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’.

(a) [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.

```python
# ============================ 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
```

```python
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
        # =========================================
```

(b) [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.

```python
# ====================== 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):
```

```python
        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)

        # =========================================
```

```python
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)
        # =========================================
```

```python
        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)
```

```python
        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.

(2) [20 Pts] Experiment results

(a) [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.

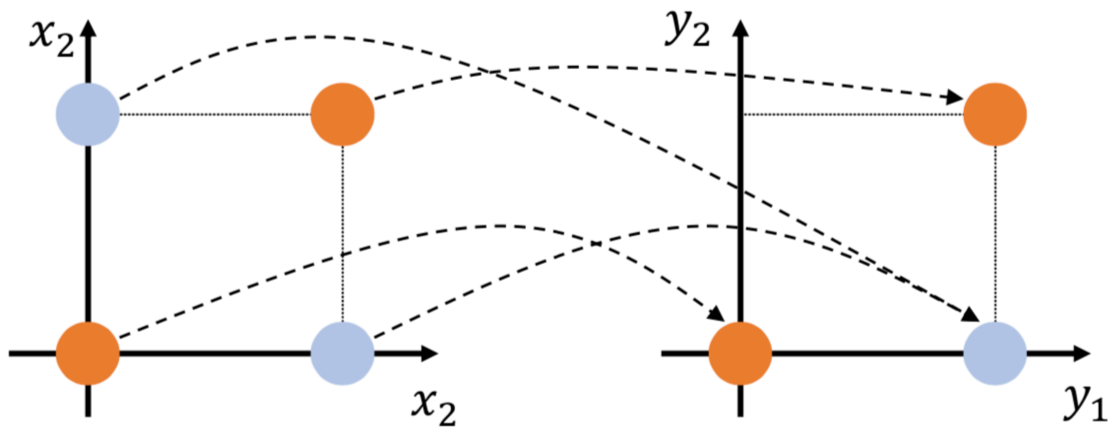
Answer: Fill your code here. You also have to submit your code to i-campus.

```
# =============== EDIT HERE ===============
num_epochs = 10000
learning_rate = 1
print_every = 1000
# =========================================
```

```
[EPOCH 1000] Loss = 0.026488
[EPOCH 2000] Loss = 0.007145
[EPOCH 3000] Loss = 0.004079
[EPOCH 4000] Loss = 0.002845
[EPOCH 5000] Loss = 0.002182
[EPOCH 6000] Loss = 0.001768
[EPOCH 7000] Loss = 0.001485
[EPOCH 8000] Loss = 0.001280
[EPOCH 9000] Loss = 0.001125
[EPOCH 10000] Loss = 0.001003

==== [TEST] ====
Pred : 0, Answer 0
Pred : 1, Answer 1
Pred : 1, Answer 1
Pred : 0, Answer 0

Hidden Node Values
                 H1    H2
[0 0]   0.00 0.02
[0 1]   0.04 0.99
[1 0]   0.04 0.99
[1 1]   0.95 1.00
```



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.

(b) **[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