mask)

# =========================================

return dz

class Sigmoid:

"""

Sigmoid Function.

Implement forward & backward path of Sigmoid.

"""

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

self.out = None

def forward(self, z):

"""

Sigmoid Forward.

z --> (Sigmoid) --> self.out

[Inputs]

z : Sigmoid input in any shape.

[Outputs]

self.out : Sigmoid(z).

"""

self.out = None

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

self.out = np.zeros(z.shape)

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

# =========================================

return self.out

def backward(self, d\_prev):

"""

Sigmoid Backward.

z --> (Sigmoid) --> self.out

dz <-- (dSigmoid) <-- d\_prev(dL/d self.out)

[Inputs]

d\_prev : Gradients until now.

[Outputs]

dz : Gradients w.r.t. Sigmoid input z.

"""

dz = None

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

dz = d\_prev\*self.out\*(1-self.out)

# =========================================

return dz

"""

\*\* Multilayer Perceptron \*\*

[input -- (input layer) --> hidden1 -- (hidden layer) --> hidden -- (output layer) --> output]

You will implement input/hiddden/output layers.

TIP : Draw computational graph and compute gradients to code BEFOREHAND.

"""

class InputLayer:

"""

Input Layer

input -- (input layer) --> hidden

Implement forward & backward path.

"""

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):

"""

Input layer forward

- Feed forward

- Apply activation function you implemented above.

[Inputs]

x : Input data (N, D)

[Outputs]

self.out : Output of input layer. Hidden. (N, H)

"""

self.x = None

self.out = None

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

self.x = x

self.out = self.act.forward(self.x.dot(self.W) + self.b)

# =========================================

return self.out

def backward(self, d\_prev):

"""

Input layer backward

x and (W & b) --> z -- (activation) --> hidden

dx and (dW & db) <-- dz <-- (activation) <-- hidden

- Backward of activation

- Gradients of W, b

[Inputs]

d\_prev : Gradients until now.

[Outputs]

None

"""

self.dW = None

self.db = None

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

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

self.dW = self.x.T.dot(dz)

self.db = dz.sum(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):

"""

Sigmoid output layer forward

- Make prediction

- Calculate loss

## Already Implemented ##

"""

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):

"""

Calcualte "Binary cross-entropy loss"

Add 'eps' for stability inside log function.

[Inputs]

y\_hat : Prediction

y : Label

[Outputs]

loss value

"""

eps = 1e-10

bce\_loss = None

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

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

bce\_loss /= len(y)

# =========================================

return bce\_loss

def predict(self, x):

"""

Make prediction in probability. (Not 0 or 1 label!!)

[Inputs]

x : input data

[Outputs]

y\_hat : Prediction

"""

y\_hat = None

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

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

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

# =========================================

return y\_hat

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

"""

Calculate gradients of input (x), W, b of this layer.

Save self.dW, self.db to update later.

x and (W & b) --> z -- (activation) --> y\_hat --> Loss

dx and (dW & db) <-- dz <-- (activation) <-- dy\_hat <-- Loss

[Inputs]

d\_prev : Gradients until here. (Always 1 since its output layer)

[Outputs]

dx : Gradients of output layer input x (Not MLP input x!)

"""

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

dx = None

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

temp = (self.y\_hat - self.y)/len(self.y)

dx = temp.dot(self.W.T)

self.dW = self.x.T.dot(temp)

self.db = temp.sum(axis=0)

# =========================================

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):

"""

Hidden layer forward

- Feed forward

- Apply activation function you implemented above.

[Inputs]

x : Input data (N, D)

[Outputs]

self.out : Output of input layer. Hidden. (N, H)

"""

self.x = None

self.out = None

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

self.x = x

self.out = self.act.forward(self.x.dot(self.W) + self.b)

# =========================================

return self.out

def backward(self, d\_prev):

"""

Hidden layer backward

x and (W & b) --> z -- (activation) --> output

dx and (dW & db) <-- dz <-- (activation) <-- output

- Calculate gradients of input (x), W, b of this layer.

- Save self.dW, self.db to update later.

[Inputs]

d\_prev : Gradients until here.

[Outputs]

dx : Gradients of output layer input x (Not MLP input x!)

"""

dx = None

self.dW = None

self.db = None

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

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

self.dW = self.x.T.dot(dz)

self.db = dz.sum(axis=0)

dx = dz.dot(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):

"""

Softmax output layer forward

- Make prediction

- Calculate loss

## Already Implemented ##

"""

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):

"""

Calcualte "Cross-entropy loss"

Add 'eps' for stability inside log function.

[Inputs]

y\_hat : Prediction

y : Label

[Outputs]

loss value

"""

eps = 1e-10

ce\_loss = None

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

ce\_loss = -np.sum(y\*np.log(y\_hat + eps))

ce\_loss /= len(y)

# =========================================

return ce\_loss

def predict(self, x):

"""

Make prediction in probability. (Not 0, 1, 2 ... label!!)

# Use softmax function above.

[Inputs]

x : input data

[Outputs]

y\_hat : Prediction

"""

y\_hat = None

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

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

y\_hat = softmax(z)

# =========================================

return y\_hat

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

"""

Calculate gradients of input (x), W, b of this layer.

Save self.dW, self.db to update later.

x and (W & b) --> z -- (activation) --> y\_hat --> Loss

dx and (dW & db) <-- dz <-- (activation) <-- dy\_hat <-- Loss

[Inputs]

d\_prev : Gradients until here. (Always 1 since its output layer)

[Outputs]

dx : Gradients of output layer input x (Not MLP input x!)

"""

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

dx = None

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

temp = (self.y\_hat - self.y)/len(self.y)

dx = temp.dot(self.W.T)

self.dW = self.x.T.dot(temp)

self.db = temp.sum(axis=0)

# =========================================

return dx

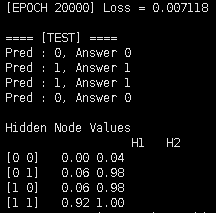
# ======================================================================

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.

Num\_epochs = 20000

Learning\_rate = 0.1



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

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 | 5000 | 0.0005 | Train : 0.794  Test : 0.805 |

Hidden 1과 Hidden 2를 모두 10으로 했을 때 epoch이 1000정도까지 Train & Test Accuracy가 급격히 오르다가 epoch이 1000넘어간 후부터는 완만하게 증가한다. 그렇게 하여 epoch 5000에 도달 했을 때 Train & Test Accuracy는 0.8 정도가 된다.

