CSC420 Assignment 2

1. Implement seam carving

    a) Compute magnitude of gradients of an image.

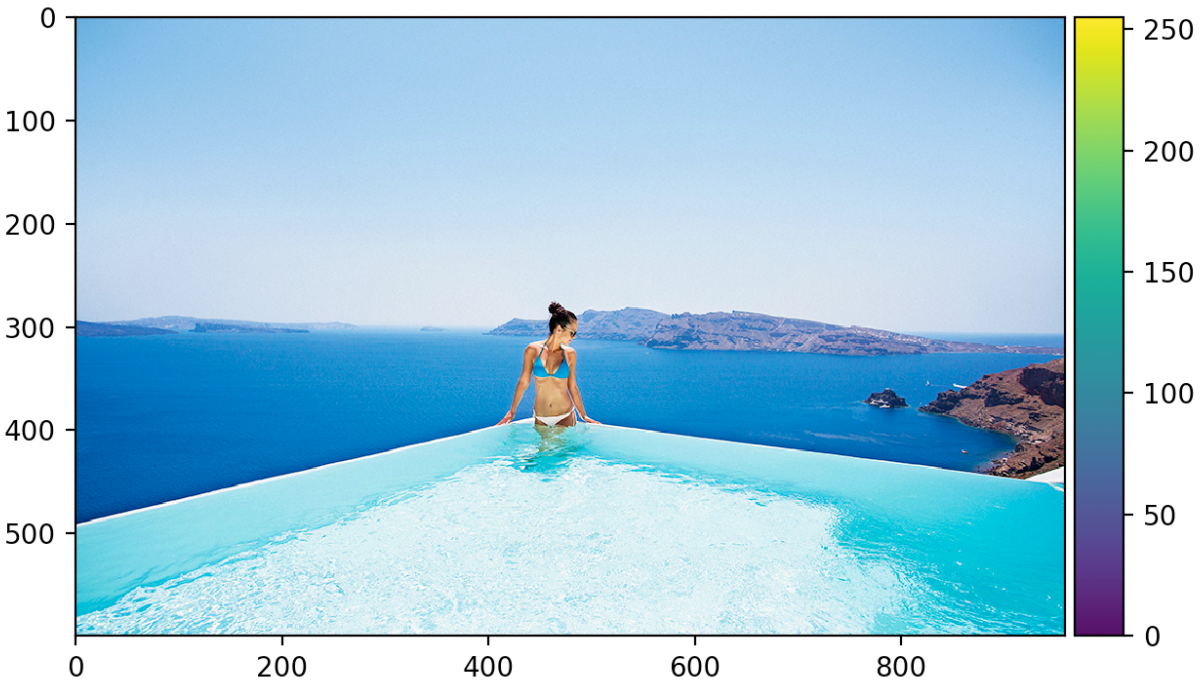
```python
def rgb_gradient(img):
    """
    Compute magnitude of gradients for the input RGB image.
    """
    height, width = img.shape[0], img.shape[1]

    # rgb channels
    red_channel = np.zeros((height, width))
    green_channel = np.zeros((height, width))
    blue_channel = np.zeros((height, width))

    for row in range(height):
        for col in range(width):
            red_channel[row][col] = img[row][col][0]
            green_channel[row][col] = img[row][col][1]
            blue_channel[row][col] = img[row][col][2]

    # compute x- and y- gradients for each channel
    red_x = ndimage.sobel(red_channel, axis=0, mode='constant')
    red_y = ndimage.sobel(red_channel, axis=1, mode='constant')
    green_x = ndimage.sobel(green_channel, axis=0, mode='constant')
    green_y = ndimage.sobel(green_channel, axis=1, mode='constant')
    blue_x = ndimage.sobel(blue_channel, axis=0, mode='constant')
    blue_y = ndimage.sobel(blue_channel, axis=1, mode='constant')

    result = np.zeros((height, width))

    # compute the magnitude of gradients for each pixel
    for row in range(height):
        for col in range(width):
            red = np.hypot(red_x[row, col], red_y[row, col])
            blue = np.hypot(blue_x[row, col], blue_y[row, col])
            green = np.hypot(green_x[row, col], green_y[row, col])
            result[row, col] = np.sqrt(red ** 2 + green ** 2 + blue ** 2)

    return result
```

b) Find the connected path of pixels that has the smallest sum of gradients.

```python
def cost(gradients):
    """
    For each pixel, calculate the smallest sum of gradients of path that
    starts from the pixel to the end row of the image.
    """
    row, col = gradients.shape
    pixel_cost = np.zeros(gradients.shape)

    # the bottom-most row, cost = the gradient of the pixel
    pixel_cost[row - 1, :] = gradients[row - 1, :]

    for i in range(row - 2, -1, -1):
        for j in range(col):
            # only consider connected neighboring pixels
            # connect to one of the column j-1, j, j+1 with minimum cost
            left, right = max(j - 1, 0), min(col, j + 2)
            pixel_cost[i][j] = gradients[i][j] + pixel_cost[i + 1, left: right].min()

    return pixel_cost


def find_seam_path(pixel_cost):
    """
    Find the connected path of pixels that has the smallest sum of gradients.
    """
    row, col = pixel_cost.shape

    path = []
    # append the index of the minimum cost among the first row
    j = pixel_cost[0].argmin()
    path.append(j)

    for i in range(row - 1):
        # only consider connected neighboring pixels
        left, right = max(j - 1, 0), min(col, j + 2)
        # start from the second row
        j = max(j - 1, 0) + pixel_cost[i + 1, left: right].argmin()
        path.append(j)

    return path
```

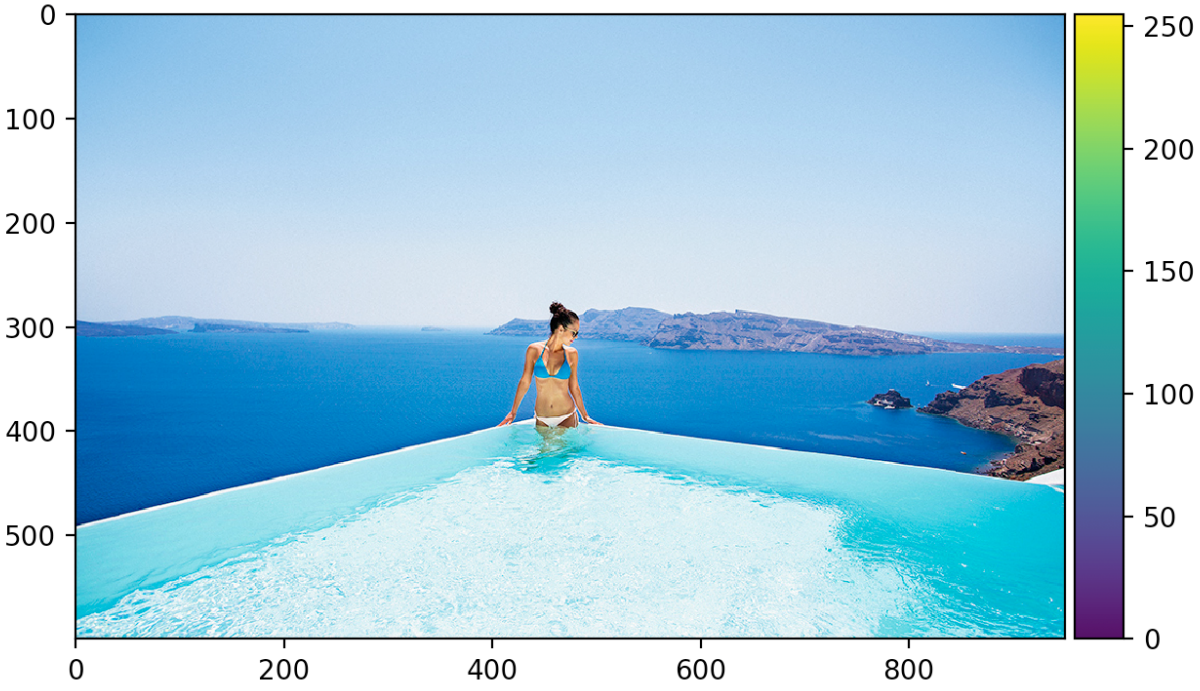
c) Remove the pixels in the path from the image. Get a new image with one column less.

```python
def remove_seam_path(img, path):
    """
    Remove the pixels in the path from the image.
    """
    row, col = img.shape[0], img.shape[1]
    new_img = np.zeros(img.shape)

    for i, j in enumerate(path):
        # copy
        new_img[i, 0:j, :] = img[i, 0:j, :]
        # remove the jth column with minimum cost
        new_img[i, j:col-1, :] = img[i, j+1:col, :]

    res_img = new_img[:, :-1, :].astype(np.int64)
    return res_img


def seam_carving(img):
    """
    Implement seam carving.
    """
    # (a) Compute magnitude of gradients of an image
    gradients = rgb_gradient(img)

    # (b) Find the connected path of pixels with the smallest sum of gradients.
    pixel_cost = cost(gradients)
    seam_path = find_seam_path(pixel_cost)

    # (c) Remove the pixels in the path from the image.
    new_img = remove_seam_path(img, seam_path)

    return new_img
```



d) Remove a few paths with the lowest sum of gradients. Create a few examples.

i. Remove 10 paths with the lowest sum of gradients.

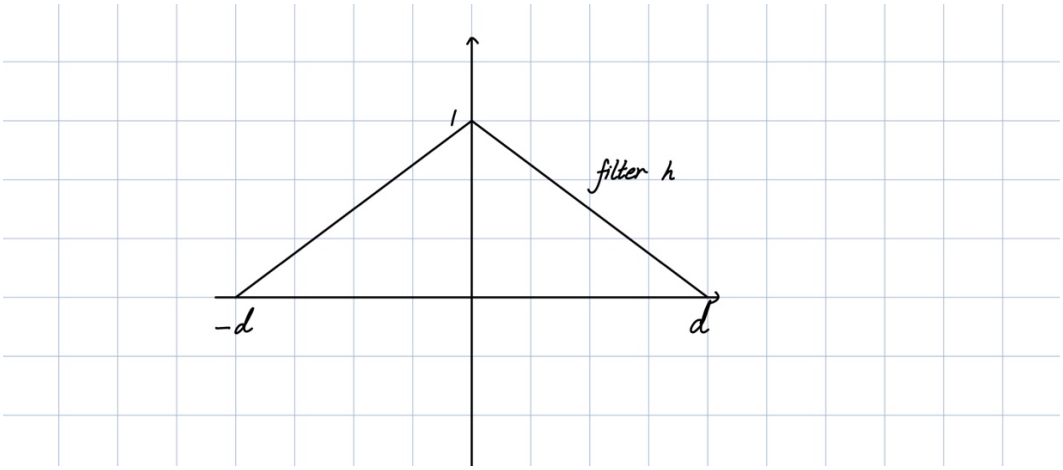


ii. Remove 50 paths with the lowest sum of gradients.

2. Image Upscaling

    1) We could use the linear interpolation method and then the linear convolution filter h would have the mathematical form:
    h = [0, 1/d, …, (d-1)/d, 1, (d-1)/d, …, 1/d, 0], where d is the upscaling factor.

    2) Plot this filter.



3. Neural Network

```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 t
    #           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 vect
    # Note: we take the average among N different samples to get the final loss.
    # number of samples
    n = prediction.shape[0]

    # Compute the cross-entropy loss
    log_probs = np.dot(np.log(prediction), label.T) + np.dot(np.log(1 - prediction), (1 - label).T)
    loss = - np.sum(log_probs) / n

    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 sigmoid_derivative(s):
    return s * (1 - s)


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})
    return np.exp(x) / np.sum(np.exp(x), axis=0)

```

```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 ma
        #       and initialize the bias matrix as full zero using np.zeros()

        self.w1 = np.random.randn(num_output_unit, num_input_unit)
        self.b1 = 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
        # Compute output: z = softmax (input_x * W_1 + b_1), where W_1, b_1 are
        # Note: If we only have one layer in the whole model and we want to use
        #       then we directly apply softmax **without** using sigmoid (or rel

        z1 = np.dot(self.w1, input_x.T) + self.b1
        a1 = softmax(z1)

        return a1

    def backpropagation_with_gradient_descent(self, loss, learning_rate, input_x
        #TODO: given the computed loss (a scalar value), compute the gradient fr
        # Note that you may need to store some intermidiate value when you do fo
        # Suggestions: you need to first write down the math for the gradient, t

        n = input_x.shape[0]
        a1 = self.forward(input_x)

        dl = a1 - label
        # the derivative of softmax here is softmax * (1 - softmax)
        dz1 = np.dot(dl, (a1 * (1 - a1)).T)
        dw1 = (1 / n) * np.dot(dz1, input_x.T)
        db1 = (1 / n) * np.sum(dz1, axis=1, keepdims=True)

        self.w1 = self.w1 - learning_rate * dw1
        self.b1 = self.b1 - learning_rate * db1
```

[Additional credit]

```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 with sig
        # the number of units in each Layer is specified in the arguments
        # Note: We recommend using np.random.randn() to initialize the weight ma
        #       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 input.
        # Suppose input_x is Nxd matrix, where N is the number of samples and d
        # Compute: first Layer: z = sigmoid (input_x * W_1 + b_1) # W_1, b_1 are
        # Compute: second Layer: o = softmax (z * W_2 + b_2) # W_2, b_2 are weig

        z1 = np.dot(self.w1, input_x.T) + self.b1
        a1 = sigmoid(z1)

        z2 = np.dot(self.w2, a1.T) + self.b2
        a2 = softmax(z2)

        return a2


    def backpropagation_with_gradient_descent(self, loss, learning_rate, input_x
        #TODO: given the computed Loss (a scalar value), compute the gradient fr
        # Note that you may need to store some intermidiate value when you do fo
        # Suggestions: you need to first write down the math for the gradient, t

        n = input_x.shape[0]
        z1 = np.dot(input_x, self.w1) + self.b1
        a1 = sigmoid(z1)
        a2 = self.forward(input_x)

        dl = a2 - label
        # softmax
        dz2 = np.dot(dl, (a2 * (1 - a2)).T)
        dw2 = (1 / n) * np.dot(dz2, a1.T)
        db2 = (1 / n) * np.sum(dz2, axis=1, keepdims=True)
        # sigmoid
        dz1 = np.dot(dw2, sigmoid_derivative(z1).T)
        dw1 = (1 / n) * np.dot(dz1, input_x.T)
        db1 = (1 / n) * np.sum(dz1, axis=1, keepdims=True)

        self.w1 = self.w1 - learning_rate * dw1
        self.b1 = self.b1 - learning_rate * db1
        self.w2 = self.w2 - learning_rate * dw2
        self.b2 = self.b2 - learning_rate * db2
```