Introduction to Machine Learning (Spring 2019)

Homework #4 (50 Pts, May 22)

Student ID ______________________

Name ______________________

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').

```python
def sign(z):
    sign_z = None
    # =============== EDIT HERE ===============
    save = z.shape
    sign_z = z.reshape(-1)

    for ind in range(len(sign_z)):
        sign_z[ind] = 1 if sign_z[ind] > 0 else -1

    sign_z = sign_z.reshape(save)
    # =========================================
    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 ===============
        num_data = x.shape[0]
        out = sign(np.matmul(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 ===============
            out = self.forward(x[i])
            if out != y[i]:
                self.W += np.multiply(x[i], y[i] * learning_rate)
                self.b += y[i] * learning_rate
                quit = False
        # =========================================
        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 ===============
        out = self.forward(x[i])
        if out != y[i]:
            dW += np.reshape(x[i] * y[i] * learning_rate, (-1, 1))
            db += y[i] * learning_rate
            quit = False

    self.W += dW
    self.b += db
    # =========================================
    if quit:
        break
```

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

```
class ReLU:

    def __init__(self):
        # 1 (True) if ReLU input < 0
        self.zero_mask = None

    def forward(self, z):
        out = None
        # =============== EDIT HERE ===============
        save = z.shape
        temp = z.reshape(-1)
        mask = np.zeros_like(temp)

        for ind in range(len(temp)):
            mask[ind] = 1 if temp[ind] >= 0 else 0
            temp[ind] = temp[ind] if temp[ind] >= 0 else 0

        out = temp.reshape(save)
        self.zero_mask = mask.reshape(save)
        # =========================================
        return out

    def backward(self, d_prev):
        dz = None
        # =============== EDIT HERE ===============

        dz = np.multiply(d_prev, self.zero_mask)
        # =========================================
        return dz

class Sigmoid:

    def __init__(self):
        self.out = None

    def forward(self, z):
        self.out = None
        # =============== EDIT HERE ===============
        save = z.shape
        temp = z.reshape(-1)

        for ind in range(len(temp)):
            temp[ind] = 1 / (1 + np.exp(-temp[ind]))

        self.out = temp.reshape(save)
        # =========================================
        return self.out

    def backward(self, d_prev):
        dz = None
        # =============== EDIT HERE ===============
        dz = np.multiply(self.out, np.add(1, -self.out))
        dz = np.multiply(dz, d_prev)
        # =========================================
        return dz
```

```python
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
        inner = np.add(np.matmul(self.x, self.W), self.b)

        self.out = self.act.forward(inner)
        # =========================================
        return self.out

    def backward(self, d_prev):
        self.dW = None
        self.db = None
        # =============== EDIT HERE ===============
        act_backward = self.act.backward(d_prev)

        self.dW = np.matmul(np.transpose(self.x), act_backward)
        self.db = np.sum(act_backward, 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
```

```python
def binary_ce_loss(self, y_hat, y):
    eps = 1e-10
    bce_loss = None
    # =============== EDIT HERE ===============
    batch_size = y.shape[0]

    temp_y = y.reshape(-1)
    temp_y_hat = y_hat.reshape(-1)

    bce_loss = 0
    for i in range(len(temp_y)):
        bce_loss -= temp_y[i] * np.log(temp_y_hat[i] + eps) +₩
                (1 - temp_y[i]) * np.log(1 - temp_y_hat[i] + eps)


    bce_loss /= batch_size
    # =========================================
    return bce_loss

def predict(self, x):
    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):
    batch_size = self.y.shape[0]
    dx = None
    # =============== EDIT HERE ===============

    # This equation is derived from the derivative of cross entropy w.r.t y_hat
    # Letting y_hat = c, the gradient is (c - y) / ( (c * (1 - c)) * batch_size )

    y_diff = np.multiply(self.y_hat, 1 - self.y_hat)
    loss_grad = np.divide((self.y_hat - self.y), y_diff * batch_size)
    sig_backward = self.sigmoid.backward(d_prev * loss_grad)

    self.dW = np.matmul(np.transpose(self.x), sig_backward)
    self.db = np.sum(sig_backward, axis=0)

    dx = np.transpose(np.matmul(self.W, np.transpose(sig_backward)))

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

    return dx
```

```python
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
        inner = np.add(np.matmul(self.x, self.W), self.b)

        self.out = self.act.forward(inner)
        # =========================================
        return self.out

    def backward(self, d_prev):
                dx = None
        self.dW = None
        self.db = None
        # =============== EDIT HERE ===============
        act_backward = self.act.backward(d_prev)

        self.dW = np.matmul(np.transpose(self.x), act_backward)
        self.db = np.sum(act_backward, axis=0)

        dx = np.transpose(np.matmul(self.W, np.transpose(act_backward)))

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

```python
    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]

        pred_log = np.log(y_hat + eps)
        ce_loss = np.sum(np.multiply(y, -pred_log))
        ce_loss /= batch_size
        # =========================================
        return ce_loss

    def predict(self, x):
        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):
        batch_size = self.y.shape[0]
        dx = None
        # =============== EDIT HERE ===============

        # Sim as sigmoid
        loss_grad = np.divide(self.y_hat - self.y, batch_size)

        self.dW = np.matmul(np.transpose(self.x), loss_grad)
        self.db = np.sum(loss_grad, axis=0)

        dx = np.transpose(np.matmul(self.W, np.transpose(loss_grad)))

        # =========================================
        return dx

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

(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.

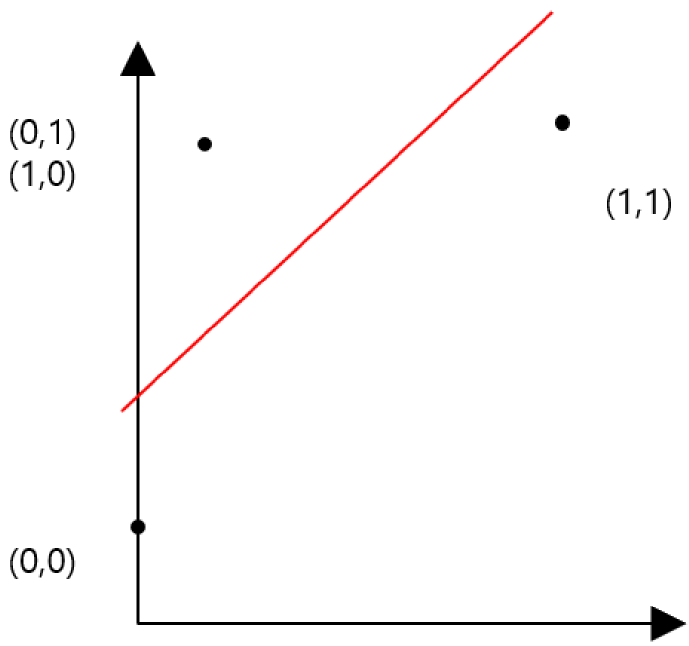
```
# ============== EDIT HERE ==============
num_epochs = 1000
learning_rate = 1
print_every = True
# =======================================
```

Result :

```
==== [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 0.06
[0 1]   0.10 0.98
[1 0]   0.10 0.98
[1 1]   0.88 1.00
```

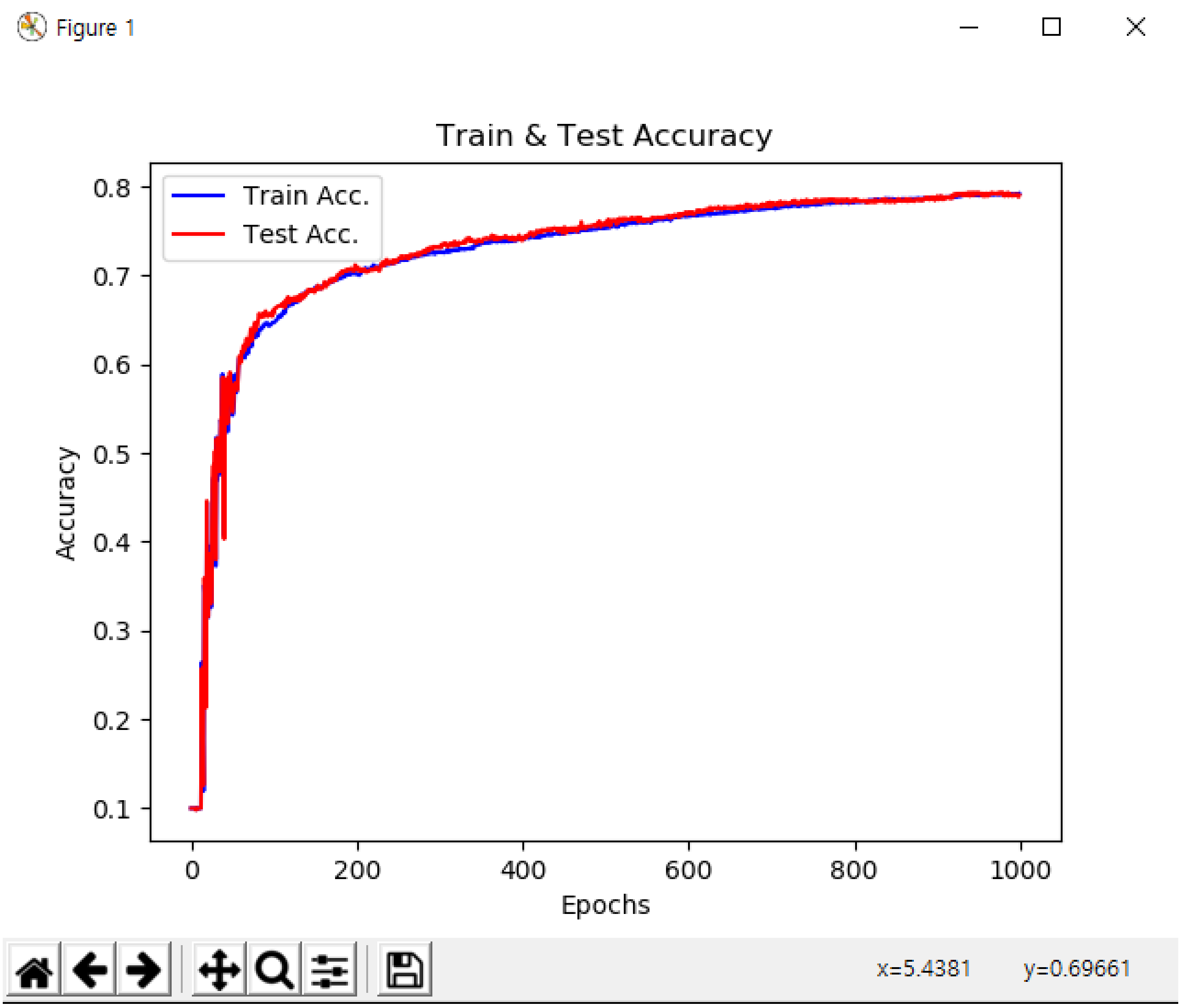
이를 그림으로 나타내면 다음과 같다.



즉, (0,1)과 (1,0)을 같은 hidden node값을 갖게 함으로 인해 다음과 같이 선 하나로 (0,0), (1,1) 과 (0,1), (1,0) 두 영역으로 분리할 수 있도록 구성되었다.

(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)

Hidden 1	**Hidden 2**	**# of epochs**	**Learning rate**	**Acc.**
100	100	1000	0.0001	0.790



초반 약 100 epoch정도에서는 조금 들쑥날쑥한 경향을 보이기는 하지만, accuracy가 급격하게 잘 증가하는 모습을 볼 수 있고, 이후 epoch이 증가함에 따라 Train과 Test accuracy가 증가하지만, 그 속도는 현저하게 낮아진다. 즉, 초반에는 optimal을 찾기 위해 많이 움직인다는 것을 알 수 있고, optimal을 어느 정도 찾은 후에는 안정되게 optimal을 향해 천천히 수렴한다는 것을 알 수 있다. 이는 SGD 외의 다른 Gradient Descent 방법을 사용한다면 더욱 효과적으로 수렴할 것이라 예상한다.

주목할 만한 점은 training과 test accuracy가 모두 증가한다는 것을 알 수 있고, 이는 overfitting등의 문제가 발생하지 않았다는 점을 보여준다.