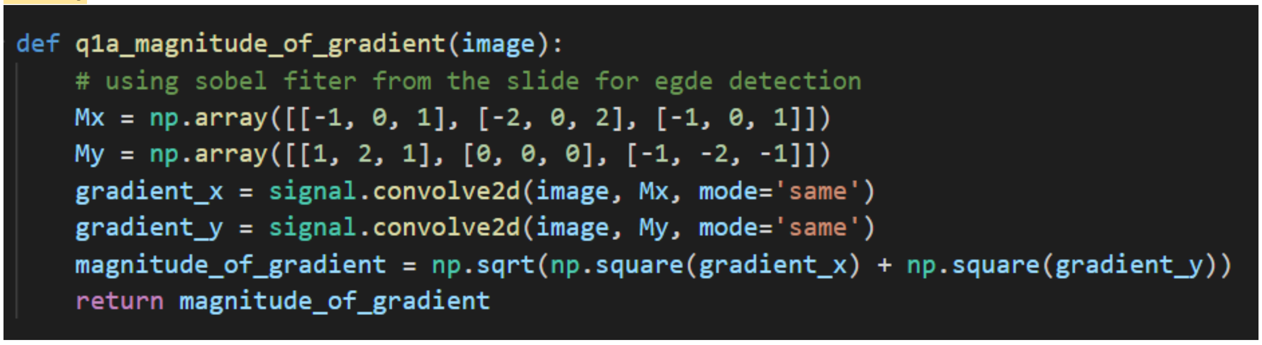
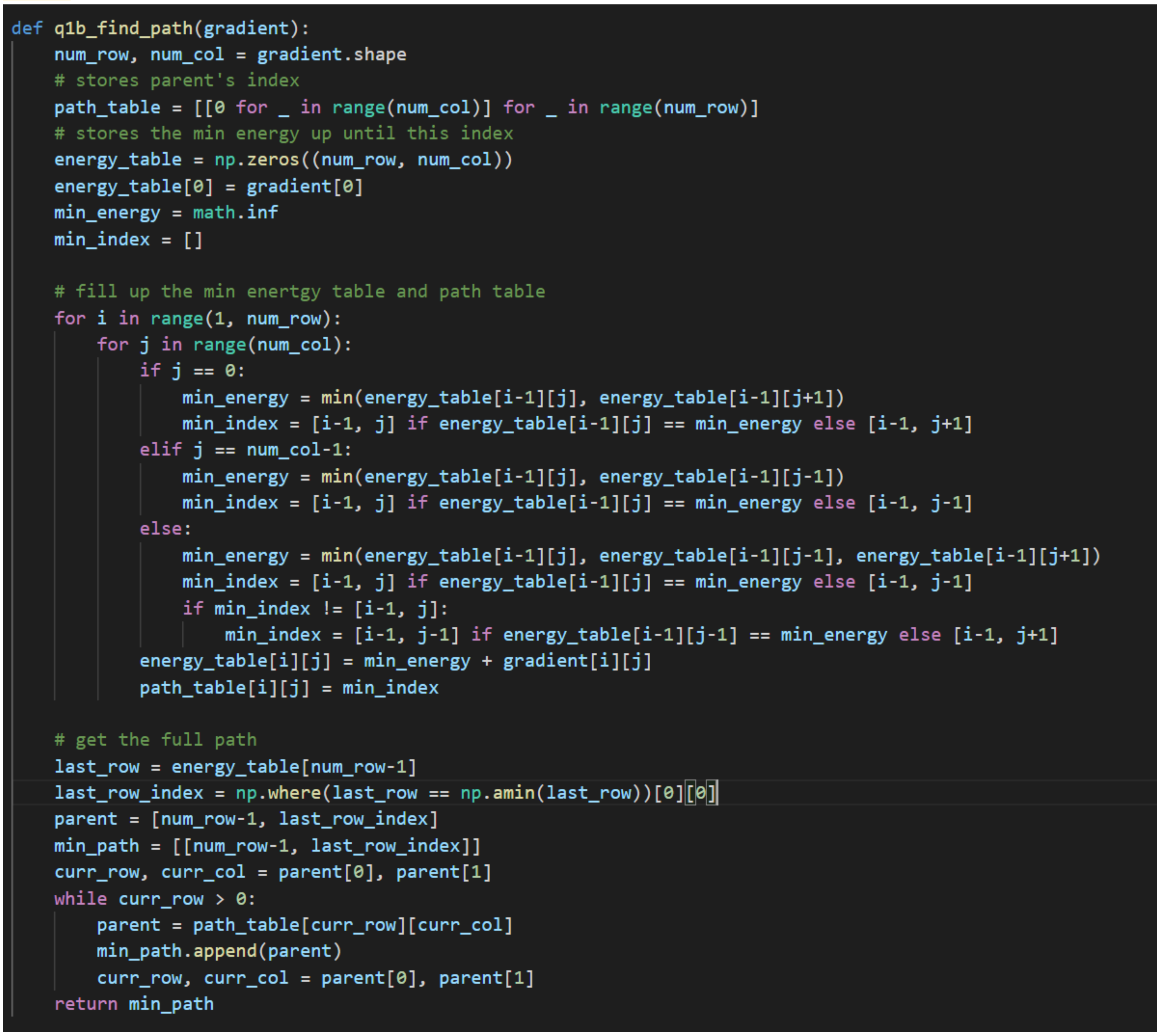
CSC420 Assignment 2
Yu-Chieh Wu 1005473202

Q1 a)



```python
def q1a_magnitude_of_gradient(image):
    # using sobel fiter from the slide for egde detection
    Mx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    My = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])
    gradient_x = signal.convolve2d(image, Mx, mode='same')
    gradient_y = signal.convolve2d(image, My, mode='same')
    magnitude_of_gradient = np.sqrt(np.square(gradient_x) + np.square(gradient_y))
    return magnitude_of_gradient
```

Q1 b)



```python
def q1b_find_path(gradient):
    num_row, num_col = gradient.shape
    # stores parent's index
    path_table = [[0 for _ in range(num_col)] for _ in range(num_row)]
    # stores the min energy up until this index
    energy_table = np.zeros((num_row, num_col))
    energy_table[0] = gradient[0]
    min_energy = math.inf
    min_index = []

    # fill up the min enertgy table and path table
    for i in range(1, num_row):
        for j in range(num_col):
            if j == 0:
                min_energy = min(energy_table[i-1][j], energy_table[i-1][j+1])
                min_index = [i-1, j] if energy_table[i-1][j] == min_energy else [i-1, j+1]
            elif j == num_col-1:
                min_energy = min(energy_table[i-1][j], energy_table[i-1][j-1])
                min_index = [i-1, j] if energy_table[i-1][j] == min_energy else [i-1, j-1]
            else:
                min_energy = min(energy_table[i-1][j], energy_table[i-1][j-1], energy_table[i-1][j+1])
                min_index = [i-1, j] if energy_table[i-1][j] == min_energy else [i-1, j-1]
                if min_index != [i-1, j]:
                    min_index = [i-1, j-1] if energy_table[i-1][j-1] == min_energy else [i-1, j+1]
            energy_table[i][j] = min_energy + gradient[i][j]
            path_table[i][j] = min_index

    # get the full path
    last_row = energy_table[num_row-1]
    last_row_index = np.where(last_row == np.amin(last_row))[0][0]
    parent = [num_row-1, last_row_index]
    min_path = [[num_row-1, last_row_index]]
    curr_row, curr_col = parent[0], parent[1]
    while curr_row > 0:
        parent = path_table[curr_row][curr_col]
        min_path.append(parent)
        curr_row, curr_col = parent[0], parent[1]
    return min_path
```

Q1 c)

```python
def q1c_remove_one_path(img, min_energy_path):
    img = img.tolist()
    for i in range(len(min_energy_path)):
        row, col = min_energy_path[i]
        img[row].pop(col)
    img = np.array(img)
    return img
```

Q1 d)

```python
def q1d_remove_paths(img_path, num_remove):
    img = io.imread(img_path, as_gray=True)
    while num_remove:
        gradient = q1a_magnitude_of_gradient(img)
        path = q1b_find_path(gradient)
        curr_img = q1c_remove_one_path(img, path)
        img = curr_img
        num_remove -= 1
    plt.imshow(img, cmap='gray')
    plt.gray()
    plt.show()
```

Q1 results

Original image 1 – shape: 110x76

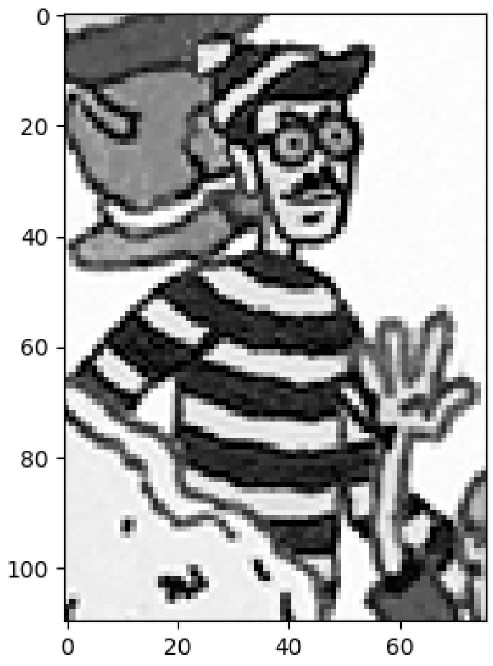
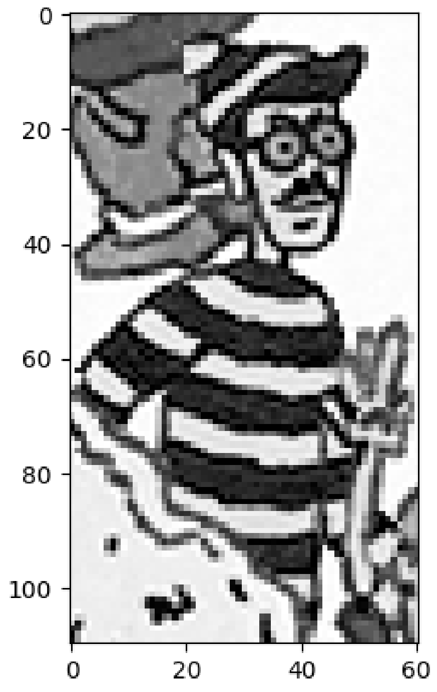


Image 1 Remove 15 paths – shape: 110x61



Original image 2 – shape: 1075x1075

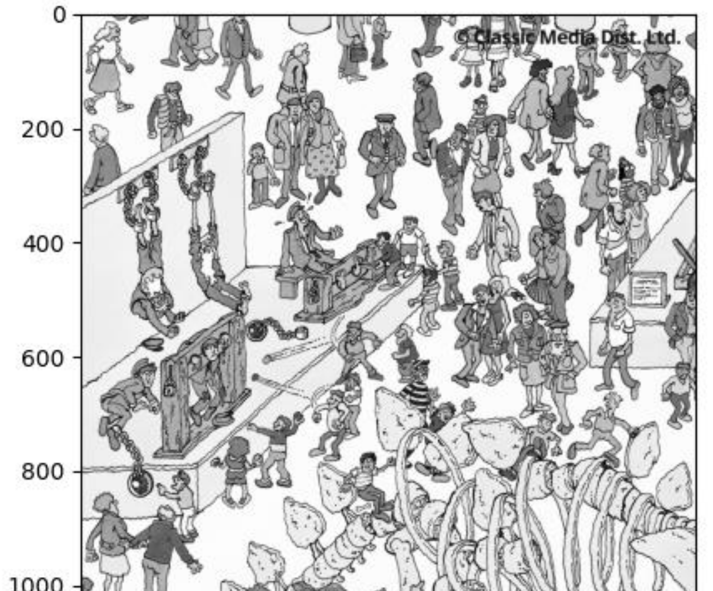
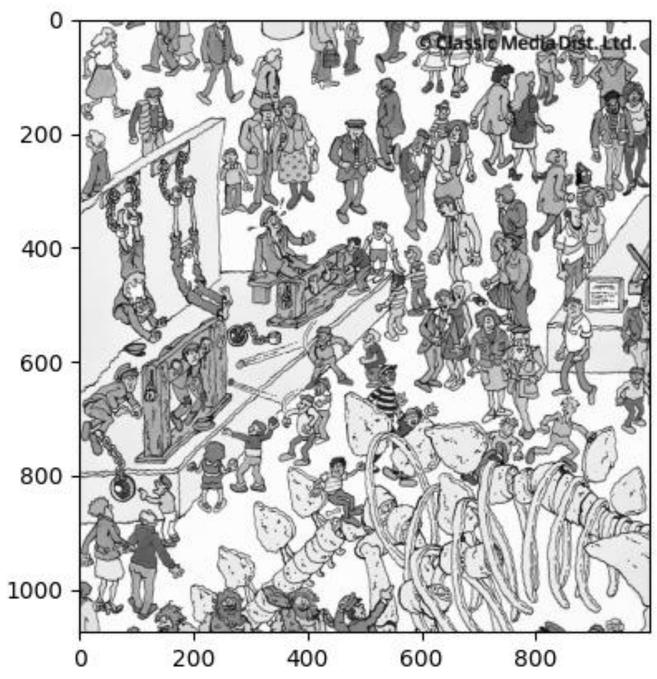


Image 2 Remove 75 paths– shape: 1075x1000



Q2.

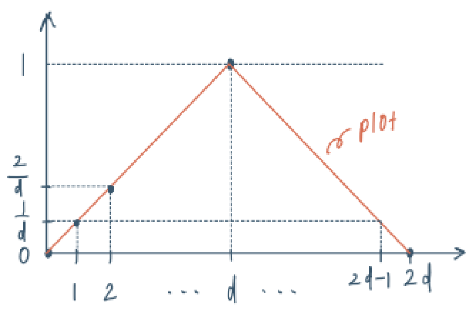
Q2.

Mathematic form of the convolution filter that performs upscaling of a 1D signal by a factor d:

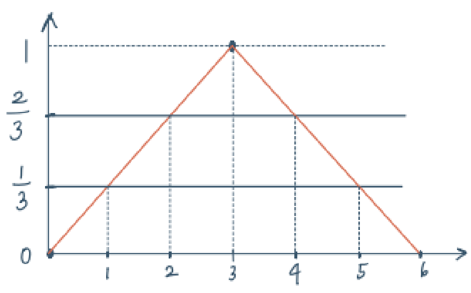
$$h = [0, \frac{1}{d}, \frac{2}{d}, \ldots, \frac{d-1}{d}, 1, \frac{d-1}{d}, \ldots, \frac{2}{d}, \frac{1}{d}, 0]$$

where d is the upsampling factor

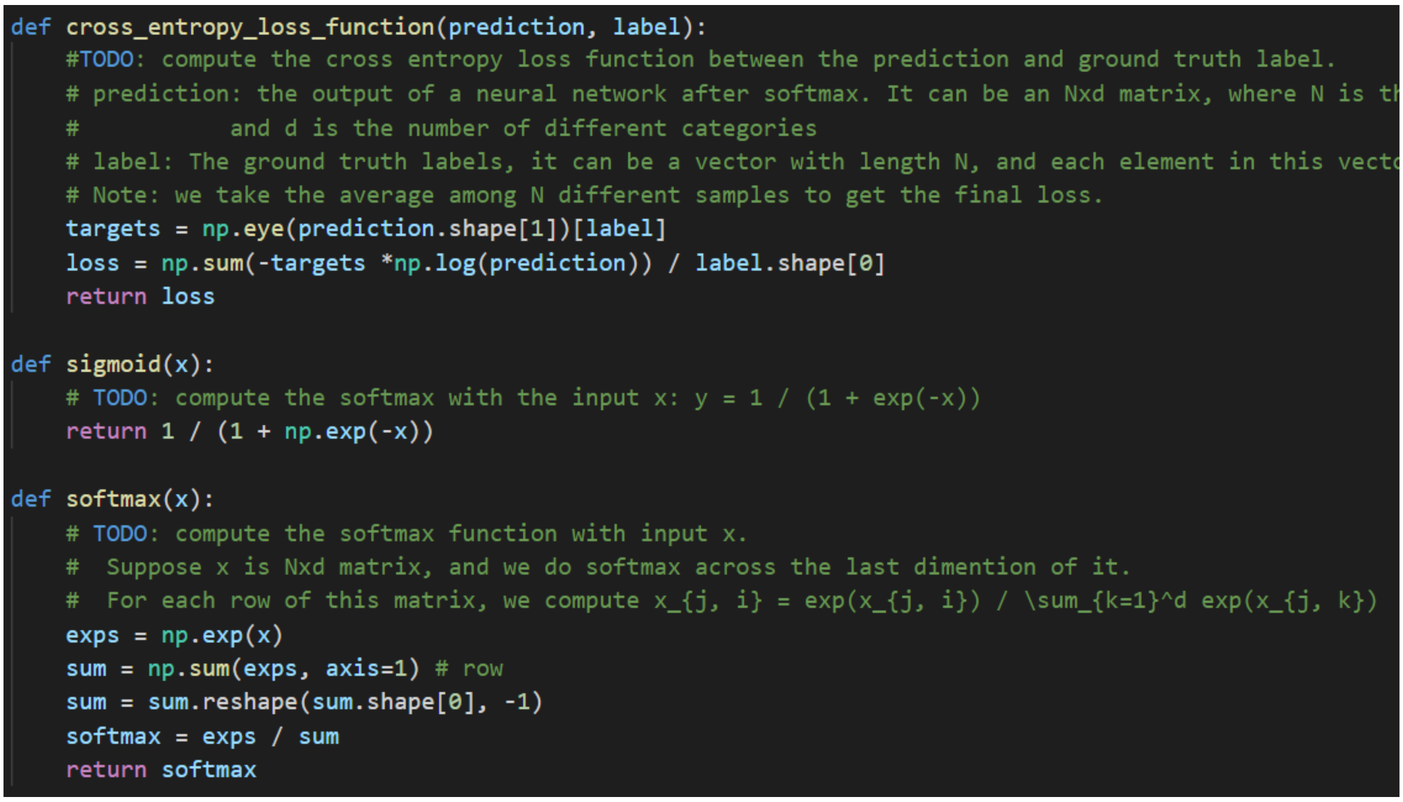
plot (general):



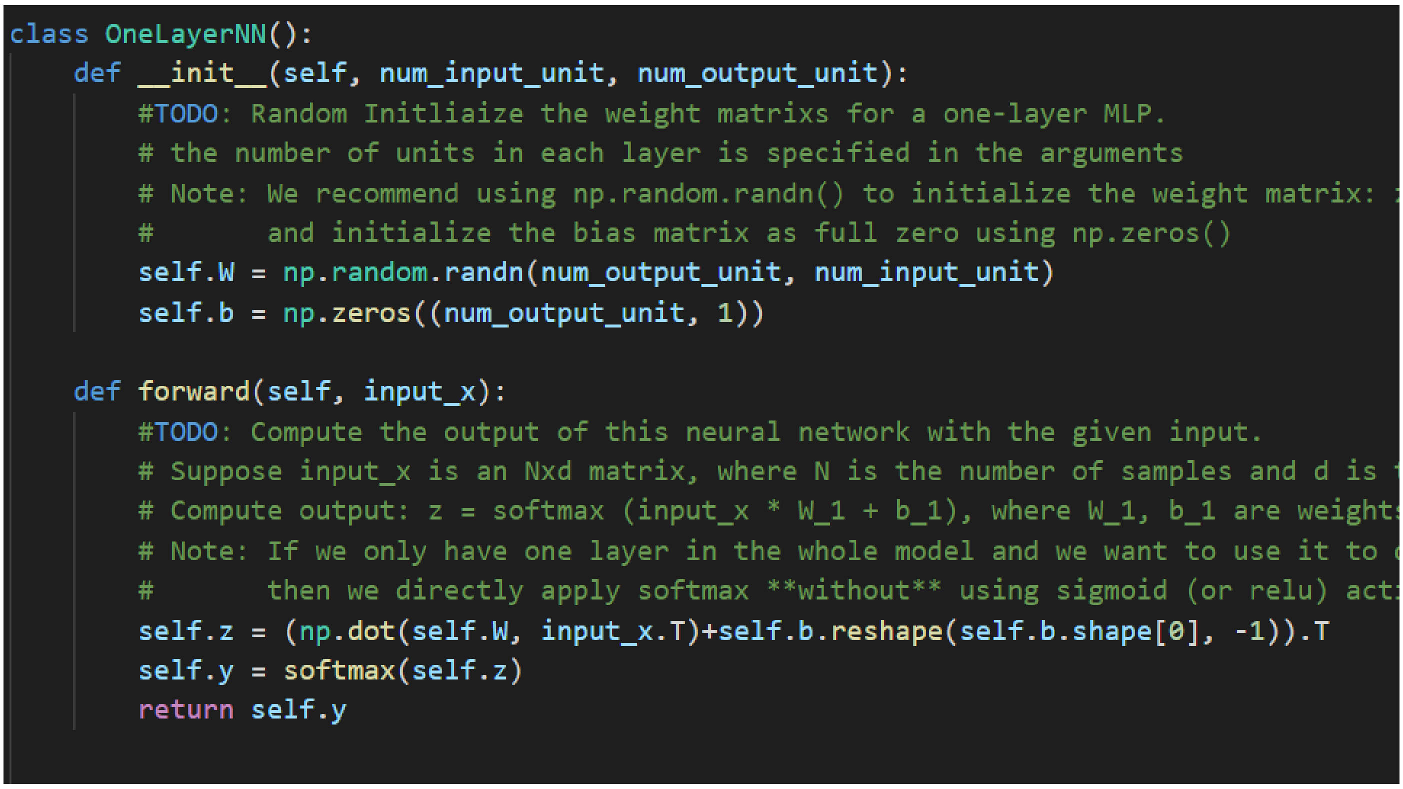
plot: (let d=3)



Q3 a)

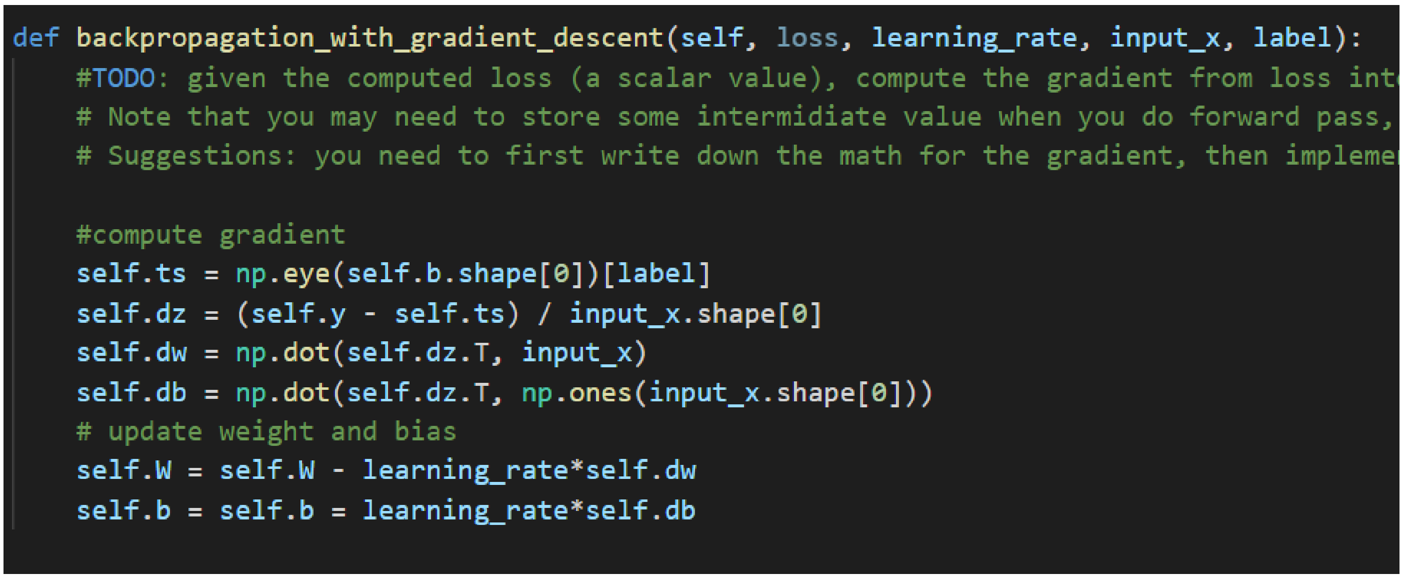


```python
def cross_entropy_loss_function(prediction, label):
    #TODO: compute the cross entropy loss function between the prediction and ground truth label.
    # prediction: the output of a neural network after softmax. It can be an Nxd matrix, where N is th
    #           and d is the number of different categories
    # label: The ground truth labels, it can be a vector with length N, and each element in this vecto
    # Note: we take the average among N different samples to get the final loss.
    targets = np.eye(prediction.shape[1])[label]
    loss = np.sum(-targets *np.log(prediction)) / label.shape[0]
    return loss

def sigmoid(x):
    # TODO: compute the softmax with the input x: y = 1 / (1 + exp(-x))
    return 1 / (1 + np.exp(-x))

def softmax(x):
    # TODO: compute the softmax function with input x.
    #  Suppose x is Nxd matrix, and we do softmax across the last dimention of it.
    #  For each row of this matrix, we compute x_{j, i} = exp(x_{j, i}) / \sum_{k=1}^d exp(x_{j, k})
    exps = np.exp(x)
    sum = np.sum(exps, axis=1) # row
    sum = sum.reshape(sum.shape[0], -1)
    softmax = exps / sum
    return softmax
```

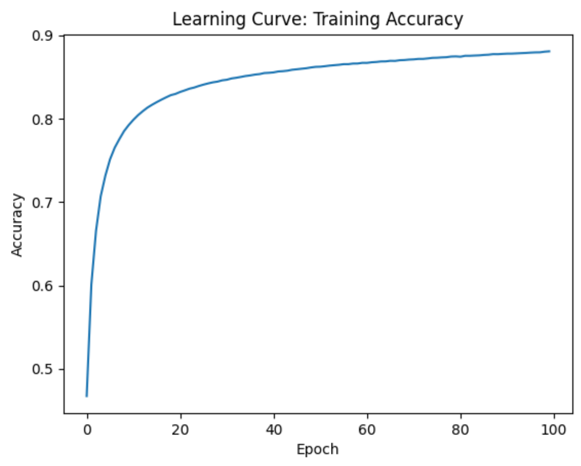


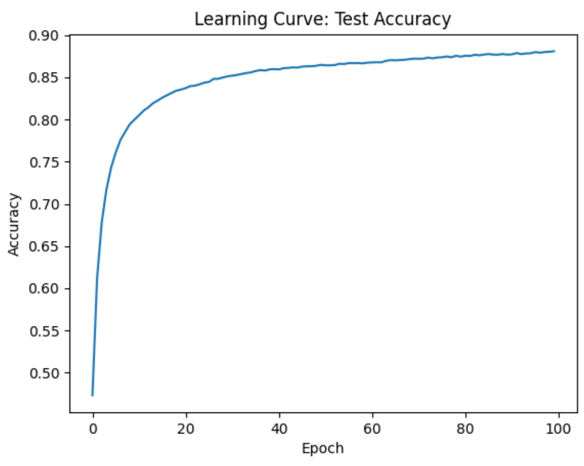
```python
class OneLayerNN():
    def __init__(self, num_input_unit, num_output_unit):
        #TODO: Random Initliaize the weight matrixs for a one-layer MLP.
        # the number of units in each layer is specified in the arguments
        # Note: We recommend using np.random.randn() to initialize the weight matrix: 
        #       and initialize the bias matrix as full zero using np.zeros()
        self.W = np.random.randn(num_output_unit, num_input_unit)
        self.b = np.zeros((num_output_unit, 1))

    def forward(self, input_x):
        #TODO: Compute the output of this neural network with the given input.
        # Suppose input_x is an Nxd matrix, where N is the number of samples and d is 
        # Compute output: z = softmax (input_x * W_1 + b_1), where W_1, b_1 are weight
        # Note: If we only have one layer in the whole model and we want to use it to 
        #       then we directly apply softmax **without** using sigmoid (or relu) act
        self.z = (np.dot(self.W, input_x.T)+self.b.reshape(self.b.shape[0], -1)).T
        self.y = softmax(self.z)
        return self.y
```

Q3 b)

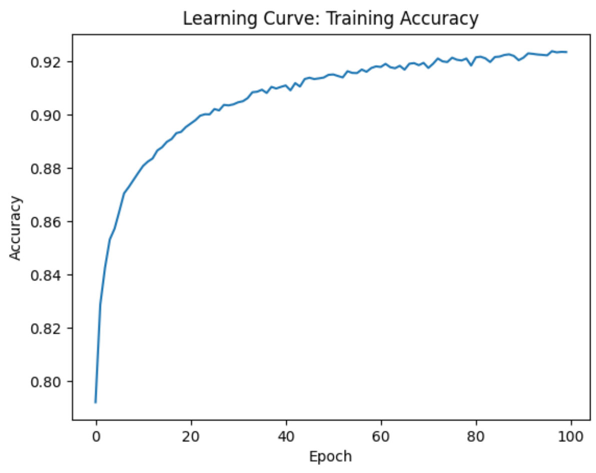

```
def backpropagation_with_gradient_descent(self, loss, learning_rate, input_x, label):
    #TODO: given the computed loss (a scalar value), compute the gradient from loss int
    # Note that you may need to store some intermidiate value when you do forward pass,
    # Suggestions: you need to first write down the math for the gradient, then impleme

    #compute gradient
    self.ts = np.eye(self.b.shape[0])[label]
    self.dz = (self.y - self.ts) / input_x.shape[0]
    self.dw = np.dot(self.dz.T, input_x)
    self.db = np.dot(self.dz.T, np.ones(input_x.shape[0]))
    # update weight and bias
    self.W = self.W - learning_rate*self.dw
    self.b = self.b = learning_rate*self.db
```

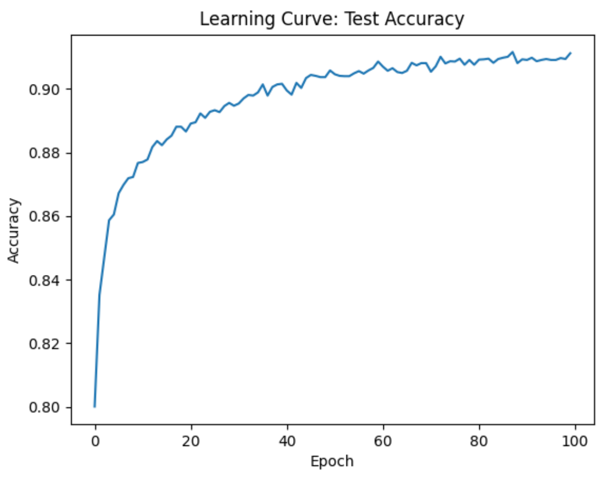
Training and Testing Curve with learning rate=1e-2





Training and Testing Curve with learning rate=1e-1



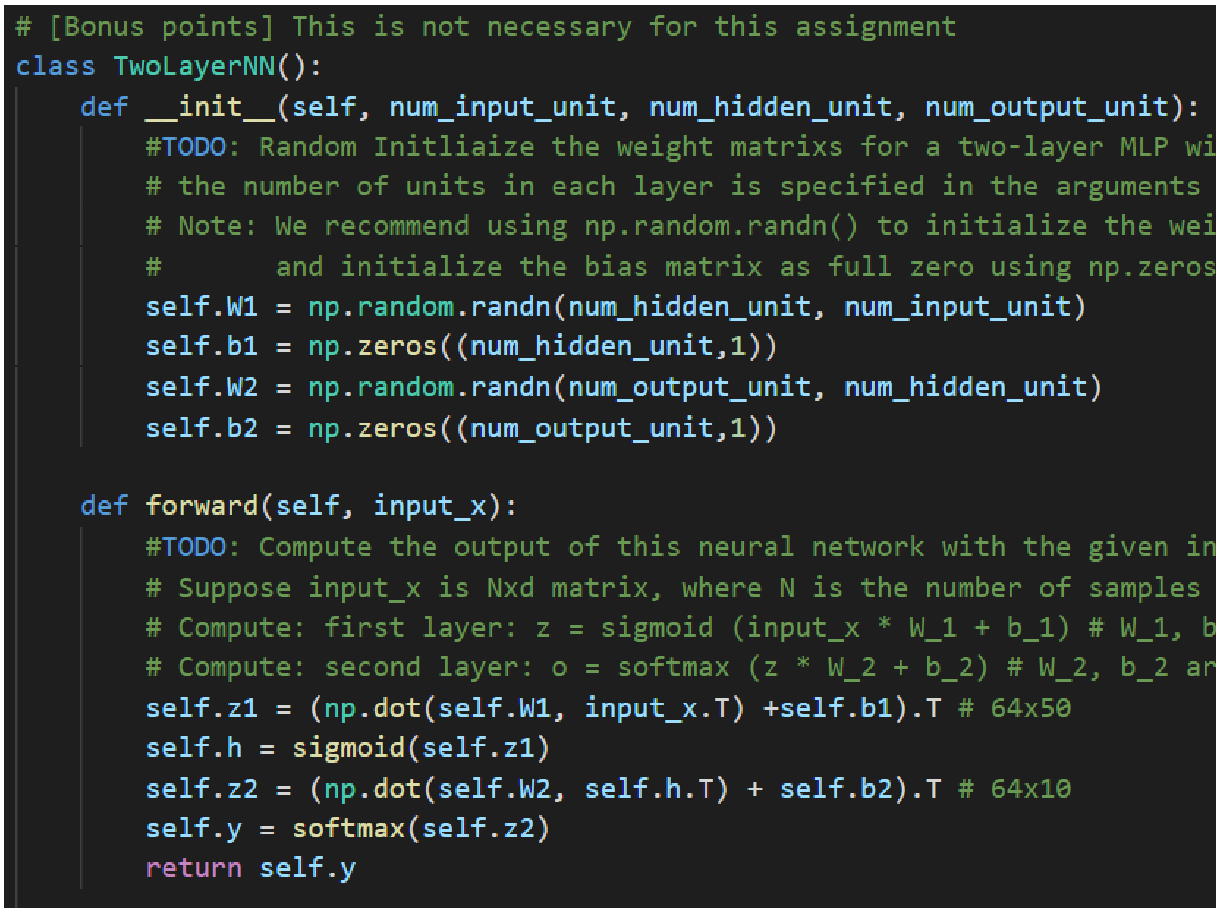


Best Training and Testing Accuracy obtained:

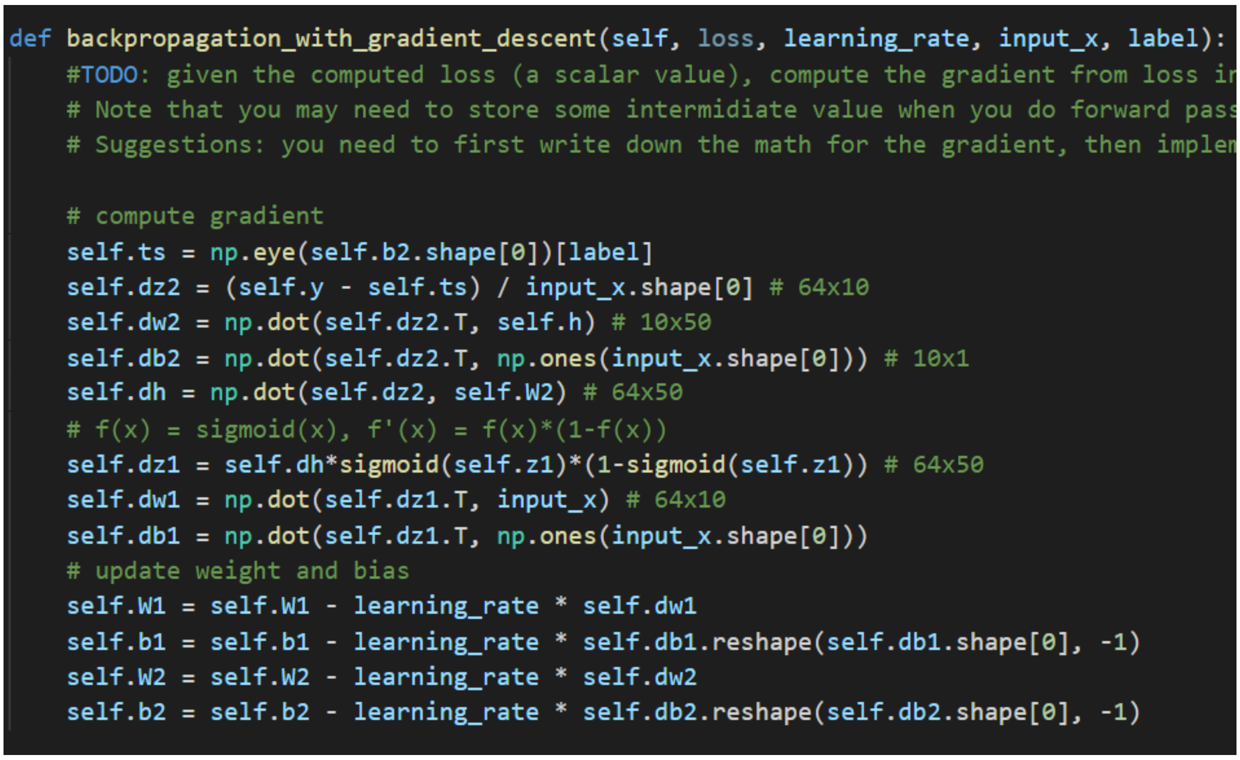
- With learning rate= 1e-2, best train acc is 0.88085 and best test acc is 0.8809.
- With learning rate= 1e-1, best train acc is 0.9239 and best test acc is 0.9116.

Bonus:

Code

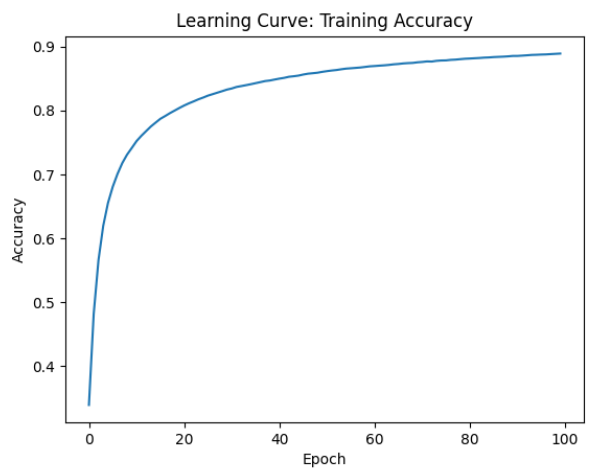


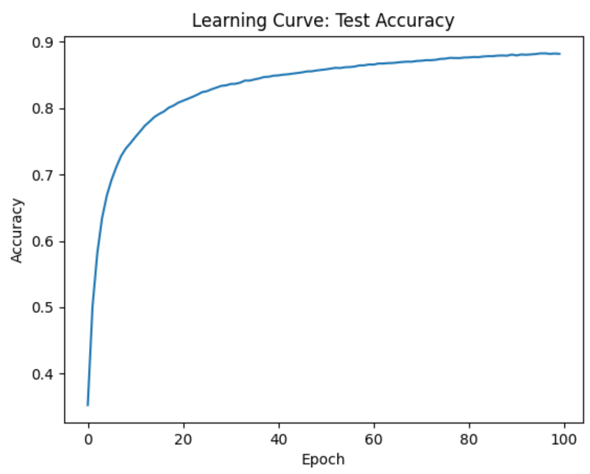
```python
# [Bonus points] This is not necessary for this assignment
class TwoLayerNN():
    def __init__(self, num_input_unit, num_hidden_unit, num_output_unit):
        #TODO: Random Initliaize the weight matrixs for a two-layer MLP wi
        # the number of units in each layer is specified in the arguments
        # Note: We recommend using np.random.randn() to initialize the wei
        #       and initialize the bias matrix as full zero using np.zeros
        self.W1 = np.random.randn(num_hidden_unit, num_input_unit)
        self.b1 = np.zeros((num_hidden_unit,1))
        self.W2 = np.random.randn(num_output_unit, num_hidden_unit)
        self.b2 = np.zeros((num_output_unit,1))

    def forward(self, input_x):
        #TODO: Compute the output of this neural network with the given in
        # Suppose input_x is Nxd matrix, where N is the number of samples
        # Compute: first layer: z = sigmoid (input_x * W_1 + b_1) # W_1, b
        # Compute: second layer: o = softmax (z * W_2 + b_2) # W_2, b_2 ar
        self.z1 = (np.dot(self.W1, input_x.T) +self.b1).T # 64x50
        self.h = sigmoid(self.z1)
        self.z2 = (np.dot(self.W2, self.h.T) + self.b2).T # 64x10
        self.y = softmax(self.z2)
        return self.y
```



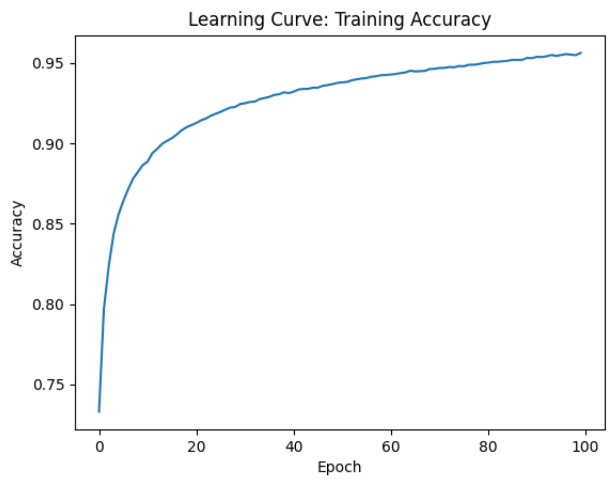
```python
def backpropagation_with_gradient_descent(self, loss, learning_rate, input_x, label):
    #TODO: given the computed loss (a scalar value), compute the gradient from loss ir
    # Note that you may need to store some intermidiate value when you do forward pass
    # Suggestions: you need to first write down the math for the gradient, then implem

    # compute gradient
    self.ts = np.eye(self.b2.shape[0])[label]
    self.dz2 = (self.y - self.ts) / input_x.shape[0] # 64x10
    self.dw2 = np.dot(self.dz2.T, self.h) # 10x50
    self.db2 = np.dot(self.dz2.T, np.ones(input_x.shape[0])) # 10x1
    self.dh = np.dot(self.dz2, self.W2) # 64x50
    # f(x) = sigmoid(x), f'(x) = f(x)*(1-f(x))
    self.dz1 = self.dh*sigmoid(self.z1)*(1-sigmoid(self.z1)) # 64x50
    self.dw1 = np.dot(self.dz1.T, input_x) # 64x10
    self.db1 = np.dot(self.dz1.T, np.ones(input_x.shape[0]))
    # update weight and bias
    self.W1 = self.W1 - learning_rate * self.dw1
    self.b1 = self.b1 - learning_rate * self.db1.reshape(self.db1.shape[0], -1)
    self.W2 = self.W2 - learning_rate * self.dw2
    self.b2 = self.b2 - learning_rate * self.db2.reshape(self.db2.shape[0], -1)
```

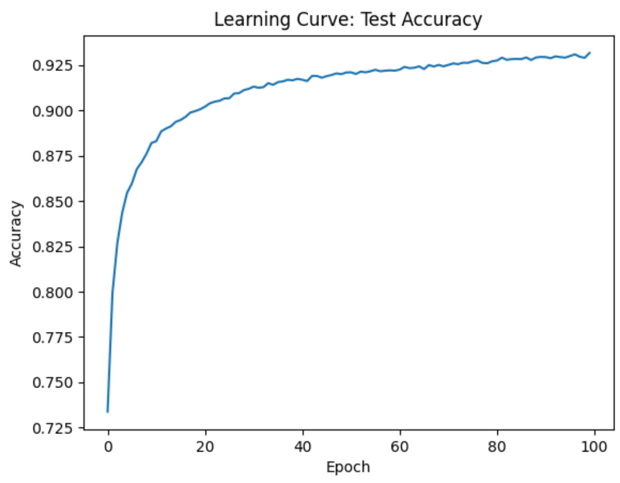
Training and Testing Curve with learning rate=1e-2





Training and Testing Curve with learning rate=1e-1





Best Training and Testing Accuracy obtained:

- With learning rate= 1e-2, best train acc is 0.88925 and best test acc is0.8824.
- With learning rate= 1e-1, best train acc is 0.9563 and best test acc is 0.9318.